Volume xx (200y), Number z, pp. 1–26

State-of-the-Art in GPU-Based Large-Scale Volume Visualization

Johanna Beyer[1], Markus Hadwiger[2], Hanspeter Pfister[1]

[1]Harvard University, USA
[2]King Abdullah University of Science and Technology, Saudi Arabia

Abstract

This survey gives an overview of the current state of the art in GPU techniques for interactive large-scale volume visualization. Modern techniques in this field have brought about a sea change in how interactive visualization and analysis of giga-, tera-, and petabytes of volume data can be enabled on GPUs. In addition to combining the parallel processing power of GPUs with out-of-core methods and data streaming, a major enabler for interactivity is making both the computational and the visualization effort proportional to the amount and resolution of data that is actually visible on screen, i.e., "output-sensitive" algorithms and system designs. This leads to recent output-sensitive approaches that are "ray-guided," "visualization-driven," or "display-aware." In this survey, we focus on these characteristics and propose a new categorization of GPU-based large-scale volume visualization techniques based on the notions of actual output-resolution visibility *and the current* working set *of volume bricks—the current subset of data that is minimally required to produce an output image of the desired display resolution. Furthermore, we discuss the differences and similarities of different rendering and data traversal strategies in volume rendering by putting them into a common context—the notion of address translation. For our purposes here, we view parallel (distributed) visualization using clusters as an orthogonal set of techniques that we do not discuss in detail but that can be used in conjunction with what we discuss in this survey.*

Categories and Subject Descriptors (according to ACM CCS): I.3.6 [Computer Graphics]: Methodology and Techniques—I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

1. Introduction

Visualizing volumetric data plays a crucial role in scientific visualization and is an important tool in many domain sciences such as medicine, biology and the life sciences, physics, and engineering. The developments in GPU technology over the last two decades, and the resulting vast parallel processing power, have enabled compute-intensive operations such as ray-casting of large volumes at interactive rates. However, in order to deal with the ever-increasing resolution and size of today's volume data, it is crucial to use highly scalable visualization algorithms, data structures, and architectures in order to circumvent the restrictions imposed by the limited amount of on-board GPU memory.

Recent advances in high-resolution image and volume acquisition, as well as computational advances in simulation, have led to an explosion of the amount of data that must be visualized and analyzed. For example, high-throughput electron microscopy can produce volumes of scanned brain tissue at a rate above 10-40 megapixels per second [BLK*11], with a pixel resolution of 3-5 nm. Such an acquisition process produces almost a terabyte of raw data per day. For the next couple of years it is predicted that new multibeam electron microscopes will further increase the data acquisition rate by two orders of magnitude [Hel13, ML13]. A current prototype with 61 beams can capture 1.2 billion pixels per second already [Mar13]. This trend of acquiring and computing more and more data at a rapidly increasing pace ("Big Data") will continue in the future [BCH12]. This naturally poses significant challenges to interactive visualization and analysis. For example, many established algorithms and frameworks for volume visualization do not scale well beyond a few gigabytes, and this problem cannot easily be solved by simply adding more computing power or disk space. These challenges require research on novel techniques for data visualization, processing, storage, and I/O that scale to extreme-scale data [MWY*09, AAM*11, BCH12].

Today's GPUs are very powerful parallel processors that enable performing compute-intensive operations such as ray-casting at interactive rates. However, the memory sizes

available to GPUs are not increasing at the same rate as the amount of raw data. In recent years, several GPU-based methods have been developed that employ out-of-core methods and data streaming to enable the interactive visualization of giga-, tera-, and petabytes of volume data. The crucial property that enables these methods to scale to extreme-scale data is their *output-sensitivity*, i.e., that they make both the computational and the visualization effort proportional to the amount of data that is actually visible on screen (i.e., the output), instead of being proportional to the full amount of input data. In graphics, the focus of most early work on output-sensitive algorithms was visibility determination of geometry (e.g., [SO92, GKM93, ZMHH97]).

An early work in output-sensitive visualization on GPUs was dealing with 3D line integral convolution (LIC) volumes of flow fields [FW08]. In the context of large-scale volume visualization, output-sensitive approaches are often referred to as being *ray-guided* (e.g., [CNLE09, Eng11, FSK13]) or *visualization-driven* (e.g., [HBJP12, BHAA*13]). These are the two terms that we will use most in this survey.

We use the term *visualization-driven* in a more general and inclusive way, i.e., these methods are not necessarily bound to ray-casting (which is implied by "ray-guided"), and they can encompass all computation and processing of data in addition to rendering. In principle, the visual output can "drive" the entire visualization pipeline—including on-demand processing of data—all the way back to the raw data acquisition stage [HBJP12, BHAA*13]. This would then yield a fully *visualization-driven pipeline*. However, to a large extent these terms can be used interchangeably.

Another set of output-sensitive techniques are *display-aware* multi-resolution approaches (e.g., [JST*10, JJY*11, HSB*12]). The main focus of these techniques is usually output-sensitive computation (such as image processing) rather than visualization, although they are also guided by the actual display resolution and therefore the visual output.

Ray-guided and visualization-driven visualization techniques are clearly inspired by earlier approaches for occlusion culling (e.g., [ZMHH97, LMK03]) and level of detail (e.g., [LHJ99, WWH*00]). However, they have a much stronger emphasis on leveraging *actual output-resolution visibility* for data management, caching, and streaming—in addition to the traditional goals of faster rendering and anti-aliasing. Very importantly, actual visibility is determined on-the-fly during visualization, directly on the GPU.

1.1. Survey Scope

This survey focuses on major scalability properties of volume visualization techniques, reviews earlier GPU volume renderers, and then discusses modern ray-guided and visualization-driven approaches and how they relate to and extend the standard visualization pipeline (see Figure 1). Large-scale GPU volume rendering can be seen as being in the intersection of volume visualization and high performance computing. General introductions to these two topics are given in books on real-time volume graphics [EHK*06] and high performance visualization [BCH12], respectively.

We mostly focus on techniques for stand-alone workstations with standard graphics hardware. We see the other core topics of high performance visualization (i.e., parallel rendering on CPU/GPU clusters, distributed visualization frameworks, and remote rendering) as an orthogonal set of techniques that can be used in combination with modern ray-guided, visualization-driven, and display-aware techniques as discussed here. Therefore, for more details on parallel visualization we refer the reader to previous surveys in this area [Wit98, BSS00, ZSJ*05]. Nonetheless, where parallel or distributed rendering methods do directly relate to our course of discussion we have added them to our exposition.

We focus on volume rendering of regular grids and mostly review methods for scalar data and a single time step. However, the principles of the discussed scalable methods are general enough that they also apply to multi-variate, multi-modal, or time series data. For a more in-depth discussion of the visualization and visual analysis of multi-faceted scientific data we refer the reader to a recent comprehensive survey [KH13]. Other related recent surveys can be found on the topics of compression for GPU-based volume rendering [RGG*14], and massive model visualization [KMS*06].

1.2. Survey Structure

This survey gives an overview of the current state of the art in large-scale GPU volume visualization. Starting from the standard visualization pipeline in Section 2, we discuss required modifications and extensions to this pipeline to achieve scalability with respect to data size and define some underlying concepts of this report.

We continue by examining scalable data structures that are the basis for rendering large data (Section 3). Next, we discuss general scalability strategies and how they relate to and are used in volume visualization (Section 4). In particular, we focus on *how, when*, and *where* data is processed and rendered to achieve scalable performance, including ways to reduce the computational load.

Section 5 discusses recent advances in large-scale volume rendering in depth. We propose a new categorization of GPU-based large-scale volume visualization techniques (Table 2) based on the notion of the active working set—the current subset of data that is minimally required to produce an output image of the desired display resolution. Based on this categorization, we review traditional GPU volume rendering techniques, their advantages and limitations (Section 5.2).

In Section 6 we present a unified categorization of address translation methods, putting popular address translation methods like tree traversal and virtual texturing into

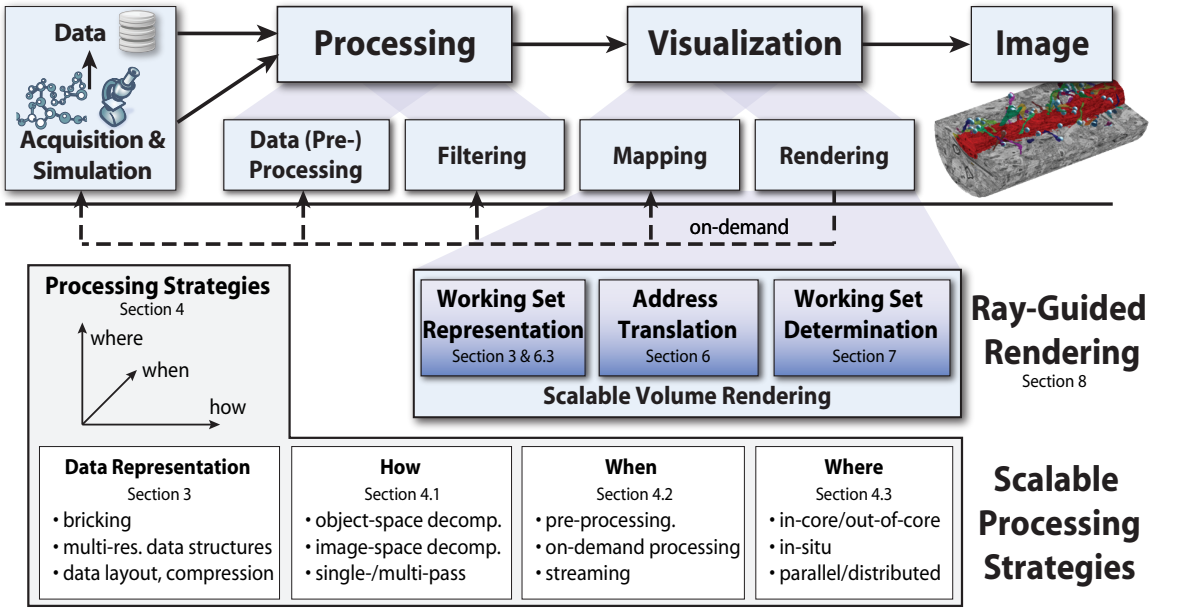

Figure 1: ***The visualization pipeline for large-scale visualization.*** *Top: Data are generated on the left (either through acquisition/measurement or through computation/simulation) and then pass through a sequence of stages that culminate in the desired output image. Bottom: Extensions to the visualization pipeline to achieve scalability: A* ray-guided *or* visualization-driven *approach can drive earlier pipeline stages so that only what is required by (visible in) the output image is actually loaded or computed. In a fully visualization-driven pipeline, this approach can be carried through from rendering (determining visibility) on the right all the way back to data acquisition/simulation on the left (stippled line). The bottom row shows different* scalable processing strategies *(related to* where, when and how *data is being processed and rendered), and where they are discussed in this report.*

the same reference framework (Section 6.1). Section 6.2 presents their individual advantages and shortcomings in more detail. We also address how GPU memory managment is handled by different address translation approaches (Section 6.3).

We discuss methods for determining the working set (i.e., culling) in Section 7, moving from basic view frusum and global attribute-based culling to occlusion culling and finally ray-guided culling.

In Section 8 we reflect on and emphasize the recent advances of ray-guided algorithms, and sum up the most important works in that area.

Finally, we review the major challenges and current limitations and give an outlook on future trends and open problems in large-scale GPU volume visualization (Section 9).

2. Fundamentals

Before delving into the depths of scalable GPU volume rendering algorithms and data structures we first give a conceptual overview of the visualization pipeline with focus on large-scale volume visualization and define and clarify some basic concepts that are used throughout this survey.

2.1. Large-Scale Visualization Pipeline

A common abstraction used by visualization frameworks is the *visualization pipeline* [Mor13]. In essence, the visualization pipeline is a data flow network where nodes or modules are connected in a directed graph that depicts the data flow throughout the system (see Figure 1). After data acquisition or generation through computation or simulation, the first stage usually consists of some kind of data processing, which can include many sub-tasks from data *pre-processing* (e.g., computing a multi-resolution representation) to *filtering*. The second half of the pipeline comprises the actual visualization, including *visualization mapping* and *rendering*. For large-scale rendering, all the stages in this pipeline have to be scalable (i.e., in our context: output-sensitive), or they will become the bottleneck for the entire application.

Ray-guided or visualization-driven approaches can drive earlier stages in the visualization pipeline so that only the required data (i.e., visible data in the output image) is loaded, processed and rendered (stippled line in Figure 1). Actual scalability in volume rendering also depends on how dynamically and accurately the working set is determined, how volumes are represented, and how ray traversal and address translation is performed.

The bottom part of Figure 1 shows the main processing strategies employed by state-of-the-art visualization-driven pipelines to achieve this scalability and lists where they are discussed in this report. Efficient data structures such as multi-resolution and compact data representations are necessary as well as deciding on how, when and where data is processed and rendered.

2.2. Basic Concepts

Large-scale visualization. In the context of this survey, large-scale visualization deals with volume data that do not completely fit into memory. In our case, the most important memory type is GPU on-board memory, but scalability must be achieved throughout the entire memory hierarchy. Most importantly, large-scale volume data cannot be handled directly by volume visualization techniques that assume that the entire volume is resident in memory in one piece.

Bethel et al. [BCH12] (Chapter 2) define large data based on three criteria: They are too big to be processed: (1) in their entirety, (2) all at one time, and (3) exceed the available memory. Scalable visualization methods and architectures tackle either one or a combination of these criteria.

Scalability. In contrast to parallel/distributed visualization, where a major focus is on *strong* vs. *weak* scaling [CPA*10], we define scalability in terms of *output-sensitivity* [SO92]. Our focus are algorithms, approaches, and architectures that scale to large data by making the computation and visualization effort proportional to both the *visible* data on screen and the actual *screen resolution*. If the required size of the *working set* of data is independent of the original data size, we say that an approach is *scalable* in this sense.

Scalability issues. Based on the notion of large data, the main scalability issues for volume rendering deal with questions on how to represent data, and how, when and where the data is processed and rendered. This includes strategies to split up the work and/or data to make it more tractable, and to reduce the amount of work and/or data that has to be handled. The bottom of Figure 1 lists these main processing strategies to achieve scalability and mentions where they are described in our report.

Acceleration techniques vs. data size. A common source of confusion when discussing techniques for scalable volume rendering is the real goal of a specific optimization technique. While many of the techniques discussed in this survey were originally proposed as performance optimizations, they can also be adapted to handle large data sizes. A well-known example of this are octrees. While octrees are often used in geometry rendering to speed up view frustum culling (via hierarchical/recursive culling), an important goal of using octrees in volume rendering is to enable adaptive level of detail [WWH*00] (thereby limiting the amount of data that has to be handled), in addition to enabling empty space skipping. This "dual" purpose of many scalable data structures and algorithms is an important issue to keep in mind.

Output-sensitive algorithms. The original focus of output-sensitive algorithms [SO92] was making their *running time* dependent on the size of the *output* instead of the size of the *input*. While this scalability in terms of running time is of course also important in our context, for the work that we discuss here, it is even more important to consider the dependence on "output data size" vs. input data size, using the concept of the *working set* as described above.

Ray-guided and visualization-driven architectures. In line with the concepts outlined above, scalable volume rendering architectures rely heavily on data management (e.g., processing, streaming, caching) and how to reduce the amount of data that has to be processed, rather than just rendering. While ray-casting intrinsically could be called "ray-guided", this by itself is not very meaningful. The difference to standard ray-casting first arises from how and which data are streamed into GPU memory, i.e., *ray-guided streaming* of volume data [CNLE09]. Again considering the working set, a ray-guided approach determines the working set of volume bricks *via* ray-casting. That is, the working set comprises the bricks that are intersected during ray traversal. It is common to determine the desired level of detail, i.e., the locally required volume resolution, during ray-casting as well.

In this way, data streaming is guided by the *actual* visibility of data in the output image. This is in contrast to the approximate/conservative visibility obtained by all common occlusion culling approaches. As described in the introduction, *visualization-driven* architectures generalize these concepts further to ultimately drive the entire visualization pipeline by actual on-screen visibility [HBJP12,BHAA*13].

3. Data Representation and Storage

Efficient data representation is a key requirement for scalable volume rendering. Scalable data structures should be compact in memory (and disk storage), while still being efficient to use and modify. Figure 2 shows common related data structures, and Table 1 lists their scalability aspects. Additional GPU representations and management of these data structures, as they are used for rendering, is discussed in Section 6.3.

3.1. Bricking

Bricking is an object space decomposition method that subdivides the volume into smaller, box-shaped sub-volumes, the so-called *bricks*. All bricks usually have the same size in voxels (e.g., 32^3 or 256^3 voxels per brick). Volumes that are not a multiple of the basic brick size are padded accordingly. Bricking facilitates out-of-core approaches because individual bricks can be loaded and rendered as required, without ever having to load/stream the volume in its entirety.

Data Structure	Acceleration	Out-of-Core	Multi-Resolution
mipmaps	no [except level of detail]	clipmaps [TMJ98]	yes
octrees / kd-trees	hierarchical traversal/culling	working set (subtree)	yes
uniform grids (bricking)	(linear) culling of bricks	working set (bricks from grid)	no
hierarchical grids (bricking)	(hierarchical) culling of bricks	working set (bricks from hierarchy)	yes

Table 1: ***Scalable data structures for volume visualization.*** *Data structures have a huge impact in how acceleration during ray-casting (skipping, culling), out-of-core processing/rendering, and multi-resolution rendering (i.e., adaptive level of detail) is supported.*

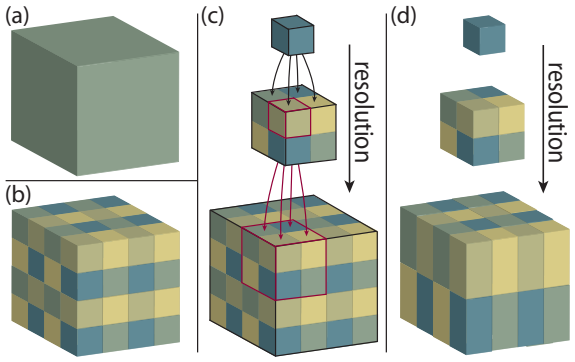

Figure 2: ***Common spatial data structures in volume rendering.*** *(a) Original volume; (b) bricked volume; (c) octree with fixed subdivision (only four of eight pointers to children shown); (d) multi-resolution bricking with arbitrary downsampling ratios, which is beneficial for anisotropic volumes.*

Bricked data often require special handling of brick boundaries. Operations where neighboring voxels are required (e.g., GPU texture filtering, gradient computation) usually return incorrect results at brick boundaries, because the neighboring voxels are not readily available. A typical example of this is tri-linear interpolation of samples during ray-casting. Although boundary voxels can be fetched individually from the neighboring bricks [Lju06a], this is in general very costly. More commonly, so-called *ghost voxels* [ILC10] are employed, which are duplicated voxels at the brick boundaries that enable straightforward, fully correct hardware texture filtering. The use of ghost voxels is the standard approach in most bricked ray-casters [BHWB07, FK10]. Ghost voxels are generally stored together with each brick on disk—which increases disk storage—, but they can also be computed on-the-fly in a streaming fashion [ILC10].

The recent OpenGL extension for virtual texturing (`GL_ARB_sparse_texture`) includes hardware support for texture filtering across brick boundaries, and thus alleviates the need for ghost voxels to some extent.

Choosing the optimal brick size depends on several criteria and has been studied in the literature [HBJP12, FSK13]. Small bricks support fine-grained culling, which facilitates smaller working sets because less unnecessary data needs to be fetched. On the other hand, however, the ghost voxel overhead grows for smaller bricks, and the total number of bricks increases as well. The latter makes multi-pass rendering approaches that render bricks individually infeasible.

For these reasons, traditional multi-pass out-of-core volume renderers typically use relatively large bricks (e.g., 128^3 or 256^3 voxels) to reduce the number of required rendering passes. In contrast, modern single-pass ray-casters use smaller bricks (e.g., 32^3 voxels), or a hybrid approach where small bricks are used for rendering, and larger bricks are used for storage on disk [HBJP12, FSK13]. For 2D data acquisition modalities such as microscopy, hybrid 2D/3D tiling/bricking strategies have also been employed successfully. One example is the on-demand computation of 3D bricks from pre-computed 2D mipmap tiles of electron microscopy images during visualization [HBJP12, BHAA*13].

3.2. Multi-Resolution Hierarchies

One of the main benefits of multi-resolution hierarchies for rendering large data is that they allow sampling the data from a resolution level that is adapted to the current screen resolution or desired level of detail, instead of only from the original (full) resolution. This can significantly reduce the amount of data that needs to be accessed, and is also important for avoiding aliasing artifacts due to undersampling.

Trees (octrees, kd-trees). Octrees [WWH*00, Kno06] and kd-trees [FCS*10] are very common 3D multi-resolution data structures for direct volume rendering. They allow efficient traversal and directly support hierarchical empty space skipping. Traditional tree-based volume renderers employ a multi-pass rendering approach where one brick (one tree node) is rendered per rendering pass. Despite the hierarchical nature of these data structures, many early approaches assume that the entire volume fits into memory [LHJ99, WWH*00, BNS01]. Modern GPU approaches support traversing octrees directly on the GPU [GMG08, CNLE09, CN09, RTW13], which is usually accomplished via standard traversal algorithms adapted from the ray-tracing literature [AW87, FS05, HSHH07, PGS*07, HL09].

In recent years, *sparse voxel octrees* (SVOs) have gained a lot of attention in the graphics and gaming industry [LK10a, LK10b]. Several methods for rendering large and complex voxelized 3D models use SVO data structures for efficient rendering [GM05, RÖ9, HN12, Mus13].

Mipmaps are a standard multi-resolution pyramid representation that is very common in texture mapping [Wil83]. Mipmaps are supported by virtually all GPU texture units. Clipmaps [TMJ98] are virtualized mipmaps of arbitrary size. They assume a moving window (like in terrain rendering) that looks at a small sub-rectangle of the data and use a toroidal updating scheme for texels in the current view. However, for general volume rendering this inherent inflexibility of clipmaps would be very restrictive.

Hierarchical grids (mipmaps) with bricking. Another type of multi-resolution pyramids are hierarchical grids (or mipmaps) where each resolution level of the data is bricked individually. These grids have become a powerful alternative to octrees in recent ray-guided volume visualization approaches [HBJP12, FSK13]. The basic approach can be viewed as bricking each level of a mipmap individually. Moreover, more flexible systems do not use hardware mipmaps and therefore even allow varying downsampling ratios between resolution levels [HBJP12]—e.g., for anisotropic data—which is not possible with standard mipmaps.

Since there is no tree structure in such a grid type, no tree traversal is necessary during rendering. Rather, the entire grid hierarchy can be viewed as a huge virtual address space (a *virtual texture*), where any voxel corresponding to data of any resolution can be accessed directly via *address translation* from *virtual* to *physical* addresses [vW09, BHL*11, OVS12, HBJP12]. On GPUs, this address translation can be performed via GPU textures that act as "page tables," which can be further extended to a *multi-level* page table hierarchy for extremely large data [HBJP12] (see Section 6).

As in all approaches that employ some form of bricking, interpolation between bricks has to be handled carefully. Especially the transitions between different resolution levels can introduce visual artifacts, and several methods have been introduced that deal with correct interpolation between different resolution levels [Lju06a, Lju06b, BHMF08].

Wavelet representations. The earliest works using wavelet transforms for volume rendering were Muraki [Mur93] and Westermann [Wes94]. Subsequent methods such as Guthe et al. [GGSe*02, GS04] compute a hierarchical wavelet representation in a pre-process, and decompress the bricks that are required for rendering on-the-fly.

Other representations. Younesy et al. [YMC06] have proposed improving the visual quality of multi-resolution volume rendering by approximating the voxel data distribution by its mean and variance at each level of detail. The recently introduced *sparse pdf volumes* [SKMH14] and *sparse pdf maps* [HSB*12], respectively, represent the data distributions more accurately. For sparse pdf volumes, this allows for consistent multi-resolution volume rendering [SKMH14], i.e., the consistent application of transfer functions independent of the actual resolution level used. For sparse pdf maps (images), this allows for the accurate, anti-aliased evaluation of non-linear image operators on gigapixel images [HSB*12]. The corresponding data structures are very similar to standard mipmaps (2D or 3D) in terms of storage and access [SKMH14, HSB*12].

3.3. Data Layout and Compression

Data layout. To efficiently access data on disk, data layout and access are often optimized. In general, reading small bits of data at randomly scattered positions is a lot more inefficient than reading larger chunks in a continuous layout. Therefore, locality-preserving data access patterns such as space filling curves, e.g., Morton (z-) order [Mor66] are often used in time-critical visualization frameworks [SSJ*11]. A nice feature of the Morton/z-order curve is that by adjusting the sampling stride along the curve, samples can be restricted to certain resolution levels. Pascucci and Frank [PF02] describe a system for progressive data access that streams in missing data points for higher resolutions. With the most recent solid state drives (SSDs), however, trade-offs might be different in practice [FSK13].

Data compression. Another major related field is data compression, for disk storage as well as for the later stages of the visualization pipeline. We refer to the recent comprehensive survey by Rodriguez et al. [RGG*14] for an in-depth discussion of the literature on volume compression and *compression-domain* volume rendering.

4. Scalable Processing Strategies

This section introduces the main considerations and techniques for designing scalable volume visualization architectures in general terms. We will focus on the questions of *when* (Section 4.2) and *where* (Section 4.3) data processing and rendering takes place. But first, we introduce different strategies of *how* data handling and processing can be made scalable by different *decomposition* strategies (Section 4.1).

Reducing the amount of data that has to be processed or rendered is a major strategy for dealing with large data. Techniques for data reduction cover a broad scope, ranging from multi-resolution data representations and sub-sampling to more advanced filtering and abstraction techniques. A distinction has to be made between data reduction for storage (e.g., compression) that tries to reduce disk or in-memory size, and data reduction for rendering. The latter encompasses visualization-driven and display-aware rendering approaches as well as more general methods such as on-demand processing and query-based visualization.

In real-world applications, these strategies for handling and rendering large data often have to be combined to achieve interactive performance and high-quality images. Furthermore, for future ultra-scale visualization and exascale computing [ALN*08, SBH*08, MWY*09, AAM*11, Mor12] it is essential that each step of the visualization pipeline is fully scalable.

4.1. Decompositions Strategies

A crucial technique for handling large data is to *partition* or *decompose* data into smaller parts (e.g., sub-volumes). This is essentially a *divide and conquer* strategy, i.e., breaking down the problem into several sub-problems until they become easier to solve. Partitioning the data and/or work can alleviate memory constraints, complexity, and allow parallelization of the computational task. In the context of visualization, this includes ideas like domain decomposition (i.e., object-space and image-space decompositions), but also entails single-pass vs. multi-pass rendering approaches.

Object-space (data domain) decomposition. This type of decomposition is usually done by using bricking with or without a multi-resolution representation, as described in Sections 3.1 and 3.2, respectively. Object-space decompositions are view-independent and facilitate scalability with respect to data size by storing and handling data subsets separately.

Image-space (image domain) decomposition. Image domain subdivision splits the output image plane (the viewport) and renders the resulting image tiles independently. A basic example of this approach is ray-casting (which is an *image-order* approach), where conceptually each pixel is processed independently. In practice, several rays (e.g., a rectangular image tile) are processed together in some sense. For example, rendering each image tile in a single rendering pass, or assigning each tile to a different rendering node. Another example is rendering on a large display wall, where each individual screen is assigned to a different rendering node.

Single-pass vs. multi-pass rendering. In single-pass approaches the volume is traversed in front-to-back order in a single rendering pass as compared to multi-pass approaches that require multiple rendering passes. The first GPU volume rendering approaches [CN93, CCF94, WE98, RSEB*00, HBH03], including the first octree-based renderers [LHJ99, WWH*00, GGSe*02, GS04, HFK05], were all based on multi-pass rendering. With the introduction of dynamic branching and looping on GPUs, single-pass approaches have been introduced to volume ray-casting [HSSB05, SSKE05].

Multi-pass approaches offer a higher flexibility, however, they also have a significant management overhead compared to single-pass rendering (i.e., context switching, final compositing) and usually result in lower performance. Furthermore, optimization techniques like early ray termination are not trivial in multi-pass rendering and create an additional overhead. Therefore, most state-of-the art ray-guided volume renderers use single-pass rendering [CNLE09, Eng11, HBJP12]. A limitation of single-pass approaches, however, is the requirement for the entire working set to fit into the cache. One way to circumvent this requirement is to use single-pass rendering as long as the working sets fits into the cache, and to switch to multi-pass rendering when the working set gets too large [Eng11, FSK13].

4.2. Time-Based and Scheduling Strategies

A careful selection of the point in time when data is being processed or rendered can have a tremendous influence of the amount of data that needs to be handled. In this section we focus on time-based processing strategies such as pre-processing, on-demand processing and streaming.

Pre-Processing. Running computationally expensive or time-consuming computations as a pre-process to compute acceleration metadata or pre-cache data can often dramatically reduce the computation costs during rendering. Typical examples include pre-computing a multi-resolution hierarchy of the data that is used to reduce the amount of data needed for rendering. On the other hand, processing data interactively during rendering can reduce the required disk space [BCH12] (Chapter 9), and enables on-demand processing, which in turn can reduce the amount of data that needs processing.

On-Demand Processing. On-demand strategies determine at run time which parts of the data need to be processed, thereby eliminating pre-processing times and limiting the amount of data that needs to be handled. For example, ray-guided and visualization-driven volume rendering systems only request volume bricks to be loaded that are necessary for rendering the current view [CNLE09, HBJP12, FSK13]. Data that is not visible is never rendered, processed, or even loaded from disk.

Other examples for on-the-fly processing for volume visualization target interactive filtering and segmentation. For example, Jeong et al. [JBH*09] have presented a system where they perform on-the-fly noise removal and edge enhancement during volume rendering only for the currently visible part of the volume. Additionally, they perform an interactive active-ribbon segmentation on a dynamically selected subset of the data. More recently, Solteszova et al. [SBVB14] have presented a visibility-driven method for on-the-fly filtering (i.e., noise removal and feature detection) of 4D ultrasound data.

Query-driven Visualization. These approaches can be considered a special kind of on-demand processing, where *selection* is used as the main means to reduce the amount of data that needs to be processed [BCH12] (Chapter 7). Prominent techniques are dynamic queries [AWS92], high-dimensional brushing and linking [MW95], and interactive visual queries [DKR97]. Shneiderman [Shn94] gives an introduction to dynamic queries for visual analysis and information seeking.

The DEX framework [SSWB05] focuses on query-driven scientific visualization of large data sets using bitmap indexing to quickly query data. Recently, approaches for query-

based volume visualization have been introduced in the context of neuroscience [BvG*09, BAaK*13], with the goal to analyze the connectivity between individual neurons in electron microscopy volumes. The ConnectomeExplorer framework [BAaK*13] implements visual queries on top of a large-scale, visualization-driven system.

Streaming. In streaming approaches, data are processed as they become available (i.e., are streamed in). Streaming techniques are closely related to on-demand processing. However, where the latter usually consists of a *pull model* (i.e., data is requested by a process), streaming can be a pull or a *push model* (i.e., new data is pushed to the next processing step).

Streaming also circumvents the need for the entire data set to be available before the visualization starts and allows rendering of incomplete data [SCC*02]. Hadwiger et al. [HBJP12] have described a system for streaming extreme-scale electron microscopy data for interactive visualization. This system has later been extended to include on-the-fly registration and multi-volume visualization of segmented data [BHAA*13]. Further streaming-based visualization frameworks include the dataflow visualization system presented by Vo et al. [VOS*10], which is built on top of VTK and implements a push and pull model.

4.3. Location-Based Strategies

In this section, we focus on processing strategies that are categorized by *where* computations are performed in the visualization pipeline, or more precisely, where the data is located and stored when it is being processed. These techniques generally lower the in-core memory requirements by using out-of-core, in-situ or distributed processing strategies.

Out-Of-Core Techniques. Unless when dealing with data that is small enough to fit into memory ("in core") in its entirety, one always has to partition the data and/or computation in a way that makes it possible to process subsets of the data independently. This enables out-of-core processing and can be applied at all stages of the visualization pipeline [SCC*02, KMS*06]. Different levels of out-of-core processing exist, depending on where the computation is performed and where the data is residing (either on the GPU, CPU, hard-disk, or network storage).

Out-of-core methods include algorithms that focus on accessing [PF02] and prefetching [CKS03] data, creating on-the-fly ghost data for bricked representations [ILC10], and methods for computing multi-resolution hierarchies [HBJP12] or other processing tasks such as segmentation [FK05], PDE solvers [SSJ*11], image registration and alignment [JST*10], or level set computation [LKHW04]. Silva et al. [SCC*02] give a comprehensive overview of out-of-core methods for visualization and graphics.

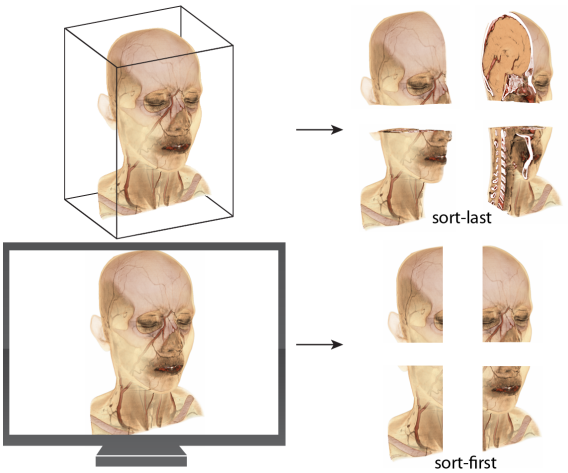

Figure 3: ***Parallel volume rendering*** *using sort-first and sort-last strategies as proposed in [MCE*94]. In sort-last rendering (top) each process is responsible for rendering one or several parts of the volume and requires a final compositing pass to generate the final image. Compositing is often the bottleneck in parallel volume rendering, even though parallel compositing algorithms exist [MPHK94, YWM08, PGR*09]. In sort-first approaches (bottom), each process is responsible for rendering a part of the viewport. This does not require compositing, but often exhibits load-balancing issues when moving the viewport, as data needs to be moved between different processes.*

In-Situ Visualization. Traditionally, visualization is performed after all data have been generated—either by measurement or simulation—and have been written to disk. In-situ visualization, on the other hand, runs simultaneously to the on-going simulation (e.g., on the same supercomputer or cluster: *in situ*—in place), with the aim of reducing the amount of data that needs to be transferred and stored on disk [BCH12] (Chapter 9).

To avoid slowing down the primary simulation, *in-transit* visualization accesses only "staging" nodes of a simulation cluster. The goal of these nodes is to hide the latency of disk storage from the main simulation by handling data buffering and I/O [MOM*11].

In-situ and in-transit visualization have been identified as being crucial for future extreme-scale computing [MWY*09, AAM*11, KAL*11, Mor12]. Furthermore, when the visualization process is tightly coupled or integrated into the simulation, these approaches can be leveraged for *computational steering*, where simulation parameters are changed based on the visualization [PJ95, TTRU*06]. Yu et al. [YWG*10] present a complete case study of in-situ visualization for a petascale combustion simulation. Tikhonova et al. [TYC*11] take a different approach by generating a compact intermediate representation of large

volume data that enables fast approximate rendering for preview and in-situ setups.

Parallel and Distributed Rendering. High-performance visualization often depends on distributed approaches that split the rendering of a data set between several nodes of a cluster. The difference can be defined such that *parallel* visualization approaches run on a single large parallel platform, whereas *distributed* approaches run on a heterogeneous network of computers. Molnar et al. [MCE*94] propose a classification of parallel renderers into *sort-first*, *sort-middle*, and *sort-last*. In the context of large data volume rendering, *sort-last* approaches are very popular and refer to bricking the data and making each node responsible for rendering one or several bricks before final image compositing. In contrast, sort-first approaches subdivide the viewport and assign render nodes to individual image tiles. Neumann [Neu94] examines the communication costs for different parallel volume rendering algorithms.

Conceptually, all or any parts of the visualization pipeline can be run as a distributed or parallel system. Recent developments in this field are promising trends towards exascale visualization. However, covering the plethora of distributed and parallel volume visualization approaches is out of scope of this survey. The interested reader is referred to [Wit98, BSS00, ZSJ*05] and [BCH12] (Chapter 3) for in-depth surveys on this topic.

5. GPU-Based Volume Rendering

In this section we categorize and discuss the individual literature in GPU-based large-scale volume rendering. We start by introducing a categorization and then give an overview of the evolution of GPU-based volume rendering techniques.

5.1. Categorization

We categorize GPU-based volume rendering approaches with respect to their scalability properties by using the central notion of the *working set*—the subset of volume bricks that is required for rendering a given view. Using the concept of this working set, our categorization distinguishes different approaches according to:

1. How the working set is determined ("culling").
2. How the working set is stored (represented) on the GPU.
3. How the working set is used (accessed) during rendering.

Earlier approaches for volume rendering large data have mainly focused on (2) and (3), but only recent developments in ray-guided working set determination (1) achieve truly scalable performance. We will discuss these respective issues bottom-up, and first describe how to access and traverse the working set during rendering (3) in Section 6.2, and elaborate on working set storage and GPU memory management (2) in Section 6.3. Techniques for determining the current working set via different methods for culling (1) are discussed in Section 7. General volume data structures have already been discussed in Section 3.

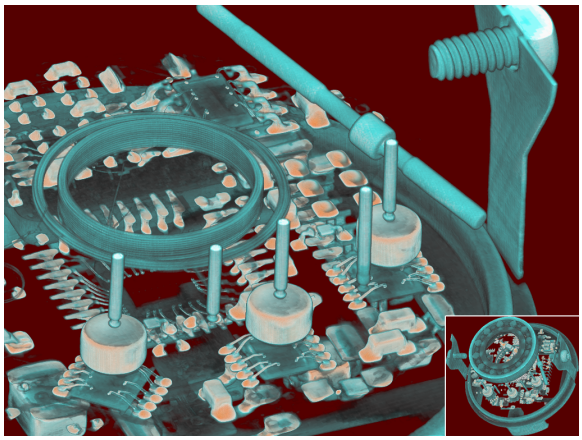

Figure 4: *Rendering a multi-gigabyte CT data set (as used in [Eng11]) at different resolution levels using a ray-guided rendering approach. Data courtesy of Siemens Healthcare, Components and Vacuum Technology, Imaging Solutions. Data was reconstructed by the Siemens OEM reconstruction API CERA TXR (Theoretically Exact Reconstruction).*

We also categorize the resulting *scalability* (low, medium, high), where only "high" scalability means full output-sensitivity and thus independence of the input volume size. The properties of different volume rendering approaches—and the resulting scalability—vary greatly between what we refer to as "traditional" approaches (corresponding to "low" and "medium" scalability in Table 2), and "modern" ray-guided approaches (corresponding to "high" scalability in Table 2).

A key feature of modern ray-guided and visualization-driven volume renderers is that they make full use of recent developments in GPU programmability. They usually include a read-back mechanism to update the current working set, and traverse a multi-resolution hierarchy dynamically on the GPU. This flexibility was not possible on earlier GPUs and is crucial for determining an accurate and tight (as small as possible) working set.

5.2. Evolution of GPU-Based Volume Rendering

GPUs have, over the last two decades, become very versatile and powerful parallel processors, succeeding the fixed-function pipelines of earlier graphics accelerators. General purpose computing on GPUs (GPGPU)—now also called GPU Compute—leverages GPUs for non-graphics related and compute-intensive computations [OLG*07], such as simulations or general linear algebra problems. Increased programmability has been made possible by APIs like the OpenGL Shading Language (GLSL) [Ros06] and CUDA [NVI13].

working set determination	full volume	basic culling (global, view frustum)		ray-guided / visualization-driven
volume data representation (storage)	linear (non-bricked) volume storage [CN93] [CCF94] [WE98] [RSEB*00] [HBH03] [LMK03]† [RGW*03] [KW03] [SSKE05] [BG05] [MHS08] [KGB*09]† [MRH10]	single-resolution grid [HSSB05] [BHWB07] grid with octree per brick [RV06]	octree [LHJ99] [WWH*00] [GGSe*02] [GS04] [PHKH04] [HFK05] kd-tree [FK10] multi-resolution grid [Lju06a] [BHMF08] [JBH*09]	octree [GMG08]‡ [CNLE09] [Eng11] [RTW13] multi-resolution grid [HBJP12] [BAaK*13] [FSK13]
rendering (ray traversal)	texture slicing [CN93] [CCF94] [WE98] [RSEB*00] [HBH03] [LMK03]† non-bricked ray-casting (multi-pass) [RGW*03] [KW03] (single-pass) [SSKE05] [BG05] [MHS08] [KGB*09]† [MRH10]	CPU octree traversal (multi-pass) [LHJ99] [WWH*00] [GGSe*02] [GS04] [PHKH04] [HFK05] [RV06] CPU kd-tree traversal (multi-pass) [FK10] bricked/virtual texture ray-casting (single-pass) [HSSB05] [Lju06a] [BHWB07] [BHMF08] [JBH*09]		GPU octree traversal (single-pass) [GMG08]‡ [CNLE09] [Eng11] [RTW13] multi-level virtual texture ray-casting (single-pass) [HBJP12] [BAaK*13] [FSK13]
scalability	low	medium		high

Table 2: ***Categorization of GPU-based volume visualization techniques*** *based on the type of* working set determination *mechanism and the resulting* scalability *in terms of data size, as well as according to the volume data representation employed, and the actual rendering technique (type of ray traversal; except in the case of texture slicing).* † *[LMK03, KGB*09] perform culling for empty space skipping, but store the entire volume in linear (non-bricked) form.* ‡ *[GMG08] is not fully ray-guided, but utilizes interleaved occlusion queries with similar goals (see the text).*

However, GPU on-board memory sizes are much more limited than those of CPUs. Therefore, large-scale volume rendering on GPUs requires careful algorithm design, memory management, and the use of out-of-core approaches.

Before discussing current state-of-the-art ray-guided volume renderers, we review traditional GPU volume rendering approaches. We start with 2D and 3D texture slicing methods, before continuing with GPU ray-casting. This will give us the necessary context for categorizing and differentiating between the more traditional and the more modern approaches. A detailed comparison of different GPU-based volume rendering techniques is shown in Table 3.

Texture slicing. The earliest GPU volume rendering approaches were based on texture mapping [Hec86] using 2D and 3D texture slicing [CN93, CCF94]. Westermann and Ertl [WE98] extended this approach to support arbitrary clipping geometries and shaded iso-surface rendering. For correct tri-linear interpolation between slices, Rezk-Salama et al. [RSEB*00] made use of multi-texturing. Hadwiger et al. [HBH03] described how to efficiently render segmented volumes on GPUs and how to perform two-level volume rendering on GPUs, where each labeled object can be rendered with a different render mode and transfer function. This approach was later extended to ray-casting of multiple segmented volumes [BHWB07]. Engel et al. [ESE00] were among the first to investigate remote visualization using hardware-accelerated rendering.

Texture slicing today. In general, the advantage of texture slicing-based volume renderers is that they have minimal hardware requirements. 2D texture slicing, for example, can be implemented in WebGL [CSK*11] and runs efficiently on mobile devices without 3D texture support. However, a disadvantage is that they often exhibit visual artifacts and less flexibility when compared to ray-casting methods.

Ray-casting. Röttger et al. [RGW*03] and Krüger and Westermann [KW03] were among the first to perform ray-casting on GPUs, using a multi-pass approach. Ray-casting (also called ray-marching) is embarrassingly parallel and can be implemented on the GPU in a fragment shader or compute kernel, where each fragment or thread typically casts one ray through the volume. Ray-casting easily admits a wide variety of performance and quality enhancements such as empty space skipping and early ray termination. Hadwiger et al. [HSSB05] and Stegmaier et

al. [SSKE05] were among the first to perform GPU ray-casting using a single-pass approach, taking advantage of dynamic looping and branching in then-recent GPUs. Proxy geometries for efficient empty space skipping can be based on bricks [HSSB05, SHN*06], spheres [LCD09], or occlusion frustums [MRH08].

With the introduction of CUDA as a higher-level GPU programming language, CUDA-based ray-casters were introduced [MHS08, KGB*09, MRH10]. They make use of CUDA's thread/block architecture, and possibly shared memory model.

Bricked ray-casting. To support volumes that are larger than GPU memory, in principle each brick can be ray-cast in a separate rendering pass, but it is much more efficient to render the bricked volume via single-pass ray-casting [HSSB05, BHWB07, JBH*09]. Single-pass ray-casters usually store the current working set of bricks in a large brick cache (or brick pool) texture. This requires some form of *address translation* (Section 6) from *"virtual"* (volume) to *"physical"* (cache) texture coordinates on-the-fly during ray-casting.

To some extent, this is similar to the early idea of adaptive texture maps [KE02], but usually the brick cache is managed fully dynamically. In volume rendering, the two main reasons why the brick cache has to be managed dynamically are to be able to accommodate changes to (a) the current view, but also—very importantly—to (b) the current transfer function.

Multi-resolution rendering. There are several motivations for multi-resolution rendering. In addition to the obvious advantages of data reduction and rendering speed-ups, choosing a resolution that matches the current screen resolution reduces aliasing artifacts due to undersampling [Wil83].

For rendering large data, several multi-resolution octree rendering methods have been proposed, most of them based on texture-slicing [LHJ99, WWH*00, GGSe*02, GS04, PHKH04]. Hong et al. [HFK05] used a min-max octree structure for ray-casting the Visible Human CT data set.

A multi-resolution data structure requires level-of-detail (LOD) or scale selection [LB03] for rendering. Weiler et al. [WWH*00] us a focus point oracle based on the distance from the center of a brick to a user-defined focus point to select a brick's LOD. Other methods to select a level of detail include estimating the screen-space error [GS04], using a combined factor of data homogeneity and importance [BNS01] or using the predicted visual significance of a brick [Lju06b]. A common method estimates the projected screen space size of the corresponding voxel/brick [CNLE09]. Whereas LOD selection is often performed on a per-brick basis, Hadwiger et al. [HBJP12] select the LOD on a per-sample basis for finer LOD granularity (see Figure 9).

Ljung et al. [Lju06a] used a multi-resolution bricking structure and adaptive sampling in image- and object-space to render large data. Beyer et al. [BHMF08] proposed a technique for correct interpolation between bricks of two different resolution levels.

The most common data refinement strategy (e.g., when quickly zooming-in on the data) consists of a "greedy" approach that iteratively loads the next higher-resolution of the brick until the desired resolution is reached [CNLE09]. A different approach, where the highest resolution is loaded directly and intermediate resolutions are skipped was proposed in [HBJP12]. Most recently, Fogal et al. [FSK13] found that the "greedy" approach converges in the fewest number of frames in their ray-guided ray-caster.

Parallel and remote volume rendering methods. A lot of research has focused on remote, parallel, or distributed visualization for rendering large data, which we cannot all cover here.

Texture slicing has been used in many distributed and parallel volume rendering systems [MHE01, CMC*06, MWMS07, EPMS09, FCS*10]. Magallon et al. [MHE01] used sort-last rendering on a cluster, where each cluster node renders one volume brick before doing parallel compositing for final image generation. For volume rendering on small to medium GPU clusters, Fogal et al. [FCS*10] introduced a load-balanced sort-last renderer integrated into VisIt [CBB*05], a parallel visualization and data analysis framework for large data sets. Moloney et al. [MWMS07] proposed a sort-first technique using eight GPUs, where the render costs per pixel are used for dynamic load balancing between nodes. They later extended their method to support early ray termination and volume shadowing [MAWM11]. Equalizer [EPMS09] is a GPU-friendly parallel rendering framework that supports both sort-first and sort-last approaches.

Ray-casting for distributed and parallel volume rendering first focused on CPU methods before first GPU methods were developed. Wang et al. [WGL*05] use a parallel CPU ray-caster on a PC cluster for rendering large time varying volumes represented as a wavelet-based time space partitioning tree. Müller et al. [MSE06] used GPU ray-casting in a sort-last parallel rendering system.

A method for rendering large remote micro-CT scans using an octree was proposed by Prohaska et al. [PHKH04].

Other large data methods. A different approach to dealing with large data was proposed by Turlington et al. [THM01], who introduced sliding thin slab (STS) visualization to limit the amount of data needed for any current view. Knoll et al. [KTW*11] optimized CPU ray-casting, achieving interactive rates using a bounding volume hierarchy (BVH) min/max acceleration structure and SIMD optimizations.

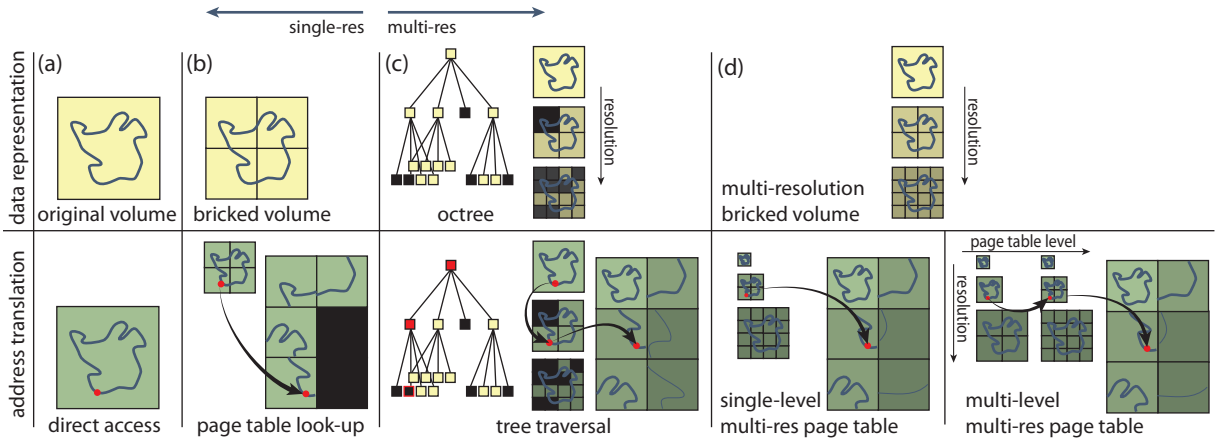

Figure 5: ***Address translation for different data representations. The red dot represents a given sample to be accessed.*** *The figure illustrates both single-resolution (left) and multi-resolution (right) data structures. (a) direct access in non-bricked volume; (b) page table look-up for bricked volume; (c) octree traversal, proceeding from the root downward; (d) multi-resolution page table look-up for multi-resolution bricked volume—page tables enable accessing volume bricks directly, without having to traverse different resolution levels. Multi-level page tables handle large volumes by also virtualizing the page tables (Fig. 7).*

6. Address Translation

One of the crucial properties of rendering large bricked volumes in an out-of-core fashion is that not all bricks have to be resident in on-board GPU memory for rendering. In order to be able to do this, it must be possible to *address* the volume data stored in individual bricks as required by any given sampling location, even if the brick storage order or format are not directly related to the original (non-bricked) volume.

For ray-casting, it is beneficial to specify the sample locations in a space that covers the whole bounding box of the volume and that is independent of individual bricks. For example, all sample locations can be specified via normalized coordinates in a $[0,1]^3$ unit cube. In order to be able to sample the corresponding volume data independently of where each brick's data are actually located, the volume coordinates of a given sample need to be *translated* to actual texture coordinates, taking the "physical" storage of the brick's data into account. The latter most commonly are the texture coordinates of a box-shaped sub-region in a specific texture that stores the data of multiple different bricks, for example all the bricks comprising the current working set.

Terminology. We refer to this process as *address translation*, and denote the volume coordinates used for specifying sample locations along viewing rays as *virtual volume coordinates* or *virtual addresses*, and the corresponding coordinates in actual texture space as *physical addresses*. We often use the term *page table* to refer to the concept as well as the data structure that actually stores the current mapping from virtual addresses to physical addresses in some form, in analogy to standard CPU virtual memory terminology [HP11]. A *page* in this context most commonly refers to a volume brick, i.e., a page table references physical brick storage.

6.1. Categorization

We categorize the different address translation methods according to the volume data representation that they work with, and according to how their "page table" is organized and accessed. Figure 5 illustrates the major categories, given different volume data representations and the corresponding ways of performing address translation in volume rendering.

Conceptual framework. For a presentation and categorization that is as uniform as possible over all important volume data representations and the corresponding rendering techniques, we will use the unified conceptual framework of address translation for all representations and rendering techniques that we discuss, even if they are not traditionally thought of as employing address translation.

For example, octree volume rendering is commonly thought of as a tree traversal process that renders volume data stored in bricks attached to the individual octree nodes. However, this process can also be thought of as a particular form of address translation, where the octree itself contains the information that is required to translate virtual to physical addresses, and the address translation itself is built on tree traversal. From this viewpoint, the octree itself represents the "page table," just in a very particular, hierarchical form, and the tree traversal traverses this page table.

We make a major distinction between *single-resolution* vs. *multi-resolution* approaches, and between approaches employing *single-level* vs. *multi-level* page tables.

Single- vs. multi-resolution. A subtle but important distinction must be made between *data* of a single or multiple resolutions, and *page tables* of a single or multiple resolutions, respectively. In our categorization (see Figure 5 and Section 6.2) of address translation methods, we categorize according to the properties of the page table. An example for this subtle difference that is described below would be adaptive texture maps [KE02]. They employ a single-resolution page table, but can nevertheless reference data of different resolutions. We put such a technique into the *single-resolution* category (Section 6.2.2), although we also mention it again in the multi-resolution category to try to minimize confusion.

Single- vs. multi-level. Most published work that uses page tables (or, equivalently, index textures) employs a single *level* of page table, i.e., each page table entry references *data*. For very large volumes, however, it is beneficial to employ *multiple levels* of page tables, where page table entries can again reference page tables, leading to a *page table hierarchy* [HBJP12]. The latter approaches we denote as using *multi-level page tables* (Figures 5 and 7, and Section 6.2.5).

6.2. Address Translation Methods

This section categorizes and discusses the major methods of address translation in volume rendering that have been reported in the literature according to the discussion above.

6.2.1. Direct Access

For completeness, we start with the simplest form of "address translation," which is essentially an identity operation.

As illustrated in Figure 5 (a), the volume is stored linearly without bricking, e.g., in a single 3D texture for the entire volume. Therefore, essentially no address translation is necessary, except mapping volume coordinates to actual texture coordinates. If the volume is addressed in a $[0,1]^3$ unit cube, and texture coordinates are also given in $[0,1]^3$ as they commonly are, this mapping is an identity operation. No page tables or similar structures are needed, but out-of-core rendering is not possible and hence scalability is very limited.

6.2.2. Single-Level, Single-Resolution Page Tables

The first really interesting form of address translation is performed for bricked, but single-resolution, volumes. As illustrated in Figure 5 (b), the original volume is bricked for out-of-core rendering, but no multi-resolution hierarchy is computed. This allows for out-of-core volume rendering, but only with volume bricks from the original resolution and no down-sampling. Address translation can be performed with a single page table look-up per sample. The page table can be stored in a linear texture, e.g., a single 3D texture, with one entry (texel) per page (brick) that contains the information necessary for address translation. In this context, the page table texture is sometimes simply called an index texture.

Adaptive texture maps. Kraus and Ertl [KE02] were the first to introduce adaptive texture maps for GPUs, where an image or volume can be stored in a bricked fashion with adaptive resolution and accessed via a look-up in a small index texture. This index texture can be seen as a page table. However, in adaptive texture maps the index texture was statically computed and not updated according to parameter changes such as a changing transfer function. Moreover, the index texture itself only has a single resolution, although it can refer to data of different resolutions. For this reason, and for a better order of introducing the different approaches, we have included adaptive texture maps in this section. However, we also mention this approach again below in Section 6.2.4.

Dynamically managed page tables. One of the earliest uses of dynamically managed page table textures for rendering volume data was introduced in the context of iso-surface ray-casting [HSSB05]. The working set of pages (bricks) potentially intersected by the iso-surface are dynamically paged (downloaded) into a cache texture, which is the "physical" storage. The cache texture is referenced by a page table texture that is likewise updated dynamically. This approach can easily be adapted to direct volume rendering [SHN*06, BHMF08]. The working set of active bricks is then simply determined according to the transfer function instead of the iso-value. However, in these early approaches, this was done independently of view-dependent occlusions.

Virtual texturing. Recent hardware support for *partially resident textures* [BSH12] (e.g., the OpenGL `GL_ARB_sparse_texture` extension) allow the GPU to manage both the page table and the corresponding physical storage, as well as performing the actual address translation in the GPU hardware texture unit. For non-mipmapped textures, this essentially implements what we have described above. For mipmapped textures, it implements what we describe in Section 6.2.4. However, current hardware limitations still limit the size of partially resident textures to 16K pixels/voxels and do not allow for automatic page fault handling. In the context of game engines, virtual texturing approaches have also been used successfully [vW09, OVS12].

6.2.3. Tree Traversal

Tree data structures and traversal algorithms (e.g., kd-trees, octrees) have been employed since the earliest multi-resolution volume renderers. Although usually not described as a form of address translation, in our conceptual framework we view tree traversal as a particular kind of address translation, and the tree itself as a particular kind of page table, as illustrated in Figure 5 (c).

Trees as "page tables." In order to highlight the differences and commonalities between tree traversal and other approaches, we can view the tree data structure as a kind of hierarchical page table, where each tree node contains the information where the corresponding physical volume data

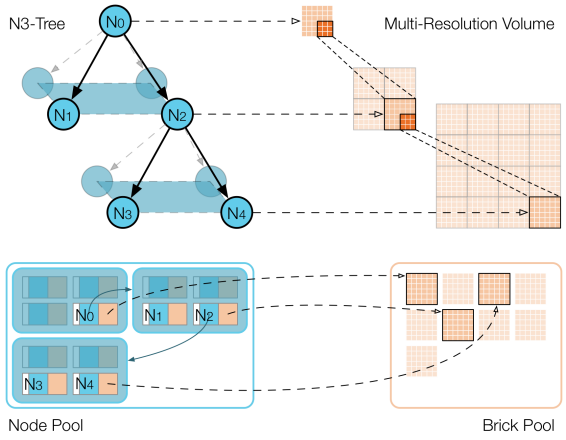

Figure 6: ***The Gigavoxels system*** *[CN09] uses an N^3 tree with node and brick pools storing the set of active nodes and bricks, respectively. Usually, $N = 2$, resulting in an octree.*

are located (the brick data attached to the node), and how to descend in the multi-resolution hierarchy represented by the tree, from coarser to finer resolutions (which can be viewed as a particular kind of page table hierarchy).

Trees vs. multi-level, multi-resolution page tables. Using the viewpoint of a tree being a kind of hierarchical page table, in principle it could also be seen as a *multi-level, multi-resolution page table*. However, we make the crucial distinction that in the case of a standard tree structure, the way in which the resolution of the *data* is reduced, and the way in which the resolution of the *"page table"* is reduced, are completely *coupled*. This means that their parameters cannot be chosen independently. In contrast, in our categorization we use the term *multi-level, multi-resolution page table* for a structure where the hierarchy of the different data resolutions (the *resolution hierarchy*), and the hierarchy of the different page table resolutions (the *page table hierarchy*) are completely *decoupled*. This leads to a structure that cannot be described as a single tree, but that is comprised of two *orthogonal hierarchies* [HBJP12], as shown in Figure 7.

This leads to crucial differences in terms of scalability, which are discussed in more detail in Section 6.2.5.

Traversal algorithms for efficiently navigating and traversing trees, such as kd-trees or octrees have been well researched in the ray-tracing community. Amanatides and Woo [AW87] were the first to introduce a fast regular grid traversal algorithm. Recently, stackless traversal methods such as kd-restart [FS05] have received a lot of attention in practical implementations [PGS*07], as they are well-suited for GPU implementation. Other variants include the kd-shortstack [HSHH07] and kd-jump [HL09] algorithms.

Octree-based systems. To traverse an octree directly on the GPU, not only the current working set of bricks, but also a (partial) tree needs to be stored on the GPU. (The latter being the equivalent of a page table that is only partially resident).

Gobbetti et al. [GMG08] use a spatial index structure to store the current subtree with neighbor information based on rope trees [HBZ98]. Each octree node stores pointers to its eight children and its six neighbors (the "ropes" [HBZ98, PGS*07]), and a pointer to the actual brick data. Traversal leverages the rope pointers.

Crassin et al. [CN09, CNLE09] use an N^3 tree (although they usually use $N = 2$, i.e., a regular octree), whose current subtree is stored in a node pool and a brick pool, respectively. Each node stored in the node pool contains one pointer to its N^3 children, and one pointer to the corresponding brick data in the brick pool (see Figure 6). Using a single child pointer is possible because all N^3 children are always stored together in a single node pool entry. Tree traversal is based on an adapted kd-restart algorithm [FS05]. Engel [Eng11] and Reichl et al. [RTW13] use the same basic structure and traversal method.

6.2.4. Single-Level, Multi-Resolution Page Tables

The simplest form of supporting multiple resolutions with page tables is to employ a separate page table per resolution level, as illustrated in Figure 5 (d, left). Everything else is then very similar to the single-resolution case (Section 6.2.2). For each sample, first the desired resolution level is determined (e.g., according to projected screen size of a voxel or brick), which then allows accessing the corresponding page table. Usually, all pages (bricks) have the same physical resolution (in terms of voxels), independently of which resolution level they correspond to. This is very beneficial in practice, because the working set can mix bricks from different resolution levels easily, and all active bricks can be stored in the same cache texture (because they are all of exactly the same size). Such approaches have been employed for relatively large data (e.g., [JBH*10]), but better scalability can be achieved by using multiple page table levels per resolution level instead of just a single one [HBJP12] (Section 6.2.5).

Virtual texturing. As already described in Section 6.2.2, virtual texturing approaches often also employ page tables. If mipmapped textures are used, at least conceptually there is a separate page table for each mipmap level, i.e., for each resolution level. However, in this case all page tables together can also be stored as a single *mipmapped page table*.

6.2.5. Multi-Level, Multi-Resolution Page Tables

For very large volume data, the concept of single-level, multi-resolution page tables described above starts to become impractical in terms of page table storage. However, it can be extended in a relatively simple, "recursive" fashion in order to achieve much higher scalability. The resulting *multi-level, multi-resolution* structure is illustrated in Figure 5 (d, right), and in much more detail in Figure 7. (Note

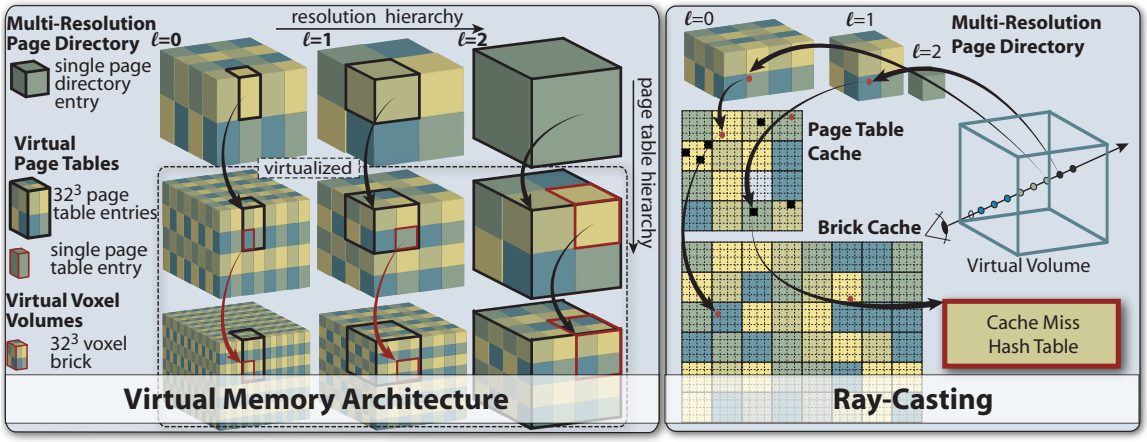

Figure 7: ***Multi-level, multi-resolution page tables*** *[HBJP12]. Left: The virtual memory architecture comprises two* orthogonal *hierarchies: the* resolution hierarchy*, and the* page table hierarchy*. Right: Address translation during ray-casting starts in the multi-resolution page directory (one directory per resolution level), and then references cache textures shared for all resolutions.*

that the resolution and page table hierarchies, respectively, are mapped differently to the horizontal/vertical axes in Figures 5 and 7, respectively.)

The basic problem of using single-level page tables is that for large volumes and small brick sizes (such as 32^3 voxel bricks, for good culling granularity), the page tables themselves become very large. That is, the page tables *virtualize* the volume data, but the page tables themselves are not virtualized. For example, the page table corresponding to the finest resolution level of a $128K^3$ volume would itself be a $4K^3$ texture. However the basic idea of using page tables to virtualize large volumes can easily be extended "recursively" to the page tables themselves, leading to the *page table hierarchy* illustrated in Figure 7. Such multi-level page table architectures have been shown to scale well to volume data of extreme scale [BHL*11, HBJP12], exhibiting better scalability than tree hierarchies such as standard octrees.

In a multi-level page table architecture, one or more additional levels of page tables are added to each resolution level [BHL*11, HBJP12]. Each top level page table is called a *page directory*, in analogy to CPU virtual memory terminology [HP11]. That is, in order to be able to address multiple resolution levels, each resolution level has its own corresponding page directory. Collectively, the page directories of all resolution levels are refered to as the *multi-resolution page directory*. Only this multi-resolution directory structure must be resident in GPU memory, i.e., is not virtualized. All other page table hierarchy levels are completely virtualized.

Now, in addition to the cache texture storing the current working set of bricks, an additional *page table cache* texture is required for storing the current *working set of page table bricks*. The brick sizes for voxel (data) bricks and for page table bricks can be chosen completely independently.

Hadwiger et al. [HBJP12] describe multi-level, multi-resolution page tables as a (conceptually) orthogonal 2D structure (see Figure 7, left). One dimension corresponds to the page table hierarchy, consisting of the page directories (the top-level page tables) and the page tables "below." The second dimension corresponds to the different resolution levels of the data. Each resolution level conceptually has its own independent page table hierarchy. However, the actual cache textures can again be shared between all resolution levels. Multi-level page tables scale very well to large data. For example, just two page table levels have been shown to already support volumes of up to several hundred terabytes, and three levels would in principle be sufficient for even exa-scale data in terms of "addressability" [HBJP12].

Address translation. The right part of Figure 7 depicts address translation during ray-casting in a multi-level, multi-resolution page table architecture. Hadwiger et al. [HBJP12] use this approach for rendering extreme-scale electron microscopy data. Their approach starts with computing a LOD for the current sample, which is then used to look up the page directory corresponding to that resolution. Next, address translation traverses the page table hierarchy from the page directory through the page table levels below.

Look-up overhead. A property of this approach for address translation that is crucial in order to obtain high volume rendering performance is that when marching from sample to sample along a given ray, the page directory and page table look-ups from the previous sample can trivially be cached (remembered) in order to exploit spatial coherence. Because the result of these look-ups changes less and less frequently the higher up in the hierarchy they come from, the number of actual texture look-ups that is required for address translation in practice is very low. Because of this fact,

the performance difference between a single-level page table architecture and a multi-level one is very low [HBJP12].

Multi-level, multi-resolution page tables vs. trees. Expanding on the previous discussion in Section 6.2.3, the crucial property why an orthogonal multi-level, multi-resolution page table hierarchy scales better to very large volume data than tree structures such as octrees is that the former completely *decouple* the hierarchy that is required for scalable address translation (the page table hierarchy) from the hierarchy that is required in order to be able to perform multi-resolution volume rendering (the resolution hierarchy). While either of these hierarchies could also be viewed as a tree (without pointers), the orthogonal structure depicted in Figure 7 cannot be described by a *single* tree.

When a hierarchical tree structure is built, a crucial decision is how many children each tree node should have. In standard tree-based volume rendering approaches, this number determines the *resolution reduction* or *down-sampling ratio* from resolution level to resolution level. In an octree (or an N^3 tree with $N = 2$), this down-sampling ratio is fixed to a factor of two per level in each dimension. This is the most common down-sampling ratio that yields relatively good results. However, for volume rendering it corresponds to an already relatively strong jump in data quality from level to level, because the total number of samples is reduced in jumps by a factor of *eight*. This fact has been observed and tackled by very few researchers [EMBM06]. This makes the use of an N^3 tree with $N > 2$ impractical for high-quality volume rendering.

On the other hand, the number of look-ups that are required for *address translation* is in principle a completely separate consideration from the desired down-sampling ratio of the volume data. Traversal performance for large data crucially depends on the number of hierarchy (tree) levels that must be traversed. For this reason, an octree ($N = 2$) leads to a relatively large number of traversal steps for large data [HBJP12]. In order to obtain traversal performance with better scalability characteristics, a tree with more children per node than 2^3 could be used, i.e., an N^3 tree with $N > 2$. The results reported in [HBJP12] can to some extent be compared on a conceptual level to using an N^3 tree with $N = 32$ for the page table structure (not for data down-sampling).

This discussion shows that two opposing considerations must be balanced for high-quality rendering and good scalability to large data—choosing N as small as possible for high-quality rendering, and choosing a rather large N for better scalability to large data. Because a single tree structure cannot choose these two factors independently, they are inherently coupled. In contrast, a multi-level, multi-resolution page table hierarchy allows completely decoupling these two considerations. This allows choosing a down-sampling ratio according to quality requirements, and a ratio for page table "size reduction" according to scalability requirements.

6.2.6. Hash Tables

An alternative data structure that can be employed for address translation on GPUs are hash tables. However, their use for this purpose has not yet received a lot of attention in the context of large-scale volume rendering.

Hastings et al. [HMG05] used spatial hashing to optimize collision detection in real-time simulations. Nießner et al. [NZIS13] use voxel hashing for real-time 3D reconstruction of iso-surfaces in dynamically constructed distance field volumes.

The crucial property of hashing approaches are how they handle collisions, i.e., how often collisions can occur in practice and how complicated and costly they are to resolve. Without the need for resolving collisions, hash tables would be rather straightforward to implement. However, proper collision handling makes their implementation much more involved, e.g., [NZIS13].

6.3. GPU Memory Management

Efficient GPU data structures for storing the working set should be fast to access during ray traversal (i.e., address translation), and should also support efficient dynamic updates of the working set. Recent approaches usually store volume bricks (actual voxel data) in a singe large 3D cache texture (or brick pool).

In addition to that, a address translation helper structures (e.g., page table, octree pointers) are required for translating from virtual to texture space. If ray traversal needs to follow tree nodes (as in octree-based renderers), the working set of tree nodes must be stored, e.g., in a *node pool* (e.g., [CNLE09, Eng11]). If ray traversal is built on virtual to physical address translation (as in page table-based renderers), the working set of page table entries must be stored, e.g., in a *page table cache* (e.g., [BHL*11, HBJP12]).

Finally, additional meta-data that is required by the renderer has to be stored on the GPU as well.

6.3.1. Texture Cache Management

Texture allocation. Early tree-based volume renderers often employed one texture per brick, rendering one after the other in visibility order using one rendering pass per brick/tree node [LHJ99, WWH*00, GGSe*02, GS04]. However, multi-pass approaches are usually less performant than single-pass approaches and are also limited in the number of passes they can efficiently perform. To circumvent rendering bottlenecks due to many rendering passes, Hong et al. [HFK05] cluster bricks in layers (based on the manhattan distance) and render all bricks of the same layer at the same time.

To support single-pass rendering, bricking approaches and modern ray-guided renderers usually use a single large 3D cache texture (or brick pool) to store the working

set [BHWB07, CN09, HBJP12], and often assume that the working set will fit into GPU memory.

When the working set does not fit into GPU memory, either the level of detail and thus the number of bricks in the working set can be reduced [HBJP12], or the renderer can switch to a multi-pass fall-back [Eng11, FSK13].

Texture updates. Whenever the working set changes, the cache textures have to be updated accordingly. Hadwiger et al. [HBJP12] compare texture update complexity between octree-based and multi-level page table approaches. Octree-based approaches usually have to do a large number of updates of small texture elements, whereas hierarchical page tables tend to perform fewer but larger updates.

To avoid cache thrashing [HP11], different brick replacement strategies have been introduced. Most common is the LRU scheme which replaces the brick in the cache that was least recently used [GMG08, CN09, FSK13]. It is also common to use a hybrid LRU/MRU scheme, where the LRU scheme is used unless the cache is too small for the current working set. In the latter case, the scheme is switched to MRU (most recently used) to reduce cache thrashing.

7. Working Set Determination (Culling)

Originally, the concept of *culling* was introduced in computer graphics for geometry rendering, where typically *view frustum culling* and *occlusion culling*[†] are used to limit the number of primitives (e.g., triangles) that have to be rendered. That is, in the former case culling everything that is *outside* the view frustum, and in the latter case culling everything that is *occluded* within the view frustum, given the current view direction and other relevant parameters. Ideally, all occluded geometry should be culled before rendering. In practice, for performance reasons a fast *conservative* estimate is computed, i.e., erring on the safe side by—to some extent—over-estimating the visible (non-occluded) geometry.

Performing culling to determine the current working set of bricks in volume rendering is crucial for being able to handle large data at interactive frame rates and with a limited amount of memory (e.g., on-board GPU memory). That is, we want to cull everything that is *not* required to be in the working set. All bricks that do not contribute to the output image can be culled. Bricks that do not contribute are mainly bricks that are either (a) outside the view frustum, (b) fully transparent (i.e., have zero opacity), given the current transfer function, or (c) are occluded by bricks in front of them (given the current view parameters as well as the transfer function).

Different culling techniques can exhibit huge differences in their effectiveness, computational complexity, and flexibility, depending on how "tightly" they estimate the working set (accurately vs. conservatively), how fast or complicated it is to perform the actual culling, and which parameters (view, transfer function, etc.) can be changed interactively while still being able to update the culling.

7.1. View Frustum Culling

Removing primitives or volume bricks outside the current view frustum is the most basic form of culling. The first step of GPU ray-casting consists of computing the ray start points and end points (often via rasterization), which already prevents sampling the volume in areas that are outside the view frustum. However, in order to prevent volume bricks outside the frustum from being downloaded to the GPU, the individual bricks have to be culled against the view frustum. Naturally, if a brick lies completely outside the current view frustum, it is not needed in GPU memory. Culling a view frustum against a bounding box, a bounding volume hierarchy, or a tree, can be done very efficiently and has been studied extensively in several contexts [AM00, AMHH08].

7.2. Global, Attribute-Based Culling

Another way to cull bricks in volume rendering is based on global properties like the current transfer function, iso value, or enabled segmented objects. Culling against the transfer function is usually done based on min/max computations for each brick [PSL*98, HSSB05, SHN*06]. The brick's min/max values are compared against the transfer function to determine if the brick is invisible (i.e., only contains values that are completely transparent in the transfer function). Invisible bricks are then culled. The downside of this approach is that it needs to be updated whenever the transfer function changes and usually needs pre-computed min/max values for each brick that have to be available at runtime for all bricks. A similar approach can be used for culling bricks against an iso-surface [PSL*98, HSSB05], or against enabled/disabled objects in segmented volume rendering [BHWB07].

7.3. Occlusion Culling

Occlusion culling tries to cull primitives (bricks) that are inside the view frustum but are nevertheless occluded by other primitives (bricks/volume data). This process usually requires a multi-pass rendering approach. While occlusion culling is easier to perform for opaque geometry, for semi-transparent geometry—or for (semi-transparent) volume rendering—this process is more involved.

Greene et al. [GKM93] introduced hierarchical z-buffer visibility. They used two hierarchical data structures—an octree in object space, and a z-pyramid in image space—to quickly reject invisible primitives in a hierarchical manner. Zhang et al. [ZMHH97] proposed hierarchical occlusion maps (HOMs), where first a set of occluders is rendered

[†] Note that occlusion culling is sometimes also referred to as *visibility culling* (in an inverted sense).

into a low-resolution occlusion map that is then hierarchically downsampled and subsequently used to test primitives for occlusion before actually rendering them.

In volume visualization, Li et al. [LMK03] introduced occlusion clipping for texture-based volume rendering to avoid rendering occluded parts of the volume. Gao et al. [GHSK03] proposed visibility culling in large-scale parallel volume rendering based on pre-computing a plenoptic opacity function per brick. Visibility culling based on temporal occlusion coherence has also been used for time-varying volume rendering [GSHK04]. The concept of occlusion culling has also been used in a parallel setting for sort-last rendering [MM10], by computing and propagating occlusion information across rendering nodes.

7.4. Ray-Guided Culling

Ray-guided culling approaches are different in the sense that instead of *culling away* bricks from the set of all bricks, they start with an empty working set that is grown by only *adding* bricks that are actually required (intersected) during rendering.

That is, the goal is that only the bricks that are actually visited during ray traversal are added to the working set of active bricks. This naturally also implies that subsequently only these bricks will be downloaded to GPU memory. In this way, ray-guided culling approaches cull all occluded bricks as well as implicitly also cull all bricks outside the view frustum. For this reason, they usually do not require separate steps for view frustum and occlusion culling, respectively, because no rays are generated outside the view frustum.

Gobbetti et al. [GMG08] used a mixture of traditional culling and ray-guided culling. They first perform culling on the CPU (using the transfer-function, iso value, and view frustum), but refined only those nodes of the octree that were marked as visible in the previous rendering pass. To determine if a node is visible they used GPU occlusion queries to check the bounding box of a node against the depth of the last visited sample that was written out during ray-casting.

Crassin et al. [CN09] originally used multiple render targets to report which bricks were visited by the ray-caster over the course of several frames, exploiting spatial and temporal coherence. In a later reported implementation [CNSE10], the same information was constructed in a more efficient way using CUDA.

Hadwiger et al. [HBJP12] divide the viewport into smaller tiles and use a GPU hash table per image tile to report a limited number of *cache misses* per frame. Over the course of several frames, this ensures that all missing bricks will be reported and added to the working set.

Fogal et al. [FSK13] use a similar approach but use lock-free hash tables.

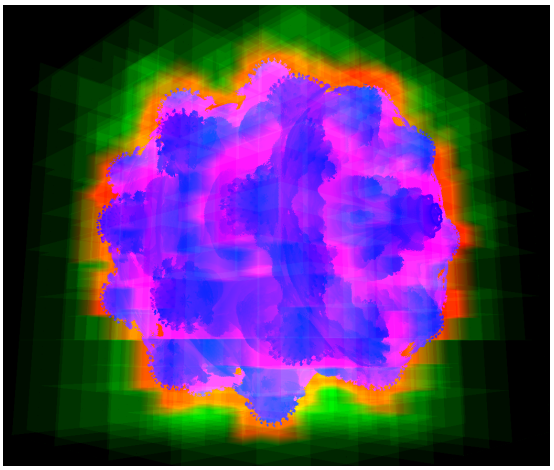

Figure 8: *Ray-guided volume rendering [FSK13] of the Mandelbulb data set. Colors indicate the amount of empty space skipping and sampling that needs to be performed (green: skipped empty brick, red: densely sampled brick, blue: densely sampled but quickly saturated). Image courtesy of Tom Fogal and Jens Krüger.*

7.4.1. Handling Missing Bricks

In contrast to traditional ray-casting approaches, where the working set is computed prior to rendering on the CPU, ray-guided volume renderers only build up the current working set during ray traversal. This implies that ray-guided volume renderers have to be able to deal with missing bricks in GPU memory, because bricks are only requested and downloaded once they have been hit during ray-casting, but not before.

Whenever the ray-caster detects a missing brick (i.e., either a missing octree node or a page table entry that is flagged as *unmapped*), a request for that missing brick is written out. Crassin et al. [CN09] use multiple render targets to report missing nodes and then stop ray traversal. More recent approaches [CNSE10, HBJP12, FSK13] use OpenGL extensions such as `GL_ARB_shader_image_load_store` or CUDA, and often GPU hash tables, to report cache misses.

During rendering, missing bricks can be either skipped, or potentially be substituted by a brick of lower resolution. After missing bricks are detected and reported, the CPU takes care of loading the missing data, downloading it into GPU memory, and updating the corresponding GPU data structures.

7.4.2. Empty Space Skipping

In addition to skipping *missing* bricks, a common optimization strategy in volume rendering that is also easily implemented in ray-guided volume rendering is *empty space skipping* (sometimes also called empty space leaping), i.e., skip-

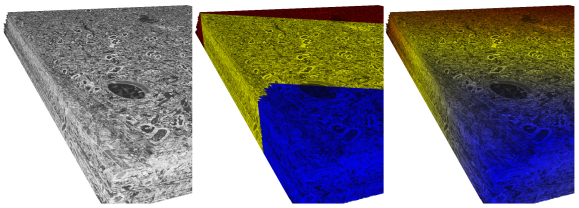

Figure 9: ***Per-sample LOD selection*** *as in [HBJP12]. Left: electron microscopy volume (90 GB). Middle and right: the LOD used for each sample is color-coded. Middle: discrete LOD for each sample (tri-linear interpolation). Right: fractional LOD for each sample, with interpolation between data of neighboring LODs ("quad-linear" interpolation).*

ping (or "leaping over") "*empty space,*" which is usually identified at the granularity of individual bricks.

This optimization relies on knowing (essentially *a priori*) which bricks are completely empty, i.e., contain no valid/useful data or are completely mapped to zero opacity according to the current transfer function, and can therefore be safely skipped during ray-casting. During ray-casting, empty space (bricks) can be identified via "missing" subtrees or marked tree nodes [GMG08, Eng11], or via page table flags [HBJP12]. Figure 8 shows a rendering with color-coded empty space skipping information.

8. Ray-Guided Volume Rendering

In this section we show how the combination of the previously discussed techniques has led to the advent of ray-guided volume renderers that achieve much better scalability and performance than previous techniques. The main novelty of ray-guided (and visualization-driven) volume rendering approaches is that they incorporate a feedback loop between the ray-caster and the culling mechanism, where the ray-caster itself writes out accurate information on missing bricks and brick usage. Thus, this type of culling mechanism determines an accurate working set directly on the GPU.

This information about the working set is then used to load missing data, and to determine which bricks can be evicted from the GPU cache because they are no longer needed. Additionally, rays automatically determine the (locally) required data resolution. This determination can be performed either on a per-sample basis [HBJP12] (see Figure 9), or on a per-brick basis [FSK13].

Gobbetti et al. [GMG08] were among the first to implement a volume ray-caster with stackless GPU octree traversal. They used occlusion queries to determine, load, and possibly refine visible nodes. This approach already has similar properties to later fully ray-guided approaches. However, it is strictly speaking not fully ray-guided, because culling of octree nodes is performed on the CPU based on the occlusion query information obtained from the GPU.

Crassin et al. [CN09] introduced the Gigavoxels system for GPU-based octree volume rendering with *ray-guided streaming* of volume data. Their system can also make use of an N^3 tree, as an alternative to an octree (which would be an N^3 tree with $N = 2$). The tree is traversed at run time using the kd-restart algorithm [FS05] and active tree nodes stored in a *node pool*. Actual voxel data are fetched from bricks stored in a *brick pool*. Each node stores a pointer to its child nodes in the node pool, and a pointer to the associated texture brick in the brick pool (see Figure 6). The focus of the Gigavoxels system is volume rendering for entertainment applications and as such it does not support dynamic transfer function changes [CNSE10]. The more recent CERA-TVR system [Eng11] targets scientific visualization applications and supports fully dynamic updates according to the transfer function in real time. It also uses the kd-restart algorithm for octree traversal. Reichl et al. [RTW13] also employ a similar ray-guided approach, but target large smooth particle hydrodynamics (SPH) simulations.

A different category of ray-guided volume renderers uses hierarchical grids with bricking, which are accessed via multi-level page tables instead of a tree structure. Hadwiger et al. [HBJP12] proposed such a multi-resolution virtual memory scheme based on a multi-level page table hierarchy (see Figure 7). This approach scales to petavoxel data and can also efficiently handle highly anisotropic data, which is very common in high-resolution electron microscopy volumes. They also compare their approach for volume traversal to standard octree traversal in terms of traversal complexity and cache access behavior, and illustrate the advantages of multi-level paging in terms of scaling to very large data.

Fogal et al. [FSK13] have performed an in-depth analysis of several aspects of ray-guided volume rendering.

9. Discussion and Conclusions

In this survey we have discussed different large-scale GPU-based volume rendering methods with an emphasis on ray-guided approaches. Over recent years, sophisticated scalable GPU volume visualization methods have been developed, hand in hand with the increased versatility and programmability of graphics hardware. GPUs nowadays support dynamic branching and looping, efficient read-back mechanisms to transfer data back from the GPU to the CPU, and several high-level APIs like CUDA or OpenCL to make GPU programming more efficient and enjoyable.

Our discussion of scalability in volume rendering was based on the notion of working sets. We assume that the data will never fit into GPU memory in its entirety. Therefore, it is crucial to determine, store, and render the working set of visible bricks in the current view efficiently and accurately. The review of "traditional" GPU volume rendering methods showed that these approaches have several shortcomings that severely limit their scalability. Traditionally, the working set

of active bricks is determined on the CPU and no read-back mechanism is used to refine this working set. Additionally, due to previously limited branching or looping functionality on GPUs, renderers often had to resort to multi-pass rendering approaches. Modern ray-guided approaches exhibit better scalability, they support dynamic traversal of multi-resolution structures on the GPU, and they allow determining the working set of active bricks based on actual visibility by employing efficient read-back mechanisms from the GPU to the CPU. Therefore, ray-guided approaches are promising for the future, where data set sizes will continue to increase.

In this survey we have focused on GPU-based approaches for single stand-alone workstations. However, there is a huge area of parallel and distributed visualization research that focuses on clusters, in-situ setups and client/server systems. Additionally, we expect web-based visualization to become more and more important, which will make it necessary to research scalable algorithms for remote visualization and mobile devices. Finally, as data sets get larger and scalable volume rendering methods more mature, it will become more and more important to have efficient workflows and integrated solutions that encompass the whole data flow through a system, from data acquisition and pre-processing to interactive visualization and analysis.

10. Acknowledgments

This work was partially supported by NSF grant OIA 1125087.

References

[AAM*11] AHERN S., ARIE S., MA K.-L., CHOUDHARY A., CRITCHLOW T., KLASKY S., PASCUCCI V., AHRENS J., BETHEL W. E., CHILDS H., HUANG J., JOY K., KOZIOL Q., LOFSTEAD G., MEREDITH J. S., MORELAND K., OSTROUCHOV G., PAPKA M., VISHWANATH V., WOLF M., WRIGHT N., WU K.: *Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization*. Tech. rep., Department of Energy, 2011. 1, 6, 8

[ALN*08] AHRENS J., LO L.-T. L.-T., NOUANESENGSY B., PATCHETT J., MCPHERSON A.: Petascale Visualization: Approaches and Initial Results. In *Workshop on Ultrascale Visualization, 2008. UltraVis '08.* (2008), pp. 24–28. 6

[AM00] ASSARSSON U., MOLLER T.: Optimized View Frustum Culling Algorithms for Bounding Boxes. *Journal of Graphics Tools 5*, 1 (Jan. 2000), 9–22. 17

[AMHH08] AKENINE-MÖLLER T., HAINES E., HOFFMAN N.: *Real-Time Rendering 3rd Edition*. A. K. Peters; Ltd., 2008. 17

[AW87] AMANATIDES J., WOO A.: A Fast Voxel Traversal Algorithm for Ray Tracing. In *Eurographics '87* (1987), pp. 3–10. 5, 14

[AWS92] AHLBERG C., WILLIAMSON C., SHNEIDERMAN B.: Dynamic Queries for Information Exploration: an Implementation and Evaluation. In *SIGCHI Conference on Human Factors in Computing Systems* (1992), CHI '92, pp. 619–626. 7

[BAaK*13] BEYER J., AL-AWAMI A., KASTHURI N., LICHTMAN J. W., PFISTER H., HADWIGER M.: ConnectomeExplorer: Query-Guided Visual Analysis of Large Volumetric Neuroscience Data. *IEEE Transactions on Visualization and Computer Graphics (Proc. of IEEE SciVis '13) 19*, 12 (2013), 2868–2877. 8, 10, 25

[BCH12] BETHEL E. W., CHILDS H., HANSEN C.: High Performance Visualization – Enabling Extreme-Scale Scientific Insight. Chapman & Hall, CRC Computational Science. CRC Press/Francis–Taylor Group, Nov. 2012. 1, 2, 4, 7, 8, 9

[BG05] BRUCKNER S., GRÖLLER M.: Volumeshop: An Interactive System for Direct Volume Illustration. In *IEEE Visualization '05* (2005), pp. 671–678. 10, 25

[BHAA*13] BEYER J., HADWIGER M., AL-AWAMI A., JEONG W.-K., KASTHURI N., LICHTMAN J., PFISTER H.: Exploring the Connectome - Petascale Volume Visualization of Microscopy Data Streams. *IEEE Computer Graphics and Applications 33*, 4 (2013), 50–61. 2, 4, 5, 8

[BHL*11] BEYER J., HADWIGER M., LICHTMAN J., REID R. C., JEONG W.-K., PFISTER H.: Demand-Driven Volume Rendering of Terascale EM Data. In *SIGGRAPH '11: Technical talk* (2011). 6, 15, 16

[BHMF08] BEYER J., HADWIGER M., MÖLLER T., FRITZ L.: Smooth Mixed-Resolution GPU Volume Rendering. In *IEEE International Symposium on Volume and Point-Based Graphics (VG '08)* (2008), pp. 163–170. 6, 10, 11, 13, 25

[BHWB07] BEYER J., HADWIGER M., WOLFSBERGER S., BÜHLER K.: High-Quality Multimodal Volume Rendering for Preoperative Planning of Neurosurgical Interventions. *IEEE Transactions on Visualization and Computer Graphics (Proc. of IEEE Visualization '07)* (2007), 1696–1703. 5, 10, 11, 17, 25

[BLK*11] BOCK D., LEE W.-C., KERLIN A., ANDERMANN M., HOOD G., WETZEL A., YURGENSON S., SOUCY E., KIM H. S., REID R. C.: Network Anatomy and In Vivo Physiology of Visual Cortical Neurons. *Nature 471*, 7337 (2011), 177–182. 1

[BNS01] BOADA I., NAVAZO I., SCOPIGNO R.: Multiresolution Volume Visualization with a Texture-based Octree. *The Visual Computer 17*, 3 (2001), 185–197. 5, 11

[BSH12] BILODEAU B., SELLERS G., HILLESLAND K.: AMD GPU Technical Publications: Partially Resident Textures (PRT) in the Graphics Core Next, 2012. 13

[BSS00] BARTZ D., SCHNEIDER B.-O., SILVA C.: Rendering and Visualization in Parallel Environments. *SIGGRAPH '00 course notes* (2000). 2, 9

[BvG*09] BRUCKNER S., ŠOLTÉSZOVÁ V., GRÖLLER M. E., HLADUVKA J., BÜHLER K., YU J., DICKSON B.: BrainGazer - Visual Queries for Neurobiology Research. *IEEE Transactions on Visualization and Computer Graphics (Proc. of IEEE Visualization '09) 15*, 6 (Nov. 2009), 1497–1504. 8

[CBB*05] CHILDS H., BRUGGER E., BONNELL K., MEREDITH J., MILLER M., WHITLOCK B., MAX N.: A Contract-Based System For Large Data Visualization. In *IEEE Visualization '05* (2005), pp. 190–198. 11

[CCF94] CABRAL B., CAM N., FORAN J.: Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *IEEE Symposium on Volume Visualization* (1994), pp. 91–98. 7, 10, 25

[CKS03] CORREA W., KLOSOWSKI J. T., SILVA C.: Visibility-Based Prefetching for Interactive Out-Of-Core Rendering. In *IEEE Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 1–8. 8

[CMC*06] CASTANIE L., MION C., CAVIN X., LEVY B., BRUNO L., CASTANI L.: Distributed Shared Memory for Roaming Large Volumes. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (2006), 1299–1306. 11

[CN93] CULLIP T., NEUMANN U.: Accelerating Volume Reconstruction with 3D Texture Hardware. In *Technical Report TR93-027, University of North Carolina at Chapel Hill* (1993). 7, 10, 25

[CN09] CRASSIN C., NEYRET F.: Beyond Triangles : Gigavoxels Effects In Video Games. In *SIGGRAPH '09: Technical talk* (2009). 5, 14, 17, 18, 19

[CNLE09] 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* (2009), Lecture Notes in Computer Science, pp. 15–22. 2, 4, 5, 7, 10, 11, 14, 16, 25

[CNSE10] CRASSIN C., NEYRET F., SAINZ M., EISEMANN E.: Efficient Rendering of Highly Detailed Volumetric Scenes with GigaVoxels. In *GPU Pro*. A. K. Peters; Ltd, 2010, ch. X.3, pp. 643–676. 18, 19

[CPA*10] CHILDS H., PUGMIRE D., AHERN S., WHITLOCK B., HOWISON M., PRABHAT, WEBER G., BETHEL E.: Extreme Scaling of Production Visualization Software on Diverse Architectures. *IEEE Computer Graphics and Applications 30*, 3 (2010), 22–31. 4

[CSK*11] CONGOTE J., SEGURA A., KABONGO L., MORENO A., POSADA J., RUIZ O.: Interactive Visualization of Volumetric Data with WebGL in Real-Time. In *16th International Conference on 3D Web Technology - Web3D '11* (2011), pp. 137–146. 10

[DKR97] DERTHICK M., KOLOJEJCHICK J., ROTH S. F.: An Interactive Visual Query Environment for Exploring Data. In *Tenth Annual ACM Symposium on User Interface Software and Technology (UIST '97)* (1997), pp. 189–198. 7

[EHK*06] ENGEL K., HADWIGER M., KNISS J. M., REZK-SALAMA C., WEISKOPF D.: *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006. 2

[EMBM06] ENTEZARI A., MENG T., BERGNER S., MÖLLER T.: A Granular Three Dimensional Multiresolution Transform. In *Eurovis/IEEE-VGTC Symposium on Visualization '06* (2006), pp. 267–274. 16

[Eng11] ENGEL K.: CERA-TVR: A Framework for Interactive High-Quality Teravoxel Volume Visualization on Standard PCs. In *Large-Data Analysis and Visualization, (LDAV '11 Posters)* (2011). 2, 7, 9, 10, 14, 16, 17, 19, 25

[EPMS09] EILEMANN S., PAJAROLA R., MAKHINYA M., SOCIETY I. C.: Equalizer: A Scalable Parallel Rendering Framework. *IEEE Transactions on Visualization and Computer Graphics 15*, 3 (2009), 436–452. 11

[ESE00] ENGEL K., SOMMER O., ERTL T.: A Framework for Interactive Hardware Accelerated Remote 3D-Visualization. In *TCVG Symposium on Visualization (VisSym '00)* (2000), pp. 167–177. 10

[FCS*10] FOGAL T., CHILDS H., SHANKAR S., KRÜGER J., BERGERON R. D., HATCHER P.: Large Data Visualization on Distributed Memory Multi-GPU Clusters. In *High Performance Graphics* (2010), pp. 57–66. 5, 11

[FK05] FRANK S., KAUFMAN A.: Distributed Volume Rendering on a Visualization Cluster. In *Ninth International Conference on Computer Aided Design and Computer Graphics* (2005), pp. 5–10. 8

[FK10] FOGAL T., KRÜGER J.: Tuvok - An Architecture for Large Scale Volume Rendering. In *15th Vision, Modeling and Visualization Workshop '10* (2010), pp. 139–146. 5, 10, 25

[FS05] FOLEY T., SUGERMAN J.: KD-Tree Acceleration Structures for a GPU Raytracer. In *Graphics Hardware* (2005), pp. 15–22. 5, 14, 19

[FSK13] FOGAL T., SCHIEWE A., KRÜGER J.: An Analysis of Scalable GPU-Based Ray-Guided Volume Rendering. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV '13)* (2013), pp. 43–51. 2, 5, 6, 7, 10, 11, 17, 18, 19, 25

[FW08] FALK M., WEISKOPF D.: Output-Sensitive 3D Line Integral Convolution. *IEEE Transactions on Visualization and Computer Graphics 14*, 4 (2008), 820–834. 2

[GGSe*02] GUTHE S., GONSER J., STRASS ER W., WAND M., STRAER W.: Interactive Rendering of Large Volume Data Sets. In *IEEE Visualization* (2002), pp. 53–59. 6, 7, 10, 11, 16, 25

[GHSK03] GAO J., HUANG J., SHEN H.-W., KOHL J. A.: Visibility Culling Using Plenoptic Opacity Functions for Large Volume Visualization. In *IEEE Visualization '03* (2003), pp. 341–348. 18

[GKM93] GREENE N., KASS M., MILLER G.: Hierarchical Z-Buffer Visibility. In *SIGGRAPH '93* (1993), pp. 231–238. 2, 17

[GM05] GOBBETTI E., MARTON F.: Far Voxels: A Multiresolution Framework for Interactive Rendering of Huge Complex 3D Models on Commodity Graphics Platforms. *ACM Transactions on Graphics 24*, 3 (2005), 878–885. 5

[GMG08] GOBBETTI E., MARTON F., GUITI I.: A Single-Pass GPU Ray Casting Framework for Interactive Out-of-Core Rendering of Massive Volumetric Datasets. *The Visual Computer 24*, 7 (2008), 787–806. 5, 10, 14, 17, 18, 19, 25

[GS04] GUTHE S., STRASSER W.: Advanced Techniques for High-Quality Multi-Resolution Volume Rendering. *Computers & Graphics 28*, 1 (2004), 51–58. 6, 7, 10, 11, 16, 25

[GSHK04] GAO J., SHEN H.-W., HUANG J., KOHL J. A.: Visibility Culling for Time-Varying Volume Rendering Using Temporal Occlusion Coherence. In *IEEE Visualization '04* (2004), pp. 147–154. 18

[HBH03] HADWIGER M., BERGER C., HAUSER H.: High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware. In *IEEE Visualization '03* (2003), pp. 301–308. 7, 10, 25

[HBJP12] HADWIGER M., BEYER J., JEONG W.-K., PFISTER H.: Interactive Volume Exploration of Petascale Microscopy Data Streams Using a Visualization-Driven Virtual Memory Approach. *IEEE Transactions on Visualization and Computer Graphics (Proc. IEEE of SciVis '12) 18*, 12 (2012), 2285–2294. 2, 4, 5, 6, 7, 8, 10, 11, 13, 14, 15, 16, 17, 18, 19, 25

[HBZ98] HAVRAN V., BITTNER J., ZÁRA J.: Ray Tracing With Rope Trees. In *14th Spring Conference On Computer Graphics* (1998), pp. 130–139. 14

[Hec86] HECKBERT P.: Survey of Texture Mapping. *IEEE Computer Graphics and Applications 6*, 11 (1986), 56–67. 10

[Hel13] HELMSTAEDTER M.: Cellular-Resolution Connectomics: Challenges of Dense Neural Circuit Reconstruction. *Nature Methods 10*, 6 (June 2013), 501–7. 1

[HFK05] HONG W., FENG Q., KAUFMAN A.: GPU-Based Object-Order Ray-Casting for Large Datasets. In *Eurographics/IEEE VGTC Workshop on Volume Graphics '05* (2005), pp. 177–240. 7, 10, 11, 16, 25

[HL09] HUGHES D. M., LIM I. S.: Kd-Jump: A Path-Preserving Stackless Traversal for Faster Isosurface Raytracing on GPUs. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1555–1562. 5, 14

[HMG05] HASTINGS E. J., MESIT J., GUHA R. K.: Optimization of Large-Scale , Real-Time Simulations by Spatial Hashing. In *Summer Computer Simulation Conference* (2005), pp. 9–17. 16

[HN12] HEITZ E., NEYRET F.: Representing Appearance and Pre-Filtering Subpixel Data in Sparse Voxel Octrees. In *ACM SIGGRAPH / Eurographics conference on High-Performance Graphics (EGGH-HPG '12)* (2012), pp. 125–134. 5

[HP11] HENNESSEY J. L., PATTERSON D. A.: *Computer Architecture: A Quantitative Approach*, fifth ed. Morgan Kaufmann, 2011. 12, 15, 17

[HSB*12] HADWIGER M., SICAT R., BEYER J., KRÜGER J., MÖLLER T.: Sparse PDF Maps for Non-Linear Multi-Resolution Image Operations. In *ACM Transactions on Graphics (Proc. of ACM SIGGRAPH Asia '12)* (2012), pp. 198:1–198:12. 2, 6

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d Tree GPU Raytracing. In *Symposium on Interactive 3D Graphics and Games - I3D '07* (2007), p. 167. 5, 14

[HSSB05] HADWIGER M., SIGG C., SCHARSACH H., BÜHLER K.: Real-Time Ray-Casting and Advanced Shading of Discrete Isosurfaces. *Computer Graphics Forum (Proc. of Eurographics '05) 24*, 3 (2005), 303–312. 7, 10, 11, 13, 17, 25

[ILC10] ISENBURG M., LINDSTROM P., CHILDS H.: Parallel and Streaming Generation of Ghost Data for Structured Grids. *IEEE Computer Graphics & Applications 30*, 3 (2010), 32–44. 5, 8

[JBH*09] JEONG W.-K. W.-K., BEYER J., HADWIGER M., VASQUEZ A., PFISTER H., WHITAKER R. T., VAZQUEZ A.: Scalable and Interactive Segmentation and Visualization of Neural Processes in EM Datasets. *IEEE Transactions on Visualization and Computer Graphics (Proc. of IEEE Visualization '09) 15*, 6 (2009), 1505–1514. 7, 10, 11, 25

[JBH*10] JEONG W.-K. W.-K. J. W.-K., BEYER J., HADWIGER M., BLUE R., LAW C., VASQUEZ A., REID C., LICHTMAN J., PFISTER H., VAZQUEZ-REINA A., REID R. C.: SSECRETT and NeuroTrace: Interactive Visualization and Analysis Tools for Large-Scale Neuroscience Datasets. *IEEE Computer Graphics & Applications 30*, 3 (2010), 58–70. 14

[JJY*11] JEONG W.-K., JOHNSON M. K., YU I., KAUTZ J., PFISTER H., PARIS S.: Display-Aware Image Editing. In *IEEE International Conference on Computational Photography (ICCP '11)* (Apr. 2011), IEEE, pp. 1–8. 2

[JST*10] JEONG W.-K., SCHNEIDER J., TURNEY S. G., FAULKNER-JONES B. E., MEYER D., WESTERMANN R., REID C., LICHTMAN J., PFISTER H.: Interactive Histology of Large-Scale Biomedical Image Stacks. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 1386–1395. 2, 8

[KAL*11] KLASKY S., ABBASI H., LOGAN J., PARASHAR M., SCHWAN K., SHOSHANI A., WOLF M., SEAN A., ALTINTAS I., BETHEL W., LUIS C., CHANG C., CHEN J., CHILDS H., CUMMINGS J., DOCAN C., EISENHAUER G., ETHIER S., GROUT R., LAKSHMINARASIMHAN S., LIN Z., LIU Q., MA X., MORELAND K., PASCUCCI V., PODHORSZKI N., SAMATOVA N., SCHROEDER W., TCHOUA R., TIAN Y., VATSAVAI R., WU J., YU W., ZHENG F.: In Situ Data Processing for Extreme-Scale Computing. In *SciDAC Conference* (2011). 8

[KE02] KRAUS M., ERTL T.: Adaptive Texture Maps. In *Graphics Hardware* (2002), pp. 7–15. 11, 13

[KGB*09] KAINZ B., GRABNER M., BORNIK A., HAUSWIESNER S., MUEHL J., SCHMALSTIEG D.: Ray Casting of Multiple Volumetric Datasets with Polyhedral Boundaries on Manycore GPUs. *ACM Transactions on Graphics 28*, 5 (2009), 1–9. 10, 11, 25

[KH13] KEHRER J., HAUSER H.: Visualization and Visual Analysis of Multifaceted Scientific Data: A Survey. *IEEE Transactions on Visualization and Computer Graphics 19*, 3 (Mar. 2013), 495–513. 2

[KMS*06] KASIK D., MANOCHA D., STEPHENS A., BRUDERLIN B., SLUSALLEK P., GOBBETTI E., CORREA W., QUILEZ I.: Real Time Interactive Massive Model Visualization. *Eurographics '06: Tutorials* (2006). 2, 8

[Kno06] KNOLL A.: A Survey of Octree Volume Rendering Methods. In *First IRTG workshop* (2006). 5

[KTW*11] KNOLL A., THELEN S., WALD I., HANSEN C. D., HAGEN H., PAPKA M. E.: Full-Resolution Interactive CPU Volume Rendering with Coherent BVH Traversal. In *IEEE Pacific Visualization Symposium '11* (Mar. 2011), pp. 3–10. 11

[KW03] KRÜGER J., WESTERMANN R.: Acceleration Techniques for GPU-based Volume Rendering. In *IEEE Visualization '03* (2003), pp. 287–292. 10, 25

[LB03] LINDEBERG T., BRETZNER L.: *Real-Time Scale Selection in Hybrid Multi-Scale Representations*. Tech. rep., KTH (Royal Institute of Technology), 2003. 11

[LCD09] LIU B., CLAPWORTHY G. J., DONG F.: Accelerating Volume Raycasting using Proxy Spheres. *Computer Graphics Forum (Proc. of EuroVis '09) 28*, 3 (June 2009), 839–846. 11

[LHJ99] LAMAR E., HAMANN B., JOY K. I.: Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In *IEEE Visualization '99* (1999), pp. 355–362. 2, 5, 7, 10, 11, 16, 25

[Lju06a] LJUNG P.: Adaptive Sampling in Single Pass, GPU-based Raycasting of Multiresolution Volumes. In *Eurographics/IEEE VGTC Workshop on Volume Graphics '06* (2006), pp. 39–46. 5, 6, 10, 11, 25

[Lju06b] LJUNG P.: *Efficient Methods for Direct Volume Rendering of Large Data Sets*. PhD thesis, Linköping University, Sweden, 2006. 6, 11

[LK10a] LAINE S., KARRAS T.: Efficient Sparse Voxel Octrees. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '10)* (2010), pp. 55–63. 5

[LK10b] LAINE S., KARRAS T.: *Efficient Sparse Voxel Octrees - Analysis , Extensions , and Implementation*. Tech. rep., NVIDIA, 2010. 5

[LKHW04] LEFOHN A. E., KNISS J. M., HANSEN C. D., WHITAKER R. T.: A Streaming Narrow-Band Algorithm: Interactive Computation and Visualization of Level Sets. *IEEE Transactions on Visualization and Computer Graphics 10*, 4 (2004), 422–433. 8

[LMK03] LI W., MUELLER K., KAUFMAN A.: Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering. In *IEEE Visualization '03* (2003), pp. 317–324. 2, 10, 18, 25

[Mar13] MARX V.: Neurobiology: Brain mapping in high resolution. *Nature 503*, 7474 (Nov. 2013), 147–152. 1

[MAWM11] MOLONEY B., AMENT M., WEISKOPF D., MÖLLER T.: Sort-First Parallel Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics 17*, 8 (2011), 1164–1177. 11

[MCE*94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H., ANDN D. ELLSWORTH M. C.: A Sorting Classification of Parallel Rendering. *IEEE Computer Graphics & Applications 14*, 4 (1994), 23–32. 8, 9

[MHE01] MAGALLÓN M., HOPF M., ERTL T.: Parallel Volume Rendering Using PC Graphics Hardware. In *Pacific Conference on Computer Graphics and Applications* (2001), pp. 384–389. 11

[MHS08] MARSALEK L., HAUBER A., SLUSALLEK P.: High-Speed Volume Ray Casting with CUDA. In *IEEE Symposium on Interactive Ray Tracing* (Aug. 2008), p. 185. 10, 11, 25

[ML13] MORGAN J. L., LICHTMAN J. W.: Why Not Connectomics? *Nature Methods 10*, 6 (June 2013), 494–500. 1

[MM10] MARCHESIN S. S., MA K.-L.: Cross-Node Occlusion in Sort-Last Volume Rendering. In *Eurographics Symposium on Parallel Graphics and Visualization* (2010), pp. 11–18. 18

[MOM*11] MORELAND K., OLDFIELD R., MARION P., JOURDAIN S., PODHORSZKI N., VISHWANATH V., FABIAN N., DOCAN C., PARASHAR M., HERELD M., PAPKA M. E., KLASKY S.: Examples of In Transit Visualization. In *Second International Workshop on Petascale Data Analytics: Challenges and Opportunities (PDAC '11)* (2011), pp. 1–6. 8

[Mor66] MORTON G. M.: *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. Tech. rep., IBM Ltd., 1966. 6

[Mor12] MORELAND K.: Oh, $#*@! Exascale! The Effect of Emerging Architectures on Scientific Discovery. *2012 SC Companion: High Performance Computing, Networking Storage and Analysis* (2012), 224–231. 6, 8

[Mor13] MORELAND K.: A Survey of Visualization Pipelines. *IEEE Transactions on Visualization and Computer Graphics (Proc. of IEEE SciVis '13) 19*, 3 (Mar. 2013), 367–78. 3

[MPHK94] MA K.-L., PAINTER J., HANSEN C., KROGH M.: Parallel volume rendering using binary-swap compositing. *IEEE Computer Graphics & Applications 14*, 4 (July 1994), 59–68. 8

[MRH08] MENSMANN J., ROPINSKI T., HINRICHS K.: Accelerating Volume Raycasting using Occlusion Frustums. In *Fifth EG/IEEE Conference on Point-Based Graphics* (2008), pp. 147–154. 11

[MRH10] MENSMANN J., ROPINSKI T., HINRICHS K. H.: An Advanced Volume Raycasting Technique using GPU Stream Processing. In *International Conference on Computer Graphics Theory and Applications (GRAPP '10)* (Angers, 2010), INSTICC Press, pp. 190–198. 10, 11, 25

[MSE06] MÜLLER C., STRENGERT M., ERTL T.: Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems. In *Eurographics Symposium on Parallel Graphics and Visualization* (2006), pp. 59–66. 11

[Mur93] MURAKI S.: Volume Data and Wavelet Transforms. *IEEE Computer Graphics and Applications 13*, 4 (1993), 50–56. 6

[Mus13] MUSETH K.: VDB: High-Resolution Sparse Volumes with Dynamic Topology. *ACM Transactions on Graphics 32*, 3 (2013), 27:1–27:22. 5

[MW95] MARTIN A. R., WARD M. O.: High Dimensional Brushing for Interactive Exploration of Multivariate Data. In *IEEE Visualization '95* (1995), pp. 271–278. 7

[MWMS07] MOLONEY B., WEISKOPF D., MÖLLER T., STRENGERT M.: Scalable Sort-First Parallel Direct Volume Rendering with Dynamic Load Balancing. In *Eurographics Symposium on Parallel Graphics and Visualization* (2007), pp. 45–52. 11

[MWY*09] MA K.-L., WANG C., YU H., MORELAND K., HUANG J., ROSS R.: Next-Generation Visualization Technologies: Enabling Discoveries at Extreme Scale. In *SciDAC Review* (2009), pp. 12–21. 1, 6, 8

[Neu94] NEUMANN U.: Communication Costs for Parallel Volume-Rendering Algorithms. *IEEE Computer Graphics & Applications 14*, 4 (July 1994), 49–58. 9

[NVI13] NVIDIA CORPORATION: CUDA C Programming Guide, 2013. http://www.nvidia.com/object/cuda_get.html. 9

[NZIS13] NIESSNER M., ZOLLHÖFER M., IZADI S., STAMMINGER M.: Real-Time 3D Reconstruction at Scale Using Voxel Hashing. *ACM Transactions on Graphics 32*, 6 (2013), 1–11. 16

[OLG*07] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T. J., KR J.: A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum 26*, 1 (2007), 80–113. 9

[OVS12] OBERT J., VAN WAVEREN J., SELLERS G.: Virtual Texturing in Software and Hardware. In *SIGGRAPH '12 Courses* (2012). 6, 13

[PF02] PASCUCCI V., FRANK R. J.: Hierarchical Indexing for Out-of-Core Access to Multi-Resolution Data. In *Hierarchical and Geometrical Methods in Scientific Visualization*. 2002, pp. 225–241. 6, 8

[PGR*09] PETERKA T., GOODELL D., ROSS R., SHEN H.-W. H.-W., THAKUR R.: A configurable algorithm for parallel image-compositing applications. In *Conference on High Performance Computing Networking, Storage and Analysis (SC '09)* (New York, NY, USA, 2009), ACM, pp. 1–10. 8

[PGS*07] POPOV S., GÜNTHER J., SEIDEL H.-P. H.-P., SLUSALLEK P., GÜNTHER J.: Stackless Kd-Tree Traversal for High Performance GPU Ray Tracing. *Eurographics 26*, 3 (2007), 415–424. 5, 14

[PHKH04] PROHASKA S., HUTANU A., KAHLER R., HEGE H.-C.: Interactive Exploration of Large Remote Micro-CT Scans. In *IEEE Visualization* (2004), pp. 345–352. 10, 11, 25

[PJ95] PARKER S. G., JOHNSON C. R.: SCIRun : A Scientific Programming Environment for Computational Steering. In *ACM/IEEE conference on Supercomputing '95* (1995). 8

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.: Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization '98* (1998), pp. 233–238. 17

[RÖ9] RÖMISCH K.: *Sparse Voxel Octree Ray Tracing on the GPU*. PhD thesis, Aarhus University, 2009. 5

[RGG*14] RODRÍGUEZ M., GOBBETTI E., GUITAN J., MAKHINYA M., MARTON F., PAJAROLA R., SUTER S.: State-of-the-Art in Compressed GPU-Based Direct Volume Rendering. *Computer Graphics Forum 33*, 6 (2014), 77–100. 2, 6

[RGW*03] ROETTGER S., GUTHE S., WEISKOPF D., ERTL T., STRASSER W.: Smart Hardware-Accelerated Volume Rendering. In *Symposium on Visualization (VISSYM '03)* (2003), pp. 231–238. 10, 25

[Ros06] ROST R. J.: *OpenGL Shading Language (2nd Edition)*. Addison-Wesley Professional, 2006. 9

[RSEB*00] REZK-SALAMA C., ENGEL K., BAUER M., GREINER G., ERTL T.: Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware* (2000), pp. 109–118. 7, 10, 25

[RTW13] REICHL F., TREIB M., WESTERMANN R.: Visualization of Big SPH Simulations via Compressed Octree Grids. In *IEEE Big Data* (2013), pp. 71–78. 5, 10, 14, 19, 25

[RV06] RUIJTERS D., VILANOVA A.: Optimizing GPU Volume Rendering. In *Winter School of Computer Graphics (WSCG '06)* (2006), pp. 9–16. 10, 25

[SBH*08] SAMATOVA N. F., BREIMYER P., HENDRIX W., SCHMIDT M. C., RHYNE T.-M.: An Outlook Into Ultra-Scale Visualization of Large-Scale Biological Data. In *Workshop on Ultrascale Visualization, UltraVis 2008.* (2008), pp. 29–39. 6

[SBVB14] SOLTESZOVA V., BIRKELAND A., VIOLA I., BRUCKNER S.: Visibility-driven processing of streaming volume data. In *Proc. of VCBM 2014* (2014), pp. 127–136. 7

[SCC*02] SILVA C., CHIANG Y.-J., CORREA W., EL-SANA J., LINDSTROM P.: Out-of-Core Algorithms for Scientific Visualization and Computer Graphics. In *IEEE Visualization '02 Course Notes* (2002). 8

[Shn94] SHNEIDERMAN B.: Dynamic Queries for Visual Information Seeking. *IEEE Software 11*, 6 (1994), 70–77. 7

[SHN*06] SCHARSACH H., HADWIGER M., NEUBAUER A., WOLFSBERGER S., BÜHLER K.: Perspective Isosurface and Direct Volume Rendering for Virtual Endoscopy Applications. In *Eurovis/IEEE-VGTC Symposium on Visualization* (2006), pp. 315–323. 11, 13, 17

[SKMH14] SICAT R., KRÜGER J., MÖLLER T., HADWIGER M.: Sparse PDF Volumes for Consistent Multi-Resolution Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics (Proc. of IEEE SciVis '14) 20*, 12 (2014), in print. 6

[SO92] SHARIR M., OVERMARS M. H.: A Simple Output-sensitive Algorithm for Hidden Surface Removal. *ACM Trans. Graph. 11*, 1 (1992), 1–11. 2, 4

[SSJ*11] SUMMA B., SCORZELLI G., JIANG M., BREMER P.-T., PASCUCCI V.: Interactive Editing of Massive Imagery Made Simple. *ACM Transactions on Graphics 30*, 2 (Apr. 2011), 1–13. 6, 8

[SSKE05] STEGMAIER S., STRENGERT M., KLEIN T., ERTL T.: A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting. *Eurographics/IEEE VGTC Workshop on Volume Graphics '05* (2005), 187–195. 7, 10, 11, 25

[SSWB05] STOCKINGER K., SHALF J., WU K., BETHEL E. W.: Query-Driven Visualization of Large Data Sets. In *IEEE Visualization '05* (2005), pp. 167–174. 7

[THM01] TURLINGTON J. Z., HIGGINS W. E., MEMBER S.: New Techniques for Efficient Sliding Thin-Slab Volume Visualization. *IEEE Transactions on Medical Imaging 20*, 8 (2001), 823–835. 11

[TMJ98] TANNER C. C., MIGDAL C. J., JONES M. T.: The Clipmap : A Virtual Mipmap. In *SIGGRAPH '98* (1998), ACM, pp. 151–158. 5, 6

[TTRU*06] TU T., TABORDA-RIOS R., URBANIC J., YU H., BIELAK J., GHATTAS O., LOPEZ J. C., MA K.-L., O'HALLARON D. R., RAMIREZ-GUZMAN L., STONE N.: Analytics Challenge - Remote Runtime Steering of Integrated Terascale Simulation and Visualization. In *ACM/IEEE conference on Supercomputing (SC '06)* (2006), ACM Press, p. 297. 8

[TYC*11] TIKHONOVA A., YU H., CORREA C. D., CHEN J. H., MA K.-L.: A Preview and Exploratory Technique for Large-Scale Scientific Simulations. In *Eurographics Conference on Parallel Graphics and Visualization (EGPGV'11)* (2011), pp. 111–120. 8

[VOS*10] VO H. T., OSMARI D. K., SUMMA B., COMBA J. A. L. D., PASCUCCI V., SILVA C. T.: Streaming-Enabled Parallel Dataflow Architecture for Multicore Systems. *Computer Graphics Forum 29*, 3 (2010), 1073–1082. 8

[vW09] VAN WAVEREN J. M. P.: id Tech 5 Challenges: From Texture Virtualization to Massive Parallelization. Talk in Beyond Programmable Shading course, SIGGRAPH '09, 2009. 6, 13

[WE98] WESTERMANN R., ERTL T.: Efficiently Using Graphics Hardware in Volume Rendering Applications. In *SIGGRAPH '98* (1998), pp. 169–178. 7, 10, 25

[Wes94] WESTERMANN R.: A multiresolution framework for volume rendering. In *Proceedings of Symposium on Volume Visualization* (1994), pp. 51–58. 6

[WGL*05] WANG C., GAO J., LI L., SHEN W.-W., SHEN H.-W.: A Multiresolution Volume Rendering Framework for Large-Scale Time-Varying Data Visualization. In *Eurographics/IEEE VGTC Workshop on Volume Graphics '05* (2005), pp. 11–223. 11

[Wil83] WILLIAMS L.: Pyramidal Parametrics. *Computer Graphics (Proc. of SIGGRAPH '83) 17*, 3 (1983), 1–11. 6, 11

[Wit98] WITTENBRINK C. M.: *Survey of Parallel Volume Rendering Algorithms*. Tech. rep., Hewlett-Packard Laboratories, 1998. 2, 9

[WWH*00] WEILER M., WESTERMANN R., HANSEN C., ZIMMERMAN K., ERTL T.: Level-Of-Detail Volume Rendering via 3D Textures. In *IEEE Symposium on Volume Visualization* (2000), pp. 7–13. 2, 4, 5, 7, 10, 11, 16, 25

[YMC06] YOUNESY H., MÖLLER T., CARR H.: Improving the Quality of Multi-Resolution Volume Rendering. In *Eurovis/IEEE-VGTC Symposium on Visualization '06* (2006), pp. 251–258. 6

[YWG*10] YU H., WANG C., GROUT R. W., CHEN J. H., MA K.-L.: In Situ Visualization for Large-Scale Combustion Simulations. *IEEE Computer Graphics & Applications 30*, 3 (2010), 45–57. 8

[YWM08] YU H., WANG C., MA K.-L.: Massively parallel volume rendering using 2–3 swap image compositing. In *ACM SIGGRAPH ASIA 2008 courses on - SIGGRAPH Asia '08* (New York, New York, USA, 2008), ACM Press, pp. 1–11. 8

[ZMHH97] ZHANG H., MANOCHA D., HUDSON T., HOFF K. E.: Visibility Culling Using Hierarchical Occlusion Maps. In *ACM SIGGRAPH '97* (1997), pp. 77–88. 2, 17

[ZSJ*05] ZHANG J., SUN J., JIN Z., ZHANG Y., ZHAI W., ZHAI Q.: Survey of Parallel and Distributed Volume Rendering: Revisited. In *International Conference on Computational Science and Its Applications (ICCSA '05)* (2005), vol. 3, pp. 435–444. 2, 9

Technique	volume data representation				rendering			address translation				working set determination		
	non-bricked	single-res bricked	multi-res (tree)	multi-res (grid)	texture slicing	single-pass RC	multi-pass RC	virt. tex.	tree traversal	multi-res	GPU	no culling	global/ view cull	ray-guided
[CN93]	●				●							●		
[CCF94]	●				●							●		
[WE98]	●				●							●		
[LHJ99]			octree				●		●	●			●	
[RSEB*00]	●				●							●		
[WWH*00]			octree				●		●	●			●	
[GGSe*02]			octree				●		●	●			●	
[HBH03]	●				●							●		
[KW03]	●						●					●		
[LMK03]	●				●								●	
[RGW*03]	●						●					●		
[GS04]			octree				●		●	●			●	
[PHKH04]			octree				●		●	●			●	
[BG05]	●					●						●		
[HFK05]			octree				●		●	●			●	
[HSSB05]		●				●		●			●		●	
[SSKE05]	●					●						●		
[Lju06a]				●		●		●		●	●		●	
[RV06]		●	○†				●		●	●			●	
[BHWB07]		●				●		●			●		●	
[BHMF08]				●		●		●		○	●		●	
[GMG08]			octree			●			●	●	●		∗	○†††
[MHS08]	●					●						●		
[CNLE09]			octree			●			●	●	●		∗	●
[JBH*09]				●		●		●		●	●		●	
[KGB*09]	●					●							●	
[FK10]			kd-tree				●		●	●	●		●	
[MRH10]	●					●						●		
[Eng11]			octree			●			●	●	●		∗	●
[HBJP12]				●		●		●		●	●		∗	●
[BAaK*13]				●		●		●		●	●		∗	●
[FSK13]				●		●	○††	●		●	●		∗	●
[RTW13]			octree			●			●	●	●		∗	●

●: full support; ○: partial support; ∗: implicit support.

Table 3: ***Comparison of GPU-based volume visualization techniques*** *based on data representation, rendering, address translation and working set determination.* † *uses an octree per brick,* †† *can fall back to multi-pass raycasting if the current working set is too large to fit into GPU memory.* ††† *is not fully ray-guided, but utilizes interleaved occlusion queries with similar goals.*