Quick Clusters: A GPU-Parallel Partitioning for Efficient Path Tracing of Unstructured Volumetric Grids

Nate Morrical[†,‡] Alper Sahistan[§] Uğur Güdükbay[§] Ingo Wald[‡] Valerio Pascucci[†]
[†]SCI Institute, University of Utah [‡]NVIDIA [§]Bilkent University

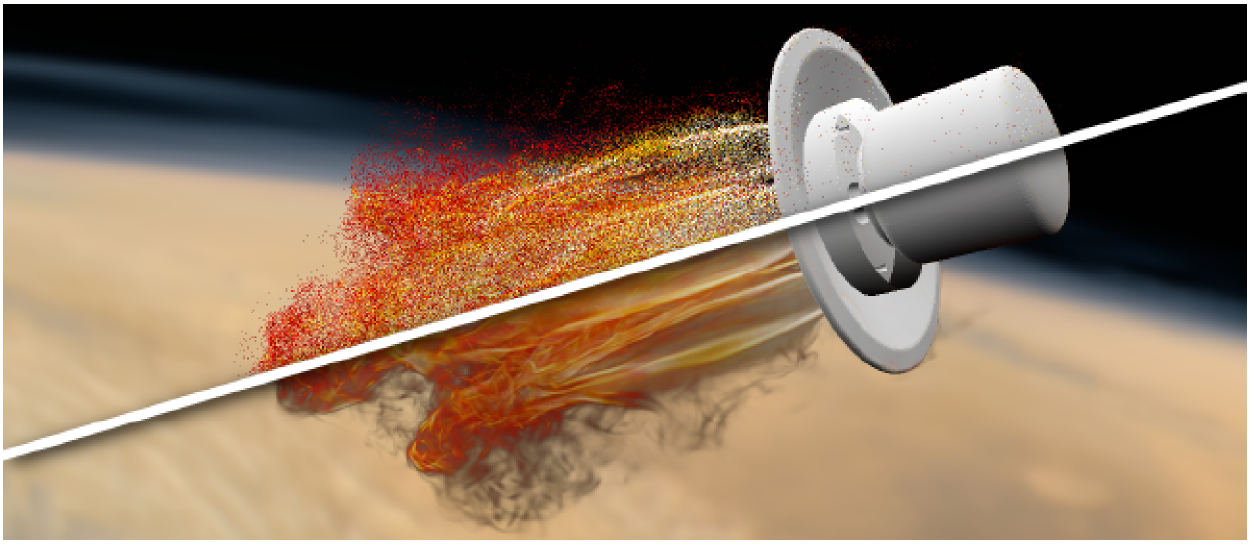


Fig. 1: Using our method, we can render volumetric shadows on large unstructured datasets while maintaining interactive frame rates. As shown in the image above (top left is one sample per pixel (spp), bottom right is rendered to $4K$ spp), we can achieve up to 12 frames per second (fps) at 1 spp, at $22.2M$ rays per second (rps) on the 788M element Mars Lander dataset on a single workstation GPU—nearly 5 × faster than the state-of-the-art that required *eight GPUs in a supercomputer* [49]—and at significantly higher image quality and lower technical complexity than prior works.

Abstract—We propose a simple yet effective method for clustering finite elements to improve preprocessing times and rendering performance of unstructured volumetric grids without requiring auxiliary connectivity data. Rather than building bounding volume hierarchies (BVHs) over individual elements, we sort elements along with a Hilbert curve and aggregate neighboring elements together, improving BVH memory consumption by over an order of magnitude. Then to further reduce memory consumption, we cluster the mesh on the fly into sub-meshes with smaller indices using a series of efficient parallel mesh re-indexing operations. These clusters are then passed to a highly optimized ray tracing API for point containment queries and ray-cluster intersection testing. Each cluster is assigned a maximum extinction value for adaptive sampling, which we rasterize into non-overlapping view-aligned bins allocated along the ray. These maximum extinction bins are then used to guide the placement of samples along the ray during visualization, reducing the number of samples required by multiple orders of magnitude (depending on the dataset), thereby improving overall visualization interactivity. Using our approach, we improve rendering performance over a competitive baseline on the NASA Mars Lander dataset from 6× (*1 frame per second* (fps) and *1.0 M rays per second* (rps) up to now *6 fps* and *12.4 M rps*, now including volumetric shadows) while simultaneously reducing memory consumption by 3× (*33 GB* down to *11 GB*) and avoiding any offline preprocessing steps, enabling high-quality interactive visualization on consumer graphics cards. Then by utilizing the full 48 GB of an RTX 8000, we improve the performance of Lander by 17× (*1 fps* up to *17 fps*, *1.0 M rps* up to *35.6 M rps*).

Index Terms—Ray Tracing, Path Tracing, Volume Rendering, Scientific Visualization, Delta Tracking

1 Introduction

Unstructured meshes are an enticing format for large-scale volumetric simulations, as elements can be adaptively distributed such that important regions receive more elements—increasing local accuracy and precision—while less important regions receive fewer elements—saving valuable memory resources. Take, for example, the NASA Landing Gear (shown in Figure 2), where the largest element is 4096× larger than the finest element. An equal precision regular grid transformation would consume nearly 4 petabytes of data, while the native unstructured mesh format requires only 11.6 gigabytes.

The power of these meshes comes from their flexibility, as elements can twist and bend to represent complex shapes and domains. Frameworks supporting these unstructured grids naturally extend to support adaptive mesh refinement data, as hexahedra can be used to represent voxels in same-level bricks. Then, a combination of hexahedra, wedges, pyramids, and tetrahedra can be used to interpolate between neighboring bricks of different resolutions [46,51]. However, this flexibility comes at a cost, as few assumptions can be made about the structure of the underlying elements.

Manuscript received 31 March 2022; revised 1 July 2022; accepted 8 August 2022. Date of publication 27 September 2022; date of current version 2 December 2022. This article has supplementary downloadable material available at https://doi.org/10.1109/TVCG.2022.3209418, provided by the authors. Digital Object Identifier no. 10.1109/TVCG.2022.3209418

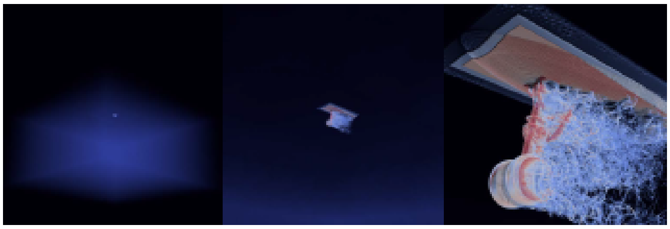


Fig. 2: Illustration of the spatial domains common to large unstructured data, here shown for the NAsA Landing Gear AMR dataset. Elements throughout the simulation adapt in resolution to reduce memory consumption. From left to right, we progressively zoom in.

Challenges emerge at the level of a single primitive. Although, in their entirety, finite element meshes can adapt in resolution to improve overall memory consumption, a single finite element consumes a considerable amount of memory. Elements must store vertices and vertex indices explicitly, unlike regular grids where the geometry of a voxel is almost entirely implicit. As a result, meshes often need to be compressed to fit within system resources; however, this compression can easily take several hours and must support on-the-fly, localized decompression during rendering [49].

Then, rendering these volumetric meshes requires quickly identifying which element contains a given query point and interpolating that element's corresponding per-vertex values. These queries require constructing auxiliary data structures (trees or mesh connectivity) over the elements to facilitate fast element point location [39,42]. However, many of these structures are surprisingly challenging to implement, and take a significant amount of time and memory to compute [6,49].

Once these structures are built, they can be used to optimize direct volume rendering for point location queries that are executed along with a set of view-aligned rays. However, determining the number of samples required to compute clean-looking images is another challenge. For regular grids, volumes can be accurately rendered by relating the step size between each query to the voxel width, thus sampling approximately every voxel along the ray. However, with unstructured meshes, the widths of fine elements are much smaller than the widths of the coarse elements (see Fig. 2). Unfortunately, relating the step size to the finest element width results in a prohibitively expensive number of samples per ray in coarser regions, while larger step sizes undersample the volume.

Thus, before these datasets can be queried, they often must undergo an expensive compression process accompanied by constructing several auxiliary data structures that can take several hours to complete. Then during rendering, the irregularity of these large unstructured meshes causes modern methods to oversample coarse regions while undersampling fine details within the data. These obstacles, and many more, hinder the adoption of direct volume rendering as a method for visualizing unstructured volumetric grids.

Fortunately, many of these obstacles can be overcome by challenging common assumptions made by prior works. One common assumption is that preprocessing trees are required to cluster nearby elements. These preprocessing trees are then discarded in favor of hardware-accelerated trees built over these clusters during runtime. However, there are much simpler and much faster means of clustering elements that do not require an expensive preprocessing step. Then, many prior methods assume alpha-composited ray marching is the only method for rendering volumetric data; and yet, alternative Monte Carlo volume rendering methods exist and have several unexplored yet appealing properties for unstructured grid visualization. These Monte Carlo methods amortize sampling expense over time, trading noise in intermediate images for significantly faster frame rates and improved final image quality. Finally, by targetting just the primary sources of memory consumption of these datasets, both the mesh and corresponding structures can be compressed in parallel immediately before runtime to enable visualization on a broader set of systems and architectures. By optimizing the implementation complexity and runtime performance of mesh compression and visualization, we believe that more applications will be able to benefit from the power and flexibility of these adaptive finite element formats.

To this end, we present a novel approach to unstructured mesh visualization that leverages the optimal clustering properties of Hilbert space-filling curves [28] in combination with high-performance GPU ray tracing APIs to enable accessible, high quality, high performance, low memory consumption rendering, all with little to no preprocessing. We sort all unstructured elements along this Hilbert curve in parallel to establish a structure for otherwise unstructured data. Then, we repeatedly leverage this resulting spatial locality to quickly generate collections of clusters that can be used to address the variety of challenges posed by unstructured grid rendering, including memory consumption of vertex indices, hardware BVH memory consumption, and rendering performance. We then show how these clusters can be combined with a Monte Carlo alternative to ray marching, called *Delta Tracking*, to amortize rendering expense over time–at the trade-off of noise at intermediate frames–to improve visualization interactivity by adaptively sampling cluster-localized density estimates.

More specifically, we present the following contributions:

- a method that leverages Hilbert curves and a parallel GPU sort to cluster unstructured elements quickly,
- a strategy that leverages these clusters to significantly reduce the memory footprint of hardware-compatible structures for accelerated unstructured point queries,
- a parallel mesh re-indexing scheme that uses coarse clusters to reduce the memory required by 2×, and
- two methods that transform object-oriented clusters into spatial partitions for use in adaptive sampling.

2 Related Works

Recent works have made significant progress in unstructured volume rendering. Here, we cover these prior works and how they address memory consumption, runtime performance, preprocessing times, and ease of implementation.

2.1 High-Performance Element Traversal

Early methods [41] rasterized tetrahedral mesh volumes by sorting elements from front to back. This sorting step scales poorly to large datasets, resulting in slow interactivity during camera manipulation. So instead of sorting elements by visibility order, modern methods use auxiliary data structures like bounding volume hierarchies (BVHs) or element connectivity data to traverse from front to back.

Recent methods for unstructured mesh visualization use hardware ray-tracing cores, otherwise known as *RT cores*. Unlike single-instruction, multiple-data cores of a GPU, modern ray-tracing cores follow a multiple-instruction, multiple-data execution model that is better suited for divergent tree traversal [1, 15, 35]. Unfortunately, these cores appear as a "black box" to end-users, as exact tree and traversal implementations vary from one vendor to the next and from hardware generation to generation. Almost all aspects of these trees—including tree width, construction, node bounds, and quantization — are made private. This makes utilizing these trees for tasks other than ray-triangle intersection testing a challenge, as tree traversal cannot be customized, nor can internal tree nodes be accessed. However, by leveraging these frameworks, users do not need to write any code to construct trees or for ray-tree traversal, dramatically reducing implementation complexity.

Recent methods [32, 45] use GPU ray tracing frameworks to improve tree construction and query performance of unstructured meshes by transforming the task of finite element point containment into a ray-tracing problem that can be hardware accelerated. Although query performance is fast, these works require auxiliary triangles from element faces to leverage ray-triangle hardware intersectors, where each triangle identifies two neighboring elements. Computing these triangles requires a lengthy sequential triangle insertion step into an unordered map, and once found, a hardware tree must be built over them. These data structures consume a prohibitive amount of time and memory to compute and store and therefore do not scale to large meshes.

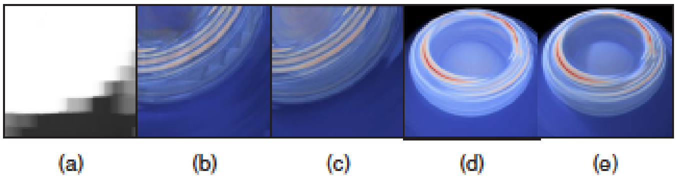


Fig. 3: a) clusters of elements from the TACC Japan Earthquake dataset, used for adaptive sampling. b) artifacts caused by adaptive ray marching methods [31, 50] due to undersampling. c) unbiased adaptive sampling using our delta tracking approach. d) and e) demonstrate the benefits of shadows on the perception of depth, which we additionally support thanks to Delta Tracking. (Intermediate images converged to 4000 SPP.)

Alternatively, rays can march from element to element using connectivity data [34]. The recent work by Sahistan et al. [40] improves the performance and complexity of tetrahedral mesh marching by using GPU ray tracing frameworks and cores to identify where rays enter and exit the volume. Then, they apply an exclusive-or compacting scheme by Aman et al. [2, 3] to reduce memory bandwidth and significantly improve performance when marching from one tetrahedron to the next. Unfortunately, computing the required connectivity data is often an arduous preprocess, and storing it quickly becomes prohibitively expensive, especially as datasets grow larger [19]. The approach by Aman et al. also requires tetahedralizing pyramids, wedges, and hexahedra, which increases the number of elements, and, therefore, memory consumption up to 6× [32, 34].

2.2 Space Skipping and Adaptive Sampling

As these data sets grow large and irregular (cf. Figure 2), it becomes a challenge to adequately sample small features within a large domain. If done naively, too small of a step size between queries results in oversampling coarse elements, which is detrimental to rendering performance. Increasing the step size improves performance but results in undersampling small features of interest. Therefore, sampling rates need to adapt to the underlying data. However, to our knowledge, very few works address this issue for unstructured meshes, and none do so in a way that is free from visual artifacts.

For adaptive sampling, prior works have proposed to adapt the step size along the ray depending on local statistics or the resolution of the data [12, 21, 26, 30, 31, 50]. Recent methods [31, 50] trace rays through non-overlapping clusters of unstructured elements, skipping empty space and adapting the sampling rate by the variance of the scalar field within each cluster. Due to the “black box” nature of hardware-accelerated trees and the requirement that clusters do not overlap, these methods require an expensive top-down KD tree build over the unstructured elements. This KD tree construction can take several hours, as each tree level requires sorting elements to compute split planes. This method also relies on opacity correction to account for the now spatially varying sampling rates, as proposed by Engel et al. [10]; however, this opacity correction is more beneficial in the context of sheer warp algorithms [24]—where the apparent thickness of a slice of voxels depends on the viewing angle—and does not correct for undersampling errors in high-density regions. As a result, artifacts can be seen during adaptive sampling, as demonstrated in Figure 3.

In the context of regular grids and cinematic rendering, some applications [23] drop alpha-composited ray marching in favor of stochastic null collision methods like Delta Tracking, as tracking methods have a lower algorithmic complexity with respect to scatter interactions than alpha-composited ray marching. This is because composited marching methods require secondary rays be traced for each sample taken along a primary ray, which results in an exponential number of rays as scatter interactions increase. To address this, shadow maps can be used to cache visibility from light sources to maintain camera interactivity with alpha-composited ray marching [5]; though, these maps must be recomputed on transfer function and lighting changes, and consume additional memory. For our application, these data structures would require separate techniques to be developed to support unstructured grids. Alternatively, if low frequency shadows are acceptable, fewer samples can be taken along shadow rays. On the other hand, tracking methods require only a linearly increasing number of rays per scatter interaction, as these methods only consider the particle at the sampled distance, and therefore do not require any precomputation to maintain interactivity.

Additionally, these tracking methods allow for artifact-free adaptive sampling [43, 44, 53] through localized maximum density estimates along the ray. Kalos et al. [44] propose traversing through an easy-to-compute grid of macrocells to identify more tightly bounding local maximum density. They use a 3D digital differential analyzer (DDA) algorithm to traverse through these cells, reading local density estimates from the macrocells to adapt the underlying sampling rate. In the context of scientific visualization, the work by Günther et al. [16] employs a macrocell-based Delta Tracking approach to render finite-time Lyapunov exponent fields adaptively. Recent prior works [18, 27] also propose moving to Delta Tracking methods to achieve adaptive, high-quality visualization of medical volumes.

To our knowledge, no prior works have explored the use of adaptive Delta Tracking for unstructured mesh visualization. However, as shown in Figure 3, this Monte Carlo approach has clear benefits for performance and image quality. We refer to our supplemental for more information on how alpha-composited ray marching and Monte Carlo Delta Tracking compare.

2.3 Compression

As these meshes grow large, storing these datasets within GPU memory is challenging. Prior works optimize the memory footprint of the mesh as well as corresponding acceleration structures to reduce memory consumption. Naive approaches construct bounding volume hierarchies down to the individual element with a branching factor of two, resulting in a significant number of internal nodes that consume a large amount of memory. To remedy this, prior works [9, 11, 48] use wider branching factors to reduce the number of internal nodes significantly and, therefore, overall memory consumption. Benthin et al. [6] additionally show that internal nodes of wide BVHs can be quantized relative to their parent bounds, further reducing the bytes used per node. Unfortunately, these trees are incompatible with ray-tracing cores and are slow to traverse in software; however, the works by Wald and Morrical [32, 45] compare the memory consumption of a four-wide quantized BVH against a hardware-accelerated BVH used by NVIDIA's Turing architecture and demonstrate a similar footprint.

Ströter et al. [42] propose to sort tetrahedra in parallel on a Morton space-filling curve, then construct a memory-efficient OLBVH similar to the LBVH by Lauterbach et al. [25]. Their approach requires precomputing neighboring element connectivity data, which can be done for tetrahedral meshes using the GPU-optimized structure by Mueller-Roemer et al. [33]. Although their method does consume less memory than hardware-accelerated trees, this compression comes at a performance cost, as their method is several magnitudes slower than that by Wald [45], even on GPUs using a software fallback to ray-tracing cores. This method also requires users to construct and traverse these trees themselves.

Wald et al. [49] recently proposed a memory-efficient encoding that compromises rendering performance to reduce memory consumption significantly. Their work constructs a wide BVH with quantized nodes and then reduces the number of bits per element index by dividing the mesh into submeshes referencing no more than 2^{16} vertices. Then, a per-leaf element reordering is used to reduce the memory footprint of child pointers within the tree. Though their method saves an impressive amount of memory, constructing these wide compressed trees is complex and requires several hours of preprocessing. Traversal is also nontrivial, as nodes must be decompressed on the fly, and it requires eight nodes on a supercomputer to achieve a semi-interactive 2.5 fps on the Small NASA Mars Lander dataset. Then, like the work by Morrical et al. [31], Wald et al. discard the top levels of their data structure, substituting with a second top-level tree that is compatible with ray-tracing cores.

3 Method

We present a high-quality direct volume rendering method for large unstructured grids with a low implementation complexity that still scales to very large data. The proposed method focuses on GPU architectures with ray-tracing cores; however, in theory, the techniques we offer should also work with CPU ray tracing frameworks and GPUs with software fallbacks to ray tracing hardware.

3.1 Baseline Unstructured Mesh Renderer

We begin by implementing a simple baseline method [32] with an off-the-shelf GPU ray tracing framework compatible with ray-tracing cores. By building off these frameworks, we significantly reduce the implementation complexity and improve the performance of our method, especially for BVH construction and traversal. Naturally, these frameworks will impose several constraints that will direct the design choices with our method, as we cannot change internal aspects of these frameworks like BVH construction or ray traversal.

First, we load our unstructured mesh into memory. Vertices of the mesh are stored in a list of `float3` representing x, y, and z coordinates, respectively. For simplicity, we assume that each mesh vertex is associated with a single corresponding scalar value. Then, we load all tetrahedra indices into a list of 32-bit `ints`, such that each tetrahedron corresponds to four unique, contiguous indices. Pyramid indices are stored in a separate list, as are wedges and hexahedra, each using five, six, and eight indices per element, respectively. Thus, our unstructured mesh consists of one list of vertices, one list of scalar values, and four lists of indices—one for each element type. With this format, we can efficiently load elements from disk by storing and loading these lists as large contiguous binary arrays that we copy all at once into memory. Once loaded, we copy these lists to the GPU for on-the-fly pre-processing.

Modern GPU ray tracing frameworks focus on ray-triangle intersection testing; however, our primitives are not triangles. Prior works [32, 45] tessellate element faces, using ray-triangle intersectors to significantly improve query performance; however, triangulating element faces consumes a prohibitive amount of memory. Therefore, we instead use a more memory-efficient custom primitive type, where ray-tracing cores traverse through a tree of bounding boxes but return to software when query points intersect the leaves. Note that hardware-accelerated rays can be made to act like point queries by setting the ray origin to the query point, then choosing an arbitrary direction and setting the `T_MIN` and `T_MAX` values to 0. So long as `T_MIN` equals 0, all primitive bounding boxes intersecting the ray origin will be returned to the user.

To use a custom primitive, these ray tracing APIs require us to supply primitive bounding boxes and a handwritten intersection test. So for every element in parallel, we compute a bounding box over that element's vertices, storing bounding boxes for tetrahedra, wedges, pyramids, and hexahedra in separate lists. We also compute a global bounding box in parallel over all mesh vertices by using atomic minimum and maximum functions over the vertex coordinates. Next, we create a custom geometry structure for each element type, allowing us to write four different intersection tests in our ray tracing framework. The appropriate intersection test will be called when a ray hits a bounding box belonging to a particular element type. We then construct a tree over these lists of bounding boxes using the ray-tracing API's high-performance tree construction method.

Next, we write a custom intersection program for each element type. When a ray hits a bounding box containing an element, we read all element indices and vertices into local registers. Then, we test to see if the ray origin is contained within our element. If so, we interpolate the corresponding per-vertex values, returning the result in a register associated with the ray—typically called a *ray payload register*. This intersection test and interpolation can be done using Newton's method, and we provide relevant code for them in our supplemental.

Finally, we need a colormap to convert scalar data values sampled from the volume into colors and corresponding alpha-transparency values to render this data volumetrically. Typically this colormap is editable at runtime to enable interactive exploration of the data, where each edit uploads the updated colormap to the GPU. To create our final image, we write a ray generation program where we trace view-aligned rays out from the camera origin and compute an intersection distance to the front and the back of the global volume bounds. We then march along these rays, sampling the volume at a user-specified sampling rate, using the colormap to transform the sampled scalar value into color and corresponding alpha value, which we can composite from front to back and store in our image framebuffer.

3.2 Tree Compression through Hilbert Clusters

This baseline method by Morrical [32] is a good starting point, as we can render unstructured volumes without lengthy preprocessing steps. Right now, we only require constructing a hardware-accelerated tree, which we can do in real-time using our GPU ray tracing framework. However, some fundamental issues with the current approach prevent us from rendering large unstructured grids. One limitation is that our trees currently consume too much memory.

3.2.1 Hilbert Leaf Clusters

Currently, we construct our trees down to the individual element. This results in a costly number of internal nodes near the leaves of the tree—as trees grow exponentially with depth—and as a consequence, our BVH consumes more memory than the unstructured mesh does. To solve this, we could follow the method by Wald et al. [50], and build a second, highly compressed BVH on the CPU in a top-down process. Then, we could follow the nested-hierarchies approach proposed by Gralka et al. [14] and construct a hardware-compatible tree over highly compressed treelets to still leverage the performance of ray-tracing cores. However, as shown by prior works, constructing compressed trees top-down on the CPU requires several hours of preprocessing time and introduces technical complexity to the method.

Interestingly, the work by Gralka et al. demonstrates that compressed treelets provide performance benefits over simpler flat lists of primitives only when a treelet stores more than 128 primitives. For smaller clusters of primitives, they suggest forgoing treelets in favor of simpler flat lists of elements contiguous in memory. With these small flat primitive clusters, memory is reduced by eliminating bounding boxes and child/node pointers in the tree. Additionally, as shown by an earlier work by Wald. [45], the hardware-accelerated BVHs we currently build have a similar memory footprint to compressed quantized BVHs like those by Benthin et al. [6]. Therefore, we can achieve nearly all the memory optimizations proposed by Wald's compressed data structure [50] by aggregating primitives into small flat lists less than 128 primitives in length, then constructing a hardware-accelerated BVH over these clusters. Furthermore, as shown by Gralka et al., for flat lists containing 16 primitives or fewer, traversal performance should still greatly benefit from hardware acceleration. From here on, we refer to these clusters as *Leaf Clusters*.

Unfortunately, in the current state of our data, the underlying elements of our mesh are ordered unpredictably. If we were to group sets of N elements into one primitive bound, neighboring elements in memory may lie on opposite sides of the spatial domain. Although we would significantly reduce our acceleration structure's memory footprint, our primitive bounding boxes could contain a significant amount of empty space and would very likely overlap. As a result, our rendering performance would quickly degrade to non-interactive frame rates due to poor culling performance. Indeed, through our experimentation, we found that simply aggregating two neighboring elements together in their original memory order resulted in a complete lock-up of the system on the data sets we use for evaluation.

Fortunately, this element ordering in memory can be easily changed without affecting the spatial data representation, and we can do so without requiring secondary trees like those used by prior works. So as a first step, we sort our elements along a space-filling curve exhibiting nice clustering properties, using an off-the-shelf GPU radix sort. For simplicity, we recommend the radix sort routine made available in NVIDIA's CUB library, though the fast four-way radix sort by Ha et al. [17] would also work well.

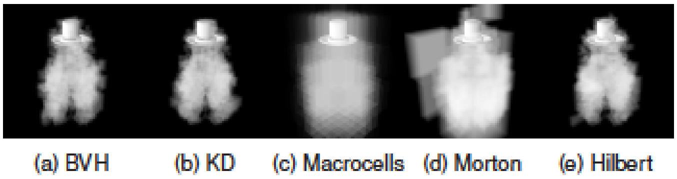


Fig. 4: A visual comparison between clustering methods on the Mars lander for 100K clusters.

The space-filling curve we use is vital to generating high-quality clusters without significant overlap. At first, we explored Morton order curves, which can be computed by first quantizing element centroids to a fixed grid spanning the bounding volume of the entire dataset, then interlacing the bits of the individual quantized dimensions of the element centroids. For the purposes of our application, we used 48-bit codes, with 16 bits allocated per dimension. These codes might not be unique per-polyhedra, as the quantization process might assign two or more polyhedra to the same cell. However, only a few polyhedra will receive identical codes, and when they do, we assume they will be sufficiently nearby in space for reasonable clustering.

Sorted elements can then be clustered by dropping N least significant bits in these codes and then clustering neighboring elements with identical sub-codes. To establish intuition on this process, these Morton codes can be thought of as a round-robin middle-split KD tree, and by dropping N least significant bits and clustering elements with identical subcodes, this process can be interpreted as extracting the internal nodes from this implicit KD tree. This process is very similar to the first step employed by parallel tree construction routines like the LBVH proposed by Lauterbach et al. [25], with the only difference being that we compute just the bounds of the leaves of LBVH, rather than the full tree.

Unfortunately, we ran into several undesirable issues with Morton codes. First, the number of least significant bits to drop is a user-defined parameter, and it is difficult to determine what this number should be. Drop too many bits, and all elements get clustered together; drop too few, and we end up with many leaf clusters containing only a single element. This is problematic because we want to keep the number of primitives per leaf less than 128 to still benefit from hardware acceleration from ray-tracing cores, but we also need more than one element per cluster; otherwise, we do not realize any memory savings. Alternatively, we could choose to cluster primitives into equal length subsets, thus guaranteeing each leaf cluster contains exactly N primitives. However, Morton codes exhibit undesirable large jumps at higher levels of the implicit KD tree, which result in suboptimal bounds when clustering equal length subsets together.

So instead, we sort elements by their centroid on a Hilbert space-filling curve. Just like a Morton curve, we can quantize element centroids to a fixed resolution grid before sorting key-value pairs with a parallel GPU radix routine. However, as Moon et al. [28] demonstrated, Hilbert curves provide more optimal clustering properties and do not exhibit the large undesirable jumps that Morton curves do. Although Hilbert codes are slightly more challenging to compute than Morton codes, they are still nearly instant to compute relative to treelets. Still, for reference, we include a *non-recursive* Cartesian-to-Hilbert implementation in our supplemental material, as described by Butz [8], then simplified by Moore [29]. A visual comparison between clusters generated from BVH nodes, KD tree nodes, Macrocells, Morton curves, and Hilbert curves can be seen in Figure 4.

3.2.2 Hilbert Instance Clusters

By sorting our elements along a Hilbert curve and clustering neighboring elements together, we can significantly reduce the final memory consumption of our BVH. However, as found by prior works [45], current GPU ray tracing frameworks exhibit a different peak memory consumption versus final memory consumption when constructing trees. If too many primitives are stored within a single acceleration structure, the scratch memory required to build this tree on the GPU can quickly become prohibitively expensive. Though future GPU tree construction implementations may reduce this peak memory consumption, a workaround to this problem is to divide elements of the unstructured mesh into different bottom-level acceleration structures, then build each of these smaller trees in series, and then instantiate these trees in a final top-level acceleration structure.

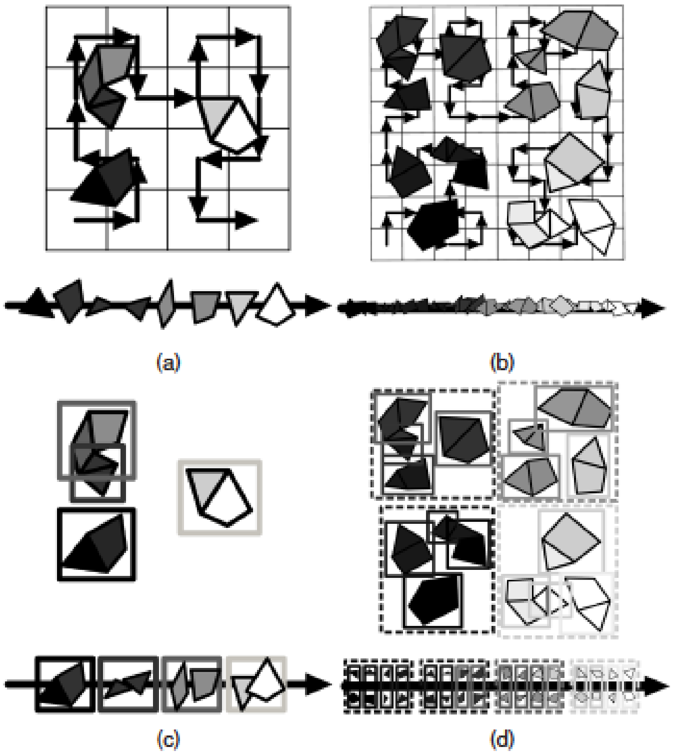
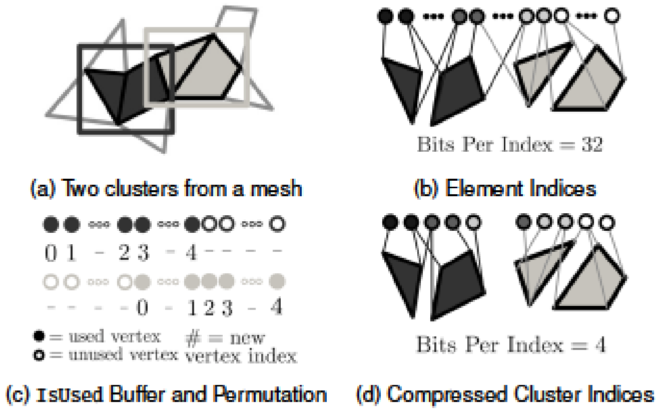


Fig. 5: An illustration of how elements are sorted along the Hilbert space-filling curve (a and b). Once sorted, neighboring elements can be clustered into small leaf clusters (in c, represented with sold lines) to reduce *final* tree memory consumption, and into larger instance clusters (in d, shown with dashed lines) to reduce *peak* tree memory consumption, as well as for mesh index compression.

To do this, prior works [32, 45] propose to divide unstructured elements in memory order into groups of one million elements each; however, as discussed before, clusters of neighboring elements with an unpredictable ordering in space relative to memory will likely generate bounds containing many empty spaces overlapping other clusters. This would then degrade instance culling performance during traversal. Therefore, we propose slightly modifying this pre-splitting strategy, where we reuse our previously Hilbert-curve-reordered elements to generate higher-quality clusters of one million elements each. From now on, we refer to this second type of cluster as an *Instance Cluster*. Like these prior works, this pre-splitting strategy reduces peak memory consumption to acceptable levels. However, with our approach, these bottom-level acceleration structures will have less overlap when instantiated in the top-level acceleration structure than the overlap of prior works. This should improve the performance of our point location queries to be similar to if we were using one unified tree, though we primarily do this pre-splitting out of necessity. An illustration of this instance clustering process—as well as the prior leaf clustering process—is shown in Figure 5.

3.3 Mesh Compression through Re-Indexing

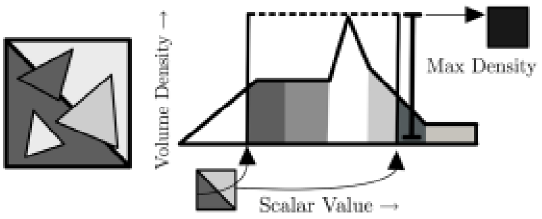
At this point, we have a relatively memory-efficient and accessible method for querying our unstructured elements. Still, prior works [50] go on to compress the unstructured mesh itself. Element indices consume a significant amount of memory and make up the majority of the footprint of our unstructured mesh. Currently, we use 32-bit integers to represent our indices, which can address up to 2^{32} vertices. If we in-



(a) Two clusters from a mesh (b) Element Indices

(c) `IsUsed` Buffer and Permutation (d) Compressed Cluster Indices

Fig. 6: An illustration of our parallel mesh reindexing to reduce bits per element index. In a), subsets of the mesh are clustered along a Hilbert curve. b) The indices of these elements reference a global vertex array and consume 32-bits per index. c) For each cluster, we mark used vertices in parallel, compacting away any unused vertices. We also create a permutation table for each cluster, mapping old vertex addresses to new ones. d) This results in smaller mesh clusters, whose bits-per-index can be reduced due to the smaller per-cluster vertex lists.



(a) Scalar Range (b) Linear Max Density Search

Fig. 7: a) Clusters store the range of scalar field values of the contained primitives. These scalar values are mapped into a color and corresponding density through a transfer function. b) During transfer function edits, we perform a linear search over this range to compute maximum extinction values per cluster for adaptive sampling.

spect the contents of these indices, we will find a considerable amount of redundancy in the most significant bits. This is because more significant bits naturally change less frequently than less significant bits when storing ascending addresses. Fortunately, we can leverage this redundancy to cut our effective mesh memory footprint in half.

To implement index compression, we can split the mesh into meshlets such that each meshlet references fewer than 2^{16} vertices. Then, we can replace the global list of vertices with smaller, per-meshlet vertex lists. As a consequence of splitting our mesh into meshlets, vertices from the global vertex list will be duplicated when referenced by more than one meshlet. Ideally, these meshlets would have a small outer surface area relative to their volume to minimize vertex replication throughout this process.

Wald et al. [50] address this issue by constructing a tree over the unstructured elements from the top down. During construction, tree nodes are split by a surface area heuristic until they reference fewer than 2^{16} vertices; however, when meshes contain billions of elements, reordering primitives at each node split is slow and expensive. So instead, we propose to transform the instance clusters previously generated in Section 3.2.2 to reduce peak memory consumption during hardware BVH construction to also serve as our meshlets for index compression. Previously, these instance clusters contained one million elements each. We reduce the size of these instance clusters to approximately $30,000-60,000$ elements, reducing the number of vertices referenced per instance cluster. Then, we make instance clusters referencing more than 2^{16} vertices use 32-bit indices, while instance clusters with fewer than 2^{16} vertices use 16-bit indices.

Given an instance cluster of elements, we need a way to efficiently compute that cluster's vertex list and the new indices that reference this list. We follow a prior mesh re-indexing method [47] that re-indexes meshes in parallel on the GPU. We propose slightly modifying this parallel re-indexing method to improve re-indexing performance. More specifically, we remove the requirement that duplicate vertices be eliminated. Removing duplicate vertices requires sorting all vertices in the global list for each instance cluster. Since we have potentially thousands of instance clusters, these sorts quickly become prohibitively expensive to do during application startup.

Instead, we assume the unstructured mesh does not contain duplicate vertices, and if it does, we preserve those duplicates to avoid sorting. We substitute this sort with a parallel flagged device selection operation to collect all vertices marked as used into a new subset. This selection process can be implemented in parallel on the GPU through an inclusive prefix sum (minus one) over an array of "`isUsed`" flags for each vertex; however, we recommend using the flagged device selection operation provided by NVIDIA's CUB library for simplicity and improved performance.

An illustration of this re-indexing process is shown in Figure 6. This data transformation is much faster than prior works but can take several minutes depending on the number of instance clusters that need re-indexing; so for large datasets, we recommend minimizing the number of instance clusters required for index compression.

3.4 Adaptive Sampling through Delta Tracking

Now that our data fits within system resources, we will re-evaluate our rendering approach. Adequately sampling these datasets with a naive ray marcher requires an unacceptably high number of samples to sufficiently sample the small cells of interest. To address this issue, prior works [43,44,53] adapt the sampling rate to the underlying data by first gathering maximum density estimates along the ray, which are stored in clusters of elements. Unfortunately, unstructured elements make these algorithms difficult to implement, as these methods require that adaptive sampling clusters do not overlap. Therefore, we propose two ways to efficiently transform our Hilbert clusters into non-overlapping spatial partitionings for use with these adaptive sampling methods.

3.4.1 Hilbert Adaptive Sampling Clusters

We introduce a third cluster type, *Adaptive Sampling Clusters*, where we again use our Hilbert-ordered elements to quickly generate a set of clusters with an equal number of primitives each. Additionally, for each cluster, we now store the minimum and maximum scalar data value within that cluster. Once these clusters are identified, we build a hardware-accelerated BVH over the cluster bounds to facilitate traversal.

Once we have a set of boxes with local estimates for minimum and maximum data values, before rendering, we need to transform these minima and maxima into a set of maximum density estimates considering the currently applied transfer function. Our approach is similar to prior works [31], but we focus on computing only maximum density rather than data variance. For every cluster, we use the per-cluster minimum and maximum data values to iterate over the range of the transfer function density values that affect the contents of the current cluster. We compute the maximum extinction value during this iteration and store this as our current maximum density estimate. (See Figure 7 for an illustration of this process.) We then calculate the maximum density for each cluster in parallel on the GPU whenever the transfer function is edited.

3.4.2 Option I: Cluster Rasterization into Ray-Centric Bins

To adaptively sample our large unstructured data sets, we need an efficient way to transform our overlapping clusters into non-overlapping maximum extinction segments along our ray (see also the pseudocode in our supplemental material). We begin by intersecting our ray with the bounding box of the entire volume, calculating the enter and exit distances along the ray. Next, we divide the interval defined between this enter and exit distance into a constant number of disjoint, equidistant segments. Then, we allocate a bin to store the maximum extinction value along each segment, initializing each bin to a value of 0 (i.e.,

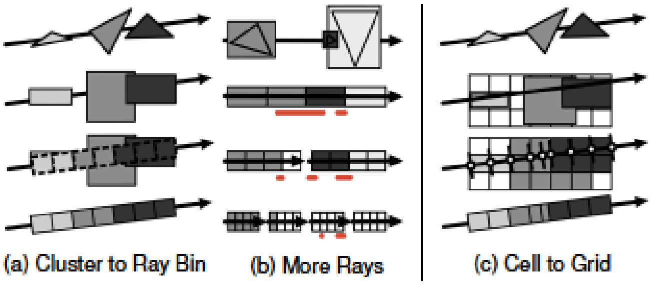


Fig. 8: In a) clusters (top middle) are generated from mesh elements (top), then maximum densities of intersected clusters are rasterized (bottom middle) into bins along the ray (bottom). In b), for highly non-uniform datasets (top), we break up our ray (bottom three rows) to increase bin counts and reduce majorant error (in red). Alternatively, in c) clusters (top middle) are generated from mesh elements (top) like before, but now we rasterize them into a grid of macrocells, which we can traverse using 3D-DDA (bottom middle), producing non-even, but also non-overlapping adaptive sampling intervals along our ray (bottom).

fully transparent).

Once the segment bins are allocated and initialized, we trace a second ray through the previously obtained adaptive sampling clusters. For each cluster we intersect, we "rasterize" the maximum extinction associated with the intersected cluster into the bins of the overlapping ray segments. This rasterization process occurs as clusters are "splat" into 1D segments along the 3D ray, which is slightly different from classical rasterization where triangles are "splat" onto pixels on a 2D screen. This process effectively transforms our object-space partitioning into spatial partitioning, though our maximum extinction estimation becomes more approximate at the compromise. See Figure 8 for an illustration of this process.

In practice, we allocate these bins using ray payload registers to avoid unnecessary VRAM traffic during traversal and cluster rasterization. At the time of writing, current GPU architectures are limited to at most 32 ray payload registers. To further improve precision, we break up this long ray into M shorter ray segments, with N bins per ray where $N <= 32$, and where M is a user-chosen parameter depending on the dataset being rendered and the precision required. Two rays give an effective resolution of 64 bins along our ray, four rays give 128 bins, and so on. (See also Subfigure 8b.) Then, we bounce back and forth between binning clusters and the Delta Tracking process. If our ray collides with a particle inside the volume, we can terminate traversal early without necessarily tracing all of our adaptive sampling rays.

3.4.3 Option II: Cluster Rasterization into a Macrocell Grid

We also explore a second method that transforms our object-oriented clusters into a spatial partitioning via a macrocell grid (see also the pseudocode in our supplemental material). At the cost of some memory, we can achieve approximately the same resolution as our ray binning method by allocating an auxiliary regular grid whose side length is $numBins/\sqrt{3}$. We make each voxel within this grid contain a minimum and maximum scalar value, which we initialize to `FLOAT_MAX` and `FLOAT_MIN`, respectively.

Then, for each adaptive sampling cluster in parallel, we use GPU atomics to rasterize each cluster's minimum and maximum scalar values into all cells of our regular grid that the cluster intersects. This region of interest can be found by rounding up and down each cluster's upper and lower bounds to the nearest regular grid cells. Once this is done, as we edit our transfer function, we compute the maximum density values for all of the regular grid bins in parallel using the same approach described in Section 3.4.1. One exception is that we assign a maximum density of 0 (i.e., fully transparent) to all regular grid bins whose scalar value ranges are uninitialized, i.e., `FLOAT_MAX` and `FLOAT_MIN`, meaning no cluster touches that bin.

Finally, during rendering, we can traverse through these non-overlapping grid cells using a 3D Digital Differential Analyzer (DDA) algorithm [4]. This algorithm draws a line through the cells of the grid in order along that line. In our case, that line is our volume sampling ray. For each cell returned by DDA, we read the majorant density at that cell and determine where the ray enters and exits that cell to compute our non-overlapping segment for adaptive sampling. (See also Subfigure 8c.) By using DDA to traverse through these cells rather than the GPU's ray-tracing cores, we reserve the use of these cores for polyhedra point location queries.

3.4.4 Adaptive Delta Tracking of Unstructured Data

Now that we have a disjoint set of bins containing maximum extinction estimates, we can use a piecewise constant adaptive delta tracking approach to reduce the number of samples taken from the volume to improve rendering performance. Instead of immediately marching through the volume, we first iterate over the ray segments made by the previous step. We compute a localized enter and exit distance for each bin allocated along the ray for just the current segment. If the maximum extinction value for the segment is 0, the current segment represents empty space and can be skipped altogether. Otherwise, we use the maximum extinction value associated with the current segment to sample free flight distances. Now that we are using accurate maximum extinction estimates, the probability of a null collision is significantly reduced, increasing the sampled step size and reducing the number of rejected samples taken from the volume. If the sampled distance from delta tracking passes the end of the segment, we iterate to the next segment allocated along the ray, resetting the sampled distance to account for the change in the maximum extinction estimate.

We can then use this additional performance to improve the quality of the visualization. After tracing a ray through the unstructured data, adaptive delta tracking will return a single representative free flight distance. For pure emission and absorption rendering modes, we return the color of the volume at that distance. To simulate shadows, we trace a shadow ray originating at this final free flight distance and in the direction of the directional light source. This shadow ray operates similarly to the adaptively sampled primary ray. First, an adaptive sampling ray is traced towards the light to collect maximum extinction estimates into a set of bins allocated along the ray. Then, an adaptive delta tracking routine is employed to determine if the light is visible or occluded depending on if the ray makes it through the entirety of the volume without an absorption event occurring. We then use the results of this shadow ray to shade our primary ray's volumetric sample.

4 Experimental Results

To evaluate our method, we use OptiX 7 [36] to access hardware-accelerated ray tracing functionality in NVIDIA GPUs. All measurements were taken using an NVIDIA RTX8000 GPU and an Intel i9 12900K processor. All images were rendered at a resolution of 1024×1024 up to 4000 samples per pixel, with one sample per frame.

4.1 Datasets

With this hardware, we performed a series of tests on a collection of unstructured data sets of varying sizes (cf. Figure 9):

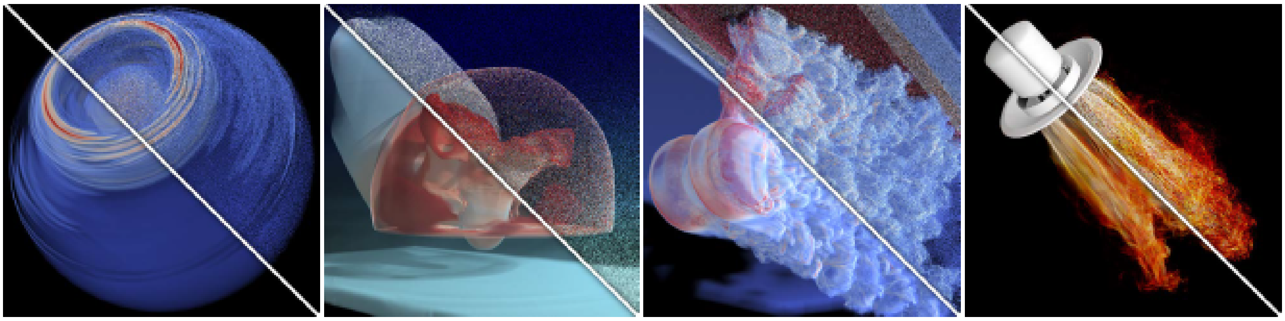
1) The TACC Japan Earthquake simulates the effects of the 2011 earthquake and subsequent tsunami that hit Japan. Scientists resolved the movement of the seismic waves using an adaptive grid; however, our copy of this data set consists of non-adaptive, roughly equally-sized hexahedra.

2) The Deep Ocean Impact is derived from AMR [37] and comes from an xRage [13] simulation of an asteroid impacting the ocean.

3) The NASA Landing Gear is derived from AMR and visualizes the vorticity of air around the landing gear, simulated with NASA's LAVA code [22]. Coarse cells are $2^{12}\times$ larger than the finest cells.

4) The NASA Mars Lander visualizes a retropropulsion study to decelerate the Mars lander entering the martian atmosphere [20]. This data was simulated using NASA's FUN3D code [7] and consists of nearly a billion mixed finite elements that vary significantly in volume.

5) Cell Variance Datasets are synthetic datasets of 3 million cells each, used to evaluate how sensitive our method is to varying cell sizes.



(a) Earthquake
48M Elements

(b) Impact 20K
247M Elements

(c) Landing Gear
291M Elements

(d) Mars Lander
789M Elements

Fig. 9: Data sets used for testing. Impact and Landing Gear are adaptive mesh refinement simulations, while Earthquake and Mars Lander are fully unstructured finite element meshes. All images are rendered at 1024x1024, at 1 spp per frame (top right of each image), with one primary ray and one shadow ray per pixel, and are converged over time to 4000 SPP (bottom left of each image).

Table 1: Memory usage of our test models. Each data type is given before and after instance cluster mesh compression. All datasets are clustered into instance clusters containing 30k-60k primitives in order to keep the referenced vertices within each cluster less than 2^{16} for mesh index compression. We also store 16 elements per leaf cluster to reduce tree memory consumption per instance cluster.

	Pre Compression (1 element / leaf cluster)				Post Compression (16 elements / leaf cluster)			
Model	Vertices	Indices	BVH size	Total	Vertices	Indices	BVH size	Total
Earth	0.59 GB	1.42 GB	0.40 GB	2.41 GB	0.69 GB	0.73 GB	0.08 GB	1.50 GB
Impact	1.97 GB	6.00 GB	2.11 GB	10.1 GB	2.28 GB	3.00 GB	0.46 GB	5.74 GB
Gear	2.93 GB	7.72 GB	6.20 GB	16.9 GB	3.37 GB	3.86 GB	0.45 GB	7.68 GB
Lander	1.62 GB	12.14 GB	16.9 GB	30.6 GB	1.86 GB	6.65 GB	1.20 GB	9.71 GB

We generate a Delaunay tetrahedralization over random points generated through a Poisson sphere sampling method, resulting in evenly-sized tetrahedra. These vertices are then displaced randomly, up to the Poisson sphere radius, to introduce variance in the derived tetrahedra sizes (see the supplemental for an illustration).

4.2 Evaluation

With these datasets, we measure our method's preprocessing time, compression effectiveness, and rendering performance.

4.2.1 Clustering and Compression

Table 1 shows the effectiveness of our Hilbert clusters and mesh re-indexing in reducing memory consumption. As shown, we achieve compression factors between 0.32 and 0.70. We also outperform the compression method by Wald et al. [49] for all but the Mars Lander dataset (ours at 9.71 *GB*, theirs at 9.3 *GB*).

Table 2 shows the evaluation of our Hilbert clusters' performance versus memory consumption. We incrementally cluster more and more elements together, building trees over these different cluster sizes and measuring the impacts on memory consumption of the BVH as well as rendering performance. For the majority of our datasets, clustering neighboring elements results in significant tree memory reductions while still maintaining interactive performance. We did notice a drop in performance on Landing Gear as more and more elements were clustered together at the leaves, which we hypothesized might be a sensitivity to highly varying cell sizes. Indeed, strong variance in cell size degrades performance somewhat, as shown by our synthetic Cell Variance benchmarks, but only by about 6.8%. Therefore, we suspect that the majorant estimate for Landing Gear is poor, resulting in many null collisions, which grow linearly in expense with leaf cluster size.

Table 3 compares time to cluster elements of the Mars Lander Dataset by splitting the mesh into equal-sized clusters using different approaches, including a KD tree, a BVH, and our Hilbert clustering. Top-down KD tree construction on the CPU quickly becomes impractical to compute as cluster sizes decrease, requiring a prohibitive amount of memory. Bottom-up LBVH construction on the CPU does much better but still is relatively impractical for generating small clusters of elements immediately before rendering. Hilbert clusters can be formed with a single sort, independent of cluster count; hence, they can be computed at runtime.

Table 2: A performance versus memory consumption evaluation of our Hilbert clusters for different cluster sizes for two NASA datasets. Baseline signifies the case where elements are not sorted along the curve.

Model		baseline	1	4	8	16	32
Lander	Accel GB	16.9	16.6	4.5	2.3	1.2	0.6
	FPS/RPS	11.5/ 24.1	13.8 / 28.9	11.7 / 24.5	10.0/ 20.9	8.2 / 17.2	5.8/ 12.1
Gear	GB	6.20	6.53	1.51	0.86	0.45	0.23
	FPS/RPS	10.1 / 21.2	15.1 / 31.7	4.8 / 10.1	2.8 / 5.9	1.7 / 3.6	0.78 / 1.6
0% Cell	Accel GB	0.60	0.56	0.15	0.08	0.04	0.02
Variance	FPS/RPS	48.3/101.3	51.4/107.8	46.7/97.9	38.0/79.7	29.8/62.5	20.9/43.8
50% Cell	Accel GB	0.61	0.58	0.16	0.08	0.04	0.02
Variance	FPS/RPS	46.7/97.9	50.2/105.3	45.3/95.0	37.1/77.8	28.9/60.6	20.1/42.2
100% Cell	Accel GB	0.62	0.59	0.16	0.08	0.04	0.02
Variance	FPS/RPS	45.0/94.4	48.5/101.7	43.6/91.4	35.9/75.3	27.8/58.3	19.3/40.5

Table 4 presents statistics related to compressing mesh indices using our parallel mesh re-indexing method for various data sets. We measure how long our parallel mesh re-indexing strategy takes to complete and compare this to how long it takes to load the dataset from the disk and how much memory we save. As shown in the table, prior works needed several hours to re-index on the CPU, while our largest data sets only take a couple of minutes. Though the Landing Gear data set is smaller than the Mars Lander data set, the hexahedra representing the Landing Gear exhibit less vertex reuse than Small Lander, thus requiring more clusters with fewer elements to guarantee each cluster references fewer than 2^{16} vertices.

4.2.2 Visual Results

As shown in Table 5, all datasets improve significantly in performance when adaptive sampling is enabled versus disabled by up to 100× in the case of the Landing Gear dataset. Compared to a traditional alpha-composited ray marcher, our results are significantly more noisy per frame (see the supplemental for what this noise looks like), but as

Table 3: A comparison of time to cluster elements of the Mars Lander Dataset using different approaches (KD tree and LBVH implementations are taken from PBRT v3 [38]).

Clustering Method	KD Tree	LBVH	Hilbert Clusters
100K Elems / Cluster	30+ hrs	16.3 mins	2.5 secs
16 Elems / Cluster	NA	23.5 mins	2.5 secs

Table 4: Time required to compress mesh indices using our parallel mesh reindexing method for various data sets.

Mesh Compression	Earth	Impact	Gear	Lander
Number of Clusters	3K	8K	16K	6K
Seconds to Load	2.72	6.31	14.6	10.9
Seconds to Compress	9.76	71.5	216	57.6
Memory Saved (in GB)	0.73	2.80	3.79	5.94

shown in Figure 9, our adaptive sampled images converge to an unbiased result—unlike adaptive ray marching, as shown in Figure 3—due to the physically-based model that Monte Carlo Delta Tracking follows. Delta tracking also easily extends to volumetric shadows, unlike alpha-composited raymarching, which requires a shadow ray for every sample taken from the volume.

Our results show noticeable performance improvements when transforming our overlapping Hilbert clusters on the fly into non-overlapping bins, despite the overhead of on-the-fly rasterization. However, our results show that DDA traversal over our macrocells performs better than our ray-binning method by 1.4× to 3×. With our ray binning method, we also have resource contention over ray-tracing cores, which is not an issue with the DDA traversal approach.

One exception we found was that, for the Landing Gear dataset, even a 512^3 grid of macrocells was insufficient to approximate maximum density estimates accurately. Compared to even the Mars Lander dataset, the actual Landing Gear model is incredibly small relative to the simulation domain. As a result, our on-the-fly ray binning method can achieve a higher effective adaptive sampling resolution at a fraction of the memory due to the adaptivity of the Hilbert clusters. In this case, on-the-fly ray binning outperforms DDA traversal, but only for highly optically dense colormaps where insufficient adaptive sampling becomes too costly and achieving sufficient resolution with macrocells requires too much memory.

Ultimately, we believe these results suggest that, for our current datasets, it is best to rasterize adaptive sampling clusters into an intentionally small set of macrocells. This may seem counterintuitive, as the unstructured meshes themselves are non-uniform, so in theory, a more adaptive cluster structure would do better at adaptive sampling; however, fewer adaptive sampling segments promote larger Delta Tracking skips forward. As more adaptive sampling segments along the ray are formed, traversing through too many segments introduces overhead that starts to dominate rendering time.

5 Limitations and Future Work

Ultimately we are very excited with these results, as we show that low memory, high-performance unstructured volume rendering is possible using only relatively simple data transformations. Still, our method has some limitations that are worth discussing.

One limitation of our approach is that our preprocessing steps to reduce memory consumption can consume a large amount of memory, as we require the entire data set to fit within GPU memory to be compressed. Therefore, it might still make sense to compress datasets on a workstation GPU with more VRAM, then save these results to disk for post-hoc visualization on a more consumer-friendly card. Alternatively, all these compression processes could be done in parallel on the CPU, which would likely still be much faster than the preprocessing required by prior work. As for future work, we believe there is still room to further reduce memory consumption without introducing significant preprocessing time, perhaps by compressing neighboring tetrahedra pairs.

Table 5: Comparing rendering performance for alpha-composited ray marching, non-adaptive delta tracking, and our adaptive approach using ray bins and DDA. Adaptive methods use 50M Hilbert clusters. Measurements are in frames per second / million rays per second. Note, our ray marcher does not include volumetric shadows, as shadows cause the marcher to not return in a reasonable time.

Model	Marching*	Tracking	Ray Bins	DDA	DDA
			2K bins	100^3	512^3
Earth	3.89* / 4.1	3.87 / 8.1	7.53 / 15.8	22.4 / 47.0	13.9 / 29.2
Impact	1.13* / 1.2	5.05 / 10.6	7.62 / 16.0	11.8 / 24.7	6.87 / 14.4
Gear	0.75* / 0.8	0.10 / 0.2	7.26 / 16.0	4.85 / 10.2	9.72 / 20.4
Lander	0.58* / 0.6	2.25 / 7.7	11.8 / 24.7	16.3 / 34.2	14.1 / 30.0

In terms of rendering, as we discussed before, our ray binning method is limited in the number of clusters that rays can traverse through, and it suffers from resource contention with the ray-tracing cores also used for point location. On the other hand, our cluster-generated macrocells quickly consume too much memory, as a doubling in spatial resolution results in a cubic number of allocated macrocells. To address this, we believe it would be worth exploring alternative data transformations that transform clusters into a hierarchy of macrocells, perhaps using the number of clusters that fall within a coarse macrocell to guide subdividing that macrocell into finer macrocells. Then, a hierarchical DDA process could be used instead of a traditional DDA. Orthogonally, we might also be able to improve rendering performance by attempting to execute multiple subsequent point containment queries within one RT core trace call by leveraging the multiple clustered elements contained within a single leaf.

Our work focuses on single scattering effects in terms of rendering quality, as single scattering along a common light direction is relatively efficient while still conveying depth information. Our approach should naturally extend to support multiple scattering effects, but how our ray-binning strategy performs, in this case, warrants further evaluation.

As our method uses a Monte Carlo approach, individual frames produce noise that must be converged over time. To reduce noise, we could benefit from a blue noise error distribution, as shown by Wolfe et al. [52]. We would also likely benefit from a neural network denoiser similar to that proposed by Hofmann et al. [18]. Caching light visibility in finite elements could also be an interesting avenue to explore, similar to how prior regular grid shadow maps work. Finally, we believe it would be interesting to explore how our approach might be used in a distributed setting, e.g., for use in in-situ rendering.

6 Conclusion

This work presented a set of simple, efficient, yet highly effective data transformations to reduce the memory footprint of large unstructured grids for rendering. To reduce implementation complexity, we substitute complex bounding volume hierarchy construction for a simple sort along a Hilbert space-filling curve. Then, we use clusters generated from this curve to compress the size of point location trees and mesh indices, all in under a couple of minutes.

We then presented two different methods that transform these clusters into a structure suitable for adaptive sampling during volumetric path tracing: one that rasterizes clusters on the fly into non-overlapping bins along a ray, and another that rasterizes clusters into macrocells which can be traversed during runtime using DDA. These approaches dramatically improve rendering performance, opening the door to higher quality volumetric renderings, including volumetric shadows. Using a Monte Carlo path tracing approach over alpha-compositing, we can amortize sampling expenses over time to improve interactivity and final image quality.

Ultimately, we hope for the high-performance data transformations in this work to improve the accessibility of these powerful unstructured grid formats, which we believe could enable new possibilities for large data exploration and high-quality visualization.

References

[1] Advanced Micro Devices, Inc. RDNA 2 Instruction Set Architecture: Reference-Guide. Technical report, Advanced Micro Devices, Inc., 2020. Available at https://developer.amd.com/wp-content/resources/RDNA2_Shader_ISA_November2020.pdf, Accessed 27 March 2022.

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

[3] A. Aman, S. Demirci, U. Güdükbay, and I. Wald. Multi-level tetrahedralization-based accelerator for ray-tracing animated scenes. *Computer Animation and Virtual Worlds*, 32(3-4):e2024, 11 pages, 2021.

[4] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *Proceedings of Eurographics*, EG '87, pp. 3–10, 1987.

[5] M. Ament, F. Sadlo, C. Dachsbacher, and D. Weiskopf. Low-pass filtered volumetric shadows. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2437–2446, 2014.

[6] C. Benthin, I. Wald, S. Woop, and A. T. Áfra. Compressed-leaf bounding volume hierarchies. In *Proceedings of the Conference on High-Performance Graphics*, 2018. Article no. 6, 4 pages.

[7] R. T. Biedron, J.-R. Carlson, J. M. Derlaga, P. A. Gnoffo, D. P. Hammond, W. T. Jones, B. Kleb, E. M. Lee-Rausch, E. J. Nielsen, M. A. Park, et al. *FUN3D Manual: 13.6*. National Aeronautics and Space Administration, Langley Research Center, 2019.

[8] A. R. Butz. Alternative algorithm for Hilbert's space-filling curve. *IEEE Transactions on Computers*, 100(4):424–426, 1971.

[9] H. Dammertz, J. Hanika, and A. Keller. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. *Computer Graphics Forum*, 27(4):1225–1233, 2008.

[10] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf. Real-time volume graphics. In *ACM SIGGRAPH Course Notes, # 29*. ACM, New York, NY, 2004.

[11] M. Ernst and G. Greiner. Multi bounding volume hierarchies. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, RT 08, pp. 35–40, 2008.

[12] D. Ganter and M. Manzke. An analysis of region clustered BVH volume rendering on GPU. *Computer Graphics Forum*, 38(8):13–21, 2019.

[13] M. Gittings, R. Weaver, M. Clover, T. Betlach, N. Byrne, R. Coker, E. Dendy, R. Hueckstaedt, K. New, W. R. Oakes, et al. The RAGE radiation-hydrodynamic code. *Computational Science & Discovery*, 1(1):015005, 63 pages, 2008.

[14] P. Gralka, I. Wald, S. Geringer, G. Reina, and T. Ertl. Spatial partitioning strategies for memory-efficient ray tracing of particles. In *Proceedings of the IEEE 10th Symposium on Large Data Analysis and Visualization*, LDAV 20, pp. 42–52. IEEE, 2020.

[15] H. Gruen. A quick guide to Intel's ray-tracing hardware. In *Game Developers Conference*, 2022. Available at https://www.intel.com/content/www/us/en/events/developer/gdc-march-2022.html, Accessed: 27 March 2022.

[16] T. Günther, A. Kuhn, and H. Theisel. MCFTLE: Monte Carlo rendering of finite-time Lyapunov exponent fields. *Computer Graphics Forum*, 35(3):381–390, 2016.

[17] L. Ha, J. Krüger, and C. T. Silva. Fast four-way parallel radix sorting on GPUs. *Computer Graphics Forum*, 28(8):2368–2378, 2009.

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

[19] 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, 2019.

[20] K. E. Jones. Summit supercomputer simulates how humans will 'brake' during Mars landing, October 2019. Oak Ridge National Laboratory, Available at https://www.ornl.gov/news/summit-simulates-how-humans-will-brake-during-mars-landing, Accessed: 27 March 2021.

[21] R. Kähler, J. Wise, T. Abel, and H.-C. Hege. GPU-assisted raycasting for cosmological adaptive mesh refinement simulations. In *Proceedings of Volume Graphics*, pp. 103–110, 2006.

[22] C. C. Kiris, M. F. Barad, J. A. Housman, E. Sozer, C. Brehm, and S. Moini-Yekta. The LAVA computational fluid dynamics solver. In *American Institute of Aeronautics and Astronautics (AIAA) SciTech Forum, 52nd Aerospace Sciences Meeting*, pp. 1–43, 2014.

[23] T. Kroes, F. H. Post, and C. P. Botha. Exposure render: An interactive photo-realistic volume rendering framework. *PloS One*, 7(7):e38586, 10 pages, 2012.

[24] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pp. 451–458, 1994.

[25] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[26] P. Ljung. Adaptive sampling in single pass, GPU-based raycasting of multiresolution volumes. In *Proceedings of Volume Graphics*, pp. 39–46. The Eurographics Association, 2006.

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

[28] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.

[29] D. Moore. Fast Hilbert curve generation, sorting, and range queries. Technical report, Rice University, Computational and Applied Mathematics, 2008.

[30] P. Moran and D. Ellsworth. Visualization of AMR data with multi-level dual-mesh interpolation. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):1862–1871, 2011.

[31] 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. IEEE, 2019.

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

[33] J. S. Mueller-Roemer and A. Stork. Gpu-based polynomial finite element matrix assembly for simplex meshes. *Computer Graphics Forum*, 37(7):443–454, 2018.

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

[35] NVIDIA Corp. NVIDIA Ampere GA102 GPU Architecture: Second-Generation RTX. Technical report, NVIDIA, 2021. Available at https://images.nvidia.com/aem-dam/en-zz/Solutions/geforce/ampere/pdf/NVIDIA-ampere-GA102-GPU-Architecture-Whitepaper-V1.pdf, Accessed: 27 March 2021.

[36] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, et al. Optix: a general purpose ray tracing engine. *ACM Transactions on Graphics*, 29(4):1–13, 2010.

[37] J. M. Patchett, F. J. Samsel, K. C. Tsai, G. R. Gisler, D. H. Rogers, G. D. Abram, and T. L. Turton. Visualization and analysis of threats from asteroid ocean impacts. Technical report, Los Alamos National Laboratory, 2016.

[38] M. Pharr, W. Jakob, and G. Humphreys. *Physically Based Rendering: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd ed., 2016.

[39] 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, 2015.

[40] 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 IEEE Visualization Conference*, VIS 21, pp. 91–95, 2021.

[41] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *ACM Computer Graphics (Proceedings of SIGGRAPH '90)*, 24(5):6370, 1990.

[42] D. Ströter, J. Mueller-Roemer, A. Stork, and D. Fellner. OLBVH: Octree Linear Bounding Volume Hierarchy for Volumetric Meshes. *The Visual Computer*, 36(10-12):2327–2340, 2020.

[43] L. Szirmay-Kalos, B. Tóth, and M. Magdics. Free path sampling in high resolution inhomogeneous participating media. *Computer Graphics Forum*, 30(1):85–97, 2011.

[44] L. Szirmay-Kalos, B. Tóth, M. Magdics, and B. Csébfalvi. Efficient free path sampling in inhomogeneous media. In *Eurographics (Posters)*, 2010.

[45] I. Wald. Computing minima and maxima of subarrays. In E. Haines and T. Akenine-Möller, eds., *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, pp. 61–70. Apress, Berkeley, CA, 2019.

[46] I. Wald. A simple, general, and GPU friendly method for computing dual mesh and iso-surfaces of adaptive mesh refinement (AMR) data. *arXiv preprint arXiv:2004.08475*, 2020.

[47] I. Wald. GPGPU-parallel re-indexing of triangle meshes with duplicate-vertex and unused-vertex removal. *arXiv preprint arXiv:2109.09812*, 2021.

[48] I. Wald, C. Benthin, and S. Boulos. Getting rid of packets-efficient SIMD single-ray traversal using multi-branching BVHs. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, pp. 49–57, 2008.

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

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

[51] G. H. Weber, H. Childs, and J. S. Meredith. Efficient parallel extraction of crack-free isosurfaces from adaptive mesh refinement (AMR) data. In *IEEE Symposium on Large Data Analysis and Visualization*, LDAV 12, pp. 31–38. IEEE, 2012.

[52] A. Wolfe, N. Morrical, T. Akenine-Möller, and R. Ramamoorthi. Scalar spatiotemporal blue noise masks. *arXiv preprint arXiv:2112.09629*, 2021.

[53] Y. Yue, K. Iwasaki, B.-Y. Chen, Y. Dobashi, and T. Nishita. Unbiased, adaptive stochastic sampling for rendering inhomogeneous participating media. *ACM Transactions on Graphics*, 29(6):177, 8 pages, 2010.