DOI: 10.1111/cgf.14857

EUROVIS 2023
S. Bruckner, R. Raidou, and C. Turkay
(Guest Editors)

COMPUTER GRAPHICS *forum*
Volume 42 (2023), Number 3
STAR – State of The Art Report

State-of-the-art in Large-Scale Volume Visualization Beyond Structured Data

J. Sarton[1], S. Zellmann[2], S. Demirci[3], U. Güdükbay[3], W. Alexandre-Barff[4],
L. Lucas[4], J.M. Dischler[1], S. Wesner[2], I. Wald[5]

[1]University of Strasbourg, France [2]University of Cologne, Germany [3]Bilkent University, Ankara, Turkey
[4]University of Reims Champagne-Ardenne, France [5]NVIDIA

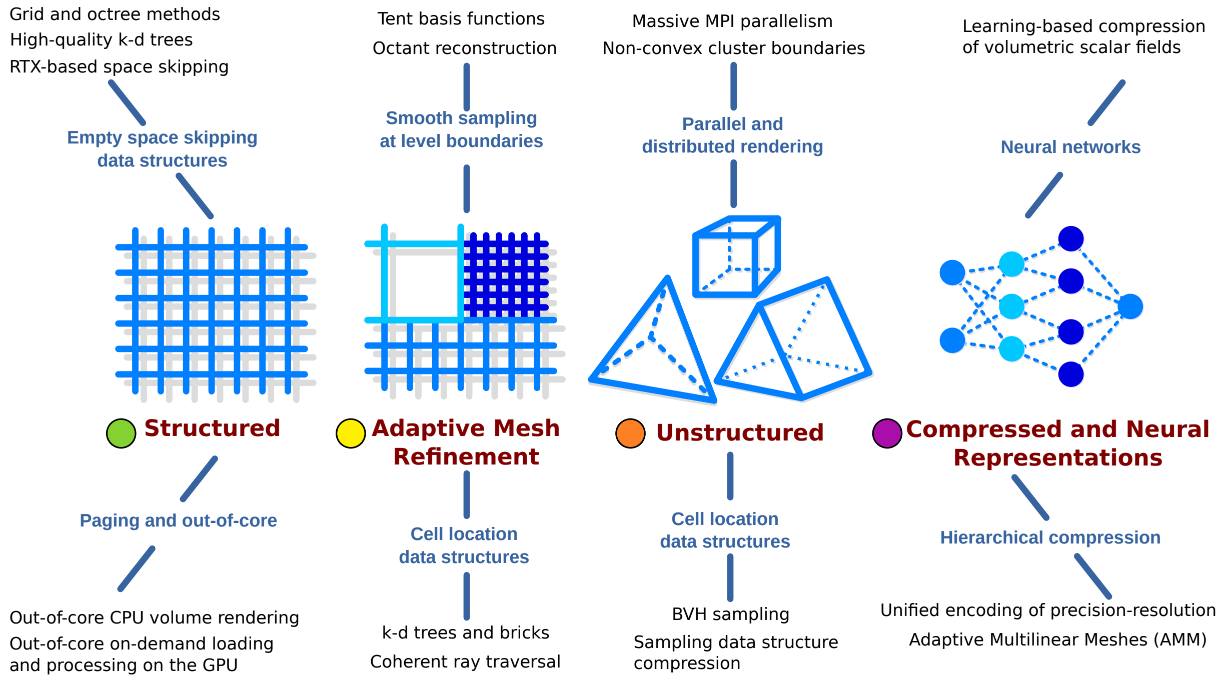
Grid and octree methods
High-quality k-d trees
RTX-based space skipping
Empty space skipping data structures
Tent basis functions
Octant reconstruction
Smooth sampling at level boundaries
Massive MPI parallelism
Non-convex cluster boundaries
Parallel and distributed rendering
Learning-based compression of volumetric scalar fields
Neural networks
Structured
Adaptive Mesh Refinement
Unstructured
Compressed and Neural Representations
Paging and out-of-core
Out-of-core CPU volume rendering
Out-of-core on-demand loading and processing on the GPU
Cell location data structures
k-d trees and bricks
Coherent ray traversal
Cell location data structures
BVH sampling
Sampling data structure compression
Hierarchical compression
Unified encoding of precision-resolution
Adaptive Multilinear Meshes (AMM)

Figure 1: *In this state-of-the-art report, we evaluate current research papers on interactive volume visualization. Nowadays, volume visualization is primarily data-driven, and how the various methods, frameworks, and tools cope with that data largely depends on the topology. This figure shows the structure we follow in this paper. We focus on different data representations (center). For each of them, we discuss different approaches of large-scale visualization (blue captions) and the different techniques used to implement them (black captions).*

Abstract

Volume data these days is usually massive in terms of its topology, multiple fields, or temporal component. With the gap between compute and memory performance widening, the memory subsystem becomes the primary bottleneck for scientific volume visualization. Simple, structured, regular representations are often infeasible because the buses and interconnects involved need to accommodate the data required for interactive rendering. In this state-of-the-art report, we review works focusing on large-scale volume rendering beyond those typical structured and regular grid representations. We focus primarily on hierarchical and adaptive mesh refinement representations, unstructured meshes, and compressed representations that gained recent popularity. We review works that approach this kind of data using strategies such as out-of-core rendering, massive parallelism, and other strategies to cope with the sheer size of the ever-increasing volume of data produced by today's supercomputers and acquisition devices. We emphasize the data management side of large-scale volume rendering systems and also include a review of tools that support the various volume data types discussed.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Rendering; Volumetric models; Ray tracing; Graphics processors; Massively parallel algorithms; Distributed algorithms;*** • ***Human-centered computing*** $\rightarrow$ ***Visualization toolkits; Scientific visualization;***

1. Introduction

Direct volume rendering has historically been associated with regular grids obtained from numerical simulation or acquisition techniques grounded in medical imaging, microscopy, and others. An inherent problem with volumetric data is its size, which can easily become hard to handle on commodity systems. Volume rendering can be quite resource intensive, particularly on the processor's memory subsystem. In the early 2000s, the introduction of GPUs with dedicated texture units boosted this performance-hungry technique, so the representation using regular grids and better-quality interpolation became very popular. Regular grids can, however, only scale to a certain degree because a doubling in resolution per spatial dimension causes the data to grow by a factor of eight. Hence, intricate techniques have been proposed to accommodate and efficiently render such data sets on GPUs; these techniques have been discussed in much detail by Beyer *et al.* [BHP15].

At the same time, in the age of ExaScale computing, it is not only the rendering algorithms that are challenged by increasing data size. Generally, memory throughput has not kept up with compute performance, which still roughly follows Moore's law. This can also be observed in simulation codes—the more popular ones (including Fun3D [Nat20], FLASH [DAC*14], Chombo [CGL*00], AMReX [AMR18], or Lava [KHB*16]) nowadays use *adaptive mesh refinement* (AMR) [BC89] or unstructured grids [ST90] to overcome these issues. These irregular data representations have become prevalent and also need to be dealt with by visualization software. At the same time, even these data representations tend to be huge with possibly time-varying, multi-variate datasets.

Regardless of the considered volume data topology, visualization techniques must incorporate solutions to cope with the size of the input data and the auxiliary data structures needed to render them. These solutions, based on data streaming, use of massive parallelism or in-situ visualization techniques, depend on the targeted hardware in terms of computing resources and memory level. On the other hand, no matter what solutions are implemented to distribute and access the data, the visualization algorithms themselves must address the quantity and/or complexity of the data. A rigorous survey has not yet explored these aspects, and our paper aims to fill this gap.

Another trend we observe is the execution model used by the rendering subsystems of modern visualization systems, such as OSPRay [WJA*17] or VisRTX [Ams19], which uses general-purpose compute and ray tracing in favor of rasterization. While rasterization APIs allow for ray tracing inside a fragment shader, general purpose compute—on either the GPU or the CPU using `ispc` [PM12]—enables the use of ray tracing APIs with optimized bounding volume hierarchy (BVH) traversal [WWB*14, NVI09b]. This trend has not only led to different types of algorithms used in terms of how the volume data is traversed and sampled but also allowed for a more flexible control flow overall, including incoherent ray workloads and Monte Carlo rendering. Where applicable, our state-of-the-art report will emphasize ray tracing-centric execution models and hardware-accelerated ray tracing.

Data Type/Repr.	Key Papers	Other
Structured ●	[SCRL20] [WWJ19]	[HAAB*18] [ZSL21] [WZM21]
AMR ●	[WBUK17] [WWW*19] [WZU*21]	[ZSM*22] [ZWS*22a]
Unstructured ●	[BPL*19] [WMZ21] [MWUP22]	[MUWP19] [SDM*21]
Compressed & Neural ●	[LJLB21] [HSB*21] [BHM*22]	[PZGY19] [QCJJ18] [WN21]

Table 1: *Papers discussed in this state-of-the-art report.*

The main contributions of our state-of-the-art report are

- an extensive review of volume visualization research in recent years that includes structured *and* unstructured volume data,
- a review of volume rendering techniques that use general purpose and hardware ray tracing APIs, and
- an engineering-focused review that also includes tools and how they support the kind of algorithms and data management routines discussed in this paper.

In this sense, we hope that our survey is beneficial not only to the scientific visualization community, which continuously advances state of the art in this field but also to practitioners who use the algorithms discussed to create high-quality visualizations and potentially get a deeper understanding of the performance of the visualization system.

1.1. Survey scope

This state-of-the-art report reviews and categorizes recent research on *direct* volume rendering techniques. Following recent trends, a major focus is placed on ray tracing techniques. While we will discuss in detail the different sampling and traversal methods of visualization algorithms that can be found in in-core or out-of-core, single or multi CPU/GPU approaches, we will not discuss in-situ visualization solutions that rely on data that will not even touch the persistent storage. A survey on volume rendering of structured data *complements* that by Beyer *et al.* [BHP15] with CPU-based methods and with an overview of empty space skipping and out-of-core techniques that emerged later on. In addition to this, we focus on unstructured, adaptive mesh refinement (AMR), compressed and neural or similar topologies beyond structured volume data. The influential papers we review are shown in Table 1. In addition to algorithmic aspects, we also emphasize system approaches and tools like OSPRay [WJA*17]. Ray-tracing hardware extensions that became widely available in 2018 have since played a significant role in scientific visualization [WUM*19, WZU*21, MSG*22, ZWB*22] and will be another focus point for us.

Data: In this state-of-the-art report, we review methods that handle volumetric data that can come from high-resolution acquisitions or large-scale simulations. This implies data whose nature can vary from a structured grid with a single time step and a single scalar field to more complex grids that evolve in time and can store several scalar and/or vector fields. For the approaches that manipulate structured grids, these are fixed, contain only one scalar field, and can reach 10^{12} regular cells for an input size of up to 1.5 TB. For AMR grids, the approaches discussed handle time-varying meshes up to several hundred million cells with several scalar fields at each time step for input data up to 4 TB. For unstructured data, it is about meshes that can contain up to a few billion elements for a size of a few GB. It is important to note that these data can be categorized as large-scale either by their initial representation (input data) or because their efficient interactive visualization requires data structures and/or sampling methods that may require a memory footprint of several orders of magnitude larger than the data itself.

Hardware: We discuss methods that focus on interactive visualization of large volume data on GPU or CPU. For the first category, the presented methods target workstations equipped with one or multiple GPUs. We do not focus on a specific vendor in particular, but note that especially when it comes to hardware-accelerated ray tracing, the literature is dominated by papers that use NVIDIA GPUs and the OptiX API. On the CPU side, typical target systems are x86 multi-core workstations as well as cluster systems with on the order of tens to hundreds of x86 CPU nodes.

1.2. Survey structure

The state-of-the-art report follows a data-driven structure. Fig. 1 shows this organization according to the addressed data types. We broadly categorize by data type into structured, regular volume data (🟢), AMR data (🟡), unstructured meshes (🟠), and data that focuses on compression, including mixed resolution-precision trees and neural network compression (🟣). The use of neural networks in sci-vis is a relatively new field. A detailed survey was recently presented by Wang *et al.* [WHss], so here we only briefly describe basic concepts and pointers to papers presenting recent developments.

We distinguish methods that focus on *traversal* (e.g., empty space skipping or space partitions assigned to cluster nodes) and on *sampling* (e.g., cell location in AMR volumes, sample point reconstruction in the presence of virtual page tables) (cf. Section 2). The papers we review can also be broadly categorized as in-core running on single CPU or GPU systems, as out-of-core using paging between the file system, the CPU, and (optionally) the GPU, and as massively parallel. The main sections are structured by the topology of the data, while the sections themselves follow the latter methodological categories. Tools focusing on volume rendering are discussed in a separate section.

2. Background and Terminology

In our survey, we concentrate primarily on the data handling aspects of sci-vis volume rendering. Scientific visualization software has to cope with bringing the data into a form amenable to rendering on the target hardware. Rendering often amounts to a fraction of the overall processing; still, the rendering algorithm determines the eventual layout necessary to visualize the data efficiently. This section introduces terminology common to the methods we summarize in this survey.

2.1. Direct volume rendering with ray traversal

We focus on *direct* volume rendering algorithms, i.e., algorithms that do not a priori extract level sets but sample and classify the data directly. Typical integration algorithms are *direct volume rendering* (DVR) with absorption and emission [Max95], *implicit isosurface rendering*, and *free-flight distance sampling* (here exemplified via delta tracking [WMHL65]) that we present in the Appendix and that help us to introduce some terminology (cf. Appendix A). These algorithms compute some quantities for volumetric domains, such as absorption and emission, iso-surfaces, and transmission coefficients for *volumetric domains*.

In recent years, we have observed a gradual shift towards volumetric path tracing, also in the scientific visualization community [KPB12, WJA*17, MHK*19, HMES20, IGMM22, MSG*22, ZWB*22, XTC*ss, ZWS*22b]. Instead of stepwise sampling the domain $[t_{min}, t_{max}]$ using ray marching, volumetric path tracers compute free-flight distances inside the domain.

Free-flight distances represent the probability with which a photon travels along the ray (o, ω) without colliding with a particle from the volume density; they are computed using tracking algorithms [WMHL65]. State-of-the-art methods use *fictitious particles* or other *control variates* [GMH*19] to *homogenize* the usually inhomogeneous volume; this requires one to store the maximum ("majorant") extinction $\bar{\mu}$ along with the domain $[t_{min}, t_{max}]$. The result is not an alpha-composited color but a collision-free distance, and sometimes, the *transmittance* (i.e., the stochastic amount of light passing through the domain $[t_{min}, t_{max}]$). Extinction coefficients (often defined as the alpha component obtained from classification) and phase function albedo (RGB component from classification) can then be sampled for that distance by the rendering algorithm as required.

Noise-free images usually result by accumulating *convergence frames*; each is typically computed by tracing one path per pixel. In contrast to ray marching, only one or few samples are taken from $[t_{min}, t_{max}]$; the number of samples usually correlates with the difference between the majorant and the actual sampled density. The majorants can hence be used to build space-skipping data structures. An appealing property of free-flight distance sampling is its unbiasedness. This property is also desirable for sci-vis since it helps reduce artifacts that can otherwise cause aliasing and obfuscate important features. Free-flight distance sampling is a technique orthogonal to the light transport algorithm (e.g., absorption plus emission, single scattering, multi scattering) used, and with spatially varying majorants, it naturally supports adaptive sampling. Regardless of the different execution flow and quantities computed, the three algorithms have two major operations in common:

Sampling: The functions $\mu_{src}(x)$ and $C_{src}(x)$ sample the volume at positions $x \in \mathbb{R}^3$ to retrieve the local extinction $\mu(x)$, or an RGBα color $(C_{src}(x))$. In scientific visualization, this operation comprises a lookup into the volume density implemented, e.g., using simple trilinear interpolation or hierarchy traversal/cell location for more complex volume data, followed by *(post)-classification* using an alpha transfer function to obtain an alpha/extinction value or an RGBα tuple. We refer to this whole operation (volume lookup *and* classification) as *sampling*.

Traversal: Obtaining the domains $[t_{min}, t_{max}]$ usually involves some type of data structure. In the simplest case, the domain is obtained as the intersection of the ray with the bounding box of the volume; the traversal operation is then trivial. More complex volume renderers use a space-skipping data structure that is traversed using DDA [SKTM11, DHTI21] or other hierarchical data structures such as a k-d tree [ZSL21] or bounding volume hierarchy (BVH) [ZHL19]. In the case of tracking, the domains are not only given by their interval $[t_{min}, t_{max}]$ but also by their majorant extinction $\bar{\mu}$. $\bar{\mu}$ presents an upper bound (and is often just the maximum value) of the extinction in the region that is traversed. However, in the presence of post-classification, the majorant extinction is, in general, not the maximum value of the volume inside the domain, but the maximum *post-classified* value.

The methods we explore in the following all cater to this common structure of first traversing domains in space and, inside these domains, compute quantities such as color, local extinction, and free-flight distances. The algorithms serve as intrinsic building blocks, and the visualization systems we explore usually employ sophisticated methods to optimize the surrounding data structures. Some of the data structures involved may be trivial, or it might even be possible to use the same data structure for sampling and traversal. One of the essential topics of this paper is how this can be done for different volume data types.

2.2. Ray tracing APIs and terminology

We provide a brief review of hardware ray tracing execution models, since this emerging technology is relevant to many of the papers discussed here. For a general ray tracing review, see the survey article by Meister *et al.* [MOB*21]. The execution model is implemented by NVIDIA OptiX [PBD*10, NVI09b], by Microsoft DXR [Mic23], and by the Khronos Vulkan Ray Tracing extensions [Koc20]. In this execution model, *ray tracing modules* are launched on a GPU with hardware ray tracing cores (RT cores). The entry point to the ray tracing module is called the *ray generation program* (RG), which is executed on the shader cores of the GPU and shares commonalities with ordinary compute kernels. RG instances are executed in parallel, according to the number of rays scheduled by the *launch configuration* (often one ray/thread per image sample). RG-generated rays can be traced into a BVH using an intrinsic function call. The execution then switches from the shader to the RT core, incurring a context switch.

Depending on the vendor, both ray/triangle intersections and BVH traversal are performed in hardware.The user can intercept execution with software programs, e.g., on closest intersection (*closest hit program* (CH)), on any intersection be it the closest or not (*any hit program* (AH)), or when the ray misses the scene (*miss program* (MI)). Some architectures support custom user primitives that are implemented via *bounds* and *intersection* (IS) programs.

Ray tracing hardware extensions were first implemented by NVIDIA, but the other vendors are slowly progressing in the same direction [Adv23, Lar22]. Hence, how high-performance volume renderers are implemented nowadays has changed in light of these new APIs. On the CPU side (and other Intel hardware), an important API is Intel's Embree [WWB*14]. While the execution model here is more flexible, and the comments about software and hardware context switches do not generally apply, the APIs share basic concepts such as BVH traversal, user geometry, and any hit callback functions.

3. Rendering of Structured Volume Data ●

Volume rendering of structured data where the volume is defined on a Cartesian grid and all cells are implicitly represented in memory has been extensively evaluated in Beyer *et al.*'s [BHP15] state-of-the-art report. Still, the field has advanced here as well, and therefore we briefly review topics that have been the focus of research papers published after Beyer's survey.

With the gap between compute and memory performance widening, we generally observe that research focuses more on hierarchical volume data (or, on *making* volume data hierarchical to suit modern compute architectures well) or on other representations with lower storage demand than structured grids. The advancements that we *do* observe regarding this simplest topology after Beyer's state-of-the-art report was published primarily focus on empty space skipping and paging and streaming techniques to accommodate volumes that exceed the available compute node's memory (e.g., cluster node, GPU with pageable host memory). Here we concentrate on an assortment of papers that we believe *complements* Beyer's state-of-the-art report and direct the reader to the latter for an extensive review.

3.1. Empty space-skipping data structures

Empty space skipping identifies coherent regions where the volume is truly empty. Depending on the context, it is sometimes desirable to skip over empty and over *homogeneous* space where all samples have the same value. The latter is essential for volumetric path tracing, where domains are considered homogeneous with roughly the same majorant. Hence, the number of samples the rejection sampling loop takes in Algorithm 3 (see Appendix A) can be reduced [YIC*10]. In contrast, true *empty* space skipping focuses on those regions of space where all the samples have an extinction/alpha value of 0. This is more relevant in the cases of Algorithms 1 and 2 (see Appendix A) that do not adapt the sampling rate to the majorant extinction.

The rendering algorithm will use space-skipping data structures to obtain domains $[t_{min}, t_{max}]$, i.e., according to our terminology, they are *traversal* data structures. *Sampling* is typically simpler in the case of structured volumes. It is required to locate a neighborhood of samples for the sampling position $x \in \mathbb{R}^3$ to perform the interpolation. Since the underlying Cartesian grid gives us an implicit indexing scheme to the data arrays involved, the neighbor lookups usually are trivial. In the case of 3D textures on the GPU, the volume density sample can be obtained using a single operation. Some sampling approaches require the use of ghost cells [PNP*17] if the data is distributed or not stored coherently in memory.

Important to note here is that empty or homogeneous space is subject to post-classification: a domain is empty only if the RGBα transfer function (TF) assigns all samples from that domain the alpha value 0. Likewise, if a domain or region of space is homogeneous, it is subject to the extinction/alpha value obtained from the

TF. Because it is required that the TF can be edited interactively in scientific visualization, space-skipping data structures should allow for interactive updates or rebuilds. Since the spatial arrangement can vary significantly based on the TF, updating space-skipping data structures is not trivial, even in the case of structured data.

Space-skipping data structures are all the more important if volumes are large because they help reduce pressure on the memory subsystem by reducing the number of samples. At the same time, one must be careful that the number of data accesses required for traversal is strictly in a lower order than the number of samples taken. With a complex hierarchy, the space-skipping algorithm can easily be limited by the number of traversal steps, which was first observed by Hadwiger *et al.* [HAAB*18]. Their *SparseLeap* data structure uses an Octree as the base empty space skipping structure. The authors observed that Octrees could have many neighboring leaf nodes with the same size and same *occupancy class* (empty/not empty) but are stored in different octants and subtrees of the Octree. To traverse from one subtree to another, traversal would have to unwind significant portions of the tree to skip from one leaf node to the next. The *SparseLeap* system uses an optimization that was formerly also proposed by Liu *et al.* [LCDP13] to combine consecutive ray segments of the same occupancy class to form coherent domains. These are stored in a per-pixel linked list [YHGT10] constructed using the GPU rasterization pipeline when the camera changes. After domain construction, a fragment shader traverses the domains via ray marching (cf. Algorithm 1 in Appendix A) to integrate color and opacity.

3.1.1. Grid and octree-based methods

Empty space skipping classification generally falls into two camps. Data structures like min/max trees by Knoll *et al.* [KWPH06] first roughly classify some coarser regions of space (such as bricks of $N \times M \times K$ voxels or Octree leaves of a certain minimum size) only by their minimum and maximum voxel value. When the transfer function changes, the min/max values are indexed into the (piecewise linear) alpha transfer function. If any alpha value inside the min/max range is non-zero, the region of space—and hence the domains computed when rays traverse the region—are non-empty. With this coarse representation, the same data structure (e.g., octree, uniform grid of bricks/macrocells) is reused without rebuilding. Empty leaf nodes or subtrees are only pruned and hence not traversed but still remain part of the space-skipping data structure.

3.1.2. High-quality k-d trees

Orthogonal to that are methods that aim at building an exact data structure whenever the spatial arrangement changes due to an update to the TF, such as the k-d trees by Vidal *et al.* [VMD08]. These approaches trade rendering for construction performance. The k-d trees by Vidal *et al.* are based on using 3D summed-area tables (3D-SAT) over coherent regions of space. The 3D-SATs are computed over a binary representation of the volume—empty voxels obtain the value 0, non-empty voxels the value 1. With 3D-SATs, querying if all the values inside an axis-aligned bounding box (AABB) are empty is an $O(1)$ operation. This allowed the authors to implement a simple top-down construction algorithm using plane sweeping, combined with a heuristic that compares the volume of the tight bounding boxes surrounding the non-empty cells to the left and to the right of the sweep plane candidates.

The algorithm by Vidal *et al.* [VMD08] served as the basis for several papers by Zellmann *et al.* [ZSL18, ZML19, ZSL21], who proposed methods to accelerate the construction algorithm up to a point where the k-d trees could be built in real-time. The authors observed that the 3D-SAT construction phase limits overall tree construction, while the ensuing plane sweeping phase was relatively cheap. 3D-SAT construction is equivalent to prefix summation. By breaking up the prefix sum into several smaller 3D prefix sums and computing 3D-SATs not for the whole volume but for bricks, the authors could reduce the computation time for this first phase and the precision required to store the 3D-SATs (and hence the memory consumption). By breaking the 3D-SAT computation apart, this phase could be trivially parallelized on a multi-core CPU. The ensuing, formerly $O(1)$ operation determining if bounding boxes are non-empty becomes $O(K)$, where K is the number of bricks. This algorithm was first proposed in [ZSL18] and was later adapted to run on the GPU [ZSL21]. For that, the authors first compute the brick-wise 3D-SATs but immediately discard them in favor of directly storing tight bounding boxes surrounding non-empty voxels inside the bricks. These bounding boxes are sorted on a Morton curve. The ensuing plane splitting phase (now realized with binning) combines these bounding boxes and computes the volume for the resulting union of boxes to evaluate the split plane heuristic.

3.1.3. Hardware accelerated ray-tracing based space skipping

Ray marching relies on the space skipping algorithm to traverse the domains in order and thus requires *spatial* data structures such as grids or k-d trees. Bounding volume hierarchies (BVHs) *can*, however, be used if the regions covered by the *leaf* nodes do not overlap [ZHL19]. This has led several authors to use hardware ray tracing to accelerate traversal. The work by Ganter and Manzke [GM19] used an OptiX *user geometry* (cf. Section 2.2) to traverse empty rectangular regions of space represented by AABBs. A source of inefficiency with *user geometry* is that the GPU needs to perform costly context switches when calling the user-supplied intersection program. This back-and-forth between tree traversal on the dedicated ray tracing core and the user intersection program executed on the shader core can result in serious performance bottlenecks.

Wald et al. [WZM21] proposed a method to leverage hardware accelerated ray tracing without user geometry but using a triangulated region boundary. Triangle primitive intersection tests can be accelerated in hardware on GPUs so that the traversal algorithm does not need to switch back and forth between dedicated ray tracing cores and shader cores. The triangulated region boundary is recomputed whenever the transfer function changes. The authors a priori build a min/max grid of coarse macrocells that can be quickly classified using the alpha TF. Full vertex and index buffers are pre-allocated for a quad geometry (realized with triangles) for all the macrocell faces. A CUDA kernel *activates* only those faces that partition macrocells of different occupancy classes (empty/not empty). The resulting triangle geometry (and BVH built from that) is used as a boundary to compute ray traversal domains, inside which a ray marcher is employed to integrate the volume density. By inserting quads/triangles only between macrocells of different occupancy classes, the algorithm also solves the same problem addressed by Hadwiger *et al.*'s [HAAB*18] *SparseLeap* algorithm.

We note that a majority of the (empty) space-skipping algorithms presented in this section are fundamentally also applicable to other volume representations. The algorithm by Wald *et al.* [WZM21], e.g., functions off an a priori space subdivision using macrocells, which can, in theory, also be obtained for unstructured data sets.

3.2. Paging and out-of-core approaches

The empty space skipping approaches described above generally apply to cases where the data (including the sampling and traversal data structures) fully fit into the memory of the GPU or compute node. This section focuses on approaches where the (structured) volume data does not fully fit into the node's main memory and requires out-of-core approaches. Out-of-core means using pageable memory to stream data to the GPU or asynchronously fetch the data from disk. Fundamentally, many of the techniques described here also apply to other types of volume representations.

For a complete review, we refer the reader to the state-of-the-art report by Beyer et al. [BHP15]. Recent papers presenting out-of-core paging and streaming approaches for structured volume data include the ones by Hadwiger *et al.* [HAAB*18], by Beyer *et al.* [BMA*19], by Wang *et al.* [WWJ19], and by Sarton *et al.* [SRL19, SCRL20]. We concentrate specifically on the data management aspects of such systems. We, therefore, review the work by Wang *et al.* as an approach for large-scale *CPU volume rendering*, as well as the work by Sarton *et al.* who present a general purpose GPU framework and an API that is used to implement several visualization-centric case studies.

3.2.1. Out-of-core volume rendering on the CPU

Due to memory limitations, out-of-core approaches for the interactive visualization of large volumes are mainly used for GPU architectures. Wang *et al.* [WWJ19] shows that out-of-core techniques can also be used for multi-core CPU architectures to decrease the loading times of large-scale volume data. They use a hierarchical data structure called *Bricktree* to represent the volume data. *Bricktree* is a multi-resolution and compressed data structure similar to the N^3 data structure by Lefebvre et al. [LHN05]. The N^3 tree is a generalization of the octree which divides the nodes into N^3 instead of 2^3. This N^3 structure has been used by several out-of-core visualization methods on GPU: the GigaVoxels [Cra11] system stores the bricks in N^3 tree structure (with N = 2) and later the CERA-TVR [Eng11] system that extended GigaVoxels stores scalar averages on the inner nodes instead of storing scalars only on the leaves, which enables them to perform progressive rendering. Wang *et al.* extends the CERA-TVR progressive rendering idea. Compared to the N^3 tree structure used in CERA-TVR, *Bricktree* has a more compact layout in memory which is more suitable for progressive rendering. Instead of representing the large volume with a single Bricktree, Wang *et al.* tile the large volume into a list of *Bricktrees* called "Bricktree forest." This allows them to use *Bricktree* structure when the volume dimensions are not equal or the dimensions are not a power of *N*.

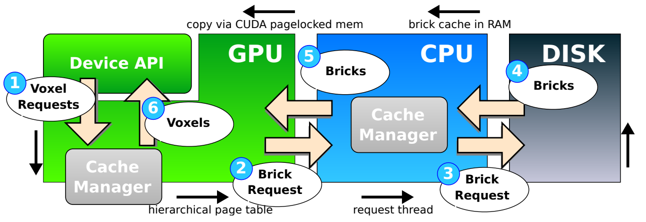

Figure 2: *The data path of the framework by Sarton* et al. *[SCRL20].*

Wang *et al.* use the *Bricktree* structure for progressive ray-marching. Streaming threads asynchronously load the bricks demanded by the rendering threads. During rendering, if a brick is not in the memory, the rendering thread approximates the scalar value by using the parent node's average scalar value and requests the brick from a data streaming thread. After a streaming thread loads the requested brick, the renderer refines the sample. Data streaming threads load the data in a breadth-first manner. Streaming threads prioritize loading the bricks by their level. Since the inner bricks constitute a tiny portion of the data, loading the inner nodes is faster than loading the leaf nodes.

The *Bricktree* structure and ray-driven progressive rendering allow visibility culling by only loading the required bricks. Additionally, the method allows Wang *et al.* [WWJ19] to perform early ray termination. *Bricktree* traversal can be stopped before reaching a leaf node to implement level-of-detail (LOD). Moreover, rays are terminated early if a predefined opacity threshold is reached. Fully transparent bricks for the given transfer function can be determined and terminated by storing the minimum and maximum scalar range in each *Bricktree* node, similar to the min/max trees from Section 3.1.1.

3.2.2. GPU out-of-core on-demand rendering and processing

[SCRL20] Frameworks that implement paging systems on GPU have been proposed, including the *SparseLeap* system by Hadwiger *et al.* [HAAB*18] that can on-demand load brick-sized data pages upon viewpoint changes, which, however, only become available at a later frame. On-demand loading of data pages is initiated when rays intersect the brick-bounding boxes. The predecessor of *SparseLeap*, the system by Hadwiger *et al.* [HBJP12], can visualize dense, anisotropic petascale volumes by decoupling 3D multi-resolution representation construction from data acquisition and decoupling sample access time during ray-casting from the multi-resolution hierarchy size. Their architecture uses a multi-resolution virtual memory architecture to detect missing data during volume ray casting as cache misses and propagates them backward for on-demand out-of-core processing.

The framework by Beyer *et al.* [BMA*19] that supports volume rendering of tera-byte-sized, pre-segmented microscopy data follows this spirit of making on-demand loading an image space decision. In contrast, the system by Sarton *et al.* [SCRL20] maintains the whole page table hierarchy in GPU memory, allowing them to implement more sophisticated mechanisms to initiate page load requests. The system implemented with CUDA is centered around a hierarchical page table fully managed on the GPU. Applications can use a device-side API to request voxels according to the data path shown in Fig. 2. The primary data structures and software components involved are i) a multi-resolution 3D mipmap hierarchy present on the disk and computed in a pre-process, ii) a host-side, asynchronous cache management component that retrieves data pages (in the form of LZ4-compressed 3D blocks) from disk on demand and places them in a page-locked memory buffer readable from the device, iii) a device-side, multi-level *least recently used* (LRU) cache comprised of 3D blocks, and iv) a device-side API that can be used to implement arbitrary volume processing applications.

From the API perspective, the *address space* of the voxels is fully opaque and extends from the device side down to the file I/O level. The user simply requests a voxel in a normalized texture coordinate system with the desired LOD. The GPU *cache manager* processes these requests and queries the hierarchical page table. If the block containing the voxel is present on one of the page table levels, the information that the block is now in the cache is updated accordingly. If the block cannot be found, the cache manager issues a block request served by the host system. To navigate the page table, the framework uses a multi-resolution page directory (MRPD), introduced by Hadwiger *et al.* [HBJP12]. The MRPD is available on the device and stores the blocks' locations in the cache hierarchy. The MRPD is intentionally shallow, so lookup costs do not depend on the LOD requested.

Block requests are maintained in a *request buffer* that all GPU threads have access to, and that is also used to update time stamps to mark blocks as *recently used*. This buffer can be accessed by multiple threads (and overlapping kernels) simultaneously. A polling thread on the CPU periodically processes and serves the requests by placing the respective blocks in page-locked host memory. The data transfer is realized using a parallel CUDA kernel, where each thread is responsible for copying one voxel to its dedicated position in the cache. When the data is placed in the cache, the page in the MRPD associated with the block gets updated to store the cache offset where the block is found.

The authors show that this asynchronous and demand-driven system can scale well for both visualization applications where the data is just being read and also for image processing applications where the framework is used to modify the data. A demand-driven rendering system can maintain consistently high frame rates because CPU/GPU communication only happens when it is strictly required, and even then, the data movement is asynchronous.

The framework by Sarton *et al.* [SCRL20] is a good example that demonstrates how visualization systems targeted at large-scale data are, in fact, sophisticated *data management* systems. The actual rendering algorithm is a small part of the overall system. The complexity of Sarton's system, in particular, lies in the sampling operation. While sampling of structured data is a relatively simple operation per se, it can become a complex task in the presence of cache hierarchies and virtual addressing, requiring careful optimization to be efficient.

Another notable, output-sensitive approach for out-of-core rendering and processing is proposed by Solteszova *et al.* [SBS*17]. Their method eliminates the need for processing and operations that have a negligible impact on the final image, by calculating the potential contribution of all voxels to the final image. The authors use a fast scheme to filter voxels that contribute significantly based on a maximal permissible error per pixel.

4. Volume Rendering of Adaptive Mesh Refinement Data

The term *adaptive mesh refinement* (AMR) has been coined by Berger *et al.* [BO84, BC89] in the field of numerical simulation and refers to semi-structured, hierarchical data topologies that the simulation code adaptively refines in regions of interest. AMR techniques provide significant computational and memory savings for producing and storing simulation data. AMR generates discrete simulation data at various levels of detail for problems that do not require uniform precision all over the simulation domain. This way, the regions of interest are represented at high resolutions, whereas the less critical areas are at lower resolutions. AMR data comes in different forms that usually (but not necessarily) have a structured, Cartesian *base domain* but vary widely in how they represent the hierarchy. Popular examples are block-structured AMR (cf. for example, the Chombo simulation code [CGL*00]), where the topology is represented by a set of overlaid grids from different refinement levels. Another popular representation is tree-based AMR, which again comes in multiple flavors, such as Octrees [LCPT11] and forests of Octrees [BWG11]. Some codes use trees structurally similar to Octrees but have different branching factors, like the data structure used by FLASH [DAC*14].

The output produced by many numerical simulation codes uses *staggered grids* [Kim17], which are cell-centric, i.e., scalar data such as density or pressure are not stored at the eight box corners, but at the center of the cell. Interpolation algorithms for first-order continuous reconstruction (e.g., the multilinear interpolation hardware on GPUs) usually require a neighborhood of eight horizontally, vertically, and depth-aligned data points (e.g., box, cube) as in Fig. 3a. For structured regular data as discussed in Section 3, the pragmatic solution employed by many frameworks is to shift the whole grid by half a cell's width, giving us the *dual grid* or *dual mesh*, where the data is vertex-centric. With structured volume data, this shift can be realized as a coordinate transformation inside the volume rendering loop (e.g., for Algorithm 1 in Appendix A by adding an offset of $\frac{1}{2}$ to the coordinate where the density field is evaluated at, in line 4).

This trick of shifting from the main grid to the dual grid using coordinate transforms only works inside the local grids but not at level boundaries for cell-centric AMR data (cf. Fig. 3b). There, the T-junction problem arises, and the locations of the data points do not line up (cf. Fig. 3c), which prohibits smooth reconstruction if not handled carefully. Prior methods, such as the one by Kähler *et al.* [KWAH06] would, in such cases, resort to reconstruction on the main grid and use a nearest-neighbor filter at level boundaries. This approach can result in all sorts of artifacts, such as isosurfaces with cracks [WWW*19] or volume renderings with visible seams where the subgrids meet [WBUK17]. Another possible solution is to convert the dual grid to an unstructured mesh comprised of hexahedra cells that are actually perfect cubes and to so-called *stitching cells*, which are generally unstructured elements (e.g., tetrahedra, pyramids, and wedges) to tessellate the level boundaries (Fig. 3d). This approach, referred to as "stitching" by the literature [WKL*01, WCM12, ME11, Wal20] has, however,

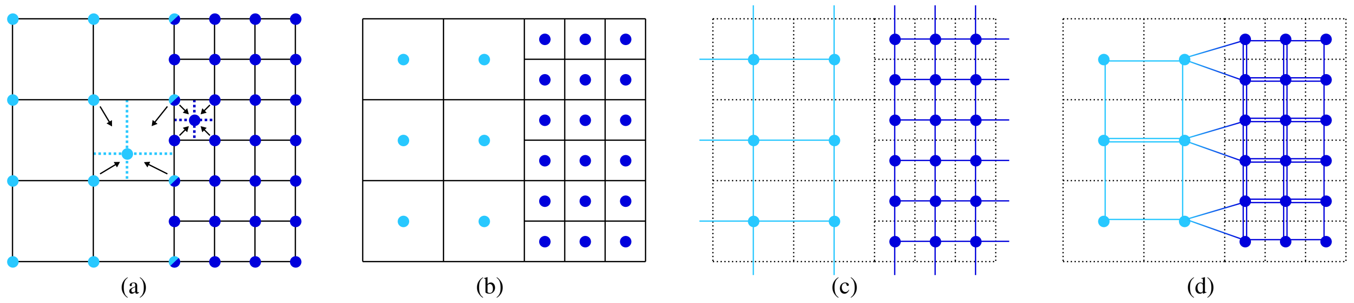

Figure 3: *The problem of rendering cell-centric AMR data with smooth, first-order interpolation at level boundaries. If the data is vertex-centric (a), reconstruction is equivalent to trilinear interpolation across the whole volumetric domain. The output generated by virtually all numerical simulation codes is cell-centric (b). Then, a common strategy is to shift the whole grid to perform reconstruction on the* dual grid *(c). Reconstruction with a rectangular interpolator at level boundaries is ill-defined because of T-junctions. One strategy to overcome this issue is to convert the whole dual to an unstructured mesh (d). This approach, however, is prohibitive when storage is limited because subgrids are now represented with hexahedra.*

extraordinarily high storage demands that come with turning all the cells that were originally just uniform voxels into full hexahedra.

A high-quality sampling of AMR data, and consequently, traversal data structures to efficiently do that on GPUs, has been the topic of several recent papers [WBUK17, WWW*19, WMU*20, WZU*21, ZWS*22a, ZSM*22, ZWS*22b] that we review in the following.

4.1. Sampling with smooth level transitions

We first review methods that fall in the sampling category (cf. Section 2). The types of AMR data that the papers we review focus on are structured regular and Cartesian; the AMR cells' sides have a uniform length (cubes), and the refinement law splits cells into eight equally sized finer cells. AMR cells are uniquely identified by the tuple $\{\mathbf{x}, l\}, \mathbf{x} \in \mathbb{N}^3, l \in \mathbb{Z}$, where $\mathbf{x}$ is a position in cell coordinates, and l is the refinement level. The uniform cell size C_w is proportional to the refinement level: $C_w = 2^l$ (some numerical simulation codes define the level to increase with higher refinement, in which case it can simply be transformed into the representation considered here). AMR cells form a *logical* coordinate system [WBUK17] whose integer coordinates have the same spacing as the finest cell size of the data set. Depending on the AMR type (e.g., block-structured, tree-based), the cells sometimes form a full space decomposition across all levels. In any case, cells *from the same level* are not allowed to overlap, although in general, cells from different levels can, and then form a level-of-detail (LOD) hierarchy that can be put to use by the renderer. We sometimes refer to the dual of a cell, obtained by shifting the cell itself by some amount. In that case, to distinguish the two types of cells, we refer to the original cell as the *leaf cell* to the dual.

4.1.1. Reconstruction with tent-shaped basis functions

The trilinear interpolation operation (cf. Fig. 3a) on (locally) structured data can also be viewed as the convolution of the eight data values with a tent-shaped basis function as shown in Fig. 4a (here in 1D). Wald *et al.* [WBUK17] extended this concept, which borrows from scattered/point-based data reconstruction [FN80], to devise a C^0 continuous filter for high-quality reconstruction of AMR data, including at the level transitions. For structured data (where the convolution in practice is folded into the trilinear interpolation operation), the *footprint* of the tent basis function just reaches one-half cell's width beyond the cell boundaries (Fig. 4a). With AMR data, we have the situation that a coarse-level cell can have many small cells from a finer level as direct neighbors or as indirect neighbors, which do not share a face with the cell but are *close-by*. In that case, if the smaller cells fall inside the footprint of the larger cell, the latter contributes to the reconstructed data value at the positions of the finer cells proportionally to its size. With Wald *et al.*'s *basis reconstruction method* for AMR data, this is accounted for by setting the footprint of the cells to twice the *actual* cell size so that the filter extends by half a cell size *of the current level* in each direction (cf. Fig. 4b). To obtain values at position $\mathbf{x}$ the authors compute a weighted average using a tent-shaped basis function centered around the cell, and reaching one half cell's width across each side.

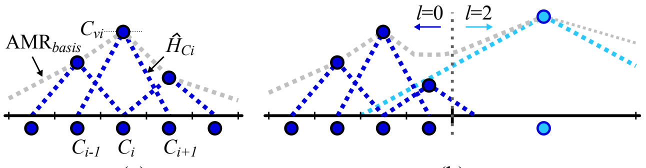

Figure 4: *AMR basis reconstruction by Wald* et al. *[WBUK17]. (a) For structured data, basis function reconstruction is equivalent to trilinear interpolation. (b) Reconstruction at a $l = 0$ to $l = 2$ level transition. The reconstructed function is C^0-continuous and adaptive, yet not interpolative.*

Note that this reconstruction method does not have an interpolative property, i.e., reconstructed values at the known data points are generally not the exact cell values at those points (Fig. 4b). Furthermore, this reconstruction method requires an operation the literature refers to as *cell location* to locate not only the direct neighbors of each cell but all the neighbors in the neighborhood defined by the footprint of its coarsest cell. Cell location is an operation commonly required by all the reconstruction methods we review and has sparked further research on efficient *cell location data structures* that we review below in Section 4.2.

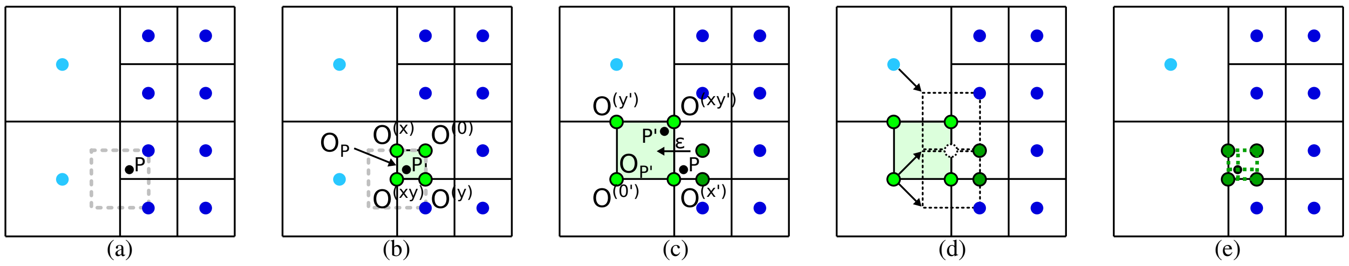

Figure 5: *Octant method by Wang* et al. *[WWW*19]. The octant method presents a rectilinear AMR interpolator that supports smooth reconstruction at level boundaries (a). Interpolation inside an octant (b) defined by corner points $O^{(..)}$ is similar to interpolation using dual cells (dashed rectangle in (a,b)); the corner values can be directly derived from the dual corners. (c) not all the octant corners directly align with data points; then, the coarsest side in the neighbor recursively provides its value to the finer side. To reach the coarse side, the current evaluation position P is offset by a small value ε (e.g., half the width of the finest cell in the data set) in the direction of the coarser cell. (d) Once inside the coarsest side, different strategies exist for how the octant vertices are interpolated from the surrounding cells. (e) When all data values are available, the value at P can be obtained via trilinear interpolation of the octant corner values.*

4.1.2. Reconstruction with octants

The method proposed by Wang *et al.* [WWW*19] is orthogonal to sampling with tent basis functions. It follows the idea of first finding *octants* that form a complete space decomposition of the refined grid and its *leaf cells*, whose corner points carry data so we can interpolate linearly inside them. Conceptually, when it comes to interpolation, an octant is similar to a *dual cell*. Dual cells can be obtained by shifting the leaf cell (whose data is stored at the center) by $\frac{1}{2}$ its width along the principal axes. The sign of the shift $(\pm X, \pm Y, \pm Z)$ does not affect the definition of a dual cell. Valid dual cells can be obtained by shifting in either direction. The octant's extent is equivalent to the intersection of the dual and the leaf cell.

A dual cell is shown in Fig. 5a. Here, two of the dual's corners align with actual data points, which can be directly used to determine the data values at the rightmost corners of the octant. The octant is given by eight (in our 2D illustration: four) corner points. The corner point $O^{(0)}$ is defined to be the one aligned with the leaf cell center. The other vertices fall on the leaf cell's corner, edge, or face and are named accordingly. For example, the corner obtained by moving along the X direction only and which falls on a face (in 2D: edge) is called $O^{(X)}$. The corner vertex obtained by moving along all principal directions is called $O^{(XYZ)}$ (in our example, $O^{(XY)}$ because we are in 2D).

If we were to interpolate inside the dual (which we do not because the duals that form a complete decomposition of space are not trivial to find), we would trilinearly interpolate the corner values. Fortunately, the octant corners can easily be derived from the corners of *a* dual by computing averages. In Fig. 5b, for example, the value at $O^{(Y)}$ can be obtained as the median of the two rightmost dual corner values. The value at $O^{(0)}$ is just the value of the leaf cell's center, as the two points line up.

For the two leftmost octant corners ($O^{(X)}$ and $O^{(XY)}$ in Fig. 5b), data values are not available because the dual's corners do not line up with vertices. In that case, the rule is that the missing data values are obtained via interpolation on the *coarser* side. If we always pick the coarser side to dictate the data value, we thereby *implicitly* find the aforementioned duals' space decomposition that is non-trivial to find *explicitly*. For that, we shift the evaluation position P by a small amount ε (e.g., half the finest cell width of the data set), so the new evaluation point P' falls inside the coarse side and there induces a new octant (cf. Fig. 5c). Inside that octant, which is now on the coarse but not necessarily the coarsest side, the procedure is repeated recursively. As either corners, edges, or faces can have level boundaries, this results in potential recursion when computing the values at the octant corners.

On the coarse side, it is mainly up to the user of this framework how to interpolate the missing corner values, as indicated in Fig. 5d. The methods proposed by Wang *et al.* [WWW*19] include linear interpolation on the coarsest or finest level, on the current level, or even the tent basis functions by Wald *et al.* [WBUK17] from Section 4.1.1. Finally, when the corner values are available and the potential recursion returned, the value at P is obtained via trilinear interpolation (cf. Fig. 5e). Note that this procedure does not require recursion inside subgrids but gracefully provides the same reconstruction that we would also obtain by performing trilinear interpolation on the dual grid, which would be obtained by shifting all cells in the neighborhood by the same amount in the same direction.

4.2. Data structures for efficient cell location

This section discusses data structures to accompany the sampling methods discussed above. Some of those data structures serve the purpose of sampling the data *and* traversing it as efficiently as possible.

Cell location fundamentally is a *neighbor search*: given the cell we are in, we need to find the (directly or indirectly) adjacent cells to form dual cells, octants, etc. Even in the simplest cases, this neighbor query needs to be supported by a data structure that directly stores or allows retrieving the adjacency information. Traditional data structures used for neighbor search are k-d trees [Pan08]. Wald *et al.* [WBUK17] also use this data structure to find all the neighboring cells inside the neighborhood of the sample point. During construction, the authors compute split candidates by first determining a list of all the level boundaries inside the region to split. They pick the boundary closest to the mid-point/spatial median along the current axis.

The k-d tree by Wald *et al.* [WBUK17] serves the very purpose of accelerating cell location—i.e., sampling and not traversal. A ray that is marched through the volume density, such as in Algorithm 1 or Algorithm 2 in Appendix A, will traverse the k-d tree over and over at each 3D sample position. This process is wasteful if the cells of multiple consecutive samples are from the same refinement level or even subgrid. Suppose we take N samples from the same subgrid of, e.g., a block-structured AMR data set. Locating the cells also includes locating the same subgrid repeatedly. A more efficient strategy would first traverse the subgrid hierarchy and take the N samples directly from the subgrid without traversing the hierarchy again. This strategy is efficient for relatively large subgrids. However, suppose the AMR structure is an octree or similar. In that case, one suffers from the same issue we also encountered in Section 3 when discussing *SparseLeap* by Hadwiger *et al.* [HAAB*18] that a multitude of leaf nodes from the same level are adjacent but do not reside in the same subtree. A solution to this problem in the context of AMR volume rendering was proposed in seminal work by Kähler *et al.* [KSH03, KWAH06]. They preprocess the AMR data by first dropping whatever hierarchy is there so that only the cells themselves are considered. They then build a k-d tree to find bricks as big as possible to contain as many *same level* cells as possible. Contrary to Wald *et al.*'s k-d tree, this one can serve as a traversal data structure, where rays first determine the *brick* they are in and then sample and integrate that brick from front to back. This procedure is repeated until the ray segment reaches the volume boundary or reaches t_{max}.

Apart from special handling for potentially overlapping block-structured AMR subgrids, the bricks generated by Kähler *et al.* [KSH03] form a complete space decomposition and are not allowed (nor designed) to overlap. Hence, the k-d trees do not help find neighboring cells near the level transition. Complete traversals are necessary to determine the neighborhood in those regions. As the setting of Kähler's approach is rather static—the method traverses the k-d tree using a rasterization API, and traversal is initiated on the CPU—their method is limited to vertex-centric data or generates the artifacts mentioned above at level boundaries.

The *ExaBricks* framework proposed by Wald *et al.* [WZU*21] extends Kähler *et al.*'s [KSH03] idea to use a spatial subdivision for coherent traversal and support smooth reconstruction at cell-centric AMR level boundaries. We note that Kähler's framework is almost two decades older than Wald's, and hence the *ExaBricks* framework only borrows basic concepts but builds upon different technology that requires other kinds of optimizations overall. While Kähler's framework uses rasterization, ray tracing in a fragment shader, and k-d tree traversal on the CPU, *ExaBricks* incorporates a GPGPU execution model, hardware ray tracing, and GPUs with abundant resources, which allows the authors to implement state-of-the-art techniques such as implicit isosurfaces (cf. Algorithm 2 in Appendix A), ray-traced ambient occlusion, or adaptive sampling. The two frameworks share the basic concept of traversing *coherent* rays through spatial partitions of, ideally, larger regions of same-sized cells.

ExaBricks is not only a (prototypical) framework but also a data structure. The basis for the *ExaBricks* data structure form Kähler *et al.*'s [KSH03] coarse k-d trees of bricks. The *ExaBricks* framework provides tools to convert any AMR format with a Cartesian base domain to the internal format, consisting of two file types. One file type stores the linear array of scalar volume data per scalar field—*ExaBricks* supports multiple scalar fields that can be rendered using multi-channel DVR or that are mapped as color fields onto an implicit isosurface. The other file type stores cells, defined by their position in logical grid coordinates, refinement level, and scalar ID. Like Kähler *et al.*, Wald *et al.*'s [WZU*21] framework discards whatever original AMR hierarchy is there and, in the following, only uses these arrays of scalars and cells.

The common scalar and cell formats serve as input to a preprocess where the cells are turned into bricks, defined by their origin, number of cells, the common refinement level, and the cell IDs indexing into the scalar field arrays. This representation is equivalent to Kähler *et al.*'s [KSH03] representation. To obtain the bricks, Wald *et al.* [WBUK17] also use a k-d tree; they, however, experimented with different heuristics when building these. Notably, the data targeted by Wald *et al.* is much more large-scale, given the different periods in which the two frameworks were developed. In the presence of billions of cells, the k-d trees generated for Wald *et al.*'s test data sets can become quite complex. The heuristic the authors recommend is similar to a surface area heuristic (SAH) [Wal07] for volumes. In contrast to Kähler *et al.*, Wald *et al.* never use the k-d tree other than for generating the bricks. Hence, the preprocessing output is bricks, while the k-d tree is immediately discarded.

To support smooth interpolation at level boundaries, Wald *et al.* [WBUK17] first decide on a reconstruction method. This method determines by how much the *filter support* of the bricks overlaps. For example, when using tent-shaped basis functions (cf. Section 4.1.1), each cell has a filter support of $\frac{1}{2}$ the cell's width at the brick boundaries; the *brick domains*, hence, reach by $\frac{1}{2}$ cell's width of the brick's refinement level beyond the tight bounding boxes of the bricks. While the brick bounding boxes form a complete space decomposition, the *brick domains* generally overlap. The overlap regions themselves (including the regions where only a single brick is active and there is no overlap), however, *implicitly* form a space decomposition as well. The authors call these the *brick overlap regions*. Brick overlap regions can form the equivalents of what in 2D would be T- or L-shapes and hence are not amenable to efficient rendering on GPUs.

ExaBricks builds yet another k-d tree to prepare the brick overlap regions for rendering, this time to split the overlap regions into convex, axis-aligned boxes called the *active brick regions* (ABRs). Again, the k-d tree hierarchy is dropped, and the framework keeps just the resulting ABRs. The resulting ABR data structure consists of a linear list of all brick IDs indexing into the brick list generated during preprocessing. The ABRs are defined by their offset and length into this *ABR leaf list*. In addition, the ABRs also store an axis-aligned bounding box generated by the k-d tree splitter. A renderer can traverse the ABRs using that data structure. When a ray hits an ABR, it uses the offset and length to retrieve the IDs of the bricks covered by the ABR. This process adds another level of indirection (from ABRs to bricks) on top of Kähler *et al.*'s [KSH03] original data structure, but allows the renderer to perform basis function interpolation inside the region covered by the ABR.

To traverse the ABRs, Wald *et al.* [WBUK17] use hardware-accelerated ray tracing with RTX by building an OptiX BVH over the ABR bounding boxes. This BVH replaces the k-d tree, which could also be used to traverse the ABRs on architectures without hardware ray tracing. To support empty space skipping and adaptive sampling, Wald *et al.* also store the min/max data values for the ABRs. During ABR construction, the authors carefully compute the actual cells' min/max values, not bricks, that constitute the ABRs. Would they use the bricks' min/max values instead, the space-skipping data structure would be of much lower quality, as later reported by Zellmann *et al.* [ZWS*22a]. This approach allows them to classify and deactivate empty ABRs whenever the transfer function changes, similar to the min/max trees by Knoll *et al.* [KWPH06]. When that happens, the OptiX BVH is rebuilt interactively. Zellmann *et al.* [ZWS*22a] later proposed to use BVH refitting to accelerate this further and showed that the resulting BVHs allow for equivalent sampling performance. The resulting data structure allows for empty space skipping because a ray traversing the ABR BVH will never report an intersection with an empty ABR, as it is not even present in the BVH. Once an ABR's bounding box is intersected, the OptiX intersection program reports a hit, and the ABR is processed using a ray marcher. The stepsize of the ray marcher is proportional to the ABR's finest cell size.

The *ExaBricks* data structure was later used by various other works, including Zellmann *et al.* who extended the data structure to support RTX-accelerated flow visualization [ZSM*22] and fast streaming updates [ZWS*22a]. In a recent pre-print [ZWS*22b], the authors extend *ExaBricks* to support volumetric path tracing with delta tracking (cf. Algorithm 3 in Appendix A). Here, the sampler is no longer coherent: in the typical case, only a single or very few samples are taken from the traversed ABR. This preliminary work suggests that for Monte Carlo path tracing and other incoherent workloads, ABRs form a viable sampling data structure but not necessarily a data structure that is optimal for (incoherent) traversal. Alternative data structures and interpolation schemes based on dual cells were later proposed by Wang *et al.* [WMU*20] but were limited to tree-based AMR.

5. Rendering Unstructured Volume Data ●

Unstructured meshes that are, e.g., produced by finite elements or finite volume codes can be thought of as the ultimate way of refining the data to adapt to regions of interest. Some codes output tetrahedra only, while others produce meshes with different types of polyhedra such as tetrahedra, hexahedra, pyramids, wedges, or high order [NLKH12], twisted and bent elements. Unlike structured volumes that can be directly sampled in texture space, volume visualization of unstructured grids requires managing the topology, geometry, and scalar field to locate and render these polyhedral cells.

The fundamental question of how to *render* such data has long been solved by seminal works. A common object-order approach also implemented by standard software like [AGL05], for example, is the cell/tetrahedra projection [ST90], which involves a costly sorting per viewpoint update to bring the cells into visibility order. Garrity [Gar90] was the first to propose ray casting for unstructured volumes, and Weiler *et al.* [WKME03] were the first to propose a parallel GPU implementation. Silva *et al.* [SCCB05] reviewed early GPU-based approaches for volume rendering of unstructured grids. However, in times of ExaScale computing, the data management aspects rather than rendering pose significant challenges. Traversal and sampling through ray casting techniques, unanimously used today, has to cope with the complexity and ever-increasing size of the unstructured volumes. At the same time, advancements in hardware technology have led to heterogeneous systems with complex, potentially distributed memory hierarchies that the visualization algorithms must adapt to.

This section discusses recent trends in traversing and sampling unstructured volumes (see Fig. 6). We discuss how these methods compress and augment the data with auxiliary structures. We will first discuss *element marching* approaches that locate the cells via connectivity information [MHDH07, MHDG11, SDM*21]. While the advantage of these approaches is that they do not require large auxiliary traversal and sampling data structures, their flexibility is limited because they always require to completely march from the entry to the exit point of the ray/volume segment traversed (cf. Fig. 6a). Alternative approaches like that by Binyahib *et al.* [BPL*19] use a fixed sampling pattern (cf. Fig. 6b), which can be helpful when targeting distributed memory systems. Another family of techniques we discuss *use point ray sampling* (cf. Fig. 6c) by traversing rays with length zero into a ray tracing acceleration structure and locating the cell by determining if the origin falls inside. Here we focus on the papers by Morrical *et al.* [MWUP22] and by Wald *et al.* [WMZ21] who proposed efficient implementations using OptiX and RTX.

5.1. Marching with element connectivity

Ray tracing-based volume traversal requires one to find ray segments $[t_{min}, t_{max}]$ to take samples from (cf. Section 2). While with structured volumes, this is often as simple as testing the ray for an intersection with the axis-aligned bounding box of the grid, with unstructured data, the volume boundary is usually non-convex and defined by an arbitrary surface mesh. The papers we review in this section propose traversing the spatial partitions by finding entry and exit points and marching from cell to cell ("element marching"). When the spatial partition we are in is left, we need to check if we actually exited the volume, or just the current local partition, in which case we need to find the next pair of entry/exit points (cf. Fig. 6a). Early work by Muigg *et al.* [MHDH07] proposed a hybrid approach for volume traversal that first decomposes the unstructured grid into bricks using a k-d tree. That allowed them to traverse the (convex) brick boundaries, not the boundary mesh.

Follow-up work by Muigg *et al.* [MHDH07] implemented the traversal step using rasterization and *depth peeling* [BPCS06]. The boundary mesh is rendered using the standard rasterization pipeline with back-face culling turned on to fill the depth buffer. Then the mesh is rendered again, but with front-face culling active to find the exit face for the entry face that is now in the depth buffer. Ray/element marching is implemented in a fragment shader. After that, the layer is "peeled off," and the process is repeated until the whole volume is traversed. This is a multipass approach: the host program initiates rendering passes whenever a layer is peeled off, and the volume is not fully rendered. The authors also proposed a very compact data structure to store the connectivity information without redundancy through a list of faces and without explicitly

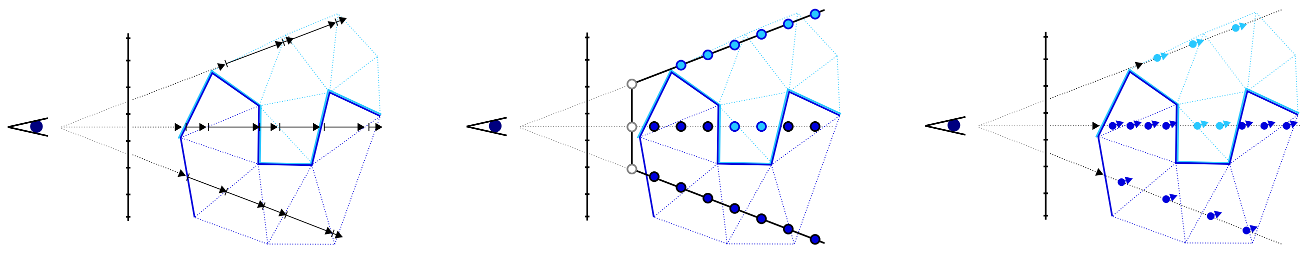

Figure 6: *Unstructured data ray-casting. (a) Element marching using connectivity information: Muigg* et al. *[MHDH07, MHDG11], Sahistan* et al. *[SDM*21, SDW*22]. (b) Fixed-width sampling: Binyahib* et al. *[BPL*19] (c) Empty space skipping and adaptive sampling with RTX-BVH traversal: Morrical* et al. *[MUWP19] and Morrical* et al. *[MSG*22]*

representing cells. They achieve this by separating the number and indices of the vertices that make up a face from the mesh topology. Both papers by Muigg *et al.* [MHDH07, MHDG11] employ interpolation with a combination of simple barycentric coordinates at the ray/face intersections and mean value coordinates [Flo03] for the cell's interior, distributing the samples equally along the ray segment within it. Additional samples can then be computed between the positions of the already computed samples by linearly interpolating between them.

More recently, Sahistan *et al.* [SDM*21] proposed to leverage hardware-accelerated ray tracing for element marching by using an OptiX triangle BVH to find entry and exit points. The BVH is built over what the authors call the *shell*—the same boundary mesh also traversed by Muigg *et al.* [MHDG11] defined by the unconnected faces of the boundary elements of the data set. The method supports tetrahedra only. The authors compress the data by what they call *XOR compaction*, an alternative to Muigg *et al.*'s to store the element connectivity. Using a ray tracing API to traverse from one spatial partition to another allows the authors to use secondary rays for ambient occlusion or shadows, which is more challenging to achieve with rasterization-based frameworks. The authors also need to find entry faces for the secondary rays to initialize sampling via element marching. For that, they first trace a ray from the current position inside the volume *backward* against the shell and then march from the shell intersection until they pass by the current position to only then start computing transmission. In contrast to the BVH sampling techniques we summarize below in Section 5.3, this approach's memory consumption is very competitive, both through compression and because the BVH is only built over the shell, which accounts for a fraction of the actual elements. A direct comparison between Sahistan *et al.*'s and Muigg *et al.*'s techniques in terms of runtime and memory performance was unfortunately not provided.

5.2. Parallel and distributed rendering

Binyahib *et al.* [BPL*19] propose a scalable unstructured volume rendering algorithm that extends the one in VisIt [CBW*12] and is based on prior work by Childs *et al.* [CDM06]. Binyahib's algorithm proceeds in three phases. It first classifies the input cells/unstructured elements as *small* or *large* by estimating the number of samples a viewing ray will take, given a fixed-width sampling as in Fig. 6b. As a rule of thumb, a cell that a ray crossing samples only once would be considered small, but the framework exposes a threshold value that allows the user to control this.

The three-phase algorithm first processes the small cells in an object-order fashion. Then, processors exchange their generated samples' results for only the small cells, as well as the remaining large cells directly. Each processor then generates a sub-image from its share of what the authors call *partial composites*—i.e., the partially composited pixel results for the small cells—and the remaining large cells in an image-order fashion.

The sampling pattern incurs a view frustum-aligned uniform grid (cf. Fig. 6b) for which a shared buffer is pre-allocated. Typical sample counts are $1K^3$ for a $1K$ viewport (i.e., $1K \times 1K$ samples in image space and $1K$ in the z-direction). For modern workloads, the sample count increases proportionally to the number of cells to sample at the Shannon-Nyquist limit. This approach, which was also employed by Childs *et al.* [CDM06], has a prohibitive memory footprint.

During the *first phase* of the algorithm, the processors evaluate their samples in object order, and the threads per processor fill the pre-allocated grid buffer in parallel (i.e., parallelism over cells). Then, in an image-parallel process, consecutive samples from that buffer are post-classified with the RGBα transfer function and partially composited to form *partial composites*. We propose to also refer to those partial composites as fragments that are associated not with samples directly but instead are the composited and associated with the (locally convex) ray segments (cf. the center ray in Fig. 6b). Each fragment carries RGB, opacity, and depth information. To compute these, the authors use *over* compositing [PD84]. The lighting model used is "absorption plus emission" (cf. Algorithm 1 in Appendix A). After this process, the fragments of a single pixel are potentially distributed across several processors.

During the *second phase*, the processors exchange the fragments/partial composites and the remaining large cells. In preparation for the next phase, the algorithm now transitions from object to image order processing and hence partitions the image into tiles distributed equally across the processors. The processors exchange the data (fragments and large cells) using direct-send [Hsu93], realized with MPI.

During the *third phase*, the processors receiving their data first process the large cells and generate partial fragments. Afterward, the fragments (coming from the other processors and/or resulting from large cell sampling) are sorted using their depth value and composited using pre-multiplied alpha, giving us the final RGB sub-image for the tile the processor is responsible for. The sub-images are finally sent to a single host processor for display.

The framework can enforce object- or image-order processing only by using the threshold to classify cells into small/large. All cells are considered large if the threshold is set to 1, so no work is performed during the first phase. Conversely, if the threshold goes to infinity, all cells are small, and during the third phase, only fragments but no large cells are processed. In that case, the rendering algorithm becomes object-order only.

A potential source of the inflexibility of this algorithm is the large pre-allocated sample buffer, which needs to scale better with growing data resolution and when sampling at the Shannon-Nyquist limit is desired. At the same time, the sampling pattern is not adaptive. With adaptive sampling, one could design an algorithm that takes fewer samples from the large than from the small cells and thus would not require a hybrid phase with separate phases for small and large cells at all. Algorithms that can adapt the sampling rate to the local frequency and naturally skip over empty space are the subject of section Section 5.3.

The framework by Sahistan *et al.*'s [SDM*21] was later also adapted to support large, non-convexly shaped distributed data as Binyahib *et al.* [BPL*19] does. The extension was first published by Zellmann *et al.* [ZWB*22], but is explained in more detail in the pre-print by Sahistan *et al.* [SDW*22]. The authors extended Sahistan's element marcher with a *deep compositing* algorithm. Instead of employing IceT [MKPH11] for compositing, which is the state of the art in most parallel distributed renderers [CBW*12, AGL05], the authors allocate *fragment buffers* to hold partially composited samples per MPI rank and data partition. The fragments are later exchanged via direct send and finally composited by the processor responsible for the pixel. Conceptually, this is similar to Binyahib's *partial composites* (cf. Section 5.2). However, the method is not restricted to fixed sample positions, nor does it require allocating large sample buffers. The deep compositing approach is also not restricted to element marching but can easily be combined, e.g., with BVH sampling (see below). As such, it represents a general traversal routine (cf. Section 2) where the ray segments for the spatial partitions are processed in parallel by different processors.

5.3. Cell location with hierarchical data structures

The previously presented methods implement sampling either by marching to the sample position using element connectivity [MHDH07, MHDG11] or by placing samples at fixed positions and using object space methods [BPL*19]. An alternative technique first established by Rathke *et al.* [RWCB15] performs the necessary cell location to obtain the samples via *point queries*. To this end, a hierarchical acceleration structure (e.g., BVH or k-d tree) is used. With a software implementation, the acceleration structure is traversed. At each hierarchy level, the sample position is tested for point containment inside the domain bounds, or half-space, until finally arriving at the leaves, where the individual elements are tested for containing the point. Point containment tests are often much cheaper than ray/element intersection tests. In the case of unstructured elements with planar faces, this can be implemented by simple tests for the signed volume that the point forms with each face. This approach is conceptually similar to the rasterization test performed by GPUs [Pin88].

5.3.1. Cell location with RT cores

On modern GPUs with ray tracing hardware, the point-in-element test can be implemented by clever use of RT cores. The accelerator of choice is a BVH, and instead of a point, one traverses a ray with zero length ($t_{min} = t_{max} = 0$), originating at the sample position, and with an arbitrary direction vector, through the hierarchy. In the simplest case, the point containment test is implemented using *user geometry*. This method of using an acceleration structure to perform cell location has the significant advantage that sampling can be decoupled from traversal. While it is perfectly viable also to use the BVH for traversal, additional (often simpler) data structures can be used to skip over empty space, or to adapt the local sampling rate, e.g., based on the frequency or density majorants (cf. Section 2). RTX-accelerated versions of this algorithm were first presented by Wald *et al.* [WUM*19] and by Morrical *et al.* [MWUP22], who proposed extensions to support higher order elements with non-planar faces. Conceptually, the method is illustrated in Fig. 6c.

Efficient implementations must make several design choices that can affect memory consumption and sampling performance. For example, one way to represent unstructured elements chosen by Morrical *et al.* [MWUP22] is to allocate eight indices per element into the shared vertex array, regardless of the element type stored. Elements with fewer indices, such as tetrahedra and pyramids, have their excess indices padded using an invalid value (e.g., -1). That way, the elements can be represented in a single list. Alternative layouts (cf., e.g., [MSG*22]) store offsets into an index list sorted by element ID, which allows representing the elements more compactly and may also be more efficient in terms of thread group divergence on the GPU. While planar elements can be efficiently tested with the aforementioned rasterization techniques, non-planar elements are tested with Newton Raphson refinement using the plane distance as an initial approximation for the root [MWUP22].

User geometry intersection programs suffer from the back and forth between shader and RT cores mentioned in Section 2.2. To overcome this issue, Morrical *et al.* [MWUP22] also propose a point query that uses ray/triangle intersection tests exclusively and can thus be performed entirely in hardware. For that, they tessellate the faces of the elements into triangles and quads and duplicate the shared faces of adjacent elements. In contrast to the user geometry-based method, this one uses the fact that when tracing a *non-infinitesimal* ray with a sufficiently set t_{max} originating inside an element will have the closest intersection with one of the element's faces. By duplicating the shared faces, using backface culling, and using the convention that face normals belonging to the element point inwards, the corresponding element ID can be obtained. The choice of the ray's t_{max} parameter can influence the performance of this method. If t_{max} is too small, it will potentially not hit the element boundary at all, while a t_{max} that is too high results in unnecessary (yet hardware-accelerated and generally cheap) ray/triangle intersection tests before finding the closest intersection. Although the authors propose regrouping the elements in an (expensive) pre-process, which allowed them to eventually get rid of the duplicate faces, this method is rather expensive regarding memory consumption. The authors report that in contrast to the user geometry-based method, the triangle-based one, on average, uses twice as much memory.

5.3.2. Space skipping and adaptive sampling

Using RT cores, cell location with point queries can use hardware acceleration. Their flexibility in sample position placement makes them a viable alternative to marching- and shared-face-based solutions. As the cell location procedure does not require face connectivity, this method is less prone to sampling issues due to degenerate geometry. Still, performing a complete BVH traversal per sample position, even in hardware, can be costly, mainly when the model is large. As discussed in Section 3.1, like in the structured case, careful sample placement can significantly improve performance. This is even more relevant in the case of BVH cell location, where the individual sample is expensive compared to, e.g., simple trilinear interpolation in a 3D array. In the case of structured data, the spatial arrangement depends on the density and the alpha value from the RGBα TF but not on the shape of the volume (which is trivially just a box). *Unstructured* meshes, in general, have non-convex boundaries so that empty space is defined by both the boundary mesh as well as the RGBα TF. Furthermore, adaptive sampling is even more critical in this case because individual elements can vary significantly in size, whereas structured voxels, though potentially spanning multiple level sets, usually vary less in local resolution.

With point query-based sampling, it is *per se* possible to adapt the local sampling rate according to some criterion arbitrarily. For that, a *traversal data structure* accompanies the BVH used for sampling. Morrical *et al.* [MUWP19] propose to base the traversal data structure of a k-d tree partitioning computed in a pre-process. The resulting domain boxes, though presenting a *space* partitioning, can still be traversed using OptiX and RTX, and the fact that the domains do not overlap allows for efficient traversal and min-max tree-based empty space skipping (cf. Section 3.1.1). To adapt the sampling rate, the authors compute the variance of the *post-classified* density values inside the domain boxes, roughly representing the local frequency after TF application. Morrical *et al.* use absorption plus emission-style ray marching (cf. Algorithm 1 in Appendix A). They use opacity correction to account for the adapted sample rate, so the resulting RGBα color roughly matches the result obtained with a fixed marching step size.

5.3.3. Mesh data and acceleration structure compression

Compared to BVH-based cell location, element marchers (cf. Section 5.1) have the primary advantage of only requiring face connectivity, which can also be compressed significantly. In contrast, the BVH and complete vertex index lists required by the above approaches can come at very high memory costs. Apart from pragmatic solutions using distributed memory computing [ZWB*22], research has also focused on compressing both the unstructured mesh and the sampling acceleration data structure. Two alternative approaches that optimize for different objectives were proposed by Wald *et al.* [WMZ21], and later by Morrical *et al.* [MSG*22].

These compression algorithms build on research on *surface* mesh compression. Solutions in this field include *multiresolution decomposition* [PDP*19], and compression of face index arrays [FFMW20, FWDF21]. The framework by Wald *et al.* [WMZ21] borrows from these concepts. The authors use a meshlet decomposition, which allows them to reorganize the mesh's vertices so their indices can form small groups. Inside each group, the indices can be represented with lower resolution (16 bits instead of 32 bits) or flattened altogether, resulting in even lower memory consumption with a careful meshlet generation scheme. Other minor optimizations include grouping neighboring tetrahedra to pairs ("*tet-pairs*"), similar to how production ray tracers sometimes group triangle pairs to quads so they can lower the storage requirement to four instead of six vertices. With tet-pairs as an extra element type, they rearrange the element list so that elements of the same type are stored next to each other, allowing for an even more compact representation.

At the center of this method is a (software) BVH compression scheme building off an unoptimized reference implementation taken from OSPRay [WJA*17]. The authors, step by step, extend this reference. They transform the reference to become an eight-wide BVH and use quantization [YKL17, BWWA18] with 16-bit precision for the domain boxes of the subtree nodes. By carefully rearranging the nodes and primitives, the authors do not require to store any indices, but instead, node and primitive lists directly. Another optimization to the sampling BVH that the authors propose is to (bottom-up) collapse up to eight leaf nodes to form what they call *multi-leaves*. These multi-leaves can be traversed linearly; however, the authors evaluated building small OptiX BVHs over each, which proved superior to linear traversal in software. In this case, the software and OptiX BVH form a *two-level acceleration structure*, with the top level realized in software and the bottom level realized with OptiX.

While Wald *et al.* [WMZ21] propose to use extreme compression for the sampling acceleration structure and element indices, which eventually allows them to visualize the 2.9 billion elements NASA Mars lander [Nat21] on a single GPU, they do not compress vertices, which renders this compression method *lossless*.

While resulting in extreme compression with highly superior peak and total memory performance, *building* the acceleration structure can take up to several hours for the largest of data sets used by Wald *et al.* [WMZ21]. The method by Morrical *et al.* [MSG*22] seeks to overcome this issue by implementing compression based on a lazy sorting and clustering scheme resulting in much lower pre-processing times. The authors first sort the elements on a Hilbert curve and let N (typically, $N \in \{4, 8, 16, \dots\}$) adjacent elements on the space-filling curve form a cluster. Those clusters are, of course, much less optimized than the exact BVH heuristic by Wald *et al.* [WMZ21] can achieve. The authors combine that with similar measures also taken by Wald *et al.*, such as meshlets to reduce the element index precision or element sorting to group elements of the same type together. In contrast to Wald's OptiX-accelerated multi-leaves, Morrical *et al.* traverses the N-sized clusters linearly, resulting in better memory yet lower sampling performance. Compared to Wald *et al.*'s method, this method trades sampling performance for pre-processing times.

Morrical *et al.* [MSG*22] also initiated what we think might become a trend—to use Woodcock tracking (cf. Algorithm 3 in Appendix A) instead of ray marching to compute transmission estimates. The authors do not use this unbiased sampling method to implement a full multi-scattering path tracer but instead evaluate a primary absorption and emission sample, plus an optional shadow ray towards a point light source. Apart from apparent trade-offs (fewer samples being taken for single pixel samples resulting in higher variance, but no bias when converged), an interesting property is that, through the use of local majorants (cf. Section 2), adaptive sampling does not have to rely on heuristics such as variance estimates. Instead, the number of (rejection) samples taken by Algorithm 3 in Appendix A is directly proportional to the local majorant density.

In summary, the methods presented in this section, including the marching and object-order approaches, BVH point location, and acceleration structure compression schemes, present several different trade-offs. The literature still lacks a rigorous evaluation comparing all the aspects of pre-processing times, memory consumption, and sampling and traversal performance within the same software framework. What we can distill from the literature and results presented therein is, however, that marching-based approaches often have a superior memory footprint and linear sampling times, whereas BVH point queries can perform cell location in hardware and logarithmic time, yet at the expense of storing an extra acceleration structure in memory. Compression schemes can bring memory costs down significantly, but their implementation is not trivial. BVH sampling seems slightly more flexible because it can more easily deal with degenerate geometry, such as duplicate shared faces, where a marcher would run into endless loops or prematurely break out the traversal loop.

6. Compressed and Neural Representations ●

Adaptive representations, some of which have been mentioned previously, have become indispensable for the interactive analysis and visualization of scientific data. A primary goal of these representations is to reduce the memory footprint and processing costs of massive data without any perceptible degradation in the visualization quality or the analysis results. The approaches we presented so far are mainly limited to multi-resolution structures such as octrees or k-d trees. However, these multi-resolution representations involve structural constraints, sometimes leading to unnecessary refinement in unimportant regions, primarily when the domain is characterized by a non-rectangular shape, such as strong and thin thread-like variations or L-shapes. The data size can also be reduced by reducing its precision, e.g., the number of bits for encoding numerical values. Recent work by Hoang *et al.* [HSB*21] and Bhatia *et al.* [BHM*22] have shown that the combination of these two, usually separate, concepts, namely resolution adaptation and precision reduction, can offer significant benefits in terms of storage reduction and/or quality versus efficiency tradeoff. We first review these two works in the following subsection. Then, we review recent works that base data size reduction on neural networks and deep learning.

6.1. Unified resolution–precision compressed representation

Multi-resolution-based techniques implement levels of details that often express themselves in the form of a tree, a set of wavelet coefficients, and/or multi-level space-filling curves. Precision-based techniques, on the other hand, generally apply quantization to truncate lower-order precision levels. Both types of methods achieve significant reductions in memory footprint. However, working individually with either approach restricts the achievable gains because they do not localize precision, meaning they either preserve all bits for some values or a few bits for all. Hoang *et al.* [HSB*21] introduce a tree structure and data layout aiming to encode precision and multi-resolution simultaneously. There are other attempts to do so. Indeed, a combination of the two is also used by VAPOR [LJP*19], a domain-specific visualization package that targets the analysis of simulation data arising from Earth system science (ESS). It uses wavelets for compression in combination with precision levels predetermined during the compression stage. When accessing the data, the tool can select an increasing number of wavelet coefficients as the user increases the quality and/or resolution levels, thereby reducing I/O and computational costs. However, the quality levels correspond to the number of wavelet coefficients to be decoded and do not allow for direct control of the data precision. The fact that only a few, generally four, precision levels are supported limits control, which the authors of [HSB*21] propose to remedy.

[HSB*21] argues that considering precision (number of bits) and resolution (wavelet coefficients) together in a single unified tree structure allows navigating more freely in a two-dimensional data encoding space. This freedom permits improving the quality and time ratio for regions of interest (ROI) queries. An ROI query defines a spatial sub-part of the whole dataset, enhanced with precision error and level of resolution. The tree is built from a regular grid of points. Unlike other multi-resolution trees, it has the same number of nodes as points. Figure 7 illustrates this in a simple 4×4 2D case. First, a multi-resolution tree (see Figure 7-b) is built from the regular data (Figure 7-a) using the notion of Z index, which is formed by the bit interleaving of the point coordinates. The figure illustrates this through the numbering of points. The bits of the Z code are partitioned into prefix and suffix bits, from which a node index corresponding to a breadth-first tree traversal order, a node level, and the parent node index can be straightforwardly computed. This tree encodes data in a multi-resolution form, e.g., using CDF 5/3 wavelet coefficients. Floating point data values are expressed as a product between a power of two integer exponent and an integer quantized value. The bits of these integers are clustered into bitplanes, so the wavelet coefficients are also mapped into a fixed number of bitplanes. Each node of the previous tree is then extended by a linked list of bitplanes, used to control data precision, as shown in Figure 7-c. As for other multi-resolution tree structures, a query corresponds to a cut in the tree. Here, all nodes above the cut define an approximation.

Once constructed, the tree is stored in a way that optimizes minimal access time, storage costs, and transfer costs. Therefore, the tree is structured into blocks, bricks, chunks, and files are put into hierarchies of directories. Each brick is encoded independently into its own local tree to permit parallelism. Data dependencies among neighboring bricks, related to the wavelet coefficients, are managed using lifting-based linear extrapolation. The local trees are also merged into a single tree to reach any desired level of hierarchy. The bricks are thus hierarchically structured using the same Z code for data points. A brick is composed of contiguous blocks, which are compressed. All data is stored on disk in files composed of chunks. Given some requested tree nodes, a function that maps a corresponding file ID using bit packing and the file path is applied iteratively over the corresponding levels and bit planes. The authors show that using this tree structure and disk layout achieves

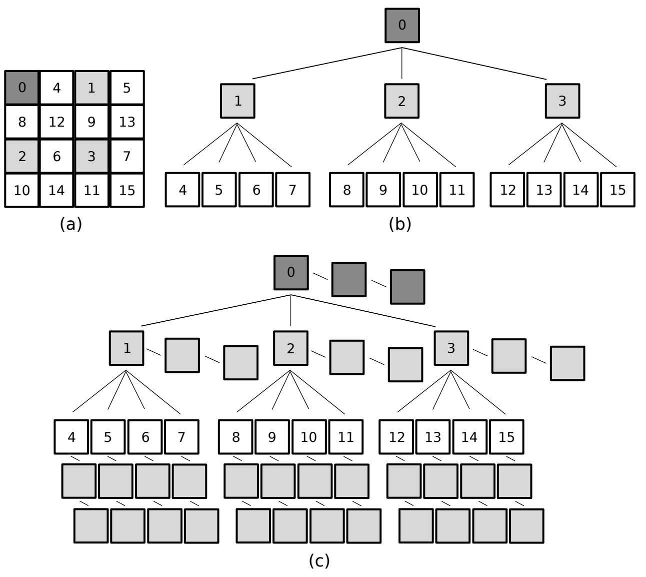

Figure 7: *Unified tree [HSB*21] example for a 4×4 dataset (a); (b) the wavelet coefficient tree using the Z index; (c) extension by a linked list of bitplanes to encode precision.*

high compression rates on large datasets and fast random access of ROI, provided that parameters are chosen correctly. Parameter optimization has been achieved through the execution of various queries on the dataset. In an exhaustive study, the authors compare their compression method to the most recent techniques, namely SZ [TDCC17], TTHRESH [BRLP20], JPEG2000 [TM13], ZFP [Lin14], as well as VAPOR [LJP*19]. They show that the decompression times and memory usage are several orders of magnitude lower than SZ and TTHRESH and that the data quality is competitive with them for a compression ratio of the order of 300 and is only slightly lower than TTHRESH at very high compression ratios. The authors compare the quality of their near-lossless compression technique with other methods using the peak signal-to-noise ratio (PSNR) and structural similarity index measure (SSIM). They also show that their method achieves ZFP's decoding speed while enabling very high compression ratios. For mid- and low-quality levels, their system can decode at lower resolutions and, therefore, achieve lower decoding time. For a compression ratio on the order of 300, their method also achieves better quality than VAPOR while avoiding blocking artifacts at very low bit rates.

In [BHM*22], an approach called *Adaptive Multilinear Meshes (AMM)*, is presented. It is also a mixed representation that allows to vary spatial precision and reduce the in-memory footprint while generating, storing, and accessing data at a reduced resolution and precision. Unlike existing multi-resolution approaches, they enable varying the precision spatially. AMM is a compact and adaptive representation of piecewise multilinear scalar fields utilizing a tensor-product wavelet basis (linear biorthogonal B-splines) [WL16]. It uses block-based mixed precision coding for the vertex values, which provides a data reduction superior to spatial adaptivity alone. It is also the first adaptive representation that can be incrementally updated using arbitrarily ordered data streams, offering new opportunities to explore dynamic and hybrid data reduction strategies. AMM provides flexible adaptivity in the representation of uniformly regular scalar data through a new type of spatial hierarchy that implements, through pointerless representation, more general subdivision operations than those used on existing tree-based hierarchies, i.e., octrees and k-d trees. The data structure reduces the representation's size through two contributions: (1) rectangular cubical cells, which ensure that a tree node is subdivided only along the required axes, and (2) improper nodes, which facilitate partial division and representation.

The construction of an AMM is associated with two adjacent refinement levels with a "spatial stencil," indicating how to combine the different wavelet coefficients. Thus, stencil vertices are "stamped" into a staging phase and combined with the vertices from coarser and finer levels later in the upstaging step. Regarding performance (on the spatial hierarchy, mixed-precision representation, and incremental updates), AMM does not offer a competitive compression rate compared to state-of-art compressors, such as ZFP or SZ. However, this approach has natural adaptation capabilities by reducing the number of vertices and cells in the representation, thus offering significant computational advantages during traversal. Current work demonstrates about 50% faster volume rendering and produces significantly smaller meshes (20–50% gain) for the same data quality. The authors evaluate the quality of their mixed-precision representation according to different ways of streaming the coefficients of the compression function. For this, they rely on the evaluation of the peak signal-to-noise ratio (PSNR). They show PSNR values between 40 and 100 dB for resolution stream techniques (transmits complete coefficients) and values between 80 and 240 dB for precision stream (transmit bits of a coefficient separately). It also raises several questions regarding performance, most notably the stamping process for vertices that requires tree traversals. It is the crucial bottleneck in further scaling and its use on sparse data.

6.2. Compression using neural networks

With the advent of learning techniques, mainly based on neural networks, it seems relevant to try to exploit this approach to reduce the data size, which is what Lu *et al.* [LJLB21] propose to do. They introduce a learning-based method for compressing volumetric scalar fields. Their approach is based on exploiting neural networks that map a continuously-defined position from the simulation domain to a scalar value [PFS*19, MON*19]. Their motivation is that a neural network learning such a mapping function places no assumptions on the data characteristics. Placing assumptions on data is a disadvantage of transformation-based compressive representations (e.g., Fourier or Wavelet bases), whose assumptions may not be valid for the volumetric scalar field under consideration.

They limit the network's capacity so that the number of network weights is less than the volume resolution. This way, they obtain optimized compressive representations that approximate the scalar field at the sampled values. They build their neural network based on SIREN [SMB*20], which has fully-connected layers and sinusoids as periodic activation functions. They use residual (skip) connections to achieve the robustness of the network. The network weights are distributed so they can be quantized with a small number of bits.

		Tool type		Data type			Platform		
Tool	License	API	Renderer	Structured	Unstruct.	AMR	CPU	GPU	Cluster
OSPRay	Apache	✓		✓	✓	✓	✓		✓
VisRTX	Proprietary	✓		✓				✓	
VisIt-OSPRay	Apache		✓	✓		✓	✓		✓
VisIt	BSD		✓	✓	✓	✓	✓	✓	✓
ParaView	BSD		✓	✓	✓	✓	✓	✓	✓
IndeX ®	Proprietary	✓	✓	✓	✓			✓	✓

Table 2: *Major sci-vis tools and their support for direct volume rendering. We distinguish by whether the tool is primarily a renderer (user-centric) or an API (developer-centric), the volume data types they support, and the platform targeted by their rendering component.*

They use a PSNR quality metric to justify their choices throughout the article. They also provide a quality comparison against TTHRESH on single time step grids and time-varying datasets using PSNR quality metric over several compression ratios. The comparison shows that the neural network method is overall an improvement over TTHRESH, except for one dataset when low compression ratios are applied. The method produces compressed representations approximately half the size of TTHRESH and provides smoother scalar fields. The authors also report that they obtain larger gains in performance the higher the compression ratio. This suggests that the method remains robust and can still obtain good approximations of scalar fields, even when a few parameters are kept in the network.

In addition to the approaches mentioned above, some other notable studies on compressed and neural representations are as follows. Pan *et al.*. [PZGY19] describe an adaptive deep learning-based approach for compressing time-varying volumetric data using an autoencoder-based neural network with quantization and adaptation. Weiss *et al.*. [WHW22] propose using GPU tensor cores to design scene representation networks. They integrate the reconstruction process into on-chip raytracing kernels. Jain *et al.*. [JGG*17] use a deep convolutional autoencoder network to achieve a data-driven representation for volumetric data by learning high-level hierarchical features. Quan *et al.*. [QCJJ18] introduce a learning-based voxel classification technique for volume rendering. Their approach uses hierarchical multi-scale 3D convolutional sparse coding to learn dictionary-based features from the input data. Mensmann *et al.*. [MRH10] exploit the CUDA computing architecture for a hybrid CPU/GPU scheme for lossless compression and data streaming of time-varying volumetric data. Their approach utilizes temporal coherence and variable-length coding with a fast block compression algorithm. Weiss and Navab [WN21] integrate deep neural networks into direct volume rendering. Instead of designing features explicitly and manually crafting transfer functions, they learn mappings for these processes by representing the rendering process in a latent color space.

Because this is a fast-growing field and many new developments are on the way, we convey the reader to the survey of Wang and Han [WHss] on deep learning-based approaches for scientific visualization for extensive coverage.

7. Tools for Volume Rendering

In addition to the recent works on large-scale volume visualization, we also review the common visualization tools and libraries (APIs) for these data types. Table 2 summarizes the visualization tools' and APIs' support for various data types. The following sections review and discuss how tools and APIs support the data types and large-scale visualization techniques discussed in this survey.

7.1. Large-scale visualization APIs

OSPRay [WJA*17] is a CPU-based ray tracing framework for scientific visualization. It is a scene-description library where developers describe the scene using *OSPRay* procedures rather than specifying the rendering process. Since it uses ray tracing, it supports shading effects, such as ambient occlusion and shadows. In addition, *OSPRay* contains an MPI module for distributed volume rendering with a sort-last compositor to render large-volume data across HPC clusters. The *OSPRay* framework relies on high-performance *Open Volume Kernel Library (OpenVKL)* [Int22] for rendering. *OpenVKL* is a collection of high-performance volume computation kernels that provides point sampling, volume traversal, and isosurface hit kernels for structured, unstructured, and block-structured AMR meshes. *OpenVKL* accelerates the spatial queries using *Embree* [WWB*14] ray tracing kernels and provides *current*, *finest*, *octant* reconstruction methods for AMR volume. The *octant* reconstruction method allows to reconstruct crack-free isosurfaces with smooth boundaries for block-structured AMR meshes. *OpenVKL* also supports OpenVDB [Mus21] meshes, an open-source library composed of a hierarchical data structure and a set of tools to store and manipulate sparse volumetric data discretized on 3D grids. OpenVDB was developed by DreamWorks Animation and is currently maintained by the Academy Software Foundation (ASWF).

Khronos' *ANARI* [SGA*22] is a high-level scientific visualization API designed to connect low-level renderers like *OSPRay* and *VisRTX* [Ams19] with front-end visualization tools like *VisIt* [CBW*12] and *ParaView* [AGL05]. Inspired by *OSPRay*, *ANARI* is a scene description library. Renderers implementing the *ANARI* specification provide high-level scene description procedures for developers to let them build cross-platform visualization tools that use rasterization or ray tracing. *ANARI* makes it easier for developers to implement visualization tools. By using *ANARI* specification, visualization tools can use different scientific renderers without implementing different renderer bindings or their own low-level rendering engines. Any visualization tool using *ANARI* can use *OSPRay* and *VisRTX*. *VisRTX* [Ams19] is an implementation of the *ANARI* specification targeted for RTX-enabled NVIDIA GPUs. Moreover, by providing new implementations to *ANARI*, researchers can easily incorporate their state-of-the-art visualization techniques into any existing visualization tool that supports *ANARI*. The interested reader may refer to [SGA*22] for an excellent overview of the 3D *ANARI* API Standard and its positioning concerning visualization applications, frameworks, and tools, such as *ParaView* [AGL05], *VisiT* [CBW*12], and VTK [Kit22], render-

Tool	Data Type	Supported File Format
VisIt	Structured	*bov*, *conduit*, *cosmos*, *pvti*, *pvtr*, *pvts*, *silo*, *visit*, *vtk*, *vti*, *vtr*, *vts*, *xdmf*
	AMR	*conduit*, *cosmos*, *silo*, *visit*
	Unstructured	*conduit*, *cosmos*, *pvtu*, *silo*, *visit*, *vtk*, *vtu*, *xdmf*
ParaView	Structured	*dem*, *dicom*, *inp*, *mrc*, *netcdf*, *pvd*, *pvti*, *pvtr*, *pvts*, *raw*, *tecplot*, *vti*, *vtk*, *vtr*, *vts*, *windblade*, *xdmf*
	AMR	*ensight*, *enzo*, *flash*, *openfoam*
	Unstructured	*exodus*, *fluent*, *mfix*, *netcdf*, *prostar*, *pvd*, *pvtu*, *TecPlot*, *vtk*, *vtu*, *xdmf*

Table 3: *Supported file types of visualization tools.*

ers, such as *OSPRay* [WJA*17], *VisRTX* [Ams19], and acceleration APIs, such as *OptiX* [NVI09b].

7.2. Large-scale visualization tools

VisIt [CBW*12], *ParaView* [AGL05], and *IndeX* [NVI09a] are common scalable tools that are used for visualizing large volumes. In this section, we review the distributed tools showing how the tools deal with the different kinds of data. Table 3 shows the file types these tools support.

7.2.1. Tools for structured data

Due to the popularity of structured grids, most volume visualization tools support this data representation. *VisIt* [CBW*12] is a popular distributed, parallel visualization tool used for 2D and 3D volume visualization. It supports various mesh types, including curves, rectilinear, curvilinear, and unstructured, as well as point-based and molecular data. *VisIt* supports various CPU and GPU rendering algorithms to visualize the volumetric data (structured and unstructured). For GPU architectures, it uses *splatting* and *3D Texture* rendering methods. These methods require a considerable amount of memory. Therefore, *VisIt* downsamples the volume to fit the data into GPU VRAM. *VisIt* also supports three ray-casting-based CPU rendering algorithms: *Compositing*, *SLIVR* and *VisIt-OSPRay* [WUP*18]. *Compositing* renderer [CDM06] is the predecessor of Binyahib *et al.*'s [BPL*19] hybrid rendering algorithm. Unlike Binyahib *et al.*'s algorithm, *Compositing* renderer does not support partial compositing of the intermediate samples. Therefore, *compositing* method requires a considerable amount of memory compared to other methods. *SLIVR* implements a sort-last rendering pipeline with parallel direct send [Hsu93]. *VisIt-OSPRay* uses fast, multi-core *OSPRay* renderer to render the structured grids. *VisIt-OSPRay* utilizes the sort-last compositing library *IceT* [MKPH11]. Unlike *Compositing* and *SLIVR* renderers, *VisIt-OSPRay* employs shared-memory parallelism by using multi-threading, and therefore *VisIt-OSPRay* is more memory efficient.

Another popular visualization tool is *ParaView* [AGL05]. *ParaView* is an open-source suite of tools for the interactive manipulation of scientific visualization data by Kitware Inc. *ParaView* is developed together with the Visualization Toolkit (VTK) [SML06, Kit22], which is an open-source for data manipulation and rendering. *ParaView* uses client-server architecture, similar to *VisIt* and it supports rendering algorithms for *CPU* and *GPU* architectures. *ParaView* distributes the large models into multiple nodes, and similar to *VisIt-OSPRay*, *ParaView* uses sort-last compositing library, IceT, to composite the individually rendered partial images. For structured volumes, *ParaView* uses three renderers: ray casting-based CPU and GPU renderers and *OSPRay* renderer. VTK-m [MSU*16, Mor23] is a complementary tool to ParaView and VTK that provides additional support for developing parallel scientific visualization algorithms. VTK-m makes it easier to create scientific visualization algorithms that run in parallel by offering a range of supportive tools and features. VTK-m provides efficient data transfer and processing tools on various parallel architectures, including multi-core CPUs, GPUs, and distributed computing clusters. Additionally, it provides a lightweight rendering module. However, it is important to note that the rendering package in VTK-m is not meant to function as a complete rendering system or library [Mor23]. Some of the VTK-m features have been integrated into *VTK* and *ParaView*, making it easier for users to take advantage of VTK-m's capabilities within these popular visualization tools..

IndeX® [NVI09a] is a 3D volumetric interactive visualization software development kit that exploits GPU clusters. *IndeX* provides functionality to visualize structured and unstructured meshes. *IndeX* also provides a plug-in that brings scalable visualization capabilities to *ParaView* [NVI20].

7.2.2. Tools for AMR data

ParaView resamples and converts the AMR volume into a structured grid to render the volume. This method allows AMR data to be rendered without using a different rendering algorithm. While resampling, *Paraview* allows the user to select the grid size. If the grid size is too small, some features will be undersampled. On the other hand, increasing the grid size requires more memory. AMR sampling in *Paraview* does not address the T-junction problem: The resulting isosurface might not be smooth and crack-free.

VisIt use Weber *et al.*'s parallel "stitching" algorithm [WCM12] (cf. Section 4) for visualizing AMR data. Weber *et al.*'s "stitching" algorithm converts the AMR data into an unstructured mesh. After the stitching, these tools use their unstructured rendering algorithms to visualize the AMR meshes. Weber *et al.*'s stitching algorithm generates crack-free, smooth isosurfaces when only transitions of one level between neighboring cells exist. With stitching *VisIt* supports crack-free, smooth surfaces at the cost of increasing the memory footprint.

Additionally, *VisIt* and *ParaView* can use *OSPRay*'s rendering capabilities to render the block-structured AMR mesh. Since *OSPRay*'s octant reconstruction method does not convert the AMR mesh into an unstructured grid, it requires less memory than the "stitching" method.

7.2.3. Tools for unstructured data

VisIt uses the same algorithms (splatting, 3d texture, compositing, SLIVR, *VisIt-OSPRay*) for structured and unstructured meshes. *Splatting*, *compositing* and *SLIVR* algorithms utilize cell/tetrahedron projection technique. *3D texture* method converts the data into a structured mesh. Then the converted mesh is rendered with a structured grid rendering method. Projected

	Platform		Architecture			Traversal		Sampling	
	CPU	GPU	in-core	out-of-core	distributed	software	hardware	hierarchical	direct/linear
● [HAAB*18] [BMA*19] [SCRL20]		✓		✓		✓		✓	
● [WWJ19]	✓			✓		✓		✓	
● [GM19] [WZM21]		✓	✓				✓		✓
● [ZSL21]	✓	✓	✓			✓			✓
● [WBUK17] [WWW*19] [WMU*20]	✓		✓			✓		✓	
● [WZU*21] [ZWS*22a] [ZSM*22]		✓	✓				✓		✓
● [ZWS*22b]		✓	✓			✓	✓	✓	✓
● [MHDH07] [MHDG11]		✓	✓			✓			✓
● [SDM*21]		✓	✓				✓		✓
● [BPL*19]		✓			✓				✓
● [SDW*22]		✓			✓		✓		✓
● [WUM*19] [MUWP19] [WMZ21] [MWUP22] [MSG*22]		✓	✓				✓	✓	
● [HSB*21] [BHM*22]				✓				✓	
● [LJLB21]		✓	✓						

Table 4: *Comparison of volume visualization techniques for structured (●), AMR (●) and unstructured (●) data, and compression and neural representation methods (●). Volume traversal technique is expressed according to whether it is based on hardware acceleration or on a software solution. Hierarchical sampling indicates that this step relies on traversing a hierarchical data structure, whereas direct/linear methods do not.*

cell/tetrahedron and *3D texture* methods either under-sample the volume or require a high amount of memory. *VisIt-OSPRay* uses *OSPRay* library to visualize the unstructured mesh. *OSPRay* uses *use point ray sampling* method for visualization.

ParaView provides four different algorithms for visualizing unstructured meshes: projected-tetra, Z-sweep, Bunyk Raycast [BKS97], *OSPRay raycaster*, *OSPRay path tracer* and *resample to image* methods. Projected-tetra, Z-sweep use cell/tetrahedron projection method, *resample to image* method converts the unstructured mesh into structured one. *Bunyk Raycast* is a *element marching*-based method that renders the unstructured volume by traversing the cells.

8. Discussion and Comparison of the Surveyed Approaches

The algorithms and techniques we surveyed fall into different camps depending on the volume data type, as reflected by the overall outline of this state-of-the-art report. Most papers these days focus on general-purpose ray tracing techniques that can trace individual rays through the volume density without using a rasterization API. This allows for fine-grained control over the path taken by those individual rays, largely independent of the behavior of neighboring rays. Such a fine-grained way of scheduling rays naturally allows for space-skipping optimizations—hence we orthogonally classify the techniques based on their focus on traversal or sampling. We also survey trends on techniques to cope with very large data. Standard measures one can take in that case is out-of-core visualization, where a lower level of the memory hierarchy is integrated into the system to asynchronously serve data pages required by the rendering subsystem, as well as massive parallelism. We broadly classify the surveyed papers according to these axes and present our findings in Table 4.

We observe several trends. Naturally, sampling involving the location of individual cells in the volume is a more complex operation the more unstructured the data is. Sampling is a trivial operation inside a 3D texture or similar, directly addressable memory segments as they are involved when rendering structured, regular volumes (cf. Section 3). This, however, changes as soon as the memory hierarchy gets involved, as in papers by Hadwiger *et al.* [HAAB*18], Beyer *et al.* [BMA*19], Wang *et al.* [WWW*19], or Sarton *et al.* [SCRL20], concentrating on out-of-core techniques. In that case, sampling can still become a dominating operation.

In the case of semi-structured, cell-centric AMR data (cf. Section 4), smooth sampling at level boundaries was not possible until recently for large data sets. Stitching [ME11] was the agreed-upon solution for smooth reconstruction at level boundaries. However, it proved too memory intensive because the hierarchical aspect of the data was abandoned, and the AMR cells were converted to individual unstructured cells. Papers re-investigating the problem of high-quality reconstruction on the original and not the dual AMR grids thus naturally focused on sampling, which, as opposed to structured volume rendering, now requires complete hierarchy (e.g., k-d tree, BVH) traversal to locate individual cells. Approaches presented treat reconstruction as weighted average computations [WBUK17] and require all the cells in a given neighborhood to be located. More complex algorithms such as the one by Wang *et al.* [WWW*19] focus on sampling with a rectilinear filter. This approach involves an even more complex operation, where an octant space decomposition of the neighborhood is recursively found, involving cell location and implicit sorting of dual cells based on that space decomposition. With these methods established, follow-up papers identified the problem that AMR sampling is expensive. Then they proposed to use more intricate traversal schemes to allow for space skipping and adaptive sampling [WZU*21, ZWS*22a, ZWS*22b].

Sampling can become even more costly when neighborhood

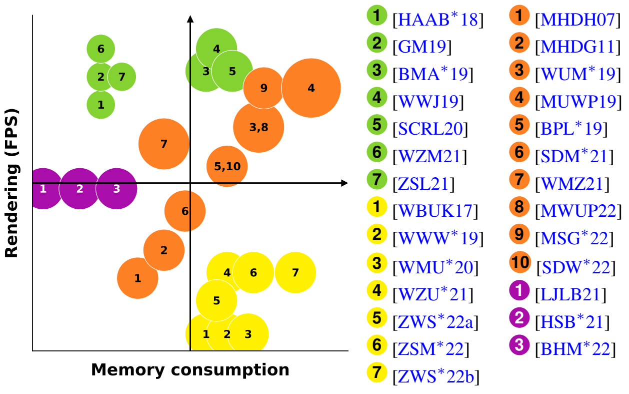

Figure 8: *Classification of the different methods in terms of rendering performance according to the memory occupation. The size of the circles indicates the pre-processing time needed. Note that distances between methods are not quantifiable and should rather be considered rough indications. Since the compression and neural representation approaches (●) do not include rendering methods, they are represented in a neutral position on the y-axis, and their evaluation concerns compression rate (x-axis) and compression time (circle sizes).*

and connectivity relations are entirely missing from the data, as in the case of unstructured meshes. In that case, two orthogonal approaches exist. One way to sample the data is to *reconstruct* the connectivity information and transform the data to support fast one-ring neighborhood queries [MHDG11, SDM*21]. Another notable approach builds a hierarchical acceleration structure over the data that allows for fast point-in-cell queries; it can be more memory intensive but has seen recent popularity due to the introduction of hardware ray tracing cores [MUWP19, MWUP22]. Our survey reveals that hardware ray tracing is limited if the data gets large. Compression schemes such as the one proposed by Wald *et al.* [WMZ21] require more control over the data layout of the internal nodes of the acceleration structure so that complete hardware-accelerated sampling is no longer possible.

A trend we observe is that data sampling workloads become increasingly incoherent. General-purpose ray tracing techniques allow one to trace secondary shadow or scattering rays into the volume [SDM*21]. Representations optimized for volume rendering are sometimes also used for arbitrary sampling, e.g., to compute flow visualizations [ZSM*22], which involves more divergent memory access patterns. Finally, we observe a recent trend that path tracing is adopted by the sci-vis community [MSG*22, ZWS*22b], which is most likely driven by the recent availability of high-quality denoising techniques [IGMM22] and requires sample placement at arbitrary positions. Taking individual samples becomes even more costly with this approach, so efficient volume traversal data structures become even more critical. Fig. 8 shows the balance between rendering time and memory usage as well as pre-processing times of the different methods for all the data types we mentioned.

Lastly, we observe a recent trend toward compressing given data representations. While not all of the papers focus on sampling and traversal of the volume data [HSB*21, BHM*22], an even more recent trend employs extremely lossy compression by encoding the volume as a neural network [WDBM22]. This approach, which borrows from recent development in photogrammetric rendering [MST*21], has extraordinarily high sampling costs, as each sample taken involves inferring a neural net. In that case, photogrammetric approaches break down the neural net into several smaller neural nets [RPLG21], which form a uniform grid—or, following our terminology—a traversal data structure. This up-and-coming field is currently still establishing itself [WDBM22], and yet, it is evident that the need for fast volume traversal is even more critical in this case.

Our review of tools supporting all those techniques shows a gradual shift towards general-purpose ray tracing. This trend is reflected by the developments concerning ANARI [SGA*22], which, although a general rendering API, has strong ties to ray tracing since its software design is highly inspired by OSPRay's [WJA*17]. Well-established tools like ParaView [AGL05] or VisIt [CBW*12] start integrating both OSPRay [WUP*18] as well as ANARI [SGA*22]. With that, even these tools that still depend on rasterization techniques are gradually adopting general-purpose ray tracing. The use of libraries like OpenVKL [Int22] opens the door for full path tracing volume rendering to be adopted by these tools.

9. Conclusion

In this state-of-the-art report, we have focused on recent developments in direct volume rendering techniques for efficient interactive visualization of large-scale volume data. We established a classification of the papers based on their volume data type. Sampling is the primary operation of reconstructing values at discrete positions and generally becomes more costly the less regular the data is. In that case, intricate traversal data structures are used to adapt the sampling rate or skip over empty space. We generally observe that data structures adapt more aggressively to how much the data frequency varies during simulation or data acquisition. That results in hierarchical and less structured volume data. While reducing the memory overhead by avoiding to represent the data with empty or homogeneous cells, visualization algorithms need to be implemented carefully to not add in memory what was gained, by introducing additional auxiliary data structures for sampling and traversal. Another trend we observe is the availability of GPUs with hardware-accelerated ray tracing. Algorithms make ample use of that, so that storing BVHs along with the volume is becoming a commodity. These hardware features help with flexibility, as it is now easier to traverse individual rays and implement more complex lighting models. At the same time, using them poses challenges in terms of memory consumption and interactive changes to the data or transfer functions, as these become more costly in the presence of data structures like BVHs that need to be rebuilt each frame.

Acknowledgments

We acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)——grant no. 456842964, and the French National Agency with LUM-Vis project—ANR-21-CE46-0005.

References

[Adv23] ADVANCED MICRO DEVICES, INC.: Vulkan® ray tracing extension support in our latest AMD Radeon™ Adrenalin driver 20.11.3, 2023. Available at https://gpuopen.com/vulkan-ray-tracing-extensions/, Accessed: 6 February 2023. 4

[AGL05] AHRENS J. P., GEVECI B., LAW C. C.: ParaView: An end-user tool for large-data visualization. In *The Visualization Handbook*, Hansen C. D., Johnson C. R., (Eds.). Academic Press / Elsevier, 2005, pp. 717–731. doi:10.1016/b978-012387582-2/50038-1. 11, 13, 17, 18, 20

[AMR18] AMREX TEAM: AMReX Adaptive Mesh Refinement Framework. https://amrex-codes.github.io/amrex/, 2018. Accessed: 7 February 2023. 2

[Ams19] AMSTUTZ J.: VisRTX, 2019. Available at https://github.com/NVIDIA/VisRTX, Accessed: 7 February 2023. 2, 17, 18

[BC89] BERGER M. J., COLELLA P.: Local adaptive mesh refinement for shock hydrodynamics. *Journal of Computational Physics 82*, 1 (1989). 2, 7

[BHM*22] BHATIA H., HOANG D., MORRICAL N., PASCUCCI V., BREMER P.-T., LINDSTROM P.: AMM: Adaptive Multilinear Meshes. *IEEE Transactions on Visualization and Computer Graphics 28*, 6 (2022), 2350–2363. 2, 15, 16, 19, 20

[BHP15] BEYER J., HADWIGER M., PFISTER H.: State-of-the-art in GPU-based large-scale volume visualization. *Computer Graphics Forum 34*, 8 (2015), 13–37. 2, 4, 6

[BKS97] BUNYK P., KAUFMAN A., SILVA C.: Simple, fast, and robust ray casting of irregular grids. In *Scientific Visualization Conference (dagstuhl '97)* (1997), pp. 30–30. doi:10.1109/DAGSTUHL.1997.1423099. 19

[BMA*19] BEYER J., MOHAMMED H., AGUS M., AL-AWAMI A. K., PFISTER H., HADWIGER M.: Culling for extreme-scale segmentation volumes: A hybrid deterministic and probabilistic approach. *IEEE Transactions on Visualization and Computer Graphics (Proceedings IEEE Scientific Visualization 2018) 25*, 1 (2019), 1132–1141. 6, 19, 20

[BO84] BERGER M. J., OLIGER J.: Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics* (1984). 7

[BPCS06] BERNARDON F. F., PAGOT C. A., COMBA J. L., SILVA C. T.: Gpu-based tiled ray casting using depth peeling. *Journal of Graphics tools 11*, 4 (2006), 1–16. 11

[BPL*19] BINYAHIB R., PETERKA T., LARSEN M., MA K.-L., CHILDS H.: A scalable hybrid scheme for ray-casting of unstructured volume data. *IEEE Transactions on Visualization and Computer Graphics 25*, 7 (2019), 2349–2361. 2, 11, 12, 13, 18, 19, 20

[BRLP20] BALLESTER-RIPOLL R., LINDSTROM P., PAJAROLA R.: TTHRESH: Tensor compression for multidimensional visual data. *IEEE Transactions on Visualization and Computer Graphics 26*, 9 (2020), 2891–2903. doi:10.1109/TVCG.2019.2904063. 16

[BWG11] BURSTEDDE C., WILCOX L. C., GHATTAS O.: p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing 33*, 3 (2011). doi:10.1137/100791634. 7

[BWWA18] BENTHIN C., WALD I., WOOP S., ÁFRA A. T.: Compressed-leaf bounding volume hierarchies. In *Proceedings of the Conference on High-Performance Graphics* (New York, NY, USA, 2018), HPG '18, Association for Computing Machinery. doi:10.1145/3231578.3231581. 14

[CBW*12] CHILDS H., BRUGGER E., WHITLOCK B., MEREDITH J., AHERN S., PUGMIRE D., BIAGAS K., MILLER M., HARRISON C., WEBER G. H., KRISHNAN H., FOGAL T., SANDERSON A., GARTH C., BETHEL E. W., CAMP D., RÜBEL O., DURANT M., FAVRE J. M., NAVRÁTIL P.: VisIt: An end-user tool for visualizing and analyzing very large data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. Chapman and Hall/CRC, New York, NY, Oct 2012, pp. 357–372. 12, 13, 17, 18, 20

[CDM06] CHILDS H., DUCHAINEAU M. A., MA K.: A scalable, hybrid scheme for volume rendering massive data sets. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (2006), Heirich A., Raffin B., dos Santos L. P. P., (Eds.), EGPGV 06, Eurographics Association, pp. 153–161. 12, 18

[CGL*00] COLELLA P., GRAVES D., LIGOCKI T., MARTIN D., MODIANO D., SERAFINI D., VAN STRAALEN B.: Chombo Software Package for AMR Applications Design Document, 2000. 2, 7

[Cra11] CRASSIN C.: *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. PhD thesis, Universite De Grenoble, July 2011. URL: http://maverick.inria.fr/Publications/2011/Cra11. 6

[DAC*14] DUBEY A., ANTYPAS K., CALDER A. C., DALEY C., FRYXELL B., GALLAGHER J. B., LAMB D. Q., LEE D., OLSON K., REID L. B., RICH P., RICKER P. M., RILEY K. M., ROSNER R., SIEGEL A., WEIDE N. T. T. K., TIMMES F. X., VLADIMIROVA N., ZUHONE J.: Evolution of FLASH, a multi-physics scientific simulation code for high-performance computing. *The International Journal of High Performance Computing Applications 28*, 2 (2014), 225–237. 2, 7

[DHTI21] DERIN M. O., HARADA T., TAKEDA Y., IBA Y.: Sparse volume rendering using hardware ray tracing and block walking. In *SIGGRAPH Asia 2021 Technical Communications* (New York, NY, USA, 2021), SA '21, Association for Computing Machinery. doi:10.1145/3478512.3488608. 4

[Eng11] ENGEL K.: CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard PCs. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization* (2011), LDAV '11, pp. 123–124. doi:10.1109/LDAV.2011.6092330. 6

[FFMW20] FELLEGARA R., FLORIANI L. D., MAGILLO P., WEISS K.: Tetrahedral trees: A family of hierarchical spatial indexes for tetrahedral meshes. *ACM Transactions on Spatial Algorithms and Systems (TSAS) 6*, 4 (2020), 1–34. 14

[Flo03] FLOATER M. S.: Mean value coordinates. *Computer aided geometric design 20*, 1 (2003), 19–27. 12

[FN80] FRANKE R., NIELSON G.: Smooth interpolation of large sets of scattered data. *International Journal for Numerical Methods in Engineering 15*, 11 (1980), 1691–1704. doi:https://doi.org/10.1002/nme.1620151110. 8

[FWDF21] FELLEGARA R., WEISS K., DE FLORIANI L.: The stellar decomposition: A compact representation for simplicial complexes and beyond. *Computers & Graphics 98* (2021), 322–343. 14

[Gar90] GARRITY M. P.: Raytracing irregular volume data. *Computer Graphics (San Diego Workshop on Volume Visualization) 24(5)* (1990), 7. 11

[GM19] GANTER D., MANZKE M.: An analysis of region clustered BVH volume rendering on GPU. *Computer Graphics Forum 38*, 8 (2019), 13–21. 5, 19, 20

[GMH*19] GEORGIEV I., MISSO Z., HACHISUKA T., NOWROUZEZAHRAI D., KŘIVÁNEK J., JAROSZ W.: Integral formulations of volumetric transmittance. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia) 38*, 6 (2019). 3

[HAAB*18] HADWIGER M., AL-AWAMI A. K., BEYER J., AGUS M., PFISTER H.: *SparseLeap: Efficient* empty space skipping for large-scale volume rendering. *IEEE Transactions on Visualization and Computer Graphics 24*, 1 (2018), 974–983. 2, 5, 6, 10, 19, 20

[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 18*, 12 (2012), 2285–2294. doi:10.1109/TVCG.2012.240. 6, 7

[HMES20] Hofmann N., Martschinke J., Engel K., Stamminger M.: Neural denoising for path tracing of medical volumetric data. *ACM Transactions on Graphics (Proceedings of SIGGRAPH '20) 3*, 2 (2020), 13, 18 pages. 3

[HSB*21] Hoang D., Summa B., Bhatia H., Lindstrom P., Klacansky P., Usher W., Bremer P.-T., Pascucci V.: Efficient and flexible hierarchical data layouts for a unified encoding of scalar field precision and resolution. *IEEE Transactions on Visualization and Computer Graphics 27*, 2 (2021), 603–613. 2, 15, 16, 19, 20

[Hsu93] Hsu W.: Segmented ray casting for data parallel volume rendering. In *Proceedings of IEEE Parallel Rendering Symposium* (1993), pp. 7–14. doi:10.1109/PRS.1993.586079. 12, 18

[IGMM22] Iglesias-Guitian J. A., Mane P., Moon B.: Real-time denoising of volumetric path tracing for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics 28*, 7 (2022), 2734–2747. doi:10.1109/TVCG.2020.3037680. 3, 20

[Int22] Intel Corp.: Open Volume Kernel Library (Open VKL), 2022. Available at https://www.openvkl.org/, Accessed: 7 FEbruary 2023. 17, 20

[JGG*17] Jain S., Griffin W., Godil A., Bullard J. W., Terrill J., Varshney A.: Compressed volume rendering using deep learning. In *Proceedings of the Large Scale Data Analysis and Visualization Symposium* (2017), LDAV '17, pp. 1187–1194. 17

[KHB*16] Kiris C. C., Housman J. A., Barad M. F., Brehm C., Sozer E., Moini-Yekta S.: Computational framework for Launch, Ascent, and Vehicle Aerodynamics (LAVA). *Aerospace Science and Technology 55* (2016), 189–219. 2

[Kim17] Kim D.: *Fluid Engine Development*. CRC Press, Taylor & Francis Group, 2017. 7

[Kit22] KitWare, Inc.: The Visualization Toolkit (VTK), 2022. Available at https://vtk.org/, Accessed: 7 February 2023. 17, 18

[Koc20] Koch D.: Vulkan ray tracing final specification release, 2020. Available at https://www.khronos.org/blog/vulkan-ray-tracing-final-specification-release, Accessed: 6 February 2023. 4

[KPB12] Kroes T., Post F. H., Botha C. P.: Exposure render: An interactive photo-realistic volume rendering framework. *PloS One 7*, 7 (2012), e38586. 3

[KSH03] Kähler R., Simon M., Hege H.-C.: Interactive volume rendering of large data sets using adaptive mesh refinement hierarchies. *IEEE Transactions on Visualization and Computer Graphics 9*, 3 (2003), 341 – 351. doi:10.1109/TVCG.2003.1207442. 10

[KWAH06] Kähler R., Wise J., Abel T., Hege H.-C.: GPU-assisted raycasting for cosmological adaptive mesh refinement simulations. In *Volume Graphics* (2006), VG '06, pp. 103–110. 7, 10

[KWPH06] Knoll A., Wald I., Parker S., Hansen C.: Interactive isosurface ray tracing of large octree volumes. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (2006), RT '06, pp. 115–124. doi:10.1109/RT.2006.280222. 5, 11

[Lar22] Larabel M.: Intel's Open-Source Vulkan Driver for Ray-Tracing Gets "Like A 100x Improvement", 2022. Available at https://www.phoronix.com/news/Intel-Vulkan-RT-100x-Improve, Accessed: 6 February 2023. 4

[LCDP13] Liu B., Clapworthy G. J., Dong F., Prakash E. C.: Octree rasterization: Accelerating high-quality out-of-core GPU volume rendering. *IEEE Transactions on Visualization and Computer Graphics 19*, 10 (2013), 1732–1745. doi:10.1109/TVCG.2012.151. 5

[LCPT11] Labadens M., Chapon D., Pomaréde D., Teyssier R.: Visualization of octree adaptive mesh refinement (AMR) in astrophysical simulations. In *Astronomical Data Analysis Software and Systems XXI*, ASP Conference Series. Astronomical Society of the Pacific, San Francisco, CA, USA, 2011. 7

[LHN05] Lefebvre S., Hornus S., Neyret F.: Octree textures on the GPU. In *GPU Gems 2*. Addison-Wesley, Boston, MA, USA, 2005. URL: https://developer.nvidia.com/gpugems/gpugems2/part-v-image-oriented-computing/chapter-37-octree-textures-gpu. 6

[Lin14] Lindstrom P.: Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014), 2674–2683. doi:10.1109/TVCG.2014.2346458. 16

[LJLB21] Lu Y., Jiang K., Levine J. A., Berger M.: Compressive neural representations of volumetric scalar fields. *Computer Graphics Forum 40*, 3 (2021), 135–146. 2, 16, 19, 20

[LJP*19] Li S., Jaroszynski S., Pearse S., Orf L., Clyne J.: Vapor: A visualization package tailored to analyze simulation data in Earth system science. *Atmosphere 10*, 9 (2019). 15, 16

[Max95] Max N.: Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics 1*, 2 (1995), 99–108. 3, 25

[ME11] Moran P., Ellsworth D.: Visualization of AMR data with multi-level dual-mesh interpolation. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 1862–1871. doi:10.1109/TVCG.2011.252. 7, 19

[MHDG11] Muigg P., Hadwiger M., Doleisch H., Groller E.: Interactive volume visualization of general polyhedral grids. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 2115–2124. 11, 12, 13, 19, 20

[MHDH07] Muigg P., Hadwiger M., Doleisch H., Hauser H.: Scalable hybrid unstructured and structured grid raycasting. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1592–1599. 11, 12, 13, 19, 20

[MHK*19] Martschinke J., Hartnagel S., Keinert B., Engel K., Stamminger M.: Adaptive temporal sampling for volumetric path tracing of medical data. *Computer Graphics Forum 38*, 4 (2019), 67–76. 3

[Mic23] Microsoft Corp.: DirectX Raytracing (DXR) Functional Spec, 2023. Available at https://microsoft.github.io/DirectX-Specs/d3d/Raytracing.html, Accessed: 6 February 2023. 4

[MKPH11] Moreland K., Kendall W., Peterka T., Huang J.: An image compositing solution at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, 2011), SC '11, Association for Computing Machinery. 13, 18

[MOB*21] Meister D., Ogaki S., Benthin C., Doyle M. J., Guthe M., Bittner J.: A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum 40*, 2 (2021), 683–712. 4

[MON*19] Mescheder L., Oechsle M., Niemeyer M., Nowozin S., Geiger A.: Occupancy networks: Learning 3D reconstruction in function space. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2019), CVPR '19, pp. 4455–4465. 16

[Mor23] Moreland K.: *VTK-m Users' Guide, version 2.0*. Tech. Rep. ORNL/TM-2023/2863, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2023. 18

[MRH10] Mensmann J., Ropinski T., Hinrichs K. H.: A GPU-supported lossless compression scheme for rendering time-varying volume data. In *Proceedings of the 8th IEEE VGTC / Eurographics International Symposium on Volume Graphics, VG@Eurographics 2010* (2010), Westermann R., Kindlmann G. L., (Eds.), Eurographics Association, pp. 109–116. URL: https://doi.org/10.2312/VG/VG10/109-116, doi:10.2312/VG/VG10/109-116. 17

[MSG*22] Morrical N., Sahistan A., Güdükbay U., Wald I., Pascucci V.: Quick clusters: A GPU-parallel partitioning for efficient path tracing of unstructured volumetric grids. *IEEE Transactions on Visualization and Computer Graphics* (2022), 1–11. 2, 3, 12, 13, 14, 19, 20

[MST*21] MILDENHALL B., SRINIVASAN P. P., TANCIK M., BARRON J. T., RAMAMOORTHI R., NG R.: NeRF: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM 65*, 1 (2021), 99–106. 20

[MSU*16] MORELAND K., SEWELL C., USHER W., LO L.-T., MEREDITH J., PUGMIRE D., KRESS J., SCHROOTS H., MA K.-L., CHILDS H., LARSEN M., CHEN C.-M., MAYNARD R., GEVECI B.: Vtk-m: Accelerating the visualization toolkit for massively threaded architectures. *IEEE Computer Graphics and Applications 36*, 3 (2016), 48–58. doi:10.1109/MCG.2016.48. 18

[Mus21] MUSETH K.: OpenVDB. In *ACM SIGGRAPH 2021 Courses, No. 4* (New York, NY, USA, 2021), SIGGRAPH '21, Association for Computing Machinery. doi:10.1145/3450508.3464577. 17

[MUWP19] MORRICAL N., USHER W., WALD I., PASCUCCI V.: Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing. In *Proceedings of IEEE Visualization* (2019), VIS '19, pp. 256–260. 2, 12, 14, 19, 20

[MWUP22] MORRICAL N., WALD I., USHER W., PASCUCCI V.: Accelerating unstructured mesh point location with RT cores. *IEEE Transactions on Visualization and Computer Graphics 28*, 8 (2022), 2852–2866. 2, 11, 13, 19, 20

[Nat20] NATIONAL AERONAUTICS AND SPACE ADMINISTRATION (NASA): FUN3D Fully Unstructured Navier-Stokes: Computational Fluid Dynamics (CFD) Suite of Tools, 2020. Available at https://fun3d.larc.nasa.gov/, Accessed: 7 February 2023. 2

[Nat21] NATIONAL AERONAUTICS AND SPACE ADMINISTRATION: Fun3D Retropropulsion Data Portal—Simulations of retropropulsion to decelerate a space vehicle entering a planetary atmosphere. https://data.nas.nasa.gov/fun3d/, 2021. Accessed: 7 January 2023. 14

[NLKH12] NELSON B., LIU E., KIRBY R. M., HAIMES R.: Elvis: A system for the accurate and interactive visualization of high-order finite element solutions. *IEEE transactions on visualization and computer graphics 18*, 12 (2012), 2325–2334. 11

[NVI09a] NVIDIA CORP.: IndeX®3D Volumetric Visualization Framework, 2009. Available at https://developer.nvidia.com/nvidia-index, Accessed: 27 January 2023. 18

[NVI09b] NVIDIA CORP.: NVIDIA OptiX Ray Tracing Engine, 2009. Available at https://developer.nvidia.com/optix, Accessed: 7 February 2023. 2, 4, 18

[NVI20] NVIDIA CORP.: NVIDIA IndeX for ParaView Plugin, Version 2.4, 2020. Available at https://www.mn.uio.no/astro/english/services/it/help/visualization/paraview/nvidia-index-paraview-plugin-user-guide-5.8.pdf, Accessed: 7 February 2023. 18

[Pan08] PANIGRAHY R.: An improved algorithm finding nearest neighbor using kd-trees. In *LATIN 2008: Theoretical Informatics* (Berlin, Heidelberg, 2008), Laber E. S., Bornstein C., Nogueira L. T., Faria L., (Eds.), Springer Berlin Heidelberg, pp. 387–398. 9

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., ET AL.: Optix: a general purpose ray tracing engine. *Acm transactions on Graphics 29*, 4 (2010), 1–13. 4

[PD84] PORTER T., DUFF T.: Compositing digital images. *ACM Computer Graphics (Proc. SIGGRAPH '84) 18*, 3 (jan 1984), 253–259. doi:10.1145/964965.808606. 12, 25

[PDP*19] PEYROT J.-L., DUVAL L., PAYAN F., BOUARD L., CHIZAT L., SCHNEIDER S., ANTONINI M.: Hexashrink, an exact scalable framework for hexahedral meshes with attributes and discontinuities: multiresolution rendering and storage of geoscience models. *Computational Geosciences 23* (2019), 723–743. 14

[PFS*19] PARK J. J., FLORENCE P., STRAUB J., NEWCOMBE R. A., LOVEGROVE S.: DeepSDF: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2019), CVPR '19, IEEE, pp. 165–174. 16

[Pin88] PINEDA J.: A parallel algorithm for polygon rasterization. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques* (New York, NY, USA, 1988), SIGGRAPH '88, Association for Computing Machinery, p. 17–20. doi:10.1145/54852.378457. 13

[PM12] PHARR M., MARK W. R.: ispc: A SPMD compiler for high-performance CPU programming. In *Proceedings of the Innovative Parallel Computing* (2012), InPar '12, pp. 1–13. 2

[PNP*17] PATCHETT J. M., NOUANESENGESY B., POUDEROUX J., AHRENS J., HAGEN H.: Parallel multi-layer ghost cell generation for distributed unstructured grids. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization* (2017), LDAV '17, pp. 84–91. 4

[PZGY19] PAN Y., ZHU F., GAO T., YU H.: Adaptive deep learning based time-varying volume compression. In *Proceedings of the IEEE International Conference on Big Data* (2019), Big Data 2019, pp. 1187–1194. 2, 17

[QCJJ18] QUAN T. M., CHOI J., JEONG H., JEONG W.-K.: An intelligent system approach for probabilistic volume rendering using hierarchical 3D convolutional sparse coding. *IEEE Transactions on Visualization and Computer Graphics 24*, 1 (2018), 964–973. 2, 17

[RPLG21] REISER C., PENG S., LIAO Y., GEIGER A.: KiloNeRF: Speeding up neural radiance fields with thousands of tiny MLPs. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (2021), pp. 14335–14345. 20

[RWCB15] RATHKE B., WALD I., CHIU K., BROWNLEE C.: SIMD parallel ray tracing of homogeneous polyhedral grids. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (2015), EGPGV 15, pp. 33–41. 13

[SBS*17] SOLTESZOVA V., BIRKELAND A., STOPPEL S., VIOLA I., BRUCKNER S.: Output-sensitive filtering of streaming volume data. *Computer Graphics Forum 36*, 1 (2017), 249–262. doi:https://doi.org/10.1111/cgf.12799. 7

[SCCB05] SILVA C., COMBA J., CALLAHAN S., BERNARDON F.: A survey of GPU-based volume rendering of unstructured grids. *Brazilian Journal of Theoretic and Applied Computing 12*, 2 (2005), 9–29. 11

[SCRL20] SARTON J., COURILLEAU N., REMION Y., LUCAS L.: Interactive visualization and on-demand processing of large volume data: A fully GPU-based out-of-core approach. *IEEE Transactions on Visualization and Computer Graphics 26*, 10 (2020), 3008–3021. 2, 6, 7, 19, 20

[SDM*21] SAHISTAN A., DEMIRCI S., MORRICAL N., ZELLMANN S., AMAN A., WALD I., GÜDÜKBAY U.: Ray-traced shell traversal of tetrahedral meshes for direct volume visualization. In *Proceedings of IEEE Visualization Conference* (2021), VIS '21, pp. 91–95. 2, 11, 12, 13, 19, 20

[SDW*22] SAHISTAN A., DEMIRCI S., WALD I., ZELLMANN S., BARBOSA J., MORRICAL N., GÜDÜKBAY U.: GPU-based data-parallel rendering of large, unstructured, and non-convexly partitioned data, 2022. doi:10.48550/ARXIV.2209.14537. 12, 13, 19, 20

[SGA*22] STONE J. E., GRIFFIN K. S., AMSTUTZ J., DEMARLE D. E., SHERMAN W. R., GÜNTHER J.: ANARI: A 3-D Rendering API Standard. *Computing in Science & Engineering 24*, 2 (2022), 7–18. doi:10.1109/MCSE.2022.3163151. 17, 20

[SKTM11] SZIRMAY-KALOS L., TÓTH B., MAGDICS M.: Free path sampling in high resolution inhomogeneous participating media. *Computer Graphics Forum 30*, 1 (2011), 85–97. doi:https://doi.org/10.1111/j.1467-8659.2010.01831.x. 4

[SMB*20] SITZMANN V., MARTEL J. N. P., BERGMAN A. W., LINDELL D. B., WETZSTEIN G.: Implicit neural representations with periodic activation functions. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Red Hook, NY, USA, 2020), NIPS'20, Curran Associates Inc. 16

[SML06] Schroeder W., Martin K. M., Lorensen W. E.: *Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 4 ed. Kitware, Inc., Clifton Park, NY, USA, 2006. 18

[SRL19] Sarton J., Remion Y., Lucas L.: Distributed out-of-core approach for in-situ volume rendering of massive dataset. In *High Performance Computing* (Cham, 2019), Weiland M., Juckeland G., Alam S., Jagode H., (Eds.), Springer International Publishing, pp. 623–633. 6

[ST90] Shirley P., Tuchman A.: A polygonal approximation to direct scalar volume rendering. *ACM Computer Graphics (Proceedings of the Workshop on Volume Visualization, VolVis '90) 24*, 5 (1990), 63–70. 2, 11

[TDCC17] Tao D., Di S., Chen Z., Cappello F.: Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium* (Los Alamitos, CA, USA, jun 2017), IPDPS '17), IEEE Computer Society, pp. 1129–1139. URL: https://doi.ieeecomputersociety.org/10.1109/IPDPS.2017.115. 16

[TM13] Taubman D., Marcellin M.: *JPEG2000 Image Compression Fundamentals, Standards and Practice*. Springer, New York, NY, USA, 2013. 16

[VMD08] Vidal V., Mei X., Decaudin P.: Simple empty-space removal for interactive volume rendering. *Journal of Graphics Tools 13*, 2 (2008), 21–36. doi:10.1080/2151237X.2008.10129258. 5

[Wal07] Wald I.: On fast construction of SAH-based bounding volume hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing* (USA, 2007), RT '07, IEEE Computer Society, p. 33–40. doi:10.1109/RT.2007.4342588. 10

[Wal20] Wald I.: A simple, general, and GPU friendly method for computing dual mesh and iso-surfaces of adaptive mesh refinement (AMR) data, 2020. doi:10.48550/arxiv.2004.08475. 7

[WBUK17] Wald I., Brownlee C., Usher W., Knoll A.: CPU volume rendering of adaptive mesh refinement data. In *Proceedings of SIGGRAPH Asia 2017 Symposium on Visualization* (New York, NY, USA, 2017), SA '17, ACM, pp. 9:1–9:8. 2, 7, 8, 9, 10, 19, 20

[WCM12] Weber G. H., Childs H., Meredith J. S.: Efficient parallel extraction of crack-free isosurfaces from adaptive mesh refinement (AMR) data. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (2012), pp. 31–38. doi:10.1109/LDAV.2012.6378973. 7, 18

[WDBM22] Wu Q., Doyle M. J., Bauer D., Ma K.-L.: Instant neural representation for interactive volume rendering. *arXiv preprint arXiv:2207.11620* (2022). 20

[WHss] Wang C., Han J.: DL4SciVis: A state-of-the-art survey on deep learning for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics* (In press). doi:10.1109/TVCG.2022.3167896. 3, 17

[WHW22] Weiss S., Hermüller P., Westermann R.: Fast neural representations for direct volume rendering. *Computer Graphics Forum 41*, 6 (2022), 196–211. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14578, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14578, doi:https://doi.org/10.1111/cgf.14578. 17

[WJA*17] Wald I., Johnson G., Amstutz J., Brownlee C., Knoll A., Jeffers J., Günther J., Navratil P.: OSPRay - A CPU ray tracing framework for scientific visualization. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 931–940. 2, 3, 14, 17, 18, 20

[WKL*01] Weber G. H., Kreylos O., Ligocki T. J., Shalf J. M., Hagen H., Hamann B., Joy K., Ma K.-L.: High-quality volume rendering of adaptive mesh refinement data. In *Proceedings of the Vision Modeling and Visualization Conference* (2001), VMV '01. 7

[WKME03] Weiler M., Kraus M., Merz M., Ertl T.: Hardware-based ray casting for tetrahedral meshes. In *IEEE Visualization, 2003. VIS 2003.* (Oct. 2003), pp. 333–340. ISSN: null. doi:10.1109/VISUAL.2003.1250390. 11

[WL16] Weiss K., Lindstrom P.: Adaptive multilinear tensor product wavelets. *IEEE Transactions on Visualization and Computer Graphics 22*, 1 (2016), 985–994. doi:10.1109/TVCG.2015.2467412. 16

[WMHL65] Woodcock E., Murphy T., Hemmings P., Longworth T.: *Techniques used in the GEM code for Monte Carlo neutronics calculation in reactors and other systems of complex geometry*. Tech. rep., Argonne National Laboratory, 1965. 3

[WMU*20] Wang F., Marshak N., Usher W., Burstedde C., Knoll A., Heister T., Johnson C. R.: CPU ray tracing of tree-based adaptive mesh refinement data. *Computer Graphics Forum 39*, 3 (2020), 1–12. 8, 11, 19, 20

[WMZ21] Wald I., Morrical N., Zellmann S.: A memory efficient encoding for ray tracing large unstructured data. *IEEE Transactions on Visualization and Computer Graphics 28*, 1 (2021), 583–592. 2, 11, 14, 19, 20

[WN21] Weiss J., Navab N.: Deep direct volume rendering: Learning visual feature mappings from exemplary images. *CoRR abs/2106.05429* (2021). URL: https://arxiv.org/abs/2106.05429, arXiv:2106.05429. 2, 17

[WUM*19] Wald I., Usher W., Morrical N., Lediaev L., Pascucci V.: RTX beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location. In *Proceedings of High-Performance Graphics - Short Papers* (July 2019), HPG '19, The Eurographics Association. 2, 13, 19, 20

[WUP*18] Wu Q., Usher W., Petruzza S., Kumar S., Wang F., Wald I., Pascucci V., Hansen C. D.: VisIt-OSPRay: toward an exascale volume visualization system. In *Proceedings of the Symposium on Parallel Graphics and Visualization* (Goslar, DEU, 2018), EGPGV '18, Eurographics Association, pp. 13–24. 18, 20

[WWB*14] Wald I., Woop S., Benthin C., Johnson G. S., Ernst M.: Embree: A kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics 33*, 4 (2014). doi:10.1145/2601097.2601199. 2, 4, 17

[WWJ19] Wang F., Wald I., Johnson C. R.: Interactive rendering of large-scale volumes on multi-core CPUs. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization* (2019), LDAV '19, pp. 27–36. 2, 6, 19, 20

[WWW*19] Wang F., Wald I., Wu Q., Usher W., Johnson C. R.: CPU isosurface ray tracing of adaptive mesh refinement data. *IEEE Transactions on Visualization and Computer Graphics 25*, 1 (2019), 1142–1151. 2, 7, 8, 9, 19, 20

[WZM21] Wald I., Zellmann S., Morrical N.: Faster RTX-accelerated empty space skipping using triangulated active region boundary geometry. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (2021), Larsen M., Sadlo F., (Eds.), PGV '21, The Eurographics Association. doi:10.2312/pgv.20211042. 2, 5, 6, 19, 20

[WZU*21] Wald I., Zellmann S., Usher W., Morrical N., Lang U., Pascucci V.: Ray tracing structured AMR data using ExaBricks. *IEEE Transactions on Visualization and Computer Graphics 27*, 2 (2021), 625–634. 2, 8, 10, 19, 20

[XTC*ss] Xu J., Thevenon G., Chabat T., McCormick M., Li F., Birdsong T., Martin K., Lee Y., Aylward S.: Interactive, in-browser cinematic volume rendering of medical images. *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization* (In press). doi:10.1080/21681163.2022.2145239. 3

[YHGT10] Yang J. C., Hensley J., Grün H., Thibieroz N.: Real-time concurrent linked list construction on the GPU. *Computer Graphics Forum 29*, 4 (2010), 1297–1304. doi:https://doi.org/10.1111/j.1467-8659.2010.01725.x. 5

[YIC*10] Yue Y., Iwasaki K., Chen B.-Y., Dobashi Y., Nishita T.: Unbiased, adaptive stochastic sampling for rendering inhomogeneous participating media. *ACM Transactions on Graphics 29*, 6 (2010). doi:10.1145/1882261.1866199. 4

[YKL17] Ylitie H., Karras T., Laine S.: Efficient incoherent ray traversal on gpus through compressed wide bvhs. In *Proceedings of High Performance Graphics* (New York, NY, USA, 2017), HPG '17, Association for Computing Machinery. doi:10.1145/3105762.3105773. 14

[ZHL19] Zellmann S., Hellmann M., Lang U.: A linear time BVH construction algorithm for sparse volumes. In *Proceedings of the IEEE Pacific Visualization Symposium* (2019), PacificVis '19, pp. 222–226. 4, 5

[ZML19] Zellmann S., Meurer D., Lang U.: Hybrid grids for sparse volume rendering. In *Proceedings of the IEEE Visualization Conference* (2019), VIS '19, pp. 1–5. doi:10.1109/VISUAL.2019.8933631. 5

[ZSL18] Zellmann S., Schulze J. P., Lang U.: Rapid k-d tree construction for sparse volume data. In *Proceedings of the Symposium on Parallel Graphics and Visualization* (Goslar, DEU, June 2018), EGPGV '18, Eurographics Association, pp. 69–77. 5

[ZSL21] Zellmann S., Schulze J. P., Lang U.: Binned k-d tree construction for sparse volume data on multi-core and GPU systems. *IEEE Transactions on Visualization and Computer Graphics 27*, 3 (2021), 1904–1915. 2, 4, 5, 19, 20

[ZSM*22] Zellmann S., Seifried D., Morrical N., Wald I., Usher W., Law-Smith J., Walch-Gassner S., Hinkenjann A.: Point containment queries on ray tracing cores for AMR flow visualization. *Computing in Science & Engineering 24*, 2 (2022), 40–52. doi:10.1109/MCSE.2022.3153677. 2, 8, 11, 19, 20

[ZWB*22] Zellmann S., Wald I., Barbosa J., Demirci S., Sahistan A., Gudukbay A.: Hybrid image-/data-parallel rendering using island parallelism. In *Proceedings of the IEEE 12th Symposium on Large Data Analysis and Visualization* (2022), LDAV '22, pp. 1–10. 2, 3, 13, 14

[ZWS*22a] Zellmann S., Wald I., Sahistan A., Hellmann M., Usher W.: Design and evaluation of a GPU streaming framework for visualizing time-varying AMR data. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (2022), Bujack R., Tierny J., Sadlo F., (Eds.), PGV '22, The Eurographics Association. doi:10.2312/pgv.20221066. 2, 8, 11, 19, 20

[ZWS*22b] Zellmann S., Wu Q., Sahistan A., Ma K.-L., Wald I.: Beyond ExaBricks: GPU Volume Path Tracing of AMR Data, 2022. arXiv:2211.09997. 3, 8, 11, 19, 20

Appendix A: Algorithms

The three algorithms below present volume integration methods we refer to throughout the paper and compute common quantities such as absorption and emission, iso-surfaces, and transmission coefficients, for the interval $[t_{min}, t_{max}]$ in ray parameter space.

Algorithm 1 presents a classic ray marching loop to compute absorption and emission [Max95] inside the volumetric partition; the result is a alpha-composited color using *over* [PD84]. Algorithm 2 implicitly computes an isosurface defined by the ISO value γ; the result is the distance t from the ray origin where (and if) the isosurface was found. Algorithm 3 estimates transmission coefficients using Woodcock (delta tracking) free-flight distance sampling used for volumetric path tracing.

Algorithm 1 Absorption plus emission DVR ray marcher.

```
1: function RayMarchingDVR($o$, $\omega$, $t_{min}$, $t_{max}$)
2:     $C_{dst} = 0$
3:     for $t = t_{min} \dots t_{max}$ do
4:         $C_{dst} = \text{Over}(C_{dst}, C_{src}(o + t * \omega))$
5:     end for
6:     return $C_{dst}$
7: end function
```

Algorithm 2 Implicit ISO-surface extraction.

```
1: function RayMarchingISO($o$, $\omega$, $t_{min}$, $t_{max}$, $\gamma$)
2:     for $t = t_{min} + \frac{1}{2} \dots t_{max} - \frac{1}{2}$ do
3:         $s_1 = \mu(o + (t - \frac{1}{2}) * \omega)$
4:         $s_2 = \mu(o + (t + \frac{1}{2}) * \omega)$
5:         if $\text{min}(s_1, s_2) \leq \gamma \leq \text{max}(s_1, s_2)$ then
6:             return t
7:         end if
8:     end for
9: end function
```

Algorithm 3 Woodcock free-flight distance sampling.

```
1: function Woodcock($o$, $\omega$, $t_{min}$, $t_{max}$, $\bar{\mu}$)
2:     $t = t_{min}$
3:     do
4:         $\zeta = \text{rand}()$
5:         $t = t - \frac{\log(1-\zeta)}{\bar{\mu}}$
6:         if $t \geq t_{max}$ then
7:             break
8:         end if
9:         $\xi = \text{rand}()$
10:    while $\xi > \frac{\mu(o+t*\omega)}{\bar{\mu}}$
11:    return t
12: end function
```