Eurographics Conference on Visualization (EuroVis) 2023
Authors' version (See the publisher's webpage for the definitive version)
Volume 0 (2023), Number 0

Memory-Efficient GPU Volume Path Tracing of AMR Data Using the Dual Mesh

Stefan Zellmann[1], Qi Wu[2], Kwan-Liu Ma[2], and Ingo Wald[3]
[1]University of Cologne [2]University of California - Davis [3]NVIDIA

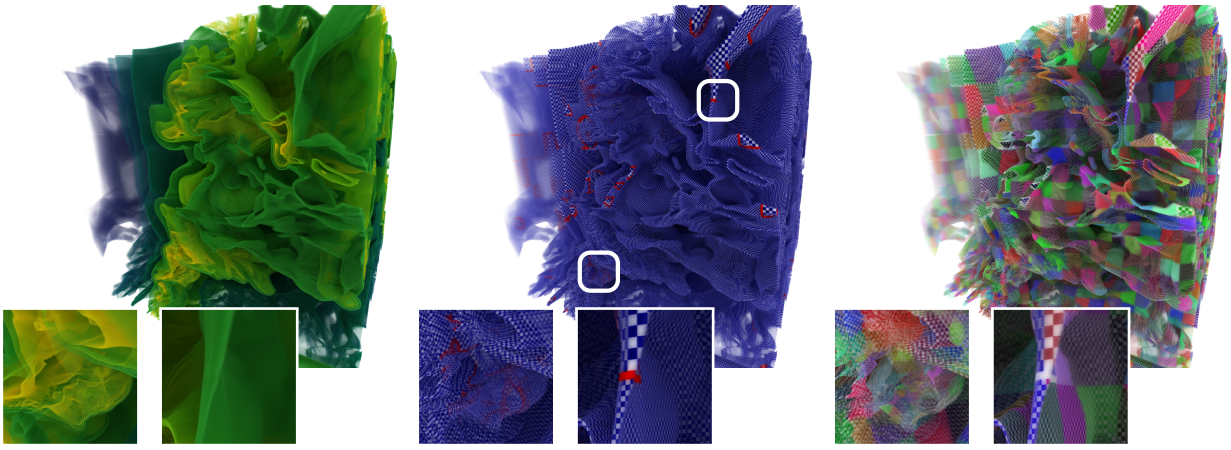


Figure 1: *Overview of our method. Given a block-structured or octree-AMR data set (left) we first create the dual mesh (middle) and split that into the truly unstructured elements used to stitch the level boundaries (red) and those that are regular voxels (blue/white checkered). We then cluster voxels to become "gridlets". Right: gridlets colorized by their ID. We build a bounding volume hierarchy over the gridlets and the remaining unstructured elements. The result is a sampleable representation that generates the exact same result as sampling on the dual mesh directly, but with significantly lower memory overhead and higher sampling speed. On our largest data sets, we see memory savings of up to 3× compared to highly compressed state-of-the-art unstructured mesh representations.*

Abstract

A common way to render cell-centric adaptive mesh refinement (AMR) data is to compute the dual mesh and visualize that with a standard unstructured element renderer. While the dual mesh provides a high-quality interpolator, the memory requirements of the dual mesh data structure *are significantly higher than those of the original grid, which prevents rendering very large data sets. We introduce a GPU-friendly data structure and a clustering algorithm that allow for efficient AMR dual mesh rendering with a competitive memory footprint. Fundamentally, any off-the-shelf unstructured element renderer running on GPUs could be extended to support our data structure just by adding a* gridlet *element type in addition to the standard tetrahedra, pyramids, wedges, and hexahedra supported by default. We integrated the data structure into a volumetric path tracer to compare it to various state-of-the-art unstructured element sampling methods. We show that our data structure easily competes with these methods in terms of rendering performance, but is much more memory-efficient.*

1. Introduction

Adaptive mesh refinement (AMR) was first proposed by Berger and colleagues [BO84, BC89] and refers to simulation codes that adaptively refine the grid used for the computation in both space and time. The resulting AMR *data sets* are collections of uniform, rectilinear subgrids at different *refinement levels*. Subgrid cells at level $l \in \mathbb{N}_0$ have a size of 2^l units of length.

The subgrids produced by virtually all physics codes are cell-centric, which gives rise to the T-junction problem and makes ordinary rectilinear interpolators inapplicable; repeating or clamping values at level boundaries for isosurface or volume rendering results in cracks and discontinuities. One way to overcome this is to sample on the *dual grid* (syn.: dual *mesh*), where the data is associated with the corners and not the cell centers (cf. Fig. 1). The dual of an AMR grid as shown in Fig. 2a is computed by shifting

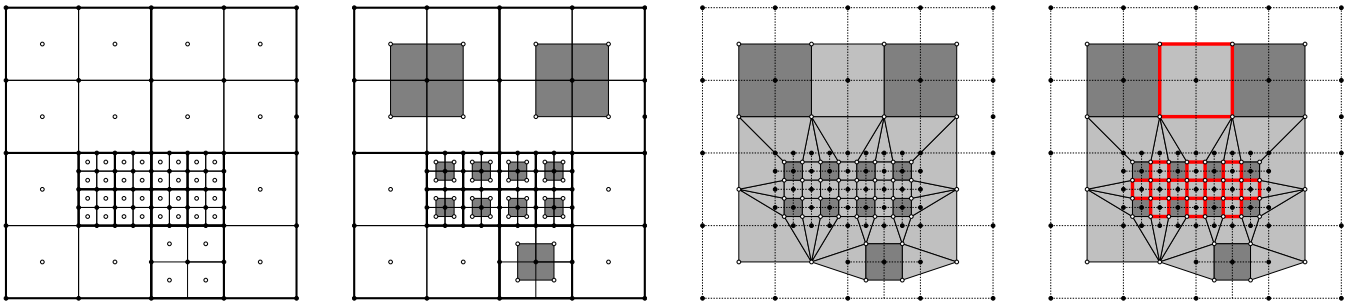


Figure 2: *Problem statement. (a) (Octree-) AMR input data. The dual mesh of each* 2×2 *subgrid (we omit the third dimension for clarity) is obtained by shifting all its cells, giving us the dark gray* dual cells *in (b). In this extreme case of* octree *AMR, the dual grid of each* 2×2 *subgrid is just a single dual cell. Original cells at the (level) boundaries have no corresponding dual cells at all. (c) The dual mesh of the AMR data set is constructed from the subgrid duals (dark gray), and by filling the gaps at the level boundaries with unstructured "stitching" elements (light gray). (d) State-of-the-art methods represent dual meshes as ordinary unstructured grids. Many of the cells are voxels (perfect cubes) though, including the stitching cells at same-level boundaries (highlighted in red). Representing those voxels with general hexahedra is memory intensive and prevents rendering of large data sets that are omnipresent in the computational sciences. We propose to address this by clustering the hexahedra so they become voxels again and their memory footprint is significantly reduced.*

all subgrids by half a cell's width so the new vertices snap to what originally were the cell centers (cf. Fig. 2b), and by introducing generally unstructured "stitching" cells at the subgrid boundaries, as shown in Fig. 2c.

This can result in data layouts with an inferior memory footprint; e.g., for octree AMR, subgrids have 2^3 voxels and generate only one inner cell, plus surrounding stitching cells at the boundaries (cf. Fig. 2c). All those cells, including the inner one, are represented with regular hexahedra; if the surrounding octree leaves however all come from the same level, not only the single subgrid cell, but all the surrounding stitching elements are perfect cubes (for example the highlighted stitching cells in Fig. 2d) and the representation using general hexahedra is extremely wasteful and can even make rendering medium-sized data sets impossible.

In this work we introduce a memory-efficient version of the dual grid interpolator by Moran and Ellsworth [ME11] for AMR volume rendering on GPUs. For that we split the generated *dual cells* into *generally* unstructured elements at the level boundaries and introduce a *gridlet* element type to cluster hexahedra that are actually cubes/voxels (cf. Fig. 1c).

This approach is very general, as gridlets *complement* the existing element types; by introducing a number of simple extensions, any unstructured renderer can accommodate the kind of large AMR data we focus on, without any additional changes to the *overall* rendering infrastructure. We integrated our data structure into a volume path tracer that uses hardware-accelerated ray tracing queries for sample reconstruction. We also implemented several state-of-the-art unstructured element samplers for comparison, most of which focus on reducing the memory footprint or excessive processing times for very large unstructured meshes. For our largest data sets, those other samplers consume several times more peak and total memory than ours. Our method is also competitive to—and in some cases even outperforms—highly optimized GPU AMR data structures that sample the data on the original grid.

2. Related Work

We review related work on volume rendering and path tracing, on scientific visualization of unstructured meshes, and on rendering AMR data sets.

2.1. Volume Path Tracing

Although direct volume rendering (DVR) is a popular scientific visualization algorithm, many implementations still use the absorption plus emission model [Max95] and biased integration methods like ray marching [Lev88]. Even though visualization tools like VisIt [Chi12] or ParaView [AGL05] still use this simple model, the field is however gradually transitioning to unbiased rendering methods using free-flight distance sampling to compute transmission estimates. This transition expresses itself in a number of recent papers on volume path tracing for scientific visualization [HMES20, MSG*22, XTC*22, ZWS*22b].

The goal is to stochastically compute *free-flight distances* that photons travel through a medium without colliding with other particles. If a collision happens, there are multiple options to consider; e.g., a shadow ray can be traced towards a light source to implement single scattering, or a random walk using phase functions could ensue to ultimately evaluate the whole rendering equation. Woodcock tracking [WMHL65] is the simplest tracking method and *homogenizes* the heterogeneous medium using fictitious particles; a collision with a fictitious particle is called a *null collision*, where the direction and throughput of the particle remain unchanged [NSJ14].

Woodcock tracking can use local *majorants*, which present an upper bound for the volume density in some region of space and can be used to build acceleration data structures [YIC*10, SKTM11]. While Woodcock tracking has in the meantime been superseded by more advanced tracking [GMH*19] or unbiased marching algorithms to perform inverse transform sampling [KDPN21], the principle of using data structures to compute upper (or lower) bounds with spatially varying majorants (or minorants) remain very relevant, as these can significantly reduce the number of rejection samples taken by the tracking algorithm.

2.2. Unstructured Mesh Rendering

While rendering algorithms for unstructured meshes are well-established [SCCB05], they primarily pose a *data management problem* for scientific visualization packages. That is because rendering algorithms based on ray tracing require a sampling or traversal data structure that adds to the usually already very high memory demand to store the unstructured elements themselves.

Ray marching approaches such as the one by Muigg *et al.* [MHDG11] or by Sahistan *et al.* [SDM*21, SDW*22] require a data structure that, given an element the marcher is currently in, and the direction where to go next, tells us how to get there; this can be done using shared faces, or by using other clever means. These approaches are usually rather restricted because in order to arbitrarily scatter into a certain direction they must travel there by marching from element to element.

Sampling methods such as the min/max BVH by Rathke *et al.* [RWCB15] or the hardware ray tracing approaches by Wald *et al.* [WUM*19] and by Morrical *et al.* [MUWP19] support cell location queries at arbitrary positions in $\mathbb{R}^3$ without first marching there to obtain interpolated values. They instead trace a zero-length ray into a bounding volume hierarchy (BVH) or similar data structure and use special intersection tests that report a hit when the ray origin falls inside an element. Here the BVH adds to the overall memory pressure.

The works by Morrical *et al.* [MSG*22] and by Wald *et al.* [WMZ22] present the current state of the art in unstructured mesh sampling with BVHs, in regards to construction time (Morrical's) and in regards to memory efficiency (Wald's). The two methods and optimizations proposed, however, focus on data that is generally unstructured, where often not even a single hexahedron is a perfect cube. Since we compare our method to them, the technical details of the two approaches are discussed in more detail in Section 6.2

2.3. High-Quality and Large-Scale AMR Rendering

Rendering cell-centric AMR data is particularly challenging, so that early works have concentrated on vertex-centric data only [MDV09], or resorted to sample reconstruction with zeroth order box filters if the data was cell-centric [KWAH06]. More recent papers have proposed high-quality reconstruction with first order C^0 continuous filters. An example are the tent-shaped basis functions by Wald *et al.* [WBUK17], which define 3D tents over each cell; the basis functions of those extend beyond the cell boundaries and therefore overlap. Computing a weighted average using the tent filter, the data can be smoothly reconstructed even at level boundaries. The sampler by Wang *et al.* [WWW*19] uses an alternative approach that first determines which neighboring cell octants a rectangular reconstruction filter falls into, and then uses these for reconstruction (the values for the octants themselves can then, e.g., again be computed using tent basis functions).

As these samplers are quite expensive—one first needs to locate each of the constituting cells by traversing a kd-tree or BVH—the *ExaBrick* data structure by Wald *et al.* [WZU*21] first identifies same-level cells and combines them into bricks; to support tent basis functions, the data structure then computes *active brick overlap regions* where the tent basis support regions of the bricks' cells overlap; rays marching through these overlap regions do not need to perform tree traversal to locate the cells therein. The data structure can also be used to adapt the sampling rate. Zellmann *et al.* later proposed extensions to use point location queries [ZSM*22], to support fast data buffer updates [ZWS*22a], and to support volume path tracing [ZWS*22b].

Hybrid AMR *and* unstructured approaches also exist. The work by Shih *et al.* [SZM*14] visualizes data that is unstructured near the mesh boundary of the simulation, and AMR in the off-body domain. The authors approach this using kd-trees and multi-pass rendering with depth peeling.

2.4. Dual Mesh AMR and Stitching

An orthogonal approach to high-quality AMR rendering is to use dual cells; the idea here is to shift the whole cell-centric AMR grid to become the dual grid; grid cells in regions without level transitions then become regular hexahedra, while at level boundaries, generally unstructured elements are introduced as "stitching cells". In 3D, those general elements can have curved faces which need to be approximated with bilinear patches to avoid artifacts.

Stitching algorithms follow simple rule sets that determine which type of boundary cell is generated depending on if we are at the face, edge, or corner of a subgrid, and on the difference in refinement levels there. Earlier stitching algorithms like the one by Weber *et al.* [WKL*01] only supported level transitions with a difference of 1. The algorithm by Moran and Ellsworth [ME11] later proposed an alternative pattern that allowed for arbitrary differences in refinement levels of neighboring subgrids. The paper by Wald [Wal20] proposes an efficient GPU algorithm to snap the original cells to their duals in parallel, resulting in the same pattern as Moran and Ellsworth's. Implicitly obtained dual cells have also been used by Wang *et al.* [WMU*20] to implement a trilinear interpolation scheme for tree-based AMR. Methods that a priori compute the dual mesh can be prohibitive if the data is large, because they map all the original rectangular cells to unstructured elements.

3. Method Overview

We propose to render large AMR data using a hybrid data structure that is based off the dual mesh, which comprises generally unstructured "stitching" elements as well as what we call *gridlets*: small uniform grids of neighboring same-sized, *not twisted* hexahedra cells that, as in *ExaBrick* by Wald *et al.* [WZU*21] are allowed to span the domain of multiple original AMR subgrids. To that end, at least for the stitching elements one can just use any off-the-shelf volume renderer that supports unstructured elements (tetrahedra, pyramids, wedges, and hexahedra with twisted faces); and by adding our gridlets as yet another unstructured element type specialized on this kind of data, any such off-the-shelf volume renderer could be extended to render our data structure.

In this paper we concentrate on sampling-based volume renderers that use bounding volume hierarchies (BVHs) for cell location [WUM*19]. The data structure allows us to efficiently render

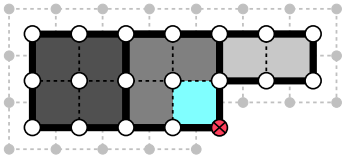


Figure 3: *Three gridlets of size 2×2 dual cells. Vertices/corners (white dots) at gridlet boundaries are duplicated (ghost layers). Gridlets are stored compactly; e.g., the rightmost gridlet consists of 2×1 cells. At level boundaries, where other stitching cells overlap, gridlet corner values can be empty (red dot). Dual cells with one or more empty corners (light blue) are skipped during filtering.*

the AMR data using volumetric path tracing; our path tracing implementation requires spatially varying majorants, which we use to adaptively sample the data using Woodcock tracking [WMHL65]. The steps involved to use our method are the following:

- **Dual mesh and gridlet generation**: we first create the dual mesh of the AMR data set and create gridlets using clustering. This is done in an offline process.
- **Acceleration structure generation**: before rendering, we build a uniform grid with macro cells that span the whole AMR data set and store spatially varying majorants. We also build a sampling BVH using NVIDIA OptiX [PBD*10] to locate stitching cells and gridlets.
- **Rendering**: we use OptiX to implement volumetric path tracing with several features typical for scientific visualization (clipping, transfer functions, etc.) and the high quality interpolant induced by our efficient stitching method.

In the following, we provide a detailed description of those steps, as well as how to efficiently implement them on GPUs with hardware ray tracing cores.

4. Dual Mesh and Gridlet Generation

In a first step we build the dual mesh and split all of its cells into perfect cubes and into stitching cells that fill the gaps at level boundaries. We then cluster the perfect cubes into gridlets of same-size voxels. Gridlets are small Cartesian grids; here we have to be careful because stitching cells and gridlets (but not the actual cells *inside* the gridlets) will generally overlap. As our primary focus in this paper is not on construction time, we do not particularly optimize for this phase and hence implemented it as an offline step. We still note that the general choice of algorithms does lend itself to efficient implementations on GPUs.

4.1. Dual Mesh Generation

To generate the dual mesh we adapt the algorithm by Wald [Wal20] to skip the marching cubes isosurface extraction step and instead write out the dual cells directly; we use the author's publicly available GPU implementation, which can extract the dual mesh of large-scale data sets such as the *NASA Exajet* (656 M hexes) in under ten seconds. Wald's algorithm produces the same stitching cells as Moran and Ellsworth's algorithm [ME11].

We also modify the algorithm to split the resulting unstructured elements into perfect cubes and into all other elements (general tetrahedra, pyramids, and wedges, as well hexahedra that are twisted, have one collapsed edge, or are not uniform in size). For that we use a slight modification to Wald's `doDualCell` algorithm (cf. [Wal20], Sec. 4.4) that snaps the dual cells to their (potentially coarser) real dual cells. Here we keep track of the refinement level of the eight cells to snap, and if all the cells' levels are the same, we know that we are not at a level boundary, or in a stitching region connecting same-level hexahedra and hence the dual cell emitted is a perfect cube.

By temporarily generating a full unstructured mesh from the AMR cells, in the same spirit of highly optimized AMR rendering data structures like *ExaBrick* [WZU*21], we also drop the original (block-structured, octree, etc.) AMR hierarchy. This allows us to build a data structure that is optimized for rendering and that can combine same-level (dual) cells from neighboring AMR subgrids.

4.2. Creating Gridlets

In a next step we combine all perfect cubes (neighboring dual cubes that were known to be cubes to begin with, as well as cubes at stitching boundaries) into *gridlets* using a clustering algorithm. Gridlets are small Cartesian grids that contain dual cells with the same refinement level; hence, the gridlet *cell* size matches the exact size of the respective dual cells on that level.

We construct the gridlets so that the data values are associated with the cell corners; per construction, gridlets have one ghost cell layer where they connect with gridlets of same-level dual cells; a gridlet with $8 \times 8 \times 8$ cells for example will store 9^3 scalars. At the same time, since gridlets and stitching cells are allowed to overlap, gridlets can contain empty cells (see below). See Fig. 3 for an overview of those concepts.

Smaller gridlets will be dominated by ghost layers, but benefit from a finer domain tessellation, whereas bigger gridlets amortize the ghost layers through their size, but contain more empty cells. We use gridlets of size $8 \times 8 \times 8$; we saw negative returns for larger gridlets, both in memory consumption as well as rendering performance.

To construct the gridlets, we first convert the perfect cubes—which are technically still hexahedra represented with eight vertex indices—to *voxels*, which are uniquely identified by their minimum corner in cell coordinates (`int3 lower`) and their refinement level (`int level`). We then sort the voxels to obtain separate lists for each refinement level. (Later, when the voxels are assigned to gridlets, we can represent them even more compactly using a single floating point value.)

For each refinement level, we define "macrocells" that form a *virtual* structured regular grid and span the number of voxels of the intended gridlet size. The macrocells will eventually *become* gridlets if they contain any voxels on the respective refinement level. Voxels are mapped to their macrocells using projection. We project the voxels on each refinement level into their macrocells; since this is done per refinement level, not nearly all macrocells on each level will be active. We lazily *activate* the macrocells (to then become gridlets) only when voxels from the current refinement level project to them. This way, gridlets that are not connected to another gridlet from the same level via a face are (implicitly) shrunk to tightly enclose their voxels, such as the rightmost, light gray gridlet in Fig. 3.

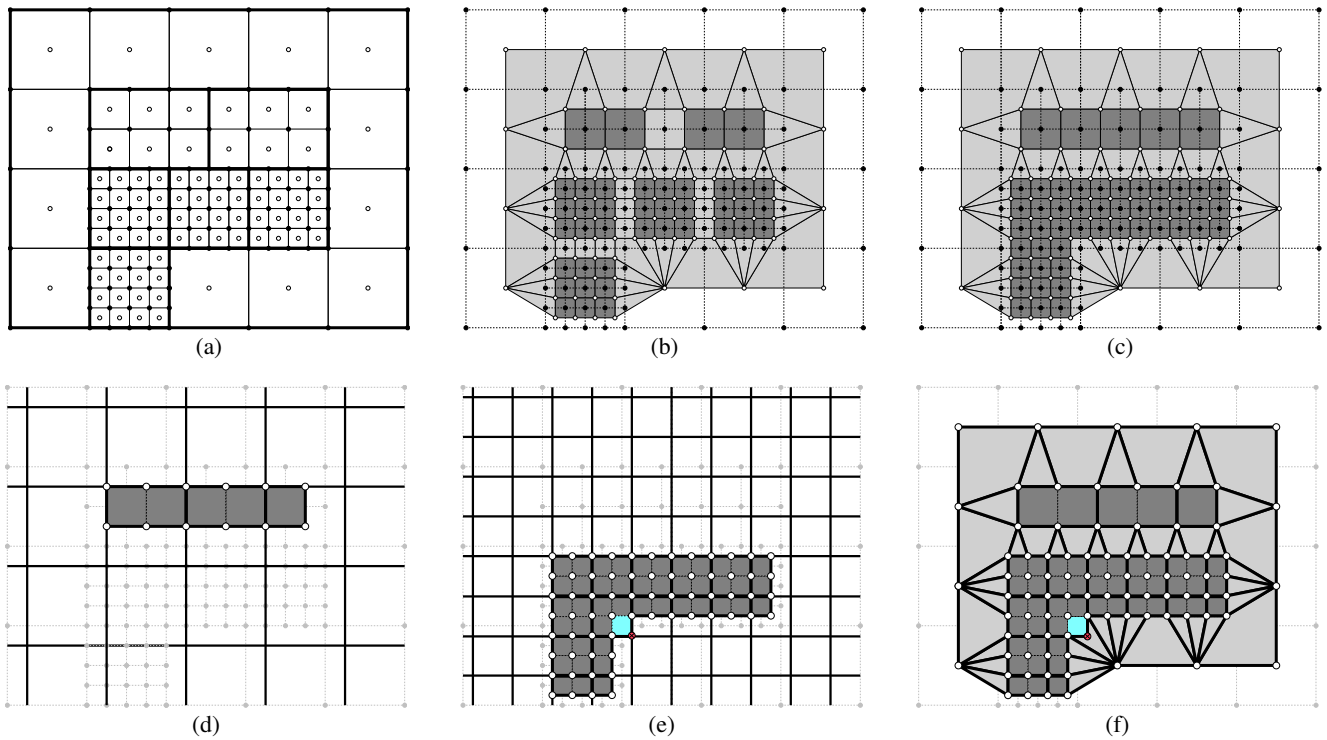


Figure 4: *Illustration of dual mesh and gridlet generation in 2D. (a) the input AMR mesh comprised of four Level-0 subgrids (finest resolution) that form an* L*, two Level-1 subgrids, and one Level-2 subgrid (coarsest resolution) that the other subgrids are embedded into. (b) the corresponding unstructured dual mesh; we first split the unstructured elements into subgrid hexahedra/cubes (dark gray) and stitching elements (light gray). (c) hexahedra elements whose corners have the same refinement level are added to the set of cubes. (d,e) we process the cubes on each refinement level separately and (here) create gridlets of* 2×2 *cells. For that we use a virtual grid that is aligned with the cubes, and activate only those macrocells that contain cubes. We insert empty cells (cf. light blue cell in (e,f)) in concave and cut-out regions where one or more of the macrocell grid's corner vertices does not correspond to a valid dual cell on that level (red corner vertex in (e,f)). Here we obtain three Level-1 gridlets (d) and 16 Level-0 gridlets (e). (f) shows the combined stitching and per-level gridlet geometry.*

Gridlets themselves are represented by their origin, the number of voxels, and a list of scalar IDs to map from voxels to data. Gridlets with $N \times M \times K$ voxels have $(N+1) \times (M+1) \times (K+1)$ scalar IDs. When a gridlet is created, initially all its scalar IDs are set to -1, denoting an empty cell; we later update the scalar IDs as required and leave only those cells empty where no data is available; per construction, empty cells only occur at cut-outs at the corners when the cells form L-shapes, T-shapes, etc.

At the construction stage, the gridlets store scalar IDs and not the data itself, so as to support multi-field data. We later however resolve the indirection when preparing the gridlets for rendering.

The overall process of generating the dual mesh, voxels, and gridlets is summarized in Fig. 4. Pseudo code for gridlet generation is shown in Algorithm 1. The algorithm is executed separately for each refinement level. In the first phase, we project the cubes onto the virtual grid. Cubes get inserted into their macrocells, which get activated if they were not active yet, and have their bounds extended accordingly. During the second phase of the algorithm, we iterate over each active macrocell. The macrocells store references to their cubes, and we now seek to initialize the macrocells' scalar IDs from these. Before we start, we initialize each scalar ID of the macrocell with -1, indicating that the scalar value does not exist. We then iterate over the cubes, and also increment the scalar ID of the macrocell we are manipulating in VTK order (function NEXT in Algorithm 1). At the level boundaries, depending on the spatial arrangement, some of the scalar IDs remain empty (-1) where the gridlet overlaps the unstructured stitching geometry, or gridlets from other refinement levels. A naïve, serial implementation might use a hashmap to store the virtual grid, and lazily create macrocells only when a cube gets inserted. Implementing this algorithm on the GPU using two compute kernels and atomic operations would however also be straightforward.

5. Sampling and Rendering

We focus on rendering using BVH sampling [WUM*19,ZSM*22]; being able to take samples at arbitrary positions before first marching there using a shared-face data structure allows us to implement advanced ray tracing methods, including Woodcock tracking [WMHL65]. Fundamentally, our data structure is also compatible with element marchers, shared face algorithms, or approaches that focus on rasterization though.

We implemented an example renderer with NVIDIA OptiX [PBD*10]. That allows us to make use of hardware ray tracing for sample location; some of the samplers we compare against (cf. Section 6) are also available as OptiX implementations,

Algorithm 1 Generating gridlets per level by projecting the perfect cubes of the dual mesh onto a virtual grid.

```
1:  function MAKEGRIDS(Level,Cubes)
2:      MacroCells[Level] = ∅
3:      for each c ∈ Cubes[Level] do
4:          mc = MacroCells[Level][c]    ▷ project c to virtual grid
5:          mc.ExtendBounds(c.Bounds())
6:          mc.Cubes.Insert(c)
7:      end for
8:
9:      for each mc ∈MacroCells[Level] do
10:         if mc.IsActive() then    ▷ bounds extended at least once
11:             numScalars = mc.Size() + {1,1,1}
12:             mc.ScalarIDs.Resize(numScalars)
13:             mc.ScalarIDs.Set(−1)
14:             mcScalarID = 0              ▷ Current scalar ID of mc
15:             for each c ∈ mc.Cubes do
16:                 for all cScalarID ∈ c.ScalarIDs do
17:                     mc.ScalarIDs[mcScalarID] = cScalarID
18:                     mcScalarID = NEXT(mcScalarID)
19:                 end for
20:             end for
21:         end if
22:     end for
23: end function
```

so implementing ours with OptiX allows us to directly integrate them within the same framework. We aim at implementing a sampler that is simple, i.e., when there is a choice between using a smart low-level optimization and implementation simplicity, we opt for the latter unless the effect of the optimization is significant.

5.1. Cell Location and Sampling

To be able to sample from the data structure, we need to set up BVHs with OptiX. For that we use separate OptiX geometries for the unstructured stitching elements and the gridlets; the stitching geometry stores pointers to the vertices (`float4`, the `w` component encodes the scalar value) and to the element indices (of type `int32_t`). We use an encoding representing every element with exactly eight indices. Elements with fewer indices are padded with −1's. Gridlets have their own geometry; they do not contain IDs or vertices, but instead store lists of scalars—one per cell corner—including empty scalars that are encoded using the special value `NAN` (for "not a number"). We also experimented with using a single geometry per stitching element type to avoid the extra padding indices. For meshes that primarily consist of tets or pyramids, storing those padding indices can be quite wasteful; here we however observed the opposite: peak GPU memory measured with `nvidia-smi` consistently *exceeded* that of the approach with two geometries, by about 1-5% depending on the data set, while rendering performance only differed within measurement error.

Gridlets cannot have shared vertices (only shared *values* due to ghost layers where the cell corners of neighboring gridlets or cells connect, cf. Fig. 4f). Hence, the per-gridlet scalar ID lists from before can be flattened so that the gridlets store the scalars directly. In that sense, gridlets can be thought of as vertex-centric 3D textures, with the special value `NAN` denoting empty texels. For lack of data sets that exceed that limit, we assume that the overall number of cells cannot exceed 2^{31}, so that gridlets can compactly be represented using 32 byte (3× `int32_t` for the gridlet origin, 1× `int32_t` for the refinement level, 3× `int32_t` for the gridlet size/voxel count, and 1× `int32_t` offset into scalar list); plus the voxels/scalars themselves (`numScalars` × `float`).

When building OptiX BVHs we cull invisible elements by classifying them using the alpha transfer function. We also do that for gridlets. Here, the min/max data values are precomputed so we can interactively cull them without traversing all the scalars when the transfer function changes. We once set up OptiX geometries containing all the primitives, and invalidate their bounding boxes before building the OptiX BVH and let OptiX determine that they are empty and will not be included in the BVH; we always cull the elements in this way whenever the transfer function changes, and then fully rebuild the BVHs. Gridlet and unstructured element geometries have their own bottom-level BVHs. We combine those using a top-level BVH, which gives us a sampleable two-level acceleration data structure.

We sample the two-level BVH with zero-length rays and use OptiX intersection programs to implement the element intersection tests. For the stitching geometry, we use an intersection test that supports unstructured elements with bilinear faces that we took from Intel's open source library OpenVKL [Int], and then ported it to CUDA to run on NVIDIA GPUs. Gridlets have their own OptiX intersection program that immediately dispatches to a general sampling function to perform a point lookup and compute the value of the gridlet at the ray origin.

The function first checks if the sampling position falls inside the gridlet's object bounds. Then, the position is transformed to gridlet coordinates using an affine transform based on the gridlet origin and cell size. That gives us the cell that overlaps the sampling position; if either of the scalars at the eight cell corners is empty (i.e., its value is `NAN`), then the gridlet was not intersected. If we encountered an intersection, we use trilinear interpolation to reconstruct the cell's data value. Back in the intersection program, an intersection is reported using OptiX to indicate that we found a hit inside the BVH.

5.2. Woodcock Tracking with Spatially Varying Majorants

Following recent trends in sci-vis, our renderer implements volumetric path tracing; from the numerous algorithms to compute free-flight distance samples that exist, we pick Woodcock tracking [WMHL65] for its simplicity. Fundamentally, our observations should however also apply to algorithms that are for example based on control variates [GMH*19] or use unbiased marching with Taylor series expansion [KDPN21].

Woodcock tracking uses fictitious particles as control variates to homogenize heterogeneous media; using rejection sampling, inside a loop (cf. Algorithm 2), the algorithm probabilistically determines if a sampling interaction is a fictitious (or, "null") collision, in which case we just sample on along the same direction and the loop iteration continues. A real collision triggers a scattering event

Algorithm 2 Woodcock tracking with null collisions.

```
1:  function WOODCOCK(o,ω,μ̄,t_min,t_max)
2:      t = t_min
3:      do
4:          ζ = RAND()
5:          t = t − log(1−ζ)/μ̄
6:          if t ≥ t_max then                ▷ outside region bounds
7:              break
8:          end if
9:          ξ = RAND()
10:     while ξ > μ(o+t∗ω)/μ̄                 ▷ exit loop ⇒ real collision
11:     return t
12: end function
```

and causes the loop to terminate. The smaller the difference between the actual extinction $\mu(x)$, $x \in \mathbb{R}^3$ and the *majorant extinction* $\bar{\mu}$, the fewer times the rejection sampling loop will be called and the procedure will become more efficient. By dividing the volume into regions of space with their own—preferably tight—local majorants $\bar{\mu}_i$, an acceleration structure can be constructed.

In sci-vis volume rendering it is customary to use post-interpolative transfer functions. The majorant density is not directly derived from the underlying scalar field, but is obtained by classifying that with the RGBα transfer function, which can frequently change. We interpret the RGB component of the transfer function as scattering albedo, and the α component as extinction coefficient. Due to that indirection, and since we want to interactively change the transfer function, it is not possible to *directly* compute local majorants; we instead have to subdivide the volume (either spatially or in object space) and compute min/max scalar ranges for the resulting volume regions (cf. min/max trees [KWPH06]); these are later, when the transfer function changes, used to compute the regions' local majorants by iterating and computing the maximum extinction/α value of the transfer function inside that range.

Similar to recent work on volume path tracing in sci-vis [MSG*22, ZWS*22b], we use a uniform grid with majorants as an acceleration structure. We compute the min/max regions offline by projecting each unstructured element and each gridlet cell onto the uniform grid. The majorants are stored in a separate list of floating point values and, as described above, are updated using post-classification when the transfer function changes.

This *min/max grid* allows us to interactively change the transfer function. It would be impractical to iterate over the entire volume all the time to recompute majorants because then this process would become non-interactive. The min/max ranges used as indices into the linear transfer function array help us to speed this procedure up significantly. As we need to store three 32-bit floating point values per cell (the precomputed min/max value ranges *and* the majorants that change on transfer function updates), GPU memory consumption can however become significant when using grids with high resolution. The renderer traverses the grid using the 3D digital differential analyzer algorithm (DDA) [SKTM11], giving us integration ranges $[t_{min}, t_{max}]$ and local majorants $\bar{\mu}_i$ for the ray $\langle o, \omega \rangle$ to pass as arguments to Algorithm 2.

In Algorithm 2, the point extinction $\mu(x)$ is obtained via BVH sampling as detailed in Section 5.1. Quite often, subsequent samples are taken from the same gridlet. For this special element type we implemented a caching optimization. When we encounter a null collision sampling the range $[t_{min}, t_{max}]$, we store the primitive ID of the current gridlet; when we later take another sample inside the same range, we first test if the sample would fall into that gridlet, and only if it does not, we take a point sample from the BVH.

5.3. Path Tracer Implementation

We use the sampling and traversal routines to implement a volume path tracer. Our renderer supports single scattering with shadow rays or multi-scattering phase functions. We use Russian roulette to terminate low-throughput paths. The RGB components of the post-classification transfer function serve as albedo and the alpha components as extinction coefficients. We support omnidirectional dome lights as well as point light sources.

We implement the path tracer in an OptiX ray generation program, which allows us to then trace sampling rays into the element BVH. We traverse the majorant grid and generate a scattering event when a collision was encountered inside the macrocell. Rays without a collision generate a boundary hit event. Their path throughput is optionally weighted by the dome light's contribution, and then their path throughput and intensity are written to an accumulation buffer. The accumulation buffer is periodically converted to sRGB and written to the frame buffer for display.

6. Results and Evaluation

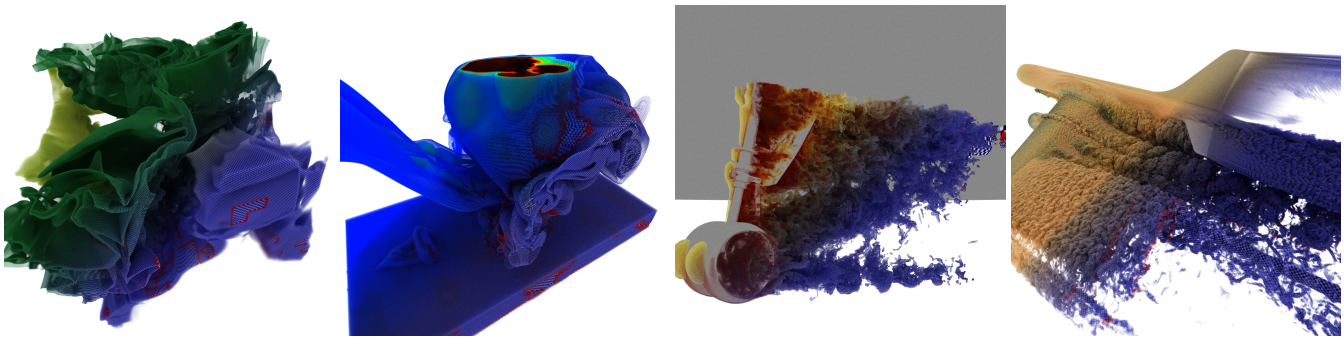
In this section we present construction, memory, and sampling/rendering performance benchmarks. We also compare our method to various state-of-the-art GPU samplers optimized with different objectives in mind. We integrated them into our framework and exercised them on an assortment of data sets that are representative for typical modern AMR simulation codes. We executed the benchmarks on a workstation with an NVIDIA A6000 GPU with 48 GB GDDR memory, an Intel i7-11700KF CPU (8 cores/16 threads), and 64 GB RAM. The system uses Ubuntu Linux 20.04, NVIDIA driver version 510.85.02, CUDA 11.6, and OptiX 7.4.

6.1. Data Sets

We use the four data sets presented below for the evaluation. See Fig. 5 for a visual overview, and Table 1 for data set statistics.

TAC Molecular Cloud. The molecular cloud data set was produced with the "SILCC-Zoom" simulation [SWG*17] that zooms in on highly detailed regions (eight refinement levels across all time steps) of gaseous molecular clouds to understand galaxy formation. The simulation is based on the FLASH code [FOR*00], which generates octrees, but with $16 \times 16 \times 16$ leaf nodes.

LANL Impact. The LANL meteor impact simulation [PST*16] uses the xRage code [GWC*08] and varies heavily over time. We use snapshot $t = 46,112$ of the "temperature in electronvolt (tev)" field variable. The AMR grid is challenging for approaches like *ExaBrick*, which generate lots of single cell bricks for this data set (see the discussion in [ZWS*22a] on challenges with min/max range computation).



(a) (b) (c) (d)

Figure 5: *Data sets used for the evaluation. (a) TAC Molecular Cloud, (b) LANL Impact (t=46,112), (c) NASA Landing Gear, (d) NASA Exajet. Top-left (a-d): direct volume rendering. Bottom-right (a-d): voxels as blue/white checkers, unstructured stitching elements in red.*

		reference method (regular dual mesh) unstructured elements					our method - gridlets w/ boundary stitching unstructured stitching elements					gridlets		
model	surf tris	verts	pyrs	wedges	hexes	Σ elems	verts	pyrs	wedges	hexes	Σ elems	gridlets	full cells	empty cells
TAC Molecular Cloud	n/a	82 M	1.3 M	2.5 M	79.8 M	83.6 M	6.5 M	1.3 M	2.5 M	1.4 M	5.2 M	163 K	112 M	72.0 K
LANL Meteor Impact	n/a	283 M	4.2 M	6.8 M	278 M	289 M	20.5 M	4.2 M	6.8 M	5.8 M	16.8 M	671 K	399 M	22.1 M
NASA Landing Gear	5.56 M	262 M	2.4 M	4.4 M	259 M	266 M	11.3 M	2.4 M	4.4 M	2.3 M	9.1 M	512 K	366 M	90.1 K
NASA Exajet	22.1 M	656 M	2.5 M	4.1 M	652 M	659 M	12.6 M	2.5 M	4.1 M	3.5 M	10.1 M	1.37 M	931 M	13.7 M

Table 1: *Statistics for our data sets, and how that translates to a regular dual mesh solely composed of unstructured elements vs. our representation using gridlets in terms of element/cell count.*

NASA Landing Gear. NASA's landing gear is a large block-structured AMR data set generated with Chombo [CGL*00]. The data set suffers severely from the "teapot in a stadium" problem (see the discussion in [ZWS*22b]). It also comes with a surface mesh consisting of 5.56 M triangles.

NASA Exajet. With over 656 M cells, Exajet by NASA is our largest data set. Exajet is an octree-AMR data set and was simulated with PowerFLOW [Exa98]. We visualize the magnitude of the derived Lambda2 vorticity field. The data set includes a highly detailed surface mesh consisting of 22.1 M triangles.

Our intention is to cover a wide range of data sets that capture different characteristics; e.g., we expect block-structured AMR data sets to contain fewer boundary stitching elements and thus more perfect cubes than tree-based AMR. The complex spatial arrangement of LANL Impact and Exajet results in magnitudes more empty gridlet cells than the other two data sets (cf. Table 1); it will be interesting to explore if this is as challenging for our renderer as the single cell bricks are for *ExaBrick* [ZWS*22a].

6.2. Comparison to Existing Methods

We compare our method against a number of state-of-the-art unstructured mesh samplers. The sampler by Wald *et al.* [WUM*19] serves as a reference that treats the dual cells as an unstructured mesh and uses an OptiX sampling BVH without any optimizations. We implement this by using our optimized sampler, but bypass the gridlet generation and treat cubes as ordinary hexahedra. We expect that the memory overhead will be severe for the larger data sets. While software systems or middleware like VTK/ParaView [AGL05] or VisIt [Chi12] use rasterization approaches, cell-projection, etc., to render unstructured meshes, and shared-face data structures will replace the BVH, we believe their memory requirements to be on a similar order as our reference.

Wald *et al.*'s [WMZ22] sampler represents what we consider to be the state-of-the-art for sampling very large unstructured meshes such as NASA's Mars Lander on GPUs. It comprises a combination of several optimizations: multi-branching BVH with a branching factor of eight children per node obtained by collapsing an initial binary BVH; multi-node encoding with 16-bit quantization for the node bounding box; and a tree optimization that collapses all subtrees with at least eight leaves into what the authors call a *multi-leaf*, which is itself realized using an OptiX BVH.

Morrical *et al.*'s [MSG*22] sampler also employs memory optimizations, but primarily focuses on build performance. It first sorts the elements on a Hilbert curve and then trivially groups N consecutive primitives into what the authors call *leaf clusters*. The choice of N can have a huge impact on performance, because the intersection test performs a linear search over the primitives. For our tests, we set $N = 4$. The authors also reindex the mesh, which allows them to use element indices with fewer bytes. We include this sampler because it deliberately trades sampling for build performance.

All samplers discussed so far—including ours—will produce the same images. For Wald *et al.*'s [WMZ22] and Morrical *et al.*'s [MSG*22] we use CUDA and OptiX 7 implementations provided by the authors; the sampler by Wald *et al.* originally used a planar intersection test that can produce artifacts. We therefore augmented all samplers to use the bilinear patch intersection test to sample general pyramids, wedges, and hexahedra.

We also compare against Wald *et al.*'s *ExaBrick* [WZU*21], which is optimized for AMR rendering, but uses the original and not the dual mesh. This method does not need to sample individual unstructured elements, nor does it require Newton-Raphson refinement for bilinear faces. We expect sampling performance to be unrivaled because even the boundary cells are voxels. This sampler

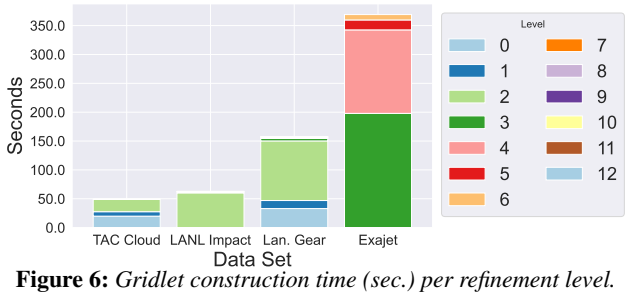


Figure 6: *Gridlet construction time (sec.) per refinement level.*

Model	Reference Perfect cubes	Ours Gridlets+voxels	Compression rate
TAC Molecular Cloud	3.46 GB	432 MB	1 : 8.2
LANL Impact	12.0 GB	1.59 GB	1 : 7.5
NASA Landing Gear	11.4 GB	1.38 GB	1 : 8.3
NASA Exajet	28.9 GB	3.56 GB	1 : 8.1

Table 2: *Compression rate (cubes/voxels only), based on data from Table 1, counting the vertices bound in perfect cubes, vs. gridlets of size* 8^3 *and data layouts using* 16 *byte / vertex+scalar,* 8×4 *bytes / hexahedron,* 32 *byte / gridlet, and* 4 *byte / voxel.*

will generate slightly different results than the others because of the different reconstruction algorithm.

We integrated all four samplers into our framework. All samplers use a min/max grid to obtain majorants (cf. Section 5.2). The unstructured element samplers—including ours—use the exact same CUDA routines to build the grid. Gridlets are special elements that are projected to the grid cell by cell. For *ExaBrick* we use the combination "active brick regions for sampling and grid+DDA majorant traversal", as described by Zellmann *et al.* [ZWS*22b]; we project the bounding box of each AMR cell's integration domain onto the grid to compute min/max values and majorants. The result of that is comparable to, yet not exactly the same as the dual mesh majorants. With this setup, we can create equivalent images using all five methods and directly compare their performance.

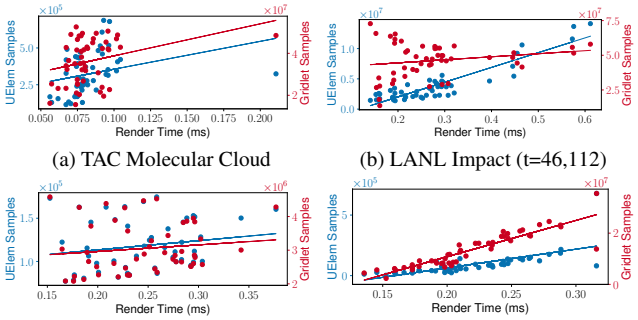
6.3. Gridlet Construction Time

We use a naïve CPU implementation of Algorithm 1 that is single-threaded and uses a C++ `std::map` for the macrocells. We made several design choices that favor simplicity over performance, so that our results are dominated by file I/O and unoptimized write accesses. In Fig. 6 we report the total construction time per AMR refinement level. We achieve between 50 seconds and six minutes for the four data sets and are confident that future work on parallel gridlet construction would allow for even higher rates.

6.4. Memory Consumption

To analyze GPU memory consumption, we first report the compression rate obtained by replacing all cubes with gridlets of 8^3 voxels (9^3 scalars) in Table 2. Encouragingly, the compression rate is on the order of 1 : 8 for all our data sets. For the more complex *LANL Impact* and *NASA Exajet* data sets, we observe compression rates that are slightly lower, yet still on the same order.

More detailed results can be found in Table 3. We report results



(a) TAC Molecular Cloud (b) LANL Impact (t=46,112)

(c) NASA Landing Gear (d) NASA Exajet

Figure 7: *Rendering performance vs. number of uelem and gridlet samples, measured over a 50 position spherical camera orbit.*

for manually summing all the data in buffers ("data"), but excluding acceleration data structures, framebuffers, min/max grids, majorants, etc. as a theoretical metric. We also report peak memory consumption ("peak") obtained via sampling with the `nvidia-smi` tool; the "total" benchmark presents what `nvidia-smi` reports during rendering, after all acceleration data structures have been fully built.

Our data structure easily beats all the unstructured mesh samplers in GPU memory. The reference method runs out of memory allocating storage buffers for OptiX BVHs for all but the *TAC Molecular Cloud* data sets. Only Wald *et al.*'s sampler [WMZ22] can accommodate all four data sets, but still consumes far more memory than our equivalent representation with gridlet compression. All the benchmarks but Morrical *et al.*'s [MSG*22] assume their maximum memory consumption when building OptiX BVHs. Morrical's peaks out during meshlet generation, the reason for which being that Morrical's is the only sampler where the whole construction happens on the GPU.

Another interesting observation is that we beat *ExaBrick* in peak memory consumption for the two complex data sets *LANL Impact* (2.3× better) and *NASA Exajet* (1.9× better). The root cause of this is that a BVH over gridlets is per construction more shallow than a BVH over single cell bricks (or the even finer "active brick overlap regions", cf. [ZWS*22a]), which also seems to positively affect temporary memory consumption during BVH builds. This is encouraging because to build OptiX BVHs the memory buffers need to be on the device and cannot reside in host or managed memory. This means that we can render large data sets on lower-end GPUs than it was possible with *ExaBrick*.

6.5. Rendering Performance

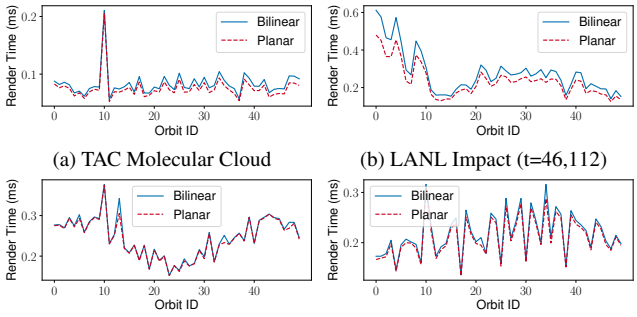
We compare the samplers' performance in Table 4 using the representative views shown in Fig. 5 and report frames per second for single convergence frames. We use multi-scattering with an isotropic phase function, a dome light with uniform intensity, and render viewports of 1024×1024 pixels. We observe that our performance is superior to that of the other unstructured mesh renderers. As expected, we cannot outperform *ExaBrick* in sampling performance, yet we note that the performance is of the same order. We also report performance numbers in Fig. 7 measured over 50 positions of a spherical camera orbit and the same viewport size

Model	Reference			Quick Clusters [MSG*22]			Compressed [WMZ22]			*ExaBrick* [ZWS*22b]			Ours		
	data	peak	total	data	peak	total	data	peak	total	data	peak	total	data	peak	total
TAC Molecular Cloud	3.71	17.7	6.83	2.65	6.40	5.17	3.14	4.62	4.62	0.34	1.70	1.60	0.68	2.36	2.16
LANL Meteor Impact	12.8	(oom)	(oom)	9.15	24.6	14.4	10.9	15.6	13.0	2.93	12.1	5.89	2.39	5.15	4.24
NASA Landing Gear	11.9	(oom)	(oom)	8.49	22.7	16.3	10.1	15.4	15.4	1.03	5.64	5.59	1.85	7.20	6.72
NASA Exajet	30.5	(oom)	(oom)	(oom)	(oom)	(oom)	25.6	36.8	33.8	5.92	21.0	12.6	5.09	11.0	10.5

Table 3: *Memory performance benchmark, in GB. We manually compute GPU memory (*data*), and measure* peak *and* total *memory (i.e., after OptiX BVH construction) using* `nvidia-smi`*. (*oom *means the process ran out of GPU memory before we could perform the measurement.)*

Model	Ref.	Quick Clusters [MSG*22]	Compressed [WMZ22]	*ExaBrick* [ZWS*22b]	Ours
TAC Mol. Cloud	10.3	2.29	4.63	22.7	18.7
LANL Impact	—	0.60	3.04	6.09	3.86
NASA Lan. Gear	—	0.52	0.66	3.26	1.48
NASA Exajet	—	—	1.58	3.67	2.09

Table 4: *Rendering performance, multi-scattered path tracing (cf. Fig. 5), in frames per second, for a single convergence frame.*



(a) TAC Molecular Cloud (b) LANL Impact (t=46,112)

(c) NASA Landing Gear (d) NASA Exajet

Figure 8: *Rendering performance comparison, for bilinear vs. a less accurate planar face stitching element intersection test.*

and rendering modalities. We put the respective rendering times in contrast to the absolute number of unstructured stitching elements vs. gridlets sampled. This benchmark also provides a render time envelope for a typical explorative session.

Unstructured mesh renderers often only support elements with planar faces, while the dual mesh contains bilinear elements. Using a planar intersection test instead of the more accurate one results in subtle rendering artifacts, but might better reflect what integration with an off-the-shelf renderer would look like. We present a comparison in Fig. 8 using the same spherical camera orbit, for our renderer, vs. a variant that uses a planar face intersection test for the stitching elements, for the 50 spherical camera orbits from before.

7. Summary and Discussion

We presented an approach that seamlessly extends unstructured mesh renderers to support very large AMR data sets. Although our method might seem like an improvement for unstructured meshes, it is still specific to AMR, because generally unstructured meshes often do not contain any perfect hexehedra at all. We therefore *by design* opted for our gridlet type to be an extension to unstructured renderers. Rendering unstructured meshes without cubes has no additional runtime overhead because of the separate OptiX geometries for gridlets and other element types. Besides, this also applies to rasterization-based renderers as found in VTK or VisIt, where using a separate gridlet shader would result in marginal, if any overhead compared to an implementation without gridlets.

Gridlets are similar to *ExaBrick*'s representation of same-level cells. *ExaBrick*'s active brick regions however can generate many small or single-cell boxes for complex data sets, while conversely, gridlets contain empty cells and duplicate scalars. One might assume that *ExaBrick* should outperform gridlets in memory because of that. However, we found that whatever memory savings are realized by *ExaBrick* avoiding duplicates, our gridlets make up for in BVH memory size, which also affects peak memory consumption. This actually allows us to render the larger data sets on GPUs with less memory. In the future, it would be interesting to replace *ExaBrick*'s internal data structure with gridlets. While *ExaBrick* still slightly outperforms our dual mesh sampler in rendering time, our tests indicate that this is not due to the shallower BVH, but due to the additional element types we have to test against.

7.1. Sampling on the Original vs. the Dual Mesh

Our data structure is optimized for efficient dual mesh sampling. We now discuss why one would prefer this to sampling on the original AMR grid in the first place. One motivation from a software engineer's perspective is that our method easily integrates with existing unstructured renders. Another reason for sampling on the dual mesh is its interpolation property. The basis [WBUK17] and octant [WWW*19] methods do not have that property. Exact reconstruction at the known data points is desirable in scientific visualization; deviations can impact both color and shape of reconstructed features. While Wang *et al.* [WMU*20] showcase examples where stitching reduces artifacts, we note that all the reconstruction methods discussed in this paper—including the dual mesh interpolator—are C^0 continuous so that, e.g., gradient-based normals can exhibit objectionable artifacts at level boundaries. The dual mesh interpolator represents the data more *faithfully* than the other methods, but does not necessarily reduce artifacts.

8. Conclusion

In this paper we contributed an efficient sampler for AMR dual mesh rendering that adds to the current state of the art in large-scale volume rendering. The sampler optimizes for GPU memory consumption, but also outperforms other unstructured mesh renderers on volume path tracing. A major advantage over its competitors is that it seamlessly integrates with existing unstructured mesh renderers by just adding an additional element type. That allows us to render the same large-scale AMR simulation data sets on GPUs that could previously only be rendered on CPUs, or with highly optimized data structures that use less faithful reconstruction methods. Our method presents a very versatile way to render such data on GPUs, and while the rendering performance is on the same order, yet a little lower than those highly optimized data structures, we beat them in peak memory performance by a significant margin.

Acknowledgments

This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—grant no. 456842964. The Landing Gear was graciously provided by Michael Barad, Cetin Kiris and Pat Moran of NASA. The Exajet was made available by Exa GmbH and Pat Moran. The TAC Molecular Cloud is courtesy of Daniel Seifried. We also express our gratitude to NVIDIA, who kindly provided us with hardware we used for the evaluation.

References

[AGL05] AHRENS J., GEVECI B., LAW C.: *ParaView: An End-User Tool for Large Data Visualization*. Visualization Handbook. Elsevier, 2005. 2, 8

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

[BO84] BERGER M. J., OLIGER J.: Adaptive Mesh Refinement for Hyperbolic Partial Differential Equations. *Journal of Computational Physics 53* (Mar. 1984), 484–512. doi:10.1016/0021-9991(84)90073-1. 1

[CGL*00] COLELLA P., GRAVES D., LIGOCKI T., MARTIN D., MODIANO D., SERAFINI D., VAN STRAALEN B.: Chombo Software Package for AMR Applications Design Document, 2000. 8

[Chi12] CHILDS H.: *VisIt: An End-User Tool for Visualizing and Analyzing Very Large Data*. High Performance Visualization. Chapman and Hall/CRC, 2012. doi:https://doi.org/10.1201/b12985. 2, 8

[Exa98] PowerFLOW User's Guide 3.0, 1998. 8

[FOR*00] FRYXELL B., OLSON K., RICKER P., TIMMES F. X., ZINGALE M., LAMB D. Q., MACNEICE P., ROSNER R., TRURAN J. W., TUFO H.: FLASH: An Adaptive Mesh Hydrodynamics Code for Modeling Astrophysical Thermonuclear Flashes. *The Astrophysical Journal Supplement Series* (2000). 7

[GMH*19] GEORGIEV I., MISSO Z., HACHISUKA T., NOWROUZEZAHRAI D., KŘIVÁNEK J., JAROSZ W.: Integral formulations of volumetric transmittance. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia) 38*, 6 (Nov. 2019). 2, 6

[GWC*08] GITTINGS M., WEAVER R., CLOVER M., BETLACH T., BYRNE N., COKER R., DENDY E., HUECKSTAEDT R., NEW K., R.OAKES W., RANTA D., STEFAN R.: The RAGE Radiation-Hydrodynamic Code. *Computational Science & Discovery* (2008). 7

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

[Int] INTEL CORPORATION: High Performance Volume Kernels. Available at https://www.openvkl.org/, Accessed: 11 November 2022. 6

[KDPN21] KETTUNEN M., D'EON E., PANTALEONI J., NOVÁK J.: An unbiased ray-marching transmittance estimator. *ACM Trans. Graph. 40*, 4 (jul 2021). URL: https://doi.org/10.1145/3450626.3459937, doi:10.1145/3450626.3459937. 2, 6

[KWAH06] KÄHLER R., WISE J., ABEL T., HEGE H.-C.: GPU-Assisted Raycasting for Cosmological Adaptive Mesh Refinement Simulations. In *Volume Graphics* (2006). 3

[KWPH06] KNOLL A., WALD I., PARKER S., HANSEN C.: Interactive isosurface ray tracing of large octree volumes. In *2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 115–124. doi:10.1109/RT.2006.280222. 7

[Lev88] LEVOY M.: Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications 8*, 3 (1988), 29–37. 2

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

[MDV09] MARCHESIN S., DE VERDIERE G. C.: High-Quality, Semi-Analytical Volume Rendering for AMR Data. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1611–1618. doi:10.1109/TVCG.2009.149. 3

[ME11] MORAN P., ELLSWORTH D.: Visualization of amr data with multi-level dual-mesh interpolation. *IEEE Transactions on Visualization and Computer Graphics 17*, 12 (2011), 1862–1871. doi:10.1109/TVCG.2011.252. 2, 3, 4

[MHDG11] MUIGG P., HADWIGER M., DOLEISCH H., GRÖLLER E.: Interactive Volume Visualization of General Polyhedral Grids. *IEEE Transactions on Visualization and Computer Graphics* (2011). 3

[MSG*22] MORRICAL N., SAHISTAN A., GÜDÜKBAY U., WALD I., PASCUCCI V.: Quick Clusters: A GPU-Parallel Partitioning for Efficient Path Tracing of Unstructured Volumetric Grids. In *2022 IEEE Visualization Conference (VIS)* (2022). 2, 3, 7, 8, 9, 10

[MUWP19] MORRICAL N., USHER W., WALD I., PASCUCCI V.: Efficient space skipping and adaptive sampling of unstructured volumes using hardware accelerated ray tracing. In *Proceedings of IEEE Visualization* (2019), VIS '19, pp. 256–260. 3

[NSJ14] NOVÁK J., SELLE A., JAROSZ W.: Residual ratio tracking for estimating attenuation in participating media. *ACM Trans. Graph. 33*, 6 (nov 2014). URL: https://doi.org/10.1145/2661229.2661292, doi:10.1145/2661229.2661292. 2

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A.: OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH)* (2010). 4, 5

[PST*16] PATCHETT J. M., SAMSEL F. J., TSAI K. C., GISLER G. R., ROGERS D. H., ABRAM G. D., TURTON T. L.: *Visualization and Analysis of Threats from Asteroid Ocean Impacts*. Tech. rep., Los Alamos National Laboratory, 2016. 7

[RWCB15] RATHKE B., WALD I., CHIU K., BROWNLEE C.: SIMD Parallel Ray Tracing of Homogeneous Polyhedral Grids. In *Eurographics Symposium on Parallel Graphics and Visualization* (2015), Dachsbacher C., Navrátil P., (Eds.), The Eurographics Association. doi:10.2312/pgv.20151153. 3

[SCCB05] SILVA C., COMBA J., CALLAHAN S., BERNARDON F.: A Survey of GPU-Based Volume Rendering of Unstructured Grids. *Brazilian Journal of Theoretic and Applied Computing 12*, 2 (2005), 9–29. 3

[SDM*21] SAHISTAN A., DEMIRCI S., MORRICAL N., ZELLMANN S., AMAN A., WALD I., GÜDÜKBAY U.: Ray-traced shell traversal of tetrahedral meshes for direct volume visualization. In *Proceedings of the IEEE Visualization Conference-Short Papers* (2021), VIS '21. 3

[SDW*22] SAHISTAN A., DEMIRCI S., WALD I., ZELLMANN S., BARBOSA J., MORRICAL N., GÜDÜKBAY U.: GPU-based Data-parallel Rendering of Large, Unstructured, and Non-convexly Partitioned Data, 2022. URL: https://arxiv.org/abs/2209.14537, doi:10.48550/ARXIV.2209.14537. 3

[SKTM11] SZIRMAY-KALOS L., TÓTH B., MAGDICS M.: Free path sampling in high resolution inhomogeneous participating media. *Computer Graphics Forum 30* (2011). 2, 7

[SWG*17] SEIFRIED D., WALCH S., GIRICHIDIS P., NAAB T., WÜNSCH R., KLESSEN R. S., GLOVER S. C. O., PETERS T., CLARK P.: SILCC-Zoom: the dynamic and chemical evolution of molecular clouds. *Monthly Notices of the Royal Astronomical Society 472*, 4 (Dec 2017), 4797–4818. arXiv:1704.06487, doi:10.1093/mnras/stx2343. 7

[SZM*14] SHIH M., ZHANG Y., MA K.-L., SITARAMAN J., MAVRIPLIS D.: Out-of-core visualization of time-varying hybrid-grid volume data. In *2014 IEEE 4th Symposium on Large*

Data Analysis and Visualization (LDAV) (Los Alamitos, CA, USA, nov 2014), IEEE Computer Society, pp. 93–100. URL: https://doi.ieeecomputersociety.org/10.1109/LDAV.2014.7013209, doi:10.1109/LDAV.2014.7013209. 3

[Wal20] WALD I.: A simple, general, and GPU friendly method for computing dual mesh and iso-surfaces of adaptive mesh refinement (AMR) data, 2020. URL: https://arxiv.org/abs/2004.08475, doi:10.48550/ARXIV.2004.08475. 3, 4

[WBUK17] WALD I., BROWNLEE C., USHER W., KNOLL A.: CPU Volume Rendering of Adaptive Mesh Refinement Data. In *SIGGRAPH Asia 2017 Symposium on Visualization* (2017). doi:10.1145/3139295.3139305. 3, 10

[WKL*01] WEBER G. H., KREYLOS O., LIGOCKI T. J., SHALF J. M., HAGEN H., HAMANN B., JOY K., MA K.-L.: High-Quality Volume Rendering of Adaptive Mesh Refinement Data. In *Proceedings of the Vision Modeling and Visualization Conference 2001* (2001). 3

[WMHL65] WOODCOCK E. R., MURPHY T., HEMMINGS P. J., LONGWORTH T. C.: Techniques used in the GEM code for Monte Carlo neutronics calculations in reactors and other systems of complex geometry. In *Proceedings of the Conference on Applications of Computing Methods to Reactor Problems* (1965), Argonne National Laboratory. 2, 4, 5, 6

[WMU*20] WANG F., MARSHAK N., USHER W., BURSTEDDE C., KNOLL A., HEISTER T., JOHNSON C. R.: CPU ray tracing of tree-based adaptive mesh refinement data. In *Computer Graphics Forum* (2020), vol. 39, pp. 1–12. 3, 10

[WMZ22] WALD I., MORRICAL N., ZELLMANN S.: A memory efficient encoding for ray tracing large unstructured data. *IEEE Transactions on Visualization and Computer Graphics 28*, 1 (2022), 583–592. 3, 8, 9, 10

[WUM*19] WALD I., USHER W., MORRICAL N., LEDIAEV L., PASCUCCI V.: RTX beyond ray tracing: Exploring the use of hardware ray tracing cores for tet-mesh point location. In *Proceedings of High-Performance Graphics - Short Papers* (2019), HPG '19. 3, 5, 8

[WWW*19] WANG F., WALD I., WU Q., USHER W., JOHNSON C. R.: CPU Isosurface Ray Tracing of Adaptive Mesh Refinement Data. *IEEE Transactions on Visualization and Computer Graphics* (2019). 3, 10

[WZU*21] WALD I., ZELLMANN S., USHER W., MORRICAL N., LANG U., PASCUCCI V.: Ray tracing structured AMR data using ExaBricks. *IEEE Transactions on Visualization and Computer Graphics 27*, 2 (2021), 625–634. 3, 4, 8

[XTC*22] XU J., THEVENON G., CHABAT T., MCCORMICK M., LI F., BIRDSONG T., MARTIN K., LEE Y., AYLWARD S.: Interactive, in-browser cinematic volume rendering of medical images. *Computer Methods in Biomechanics and Biomedical Engineering: Imaging & Visualization 0*, 0 (2022), 1–8. doi:10.1080/21681163.2022.2145239. 2

[YIC*10] YUE Y., IWASAKI K., CHEN B.-Y., DOBASHI Y., NISHITA T.: Unbiased, adaptive stochastic sampling for rendering inhomogeneous participating media. *ACM Transactions on Graphics 29*, 6 (2010), 177, 8 pages. 2

[ZSM*22] ZELLMANN S., SEIFRIED D., MORRICAL N., WALD I., USHER W., LAW-SMITH J., WALCH-GASSNER S., HINKENJANN A.: Point containment queries on ray tracing cores for AMR flow visualization. *Computing in Science Engineering* (2022), 1–1. doi:10.1109/MCSE.2022.3153677. 3, 5

[ZWS*22a] ZELLMANN S., WALD I., SAHISTAN A., HELLMANN M., USHER W.: Design and Evaluation of a GPU Streaming Framework for Visualizing Time-Varying AMR Data. In *Eurographics Symposium on Parallel Graphics and Visualization* (2022), Bujack R., Tierny J., Sadlo F., (Eds.), The Eurographics Association. doi:10.2312/pgv.20221066. 3, 7, 8, 9

[ZWS*22b] ZELLMANN S., WU Q., SAHISTAN A., MA K.-L., WALD I.: Beyond ExaBricks: GPU Volume Path Tracing of AMR Data, 2022. arXiv:2211.09997. 2, 3, 7, 8, 9, 10