Visualization of Large Non-Trivially Partitioned Unstructured Data With Native Distribution on High-Performance Computing Systems

Alper Sahistan, Serkan Demirci, Ingo Wald, Stefan Zellmann, *Member, IEEE*, João Barbosa, Nate Morrical, and Uğur Güdükbay, *Senior Member, IEEE*

***Abstract*—Interactively visualizing large finite element simulation data on High-Performance Computing (HPC) systems poses several difficulties. Some of these relate to unstructured data, which, even on a single node, is much more expensive to render compared to structured volume data. Worse yet, in the data parallel rendering context, such data with highly non-convex spatial domain boundaries will cause rays along its silhouette to enter and leave a given rank's domains at different distances. This straddling, in turn, poses challenges for both ray marching, which usually assumes successive elements to share a face, and compositing, which usually assumes a single fragment per pixel per rank. We holistically address these issues using a combination of three interoperating techniques: first, we use a highly optimized GPU ray marching technique that, given an entry point, can march a ray to its exit point with high-performance by exploiting an exclusive-or (XOR) based compaction scheme. Second, we use hardware-accelerated ray tracing to efficiently find the proper entry points for these marching operations. Third, we use a "deep" compositing scheme to properly handle cases where different ranks' ray segments interleave in depth. We use GPU-to-GPU remote direct memory access (RDMA) to achieve interactive frame rates of 10–15 frames per second and higher for our motivating use case, the Fun3D NASA Mars Lander.**

***Index Terms*—Deep compositing, ray-marching, scientific visualization, sort-last compositing, unstructured volumetric mesh, volume rendering.**

I. Introduction

LARGE-SCALE simulation is essential for the computational sciences. Efficiently visualizing simulation outcomes is crucial for comprehending results and guiding the trajectory of ongoing simulations. Modern simulation systems often operate on High-Performance Computing (HPC) systems where the data is distributed across multiple nodes. In response to the scale and complexity of simulation data, researchers often rely on *in-situ* visualization techniques that integrate visualization capabilities directly into the simulation workflow, enabling data to be visualized as generated. Visualizing such distributed data while the simulation is running poses its own problems.

Simulation systems are rarely designed with rendering frameworks in mind. For example, GPU-based simulation systems like NASA's Fun3D [1], [2] may generate non-trivially split, non-convex data partitions known as *clusters*. While such partitioning may improve the simulation performance, it poses significant difficulties for rendering and compositing methods. This disparity between the simulation and rendering systems makes it difficult to visualize the large-scale data using the native distribution efficiently, especially for ray tracing-based direct volume renderers. The presence of non-convex partitions introduces complications where rays may exit and then re-enter a partition while traversing clusters loaded on other nodes in-between. Sort-last approaches [3] can solve this problem by deferring it to a compositing stage. However, determining a consistent depth order for these non-convex partitions proves challenging, as partitions from multiple processors often interleave each other. Changing the viewpoint can result in the same viewing rays traversing clusters with significantly different depth orders.

One approach to address these problems is redistributing the data. However, this solution is impractical for in-situ visualization of large-scale simulations due to the high costs associated with data transfer. As such, alternative strategies are needed to effectively visualize distributed simulation data without incurring excessive overhead. We aim to render unstructured, mixed-element meshes using a data-parallel approach directly on the system where they were generated without redistributing or altering their original node assignment. This approach caters to typical in-situ visualization use cases. It facilitates more efficient post hoc visualization by eliminating the need for costly data redistribution, which can take several hours for large datasets [4].

To address these challenges, we propose a large-scale, data-parallel, and GPU-optimized direct volume rendering system

Manuscript received 9 December 2023; revised 19 June 2024; accepted 9 July 2024. Date of publication 15 July 2024; date of current version 1 August 2025. This work was supported by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Grant 456842964. Recommended for acceptance by K. Moreland. *(Alper Sahistan and Serkan Demirci contributed equally to this work.) (Corresponding author: Uğur Güdükbay.)*

Alper Sahistan and Nate Morrical are with the SCI Institute, University of Utah, Salt Lake City, UT 84112 USA (e-mail: asahistan@sci.utah.edu; natemorrical@gmail.com).

Serkan Demirci and Uğur Güdükbay are with the Department of Computer Engineering, Bilkent University, 06800 Ankara, Türkiye (e-mail: serkan.demirci@bilkent.edu.tr; gudukbay@cs.bilkent.edu.tr).

Ingo Wald is with NVIDIA, Santa Clara, CA 95051 USA (e-mail: iwald@nvidia.com).

Stefan Zellmann is with the Department of Computer Science, University of Cologne, 50923 Cologne, Germany (e-mail: zellmann@uni-koeln.de).

João Barbosa is with INESC-TEC, 4200-465 Porto, Portugal (e-mail: jbarbosa@macc.fccn.pt).

Digital Object Identifier 10.1109/TVCG.2024.3427335

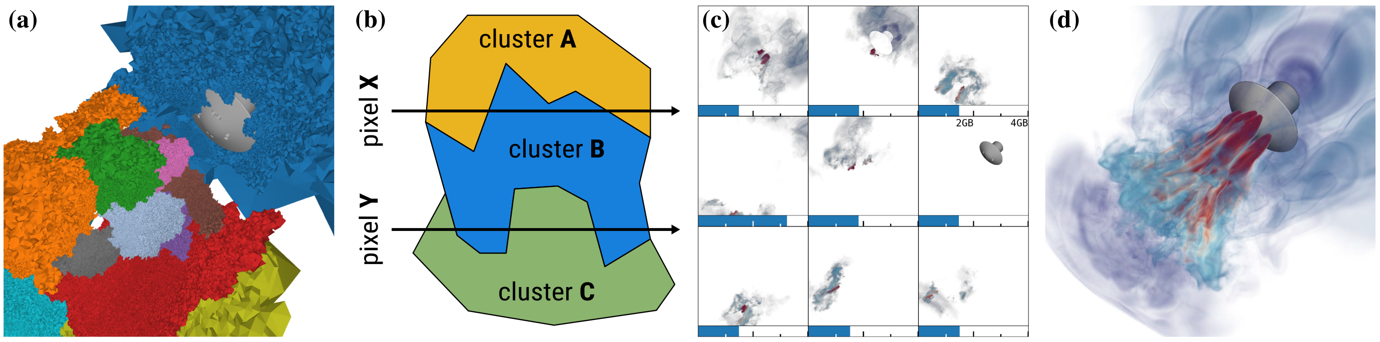


Fig. 1. Highlighting our framework's capabilities with the non-convex, non-trivially partitioned Small Mars Lander data set: (a) Rendering boundary surfaces of some partitions color-coded by the processor ID they were assigned; (b) Ray marching process through these distributed partitions; (c) Per-rank local data associated with these the partitions (bars indicate GPU memory consumption); (d) Final composited image; our data-parallel renderer can generate this image with 14 frames per second.

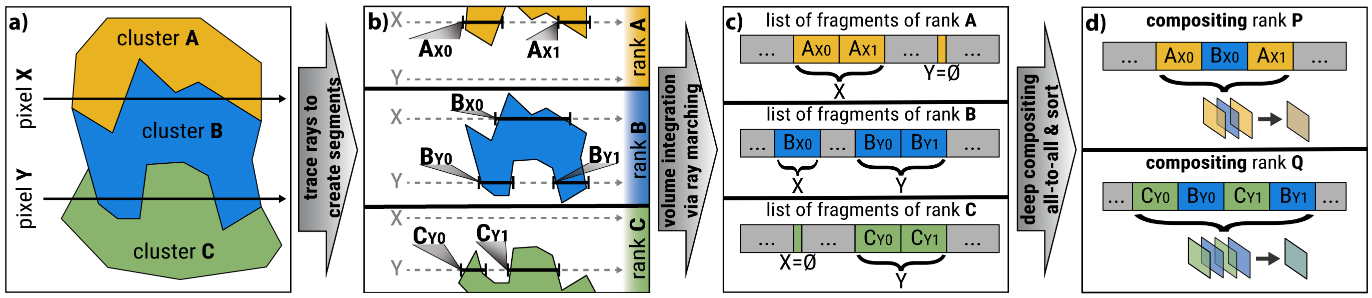

a)
cluster A
cluster B
cluster C
pixel X
pixel Y
trace rays to create segments
b)
rank A
rank B
rank C
volume integration via ray marching
c)
list of fragments of rank A
list of fragments of rank B
list of fragments of rank C
Y=Ø
X=Ø
deep compositing all-to-all & sort
d)
compositing rank P
compositing rank Q

Fig. 2. Overview of our framework over an example scene: (a) Rays X and Y are traced through three clusters A, B, and C that reside in three different ranks; (b) Each of the three ranks generates ray segments via shell-to-shell traversal by recording entry and exit distances of X and Y; (c) Colors of each ray-segment are calculated via ray marching and these colors are combined with their entry distance to be stored in fragment lists; (d) Deep compositing, where fragments for each ray are communicated and composited by their respective ranks, P and Q, to generate the final pixels.

based on volume ray marching. We draw inspiration from several prior works, including the compressed element marching and shell-to-shell traversal (where *shell* refers to the non-convex polygonal hull of an unstructured mesh) by Sahistan et al. [5]. Fig. 1 highlights our system's capabilities on a non-convex, non-trivially partitioned data set, and Fig. 2 gives an overview of our system design. Our contributions are

- an efficient, GPU-optimized ray-marching renderer supporting mixed-element unstructured meshes and uses ray tracing cores to perform shell visibility tests,
- a *deep compositing* algorithm implemented in CUDA that efficiently composite intermediate results from non-convex, interleaving meshes rendered using the sort-last algorithm and
- an efficient state-of-the-art data-parallel GPU rendering system using GPU-aware Message Passing Interface (MPI) to implement the communication across compute nodes.

We prove our system's effectiveness, efficiency, and scalability by conducting a thorough evaluation using the state-of-the-art size Fun3D data. We aim to target the state-of-the-art data given to us as raw data dumps. Yet, we emphasize that our system is intended for post hoc and in situ visualization and simulation steering.

II. Related Work

A. Unstructured Mesh Visualization

Notable approaches have been developed to render unstructured finite element meshes [6], [7], [8], [9]. Two predominant strategies to render unstructured volumes are *point-query sampling*, e.g., [10] and *element-marching* [9], [11].

In point-query sampling, free-flight distances of rays are calculated to extract information from an acceleration data structure, such as a *bounding volume hierarchy* (BVH), to sample volumetric data adaptively. These structures are often used to identify collisions between rays and particles for every pixel in a frame. Rathke et al. [12] propose a min/max BVH that speeds up the process of looking up elements for samples and iso-surfaces. Wald et al. [13] use point location queries on tetrahedral meshes with NVIDIA's ray-tracing (RT) cores, and this work was later extended by Morrical et al. [10] to also support pyramids, wedges, and hexahedra. These methods can be fast but may produce noisy results, so it is necessary to take multiple samples over time to obtain a converged image. To improve convergence and sampling, techniques such as empty space skipping [14], [15] or adaptive sampling [16], [17] can be employed. RT cores can also be used for empty space skipping and adaptive sampling [18]. Another work by Morrical et al. presents quick-clusters [19] as a memory-efficient data structure with low build times that allows for simple and efficient adaptive volume sampling. Nevertheless, this approach is still a Monte Carlo process where convergence is required over multiple frames.

Standard element-marching involves tracing rays and accumulating samples along the ray without using external acceleration structures [20], [21]. *Marching* is typically performed using visibility sorting or element connectivity. Shirley and Tuchmann [11] propose a method for rendering tetrahedral meshes without connectivity data. However, visibility sorting can be costly in interactive environments. Prior works have focused on storing connectivity information to avoid the need for sorting. Element-marching techniques that utilize connectivity data can be helpful for our purposes because many modern simulation systems already store this information, allowing us to avoid increasing memory usage. Aman et al. [22], [23] introduce a tetrahedra traversal algorithm that optimizes intersection tests using 2D projection while maintaining a connectivity list, but this approach is limited to pure tetrahedral meshes. Muigg et al. [9] propose a marching algorithm to handle non-tetrahedral elements and non-convex bounding geometry by storing compact face-based connectivity lists. Finding the element where the ray enters the volume is necessary during ray-marching. Sahistan et al. [5] demonstrate that this can be done with RTX (NVIDIA's RT core hardware) by using ray tracing with a BVH over the non-convex bounding geometry.

B. Data-Parallel Rendering

Increases in simulation output size have forced visualization to move to data-parallel methods. Data-parallel rendering is then usually realized using sort-last approaches [24], [25]. Sort-last parallel rendering requires an a priori assignment of clusters to compute nodes. While sort-first approaches also technically can be used for data-parallel rendering [3], nowadays, this is primarily used with multi-threading, GPU-parallel rendering, or replicated data [26], [27]. Hybrid approaches [28], [29], [30] aim to address load-balancing issues by leveraging both perspectives.

Sort-last algorithms allow data distribution at the cost of exchanging and compositing intermediate images. When the distributed data clusters have convex domains, image-based compositors, such as IceT [31], [32], are suitable for compositing as they typically merge single intermediate images per node. Also, sorting these intermediate images into the correct order instead of all their constituting fragments is sufficient with convex domains. However, these methods produce incorrect results when domains are not convex. Ma [33] uses a data-parallel unstructured volume rendering method that can handle non-convex data boundaries properly. Similar to our shells, their technique uses a *hierarchical data structure* to access the boundary faces and ray-casting operations from these faces. However, unlike our deep compositing, they prefer sending many smaller messages between compute nodes during rendering. Some of these ideas are later extended to utilize asynchronous load balancing via object and image-order techniques [34]. However, this work uses cell-projection rather than ray-casting, which may yield scalability issues. By implementing a spatiotemporally-aware compositor, Grosset et al. [35] reduces delays and communications. Their approach uses "chains" that determine the blending order of each image strip.

Using the *distributed framebuffer*, Usher et al. [36] accomplishes asynchronous tiled rendering over multiple nodes, reducing bottlenecks incurred by rendering and compositing. Childs et al. [37] show 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. At the sampling stage, they first sample small-sized elements. Then, they distribute the large elements to processors sampled to balance the load. Binyahib et al. [38] extend this work by proposing a many-core hybrid scheme employing sampling over a similar view-aligned grid. This scheme allows k successive samples in the same pixel and node to be partially composited, reducing the memory footprint. Our deep compositing algorithm builds upon the distributed framebuffer concept and is an extension of the algorithms by Childs and Binyahib et al. in that ours can also handle jagged cluster boundaries. In theory, 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, these works' image-order load balancing methods require large elements to be replicated or moved to other nodes, thus requiring additional memory, which may not always be available given an in situ scenario.

The standard approach for GPU rendering is to render all the pixels at once as we do or, at the very least, retire fragments in wavefronts to match the parallel execution models of GPUs. To this day, many frameworks still use CPU ray tracing also for sci-vis [39] though, where different rules and optimizations apply. Galaxy [30] is an example of a sci-vis ray tracer focusing on different data than ours, which employs an asynchronously operating framebuffer that is progressively updated in a frameless rendering fashion (single pixels are written to framebuffer locations by individual threads while other pixels still integrate light transport). In their framework, color fragments from BRDF evaluation can (through a special BRDF model) even be composited over already rendered primary-visible results. Following these footsteps, one could imagine retiring RGBA-z fragments progressively, yet our framework would require the use of a wavefront approach instead of the frameless rendering approach employed by Galaxy.

Modern ray-based hybrid approaches increase scalability by maximizing GPU utilization. Zellmann et al. [40] propose *island-parallelism*; they replicate data into N islands to increase utilization. Ray queue cycling by Wald et al. [41] uses very simple node assignments, including the potential for replication through islands, by cycling all rays on all GPUs without prior culling. This work is promising for lightweight clusters to compute node assignment and proves hybrid island parallelism's scalability.

Our method is tailored for modern GPUs, which minimizes the costs of compositing and sorting operations. We do not have to buffer every sample along the pixels since we ray-march through each segment to determine partial samples. Unlike previous works, our approach does not require re-distributing or replicating elements across nodes to render the data.

C. Compositing

We propose deep compositing as one approach to retiring RGBA-z fragments to GPU framebuffers because other data-parallel unstructured renderers do not support this operation holistically. Volumetric Depth Images (VDIs) [42] are an extension of Layered Depth Images (LDIs) [43] for view-dependent visualization of volume data. VDIs store *supersegments* that contain depth range, composited color, and opacity for the viewing rays of one viewpoint. They are subsequently used to render other viewpoints as proxies to the original volume data. Later extensions have focused on frame-to-frame coherence [44] and on parallel rendering [45], [46]. Noteworthy here is the work by Gupta et al. [45], who utilized VDIs for parallel compositing and, by that, presented an alternative to our deep compositing algorithm. Although VDIs are usually used for rendering structured volumes, the method can also render unstructured data as long as the renderer can produce semitransparent fragments with depth. Using VDIs for unstructured volumes has yet to be explored in the literature. Fundamentally, their algorithm replaces the fragment sorting operation we require with a (pixel space) ray casting operation.

D. In situ Visualization

File and network I/O have been bottlenecks in high-performance computing. In situ visualization aims to overcome this limitation by combining computation and visualization to allow users to access a running simulation. This approach has several benefits, including the potential for simulation steering by changing the parameters of the running simulation [47].

There are several in situ applications used in production [48], [49], such as Strawman [50] or Ascent [51]. In addition to these standard systems, various algorithms have been developed to handle time-varying data generated by simulations. For example, Yamoka et al. [52] propose a method that adapts the timestep sampling rate based on variations in the probability distribution function estimation of the connected simulation. Aupy et al. [53] present a model that allows for the analysis of simulations and the formulation of high-throughput scheduling. DeMarle and Bauer [54] propose a temporal cache scheme that stores time-varying information produced by a running simulation, which can later be saved according to a pre-defined trigger. Marsaglia et al. [55] introduce an error-bound in situ compression scheme that saves complete spatiotemporal simulation data. Our proposed method only requires a couple of lightweight structures in addition to what is already being kept in simulations. Moreover, based on the trends in these approaches, we foresee no significant issues that would prevent our method from being used in conjunction with current in situ systems.

III. Proposed Framework

We employ a data-parallel strategy to render clusters already distributed to different nodes by the simulation software. Borrowing terminology from triangle rasterization, we call each such tuple of color C, opacity α, and depth z, a *fragment* $F = (F_C,\ F_\alpha,\ F_z)$. Our approach generates fragments per *ray segment*. Ray segments are defined between an entry and exit position of a *shell* (faces defined by cluster boundaries), so for non-convex cluster boundaries, there is often more than one fragment since there might be more than one ray segment. Furthermore, these non-convex shells can be on different nodes and interleave each other, making correct order compositing extremely difficult. The shells serve as a means to traverse from cluster to cluster. To accomplish this, we rely on a hardware-accelerated BVH (shell-BVH) specifically constructed for the shells. We also need a data structure to traverse the unstructured elements, and to this end, we use element connectivity in local neighborhoods. To render this kind of data, our framework has to take the following steps (see Fig. 2):

- Only once, in a pre-process, nodes generate element-connectivity, shell-BVH, and XOR-compaction (see Section III-A).
- For every rank of every node, rendering starts by creating ray segments — i.e., intersecting entry and exit faces of every cluster along each ray direction. (see Section III-B).
- The colors of each ray segment are calculated using volume integration (see Section III-C).
- The fragments —that are the integrated colors of each ray-segment coupled with the entry depth— are used to *deep composite* a correct final image (see Section III-D).

A. Cluster Preparations

We describe the data preparations needed for the following rendering steps. These steps occur for every cluster in parallel.

1) Initial Per-Rank Preprocessing: We frequently use topological information about each rank's unstructured elements in the following sections. For example, the segment generation needs to know all the faces that form the outer "shell" of the cluster, and the marching needs connectivity information. While it seems likely that the simulation code already has that information, we wanted to avoid assuming that it has; thus, compute it upon startup.

2) Connectivity Information: Our method utilizes ray connectivity between unstructured mesh elements to jump from one element to another. Ideally, we can leverage pre-existing connectivity information output by the CFD simulation, but not all simulations supply that [56]. A compression technique tailored for element traversal, such as Aman et al.'s [23], can be implemented to minimize the additional memory usage.

We store the connectivity for each cluster locally—each rank computes connectivity for its set of loaded clusters. We store connectivity as a list of indices to neighboring elements. For instance, given the tetrahedron with ID i, indices of the neighbors of this tetrahedron can be found from the connectivity list at positions $[4i, \ldots, 4i+3]$. First, we extract individual faces from each element and store this face's vertex indices alongside its element ID in a list to compute the connectivity list. Then, we sort the vertex indices of faces for each face internally, followed by sorting these faces over the entire list. So, the first internal sort ensures shared faces are represented the same way, and the second list-wide sort ensures that the faces with the same vertex indices are stored next to each other. Now that the list is sorted, matching faces can be found easily by linearly iterating through the list and comparing pairs. If the face does not have a match, that face is a shell face, and we write -1 into the connectivity list. Otherwise, we write the neighboring elements indices to their respective position in the connectivity list.

3) Shell and Shell BVH: The element-marching process requires the starting element to be known. We build a BVH over the shells for storing the data necessary to reconstruct the boundary element behind the shell face. Additionally, we use shell-BVH to traverse the clusters.

Using the connectivity array, we can easily find all faces that make up the outer shell of each cluster by simply iterating over all unique faces and finding those for which there is no element on one side. We can create a guaranteed outward-facing face based on which sides were not covered. After finding the elements without neighbors, we construct an OptiX [57], [58] acceleration structure over those triangles, i.e., a shell-BVH. We store an int4 for each shell face, three integers for the vertices of the shell face, and the fourth integer stores the type and index of the element behind the face. The encoding of the fourth integer is in the style of PBRT's BVH-nodes [59] where the lower two bits of the fourth index signifies the element type (i.e., tetrahedron, pyramid, wedge, or hexahedron), and the remaining 30 bits are an index into the list of elements.

4) XOR-Compaction: We significantly reduce the storage cost for vertex indices by utilizing pre-computed exclusive-or (XOR) bitmasks. Our scheme exploits two facts; the inverse operation of XOR is itself, $(a \oplus b) \oplus b = a$, for two integers a and b represented in binary, and every internal element shares three or four vertices with another element. The individual memory layouts and their geometric illustrations are shown in Fig. 3.

In essence, the scheme relies on the fact that the element marcher will be coming from an element that shares a face with the element it is entering. However, one caveat with this idea is that the marching process needs to start from an initial shell face and its corresponding element. Conveniently, the shell-BVH stores this information for the boundary elements. During element-marching, we do not have to fetch the vertices we already have in our memory. Moreover, we do not have to keep some vertex indices in the memory. The missing vertex indices can be inferred by using pre-computed XOR fields. To maintain brevity, this section will focus on calculating the XOR masks. The reconstruction of the elements from their compacted forms will be covered in Section III-C. We maintain VTK [60] vertex ordering for our elements as they must be consistently ordered during sampling. Although we found more compact schemes for some elements, we did not use them as they cannot be reconstructed consistently.

```
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];
};
```

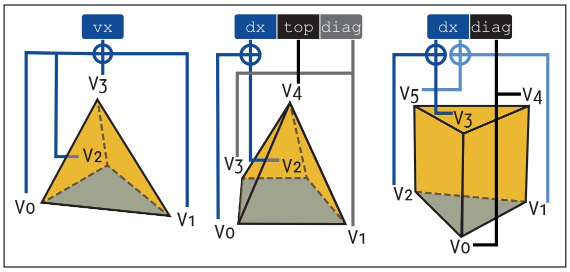


Fig. 3. XOR-compacted memory layouts (top) and geometric illustrations of XOR calculations for Tetrahedron (Tet), Pyramid (Pyr), and Wedge (Wed) structures (bottom). Hexahedron (Hex) has no compaction scheme. `uint` stands for an unsigned integer. The "$\oplus$" symbol indicates the XOR operation. The total sizes of each struct are 4, 16, 16, and 32 bytes for Tet, Pyr, Wed, and Hex, respectively. The vertices are in VTK [60] ordering.

For tetrahedra, we use the idea from Aman et al. [22], [23], where they exploit the fact that a tetrahedron shares three of its four vertices. Therefore, instead of keeping four integers per tetrahedron, we can store a single integer XOR mask, v_x (vx field in the Tet structure in Fig. 3), derived as $v_x = v_0 \oplus v_1 \oplus v_2 \oplus v_3$. This way, we reduce the 16-byte naïve tetrahedron storage to four bytes.

For the 16-byte pyramid, we store a single dx field that is the XOR of the 0th and 2th vertex indices, two vertex indices that correspond to the other diagonal of the quadrilateral (1th and 3th vertices), and a top vertex index, which is always the 4th vertex. Using this scheme reduces four bytes from naïve pyramid storage.

Our 16-byte wedge structure includes two dx and two diag fields. The dx fields contain two XORs: the first one is the XOR of the 2th and 3th vertex indices, and the second one is the XOR of the 1th and 5th vertex indices. The diag fields explicitly store the 0th and 4th vertex indices. The compaction allows us to cut 8 bytes per wedge compared to naïve storage.

Hexahedron has the lowest shared vertex ratio among all element types, requiring four vertices to be obtained upon entry. Finding an XOR-based hexahedra compaction scheme aligning closely with a 16-byte size is difficult. Hence, we store all hexahedra indices without compaction, following the VTK mesh ordering.

B. Ray-Segment Generation Via Shell Traversal

Rendering begins with a shell-to-shell traversal step akin to [5], where ray segments are generated for each cluster loaded in each rank. Ray segments span between entry and exit faces along the ray's path as it enters and exits clusters. As the clusters can be non-convex, rays may enter the same cluster multiple times, which is an aspect not encountered in convex clusters, posing challenges for IceT-style compositing. We allow multiple ray segments to be generated per ray and shift the problem to the compositing stage (see Section III-D). Moreover, we can use hardware-accelerated ray tracing on GPU nodes with RT cores. The stages of this scheme are as follows:

1) Trace the ray through the shell-BVH with front-face culling from the ray origin.
2) If a ray hits a shell face, we mark that face as the exit face and create a backward ray with the origin at the hit position.
3) This backward ray is again traced using front-face culling to find an entry face.
4) The found entry face contains four index values (Section III-A), and the last one encodes the ID and the type of the element from where we start our ray-marcher (Section III-C).

In real-life data sets, volume boundaries may not perfectly align, potentially intersecting or being slightly separated. The robust handling of such cases involves casting two front-face culled rays to determine exit and entry points, thereby mitigating sampling and compositing errors.

C. Per-Segment Volume Integration

The volume integration process determines the color and transparency of a fragment by ray-marching. We march rays between neighboring elements to sample equidistant points from the volume. Samples are calculated by linear interpolation of the vertex scalars of elements that the ray passes through. Suppose that at least one barycentric coordinate of a sample point is outside the range [0, 1], i.e., the sample point is outside the checked primitive. In that case, we march to the next element by fetching the element ID from the connectivity buffer and reconstructing it from the XOR-compacted form. The marching terminates when the particle becomes completely opaque, or the sample point leaves the cluster mesh.

We preserve the current element's information in the march-state structure to reduce memory access. The structure also helps us reconstruct the next element from a neighbor. The state keeps and updates the last intersected face type (triangle or quad), the current element's type, index, and vertex indices. We place the entry face indices into the same positions in the march state to ignore the entry face during the following exit face selection (since we now know which vertices belong to the entry face). We test the remaining faces of an element using a "projected tetrahedra" like approach [11], where upon entry, the vertices of an element are projected to a 2-D ray-centric coordinate system space whose origin coincides with the ray origin, and the z-axis is the ray's direction vector [23]. This way, we reduce the floating point instruction count for orientation (point-line classification) tests as they are performed in 2D space. We illustrate this process for a tetrahedron in Fig. 4. Each volume element is unique in its geometry and face arrangement, and it is hard to make a *simple* algorithm that handles all combinations. Hence, our element marching handles various elements in a case-by-case fashion.

Tetrahedra are handled similarly to Sahistan et al. [5], but we allow arbitrary points inside the elements to be sampled. To reconstruct a tetrahedron from the compacted format, we XOR the indices from the entry face with the vx field as shown on Fig. 5. We generate vx by XOR'ing all vertices with known

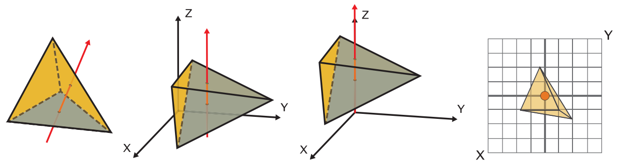


Fig. 4. Illustration of the steps for projection to the ray-centric coordinate system and face intersection. The ray's origin is placed to (0, 0, 0), and its direction is oriented with the Z-axis. The same transformations are applied to the tetrahedron's vertices using dot products. Finally, 2D orientation tests are used to determine the exit face.

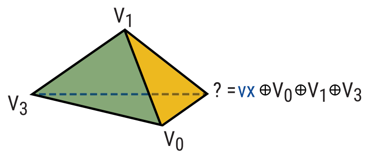


Fig. 5. Example reconstruction case for a tetrahedron XOR'ing known vertices from entry face (V_0, V_1 and V_2) and the vx field in the Tet structure in Fig. 3.

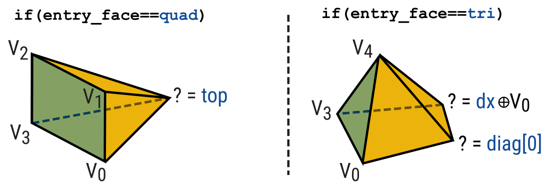


Fig. 6. Two example pyramid reconstruction cases are illustrated. On the left, with the quad face as the entry face, only the missing vertex is explicitly stored at the top field. On the right, where the entry face is a triangle, vertices V_0, V_3, and V_4 are matched to indices stored at top and diag fields. The index diagonal to V_3 (matched to diag[1]) is represented by diag[0], and the last unknown vertex is obtained by XOR'ing V_0 and dx.

three vertices, which will reveal the missing vertex index. When leaving a tetrahedron, we use the exact process depicted on the right-most image of Fig. 4.

Pyramids can be reconstructed by looking at the entry face type. If the face type is a quad, then only the missing vertex —top vertex— is already explicitly stored. If the entry is through one of the triangle faces, we need to look at diag and top fields to match them with two of the entry face indices. The first missing index is the unmatched index from diag. The remaining unmatched index from the entry face can be XOR'ed with dx to find the last missing index (see Fig. 6). When leaving the pyramid, exit faces can be determined similarly to tetrahedra. Four triangles in 2D space are tested against point (0, 0) if the entry face is a quad. However, if the entry face is a triangle, we check against the quad face and then three triangles.

Wedges reconstruction from the compacted form starts by identifying the entry face type. If the entry is from a triangular face, we need to obtain three indices, and the entry face must be composed of one of the diag[2] field indices. With this information, we can obtain one of the missing indices (from diag[2]) and identify if we entered from v_0, v_1, v_2 or v_3, v_4, v_5 face. The remaining two indices are encoded in the different dx[2] fields. By matching one of the diagonal vertex indices to one of the

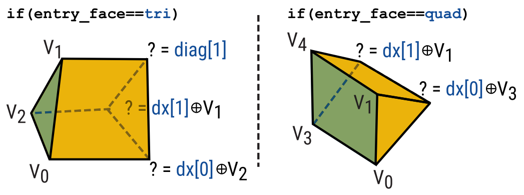


Fig. 7. Two wedge reconstruction cases are depicted. On the left, the entry is from a triangle face, and V_0 matches diag[0], so we retrieve three missing indices as follows: diag[1], dx[1] XOR V_1, and dx[0] XOR V_2. On the right, the entry is from a quad face (V_0, V_1, V_4, and V_3). Entry indices V_0 and V_4 match with the ones at diag; so, missing indices are found by XOR'ing dx[1] with V_1 and dx[0] with V_0.

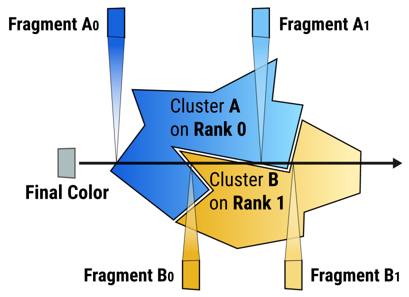


Fig. 8. Compositing on concave clusters (A and B) where a ray generates four fragments distributed across two nodes in interleaved order. Fragments A.0 and A.1 are generated in node 0, and fragments B.0 and B.1 are generated in node 1.

diagonal fields, we can construct the two missing indices from the dx[2] fields. If the ray enters from a quadrilateral face, it must contain one or both indices stored in diag[2]. By matching diagonal indices, we can determine the entry quadrilateral. Then, we have two subcases. In the first subcase, two of the entry quadrilateral's indices match both indices in diag[2]. We can obtain the two missing indices by XOR'ing the dx[2] fields with the unmatched indices of the entry quadrilateral. In the second subcase (where either one of the diagonals matches with one of the quadrilateral face indices), we can immediately get one of the missing vertices from the unmatched diag[2] field. Finally, we can use one of the dx[2] fields to get the other missing index. See Fig. 7 for example cases. Finding the exit face is similar to pyramids, where we first test the quad faces and then triangle faces, ignoring the entry face.

Hexahedra are not compacted, so we fetch all indices upon entry. Hexahedra are uniform like tetrahedra; however, they have more faces. Therefore, finding the exit intersection requires the highest number of orientation tests. In the worst case, hexahedra require 13 2D orientation tests, whereas wedges, pyramids, and tetrahedra require 7, 5, and 2 orientation tests, respectively.

D. Deep Compositing

After the steps in Sections III-B and III-C, each rank owns several fragments that can belong to any pixel. This section discusses how these fragments are composited to form a *correct* image and why the standard approaches like IceT do not work.

Given all of a given pixel P's fragments $F_0^{(P)}, F_1^{(P)}, \ldots, F_{N^{(P)}}^{(P)}$, the correct final pixel color is the result of first sorting these fragments by their depth and compositing them using the *over* ($\widehat{O}(A,B)$) and *under* ($\widehat{U}(A,B)$) operators, as described in [61], [62]. The challenge is that any given pixel's fragments may get produced on many different ranks, requiring some merging of different ranks' results. 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 (see Fig. 8). Our data sets' fragment generation and distribution details are presented in Section IV.

The simplest approach to compositing this would be to first composite all ranks' fragments to a single fragment per pixel per rank and then use some optimized compositing library like IceT [32] to produce the final image. However, as neither $\widehat{O}$ nor $\widehat{U}$ are commutative, this will give wrong results every time a ray enters the same shell more than once. Fragments must be composited in visibility order, considering that the fragments along a ray are distributed unevenly across the ranks.

1) Compositing With More Than One Fragment/Pixel: We developed a new framework to solve the compositing problem, explicitly allowing each rank to have multiple fragments per pixel. At an abstract level, our method expects each pixel to store one counter that specifies the number of fragments, N, plus an address (or offset) to a list of fragments, $F_0, \ldots, F_{N-1}$. Similar to parallel-direct-send [63], [64], we then split the frame buffer into R distinct *regions* of pixels (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.

The bandwidth required for compositing is often a bottleneck in data-parallel parallel rendering, even with only a single fragment per pixel. To minimize bandwidth, users can use lower-precision encoding with 8-bit fixed-point for RGBA and *float* solely for the depth value instead of full float precision (five floats for r, g, b, opacity, and depth). We also automatically discard fragments with zero opacity value, as these will not contribute to the image.

Aside from sending the fragments, sending the per-pixel counters (see step 2) has high bandwidth requirements. To reduce that, we use specialized encodings with 2, 4, 8, or 32 bits for those counters, depending on the longest fragment list length. We use dedicated CUDA kernels for encoding and decoding the 32-bit counter arrays into this lower-precision representation before and after the counter exchange using MPI_Alltoallv, a collective communication provided by MPI in which all processes send/receive data to/from all other processes. All MPI calls involved in compositing use GPU-to-GPU RDMA, an extension available when using GPU-aware MPI distributions where MPI functions recognize when GPU pointers are passed. In that case, MPI communication across nodes goes directly from the PCIe network card to the GPU, not through the main memory first. Compositing works in the following steps (see Fig. 9):

1) Generating a contiguous send buffer: Given each pixel's fragment lists, each rank computes a GPU-parallel prefix sum over all its fragment counts, yielding the total number of fragments on this rank. We then allocate a single contiguous memory

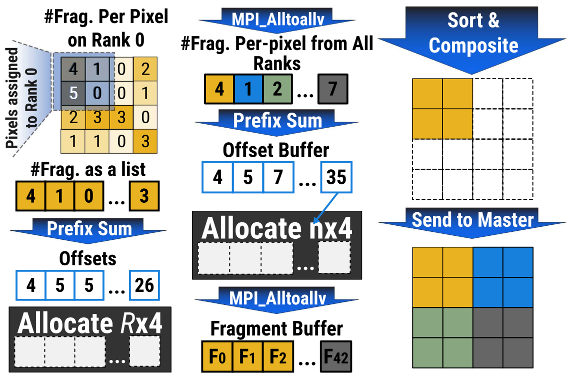


Fig. 9. Illustration of deep compositing for $R = 4$: Rank 0 handles pixels highlighted in blue in the top-left. We create a fragment list and calculate the per-pixel offset list using prefix sum. A contiguous memory region, sized $R \times 4$ (number of pixels assigned to rank 0), is allocated for receiving fragment counts from other ranks. Each rank communicates the number of fragments it will send to other ranks using different intervals of the previously calculated offsets; these received fragment counts per pixel are once again prefix summed to form a universal offset buffer. Then, a memory of size $n \times 4$ is allocated for receiving the actual fragments, where n is the number of fragments per pixel. Subsequently, fragments from other ranks are transferred, depth-sorted, composited, and sent to the master rank for the final image.

region for these fragments and compact the individual fragments into this buffer (using the prefix sum result as offsets). By design, this buffer will contain all fragments going to all other ranks in order.

2) Exchanging per-pixel fragment count ranges: Given the assigned range of pixels, each rank computes which range of per-pixel counters it needs to send to any other rank. To this end, each rank allocates a per-rank counter buffer with a size R times the number of pixels in its region. Next, each rank computes the offsets to store the counters from other ranks. We then perform a collective MPI_Alltoallv on these buffers, after which each rank has the fragment counts from every other rank for its assigned region of pixels.

3) Exchanging Fragment Lists: Having received all other ranks' per-pixel fragment counts for its range of pixels, each rank then performs a GPU prefix sum over those counters, the result of which can once again 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. We then allocate a receiving buffer of the required size, look up where each other rank's fragments will go in this buffer, and issue a second MPI_Alltoallv that, in this case, collectively moves all fragments into the receive buffer of the rank assigned to that fragment's corresponding pixels.

4) Local Compositing: The result of the previous steps is that each rank has two buffers containing all fragment lists for its assigned pixels. The first buffer —*fragment buffer*— stores all fragments for that rank's pixels received from all other ranks, ordered by ranks and pixels within each rank. Given 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 second buffer —*offset buffer*— stores the results of prefix sum operations. It, by design, provides the offsets where the fragment lists start. For example, if P is the number of pixels for which this rank is responsible, then the fragments from rank r for pixel j start at offset offsets[r*P+j]. Using this, we can launch a CUDA kernel that, for each pixel p, looks up the R different lists of fragments and composites them in the visibility order. For this k-way merge task, we used a naïve direct k-way merge algorithm with $\Theta(RF)$ runtime, merging the F fragments by iteratively scanning R already sorted lists to find the nearest fragments. Although k-way merge algorithms with better asymptotic computational costs exist, the naïve algorithm serves the purpose because F and R are small in our case.

5) Sending final results to master: The output of the previous CUDA kernel is, on each rank, a fully composited RGBA value for each pixel in that rank's range of pixels. We send these to the master using an MPI_Send; the master sets up R matching MPI_Irecv calls, each using the appropriate part of the final frame buffer as the receive buffer. Once these are completed, the master has the final assembled frame buffer, and compositing is complete.

This method is a natural extension of the parallel direct-send technique described by Grosset et al. [63] and Favre et al. [64], with the main difference is 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* [65].

2) Fragment List Management: Though the compositing itself is easy to use from the host side, properly setting up the device-side inputs (fragment lists and counters) would require the renderer to handle what are akin to device-side dynamic memory allocations to manage those per-pixel variable-size fragment lists during rendering.

To relieve the renderer of this low-level fragment list management, we also developed what we call a *device interface* for this library, through which a renderer can *write* new fragments into a pixel, with that interface, then handling the proper storage of those fragments—which significantly simplifies the rendering code.

Two-Pass, Flexible-length Fragment Lists: The main challenge for developing this interface was that we could not simply allocate more device memory during rendering, so we needed *some* limit on how many fragments a renderer could generate in any frame. We first developed a two-stage interface in which the renderer would be run twice: in the first stage, the interface would only count the fragments produced per pixel but not store any. After this stage, it would compute a prefix sum over those counters to allocate a big enough buffer, with the prefix sum values serving as offsets into this buffer. A second pass would render the same but store the fragments at the provided offsets.

Single-Pass, Fixed-Length Fragment Lists: The two-pass method allows for arbitrary-sized fragment lists (up to device memory, obviously) but requires running the shell traversal at least twice, which may or may not be acceptable. We, therefore, also developed a second, single-pass device interface in which the renderer—upon initialization—specifies a maximum allowed number of fragments per pixel, per rank (F_{max}), which

TABLE I
THE FUN3D DATA SETS STATISTICS

	Element counts					
Model	Vertices	Tetrahedra	Pyramids	Wedges	Clusters	Size
Airplane	253 M	50 M	32 M	415 M	400	14
Small Lander	145 M	766 M	47.5 K	32 M	72	14
Huge Lander	1.2 G	6.12 G	285 K	256 M	552	112

The size is in GBs.

can then be used to pre-allocate lists to add fragments. 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.

IV. EVALUATION OF THE FRAMEWORK

We conducted our experiments on Frontera RTX nodes of Texas Advanced Computing Center (TACC) [66], where each of the 22 nodes had four NVIDIA Quadro RTX 5000 GPUs with 16 GB of VRAM. We utilize all four GPUs available per node for every data point of our experiments. In the Frontera system, we use the gcc compiler, version 12.2, and Intel MPI Library 2021.9, with CUDA 12.2 and OptiX 7.1 over the CentOS 7.9.2009 operating system.

We use three large-scale datasets simulated by NASA using the Fun3D [1] solver (see Table I). Those comprise two versions of the NASA Mars Lander CFD simulation [67] and another CFD mixed-element simulation of an airplane. Statistics for the data sets can be found in Table I. We utilized a maximum of 72 GPUs (ranks) for the Small Mars Lander dataset, corresponding to the number of clusters in that dataset. Airplane and Huge Mars Lander data sets have more clusters than the maximum number of GPUs at Frontera; therefore, we halt our experiments before reaching the one-cluster-per-GPU point.

Nevertheless, we observe that we reached the saturation points for benchmarks to conclude the experiments meaningfully. In most of our experiments, the number of clusters exceeds the number of available ranks. We distribute the clusters among ranks using a round-robin fashion. While this approach may not be optimal because it leads to uneven workloads across ranks, we chose it because it aligns with our in-situ argument, where we do not have any control over how the parts are distributed. Our experiments use the single-pass compositing strategy with a maximum of eight fragments per pixel, providing a good balance between compositing performance, memory, and accuracy.

While visualization solutions like VisIt [68] and Paraview [69] offer a broad range of rendering methods and pipelines, our investigation reveals that none allow meaningful comparisons. This limitation arises from issues such as producing incorrect images, requiring tetrahedralization of the geometry, or failing to achieve interactive rates even with partial data. We compare the correctness of our deep compositor to the widely used IceT compositor [31], [32]. However, irrespective of correctness, many of these visualization solutions struggle to handle the intricacies of these data sets.

In exploring various solutions, Ascent [51] emerges as the closest contender for rendering the Small Mars Lander data. However, it encounters limitations when dealing with the native data format, lacking support for mixed unstructured mesh geometry and requiring tetrahedralization for the entire dataset. The tetrahedralization process significantly increases memory consumption. Even with tetrahedralized data, configuring Ascent for accurate rendering proves challenging. Despite our earnest attempts, Ascent generates an incorrect visual output. Furthermore, the rendering time for an incorrect frame using Ascent is approximately three times slower than our approach, with both executing on 72 ranks.

We evaluate our framework concerning its memory overhead (Section IV-A), correctness (Section IV-B), scalability (Section IV-C), and discuss the limitations (Section IV-D).

A. Memory Overhead

We examine data distribution and memory footprints. Fig. 10 depicts the average per-rank memory footprints of our larger data structures that may not be present in a simulation environment. These include compacted unstructured elements, shell BVH, and connectivity data.

Fortunately, most memory usage is attributed to connectivity data, typically available in modern CFD solvers [1], [2]. Our XOR-compaction scheme achieves higher compression rates with data sets containing more tetrahedra, such as Small and Huge Mars Landers. This aligns with expectations, given that tetrahedra are the most compressed elements in our scheme, with a $4 : 1$ ratio. Some memory savings are also observed with the airplane data set, albeit to a lesser extent, as the dominant element type is wedges, with a $6 : 4$ compression ratio.

B. Correctness

We compare multiple configurations of our deep compositor and IceT compositor against a ground truth to verify the correctness of our compositing scheme. We use the same volume integrator for both compositors to minimize the difference in generated fragments. As IceT is not designed to handle more than one fragment per pixel, we first use front-to-back compositing to reduce each rank's per-pixel fragment count to one. Then, we use IceT to composite the fragments between the ranks. Fig. 11 demonstrates these experiments' visual and numeric results.

Our deep compositor provides correct visualization with minimal compromise in performance and memory efficiency, while the industry-standard IceT cannot be used to determine the correct compositing order. We substantiate our method's correctness through PSNR measurements, difference images, and showcasing a cross-section where compositing errors are evident. With a small loss in performance, we achieve correct composited visuals using 32 fragments per pixel. Notably, even

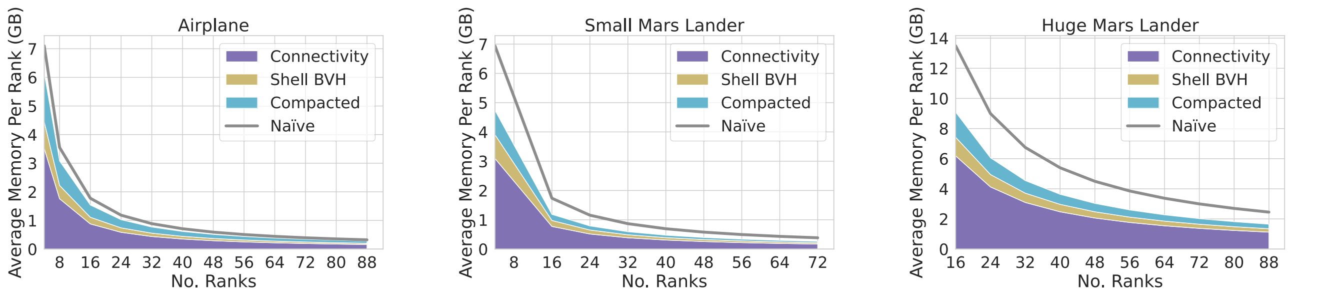
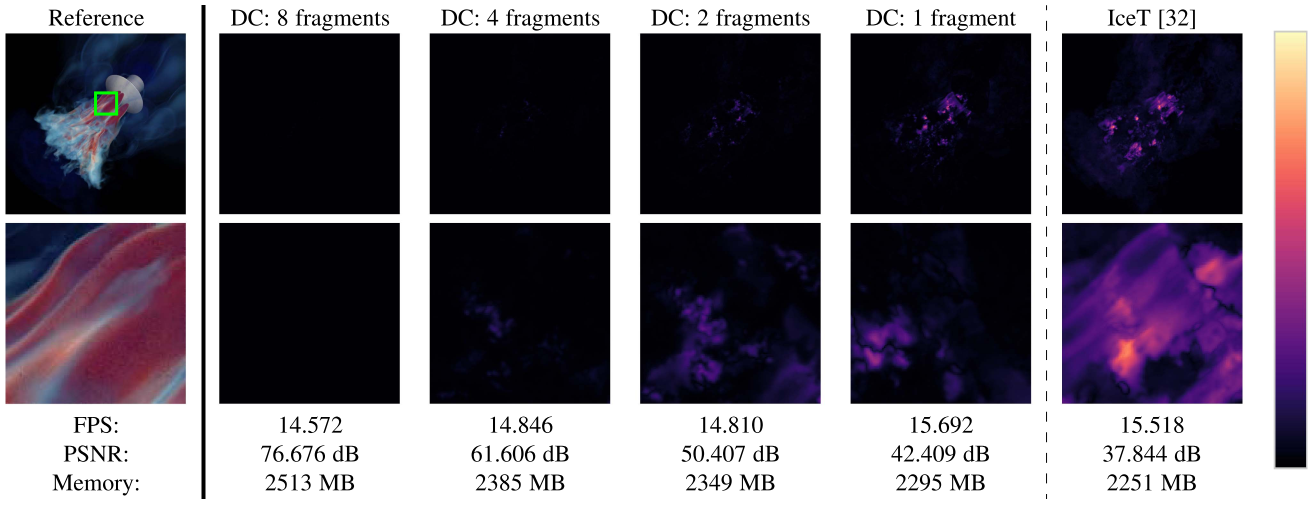


Fig. 10. Per rank average memory consumption calculations of various buffers for increasing MPI sizes: Airplane, Small Mars Lander, Huge Mars Lander. The average memory usage of XOR-compacted elements is stacked over the average shell-BVH size, which again is stacked over the average connectivity buffer 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).



	DC: 8 fragments	DC: 4 fragments	DC: 2 fragments	DC: 1 fragment	IceT [32]
FPS:	14.572	14.846	14.810	15.692	15.518
PSNR:	76.676 dB	61.606 dB	50.407 dB	42.409 dB	37.844 dB
Memory:	2513 MB	2385 MB	2349 MB	2295 MB	2251 MB

Fig. 11. Visual evaluation of correctness for our deep compositor (DC) with 8, 4, 2, and 1 fragment configurations and IceT compositor: Top row leftmost image depicts reference rendering for Small Mars Lander using deep compositor with 32 maximum fragments. This reference image represents the correct composited images, as this rendering requires a maximum of 21 fragments per pixel per rank. We present error visualizations (brighter means higher error) computed with ꟻLIP [70], comparing reference to deep compositor configurations and IceT. The bottom row emphasizes a specific region (in green) of the ꟻLIP images. Finally, we display each column's total rendering time (fps), peak signal-to-noise ratio (PSNR), and memory usage.

with two fragments per pixel, our compositor significantly reduces errors and attains a more accurate compositing order than IceT.

C. Scalability

We conducted several experiments to see how our framework scales with increasing MPI ranks and workloads. Fig. 12 depicts scaling experiments conducted over the three data sets where we measured the average total rendering time for the given frames. The plot reports the total time as stacked timings of two main sub-processes: volume integration and compositing.

With ray-marching, finding suitable stepping sizes (or sampling rates) is essential where performance-to-quality compromise is reasonable. We document our scalability sensitivity to ray stepping size in Fig. 13 where sensible stepping sizes are selected and run through the same data points in Fig. 12. These step sizes are defined in the world space. We select step sizes that preserve dataset features while offering optimal performance. Given that the elements in the Airplane dataset are larger than in the other datasets, we opt for a higher step size.

Looking at Figs. 12 and 13, a significant part of the total rendering time is spent on volume integration. We observe a decent scalability trend on Small and Huge Mars Lander data sets as total rendering time improves $\approx$ 3-2.6x from their worst case to optimal GPU counts. These improvements are mostly related to the data-parallel volume integration process's scalability. We observe a lesser scalability benefit for the Airplane data set (around $1.27\times$). We also discovered that rendering performance in the same data set is more sensitive to changes in the stepping size. We believe this is due to the following reasons:

- It is more efficient to traverse tetrahedral elements as their significantly smaller sizes affect the caching performance less. Despite being around the same size, the Airplane has $\approx$ 15 times fewer tetrahedra and $\approx$ 12 times more wedges than the Small Mars Lander (see Table I).
- Our volume integrator is tailored to amortize the initialization costs over longer traversals. Although being around

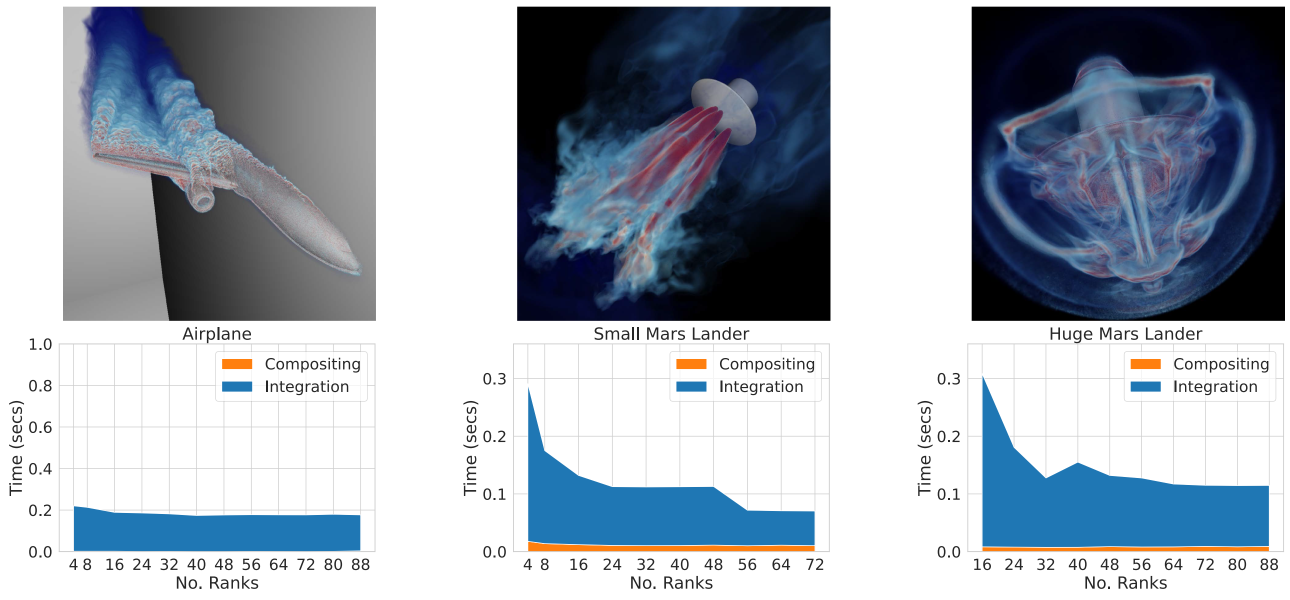


Fig. 12. Strong scaling benchmarks for Airplane (step size $= 2.0$), Small Mars Lander (step size $= 0.1$), and Huge Mars Lander (step size $= 0.2$) models: Volume integration process timings are stacked over compositing process timings for increasing numbers of ranks, forming total timing.

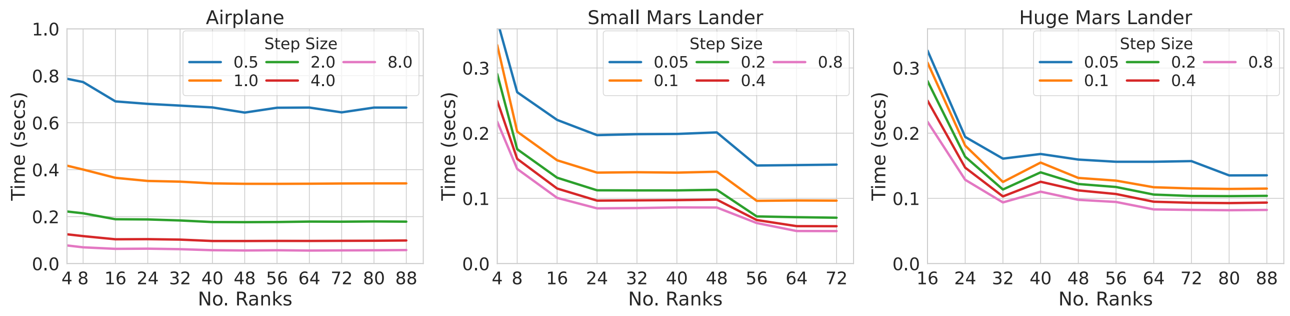


Fig. 13. Scalability (in seconds) benchmarks for various step size dt (in world space units).

the same size, the Airplane data set has significantly more clusters (≈ 5.56 times) than the Small Mars Lander (see Table I). With the shallower clusters, we are hitting the scaling benefit of our method much earlier in MPI ranks.

Fig. 12 also includes scalability plots for deep compositing, where we observe that performance is largely unaffected by adding more MPI processes. Looking at the Airplane data set results, which exhibit ample empty regions, we conclude that the cost corresponds to the number of fragments exchanged, which only indirectly depends on the processor count.

To understand the compositing costs better, we also conducted experiments evaluating the deep compositor in isolation and compared it to IceT. The results of those experiments can be found in Fig. 14. We note that in GPU-aware MPI calls and RDMA presence, it is difficult to profile whether the compositor is dominated by GPU compute or communication cost. However, we assume that on-device memory accesses are typically orders of magnitude faster than device-*to*-device accesses. Although IceT usually performs slightly better than our deep compositor regarding computational cost, our execution times are comparable to IceT's for all of our experiments. It should be noted that the amount of IceT compositor's work is generally lower because IceT uses a single fragment per pixel, whereas our deep compositor composes multiple fragments per pixel generated by non-convex shell topologies. Hence, our deep compositor offers reasonable performance over challenging topologies while maintaining correctness.

We also observe again that the scalability of our compositor is proportional to the number of fragments retired. That number, of course, depends not only on the number of MPI ranks but also, more importantly, on the spatial arrangement of the clusters; this becomes obvious when considering the maximum number of fragments per ray F_{max}. Regardless of how it is partitioned into local per-rank work loads F_{pixel}, this number does not change and presents an upper bound on the overall compositing workload per ray. Furthermore, in our experience, F_{max} will be much smaller than the number of clusters, and hence, scalability is not affected by that but by their spatial arrangement.

Our method generally achieves interactive rates across all the configurations we tested. We have identified optimal load points at 40, 72, and 72 GPUs for Airplane, Small, and Huge Mars Lander, respectively. Since we emulate an in situ process, we

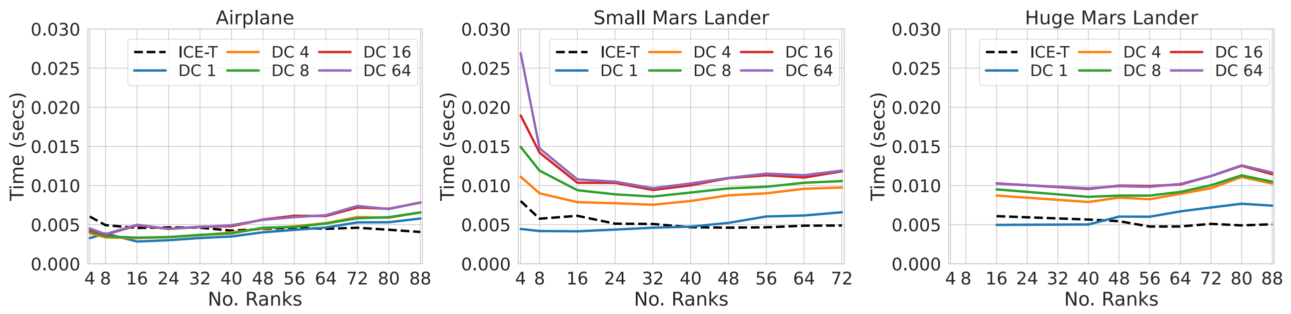


Fig. 14. Compositing times versus increasing ranks over our three datasets: Each plot gives measurements for our deep compositor (DC) with 1, 4, 8, 16, and 64 fragments per pixel. Note that it is possible to end up with more fragment allocations than needed, which produces equivalent curves(e.g., the DC 64 curve closely follows DC 16). We also include measurements for the ICE-T compositor for reference.

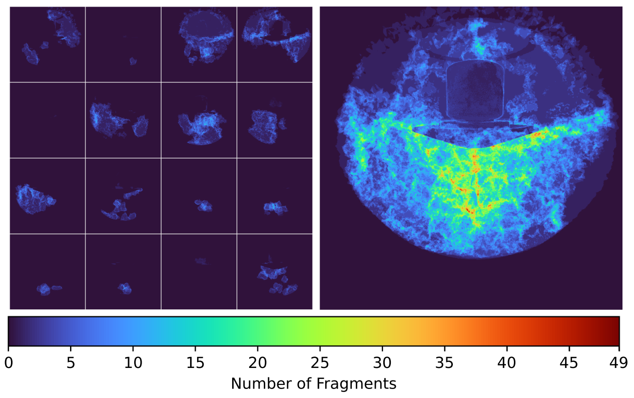


Fig. 15. Heatmaps for fragment counts. Left: per pixel, per rank (F_{local}). Right: per pixel, across all ranks (F_{pixel}).

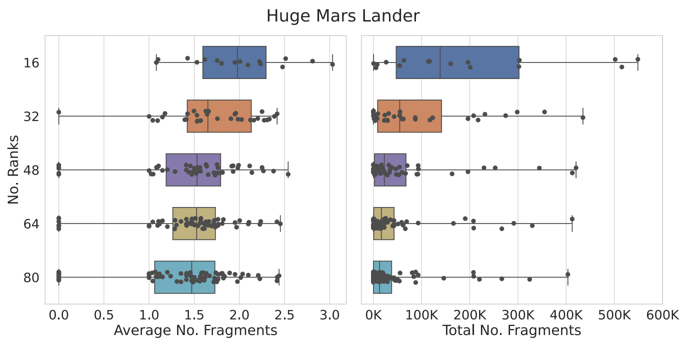


Fig. 16. Box plots for the 16, 32, 48, 64, and 80 rank counts of average (left) and total (right) number of fragments generated by individual ranks while rendering the Huge Mars Lander. We take averages over non-empty pixels where their opacity$>$ 0. Scattered points signify individual ranks' average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

load clusters as is and distribute them in a round-robin order when needed. This approach introduces the possibility of some compute nodes doing less work or rendering empty images. As depicted in Fig. 15 over a view of Huge Mars Lander data, it is evident that certain ranks yield fewer fragments, leading to load-balancing issues. Examining Fig. 16, the standard deviation (STD) of the number of fragments generated per rank is notably high, especially for smaller MPI sizes. There is a considerable reduction in STD for the total number of fragments per rank when moving from 16 to 32 and 32 to 48. However, the improvement is less pronounced when transitioning from MPI size 64 to 80. The number of fragments generated correlates with rendering performance, evident in similar trends around MPI sizes 32 and 64 (see Fig. 12).

D. Limitations

One problem of this framework is load balancing, as naively using the native distribution may cause some ranks to overwork while others wait for that rank to finish. While our focus is primarily on in situ and in-transit use cases, there is room for better distribution schemes. For instance, one straightforward scheme can consider the transfer function and a cluster's scalar value range before loading it into rendering. Another approach could leverage a load balancing concept, as in [37], [38]. While these ideas may enhance rendering times, dynamically moving data around or communicating clusters based on metrics like visibility could limit interactivity, depending on the simulation configuration. We observe that our scalability improves with longer traversals over deeper volumes. However, the initialization cost could be amortized more in shallower volumes, such as the airplane data set.

Another limitation of our element-marching scheme is that it lacks empty space skipping, as it needs to march to the element to realize it is empty. A hybrid structure such as [22] or clustering techniques like [19] could identify and skip those regions.

Our marcher relies on the winding order of the vertices to be consistent and exact. In rare cases, despite meticulous preprocessing, data acquisition and conversion discrepancies can occasionally lead to inconsistent winding orders. Simulation codes are often more robust to such issues than ray marchers. While we did not encounter this issue in our evaluations, it is possible for the marcher to "get lost," meaning that due to incorrect evaluation of orientation tests, the marcher repeatedly visits the same elements, resulting in infinite loops. In such cases, we propose to use mailboxing and similar strategies to break up these loops.

Our connectivity generation (see Section III-A2) is currently done in an offline pre-process. This must be done on the fly for a true in-situ operation. While this should not be hard, it has not yet been done.

V. Conclusion

Our framework addresses challenges posed by large-scale 3D simulations generating unstructured meshes with non-convex domains. Our proposed solution comprises a memory-efficient and low-overhead mixed element ray-marching algorithm, a shell-to-shell traversal scheme, and a deep compositing scheme that allows compositing of the RGBA-Z values obtained across multiple compute nodes in the correct order. These collectively enable interactive rendering of massive data sets with non-trivial and non-convex geometries.

Our evaluation highlights that element connectivity data dominates the memory footprint of the method, which we address using XOR-compaction. This method proved successful, particularly for the larger data sets we tested. Our deep compositor presents a generalization of existing frameworks using a GPGPU API. It enables GPU-to-GPU RDMA in contrast to frameworks using rasterization. Our compositor also gracefully generalizes to a single fragment per pixel per node execution as IceT does.

Our contributions encompass a holistic approach to large-scale 3D simulation visualization, addressing critical issues related to correctness, memory efficiency, and scalability in data-parallel applications. We consider our framework suitable for both in situ and post hoc applications. The proposed framework showcases promising results, paving the way for robust, time-efficient, and in-place analysis of complex simulation data.

Acknowledgments

The authos would like to thank Qi Wu for his valuable support. The Texas Advanced Computing Center (TACC) graciously provided hardware for development and testing. Fun3D data sets are provided by The National Aeronautics and Space Administration (NASA).

References

[1] NASA, “Fun3D manual,” Accessed: Mar. 24, 2022. [Online]. Available: https://fun3d.larc.nasa.gov

[2] P. J. Moran, “FUN3D retropropulsion data Portal-NASA,” 2020, Accessed: Nov. 27, 2023. [Online]. Available: https://data.nas.nasa.gov/fun3d

[3] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, “A sorting classification of parallel rendering,” *IEEE Comput. Graph. Appl.*, vol. 14, no. 4, pp. 23–32, Jul. 1994.

[4] I. Wald, N. Morrical, and S. Zellmann, “A memory efficient encoding for ray tracing large unstructured data,” *IEEE Trans. Vis. Comput. Graph.*, vol. 28, no. 1, pp. 583–592, Jan. 2022.

[5] A. Sahistan et al., “Ray-traced shell traversal of tetrahedral meshes for direct volume visualization,” in *Proc. IEEE Vis. Conf.-Short Papers*, 2021, pp. 91–95.

[6] B. Nelson and R. M. Kirby, “Ray-tracing polymorphic multidomain spectral/hp elements for isosurface rendering,” *IEEE Trans. Vis. Comput. Graph.*, vol. 12, no. 1, p. 114–125, Jan./Feb. 2006.

[7] H. T. Vo et al., “iRun: Interactive rendering of large unstructured grids,” in *Proc. Eurograph. Symp. Parallel Graph. Vis.*, J. M. Favre, L. P. Santos, and D. Reiners, Eds., 2007, pp. 93–100.

[8] G. Marmitt, H. Friedrich, and P. Slusallek, “Efficient CPU-based volume ray tracing techniques,” *Comput. Graph. Forum*, vol. 27, no. 6, pp. 1687–1709, 2008.

[9] P. Muigg, M. Hadwiger, H. Doleisch, and E. Groller, “Interactive volume visualization of general polyhedral grids,” *IEEE Trans. Vis. Comput. Graph.*, vol. 17, no. 12, pp. 2115–2124, Dec. 2011.

[10] N. Morrical, I. Wald, W. Usher, and V. Pascucci, “Accelerating unstructured mesh point location with RT cores,” *IEEE Trans. Vis. Comput. Graph.*, vol. 28, no. 8, pp. 2852–2866, Aug. 2022.

[11] P. Shirley and A. Tuchman, “A polygonal approximation to direct scalar volume rendering,” *ACM Comp. Graph.*, vol. 24, no. 5, pp. 63–70, 1990.

[12] B. Rathke, I. Wald, K. Chiu, and C. Brownlee, “SIMD parallel ray tracing of homogeneous polyhedral grids,” in *Proc. Eurograph. Symp. Parallel Graph. Vis.*, 2015, pp. 33–41.

[13] 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 *Proc. High- Perform. Graph.*, 2019, pp. 7–13.

[14] J. Kruger and R. Westermann, “Acceleration techniques for GPU-based volume rendering,” in *Proc. IEEE Vis.*, 2003, pp. 287–292.

[15] M. Hadwiger, A. K. Al-Awami, J. Beyer, M. Agus, and H. Pfister, “SparseLeap: Efficient empty space skipping for large-scale volume rendering,” *IEEE Trans. Vis. Comput. Graph.*, vol. 24, no. 1, pp. 974–983, Jan. 2018.

[16] H. Wang, G. Xu, X. Pan, Z. Liu, R. Lan, and X. Luo, “A novel ray-casting algorithm using dynamic adaptive sampling,” *Wireless Commun. Mobile Comput.*, vol. 2020, 2020, Art. no. 8822624.

[17] L. Szirmay-Kalos, B. Tóth, and M. Magdics, “Free path sampling in high resolution inhomogeneous participating media,” *Comput. Graph. Forum*, vol. 30, no. 1, pp. 85–97, 2011.

[18] N. Morrical, W. Usher, I. Wald, and V. Pascucci, “Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing,” in *Proc. IEEE Vis.*, 2019, pp. 256–260.

[19] N. Morrical, A. Sahistan, U. Güdükbay, I. Wald, and V. Pascucci, “Quick clusters: A GPU-parallel partitioning for efficient path tracing of unstructured volumetric grids,” *IEEE Trans. Vis. Comput. Graph.*, vol. 29, no. 1, pp. 537–547, Jan. 2023.

[20] G. Marmitt and P. Slusallek, “Fast ray traversal of tetrahedral and hexahedral meshes for direct volume rendering,” in *Proc. Eurograph. IEEE VGTC Vis. Symp.*, 2006, pp. 235–242.

[21] M. Weiler, M. Kraus, M. Merz, and T. Ertl, “Hardware-based ray casting for tetrahedral meshes,” in *Proc. IEEE Vis.*, 2003, pp. 333–340.

[22] A. Aman, S. Demirci, U. Güdükbay, and I. Wald, “Multi-level tetrahedralization-based accelerator for ray-tracing animated scenes,” *Comput. Animation Virtual Worlds*, vol. 32, no. 3/4, 2021, Art. no. e2024.

[23] A. Aman, S. Demirci, and U. Güdükbay, “Compact tetrahedralization-based acceleration structures for ray tracing,” *J. Vis.*, vol. 25, no. 5, pp. 1103–1115, Oct. 2022.

[24] M. Larsen, J. S. Meredith, P. A. Navrátil, and H. Childs, “Ray tracing within a data parallel framework,” in *Proc. IEEE Pacific Visual. Symp.*, 2015, pp. 279–286.

[25] L. Castanie, C. Mion, X. Cavin, and B. Levy, “Distributed shared memory for roaming large volumes,” *IEEE Trans. Vis. Comput. Graph.*, vol. 12, no. 5, pp. 1299–1306, Sep./Oct. 2006.

[26] C. Brownlee, T. Ize, and C. D. Hansen, “Image-parallel ray tracing using OpenGL interception,” in *Proc. Eurograph. Symp. Parallel Graph. Vis.*, 2013, pp. 65–72.

[27] T. Biedert, P. Messmer, T. Fogal, and C. Garth, “Hardware-accelerated multi-tile streaming for realtime remote visualization,” in *Proc. Eurograph. Symp. Parallel Graph. Vis.*, 2018, pp. 33–43.

[28] T. Biedert, K. Werner, B. Hentschel, and C. Garth, “A task-based parallel rendering component for large-scale visualization applications,” in *Proc. Eurograph. Symp. Parallel Graph. Vis.*, 2017, pp. 63–71.

[29] Y. Cao, Z. Mo, Z. Ai, H. Wang, and Z. Zhang, “Parallel visualization of large-scale multifield scientific data,” *J. Visual.*, vol. 22, pp. 1107–1123, 2019.

[30] G. Abram, P. Navrátil, P. Grossett, D. Rogers, and J. Ahrens, “Galaxy: Asynchronous ray tracing for large high-fidelity visualization,” in *Proc. IEEE 8th Symp. Large Data Analy. Vis.*, 2018, pp. 72–76.

[31] K. Moreland, W. Kendall, T. Peterka, and J. Huang, “An image compositing solution at scale,” in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–10.

[32] K. Moreland, “IceT users' guide and reference,” *Sandia Nat. Lab.*, Tech. Rep. SAND2011-5011, Jan. 2011. [Online]. Available: https://www.sandia.gov/app/uploads/sites/150/2021/10/IceTUsersGuide-2-1.pdf

[33] K.-L. Ma, “Parallel volume ray-casting for unstructured-grid data on distributed-memory architectures,” in *Proc. IEEE Symp. Parallel Rendering*, 1995, pp. 23–30.

[34] K.-L. Ma and T. Crockett, “A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data,” in *Proc. IEEE Symp. Parallel Rendering*, 1997, pp. 95–104.

[35] A. V. P. Grosset, A. Knoll, and C. Hansen, “Dynamically scheduled region-based image compositing,” in *Proc. Eurograph. Symp. Parallel Graph. Visual.*, 2016, pp. 79–88.

[36] W. Usher, I. Wald, J. Amstutz, J. Günther, C. Brownlee, and V. Pascucci, “Scalable ray tracing using the distributed framebuffer,” *Comput. Graph. Forum*, vol. 38, no. 3, pp. 455–466, 2019.

[37] H. Childs, M. A. Duchaineau, and K. Ma, “A scalable, hybrid scheme for volume rendering massive data sets,” in *Proc. 6th Eurograph. Symp. Parallel Graph. Vis.*, 2006, pp. 153–161.
[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 Trans. Vis. Comput. Graph.*, vol. 25, no. 7, pp. 2349–2361, Jul. 2019.
[39] I. Wald et al., “OSPRay - A CPU ray tracing framework for scientific visualization,” *IEEE Trans. Vis. Comput. Graph.*, vol. 23, no. 1, pp. 931–940, Jan. 2017.
[40] S. Zellmann, I. Wald, J. Barbosa, S. Dermici, A. Sahistan, and U. Güdükbay, “Hybrid image-/data-parallel rendering using island parallelism,” in *Proc. IEEE 12th Symp. Large Data Analy. Vis.*, 2022, pp. 1–10.
[41] I. Wald, M. Jaroš, and S. Zellmann, “Data parallel Multi-GPU path tracing using ray queue cycling,” *Comput. Graph. Forum*, vol. 42, no. 8, 2023, Art. no. e14873.
[42] S. Frey, F. Sadlo, and T. Ertl, “Explorable volumetric depth images from raycasting,” in *Proc. XXVI Conf. Graph., Patt. Images*, 2013, pp. 123–130.
[43] J. Shade, S. Gortler, L.-W. He, and R. Szeliski, “Layered depth images,” in *Proc. 25th Ann. Conf. Comp. Graph. Interact. Tech.*, New York, NY, USA, 1998, pp. 231–242, doi: 10.1145/280814.280882.
[44] O. Fernandes, S. Frey, F. Sadlo, and T. Ertl, “Space-time volumetric depth images for in-situ visualization,” in *Proc. IEEE Symp. Large Data Analy. Vis.*, 2014, pp. 59–65.
[45] A. Gupta et al., “Parallel compositing of volumetric depth images for interactive visualization of distributed volumes at high frame rates,” in *Proc. Eurograph. Symp. Parallel Graph. Vis.*, 2023, pp. 25–35.
[46] A. Gupta et al., “Efficient raycasting of volumetric depth images for remote visualization of large volumes at high frame rates,” in *Proc. IEEE 16th Pacific Vis. Symp.*, Los Alamitos, CA, USA, 2023, pp. 61–70.
[47] H. Childs et al., “A terminology for in situ visualization and analysis systems,” *Int. J. High Perform. Comput. Appl.*, 2020, pp. 676–691.
[48] U. Ayachit et al., “ParaView catalyst: Enabling in situ data analysis and visualization,” in *Proc. 1st Workshop Situ Infrastructures Enabling Extreme-Scale Anal. Vis.*, New York, NY, USA, 2015, pp. 25–29.
[49] B. Whitlock, J. M. Favre, and J. S. Meredith, “Parallel in situ coupling of simulation with a fully featured visualization system,” in *Proc. Eurograph. Symp. Parallel Graph. Vis.*, 2011, pp. 101–109.
[50] 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 *Proc. 1st Workshop Situ Infrastructures Enabling Extreme-Scale Anal. Vis.*, Austin, TX, USA, 2015, pp. 30–35.
[51] M. Larsen et al., “The ALPINE in situ infrastructure: Ascending from the ashes of Strawman,” in *Proc. Situ Infrastructures Enabling Extreme-Scale Anal. Vis.*, New York, NY, USA, 2017, p. 42–46.
[52] 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 *Proc. Situ Infrastructures Enabling Extreme-Scale Analy. Vis.*, New York, NY, USA, 2019, pp. 12–16.
[53] G. Aupy, B. Goglin, V. Honoré, and B. Raffin, “Modeling high-throughput applications for in situ analytics,” *Int. J. High Perform. Comput. Appl.*, vol. 33, no. 6, pp. 1185–1200, 2019.
[54] D. E. DeMarle and A. C. Bauer, “In situ visualization with temporal caching,” *Comput. Sci. Eng.*, vol. 23, no. 3, pp. 25–33, 2021.
[55] 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., Berlin, Germany: Springer, 2018, pp. 273–293.
[56] 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 *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, New York, NY, USA, 2019, Art. no. 61.
[57] NVIDIA Corp., “NVIDIA OptiX ray tracing engine,” Accessed: Nov. 27, 2023. [Online]. Available: https://developer.nvidia.com/optix
[58] I. Wald, N. Morrical, and E. Haines, “OWL–the optix 7 wrapper library,” 2020. Accessed: Nov. 27, 2023. [Online]. Available: https://github.com/owl-project/owl
[59] M. Pharr, W. Jakob, and G. Humphreys, “Primitives and intersection acceleration,” in *Physically Based Rendering: From Theory To Implementation*. San Mateo, CA, USA: Morgan Kaufmann, 2021, pp. 169–225.
[60] W. Schroeder, K. Martin, and B. Lorensen, *The Visualization Toolkit*, 4th ed.. Clifton Park, NY, USA: Kitware, 2006.
[61] M. Ikits, J. Kniss, A. Lefohn, and C. Hansen, “Volume rendering techniques,” in *GPU Gems*, R. Fernando, eds., Reading, MA, USA: Addison-Wesley, 2004, pp. 667–692.
[62] T. Porter and T. Duff, “Compositing digital images,” *ACM Comput. Graph.*, vol. 18, no. 3, pp. 253–259, Jul. 1984.
[63] 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 Trans. Vis. Comput. Graph.*, vol. 23, no. 6, pp. 1677–1690, Jun. 2017.
[64] S. Eilemann and R. Pajarola, “Direct send compositing for parallel sort-last rendering,” in *Proc. Eurograph. Symp. Parallel Graph. Vis.*, 2007, pp. 29–36.
[65] R. Gershbein and P. Hanrahan, “A fast relighting engine for interactive cinematic lighting design,” in *Proc. 27th Annu. Conf. Comput. Graph. Interactive Techn.*, 2000, pp. 353–358.
[66] D. Stanzione, J. West, R. T. Evans, T. Minyard, O. Ghattas, and D. K. Panda, “Frontera: The evolution of leadership computing at the national science foundation,” in *Proc. Pract. Experience Adv. Res. Comput.*, 2020, pp. 106–111.
[67] K. E. Jones, “Summit supercomputer simulates how humans will ‘brake’ during mars landing,” 2019. [Online]. Available: https://www.ornl.gov/news/summit-simulates-how-humans-will-brake-during-mars-landing
[68] H. Childs et al., “VisIt: An end-user tool for visualizing and analyzing very large data,” in *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, New York, NY: Chapman and Hall/CRC, 2012, pp. 357–372.
[69] J. Ahrens, B. Geveci, and C. Law, “ParaView: An end-user tool for large-data visualization,” in *Visualization*, C. Handbook, D. Hansen, and C. R. Johnson, Eds., London, U.K.: Butterworth-Heinemann, 2005, pp. 717–731.
[70] P. Andersson, J. Nilsson, T. Akenine-Möller, M. Oskarsson, K. Åström, and M. D. Fairchild, “ꟻLIP : A difference evaluator for alternating images,” in *Proc. ACM Comput. Graph. Interactive Techn.*, vol. 3, no. 2, pp. 15:1–15:23, 2020.

Alper Sahistan received the BS and MS degrees in computer engineering from Bilkent University, Ankara, Türkiye, in 2019 and 2022, respectively. He is currently working toward the PhD degree from the University of Utah, Salt Lake City, Utah, and is currently working under the supervision of valerio pascucci as a member of the CEDMAV Group, Scientific Computing and Imaging (SCI) Institute. His research interests include high-performance computing, real-time ray tracing, and direct volume rendering.

Serkan Demirci received the BS and MS degrees in computer engineering from Bilkent University, Ankara, Türkiye, in 2018 and 2021, respectively. He is working toward the PhD degree in computer engineering with Bilkent University. His research interests include rendering techniques for three-dimensional scenes, volume visualization, virtual and augmented reality, crowd simulation, interactive conversational agents, deep learning, and computational geometry.

Ingo Wald received the master’s degree from Kaiserslautern University, Kaiserslautern, Germany, and the PhD degree from Saarland University, Saarbrücken, Germany both in ray tracing related topics. He also served as a post-doctoral researcher with the MPI Saarbrücken, as a research professor with the University of Utah, and as the technical leader for Intel’s software-defined rendering activities (in particular, Embree and OSPRay). He is a director of Ray Tracing, NVIDIA. He has co-authored more than 100 papers, multiple patents, and several widely used software projects around ray tracing. His research interests revolve around all aspects of efficient and high-performance ray tracing, from visualization to production rendering, from real-time to offline rendering, and from hard- to software.

Stefan Zellmann (Member, IEEE) received the PhD degree in computer science from the University of Cologne, in 2014. He is a senior researcher and computer science Lecturer with the University of Cologne. Zellmann also holds a degree in business information systems. His research focuses on the interface between high-performance computing and real-time rendering. He is also the maintainer of and contributor to various visualization-related open-source projects.

João Barbosa received the master’s degree from the University of Minho in Braga, Portugal. He is currently working toward the PhD degree. He is an assistant researcher with INESC TEC (Institute for Systems and Computer Engineering, Technology and Science) in Porto, Portugal. His research is focused on high-performance computing, with a special interest in high-performance computer graphics. João Barbosa is also the director of Operations for the Deucalion supercomputer with the Minho Advanced Computing Center, pivotal in advancing Portugal’s computing capabilities.

Nate Morrical received the BS degree in computer science from Idaho State University, Pocatello, Idaho, where he researched interactive computer graphics and computational geometry under Dr. John Edwards. He is currently working toward the PhD degree with the University of Utah, Salt Lake City, Utah, and is currently working under Valerio Pascucci as a member of the CEDMAV Group, Scientific Computing and Imaging Institute (SCI). His research interests include high-performance GPU computing, real-time ray tracing, and human-computer interaction.

Uğur Güdükbay (Senior Member, IEEE) received the BS degree in computer engineering from the Middle East Technical University, Ankara, Türkiye, in 1987, and the MS and PhD degrees in computer engineering and information science from Bilkent University, Ankara, in 1989 and 1994, respectively. He did research as a postdoctoral fellow with the Human Modeling and Simulation Laboratory, the University of Pennsylvania, Philadelphia, PA, USA. He is a professor in the Department of Computer Engineering, Bilkent University. His research interests include computer graphics and computational geometry. He is a senior member of ACM. He is an associate editor of Computer Animation and Virtual Worlds and Signal, Image and Video Processing.