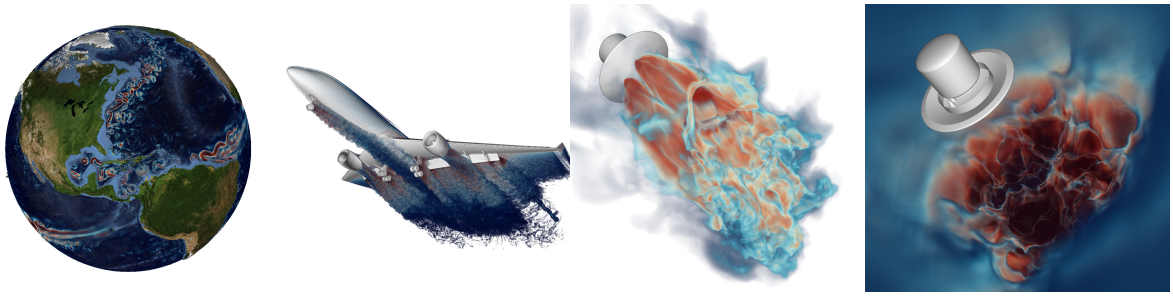
This is the authors' version of the article that has been accepted at IEEE Vis 2021, to eventually be published in IEEE TVCG.

A Memory Efficient Encoding for Ray Tracing Large Unstructured Data

Ingo Wald Nate Morrical Stefan Zellmann



DKRZ full ocean
485 M verts, 749 M elements
compression rate 5.9 : 1

NASA Exa-Jet
656 M verts, 652 M elements
compression rate 4.9 : 1

NASA Mars Lander (small)
143 M verts, 789 M elements
compression rate 14.0 : 1

NASA Mars Lander (large)
576 M verts, 2.9 B elements
compression rate 12.3 : 1

Figure 1: Compression rates we achieve for four very large unstructured-mesh data sets: *Ocean*, *ExaJet*, and two resolutions of the *Fun3D* Mars Lander Retropulsion Study. Quoted memory consumption includes, for both compressed and uncompressed versions, both the unstructured mesh elements and the acceleration data structure that allows for fast random-access sampling. Our representation encodes the same information as an uncompressed reference implementation, but at up to $14\times$ less memory.

Abstract

In theory, efficient and high-quality rendering of unstructured data should greatly benefit from modern GPUs, but in practice, GPUs are often limited by the large amount of memory that large meshes require for element representation and for sample reconstruction acceleration structures. We describe a memory-optimized encoding for large unstructured meshes that efficiently encodes both the unstructured mesh and corresponding sample reconstruction acceleration structure, while still allowing for fast random-access sampling as required for rendering. We demonstrate that for large data our encoding allows for rendering even the 2.9 billion element Mars Lander on a single off-the-shelf GPU–and the largest 6.3 billion version on a pair of such GPUs.

1 Introduction

Our computational capabilities are rapidly evolving. Year over year, supercomputing power improves by about 1.5 to $2\times$ [29]. As we improve our ability to simulate the world around us, our simulations naturally grow larger to match these increased computational budgets. Take—for example—the NASA Mars Lander Study [1] shown in Figure 1. The largest unstructured mesh used for this study consists of 1.14 billion vertices and more than 6.3 billion finite elements per time step, and many different time steps thereof.

What not long ago used to be simple structured volumes have become complex, largely unstructured data sets. These data sets commonly come in one of two predominant formats: semi-structured grid data, and unstructured finite elements. Semi-structured data sets like adaptive mesh refinement (AMR) data consist of a set of bricks or trees containing grids of voxels at varying resolutions. Unstructured finite elements on the other hand consist of a mix of tetrahedra, pyramids, wedges, and hexahedra that can twist and bend to more effectively adapt to the computational domain.

Today, both AMR and unstructured meshes seem equally important, with some applications preferring the one, and vice versa. However, from the standpoint of *visualizing* the computed data, the difference between AMR and unstructured meshes can be quite pronounced. AMR data can come in many different forms and thus require many different codes to handle. Conversely, unstructured mesh representations are relatively standardized and thus easy to support across many different tools. Unstructured meshes are also arguably more general, in that AMR representations can always be converted into an unstructured mesh (by computing their dual mesh), but not vice versa. Consequently, any advances in quality or performance of rendering unstructured meshes should benefit both unstructured and AMR codes.

For the remainder of this paper, we assume that truly high-quality rendering of unstructured data involves sample-based volume ray marching (with or without volumetric scattering), combined with surface based rendering for embedded geometry. Within that context, large unstructured data sets cause two apparent issues:

1. Unstructured meshes by design have little implicit structure, meaning that reconstructing samples requires expensive cell location kernels with often complex and incoherent memory accesses, pointer chasing, and code divergence.

2. The situations where unstructured codes are most useful are those where the simulation needs to adapt to high-frequency features in the computed function. Consequently AMR and unstructured data often suffer from large differences in the size of the features of interest relative to the computational domain. This, in turn, requires advanced sampling strategies or a large number of samples to resolve features of interest during visualization.

The resulting high cost for rendering such models would suggest the use of modern GPUs. This, however, is further complicated by another, less obvious problem—memory. Since unstructured meshes have to store *both* scalar values and mesh topology, their storage cost per scalar value is often much higher than that of more structured or semi-structured formats. For example, for the Mars Lander data set shown in Figure 1, the 576 million scalar field values require an additional 576 million vertices to represent the positions of these scalars, and yet another 2.9 billion cells for the connectivity, for a total of

roughly 30× as much memory for vertices and indices as for scalar values. Even worse, to perform the sample reconstruction required for sample-based volume rendering, a corresponding acceleration structure built over these elements must also be stored, introducing further memory requirements that complicate their ability to benefit from GPUs. Thus, we end up in a situation where unstructured mesh visualization and processing should in theory be a prime candidate for GPU acceleration, yet we often cannot fit these data sets into GPU memory because of their high memory footprint.

In this paper, we look at how to store unstructured meshes in a more memory-efficient way. In particular, we focus on a strategy to reduce the memory footprint of the acceleration data structures required for high-quality, sample-based ray tracing. We do this by analyzing where a state-of-the-art data structure that was optimized for CPUs spends most of its memory. Step by step, we adopt strategies that reduce this memory overhead while still maintaining what is essentially the same implicit structure. We do so with the explicit goal of creating an encoding that is so compact that even some of the largest unstructured data sets—including everything required for random-access sampling—can be fit onto a single high-end GPU.

We observe that reducing this memory consumption is entirely orthogonal to the question of *where* to place samples during ray marching. Therefore, we leave a discussion of space skipping or adaptive sampling to another paper, and in this work focus exclusively on the problem of memory consumption and on the influence that the proposed technique has on raw sample throughput.

2 Related Work

Rendering of large unstructured meshes has posed a challenge to visualization researchers for some time, and a large body of work has set out to tackle the various challenges involved. Prior work has focused on rendering performance, memory consumption, and compression strategies, either independently or together in a holistic approach. Our work addresses challenges specific to memory consumption; however, we review relevant work across all these categories to provide more context to the challenges involved in rendering these data sets.

2.1 Unstructured Volume Rendering

Some prior works has focused on the challenges involved in rendering unstructured data. Early work looks at either splatting the unstructured elements into the frame buffer [36, 27] or on marching view-aligned rays from element face to element face [19, 20]. A still excellent survey of early GPU-accelerated techniques can be found in Silva et al. [28]. Today, high-quality volume rendering (with or without unstructured elements) typically relies on some form of volume ray marching as originally proposed by Drebin et al. [7], which for unstructured meshes requires some form of cell location to find—and then, interpolate within—the elements that a given sample is in.

OSPRay [33], a widely used open-source framework for scientific visualization, performs volume rendering of unstructured meshes using the approach presented by Rathke et al. [24]. In the method described by Rathke, a series of point queries are taken per view-aligned ray in a volumetric ray marcher. These point queries require traversal of a bounding volume hierarchy-based acceleration structure, in combination with several point-in-element tests.

The performance of these point queries is a critical component in the performance of a volumetric ray marcher. Garth and Joy [11] proposed the celltree, an optimized and memory-efficient data structure to perform point queries that is based on bounding interval hierarchies. Recent work by Wald et al. [34] and by Morrical et al. [17] leverages the ray traversal units found in modern GPUs to accelerate and optimize these point queries proposed by Rathke. While these two papers only looked at tetrahedral meshes, more recent work by Morrical et al. has also looked at extending this same hardware-accelerated concept to more general unstructured meshes consisting of mixed tetrahedra, pyramid, wedge, and hexahedra element types [18]. This latter paper in particular can handle all the *types* of model used in this paper, but requires too much memory to render our larger data sets due to their additional triangles structure.

2.2 Acceleration Structure Compression

Bounding volume hierarchies (BVHs) [25] have become the de facto standard for interactive ray tracing. When using a naïve encoding of a BVH, the overall memory limiting factor will—for both unstructured mesh and surface rendering—typically be the BVH structure itself. Prior work has sought to reduce the size of the acceleration structure by reducing the number of internal nodes. One such way is to use a BVH with a wide branching factor, as demonstrated by Dammertz [5], Ernst and Greiner [8], and Wald [30]. Additionally, as shown by Benthin et al. [2], wide BVHs can be further compressed by quantizing child node bounds relative to their parent's bounds using a fixed point encoding. By constraining the child node bounds from 32-bit floating point values to a small set of finite values, these nodes can be represented using a smaller integer type to compress them. Ylitie et al. [37] used a similar BVH compression scheme and wide BVH, with the goal of reducing traversal memory traffic when tracing incoherent rays.

2.3 Mesh Compression

Beyond compressing just the acceleration structure, prior work has explored strategies for compressing the mesh data itself. Generally speaking, unstructured meshes are represented using a list of vertices followed by potentially multiple lists of primitive indices that connect these vertices together to form the mesh primitives. In elements comprised of multiple primitive types (e.g., tetrahedra, wedges, and hexahedra), primitive indices can connect a variable number of vertices together depending on their type.

A common, though lossy, approach to compressing unstructured meshes is to quantize the vertices [26]. However, for unstructured meshes the amount of memory required for vertices is typically small compared to vertex indices and BVH nodes, so any savings in the vertex positions tends to be limited. Consequently, compression of meshes typically focusses on compressing the primitives' vertex indices and not on vertex positions. An orthogonal approach to ours is to use sequential-range encoding as proposed by Fellegara et al. [9, 10].

Mesh compression and simplification techniques are also used for surfaces [14] and often employ adaptive tessellation and multi-level approaches such as proposed by Cignoni et al. [4].

More relevant to our work are progressive multi-resolution mesh compression techniques. An advantage of these techniques is that meshes can be progressively decoded and visualized, possibly at successive levels of detail. Pajarola et al. [21] proposed collapsing and decollapsing tetrahedral edges, and Danovaro et al. [6] suggested incrementally subdividing a base tetrahedral mesh; Castro et al. [3] suggested a wavelet-based decompression scheme for decoding tetrahedral meshes; and Peyrot et al. [23] suggested a multi-resolution technique that supports efficient encoding of hexahedral meshes. Although these approaches allow for fine control over the level of detail, as meshes grow larger the decoding process can become prohibitively expensive. In particular, any ray marching-based approach to rendering unstructured meshes will require efficient decoding *per sample*, limiting what kind of encoding can be done.

3 Mesh and BVH Encoding for a Reference Uncompressed Mesh and BVH

Our work aims to improve state of the art in memory efficient encoding of unstructured mesh data for rendering using a sample-based ray caster. First, it is worth noting that there exist other methods than random-access sampling that have been proven to be effective at

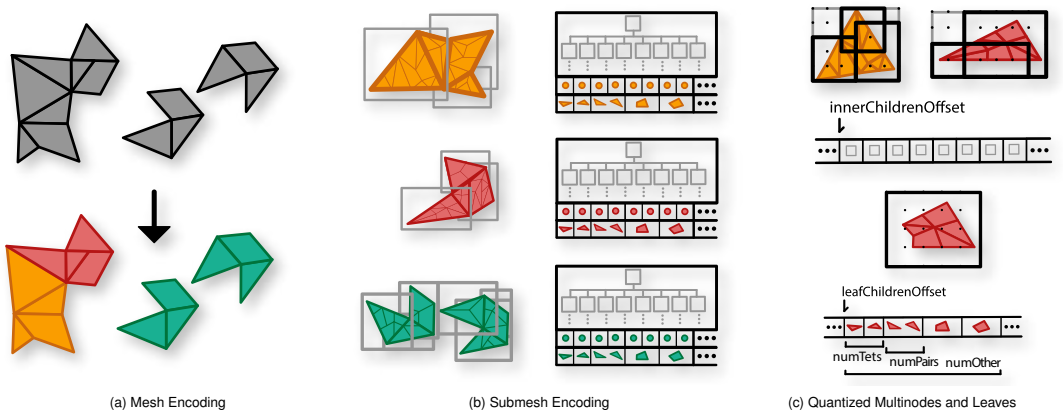


Figure 2: Illustration of our method. In (a), the input mesh is split into several submeshes as described in Section 4.2. In (b), each submesh contains a multi-branching BVH for sample reconstruction (See Section 5.2), as well as a list of vertices and elements (tetrahedra through four indices, and higher dimensional elements using eight). In (c), node bounds are quantized to reduce the memory footprint of each node. Item lists are replaced by offsets into a common per-submesh list of either multi-nodes or primitives (See Section 5.4).

rendering large-scale unstructured meshes. For the purposes of this work, we consider these alternative techniques orthogonal to the one that we use. The choice of mesh traversal technique ultimately has a strong influence on the design choices we make in the following sections. We discuss these alternative techniques in Section 7.

To better understand the design choices that need to be taken when implementing random-access sampling for large-scale unstructured meshes, we first investigate the memory layout used by what we consider a good reference implementation: OSPRay [33]. Before we go into this analysis though, we want to recognize two important caveats: First, that design choices in OSPRay were made under the assumption that memory pressure is less severe for CPUs; that OSPRay could adopt a more efficient BVH encoding like ours too; and that OSPRay can do many tasks that our implementation cannot. Second, that OSPRay's choice of BVH and mesh encoding is by no means a wasteful outlier, but is instead representative of what any non-compressed method would use; in fact, an almost identical encoding for BVH and/or unstructured meshes was also used—including by some of this paper's authors—for ray tracing dynamic geometry [31], for iso-surface ray tracing by Rathke et al. [24] and by Wald et al. [32], and recently by Morrical et al. [18] for GPU tet-mesh point location.

As such, we emphasize that this paper is not intended to be a head-to-head comparison to OSPRay specifically, but rather, a step towards exploring just how much memory could be saved relative to a typical non-compressed encoding for sample-based rendering. Arguably, memory is of major concern only on certain architectures such as for example GPUs. In that context, we see OSPRay as merely the most easily accessible "proxy" for what any other (uncompressed) state of the art solution would likely spend its memory on.

As a specific data set to do this analysis with, we chose the "medium" version of the NASA Mars Lander. In total, for this model of 577 million vertices and 2.9 billion elements the total memory for unstructured mesh and BVH (in OSPRay's chosen memory layout) sum up to approximately 333 GB.

3.1 Mesh Data

The input unstructured data set consists of vertex positions, scalar field values, and vertex indices for the unstructured mesh elements (which comprise mostly of tetrahedra, but also several million pyramids, wedges, and hexahedra). Vertex and scalar data for the Lander comes in double-precision floats, but in OSPRay (as well as others) is stored in single precision floats. Each vertex stores floats: three for the position, and the fourth for the scalar value. For a total of 576 million vertices, this costs 8.6 GB.

Unstructured mesh elements in the input are stored as arrays of 64-bit indices, with separate arrays for tetrahedra, pyramids, wedges, and hexahedra, using either 4, 5, 6, or 8 such 64-bit ints, respectively. To store all elements in a single array, OSPRay instead stores each element as a record of eight 32-bit integers, with the first element of each such record encoding the type of element: for a hex, all eight indices are non-negative; for tetrahedra, pyramids, and wedges the first index is a negative number encoding the type of element, and the last 4, 5, or 6 indices, respectively, encode that element's vertex indices. In this single array format, each element consumes exactly $8 \times 4 = 32$ bytes, for a total of 96 GB for the Mars Lander. We observe that this alone is already roughly $11\times$ the memory stored for the vertices (and $44\times$ that for actual scalar field data), and already as much as two NVLinked RTX 8000 GPUs could possibly store.

3.2 BVH Memory

To allow for random access sample reconstruction, OSPRay uses a binary min-max BVH [24] in which each node stores both the spatial bounds and the minimum and maximum scalar value of any vertex within this node's subtree (the latter isn't required for cell location, but is useful for implicit iso-surface ray tracing). In OSPRay, each such node consists of six floats for the spatial bounds, two floats for the min and max of the scalar field, and a 64-bit integer, where three bits encode how many primitives this node contains (a value of 0 primitives indicates an inner node), and the remaining bits encode an offset into either the node array (for inner nodes) or into a list of 64-bit primitive indices (for leaves).

In total, this memory layout stores exactly one 64-bit integer per each unstructured element across the leaf item lists, plus $6 \times 4 + 2 \times 4 + 8 = 40$ bytes per each BVH node. In OSPRay, the number of BVH nodes created by the builder is decided by a surface area heuristic (SAH [16, 13]) based termination criterion, which in turn depends on the actual model[1].

For the Mars Lander data set, the OSPRay unstructured BVH builder creates a total of 5.75 billion BVH nodes, for a total of 214 GB in BVH nodes, and 22 GB in item lists.

Total memory used by OSPRay for the Mars Lander sums up to a total of 333 GB with roughly 71% going into the BVH, 26% going into unstructured element indices, and 2.6 % going into vertex data. A tabulated summary of this data is given in Table 1. Other data sets

[1]The BVH builder itself does not actually use a SAH criterion in OSPRay's unstructured mesh module, but the termination criterion does.

may have slightly different numbers, but the overall ratios will be roughly the same. Note that this is excluding any data that OSPRay would usually have computed for per-vertex gradients or similar, as well as any temporary memory used during BVH construction.

Datatype	number of	size/elt	bytes total
vertex positions	576.3 M	12 B	6.4 GB
vertex scalars	576.3 M	4 B	2.2 GB
sum vertices			**8.6 GB**
element index records	2.9 B	32 B	87.8 GB
sum indices			**87.8 GB**
sum mesh data (vertices+indices)			**96.4 GB**
BVH nodes	575M	40 B	214 GB
BVH item list	2.9B	8 B	22 GB
sum BVH			**236 GB**
sum total (mesh+BVH)			**333 GB**

Table 1: Memory used for the Mars Lander data set when using the unstructured mesh implementation of Rathke et al. [24], as measured by loading into an instrumented version of OSPRay. (Element count K/M/G use multiples of 1000, bytes use 1024).

4 Encoding Mesh Data

Table 1 shows that the primary target for memory optimization should be the acceleration structure. However, the mesh data of Mars Lander alone exceeds any available GPU memory.

For the vertex positions and scalar values, we briefly considered some lower-precision encoding in the spirit of Segovia et al.'s hierarchical mesh quantization [26], but eventually discarded this primarily because of the scalar field data: while vertex positions do exhibit some spatial coherence, the scalar values can (and in practice, do) cover some very large range of numbers that cannot easily be quantized. We therefore opted to use the same four-float vertex layout as OSPRay.

For the unstructured mesh elements there are two sources of potential savings: reducing the average number of indices stored per element, and reducing the number of bytes per index.

4.1 Reducing Number of Element Indices

OSPRay stores eight indices per element, even though most elements are tetrahedra that would require only four indices. This gives us the opportunity to roughly halve the memory needed for vertex indices by adopting an encoding in which elements use only as many indices as required. The downside of this strategy is that elements become harder to address (because they are no longer all multiples of a common size). As a consequence, we would then need another way of encoding each element's type. An obvious choice for variable offsets and element types is to use unused bits in the leaves' item lists, but as we later show, it is better to eliminate these item lists altogether (cf. Section 5.1).

We initially adopted a hybrid solution in which all primitives are encoded in multiples of four indices; i.e., tetrahedra use four indices, and wedges, pyramids, and hexes all use eight (with any unused indices marked using a special "invalid index" value). This required encoding for only two element types (either four indices, or eight), and already produced significant savings; however, later experiments showed that leaves with multiple tetrahedra still contained many repeated vertex indices because nearby tets often share vertices, edges, or faces. To exploit this redundancy, we added a special *tet-pair* primitive where, for each leaf, we identify pairs of tets that share a face and encode these using only 5 indices (three for the shared face, and two for the other two vertices) instead of $2 \times 4 = 8$ for individual tets. The idea is similar to what was proposed by Gurung et al. [12] who grouped triangle pairs to quads to obtain a more compact memory representation for triangle meshes. In our case, the savings of using such tet-pairs vary, but for models with many tets is usually in the range of 10% of final data size. Pyramids, wedges, and hexes still all use eight indices, meaning that we eventually need to encode only three different primitive types, which will be useful later on (cf. Section 5.1). On average our primitive encoding gives us roughly a $2\times$ reduction in vertex indices that we need to store.

4.2 Sub-Mesh Encoding

For the Mars Lander, even after a $2\times$ reduction of indices, storing four or eight 32-bit integers per element would still require over 40 GBs (and with 8.6 GB for vertices, we would already exceed an RTX 8000's total GPU memory).

To further reduce this index memory, we adopt some ideas from Segovia et al. [26], and observe that *if* we pre-partitioned the whole mesh into several smaller, and independently encoded meshes with at most 2^N vertices per mesh, then each such mesh would require only up to N bits per index. This strategy does require that those vertices shared by primitives that end up in different meshes would need to get replicated into more than one mesh, which leads to a trade-off between lower memory for indices but more memory for replicated vertices. Starting with a single input mesh, we evaluated this concept by recursively partitioning each mesh into two submeshes until each submesh has at most 2^N vertices left. To do this we use a surface area heuristic (SAH [16, 13]). This is similar to the splits OSPRay's BVH builder would have performed, meaning that topologically the resulting partitions are very similar.

Using this pre-partitioner, we can now evaluate the trade-off between the number of index bits and vertex replication: Given the resulting data in Table 2, we adopted a number of $N = 16$ index bits: this is only about 10% worse than the optimum (at $N = 12$ bits), but unlike $N = 12$, results in a natively supported integer type. While $N = 8$ also would have resulted in a native data type, $N = 8$ requires an unacceptable amount of vertex replication, and consequently results in worse total memory usage than for $N = 16$.

Index bits	#gen. groups	Average num prims/grp	Average num vtx/grp	Total num vertices		Approx memory
32	1	2.7G	570M	570M	(+0%)	50 GB
16	15.1K	191K	32K	620M	+12.8%	28 GB
14	63.7K	45.2K	10.8K	671M	+22.1%	26 GB
12	286K	10.1K	2.7K	765M	+39.1%	25 GB
8	<4M	oom	oom	<1.7G	<+200%	<30GB

Table 2: Impact of pre-partitioning with different number of bits per vertex index (Section 4.2). Fewer bits for encoding indices requires generating more groups, which in turn triggers more vertex replication.

4.3 Encoding of Mesh Data: Summary

In total, we represent our input mesh as a set of multiple sub-meshes, with four floats for each vertex, and either four, five, or eight unsigned 16-bit integers per element. In this representation (and including vertex replication) for the Mars Lander, we end up with a total of 25 GBs for all mesh data—or almost $4\times$ less than our reference uncompressed layout.

5 Acceleration Structure Encoding

Even though we can now reduce mesh memory by roughly $4\times$, more work must be done to carefully encode the acceleration structure to reduce memory overhead. Once again referring to Table 1, there are three major avenues for reducing BVH memory: reducing the number of BVH nodes, reducing the size of each BVH node, and reducing the size of—or ideally, entirely eliminating—the item lists.

We observe that we are not going to build a single BVH over all primitives, but rather, adapt our BVH to the pre-partitioning as described before: i.e., we need each BVH to cover only one submesh. Since each submesh's size is necessarily limited, this also means we can use smaller integers to index into the (per-submesh) node array and index array.

5.1 Eliminating the BVH Leaf Lists

The first thing we can get rid of are the item lists. Since each primitive in the BVH is referenced by exactly one leaf, instead of each leaf node storing a pointer to a list of N node IDs, we can instead re-arrange the primitives in the order they are referenced by the nodes, and have each node store only the offset into a common list, and how many primitives this leaf contains. These two values (offset and count) could be stored in the same values that the address and length of the original item list would have been stored in, so the structure of the node itself does not change, but the item list—for the Mars Lander in OSPRay, a total of 22 GBs—completely disappears.

One caveat with this is that our primitives no longer have a uniform size and type (cf. Section 4.1). This requires us to encode which of the primitives in the leaf are four-index tetrahedra, which ones are five-index tet-pairs, and which ones are eight-index pyramids, wedges, or hexes. One way to do this is through two bits in each entry of the leaf list; however, we just eliminated these, so this is not possible. Instead, we solve this by encoding the type in the primitive ordering: we first store all the leaf's tetrahedra, then all pairs, and all others at the end; requiring only three small counters for all types; plus one offset to the start of this list (also see Section 5.3).

One challenge was that in order not to increase the final node layout (Section 5.3) we needed to squeeze all three counters into a single byte. This creates a trade-off in how many bits to spend on each type, because the number of bits available influences which kinds of leaves the BVH builder can possibly produce. We experimented with different values, and eventually chose to use 4 bits for tet pairs, and 2 bits each for individual tets and non-tet elements.

5.2 Adopting a Multi-Branching BVH

After eliminating the item lists, the first step we can take towards reducing BVH node memory is to adopt ideas from existing ray tracers such as Embree, and switch from a binary BVH to a multi-branching BVH with 8 children per node. This does not change the number of leaf nodes, but will create significantly fewer inner nodes.

We experimented with different branching factors, but eventually adopted eight: Always having 8 children per node means that no matter how many bits we use for any given per-node variable, we can always create byte-aligned (and thus, easily accessible) data by storing 8 of those together.

Unlike Embree's eight-wide BVH, we do not, however, build our BVH top-down. Rather, we first build a binary BVH, and then re-collapse this backwards from the leaves: when building top-down, one often ends up with leaves with fewer than 8 "active" children, which in turn leads to a low average number of active children per multi-node. Merging bottom-up cannot guarantee always fully occupied multi-nodes either (e.g., two subtrees with five children each cannot be merged into 10), but the average number of children per leaf is generally higher—-which in turn means less total nodes.

We observe that despite similar partitioning and termination criteria our initial binary BVH (before collapsing) has significantly fewer leaves than OSPRay's: OSPRay's BVH builder almost always partitions down to individual primitives, which seems excessive.

5.3 Multi-Node Encoding

After eliminating item lists and reducing the number of nodes, our final means of reducing BVH size is through more efficient encoding of the (8-wide) nodes themselves. For each of the 8 children of such a multi-node, we have to encode the following information: the 3D bounding box; the offset into either primitive list (if a leaf node) or node array (if a inner node); and the tree leaf counters (if a leaf).

For the bounding boxes, we initially considered incremental encoding as proposed by Mahovsky and Wyvill, but eventually abandoned that as being too expensive to traverse (due to a much higher stack requirement). We also considered the quantization proposed by Benthin et al. [2], with one float-precision box shared across the entire multi-node, and 8-bit quantization relative to this box.

Eventually, we use the same core idea as Benthin, but even more aggressively: instead of using floats for the shared bounding box we store this box using 16-bit quantization relative to the bounding box of the subtree (i.e., we use two layers of quantization: one relative to the parent submesh, then another relative to that shared box). For the individual node values, we then use only 4 bits (instead of 8).

5.4 Reducing Child Pointers

As observed by Benthin et al. [2], after quantizing the bounding information the size of a multi-node eventually becomes dominated by the 8 pointers (or offsets) and counters with which each of the eight children point to their children and contained primitives, respectively.

In order to not having to store 8 distinct pointers, we observe that by properly arranging the nodes and primitives, we can reduce this to only two offsets (and will eliminate one of those in the next section). First, we look at all of the node's 8 children that are inner nodes, and store them sequentially in the node array. With this, all an inner node child needs to compute its offset is this first index (which need be stored but once per multi-node), plus how many of its siblings to the left were inner nodes. Similarly, we can look at all children that are leaves, and store all their primitives sequentially in the item list. Again, each child only needs to know one shared offset value, plus how long each of its left siblings' item lists are.

5.5 Removing Inner Nodes Completely

The way just described each node would store only two offset values (one for inner-node children, and one for leaf node children), plus the 8 children's counter values (with a node's three counters being all zero indicating it is an inner node). Early experiments using this layout did indeed have good memory statistics, but at least with our naïve stack-based CUDA implementation exhibited a rather high traversal cost. This was root-caused to three factors: first, divergent code when different threads in the same warp traverse different children of different types, different primitive types, etc; second, a large amount of live state in the inner loop (for decompressing entire multi-nodes and primitives); and third, the need to maintain two large traversal stacks (one for the BVH over sub-meshes, and one within each sub-mesh).

To reduce this cost, we also implemented a scheme where we eliminated all interior nodes completely by always collapsing all subtrees with at most 8 leaves into a single multi-*leaf* node (in which all children are then leaves), and discarding all other nodes. To perform sample location without interior nodes, we then borrow from Wald et al. [34], and build an OptiX [22] BVH over the multi-leaves, using an epsilon-length ray to find all multi-leaves that overlap the sample point. Within the intersection program, we then decompress the multi-node, test the child boxes, and sample the primitives of those children that overlap the query point. An added advantage of this scheme is that we can put all multi-leaves (even across different sub-meshes) into the same BVH.

The downside of this scheme is that OptiX will spend more memory on the inner nodes than our representation would have done; we currently see the OptiX BVH costing roughly 50% more for inner nodes than what our inner nodes would have cost. Amortized over all other data that needs to get stored (vertices, primitive, and leaf data), the OptiX BVH costs roughly 15% of total model size, as opposed to only about 8% for inner nodes in our format. On the upside, this scheme offers three advantages: first, it allows for a slightly better (because 16-byte aligned) node layout of 48 vs 52 bytes because we can save the offset value for inner nodes. Second, we do not need a second-level BVH over sub-meshes because we can put all multi-leaves in a single OptiX BVH. Third, and most importantly, this scheme allows for leaving all but the final node

layer of BVH traversal entirely to OptiX, which early experiments showed to give a roughly $10\times$ speed improvement over our own CUDA BVH traversal.

5.6 Final Node Layout

Our final node layout (assuming an OptiX BVH for inner nodes) is given in Figure 3, and sums to a total of 48 bytes: $6\times 2 = 12$ bytes for the shared bounding box, 4 bytes for the shared item list offset, 8 bytes for the eight counters bytes (each being $4+2+2$ bits), and $8\times 3\times(4+4)$ bits (24 bytes) for the quantized child boxes. We observe that these 48 bytes encode up to 8 nodes, as compared to 40 bytes per each node in our reference uncompressed layout.

```
struct MultiLeaf {
  box3us   quantizedBounds;   // 12 bytes
  uint32_t leafChildrenOffset; // 4 bytes
  struct {
    struct {              // bytes 16..40
      struct {
        uint8_t bounds_lo:4;
        uint8_t bounds_hi:4;
      } child[8];  // 8 bytes
    } dim[3];     // 24 bytes
    struct {              // bytes 40..48
      uint8_t numTets :2;
      uint8_t numPairs:4;
      uint8_t numOther:2;
    } primCount[8]; // 8 bytes
  } children;
}; // 48 bytes total
```

Figure 3: Final data layout for our quantized multi-nodes. When not using the OptiX-BVH we need an extra 4 bytes for the children offset.

5.7 Controlling BVH Size

Just as in OSPRay, our BVH builder uses a surface area heuristic (SAH [16, 13]) to determine whether to split a node, or make a leaf. We first construct a binary BVH using this builder, then use this to create our multi-leaves as described in Section 5.5.

An SAH builder usually includes a *SAH termination criterion* that decides when to make a leaf based on estimated costs for a traversal step and primitive intersection, respectively. We do use this criterion, but with two caveats: First, we chose a traversal cost estimate to be twice that of the primitive cost estimate, which should produce slightly shallower BVHes. On the other hand, we make sure that this stage will only create leaves that could actually be encoded with the final 4:2:2 bit encoding described in Section 5.6, which may force some splits where the regular SAH would have made a leaf.

5.8 Implicit LOD Information

Though rendering is not the focus of this paper, we do observe that the final data structure also offers several interesting means of looking at the data in a hierarchical way: In particular, the averages of a given BVH leaf's, multi-leaf's, or even sub-mesh's value range, respectively, are close to the value of any samples taken within these parts of the data structure. This offers obvious potential for using this information to employ a level-of-detail technique that could be used to, for example, generate faster samples, to provide information for space skipping or adaptive sampling, or to bridge loading latencies.

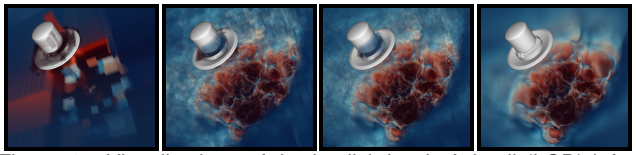


Figure 4: Visualizations of the implicit level of detail (LOD) information across our data structure's four different levels of encoding: a) rendered with the average of each sub-mesh's scalar field range; b+c) with the average of the multi-leaf and leaf node's value range, respectively; d) with the real, per-element reconstructed samples.

To demonstrate this, we modified our encoding to also store these scalar ranges, and used this to render three different images (Figure 4) where the correct samples have been replaced with the average of the first leaf, multi-leaf, or sub-mesh, respectively. However, a detailed discussion of how to use this level of detail during rendering is completely orthogonal to, and beyond the scope of, this paper.

6 Results

To evaluate our method's efficacy, we collected several of the largest unstructured mesh data sets we could get access to; an overview of these model, and statistical data on their primitive and vertex counts, is given in Figure 5. Of these models TACC Japan, DKRZ Ocean, and the three versions of the Mars Lander are native unstructured meshes from various unstructured codes; ExaJet was natively octree-AMR and converted to an unstructured mesh by computing its dual mesh. We explicitly point out that even the smallest of our data sets was among the largest data sets used by previous work.

6.1 Encoding Efficiency

Table 3 gives statistical data on those data sets, including vertex and primitive counts as well as the total number of vertices, sub-meshes, and multi-leaves; the aggregate memory used by these various data types, and the total amount of memory that a renderer using this data structure would require.

For all but the biggest version of the Mars Lander, our total memory usage would comfortably fit into the 48 GBs of an RTX 8000 or Ampere A6000 GPU. For the big Mars Lander this is—just barely—no longer possible, requiring the model across two such GPUs. We currently do the latter by either using a prototypical sort-last data-parallel renderer (cf. Section 7.3), or by using NVLINK and managed memory to distribute data across two such GPUs.

6.1.1 Comparison to Morrical et al.

For a comparison of the encoding efficiency, in addition to our encoded data structure, we also implemented the data structures discussed by Morrical et al. [17] which either use a shared face representation, or represent the unstructured elements with OptiX user geometries. We are primarily interested in the encoding efficiency of the three methods and report that in Table 4. For the comparison we aimed at saturating the available GPU memory with the respective methods. We therefore split the data structure for the big NASA lander into 64 consecutive chunks, allowing us to only partially upload the data set to the GPU. The most memory-intensive data representation is the shared-faces representation by Morrical et al.—we were able to fit four out of 64 chunks on a single RTX 8000 GPU (48 GB DDR memory total). We were able to load 12 out of 64 chunks with the user geometry data representation that was also discussed by Morrical et al. before eventually running out of memory. With the same amount of unstructured elements, our data structure consumes a total of 6.6 GB GPU memory.

6.1.2 Comparison to OSPRay

To compare our encoding to that used by OSPRay we also loaded these models into OSPRay, and measured memory as discussed in Section 3. Since OSPRay has recently undergone major changes in how it handles unstructured data we have done this for both versions 1.8.5 and 2.0. The result of this comparison is given in Table 5.

On average we see a reduction in memory usage of roughly $9.4-10\times$ relative to OSPRay 1.8.5, and $4.7-14\times$ relative to OSPRay 2.0. Generally speaking tetrahedral meshes seem to see better memory savings than hexahedral ones, largely because we can compress the BVH better than index memory, and tetrahedral meshes tend to spend more of their memory on BVH nodes than hexahedral ones do.

We intentionally limit our comparison to only the memory consumption aspect, as we could neither get our ray marcher integrated

model	input		memory (in bytes)		**Variant 1**: multi-leaves + OptiX BVH				**Variant 2**: inner nodes in our format		
	verts	elts	verts	indices	#nodes	mem(nodes)	OptiX	**total mem**	#nodes	mem(nodes)	**total mem**
Jets	2M	12M	34 MB	58 MB	262 K	11 MB	13 MB	**116 MB**	2 M	14 MB	**106 MB**
Impact 5K	17M	31M	445 MB	457 MB	2 M	104 MB	112 MB	**1.13 GB**	2 M	136 MB	**1.05 GB**
Impact 20K	176M	247M	2.85 GB	2.96 GB	15 M	438 MB	758 MB	**7.01 GB**	19 M	904 MB	**6.71 GB**
Impact 46K	299M	356M	4.84 GB	4.73 GB	26 M	1.16 GB	1.27 GB	**12.0 GB**	31 M	1.50 GB	**11.1 GB**
TACC Japan	93M	47.8M	886 MB	729 MB	4 M	170 MB	186 MB	**1.97 GB**	4 M	234 MB	**1.84 GB**
DKRZ Full Ocean	485M	749M	6.62 GB	11.2 GB	54 M	2.41 GB	2.67 GB	**22.9 GB**	68 M	3.20 GB	**21.0 GB**
NASA Exa-Jet	656M	652M	10.8 GB	10.4 GB	52 M	2.34 GB	2.59 GB	**26.9 GB**	64 M	3.33 GB	**24.5 GB**
Mars Lander Small	143M	789M	2.39 GB	4.5 GB	33 M	1.16 GB	1.27 GB	**9.3 GB**	42 M	1.49 GB	**8.38 GB**
Mars Lander Big	577M	2.9G	9.5 GB	17.7 GB	128 M	5.72 GB	6.36 GB	**39.3 GB**	162 M	7.87 GB	**35.1 GB**
Mars Lander Huge	1.14G	6.3G	19 GB	35.4 GB	267 M	11.95 GB	13.27 GB	**79.6 GB**	340 M	16.49 GB	**70.9 GB**

Table 3: Input statistics, number of components generated by our builder, and final memory consumption (*excluding* any third-party GPU data like frame buffer, surface mesh, transfer function, space skipper data, etc) using our memory-optimized encoding.

		Morrical et al.			
model	ours	User Geom		Shared-Faces	
Lander (4/64)	2.4GB	14.5GB	6.0×	34GB	14×
Lander (12/64)	6.6GB	39.3GB	5.9×	oom	*X*×

Table 4: Encoding efficiency of our data structure compared to Morrical et al.'s OptiX user geometry and shared faces methods [17]. We split the big NASA lander data set into 64 chunks, which allows us to partially upload the data set to the RTX 8000 GPU (48 GB DDR memory). Where the more memory-efficient user geometry representation runs out of GPU memory, our data structure uses only 6.6 GB.

into OSPRay, nor could we use OSPRay's ray marcher or data layouts in our framework. We observe that we do see interactive performance with our prototype renderer, which for these models and quality settings we have not managed to achieve with OSPRay's. However, we conversely point out that OSPRay is a complete rendering framework that can do many things that our current framework can not.

model	ours	OSPRay 1.8.5		OSPRay 2.0	
TACC Japan	1.84 GB	—	—	8.6 GB	4.7×
DKRZ Ocean	21.0 GB	—	—	123 GB	5.9×
Exa-Jet	24.5 GB	—	—	120 GB	4.9×
Lander Small	8.38 GB	90.2 GB	10.1×	117 GB	14.0×
Lander Big	35.1 GB	333 GB	9.4×	432 GB	12.3×

Table 5: Comparison of our representation's memory usage to essentially the same data structure (but differently encoded) in OSPRay 1.8.5, and 2.0. ("—" indicates that our instrumented OSPRay did not have an importer for this data set's format).

6.2 Encoding Time

Computing our encoding is currently an offline and out-of-core preprocess that can, for the largest model, take several hours. This could probably be significantly improved, but not to interactive rates: at least for our largest models, just reading the input model takes several minutes, as does writing the final outputs. On the upside, once a model's compressed encoding has been computed and stored on disk, it can then be read and re-used again at any time; and reading mesh *and* BVH together in encoded form takes roughly as long as reading just the input data would have taken in uncompressed form.

Part of the reason our build currently takes so long are a direct consequence of data size: wrangling hundreds of gigabytes of data takes time, even assuming enough (host) memory is available to handle it; the same argument is true for computing billions of bounding boxes, billions of BVH nodes, etc. For example, just re-computing the new mesh connectivity after splitting the model into sub-meshes takes many minutes. Other reasons are more specific to our current implementation: First, our current implementation prioritized easy experimentation and evaluation of different trade-offs such as different bit counts, data layouts, tet-pair merge heuristics, etc; this flexibility comes at a cost (in particular, at the cost of frequent memory allocations and data copies to frequent use of STL data structures). Second, our implementation currently performs a large number of validations and checks for corner cases, as even unlikely errors can compound over billions of executions. Third, much of the data currently has to be held several times, in different formats (e.g., temporarily keeping both binary BVH and multi-node BVH, etc), further compounding memory pressure. Finally, our current build has a lot of intrinsic dependencies: for example, we currently cannot predict how many entries in the index vector a given subtree will require until that subtree is built, severely limiting the amount of parallelism we can use. Many of these factors could be addressed by a more efficient, parallel encoding—ideally building each subtree in parallel on the GPU—but this will be left for future work.

6.3 Rendering Quality and Performance

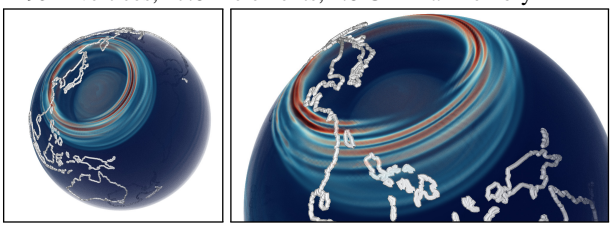
So far, we have only talked about the memory efficiency of our encoding, and not much about how to sample it, let alone about the best ways to write a volume ray marcher that uses it. This is intentional: *where* a volume renderer places its samples (i.e., what version of space skipping and/or adaptive sampling it uses) will have a massive influence on rendering performance that for models of this complexity will easily dwarf the impact from sample cost. Thus, unless one is very careful when comparing different renderers this can lead to skewed and sometimes misleading results.

As such, we intentionally refrain from direct comparisons to any secondary code bases, and perform all evaluations in our own, thus guaranteeing all variants to use the exact same samples, transfer functions, hardware, etc. To this end, we implemented two different prototypical renderers on top of our data structure. For reference, we first implemented a naïve direct volume renderer that uses equidistant samples and does but limited space skipping. In addition, we also implemented a high-quality GPU volume renderer based on Monte Carlo sampling (all images in Figure 5 are rendered with this renderer). With both renderers we do, in fact, see interactive results (albeit with much higher quality when using the latter). However, due to the size of our models, we could not yet use the same space skipping and adaptive sampling techniques used by previous work such as Morrical's [17], and though we have some early results that achieve similar sampling efficiency, a detailed discussion of this is beyond the scope and focus of this paper.

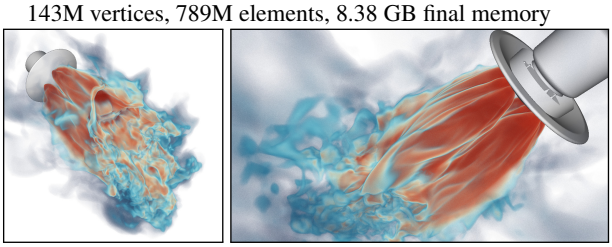
One important question to answer is how much our encoding's memory savings will cost in terms of performance: Where less efficient encodings can directly operate on individual primitives our method requires operating on entire multi-leaves, including some non-trivial decoding and dequantization of the data contained therein—and this decoding cost is not cheap.

To evaluate this trade-off we also took our implementations of the *shared faces* and *user geom* methods mentioned in Section 6.1.1, added the respective sample routines, and—for some suitably small enough models to fit those encodings—ran those against our compressed encoding. According to Table 6, this places this trade-off at roughly $2\times$ lower performance for an associated $6-7\times$ reduction

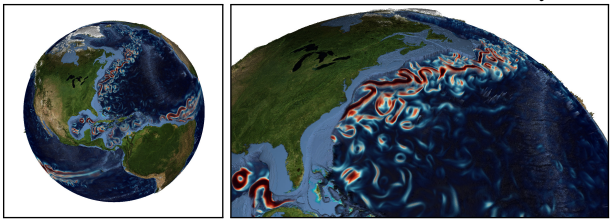
TACC Japan Earthquake
93M vertices, 47.8M elements, 1.8 GB final memory



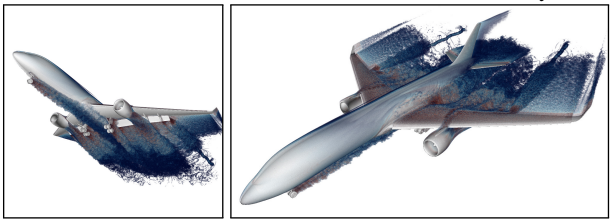
NASA Mars Lander ("small"), Velocity field
143M vertices, 789M elements, 8.38 GB final memory



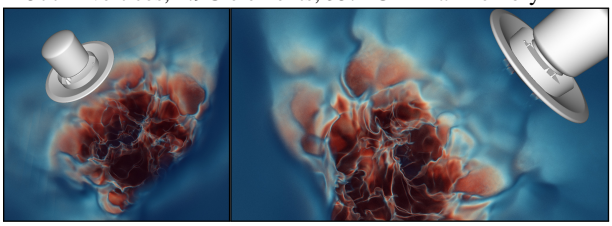
DKRZ Full Ocean
485M vertices, 749M elements, 21.0 GB final memory



NASA Exa-Jet (dual mesh)
656M vertices, 652M elements, 24.5 GB final memory



NASA Mars Lander ("big"), Density field
577M vertices, 2.9G elements, 35.1 GB final memory



model	vertices	tets	pyrs	wedges	hexes	(sum)
TACC Japan	93M	0	0	0	48M	48M
Mars Lander Small	143M	756M	47K	33M	0	789M
DKRZ Full Ocean	485M	0	0	749M	0	749M
NASA Exa-Jet	656M	580K	2.1M	2.6M	647M	652M
Mars Lander Big	577M	2.7G	283K	247M	0	2.95G

Figure 5: The five data sets used for evaluating the memory efficiency of our framework (all images are rendered with our prototype renderer framework). Final memory consumption is final memory uploaded to GPU after applying our quantized/compressed encoding. Note that even our smallest model is at the upper range of what previous GPU based unstructured renderers can handle.

in memory relative to the *user geom* method, and roughly 6× lower performance for order 15× memory savings for the all hardware accelerated *shared faces* method.

method	1/8	2/8	3/8	4/8	5/8	6/8	7/8	8/8
	Time per Frame (seconds)							
Ours	0.34	0.73	1.03	1.30	1.70	2.02	2.15	2.18
User Geom	0.57	1.42	2.24	2.90	3.79	4.43	4.72	4.96
Shared Faces	0.05	0.12	0.17	0.21	0.28	0.32	0.35	0.39
Tet Marching	0.03	0.07	0.10	0.15	0.20	0.23	0.24	0.24
	Memory Consumption (MB)							
Ours	16	31	50	62	81	99	111	123
User Geom	102	206	311	416	520	625	730	831
Shared Faces	255	512	768	1020	1280	1540	1790	2040
Tet Marching	65	120	175	230	285	340	395	440

Table 6: Performance and memory use for our encoding relative to the *shared faces* and *user geom* methods proposed by Morrical et al. [18] as well as a reference *tetrahedra marcher* on the (small) *Jets 12K* data set. Jets was chosen as it consists solely of tetrahedra (required for tet marching) and is small (highlighting compression overhead during rendering).

7 Discussion

We can conclude from the results of the studies that our data structure is efficient at interactively rendering and fitting models into the GPU memory of workstations that before couldn't be rendered interactively with off the shelf software. In aggregate, our scheme allows for representing a data structure very similar to the one used by OSPRay's unstructured mesh renderer in $4.7-14\times$ less memory (with largely tetrahedral data benefiting more than largely hexahedral data). This in turn allows for fitting all but the largest of our data set onto a single GPU (the largest needs two NVLinked ones), which then allows for sampling those in CUDA, and, if so desired, with RTX acceleration for the cell location. We acknowledge that the focus of a software system like OSPRay is a potentially different one than ours and that CPU memory is often less limited in typical workstations than GPU memory. We however find the memory savings compared to both OSPRay, as well as compared to the approach by Morrical et al. [17], compelling.

We note that a limitation of our approach is that the data structure can neither be rebuilt nor updated interactively. While this wasn't the primary objective of our study, exploring potential trade-offs between build/update times on the one hand, and memory efficiency on the other hand, would be an interesting future direction.

7.1 Comparison to Ray / Mesh Traversal

An approach that is orthogonal to ours is that of traversing the volumetric mesh face to face and thereby integrating the volume over the exact distance that the ray travels through each unstructured element. Memory-efficient data structures for this approach have for example been proposed by Muigg et al. [19]. An advantage of such approaches is that they do not require an auxiliary data structure other than some connectivity information stored with the elements. With careful data handling and taking the winding order of face vertices into account, the memory requirements for tetrahedral meshes can be reduced significantly with such approaches. To the best of our knowledge, there does not exist a publicly available state-of-the-art implementation of such mesh traversal approaches that is accessible enough, so that we instead discuss the difference in some detail in the following. We consider a quantitative comparison interesting future work.

There are a number of major advantages of our approach over such mesh traversal methods. Mesh traversal makes use of connectivity information. For "real-world data sets", it is however often hard to obtain *reliable* connectivity information, for example in cases where mesh faces or vertices are duplicated, or at refinement boundaries

as they occur during numerical simulation [35], where elements of different mesh resolution are connected and faces are coplanar but do not have all their vertices in common. Similar issues arise with partially overlapping, unstructured elements. In those cases with degenerate or generally "difficult" faces or elements, our technique is robust in a sense that elements sporadically might not be sampled, but traversal for the entire ray stays unaffected. Mesh traversal methods on the other hand are highly sensitive to those issues and require significant handling of corner cases to not fail completely.

Another advantage of our approach is scalability. As mesh traversal approaches rely on marching from one element to another, conceptually, all those elements need to be touched and pulled from memory, which makes optimizations such as adaptive sampling hard to implement. In contrast to that, a sampling approach such as ours will eventually outperform mesh traversal approaches, but at the expense of potentially undersampling the volume. With our approach, it is however trivial to implement jittered ray offsets and accumulate convergence frames over time—in fact, this is what we do in our prototypical renderer—so that time to first image is highly reduced with our approach for large data sets. Generally, we consider the two approaches orthogonal, and while both have their individual merits, our data structure is one step into the direction of further amortizing the memory overhead of acceleration data structures and closing the memory gap between the two approaches.

7.2 Generality of Our Approach

Our prototypical implementation and the evaluation currently only use NVIDIA hardware. Nevertheless, the optimizations we propose aren't specific to that hardware platform and should be similarly applicable to GPUs (or even CPUs) by other vendors.

7.3 Compression vs Data-Parallel Rendering

While our method significantly increases the limits of what data can still be rendered on a single machine, data size rises at a rate that eventually even with our technique a single GPU will not be enough. Furthermore, data-parallel rendering is common in scientific visualization, often because data-parallel computing is necessary for other parts of the simulation and/or visualization pipelines. As such, an important question to address is whether our approach is addressing the wrong problem, and whether it would not make more sense to rather focus on improving scalability rather than what can be done on a single GPU.

Though we absolutely acknowledge the need for ultimately going data-parallel, we point out two caveats of this argument. First, today's approaches to data parallel rendering predominantly rely on sort-last compositing, which hampers techniques like space skipping, and precludes the kind of high-quality path tracing that our single-node renderer can do. As such, our technique not only changes how much data can still be renderer on a single GPU, but also how much data can still be rendered with the high-quality renderer.

Second, we observe that our technique and going data-parallel are not at all mutually exclusive, as reducing how much memory is required for rendering only *improves* the choices for data parallel rendering: reducing how much memory is needed for rendering allows to either use less nodes (reducing both cost and communication); or to use that memory for other means.

To evaluate this trade-off we integrated our data structure into a data parallel renderer that spatially distributes the models across multiple GPUs, renders the resulting chunks using (non path traced) direct volume rendering, and then combines the intermediate results using MPI-based sort last compositing. In Figure 6 we report results running this data-parallel renderer on a workstation equipped with eight NVIDIA A6000 GPUs (48 GB memory per GPU); once using an uncompressed BVH representation, and once using our compressed one. As expected, for any given number of GPUs used the uncompressed version is usually faster; however, most models require multiple GPUs to render at all, while all but the largest lander would run with a single GPU. This would allow our compressed version to instead use all eight GPUs for data-*replicated* rendering, at much higher performance *and* higher quality.

To demonstrate this we also ran another set of experiments on a workstation with four RTX 8000 GPUs (48 GBs each), using the small and big lander models and our data-parallel renderer (see Table 7): Using Morrical's *user geom* representation we can just barely fit the big lander across all four GPUs (in fact, some data gets paged out over managed memory), while with our encoding, the same model fits into one GPU, allowing to either run with a single GPU (if only one was available), or to instead run data-replicated across all four.

	user geom	same renderer, using our encoding		
	sort-last (4 GPUs)	sort-last (4 GPUs)	one GPU (1 GPU)	replicated (4 GPUs)
small lander	1.49	1.49	0.59	1.96
big lander	2.2	1.85	0.88	2.9

Table 7: Performance (in frames per second) running two large models in either sort-last data-parallel using a reference BVH, vs. the same using our encoding. Using our lower memory footprint we can not only render the same model at similar performance in sort-last, but could also render either stand-alone on a single GPU, or data-replicated.

8 Summary and Conclusion

The main goal of this paper was to develop—and evaluate—a more memory-efficient encoding of all the data required to perform sample-based rendering of large unstructured data on a GPU. To do this we have proposed a scheme that uses hierarchical encoding and quantization to reduce the total memory size of both unstructured element data and BVH built over these primitives, which in summary achieves roughly an order of magnitude memory reduction relative to comparable uncompressed representations.

Being now able to fit these models on a GPU, the next big question is how to best use this data structure for rendering. We do have prototypical rendering implementations; however, truly real-time rendering for models of this size will require more advanced techniques for space skipping, adaptive sampling, and probably level-of-detail than what we have so far implemented. In that vein the biggest issue is that space skipping and adaptive sampling are most easily used with spatial subdivision techniques in which subtrees do not overlap—this is not the case for our BVH, but building an additional data structure just for space skipping raises obvious concerns regarding memory use.

In terms of algorithms operating on this data structure we have so far only looked at sample reconstruction. In OSPRay, the Min/Max BVH is also used for implicit iso-surface ray tracing, and though the same *should* work with ours, too, we have not yet implemented this.

Maybe most importantly, our technique provides an interesting proof of concept, but to be truly useful, would eventually have to get integrated into a larger and more easily end user-accessible software framework such as VTK, VisIt, or ParaView; possibly through an OSPRay or ANARI API [15]. How to best achieve this, however, is still an open question.

Acknowledgements

This paper could not have happened without generous help by Pat Moran (NASA) in sharing and interpreting the Mars Lander data. Other data sets were provided by the Texas Advanced Computing Center (TACC), the German Climate Computing Center (DKRZ), and both the visualization and LAVA groups at NASA. Compute resources were provided by TACC (for some of the data-parallel rendering work), the Scientific Computing and Imaging Institute

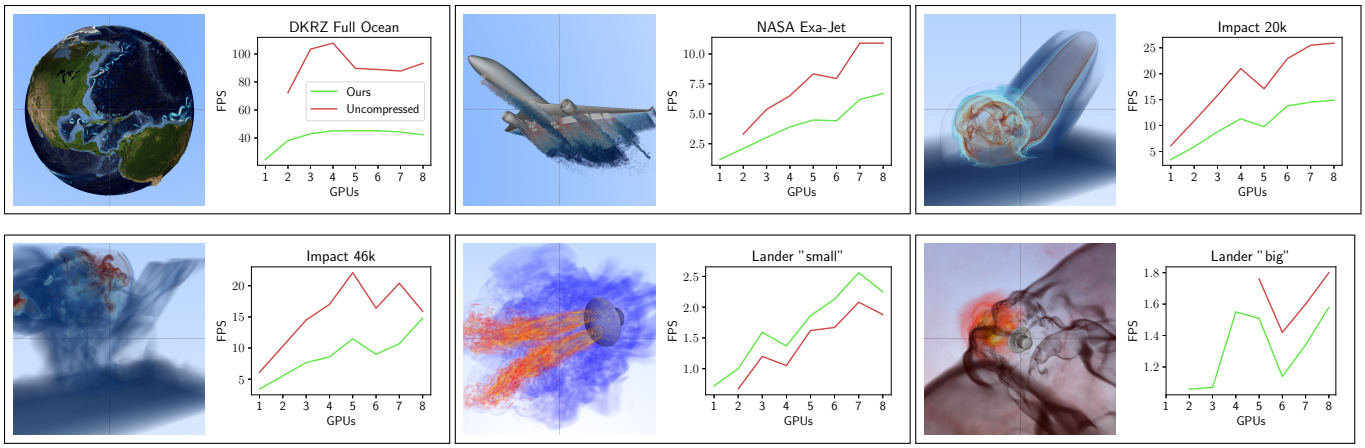


Figure 6: Results of a scalability study we ran on a server with eight NVIDIA A6000 GPUs (48 GB DDR memory each), using a slightly older and less efficient encoding than the final one described in this paper. We compare the performance of rendering with our data structure against an uncompressed data representation, using a naïve ray marching approach and prototypical sort-last data-parallel renderer. Green line denote our compressed data representation, red indicates the reference data representation. In cases where plots don't fully extend to the left, we could not fit the model into the combined GPU memory.

(Valerio Pascucci) as well as NVIDIA, which also provided hardware for this research. Finally, we are deeply indebted to Will Usher for help in getting all the data for the OSPRay comparisons.

REFERENCES

[1] NASA: Fun3D Retropropulsion Data Portal—Simulations of retropropulsion to decelerate a space vehicle entering a planetary atmosphere. https://data.nas.nasa.gov/fun3d/, 2021.

[2] 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.

[3] S. Castro, L. Castro, L. Boscardín, and A. De Giusti. Multiresolution wavelet based model for large irregular volume data sets. In *Proceedings of the The 14-th international Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, 2006.

[4] P. Cignoni, F. Ganovelli, E. Gobbetti, F. Marton, F. Ponchio, and R. Scopigno. Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Transactions on Graphics*, 23(3), 2004.

[5] H. Dammertz, J. Hanika, and A. Keller. Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. 27(4), 2008.

[6] E. Danovaro, L. De Floriani, M. Lee, and H. Samet. Multiresolution tetrahedral meshes: an analysis and a comparison. In *Proceedings Shape Modeling International (SMI)*, 2002.

[7] R. A. Drebin, L. Carpenter, and P. Hanrahan. Volume rendering. *SIGGRAPH Comput. Graph.*, 22(4), 1988.

[8] M. Ernst and G. Greiner. Multi bounding volume hierarchies. In *2008 IEEE Symposium on Interactive Ray Tracing*, 2008.

[9] R. Fellegara, L. D. Floriani, P. Magillo, and K. Weiss. Tetrahedral Trees: A Family of Hierarchical Spatial Indexes for Tetrahedral Meshes. *ACM Transactions on Spatial Algorithms and Systems*, 6(4), 2020.

[10] R. Fellegara, K. Weiss, and L. De Floriani. The stellar decomposition: A compact representation for simplicial complexes and beyond. *Computers & Graphics*, 2021.

[11] C. Garth and K. I. Joy. Fast, memory-efficient cell location in unstructured grids for visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1541–1550, 2010.

[12] T. Gurung, D. Laney, P. Lindstrom, and J. Rossignac. SQuad: Compact representation for triangle meshes. In *Computer Graphics Forum*, vol. 30, pp. 355–364. Wiley Online Library, 2011.

[13] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech TU in Prague, 2001.

[14] A. Hunter and P. Willis. Classification of quad-encoding techniques. In *Computer graphics forum*, vol. 10, pp. 97–112, 1991.

[15] ANARI: Analytic Rendering Interface for Data Visualization. Khronos Press Release, https://www.khronos.org/anari, 2020.

[16] J. D. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *Visual Computer*, 6(6):153–65, 1990.

[17] N. Morrical, W. Usher, I. Wald, and V. Pascucci. Efficient Space Skipping and Adaptive Sampling of Unstructured Volumes Using Hardware Accelerated Ray Tracing. In *IEEE Visualization Conference*, 2019.

[18] N. Morrical, I. Wald, W. Usher, and V. Pascucci. RTX Beyond Ray Tracing: Extending the Use of Hardware Ray Tracing Cores for Unstructured Mesh Point Location. *Transactions on Visualization and Computer Graphics*, 2020. Under Review.

[19] P. Muigg, M. Hadwiger, H. Doleisch, and E. Gröller. Interactive Volume Visualization of General Polyhedral Grids. *IEEE Transactions on Visualization and Computer Graphics*, 2011.

[20] B. Nelson, E. Liu, R. M. Kirby, and R. Haimes. ElVis: A System for the Accurate and Interactive Visualization of High-Order Finite Element Solutions. *IEEE Transactions on Visualization and Computer Graphics*, 2012.

[21] R. Pajarola, J. Rossignac, and A. Szymczak. Implant sprays: Compression of progressive tetrahedral mesh connectivity. In *Proceedings Visualization'99 (Cat. No. 99CB37067)*, 1999.

[22] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, and A. Robison. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)*, 2010.

[23] J.-L. Peyrot, L. Duval, F. Payan, L. Bouard, L. Chizat, S. Schneider, and M. Antonini. HexaShrink, an exact scalable framework for hexahedral meshes with attributes and discontinuities: multiresolution rendering and storage of geoscience models. *Computational Geosciences*, 23(4), 2019.

[24] B. Rathke, I. Wald, K. Chiu, and C. Brownlee. SIMD Parallel Ray Tracing of Homogeneous Polyhedral Grids. In *Eurographics Symposium on Parallel Graphics and Visualization*, 2015.

[25] S. M. Rubin and T. Whitted. A 3-Dimensional Representation for Fast Rendering of Complex Scenes. *SIGGRAPH Comput. Graph.*, 14(3), 1980.

[26] B. Segovia and M. Ernst. Memory efficient ray tracing with hierarchical mesh quantization. In *Proceedings of Graphics Interface*. 2010.

[27] P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. In *Proceedings of the 1990 Workshop on Volume Visualization*, p. 63–70, 1990.

[28] C. T. Silva, J. L. D. Comba, S. P. Callahan, and F. F. Bernardon. A survey of GPU-based volume rendering of unstructured grids. *Revista de informática teórica e aplicada*, 12(2), 2005.

[29] E. Strohmaier, J. Dongarra, H. Simon, M. Meuer, and H. Meuer.

Top500. *Online] http://www. top500. org*, 2020.

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

[31] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1), 2007.

[32] I. Wald, H. Friedrich, A. Knoll, and C. D. Hansen. Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes. *IEEE Transactions on Visualization and Computer Graphics (Proceedings of IEEE Visualization/InfoVis 2007)*, 13(6), 2007.

[33] I. Wald, G. P. Johnson, J. Amstutz, C. Brownlee, A. Knoll, J. Jeffers, J. Günther, and P. Navrátil. OSPRay – A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 2017.

[34] 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 *High-Performance Graphics - Short Papers*, 2019.

[35] 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*, 2020.

[36] B. Wylie, K. Moreland, L. A. Fisk, and P. Joyce. Tetrahedral projection using vertex shaders. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics*, 2002.

[37] H. Ylitie, T. Karras, and S. Laine. Efficient incoherent ray traversal on GPUs through compressed wide BVHs. In *Proceedings of High Performance Graphics*, pp. 1–13. 2017.