HARDWARE-ACCELERATED DIRECT VISUALIZATION OF UNSTRUCTURED VOLUMETRIC MESHES

A THESIS SUBMITTED TO

THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF

MASTER OF SCIENCE

IN

COMPUTER ENGINEERING

By

Alper Şahıstan

July 2022

Hardware-accelerated Direct Visualization of Unstructured Volumetric Meshes
By Alper Şahıstan
July 2022

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Uğur Güdükbay(Advisor)

Özgür Ulusoy

Ahmet Oğuz Akyüz

Approved for the Graduate School of Engineering and Science:

Orhan Arıkan
Director of the Graduate School

ABSTRACT

HARDWARE-ACCELERATED DIRECT VISUALIZATION OF UNSTRUCTURED VOLUMETRIC MESHES

Alper Şahıstan
MS in Computer Engineering
Advisor: Uğur Güdükbay
July 2022

Computational fluid dynamic simulations often produce large clusters of finite elements with non-trivial, non-convex boundaries and uneven distributions among compute nodes, posing challenges to compositing during interactive volume rendering. Correct, in-place visualization of such clusters becomes difficult because viewing rays straddle domain boundaries across multiple compute nodes. We propose a GPU-based, scalable, memory-efficient direct volume visualization framework suitable for in situ and post hoc usage. Our approach reduces memory usage of the unstructured volume elements by leveraging an exclusive or-based index reduction scheme and provides fast ray-marching-based traversal without requiring large external data structures built over the elements. Moreover, we present a GPU-optimized deep compositing scheme that allows correct order compositing of intermediate color values accumulated across different ranks that works even for non-convex clusters. Furthermore, we illustrate that we can achieve secondary effects such as shadows and gradient shading using our method for single GPU setups. Our approach scales well on large data-parallel systems and achieves interactive frame rates during visualization. We can interactively render Fun3D Small Mars Lander (14 GB / 798.4 million finite elements) and Huge Mars Lander (111.57 GB / 6.4 billion finite elements) data sets at 14 and 10 frames per second using 72 and 80 GPUs, respectively, on the Frontera supercomputer at The Texas Advanced Computing Center (TACC).

Keywords: Direct volume visualization, unstructured volumetric mesh, ray tracing, acceleration structure, tetrahedralization, bounding volume hierarchy, k-d tree, hardware-acceleration, Graphics Processing Unit.

ÖZET

DÜZENSİZ HACİMSEL AĞLARIN DONANIMSAL HIZLANDIRICI YÖNTEMLERİ İLE DOĞRUDAN GÖRÜNTÜLENMESİ

Alper Şahıstan
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Danışmanı: Uğur Güdükbay
Temmuz 2022

Hesaplamalı akışkan dinamiği simülasyonları genellikle büyük, karmaşık ve dış bükey yapıda olmayan sonlu eleman kümeleri üretir. Bu verilerin dağıtık sistemlerde interaktif doğrudan hacim görüntülenmesi esnasında hesaplama düğümleri arasında meydana gelen adil olmayan iş bölümü, elde edilen kısmi çözümlerin birleştirilmesi aşamasında sorun teşkil edebilir. Bu tarz durumlarda görüntülenmek isteyen hacim modelleri için doğru ve yerinde çalışan bir yöntem önermek ışın takibi yapılırken ışınların küme sınırları arasında farklı hesaplama düğümlerine erişmelerinden ötürü zor bir süreçtir. Bu tezde Grafik İşlemci Ünitesi tabanlı, ölçeklenebilir, hafıza verimi yüksek bir doğrudan hacim görselleştirme yöntemi sunuyoruz. Bu yöntem hem simülasyon yeri ve anında, hem de sonrasında kullanım için tasarlanmıştır. Yaklaşımımız XOR-tabanlı sıkıştırma tekniklerini kullanarak düzensiz hacim elemanlarının bellek kullanımını azaltmakta ve büyük hierarşik yapılarak gerek kalmadan hızlandırılmış ışın yürüyüşü sağlamaktadır. Ayrıca, bu karmaşık dış bükey yapıda olmayan ağların ışın yürüyüşü sırasında elde edilen kısmi çözümleri (renk değerleri) birleştiren "derin-birleştirici" algoritması sunulmuştur. Buna ek olarak, önerilen yöntemin tek ekran kartı ile çalışan bir senaryoda ikincil efektlerin kullanımına uygun olduğu gösterilmektedir. Yöntemimiz büyük veri-paralel sistemlerde iyi bir şekilde ölçeklenebilmekte ve interaktif hızlarda görüntü alabilmektedir. Fun3D'nin Küçük Mars İniş Aracı (14GB / 798.4 milyon eleman) ve Devasa Mars İniş Aracı (111.57GB / 6.4 milyar eleman) modellerini sırasıyla saniyede 14 ve 10 kare olarak görüntüleyebilmektedir. Bu performansa Texas Advanced Computing Center (TACC)'da yeralan Frontera süper bilgisayarında küçük model için 72, devasa olan için 80 GPU kullanılarak erişilmiştir.

Anahtar sözcükler: Doğrudan hacim görüntüleme, düzensiz hacimsel ağ, ışın izleme, hızlandırıcı yapısı, dörtyüzleme, sınırlayıcı hacim hiyerarşisi, k-d ağacı, donanımsal hızlandırıcı, grafik işlemci ünitesi.

Acknowledgement

I want to offer my gratitude to Prof. Dr. Uğur Güdükbay for introducing me to the world of Computer Graphics and pushing me forward in my research. I sincerely appreciate Prof. Dr. Özgür Ulusoy and Prof. Dr. Ahmet Oğuz Akyüz for kindly accepting to be my thesis jury.

I want to thank Dr. Ingo Wald, who let me collaborate with him and his research group and shared his unique resources and knowledge.

I want to thank Dr. Aytek Aman, Dr. Stefan Zellman, Nate Morrical, and Serkan Demirci for the tremendous wisdom they shared with me.

I would also like to thank my mom, my dad, and Tuğba Selvi for their emotional and financial support.

We gratefully acknowledge the providers of the datasets and hardware used in our experiments. Mars Lander datasets are courtesy of National Aeronautics and Space Administration (NASA) Fun3D Library. The Agulhas dataset is courtesy of Dr. Niklas Röber (DKRZ); the Deep Water Asteroid Impact is courtesy of John Patchett and Galen Gisler of LANL. Plasma64 dataset is courtesy of AIM@SHAPE. Jets and Fusion datasets are courtesy of the SCI Institute of the University of Utah. Hardware for development and testing was graciously provided by NVIDIA Corporation and The Texas Advanced Computing Center (TACC).

This work is supported by the Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant No. 117E881.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the increasing computation power over the years, computational fluid dynamic (CFD) simulations have become more complex. To understand the underlying phenomena, scientists rely on scientific visualization methods. One of the most popular visualization methods is direct volume rendering (DVR). A robust way to achieve DVR is to utilize ray-tracing, where view-aligned rays are traced through the volume sampling scalar values stored within volume elements. Ray-tracing is embarrassingly parallel and allows for secondary effects such as ambient occlusion or single or multiple scattering. These effects usually convey more information to users about the visualization by giving depth cues and occlusion. From improved divergence handling to built-in ray-tracing (RT) cores, recent advancements in the Graphics Processing Units (GPU) open many possibilities in terms of applications of DVR. Nevertheless, single-node visualization techniques stop offering feasible solutions with the increasing sizes of simulations. For these reasons, many modern visualization systems rely on distributed rendering solutions that can achieve ray-traced effects.

Contemporary large-scale volumetric simulations often operate over unstructured

meshes, as individual elements can be adaptively distributed. These distributions ensure detailed regions receive more elements while less critical areas receive fewer elements, thus increasing accuracy while preserving the memory when necessary. However, as these meshes bend and twist around to achieve this flexibility, implement robust algorithms to traverse and store these meshes. Moreover, many of these meshes produce non-convex or non-trivially shaped boundaries that present difficulties for some visualization algorithms. Many simulation data represent their unstructured geometry with tetrahedra to maintain algorithmic simplicity, but it often produces a high memory footprint to keep many smaller elements rather than a few larger elements. Hence, extensive CDF simulations construct their geometry using mixed elements like tetrahedra, pyramids, wedges, and hexahedra.

A popular way to traverse through these unstructured volumes is using ray-marching via connectivity. This approach uses view-aligned rays to sample volumes while intersecting individual faces of elements to jump to the next element in succession. With the added benefit of not using external hierarchical data structures like Bounding Volume Hierarchy (BVH) or KD-tree, the ray-marchers that can handle mixed element unstructured meshes with non-convex boundaries are particularly appealing.

Although memory bandwidth and sizes have significantly increased over the years, it has not been at the same level as the compute power. Supercomputer systems that solve large-scale simulations tend to distribute their memory across nodes to fully utilize the underlying compute power, which requires data-parallel processing. Due to the sheer size of the simulation data, time and effort put into one compute round can be tremendous. For this reason, simulation frameworks often implement in situ visualization and steering to tap into a running simulation. However, distributing portions of the topology and running parallel rendering runs requires each partial image generated to be combined. The standard way to achieve a final image is via compositing, yet as mentioned before, some clusters of unstructured meshes come with non-convex boundaries, making the composting process challenging. Since CFD simulations tend to take up a significant amount of resources and redistributing data is infeasible due to high I/O costs, interactive, lightweight, correct, and in-place visualization algorithms have become necessary.

This work explores a data-parallel GPU-based in situ DVR framework that can interactively render large-scale mixed element volumetric datasets with non-convex boundaries. Our contributions mostly accumulate in [1] and [2]. More specifically, we present:

- a compact, cache- and GPU-friendly memory layout that facilitates fast ray-element intersection and efficient traversal;
- a shell-to-shell traversal scheme that allows robust entry through convex and non-convex boundaries of clusters, non-common-origin rays;
- an extension that allows secondary effects like ray-traced shading, shadows, or ambient occlusion to be applied on meshes that fit into a single GPU;
- a GPU-optimized deep compositor that supports convex and non-convex geometry with proper depth ordering.

With our framework, we observe significant memory savings provided by our geometry compaction scheme. We also observe interactive rates while correctly rendering large-scale data sets on the simulation system's native distributions. Although most communications between nodes occur during our fragment compositing step, increasing the compute node count does not significantly impact the overall compositing time, which tells us our compositing method scales well. We achieve interactive framerates (10-15 frames per second) for reasonably large data sets while using the native distributions of their respective simulations.

The organization of this thesis is as follows. Related work on direct volume rendering alongside similar approaches is given in Chapter 2. Our volume integrator and its novelties are given in Chapter 3. We describe a deep-compositor that works with convex and non-convex meshes in Chapter 4. We present and discuss our experimental results in Chapter 5. Chapter 6 concludes and provides further research directions.

Chapter 2

Background and Related Work

2.1 Unstructured Volume Rendering

There are variety of techniques proposed for rendering unstructured meshes [3, 4, 5, 6]. The two popular schools of visualizing unstructured volumes are *point-query sampling*, e.g., [7] and *ray-marching* [8].

Traditional point-query sampling casts rays using external acceleration structures built over individual volume elements like KD-trees or *bounding volume hierarchies* (BVH) to take samples at determined coordinates. Using point-query sampling offers advantages like adaptive sampling and space skipping techniques. Rathke et al. [9] speeds up the element look-up processes using a min/max BVH. Wald et al. [10] exploit NVIDIA's ray-tracing (RT) cores for point location queries on tetrahedral meshes. Extention of this work is proposed by Morrical et al. [7] where queries can handle all unstructured elements. These approaches can produce quick but noisy results that can converge over time. To further accelerate convergence times and pick necessary samples, adaptive sampling strategies [11, 12], or empty space skipping [13, 14] can be utilized. Morrical et al. show that empty space skipping and adaptive sampling [15] can be accelerated via the RTX hardware. These techniques prove helpful in many visualization scenarios; however, waiting for convergence for a clean-looking image and

usage of auxiliary hierarchical structures makes these approaches less attractive for other needs, such as in situ visualization, where partial images are composited while a running simulation throttles computing and memory resources.

Conventional ray-marching techniques calculate individual pixel colors by accessing all volume elements in visibility-ordering without external acceleration structures. *Marching* is performed via visibility sorting or element connectivity. Shirley and Tuchmann [8] is a well-known rasterization-based method for tetrahedral mesh rendering. Due to the high cost of sorting that incurs every time viewing angle changes, many researchers turned their interests to traversal via connectivity. A recent method by Aman et al. [16, 17] explores a tetrahedra traversal algorithm that reduces memory consumption and instruction count introduced by intersection tests. Although these works propose solutions for pure tetrahedral meshes, Muigg et al. [6] addresses the problem of rendering mixed element meshes that includes other element types than tetrahedra. Their work can handle non-convex bounding geometry by storing compact face-based connectivity lists and projecting vertices to a ray-centric coordinate system for intersections. For some ray-marching applications finding the first element where the ray first enters the volume may be required to start the marching; Sahistan et al. [2] describe doing this with RTX hardware by building a BVH over the shell and tracing rays.

Since many CFD simulations already maintain a connectivity list or a connectivity matrix for their elements, ray-marchers that can leverage those already present lists are particularly appealing for in situ purposes.

2.2 Secondary Effects

Secondary effects such as ambient occlusion, shadows, shading, and border contours improve perception and visualization quality. These effects usually require tracing secondary rays, cast after primary rays interact with the scene geometry. These secondary rays, unlike primary rays, are arbitrary and incoherent; achieving these effects in raster-based pipelines is challenging [18] and often requires extra raster passes.

Volumetric shadows and ambient occlusion(AO) are secondary effects that increase depth perception. When combined, these effects can create soft-looking shadows as well. We use the standard ray traced AO method, proposed in [19, 20].

Volumetric gradient shading is also a well-known technique where a normal to a given sample point is calculated alongside its color and transparency. The local shading underlying the phenomena can be achieved using the normal direction. Although there are various ways to calculate this normal direction, we utilize central differences [21, 22, 23].

2.3 Data-parallel Rendering

For the massive simulations that cannot fit into a single compute node, parallelization comes to the rescue. Various works are proposed to distribute partitions of the simulation data over a many-cluster environment. Distributing data pieces (clusters) between nodes (i.e., data-parallel rendering or *sort-last*) is a popular method employed by recent works [24, 25]. Work-load can also be distributed in an image plane where pixel regions are assigned to different compute nodes [26, 27]. These methods are often called *image-order partitioning* or *sort-first*. There are also hybrid approaches [28, 29], which aim to address load-balancing issues by leveraging both perspectives.

The approaches that distribute clusters across compute nodes allow static and non-replicated geometry assignment. The main caveat with these techniques is that partial images generated by each cluster need to be composited to create a final image. Although static geometry assignment makes sort-last methods popular, correct and efficient compositing remains challenging. This challenge is very much the case for image-based compositors like IceT[30, 31], which produce a single intermediate image per node, which is unsuitable for clusters with non-convex domain boundaries. Recently, a work by Grosset et al. [32] tackled the sort-last compositing problem, which reduces delays and communications by utilizing a spatiotemporally-aware compositor. Their approach uses "chains" that determine the blending order of each strip of the image. Usher et al. [33] describe *Distributed FrameBuffers*, an algorithm that breaks

image processing operations into tiles of ranks via dependency trees. The Galaxy framework [34] implements an asynchronous frame buffer, which exploits independent pixel updates sent from a server while allowing incremental refinements to the final image over time.

Like our work, many researchers proposed frameworks to reduce node-to-node communications and optimize workloads while maintaining correctness.Ma [35] introduces a data-parallel unstructured volume rendering method with the ability to handle non-convex data boundaries properly. Like our shell-to-shell traversal, they also offer traversal for convex and non-convex cluster boundaries via a *hierarchical data structure*. Their work also focuses on correct-order compositing for non-trivial meshes; however, unlike our *deep compositing*, they opt for many smaller MPI messages between compute nodes. Although this might have been more efficient for the times CPUs, modern GPUs generally perform better while processing large bulks of similar work(i.e., SIMD). Ma et al.'s work is later extended to utilize asynchronous load balancing via object and image-order techniques [36]. However, this work uses cell-projection techniques rather than ray-casting.

Childs et al. [37] layout a two-stage framework that first samples a $m \times n \times k$ view-aligned grid —where m and n denote the pixel resolution and k is the sample per pixel— then composites these samples in the proper viewing order. Their pipeline distributes the workload by sampling "small" and "large" elements by classifying elements via a parameter. First, small elements are sampled, and then large elements are distributed across processors to be sampled more load-balanced. Binyabib et al. [38] extend this framework by allowing successive samples to be partially composited before final image generation, thus reducing memory usage. Although the framework we propose extends these frameworks by Childs et al. [37] and Binyabib et al. [38], in the regard that ours can also handle jagged cluster boundaries, their grid-based sampler is infeasible for our purposes as, in theory, it will waste precious memory resources for large framebuffers.

Moreover, the 3D rasterization process required by Childs et al. will also be sensitive to overdraw when millions of elements fall within the same grid cell. Finally, both of these works' image-order load balancing method requires large elements to be

replicated or moved to some other nodes, thus requesting additional memory, which may not always be present given an in situ scenario. Furthermore, we also acknowledge that GPU architectures improve with new divergence handling methods and ray-tracing (RT) cores, so the adaptations we propose in this work are necessary.

Unlike previous research, our technique is optimized for contemporary GPUs, lowering the costs of compositing and sorting operations. Because we ray-march through each segment to determine partial samples, we do not need to buffer every sample along with the rays. Finally, unlike previous approaches, ours does not necessitate redistributing or reproducing pieces between nodes to present data.

2.4 In situ Rendering

File I/O has long been a bottleneck of high-performance computing. In situ visualization combines computation and visualization to overcome this barrier, allowing users to access a running simulation. In situ visualization has several advantages, including examining data, running numerical queries, and generating graphical outputs while the simulation is running. It also enables verification, allowing the simulation to be interrupted or updated, saving time and computing resources. [39]. Due to our ability to create correct pictures with little to no help from extra acceleration structures at interactive rates, we believe our technique is appropriate for in situ applications.

Infrastructures such as Strawman [40] or Ascent [41] are used to generate high-quality CFD simulations, which can be rendered by in situ many frameworks [42, 43]. Various new algorithmic improvements have been proposed to manage time-varying data generated by simulations and traditional systems. Yamoka et al. [44] introduce a method that adapts the timestep sampling rate according to variations in the probability distribution function (PDF) via an approximation of the connected simulation. A model by Aupy et al. [45] allows them to analyze simulations to create high-throughput scheduling. DeMarle and Bauer [46] present a temporal cache technique that stores a large amount of time-varying information generated by a running simulation and can be stored later according to a pre-defined trigger. Marsaglia et al. [47] propose an

error-bound in situ compression strategy that permits whole spatiotemporal simulation data to be saved. In addition to what is presently retained in simulations, our proposed method requires a few lightweight structures. Furthermore, based on recent developments from these approaches, we perceive no fundamental limitations that prevent our method from being deployed alongside current in situ systems.

2.5 The Fun3D "Mars-Lander" Simulation Dataset

Our work is motivated by the Fun3D Mars Lander simulation data set containing a collection of massive retropropulsion simulations with mixed elements and various per-vertex scalar fields. The scalar fields come as an animation sequence with multiple timesteps. While the scalar fields are animated, the mesh topology (vertex position and connectivity) remains stationary over time, which is typical for such kinds of simulations [48, 49, 50].

We aim to use the exact data distribution (geometry and clusters) generated by the Fun3D system as provided [50] by NASA. Since such Computational Fluid Dynamics (CFD) simulations produce non-trivially partitioned and non-convex data, they pose significant challenges for visualization algorithms. Table 2.1 provides data set statistics of two such data sets, i.e., the small and huge Mars Lander. Our approach is not *directly* in situ but rather emulates such a scenario, and if the simulation code were adapted accordingly, it would be directly applicable to in situ strategies, simulation steering, and post hoc analysis.

Table 2.1: The Fun3D "Mars Lander" data set statistics. These data sets do not contain hexahedral elements.

		Element counts				
Model	Vertices	Tetrahedra	Pyramids	Wedges	Clusters	Size (GB)
Small	145 M	766 M	47.5 K	32 M	72	14
Huge	1.2 G	6.12 G	285 K	256 M	552	112

The data-parallel partitions of finite elements, which we call *clusters*, are typically

non-convex and come from different compute nodes touching each other at the boundaries. This has numerous implications for scientific visualization algorithms and their particular implementation. Many out-of-the-box standard *software* solutions for DVR fall short when handling such clusters [51, 52]. On top of that, data-parallel rendering usually employs a compositing scheme that works best with convex clusters. Another alternative to handle large-scale data can be compression-like big mesh compaction [53]; applications might use a data-parallel paradigm due to memory and bandwidth limitations. However, long build times and extra complexity introduced with multiple scalar fields/timesteps make it an unviable option for our purposes. Besides, in an in situ scenario, there may be insufficient memory and computation resources available to achieve that type of data wrangling.

Chapter 3

Volume Integration Using Ray Marching

This chapter describes the MPI process steps that lead to partial images generated via DVR integration. These partial images are later composited using the technique described in Chapter 4. Our volume integration scheme handles non-convex cluster boundaries and convex boundaries while ensuring robustness when it comes to discontinuities, holes, and self-intersections. We utilize a memory compaction scheme proposed by Aman et al. [17, 16] and extend it to work with other volume primitives. We also review an extension of our scheme for specific secondary effects for single GPU setups. The overview of our framework is as follows:

1. Each node generates connectivity information, shell-BVH, and XOR-compacted geometry representation required by our ray-marcher (see Section 3.1).

2. Rendering starts at each node by tracing two rays through each shell to create segments (Section 3.2). Figure 3.1 (a) illustrates the shell-to-shell traversal.

3. Each node performs volume integration (cf. Figure 3.1 (c)) via ray-marching, creating one RGBA-Z tuple; i.e., fragment per each segment, resulting in potentially multiple fragments for each pixel (Section 3.3). Figure 3.1 (d) depicts an integrated volume output using the shells from Figure 3.1 (b).

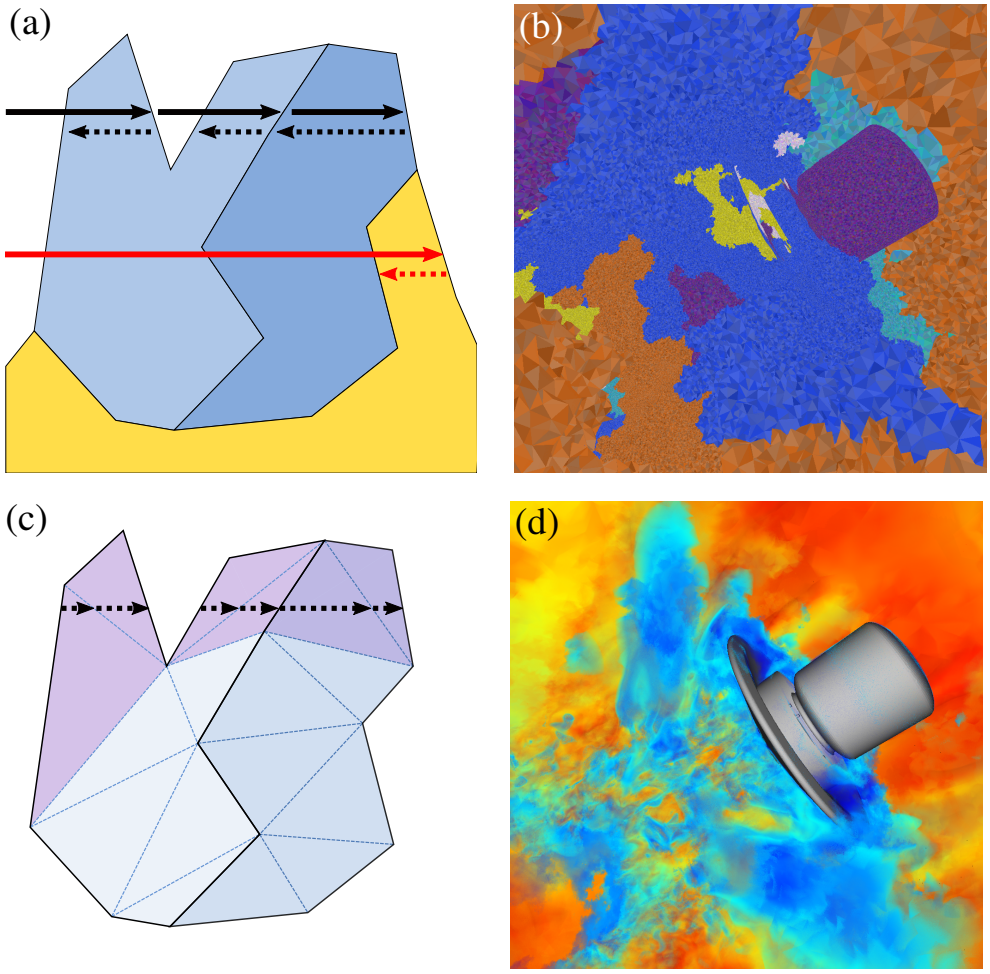

Figure 3.1: Overview of the volume integration process. (a) Shell-to-shell traversal for non-convex elements is depicted in this diagram: rays with front-face culling are sent to identify exit faces, and backward rays are cast from exit faces to seek entry faces across the shells. The same MPI rank is responsible for the blue and light blue clusters; however, the yellow cluster is in a different rank; hence, the shells of the two blue clusters are traced in order, whereas the shells of the yellow cluster are traced in parallel. (b) The shells of the first 46 clusters of Small Lander are visualized with different base colors for each cluster. (c) The ray-marching between identified entry and exit faces in (a). (d) DVR of the same subset given in (b).

4. Finally, we apply a GPU-optimized "deep compositing" technique in which different ranks exchange their respective fragments and composite them in the proper order (see Chapter 4).

3.1 Data Preparation

This section details connectivity generation, shell-BVH construction, and XOR-compaction of the geometry, which will be utilized by the ray-marcher we describe. Specifically, our ray-marcher uses this shell-BVH to find exit and entry points, XOR-compacted geometry representation is utilized to construct volume elements, and connectivity is used to select the following element index in the marching direction.

3.1.1 Connectivity Generation

Our approach needs to know the indices of the neighboring elements to execute element marching. We produce the connection information by matching the element faces as a preprocessing stage. To maintain the elements and neighbor indices aligned in memory, we separate the vertex indices and connectivity information and store the neighbor indices in an external buffer. Although we chose this method of processing connectivity, the buffer can take any shape or form as long as we can access the next element from the current element via a shared face. As a result, this component may be customized to meet the demands of a simulation or application.

3.1.2 Shell-Construction

Our approach handles convex and non-convex clusters while exploiting hardware acceleration of NVIDIA GPU's RTX cores. To construct the hardware-accelerated OptiX [54] BVH, we compute the faces that form the boundaries of the clusters. These

faces are called *shell-faces*. Shell-faces are easily identified after connectivity information is calculated since the elements with missing neighbors should indicate which of their faces lie in the boundary.

We utilize the approach described in Sahistan et al. [2] for shell construction. Since our method handles mixed volume elements other than tetrahedra, we also process quadrilateral faces by triangulating them. The Optix BVH we build over shell-faces requires an index list that points to a vertex list. We group three previously identified triangle indices with the encoded indices of the elements behind the shell faces facilitating ray-marcher with the entry element's index. So the first three vertex index that belongs to a triangle is used to create a bounding box for the OptiX framework; the last index stored alongside the first three allows swift access to the volume element behind during marching. The lower two bits of the fourth index signify the element type (i.e., tetrahedron, pyramid, wedge, or hexahedron), and the remaining 30 bits is an index into the list of elements; this index is required to start marching. This encoding is similar to the BVH-node memory layout used by PBRT [55].

3.1.3 XOR-Compaction of indices

Exclusive-or (XOR) is a bitwise logical operation we exploit in our geometry representation. We call this process xor-compaction and use the following property of XOR operations: $(a \oplus b) \oplus a = b$ where a and b are values of the same number of bits. So this can be easily generalized to n integers if we know the XOR of $n-1$ integers. Specifically, we can utilize this idea if it is guaranteed that we can access the specific data values in order like doubly-linked-lists [57] or tetrahedral mesh traversal via ray-marching [17, 16]. We propose a memory compaction scheme that exploits this property of XOR operations for mixed element meshes that may include tetrahedra, pyramids, wedges, and hexahedra. We can use previously calculated XOR fields to minimize index information per element because some of the vertices are shared between neighboring elements. This scheme requires the previous element to be known; this can easily be obtained from shell-BVH. After the first element is constructed, each

```
struct Tet{
 uint vx;
};
```

```
struct Pyr{
 uint dx;
 uint diag[2];
 uint top;
};
```

```
struct Wed{
 uint dx[2];
 uint diag[2];
};
```

```
struct Hex{
 uint v[8];
};
```

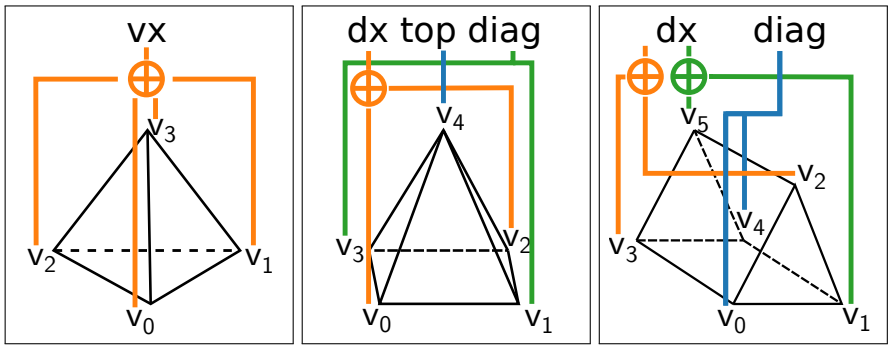

Figure 3.2: XOR-compacted memory layouts (top) and geometric illustrations of XOR calculations for `Tet`, `Pyr`, and `Wed` (bottom). `Hex` does not have a compaction scheme. `uint` stands for unsigned integer. The "$\oplus$" symbol indicates the XOR operation. The total sizes of each struct are 4, 16, 16, and 32 bytes for Tet, Pyr, Wed, and Hex, respectively. The vertices are in VTK [56] ordering.

step utilizes the march state to access the previous step's information. We store a different XOR-compacted structure for each element, except hexahedra (cf. Figure 3.2).

The tetrahedra compaction results in one int size structs where the only field is `vx` which is the XOR of all four vertex indices — $\texttt{vx} = v_0 \oplus v_1 \oplus v_2 \oplus v_3$ where v_0, v_1, v_2, v_3 are the vertex indices —. Assuming we know the exit face from the current element, and since any element neighboring tetrahedra share three of their vertices with it, we can use this 4-byte `vx` field to get the unshared vertex index by XOR'ing the three shared vertex index of the current element and `vx` of the next tetrahedron. This scheme allows us to reduce a naïve 16-byte tetrahedron to 4 bytes.

The struct size would be 20 bytes with a naïve pyramid implementation since the pyramid has five vertices. Our scheme reduces the pyramid's memory requirement to 16 bytes. We store one `dx` field that is the XOR of 0^{th} and 2^{nd} vertex indices (according to VTK ordering), two vertex indices which happens to be the other diagonal of the quad (1^{st} and 3^{rd} vertices), and a top vertex index, which is always the 4^{th} vertex (cf. Figure 3.2). To construct the next pyramid first, we check the entry face type to the pyramid. If it is a quad face case is simple since the unknown vertex index is stored at the `top` field. Otherwise, every triangle face should be made up of one of the diagonal vertices explicitly stored, the top vertex, and one vertex encoded in the `dx` field. We can determine which index to XOR with `dx`, thereby retrieving one of the missing vertices by matching one of the vertices with one of the diagonal fields. The other missing index for this case is the unmatched integer from `diag[2]`.

Our 16-byte wedge struct is 8 bytes smaller than the naive implementation. It is composed of two `dx` and two `diag` fields. `dx` fields contain two XORs: the first one is the XOR of 2^{nd} and 3^{rd} vertex indices; the second one is the XOR of 1^{st} and 5^{th} vertex indices. `diag` explicitly stores 0^{th} and 4^{th} vertex indices. Construction of the next wedge during traversal starts similar to pyramids where we first classify the entry face. If it is a triangular face, we know that both triangle faces are made up of explicitly stored diagonal vertex indices and two vertices encoded in the different `dx` fields. The last vertex index that needs to be fetched for this case is the other integer from `diag[2]`. If a ray enters the wedge by a quadrilateral face, it must contain one or both of the indices stored in `diag`. So by matching diagonal vertices, we can understand which

one of the three quadrilaterals we entered through. So, this case boils down to two sub-cases. In the first sub-case, the ray enters from the quadrilateral, matching two vertex indices in `diag`, so we have two missing vertices. To get those, we can XOR `dx` fields with unmatched indices of the entry quadrilateral two times. In the second sub-case —either one of the diagonals match with one of the quadrilateral face indices—, we can immediately get one of the missing vertices from unmatched `diag`. Finally, we can obtain the other missing vertices using one of the `dx` variables.

Since hexahedra have the worst ratio of shared vertices over all vertices (0.5), it is challenging to realize a similar XOR-compacted scheme. Although it might be possible to reduce down a couple of bytes from this structure, the naive size of 32 bytes gives better memory alignment when it comes to performance. Therefore, we choose to store all hexahedra indices without compaction according to the VTK mesh ordering.

3.2 Ray-segment Generation via Shell-to-Shell Traversal

Many real-life data sets contain non-trivially shaped non-convex cluster shells with holes and jagged edges. On top of that, these volumes are designed for simulations; therefore, visualization is a secondary concern. For this reason, the outputs of these programs tend to have tiny overlaps and discontinuities that may throw off element-based ray-marchers, as we propose. Our marcher requires correct entry and exit information to start and end properly. The traversal passing the wrong parameter may result in infinite loops, incorrect early terminations, and undefined behaviors. Additionally, as ray-segments boundaries define the fragment compositing order, false identification of the entry and exit positions can also disrupt the compositing scheme we propose, yielding an incorrect final image. To overcome these robustness issues, we offer a shell-to-shell traversal scheme.

Each MPI rank identifies an exit and entry point using hardware-accelerated tree

traversal and intersection for the given shell-BVH of the cluster. This concept is an extension of Sahistan et al. [2], yet this approach handles entry to other types of primitives than tetrahedra. Figure 3.1 (a) illustrates this process, and the steps are as follows:

1. We trace the ray through the shell-BVH with front-face culling from the ray origin.
2. If a ray hits a shell face, we then mark that face as the exit face and create a backward ray with the origin at the hit position.
3. We retrace this backward ray using front-face culling to find an entry face.
4. The found entry face contains four index values (cf. Section 3.1.2), and the last one encodes the ID and the type of the element from where we start our ray-marcher (cf. Section 3.3).

If we were to naïvely to find the closest hits to degenerate volume boundaries where two neighboring faces might be intersecting or slightly apart instead of tightly interlocking, it would create incorrect ray segments, thus causing sampling and compositing errors. Casting two front-face culled rays first to find an exit and then an entry face allows us to handle traversal in real-life data sets robustly. We also check shell IDs of entry and exit faces and the distance in-between them to improve the robustness further. If the shell IDs of entry and exit faces intersected by the same routine do not match, or the distance in-between is smaller than a certain Δ threshold, we look for the next closest intersection for the entry face.

3.3 Ray-segment Volume Integration

The segments generated by shell-to-shell traversal (cf. Section 3.2) needs to be sampled. For each segment, rays sample equidistant points via linearly interpolating per-vertex scalar values from the element containing them. Our method leaps from element to element until the element that contains the sampling point is reached. The linear interpolation coefficients are also used to check for point containment. If not all of these

coefficients are between 0 and 1, we keep marching until that becomes the case. When a sample is taken, a transfer function calculates transparency and color information (RGBA) for that scalar value. The current sample's RGBA values are composited over the segment's cumulative RGBA value. We terminate marching for that segment when the ray goes opaque —early ray termination— or segment ends. Early terminated rays are not followed up for that MPI rank; however, since depth ordering is unknown among computer nodes before compositing, a copy of an early terminated ray may be traced in another node. Although this might be wasteful, it is not a notable concern in the distributed rendering domain, where more significant performance bottlenecks are present.

Ray-marching processes begin with a cluster's shell, comprising element information encoded in pbrt-style [55] format. We can construct the first element beneath the shell face using this information. After entering the shell, the connection buffer and compressed element information are sufficient to retrieve and create the following elements along the given ray segment. On the other hand, our compaction necessitates traversing components sequentially without skipping. Since this causes sampling artifacts, the vertices cannot be in any order when an element is reconstructed from the XOR-compacted form. As a result, our technique not only re-obtains vertex indices for each element but also consistently positions them according to VTK mesh ordering [56].

To pick the next element in the ray's path, we check intersections with the element's faces. To select the intersected face (exit face) for a given element, our marcher uses an approach similar to "Projected Tetrahedra" [8]. However, we do not rasterize the elements directly to the screen; instead, we use these projections to determine exit faces, which is in line with Aman et al. [16, 17] and Sahistan et al. [2]. We do, meanwhile, deal with primitives other than tetrahedra. The intersection routine projects the element vertices to a ray-centric coordinate system to conduct 2D intersection tests to find the exit face. Projection and 2D tests yield fewer floating-point operations compared to 3D intersection tests.

We also maintain a *march state* that bookkeeps the last intersected face type (triangle or quad), current element's type, index, and vertex indices for every marching step.

Moreover, while traversing the volume, we follow a general rule of placing the entry face indices in the same positions in the march state during marching. The rearranged vertices allow us to ignore the entry face while selecting the exit face (since we now know which vertices belong to the entry face). When the marcher advances to a new element, we replace the previous entry face indices with exit face indices in the march state. It is challenging to create an *simple* algorithm that can handle all conceivable combinations because each volume element is unique in terms of shape and face lay-out. As a result, our element marching deals with different elements on a case-by-case basis.

Tetrahedral elements are handled similarly to [2]. However, we enable intermediate places within the elements to be sampled, unlike Sahistan et al. To find the exit face for tetrahedral elements, we transform the vertices to the previously specified ray-centric coordinate system. After transforming each vertex, we use a maximum of two 2D left tests to find the face containing point $(0,0)$.

2D left tests can be used more within a pyramid to find the exit face. We use the last intersected face type information kept in march state to simplify our left test cases because the pyramids have one quad face. If the entry face is a quad face, we determine the exit face among four triangles using projected vertices (similar to the tetrahedron case). Otherwise, we check if the quad face has a point $(0,0)$ before testing the other three triangles for the same condition. Finally, we update the last intersected face type accordingly.

Wedges are similar to pyramids in terms of determining the exit face. It is worth mentioning that wedges have three quad faces; therefore, the exit face might be another quad face, even if the intersected face type is a quad. We disregard the entrance face to eliminate unnecessary left checks, as with other element types. We update intersected face type once more after determining the exit face.

Hexahedra, like tetrahedra, are uniform but have more faces than tetrahedra. As a result, finding the exit intersection demands the most left tests. Hexahedra require 13 left tests in the worst-case scenario, whereas wedges, pyramids, and tetrahedra require 7, 5, and 2 left tests, respectively.

3.4 Extention to Single-node Secondary Effects

The application of secondary effects may increase depth perception and allow more information to be conveyed about the phenomena depicted in the frame. The ray casting scheme described in this chapter can be generalized to other types of rays that achieve secondary effects. Since the data-parallel paradigm distributes volume pieces into many nodes, it complicates the generalization of the secondary effects. Although attaining the described effects for distributed environments is possible, the scope of this thesis does not include this concept. Hence we refrained from this discussion.

As a proof of concept, we implemented some secondary effects that will only work on single GPU setups. These effects are volumetric gradients (cf. Section 3.4.1), shadows (cf. Section 3.4.2), and ambient occlusion (cf. Section 3.4.3). Experiments conducted for this section are performed on a workstation with an Nvidia Quadro RTX8000 GPU and Ubuntu 18.04.

3.4.1 Gradient Calculation

Gradients are commonly used as surface normals for local shading. As our method supports traversal starting at arbitrary origins, central-difference gradients [21, 22, 23] can be computed by marching six rays in orthogonal directions starting at the sample position, giving us accurate, high-quality gradients even if the gradient sample positions Δx fall outside the tetrahedron that the current sample is inside. Figures 3.3 and 3.4 illustrate this effect using Phong model.

3.4.2 Volumetric Shadows

Another technique to increase depth perception is using arbitrary ray tracing to produce shadows, allowing volumes to cast shadows on surfaces. We use shell-BVH to cast a shadow ray from a given position to determine the illumination that reaches that spot. If the shadow ray collides with a shell, we begin marching in tetrahedra to collect

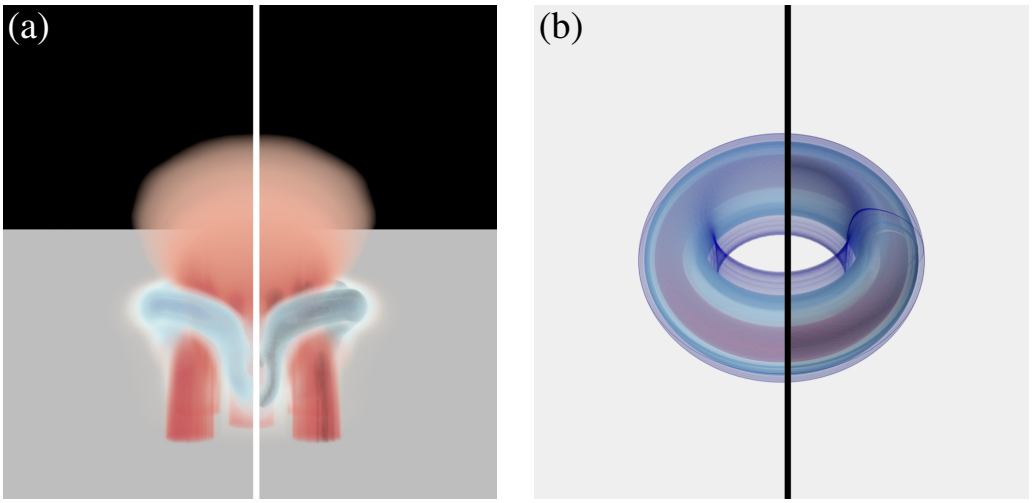

Figure 3.3: Gradient-shaded depth cues: (a) Jets dataset with shading off (left) and on (right) (113.3 fps vs. 70.45 fps). (b) Fusion dataset with shading off (left) and on (right) (87.3 fps vs. 12.2 fps).

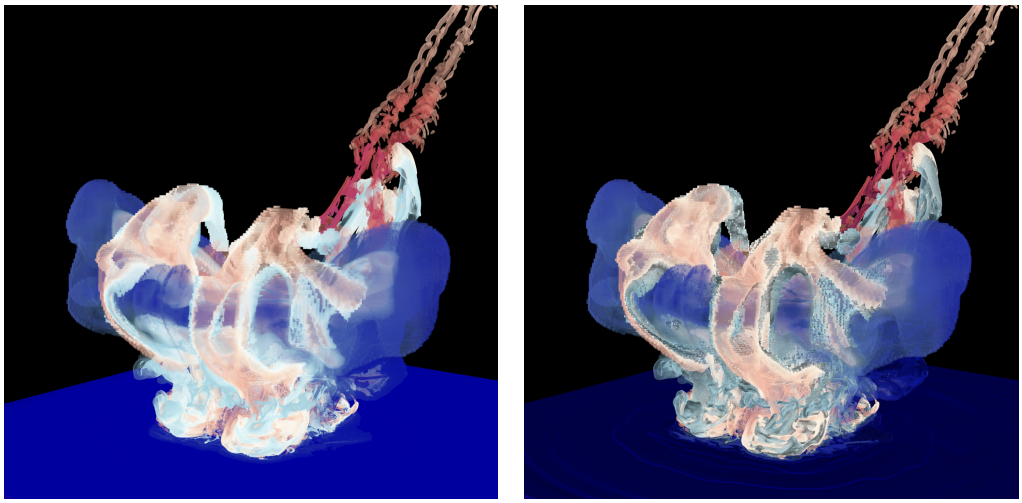

Figure 3.4: The Impact dataset. Left: with emission and absorption. Right: with gradients.

transmittance. If a shadow ray intersects a surface, we assume the surface is fully opaque and cut the illumination off.

When a volume and surface mesh intersect, we cannot just start from the tetrahedron containing the incident point as marching needs to start at a shell-face. However, we can exploit the fact that the sequence of tetrahedra on the shadow ray's path is irrelevant because we are just interested in transmittance, not brightness emitted from the volume. Therefore, we begin marching backward, starting from the closest back-facing shell to the light. Rather than advancing in the direction of the shadow ray, we advance in the direction of the actual light (that reaches the incident point). If the light source is inside the volume, we begin to accumulate transmittance after passing by the light's position. Figures 3.5 (c, f) display this effect.

3.4.3 Ambient Occlusion

Ambient occlusion (AO) can help even more with depth perception and overall rendering quality. We use the standard ray traced AO method, for example, proposed in [19, 20]. Tracing the required shadow rays is technically very similar to tracing rays towards point light sources located at a distance of r. We compute AO by averaging N hemisphere samples, the effect of which can be seen in Figures 3.5 (b, e).

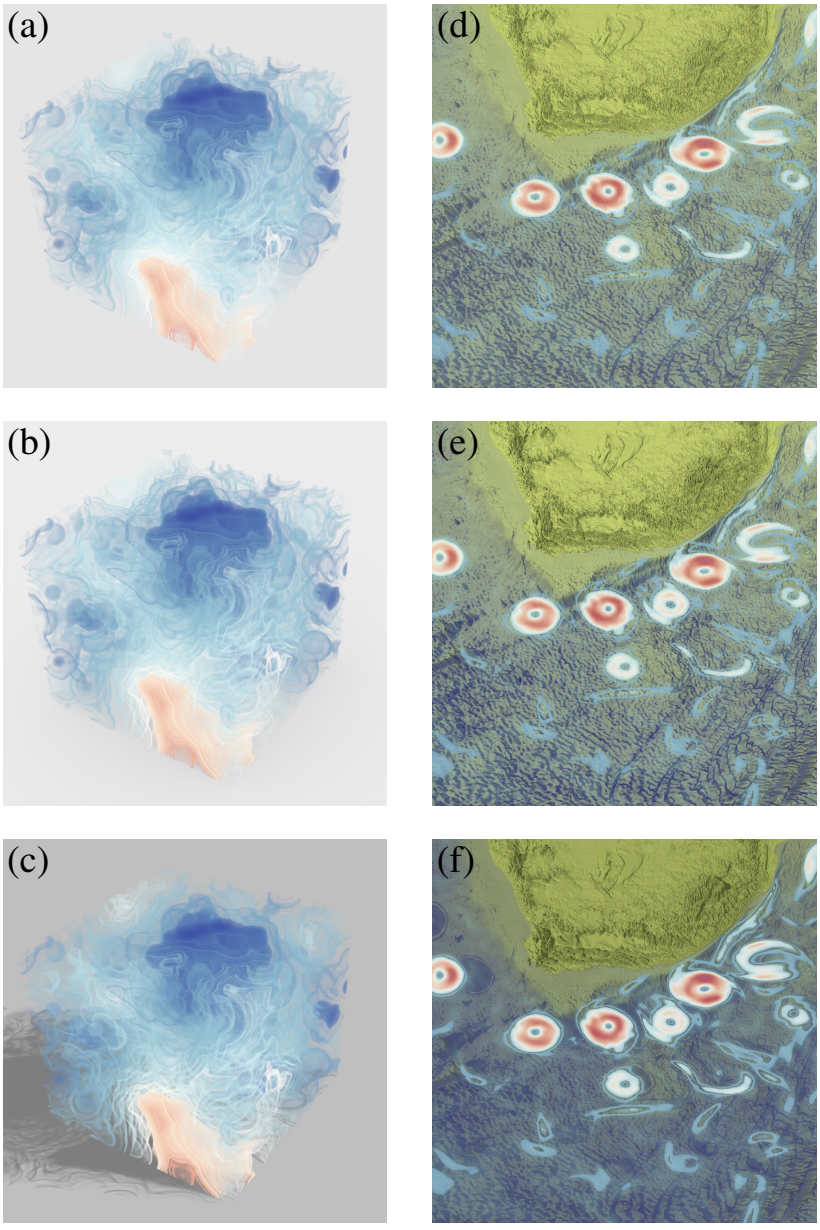

Figure 3.5: Shadows and ambient occlusion: (a) Plasma64 dataset rendered with emission and absorption at 141.5 fps, (b) with AO at 32.3 fps, and (c) with shadows at 104.1 fps. (d) Agulhas dataset rendered with emission and absorption at 42.4 fps, (e) with AO at 4.5 fps, and (f) with shadows at 30.9 fps.

Chapter 4

Deep Compositing

In Chapter 3 we describe volume integration process via ray marching. Specifically, Section 3.2 describes how each rank finds entries and exits to volume segments and Section 3.3 discusses how each rank integrates calculated segments, which returns color and transparency tuples (RGBA values). In the scientific visualization domain, the renderings must be accurate and should not contain any artifacts to allow users to make correct observations. Efficiently determining the correct order and compositing those tuples is a challenging task. While depth sorting convexly shaped clusters before or after the integration offers a viable solution, this is not the case for non-convex clusters. Rendering non-convexly shaped clusters may produce more than one segment per ray, which disallows us to make assumptions we would have for the convex clusters, thus further complicating the compositing process for many real-life datasets. To address these challenges, we propose a deep compositing algorithm that takes a *fragment* which is the RGBA tuple plus the depth of the entry position to that segment and composites them for every pixel in the correct order in a GPU-efficient manner.

To further understand the compositing order problem consider pixel P and all of its fragments $F_0^{(P)}$, $F_1^{(P)}$, ..., $F_{N^{(P)}}^{(P)}$. The correct final color of P can be obtained by first sorting these fragments by their depth and compositing them using over//under operators $\widehat{O}(A,B)$ / $\widehat{U}(A,B)$ [58, 59]. Since our method may produce each given pixel's fragments on several ranks, the sorting cost is bounded by the communication costs of

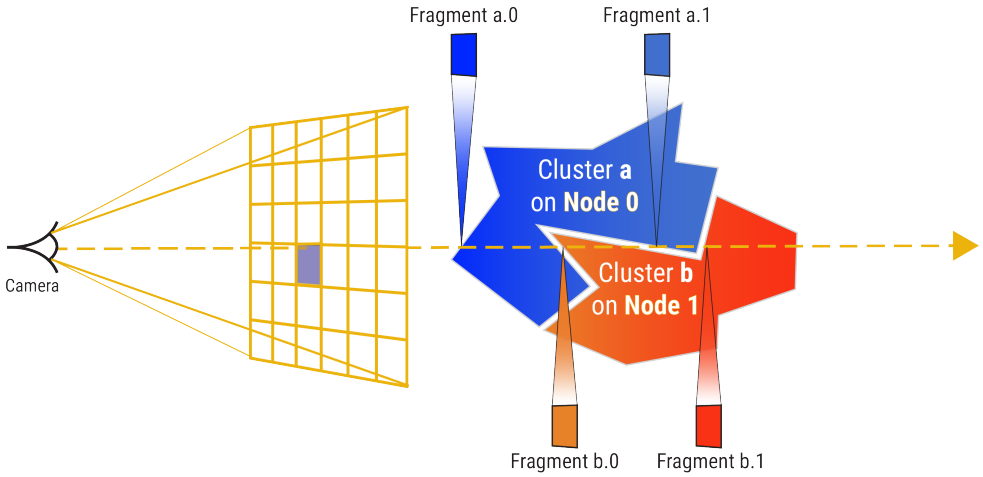

Figure 4.1: An illustration of the fragment generation process on an example scene of two non-convexly partitioned clusters distributed across two nodes.

each rank. Even worse, the irregular shape of the shells means that any ray can enter and leave the same shell multiple times at multiple distances, producing multiple—and in some cases, many—fragments for the same pixel. Figure 4.1 illustrates the problem our compositor tackles by providing an example scene that generates interleaved clusters distributed across different nodes. Figure 4.2 shows how the average and the total number of fragments are distributed for a view of the Huge Lander with growing rank counts. As it can be observed from the case where the rank count is 16,—depicted in Figure 4.3— each pixel may have more than one fragment generated from multiple ranks.

As previously stated, simply depth sorting fragments on a single fragment does not offer a feasible solution to our non-convex shells. Therefore, industry-standard compositors like IceT [31] that utilize a single fragment for compositing can be considered a naïve implementation. For our needs, the compositing needs to happen in the visibility order of the fragments and not the clusters. Let $\otimes$ denote compositing operation given any ray that produces two fragments $F_0^{(A)}$ and $F_1^{(A)}$ on the same rank must also have had at least one other fragment $F^{(B)}$ on at least one other rank. One way to visualize this is by considering a jagged shell of internal cluster A loaded on a rank. The implication here is that the cluster(s) —in this case, cluster B— that fit into the jagged cavity may be loaded on a different rank and rendered separately. This requires deep compositing as $F_0^{(A)} \otimes F^{(B)} \otimes F_1^{(A)}$, which in general is different from both

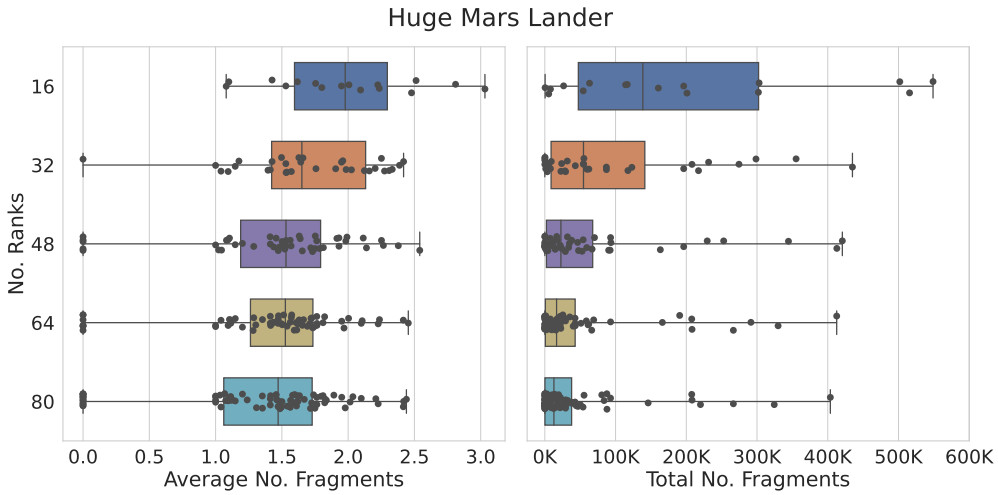

Figure 4.2: Box plots of average (left) and total (right) number of fragments generated by individual ranks while rendering the Huge Lander. We take averages over non-empty pixels where their opacity is greater than 0. The plots are for the rank counts of 16, 32, 48, 64, and 80. Scattered points signify individual ranks' average (left) and total (right) fragment counts at a given MPI size.

$F^{(B)} \otimes (F_0^{(A)} \otimes F_1^{(A)})$ and $(F_0^{(A)} \otimes F_1^{(A)}) \otimes F^{(B)}$. The following sections describe our deep compositing algorithm in detail that solves this problem using GPUs efficiently.

4.1 Multiple Fragment Compositing

We describe a compositing scheme that allows multiple fragments to be composited per pixel. A counter representing the number of fragments N and a pointer—-or offset—- to a list of fragments $F_0, \ldots, F_{N-1}$ are stored for each pixel. In a likeness to parallel-direct-send [60, 61], we then divide the frame buffer into R distinct *regions* of pixels—sub-screens—(where R is the number of ranks); each rank will be responsible for receiving, compositing, and delivering the final composited results of one region of pixels. Compositing starts when each pixel's Fragment lists are collected. Then it works in the following steps:

1) Generating a contiguous send buffer. Each rank computes a GPU-parallel prefix sum over all its pixel's fragment counts. This computation also yields the total number

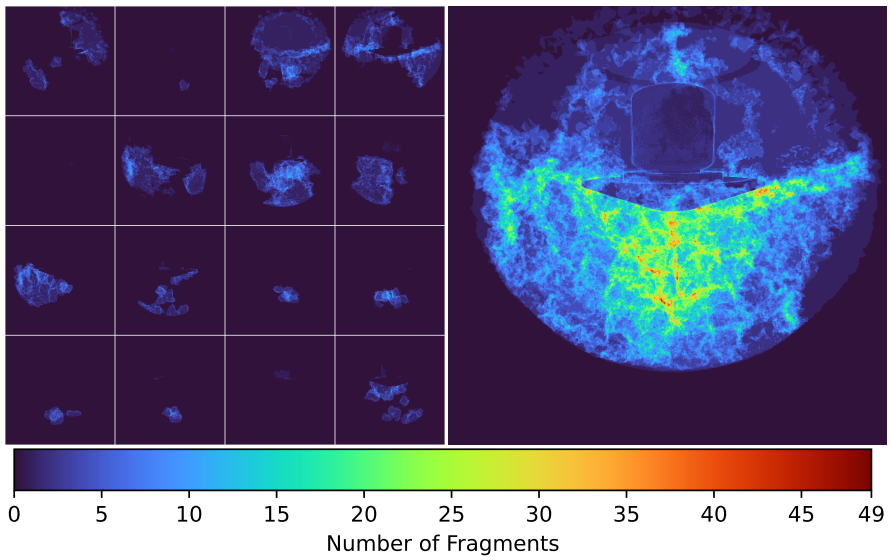

Figure 4.3: Heatmaps for the number of fragments for a view of the Huge Lander rendered with 16 MPI ranks (first box given in Figure 4.2): On the left, fragments for every 16 rank, and on the right, all heatmaps combined into one image.

of fragments on this rank. Using the prefix sum result as offsets, we allocate a single contiguous memory region for these fragments. Then we compact the individual fragments into this buffer. These buffers are organized so that each rank will contain all fragments going to all other ranks in order.

2) Exchanging per-pixel fragment count ranges. Using the assigned range of pixels—computed in the previous step—, each rank calculates the range of per-pixel counters it needs to send to any other ranks. Each rank allocates a *per-rank counter buffer* with a size R times the number of pixels in a sub-screen. So we end up with a large buffer to house a count from each rank for all its pixels. Next, each rank computes the offsets to store the counters from other ranks. We then execute a collective `MPI_Alltoallv` on these buffers, after which each rank has, for its assigned region of pixels, the fragment counts from every other rank. With this information, we know how many fragments will be received from each rank for any given rank.

3) Exchanging Fragment Lists. Having received all other ranks' per-pixel fragment counts for its sub-screen, a GPU prefix sum over those counters performed by each rank, which can be seen as offsets into a compact buffer of all fragments for its range of

pixels. Looking up the prefix sums at the correct offsets specifies how many fragments each rank will receive from any other rank and how many fragments it will receive altogether. Next, we create a receiving buffer of the necessary size, determine where each rank's fragments will go in this buffer, and then issue a second `MPI_Alltoallv` that moves each fragment collectively into the receive buffer of the rank assigned to the sub-screen that contains matching fragment's pixel.

4) Local Compositing. After the previous step, each rank now has two buffers containing all fragment lists for its assigned pixels. One of the buffers —*fragment buffer*— stores all fragments for that rank's pixels received from all other ranks, ordered by rank number and pixels within each rank. So for a specific MPI rank, this buffer stores all fragments for that rank's first pixel from rank 0, then all those for its second pixel from rank 0, and so on, followed by all fragments from rank 1, then all fragments from rank 2, and so on. The other buffer —*offset buffer*— stores the results of prefix sum operations. It works as a look-up table that provides the offsets to where the fragment list starts. For instance, if P is the number of pixels for which this rank is responsible, the fragments from rank r for pixel j start at index `offsets[r*P+j]`. We launch a CUDA kernel that, for each pixel p, looks up the R lists of fragments and composites them in the visibility order using the indexing arithmetic.

5) Sending final results to master. Now, each rank houses a fully composited RGBA value for their assigned sub-screen. To generate a complete image, we send these to the master using a `MPI_Send`; the master sets up R matching `MPI_Irecv` s, that writes to the appropriate part of the final frame buffer. After this, a final image is ready to be displayed.

This method is a natural extension of the parallel direct-send technique described by Grosset et al. [60] and Favre et al. [61], with the main difference that we not only send one fragment per pixel but variable-sized lists of fragments. We term this method *deep compositing* because it merged the concepts of image-based compositing with the orthogonal concept of *deep frame buffers* [62].

4.2 Fragment List Management

While the compositing alone is simple to use from the host side, adequately setting up the device-side inputs (fragment lists and counters) will necessitate the renderer to handle device-side dynamic memory allocations for the per-pixel variable-size fragment lists throughout rendering.

We also created a *device interface* for this library, which allows a renderer to simply *write* new fragments into a pixel while the interface takes care of the proper storage of those fragments — greatly simplifying the rendering code. This interface relieves the renderer of this low-level fragment list management.

4.2.1 Two-Pass, Flexible-length Fragment Lists

The primary obstacle in creating this interface was the inability to increase device memory allocation during rendering. Therefore we needed to set a cap on the number of pieces a renderer could produce in a frame. First, we created a two-stage interface that would execute the renderer twice. The interface's first step would merely count the number of pieces produced for each pixel, with no storage. After this stage, constructing a large enough buffer would compute a prefix sum over those counters, with the prefix sum values acting as offsets inside the buffer. We could perform the identical rendering in a second pass, but we would store the pieces at the specified offsets this time.

4.2.2 Single-Pass, Fixed-Length Fragment Lists

The two-pass solution involves performing the shell traversal at least twice, which may or may not be acceptable but allows for arbitrary-sized fragment lists (up to device memory, obviously). Therefore, we also created a second, single-pass device interface where the renderer specifies the maximum permitted amount of fragments per pixel before startup. We can utilize this information and preallocate lists to add

fragments. Performing a single pass is straightforward but requires some form of *overflow*-handling if a render wants to submit fragments to a pixel whose list is already complete. We currently implement two methods for this overflow handling: In the *drop* method, we perform insertion sort into the existing list and drop the latest fragment. In *merge*, we find the fragment with the lowest opacity and perform a *over* compositing of this element onto the one in front of it (i.e., using the depth from the previous one), then insert the new fragment into the list.

Chapter 5

Experimental Results

We conducted our experiments on Frontera RTX nodes of Texas Advanced Computing Center (TACC), where each of the 22 nodes had four NVIDIA Quadro RTX 5000 plugged into it. We utilize all four GPUs available per node for every data point of our experiments.

5.1 Evaluation of the Framework

We evaluate our rendering framework on Small Mars Lander and Huge Mars Lander data sets. Figure 5.1 shows images of the Small Mars Lander rendered using our framework. Since Small Mars Lander has 72 clusters, we evaluate our framework using 72 GPUs distributed over 18 compute nodes where each cluster is loaded on a separate GPU. For Small Mars Lander, we achieve our peak performance using 72 GPUs yielding the average frame rate of 14.35 frames per second (fps). Since the TACC supercomputer does not have more than 22 RTX nodes, we could not test one cluster per GPU scenario for the Huge Mars Lander data set. Therefore, we scale up to a maximum GPU count of 88, yielding 9.83 fps. However, we observe our average peak performance of 10.25 fps for the Huge Mars Lander at 80 GPUs.

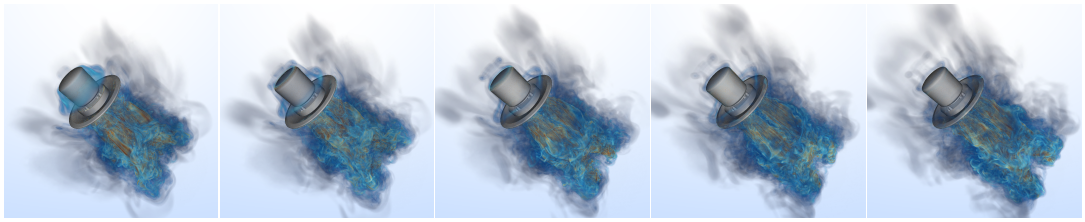

Figure 5.1: Data-parallel rendering of the "Small Mars Lander" computational fluid dynamic data set, which comprises 72 clusters of finite elements. The images are rendered on 18 compute nodes and 72 NVIDIA Quadro RTX5000 GPUs of the Frontera RTX partition at the Texas Advanced Computing Center (TACC). The image resolution is 1024×1024. The average frame rate of this image sequence is around 15.00 fps.

Moreover, we evaluate our deep compositing scheme's correctness compared to a single fragment compositing technique. Figure 5.2 shows an image rendered by single fragment compositing and a heatmap that compares the difference between single fragment compositing and our deep compositing. The single fragment compositing method depicted is similar to the image-based single image per node compositing techniques such as IceT [31].

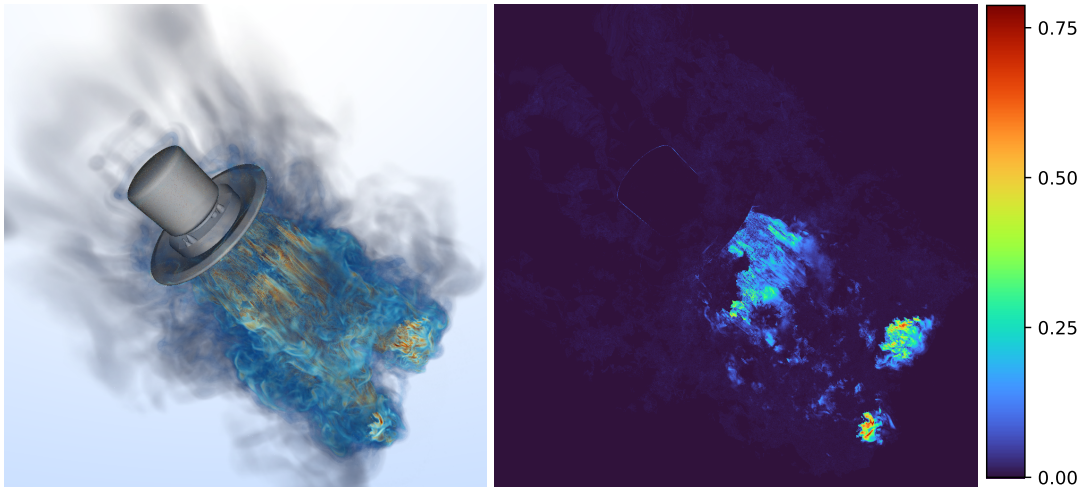

Figure 5.2: Comparison of single fragment compositing and deep compositing. The left image is the rendering with single fragment compositing (similar to IceT [31]). The right image is the heatmap showing the $L2$ difference between single fragment compositing and our deep compositing.

5.2 Memory Overhead

We examine data distribution and memory footprints. Table 5.1 displays the minimum, maximum and average counts per rank of volume elements and shell faces for our two data sets. Figure 5.3 illustrates the average memory footprints of our extensive data structures that may not be present in a simulation environment. Although connectivity information will likely be in most simulation systems, we wanted to include connectivity here for simulation systems like [63]. The MPI sizes (rank counts) in the table are the sizes that experience the highest level of changes in rendering times presented in Figure 5.4.

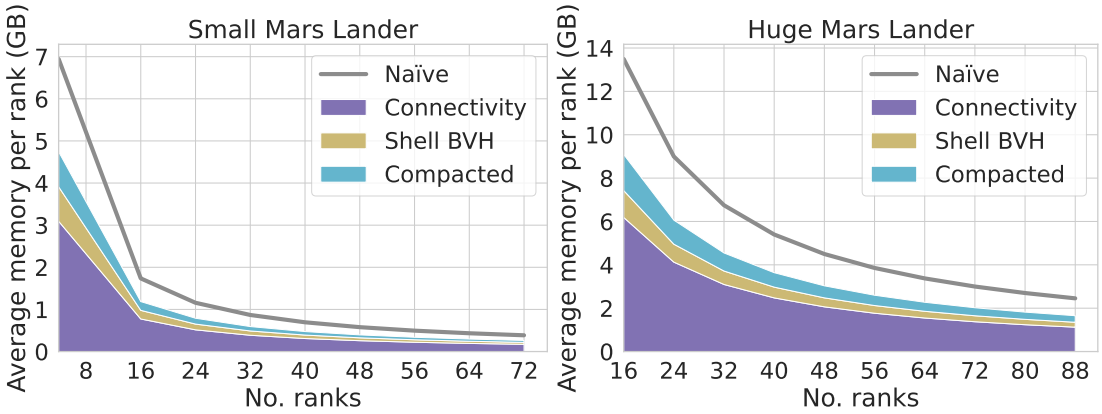

Figure 5.3: Per rank average memory consumption of various buffers for increasing MPI sizes: Small Mars Lander (left) and Huge Mars Lander (right). Average memory usage of XOR-compacted elements is stacked over average shell-BVH size, which again is stacked over average connectivity buffer size, providing the total memory usage introduced by these data. We also include a line that indicates the per rank average memory usage without XOR-based compaction (size of Shell-BVH + connectivity buffer + non-compact elements).

5.3 Scalability

To assess the scalability, we test our approach for increasing the number of ranks (MPI sizes). We also measure sub-process timings, specifically the segment integration and

Table 5.1: Per rank statistics for selected MPI sizes for two test data sets. The first column gives the MPI size with a number and data set as "small" for Small Mars Lander and "huge" for Huge Mars Lander. Columns depict minimum, maximum, and average volume element counts and minimum, maximum, and average shell face counts per rank.

MPI Size	Elements			Shell		
& data set	min.	max.	avg.	min.	max.	avg.
4-small	192.2 M	203.4 M	199.6 M	14.4 M	16.2 M	15.2 M
24-small	30.8 M	34.9 M	33.3 M	2.0 M	2.9 M	2.5 M
40-small	10.0 M	23.3 M	20.0 M	0.8 M	1.9 M	1.5 M
72-small	9.9 M	11.9 M	11.1 M	0.6 M	1.1 M	0.8 M
16-huge	365.2 M	414.6 M	399.2 M	21.4 M	25.1 M	23.0 M
32-huge	177.8 M	213.8 M	200.0 M	10.3 M	13.2 M	11.5 M
56-huge	93.8 M	121.0 M	114.1 M	5.4 M	7.3 M	6.6 M
88-huge	60.9 M	85.1 M	72.6 M	3.3 M	5.3 M	4.2 M

compositing times. We compute them using `MPI_Barrier` s before and after integration calls to synchronize the processes before starting and ending the timers. Since these barriers stall early terminating integration processes, it increases the total rendering time. For this reason, we measure the total rendering time and integration time in different runs and derive the compositing time by subtracting the total time from the integration time. We calculate the timings reported in Figure 5.4 by taking an average for 20 sequential timesteps of the selected scalar field over 30 runs. Then we take the mean of these 20 average values to form the data points for the given MPI sizes.

We compare our integration algorithm with a state-of-the-art OptiX accelerated point query technique by Morrical et al. [7]. In order to fairly compare the two methods' volume integration steps and their scalability on the setup presented, we use the same sampling rate and fragment distribution as well as the same compositor —our deep compositor—. Alongside fixing the sampling rate, we also disable the empty space skipper to allow the compositor to operate on the same input. A direct comparison of total rendering times of a frame for an increasing number of GPUs is given in Figure 5.5. Speedups for the same results are given in Figure 5.6. The method of Morrical et al. relies on an external hierarchical data structure to query the points, and the memory cost of those data structures overflowed the memory for the GPU counts between 16 and 40. Since they do not claim their method is in situ, we did not generate element connectivity when running the respective method's benchmarks.

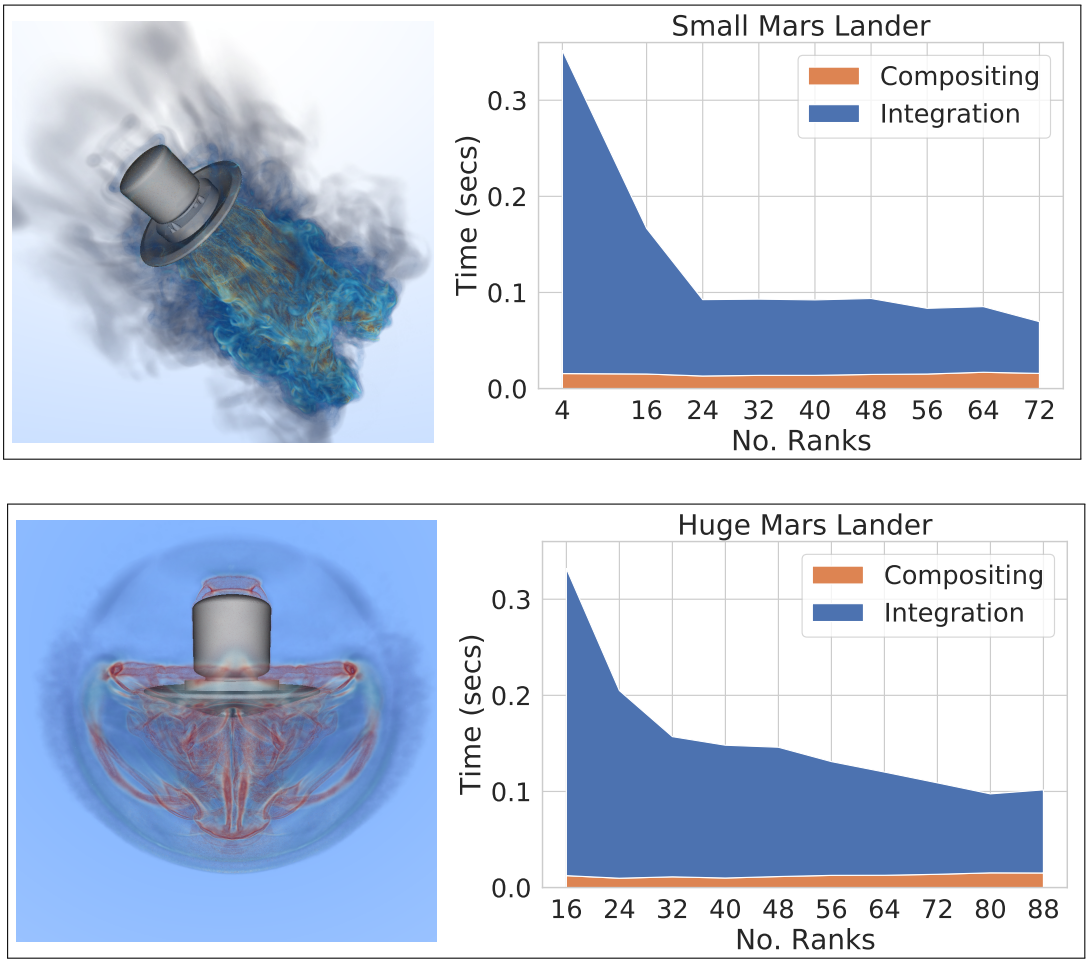

Figure 5.4: Results of the scalability benchmarks for Small Mars Lander (top) and Huge Mars Lander (bottom) on RTX nodes of the Frontera system on TACC. Integration process timings are stacked over compositing process timings for the given number of ranks to form total rendering times.

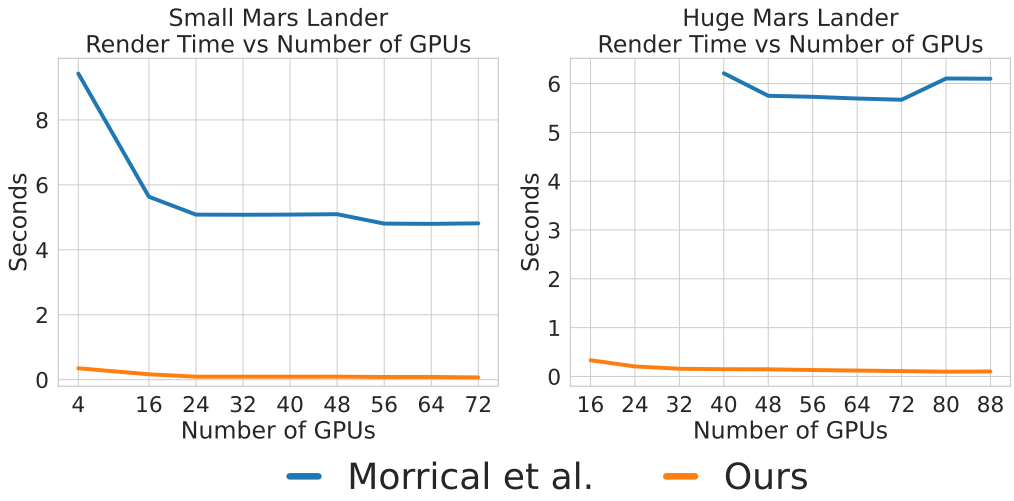

Figure 5.5: Comparison of total rendering times between a state-of-the-art hardware-accelerated point query method by Morrical et al. [7] and ours for an increasing number of GPUs. Both use our deep compositor to composite nearly identical fragments.

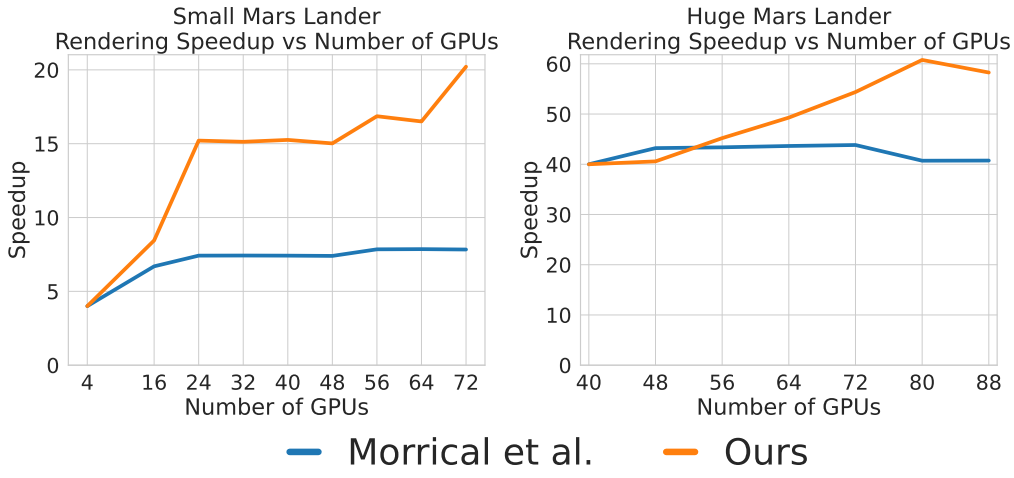

Figure 5.6: Comparison of speedups between a state-of-the-art method by Morrical et al. [7] and ours for an increasing number of GPUs. We compute the speedup values with respect to 4 GPUs for the Small Mars Lander and 40 GPUs for the Huge Mars Lander.

5.4 Fragment Distribution

We measure fragment counts generated by each GPU during ray segment generation because it is crucial to assess workload distribution across compute nodes for compositing and integration steps. Figure 4.2 depicts a box plot showing the total and average fragment counts for specific MPI sizes on the Huge Mars Lander. We also visualize a series of "heat images" for the case where the rank count equals 16, illustrating the fragments composited for a view of the Huge Mars Lander. Each small image shows given nodes generated fragment counts, and the final image accumulates them on top of each other.

5.5 Discussion

We evaluate our approach regarding memory consumption, rendering correctness, and scalability. Among the data we precompute and store, the connectivity information takes up the lion's share with ratios around $\approx 62.72\%$ for our test cases. We expect this since our approach stores one integer for all faces of a given volume element. XOR-compacted volume elements are the second-largest structure with ratios around $\approx 17.11\%$, closely followed by Shell-BVH sizes ($\approx 12.74\%$) of the total memory consumption. We observe that the total memory footprint of XOR-compacted representations is $\approx 72.49\%$ smaller than their uncompacted versions. The presented framework is memory-wise compatible with in situ scenarios because many modern simulation systems already store connectivity information. Our XOR-compaction reduces the space required for geometry information, and our shell-BVH sizes stay relatively small despite the large counts of shell faces. When rendering Huge Mars Lander, this memory reduction allowed us to test with even fewer GPUs than Morrical et al.'s method [7] (cf. Figure 5.5). Their method requires at least 40 RTX 5000 GPUs to render Huge Lander, whereas ours can render it with 16 GPUs.

We observe that simple image compositing is not an option for non-trivially partitioned data sets like we present. As the error metric in Figure 5.2 confirms, single

image compositing gives inaccurate results. Moreover, we found our compositing process to create low overheads even with the GPU counts going up to 88. Although the communication cost for compositing caused an increasing trend in terms of time, it never surpassed 22.68% of the total rendering time, indicating that our deep compositing method is highly scalable and generates correctly composited images interactively (see Figure 5.4).

Although the proposed ray-marcher is suitable for the use-cases described, any other ray-marcher that can adequately handle non-convex boundaries can be utilized. For instance, point-query sampling techniques leveraging adaptive sampling or space skipping [7, 15, 11] may produce much faster results. However, such methods rely heavily on hierarchical data structures to sample the volume, which would create more additional memory overhead. One could claim point-query sampling techniques negate this memory over-head by not storing connectivity; however, as pointed out before, many simulation environments have that data out of the box. Furthermore, point-query sampling techniques that rely on taking few samples—like delta-tracking based sampling schemes [64, 65]— may produce a noisy image that requires some time to converge, whereas our marching method generates deterministic noise-free images. For these reasons, we consider our marching algorithm to be more pragmatic in the context of data-parallel rendering and deep compositing. The results provided in Figures 5.5 and 5.6 support our claims. As our approach gains more speedup from the increasing number of GPUs, it also performs significantly better when taking an equal number of samples.

Finally, examining data distributions from Table 5.1, we see that directly utilizing native partitioning of the data causes uneven load balancing for some MPI sizes. Figure 4.2 reveals this phenomenon where the average number of fragments per rank distribution varies. The effects of this phenomenon can also be observed in Figure 4.3, where ranks 1, 4, and 14 have significantly fewer fragments than the others. Even though native partitioning causes uneven workloads, our timing experiments (cf. Figure 5.4) display decent scalability with the increasing number of GPUs we utilized. We smoothly achieve interactive rates with both of our data sets. For the small Mars Lander that has 72 clusters, we benchmark *14.35 fps* using 72 GPUs, and for the 552-cluster huge Mars Lander, we measure *10.27 fps* using 80 GPUs. We also observe an

ongoing downwards trend for the timings with the increasing number of GPUs, so it is worth mentioning that our application can achieve even higher frame rates given a more extensive hardware setup.

Furthermore, it is clear from Figure 5.4 that dominating term of rendering times is volume integration via ray-marching. Nevertheless, we observe a sharp increase in integration performance at $n = 24$ and $n = 32$ for Small and Huge Mars Lander, respectively. At the same time, it is expected for an embarrassingly parallel ray-casting algorithm to get faster with the increasing number of ranks; it is also likely for a compositing algorithm to slow down due to communication costs. We observe little to no increase in timings with our deep compositing, where it nearly behaves like a constant.

Chapter 6

Conclusions and Future Work

We introduce a GPU-based direct volume visualization framework that allows correct and interactive rendering even for non-convexly partitioned data. Our framework presents a mixed element ray-marching algorithm to integrate ray segments along the viewing direction. We achieved memory savings by exploiting XOR-based compaction schemes on our finite element data structures. Furthermore, we illustrate a deep compositing algorithm that allows proper order compositing of the RGBA-Z values obtained across multiple compute nodes.

Specifically, we demonstrate interactive frame rates for both of our datasets. With a small lander, we used the exact number of nodes as the simulation does while achieving $\approx$ 15 FPS, and with the Huge lander, we used even fewer nodes than the original simulation while still getting the peak performance of $\approx$ 10 FPS. Alongside correctness, our deep compositor displays little to no increase when increasing the node counts, thus proving it is optimized for modern multi-GPU environments on multiple nodes. Our framework scales well for increasing GPU counts while using native partitioning of non-convex data sets. We consider our framework suitable for both in situ and post hoc applications.

Possible areas for further research are as follows. While we allow visualizations of multiple scalar fields and timesteps, we do not use double/triple buffering techniques

that can hide the buffer loading times. Our implementation naively takes one scalar set on request (i.e., does not pre-fetch anything) [66]. To improve the time steps loading performance, buffering and pre-fetching the time steps in GPU and main memory can be employed [67]. Moreover, currently, we assume that the topology of the volumetric data does not change through time, yet this may not be the case.

Our sampling method does not support bilinear elements since determining vertex index order after element construction is difficult for them using our XOR-compaction. Also, we would utilize another compaction scheme over connectivity information as other works do [6, 17]. Image-based partitioning may increase our method's efficiency; however, it can get challenging with the in situ emphasis. Finally, integrating our approach into existing frameworks is another future work, field testing our claims.

Bibliography

[1] A. Sahistan, S. Demirci, I. Wald, S. Zellmann, N. Morrical, and U. Güdükbay, "GPU-based data-parallel rendering of large, unstructured, and non-convexly partitioned data." Submitted.

[2] A. Sahistan, S. Demirci, N. Morrical, S. Zellmann, A. Aman, I. Wald, and U. Güdükbay, "Ray-traced shell traversal of tetrahedral meshes for direct volume visualization," in *Proceedings of the IEEE Visualization Conference-Short Papers*, VIS '21, pp. 91–95, 2021.

[3] B. Nelson and R. M. Kirby, "Ray-tracing polymorphic multidomain spectral/hp elements for isosurface rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 1, pp. 114–125, 2006.

[4] H. T. Vo, S. P. Callahan, N. Smith, C. T. Silva, W. Martin, D. Owen, and D. Weinstein, "iRun: Interactive Rendering of large Unstructured grids," in *Eurographics Symposium on Parallel Graphics and Visualization* (J. M. Favre, L. P. Santos, and D. Reiners, eds.), The Eurographics Association, 2007.

[5] G. Marmitt, H. Friedrich, and P. Slusallek, "Efficient CPU-based volume ray tracing techniques," *Computer Graphics Forum*, vol. 27, no. 6, pp. 1687–1709, 2008.

[6] P. Muigg, M. Hadwiger, H. Doleisch, and E. Groller, "Interactive volume visualization of general polyhedral grids," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 2115–2124, 2011.

[7] N. Morrical, I. Wald, W. Usher, and V. Pascucci, "Accelerating unstructured mesh point location with RT cores," *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–14, In press.

[8] P. Shirley and A. Tuchman, "A polygonal approximation to direct scalar volume rendering," *ACM Computer Graphics (Proceedings of SIGGRAPH '90)*, vol. 24, no. 5, pp. 63–70, 1990.

[9] B. Rathke, I. Wald, K. Chiu, and C. Brownlee, "SIMD parallel ray tracing of homogeneous polyhedral grids," in *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, EGPGV '15, pp. 33–41, The Eurographics Association, 2015.

[10] I. Wald, W. Usher, N. Morrical, L. Lediaev, and V. Pascucci, "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*, HPG '19, The Eurographics Association, 2019.

[11] H. Wang, G. Xu, X. Pan, Z. Liu, R. Lan, and X. Luo, "A novel ray-casting algorithm using dynamic adaptive sampling," *Wireless Communications and Mobile Computing*, vol. 2020, Article no. 8822624, 12 pages, 2020.

[12] L. Szirmay-Kalos, B. Tóth, and M. Magdics, "Free path sampling in high resolution inhomogeneous participating media," *Computer Graphics Forum*, vol. 30, no. 1, pp. 85–97, 2011.

[13] J. Kruger and R. Westermann, "Acceleration techniques for GPU-based volume rendering," in *Proceedings of IEEE Visualization*, VIS '03, pp. 287–292, 2003.

[14] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister, "*SparseLeap*: Efficient empty space skipping for large-scale volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 974–983, 2018.

[15] N. Morrical, W. Usher, I. Wald, and V. Pascucci, "Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing," in *Proceedings of IEEE Visualization*, VIS '19, pp. 256–260, 2019.

[16] A. Aman, S. Demirci, U. Güdükbay, and I. Wald, "Multi-level tetrahedralization-based accelerator for ray-tracing animated scenes," *Computer Animation and Virtual Worlds*, vol. 32, no. 3-4, Article no. e2024, 11 pages, 2021.

[17] A. Aman, S. Demirci, and U. Güdükbay, "Compact tetrahedralization-based acceleration structures for ray tracing," *Journal of Visualization*, In press.

[18] N. Max, "Optical models for direct volume rendering," *IEEE Transactions on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99–108, 1995.

[19] J. Díaz, P.-P. Vázquez, I. Navazo, and F. Duguet, "Real-time ambient occlusion and halos with summed area tables," *Computers & Graphics*, vol. 34, no. 4, pp. 337–350, 2010.

[20] M. Ament, F. Sadlo, C. Dachsbacher, and D. Weiskopf, "Low-pass filtered volumetric shadows," *IEEE Transactions on Visualization and Computer Graphics*, vol. 20, no. 12, pp. 2437–2446, 2014.

[21] M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, and K. Engel, *Real-Time Volume Graphics*. Wellesley, MA: A. K. Peters, Ltd., 2006.

[22] C. Brownlee and D. Demarle, "Fast volumetric gradient shading approximations for scientific ray tracing," in *Ray Tracing Gems II* (A. Marrs, P. Shirley, and I. Wald, eds.), Apress Open, 2021.

[23] I. Wald, G. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navratil, "OSPRay - A CPU ray tracing framework for scientific visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 1, pp. 931–940, 2017.

[24] M. Larsen, J. S. Meredith, P. A. Navrátil, and H. Childs, "Ray tracing within a data parallel framework," in *Proceedings of IEEE Pacific Visualization Symposium*, PacificVis '15, pp. 279–286, 2015.

[25] L. Castanie, C. Mion, X. Cavin, and B. Levy, "Distributed shared memory for roaming large volumes," *IEEE Transactions on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 1299–1306, 2006.

[26] C. Brownlee, T. Ize, and C. D. Hansen, "Image-parallel ray tracing using OpenGL interception," in *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (F. Marton and K. Moreland, eds.), The Eurographics Association, 2013.

[27] T. Biedert, P. Messmer, T. Fogal, and C. Garth, "Hardware-accelerated multi-tile streaming for realtime remote visualization," in *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (H. Childs and F. Cucchietti, eds.), The Eurographics Association, 2018.

[28] T. Biedert, K. Werner, B. Hentschel, and C. Garth, "A task-based parallel rendering component for large-scale visualization applications," in *Eurographics Symposium on Parallel Graphics and Visualization* (A. Telea and J. Bennett, eds.), The Eurographics Association, 2017.

[29] Y. Cao, Z. Mo, Z. Ai, H. Wang, and Z. Zhang, "Parallel visualization of large-scale multifield scientific data," *Journal of Visualization*, vol. 22, p. 1107–1123, 2019.

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

[31] K. Moreland, "IceT Users' Guide and Reference," tech. rep., Sandia National Laboratories, 2011.

[32] A. V. P. Grosset, A. Knoll, and C. Hansen, "Dynamically scheduled region-based image compositing," in *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, vol. 2016 of *EGPGV '16*, 2016.

[33] W. Usher, I. Wald, J. Amstutz, J. Günther, C. Brownlee, and V. Pascucci, "Scalable ray tracing using the distributed framebuffer," *Computer Graphics Forum*, vol. 38, no. 3, pp. 455–466, 2019.

[34] G. Abram, P. Navrátil, P. Grossett, D. Rogers, and J. Ahrens, "Galaxy: Asynchronous ray tracing for large high-fidelity visualization," in *Proceedings of the*

IEEE 8th Symposium on Large Data Analysis and Visualization, LDAV '18, pp. 72–76, 2018.

[35] K.-L. Ma, "Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures," in *Proceedings of the IEEE Symposium on Parallel Rendering*, PRS '95, (New York, NY, USA), pp. 23–30, Association for Computing Machinery, 1995.

[36] K.-L. Ma and T. Crockett, "A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data," in *Proceedings IEEE Symposium on Parallel Rendering*, PRS '97, pp. 95–104, 1997.

[37] H. Childs, M. A. Duchaineau, and K. Ma, "A scalable, hybrid scheme for volume rendering massive data sets," in *Proceedings of the 6th Eurographics Symposium on Parallel Graphics and Visualization, EGPGV@EuroVis/EGVE 2006, Braga, Portugal, May 11-12, 2006* (A. Heirich, B. Raffin, and L. P. P. dos Santos, eds.), pp. 153–161, Eurographics Association, 2006.

[38] R. Binyahib, T. Peterka, M. Larsen, K.-L. Ma, and H. Childs, "A scalable hybrid scheme for ray-casting of unstructured volume data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 25, no. 7, pp. 2349–2361, 2019.

[39] H. Childs et al., "A terminology for in situ visualization and analysis systems," *The International Journal of High Performance Computing Applications*, vol. 34, no. 6, pp. 676–691, 2020.

[40] M. Larsen, E. Brugger, H. Childs, J. Eliot, K. Griffin, and C. Harrison, "Strawman: A batch in situ visualization and analysis infrastructure for multi-physics simulation codes," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization (ISAV), held in conjunction with SC '15*, (Austin, TX), pp. 30–35, 2015.

[41] M. Larsen, J. Ahrens, U. Ayachit, E. Brugger, H. Childs, B. Geveci, and C. Harrison, "The ALPINE in situ infrastructure: Ascending from the ashes of Strawman," in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ISAV'17, (New York, NY, USA), pp. 42–46, Association for Computing Machinery, 2017.

[42] U. Ayachit, A. Bauer, B. Geveci, P. O'Leary, K. Moreland, N. Fabian, and J. Mauldin, "Paraview catalyst: Enabling in situ data analysis and visualization," in *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV2015, (New York, NY, USA), pp. 25–29, Association for Computing Machinery, 2015.

[43] B. Whitlock, J. M. Favre, and J. S. Meredith, "Parallel in situ coupling of simulation with a fully featured visualization system," in *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (T. Kuhlen, R. Pajarola, and K. Zhou, eds.), PGV '17, The Eurographics Association, 2011.

[44] Y. Yamaoka, K. Hayashi, N. Sakamoto, and J. Nonaka, "In situ adaptive timestep control and visualization based on the spatio-temporal variations of the simulation results," in *Proceedings of the Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ISAV '19, (New York, NY, USA), pp. 12–16, Association for Computing Machinery, 2019.

[45] G. Aupy, B. Goglin, V. Honoré, and B. Raffin, "Modeling high-throughput applications for in situ analytics," *The International Journal of High Performance Computing Applications*, vol. 33, no. 6, pp. 1185–1200, 2019.

[46] D. E. DeMarle and A. C. Bauer, "*In Situ* visualization with temporal caching," *Computing in Science Engineering*, vol. 23, no. 3, pp. 25–33, 2021.

[47] N. Marsaglia, S. Li, and H. Childs, "Enabling explorative visualization with full temporal resolution via in situ calculation of temporal intervals," in *High Performance Computing* (R. Yokota, M. Weiland, J. Shalf, and S. Alam, eds.), (Cham), pp. 273–293, Springer International Publishing, 2018.

[48] W. K. Anderson, W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Achieving high sustained performance in an unstructured mesh CFD application," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, SC '99, (New York, NY, USA), pp. 69–es, Association for Computing Machinery, 1999.

[49] The National Aeronautics and Space Administration (NASA), "Fun3D Manual." Available at `https://fun3d.larc.nasa.gov/`, Accessed: 24 March 2022.

[50] P. J. Moran, "FUN3D Retropropulsion Data Portal-NASA," 2020. Available at https://data.nas.nasa.gov/fun3d/, Accessed: 20 July 2022.

[51] M. P. Garrity, "Raytracing irregular volume data," *ACM Computer Graphics (Proceedings of SIGGRAPH 90)*, vol. 24, no. 5, pp. 35–40, 1990.

[52] M. Weiler, M. Kraus, M. Merz, and T. Ertl, "Hardware-based ray casting for tetrahedral meshes," in *Proceedings of IEEE Visualization*, VIS '03, pp. 333–340, 2003.

[53] I. Wald, N. Morrical, and S. Zellmann, "A memory efficient encoding for ray tracing large unstructured data," *IEEE Transactions on Visualization and Computer Graphics*, vol. 28, no. 1, pp. 583–592, 2022.

[54] NVIDIA Corporation, "NVIDIA OptiX Ray Tracing Engine." Available at https://developer.nvidia.com/optix, Accessed: 19 July 2022.

[55] M. Pharr, W. Jakob, and G. Humphreys, "Primitives and intersection acceleration," in *Physically Based Rendering: From Theory To Implementation*, ch. 4, pp. 169–225, Burlington, MA: Morgan Kaufmann, 2021.

[56] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit*. Kitware, 4 ed., 2006.

[57] P. Sinha, "A memory-efficient doubly linked list," *Linux Journal*, 2004. Available at https://www.linuxjournal.com/article/6828, Accessed: 20 July 2022.

[58] M. Ikits, J. Kniss, A. Lefohn, and C. Hansen, "Volume rendering techniques," in *GPU Gems* (R. Fernando, ed.), pp. 667–692, Addison-Wesley, 2004. Available at https://developer.nvidia.com/sites/all/modules/custom/gpugems/books/GPUGems/gpugems_ch39.html, Accessed: 19 July 2022.

[59] T. Porter and T. Duff, "Compositing digital images," *ACM Computer Graphics (Proceedings of SIGGRAPH '84)*, vol. 18, no. 3, pp. 253–259, 1984.

[60] A. V. P. Grosset, M. Prasad, C. Christensen, A. Knoll, and C. Hansen, "TOD-Tree: Task-Overlapped Direct send Tree image compositing for hybrid MPI parallelism and GPUs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 23, no. 6, pp. 1677–1690, 2017.

[61] S. Eilemann and R. Pajarola, "Direct send compositing for parallel sort-last rendering," in *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization* (J. M. Favre, L. P. Santos, and D. Reiners, eds.), EGPGV '07, The Eurographics Association, 2007.

[62] R. Gershbein and P. Hanrahan, "A fast relighting engine for interactive cinematic lighting design," in *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, (USA), pp. 353–358, ACM Press/Addison-Wesley Publishing Co., 2000.

[63] M. Ishii, M. Fernando, K. Saurabh, B. Khara, B. Ganapathysubramanian, and H. Sundar, "Solving PDEs in space-time: 4D tree-based adaptivity, mesh-free and matrix-free approaches," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, Article no. 61, 22 pages, (New York, NY, USA), Association for Computing Machinery, 2019.

[64] L. Morgan and D. Kotlyar, "Weighted-delta-tracking for Monte Carlo particle transport," *Annals of Nuclear Energy*, vol. 85, pp. 1184–1188, 2015.

[65] E. R. Woodcock, T. Murphy, P. J. Hemmings, and T. C. Longworth, "Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry," in *Proceedings of the Conference on the Application of Computing Methods to Reactor Problems*, (Argonne, IL, USA), pp. 557–579, Argonne National Laboratories, 1965.

[66] S. Zellmann, I. Wald, A. Sahistan, M. Hellmann, and W. Usher, "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* (R. Bujack, J. Tierny, and F. Sadlo, eds.), EGPGV '22, 2022.

[67] M. Shih, Y. Zhang, K.-L. Ma, J. Sitaraman, and D. Mavriplis, “Out-of-core visualization of time-varying hybrid-grid volume data,” in *Proceedings of the IEEE 4th Symposium on Large Data Analysis and Visualization*, LDAV ’14, pp. 93–100, IEEE, 2014.