Computers & Graphics 138 (2026) 104638

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Shape Modeling

Direct volume rendering of tree-based tetrahedral adaptive mesh refinement data

Musa Ege Ünalan [a,1], Serkan Demirci [a,1], Stefan Zellmann [b], Uğur Güdükbay [a,*]

[a] Department of Computer Engineering, Bilkent University, Ankara, Türkiye
[b] Institute of Computer Science, University of Cologne, Cologne, Germany

ARTICLE INFO

Dataset link: https://github.com/Bilkent-ModVis/tet-amr-rendering

MSC:
65D18

Keywords:
Direct volume rendering
Ray tracing
Delta tracking
Acceleration structure
Adaptive mesh refinement

ABSTRACT

Tree-based tetrahedral AMR data (Tet-AMR) combines the benefits of unstructured meshes and AMR by maintaining both a coarse unstructured tetrahedral mesh and a forest of refinement trees, each rooted at a coarse mesh element. Tet-AMR data can be visualized by combining all leaf tetrahedra in the refinement trees into a single unstructured mesh; however, this approach significantly increases the memory usage. We propose a volume rendering method that leverages the regular subdivision of refinement trees by storing only the coarse elements. We construct a bounding volume hierarchy over the coarse mesh elements to efficiently locate the refinement tree to sample. We then generate the geometry of the finer-level elements on the fly as we traverse the refinement tree to find the leaf element. We further demonstrate that the tree structure can be used to implement a dynamic, view-dependent level-of-detail effect, thereby improving performance by reducing fidelity in less impactful regions. Additionally, we enhance rendering performance by utilizing the coarse mesh as both a space-skipping structure for ray marching and a macrocell structure for Woodcock renderers.

1. Introduction

Direct volume rendering for scientific visualization is a method that is used across many fields, from fluid simulations to medical imaging, to create visual representations of volume data and allow a correct and easy interpretation of the data, by providing a model that simulates the various properties of light inside a volume, and is also easily modifiable through the editing of transfer functions and camera parameters.

Volume data can come in various flavors, the most popular being structured grids, unstructured meshes, and Adaptive Mesh Refinement (AMR) [1] data. Unstructured meshes are often used in simulations for their ability to represent complex geometries, whereas AMR is preferred for dynamically and locally refining resolution where accuracy is needed.

Tetrahedral AMR meshes (Tet-AMR) have been proposed to combine the benefits of unstructured meshes and AMR data representations for simulations. They begin with a coarse, unstructured mesh that conforms to the complex simulation domain. Each coarse tetrahedron can then be recursively refined, where higher accuracy is needed during the simulation. The resulting refinements form a tree associated with that coarse tetrahedron. Despite their advantages, tetrahedral AMR meshes are more challenging to manage.

Although methods exist for rendering AMR and unstructured meshes, no techniques have been proposed specifically for rendering tetrahedral AMR meshes. A straightforward approach is to flatten the Tet-AMR hierarchy into an unstructured mesh by extracting the geometry of all leaf-level tetrahedra. However, explicitly storing these fully refined elements results in a significant increase in the memory usage, making this approach impractical for rendering large meshes.

To mitigate memory overhead in hierarchical subdivision, Greiner and Grosso [2] proposed computing irregular transition elements on demand during traversal rather than storing them explicitly. Similarly, instead of converting the Tet-AMR into an unstructured mesh, we propose a volume rendering method that stores only the coarse unstructured tetrahedral mesh and represents refined elements implicitly through the refinement hierarchy. During the rendering, the geometry of finer-level tetrahedra is generated on demand by traversing the refinement trees to locate the appropriate leaf element for sampling. We propose two refinement tree traversal methods for the on-the-fly traversal and geometry generation. Additionally, we utilize hardware-accelerated point location on the coarse, unstructured tetrahedral mesh to efficiently identify the corresponding refinement trees. Using a refinement tree traversal and coarse-mesh point location, we implement two volume rendering techniques: a ray tracer and a Woodcock tracker.

* Corresponding author.
E-mail address: gudukbay@cs.bilkent.edu.tr (U. Güdükbay).
[1] Joint first authors.

https://doi.org/10.1016/j.cag.2026.104638
Received 7 February 2026; Received in revised form 22 May 2026; Accepted 22 May 2026
Available online 1 June 2026

We show that a macrocell grid can be constructed by treating each coarse mesh element as a macrocell, storing the minimum and maximum post-classification density values within it, and using empty-space skipping and adaptive free-path sampling for Woodcock tracking. Lastly, we demonstrate that the Tet-AMR traversal can be modified to implement a dynamic screen-space level of detail (LOD) effect, which adjusts the traversal depth based on the screen-space size of the elements. This modification enables us to improve performance by reducing fidelity in regions that have minimal impact on the overall image quality. Our contributions are as follows.

- a volume rendering method for tree-based tetrahedral AMR data,
- two refinement tree traversal methods for on-the-fly geometry generation during rendering,
- a coarse mesh macrocell structure and a dynamic screen-space level-of-detail scheme to improve performance in Tet-AMR rendering.

2. Background and related work

We provide an overview of the volume data formats commonly used for volume rendering, their structure, and the methods employed to sample and render them. We also briefly introduce volume rendering for scientific visualization and the two volume rendering methods proposed in this work.

2.1. Volume data formats

Volume data refers to functions that map the spatial domain to scalars (such as density or temperature), vectors (such as velocity), or multi-component values. These continuous functions are captured through discrete samples distributed throughout the domain. The data used in volume rendering can be classified by topology, such as structured grids, unstructured meshes, and AMR. The method used is generally not an arbitrary choice, but a direct consequence of how data was acquired, such as medical imaging and microscopy producing structured grids, numerical simulations generating AMR meshes, and finite element methods with unstructured meshes [1].

The way these volumes store their scalar data can also differ: some store it at the centers of their elements as cell-centered scalars, while others store it at the vertices of their cells as vertex-centered scalars. Generally, simulation and imaging sensors produce cell-centered data. Directly visualizing cell-centered data with interpolation requires locating the sampled cell and its neighbors. Visualization frameworks like VTK [3], Paraview [4], or Visit [5], on the other hand, do not support interpolation of cell-centric data; they either render the cells with uniform, non-interpolated values or they convert the data to a vertex-centric representation by constructing a dual mesh, which in turn increases memory usage. When the data is vertex-centered, rendering the data involves identifying the cell containing the sample points and interpolating the scalar values at its vertices. After the containing cell is determined, the scalar values are interpolated with respect to the geometry of that cell.

2.1.1. Structured and voxel grids

Structured grid volumes consist of cells on a Cartesian grid, featuring uniform data resolution throughout the volume. Since the data is laid out on a grid with a known constant resolution, the grid structure is implicit [6] without requiring any external data structures. The data from the eight neighboring cells can be tri-linearly interpolated to create the value at the sample point. The primary drawback of this method is that empty regions of the volume still require storage. When we need to increase the resolution of data in a particular region, we must also increase the global grid resolution, which can result in unnecessarily large grids.

Recent work on structured grids [1] focuses on skipping empty space, which requires recomputation whenever the transfer function changes. Wald et al. [7] build a macrocell grid over the data, and utilize a bounding volume hierarchy (BVH). A macrocell acts as a low-resolution grid (usually a structured grid) storing maximum alpha values to be used for performance optimization. Volume renderers march the macrocells one by one and skip regions when the maximum opacity is close to zero, whereas Woodcock tracking uses these macrocells to provide localized majorants, reducing null collisions and improving free-path sampling efficiency. Zellmann et al. [8] propose parallel (re)construction of k-d trees for skipping with sparse volumes.

Semistructured grids bridge the gap between pure voxel grids and structured AMR, using formats such as sparse, shallow trees like NanoVDB [9] or primal–dual rectilinear grids [10], where data is stored simultaneously at both vertices and cell centers. For blocked multiresolution data, other methods enable direct, smooth interpolation across arbitrary levels of detail without padding [11]. Additionally, Wang et al. [12] compress structured volumes by tiling them into a forest of shallow "Bricktrees" to avoid the overhead of a single deep hierarchy.

2.1.2. Unstructured grids

Unstructured grids typically consist of tetrahedral elements, but can also be comprised of other polyhedral types such as hexahedra, pyramids, and wedges [1]. Unlike structured grids, sampling unstructured grids is more challenging because they lack an implicit structure. Sampling unstructured meshes at a point in space involves identifying the element that contains the sample. There are two main approaches for finding the containing element: element-marching and point-location methods. Element-marching methods traverse the mesh one element at a time, using connectivity information to locate the containing element. On the other hand, point-location methods use spatial acceleration structures to directly query the element that contains the point.

Recent work on element-marching methods focuses on improving the efficiency of element traversal. Maria et al. [13] propose a fast tetrahedral mesh traversal method that uses Plücker coordinates. Aman et al. [14] introduce an XOR-compacting scheme for tetrahedral mesh marching, improving traversal performance and reducing the memory required to store the mesh connectivity information. They perform an additional traversal to locate the initial element, starting from an arbitrary element. Sahistan et al. [15] extend this idea with a shell-to-shell traversal strategy, where they construct a hardware-accelerated BVH over the boundary faces of the tetrahedral mesh, or the "shell", and initialize traversal from these shells, avoiding the cost of an additional search. They send one forward-facing and one backward-facing ray to robustly find the shell faces. Later, Sahistan et al. [16] extended the XOR-compacting scheme to other polyhedral types: wedges, pyramids, and hexahedra.

Point-location methods use a spatial acceleration structure constructed over the elements. Recent methods leverage ray tracing hardware in modern GPUs, with accelerated BVH traversals. Wald et al. [17] proposed using hardware-accelerated BVH traversal on tetrahedral meshes for point location. Morrical et al. [18] extended this to other element types.

2.1.3. Adaptive Mesh Refinement

Adaptive Mesh Refinement (AMR) [19] refines the mesh in regions where further detail and data precision are needed. Since the mesh can be refined and coarsened in small regions, it does not suffer from the size drawbacks of structured grids while having a similar structure. In *block-structured AMR*, the data at different refinement levels are stacked on top of each other, maintaining multiple levels of data for the same location. *Tree-structured AMR* stores the data in a hierarchical manner, such as octrees and forests of octrees. Furthermore, some Tree-structured AMR representations, such as FLASH [20], use hierarchies similar to octrees but have different branching factors.

Most AMR simulations generate cell-centered grids [21,22]. However, scientists typically want smooth interpolation when rendering the data, which standard tools only support for vertex-centered data. To efficiently sample and interpolate these volumes, one approach is to shift each grid cell by half of its width, creating "dual cells" on the fly that effectively shift the data to the vertices. Sampling within AMR regions using the dual grid is as simple as applying the structured-grid sampling methods. However, the problem arises at level boundaries, where regions of two different refinement levels meet and cause a "T-junction". The data points on each side of the dual grid cannot be used to tri-linearly interpolate a sample at these boundaries, as the vertices are not equally spaced. Correctly sampling from these regions requires different approaches.

A straightforward solution is to use nearest-neighbor interpolation, which has been shown to cause artifacts, such as cracked isosurfaces or seams where levels meet during rendering [1]. Properly rendering the data at level boundaries requires converting the dual-grid cells to hexahedral elements and "stitching" them together. However, this approach requires more memory, as the grid elements are now stored in an unstructured fashion with additional non-voxel cells connecting the boundaries. Zellmann et al. [23] introduce a memory-efficient way to render AMR data using dual meshes with stitching, by clustering voxel grid cells and converting them into "gridlets", small grids of neighboring voxels of the same size.

Wald et al. [24] use a tent-shaped basis function to compute the weighted average of the cell samples that fall inside the footprint of the sampled cell. Wang et al. [25] propose to handle discontinuities at level boundaries by decomposing each cell into its octants, first interpolating the data for the octant vertices, using the data of the dual mesh and the cell centers, then the sample point itself. Unlike stitching, these works focus on constructing the samples at the boundaries without introducing additional elements.

AMR data is generally composed of voxels, which can limit the domain it can represent. Recently, Holke [26] proposed tree-based AMR meshes with general element types with tetrahedra, wedges, pyramids, and hexahedra. Unlike voxel-based AMR, their data structure consists of a coarse unstructured mesh with a forest of trees, each tree refining one of the coarse cells. The coarse mesh can better represent the simulation domain, while refinement trees can be refined locally when higher accuracy is needed, similar to voxel-based AMRs.

`t8code` [27] is a library that supports the creation and adaptation (refinement and coarsening) of tree-based AMR meshes of hexahedra, tetrahedra, prisms, pyramids, or any combination of these elements [26,28]. Recent works focus on AMR meshes with general element types. Burstedde and Holke [29] present a method for partitioning the coarse mesh into multiple parts, ensuring that the leaf elements on each node contain a balanced number of elements and the required metadata for the coarse neighbors. They also introduce a tetrahedral space-filling curve [30] with accompanying algorithms to perform various operations for adaptive meshes.

Although numerous works focus on rendering voxel-based AMR data, rendering AMR data with other element types remains unexplored in the literature. Unlike voxel-based AMR meshes, rendering AMR meshes with other element types requires more complex operations to locate the containing elements. In voxel-based AMR, this task is straightforward. In contrast, AMR data with other element types use an unstructured coarse mesh, which makes it more challenging to efficiently locate the containing coarse element and requires a spatial acceleration structure. Similarly, traversing refinement trees is more complex, as identifying the correct child element requires more computations. We focus on rendering tree-based tetrahedral AMR meshes. Fig. 1 illustrates the hierarchical refinement levels for a tree-based tetrahedral AMR mesh.

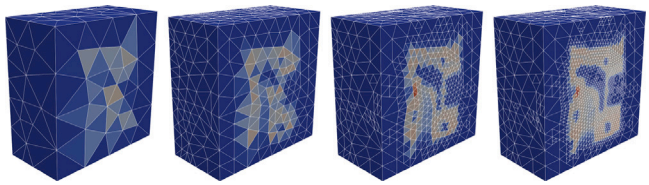


Fig. 1. Hierarchical refinement levels for the tree-based tetrahedral AMR mesh of the Engine dataset.

2.2. Direct volume rendering

Direct volume rendering creates an image from volumetric data without first creating a proxy representation (such as an isosurface). When photons travel through and interact with particles in volumes, they are subject to various phenomena, such as light emission, absorption, and scattering [31]. The primary operations of volume rendering are *sampling* and *classification*. Sampling is the acquisition of data from a point within a volume, and classification is the mapping of the acquired data samples into color and opacity values, using transfer functions. Different approaches to direct volume rendering exist, including stepwise sampling and approximating physical interactions, as well as approximating the interactions between light photons and particles in the medium. We briefly summarize the two methods: *ray marching* and *free-flight distance sampling via Woodcock tracking*.

2.2.1. Ray marching

Volume ray marching, introduced by Levoy [32], renders the volume by taking fixed-size steps along a ray. At each step, the volume is sampled, and the corresponding color and opacity values are computed and composited along the ray in a front-to-back order using the following formulation:

$$\begin{aligned} C &= C + (1-\alpha) \times \alpha_{sample} \times C_{sample} \\ \alpha &= \alpha + (1-\alpha) \times \alpha_{sample} \end{aligned} \tag{1}$$

The marching process continues until the ray terminates, either when the accumulated opacity reaches a threshold or the ray exits the volume. The final accumulated color determines the color of the corresponding pixel in the rendered image. The pseudo-code for ray marching is provided in Algorithm 1.

Algorithm 1 Ray marching.

```
function RayMarch(volume, ray, stepSize, transferFunction)
    color ← Color(0,0,0,0)
    while volume.contains(ray.origin + ray.dir × ray.t) do
        sampledScalar ← volume.sample(ray.origin + ray.dir × ray.t)
        sampledColor ← transferFunction.mapColor(sampledScalar)
        color ← composite(color, sampledColor)
        ray.t ← ray.t + stepSize
    end while
    return composite(color, backgroundColor)
end function
```

The step size can be a fixed value or adjusted dynamically to better suit the different regions of a heterogeneous volume. Setting the step size too large can lead to undersampling, resulting in the omission of high-frequency features of the volume during rendering. On the other hand, setting the step size low (oversampling) ensures accurate rendering; however, taking more samples reduces the rendering performance.

2.2.2. Woodcock tracking

Tracking methods simulate the movement of photons inside a volume and the various collisions they may have by sampling a free path, the distance without any scattering or absorption collision events, instead of directly sampling the data [33]. Closed-form tracking can

perform free-path sampling in homogeneous volumes with constant extinctions. However, this is impossible if the volume is heterogeneous, as is often the case with many volumes we would like to render.

Woodcock tracking (also known as delta tracking) [34] solves this by introducing fictitious particles. It homogenizes the volume's density with fictitious particles to an upper bound, known as the volume's majorant extinction. The majorant extinction acts as an absolute upper bound for the maximum density within the volume, allowing a heterogeneous medium to be safely evaluated as if it were homogeneous. It simplifies the problem to one that can be solved with closed-form tracking. When the ray samples a fictitious particle, it is treated as a null-collision. A null-collision is treated as another type of collision, in which the ray has no interruption in its path and continues as normal. The traversal stops if a ray samples a real collision.

Woodcock tracking is a Monte Carlo method in which the rays are sampled probabilistically to estimate particle collisions. Over many iterations, the Monte Carlo estimate converges to an unbiased and accurate rendering of the volume. Compared to ray-marching, Woodcock tracking typically requires fewer volume samples. However, it requires the accumulation of results from many rays. Algorithm 2 describes the Woodcock tracking for a ray.

Algorithm 2 Woodcock tracking.

```
function Woodcock(volume, ray, majorant, transferFunction)
    while volume.contains(ray.origin + ray.dir × ray.t) do
        ray.t←ray.t−log(1−rand())/majorant        ▷ Sampling the free-path
        sampledScalar ← volume.sample(ray.origin + ray.dir × ray.t)
        sampledColor ← transferFunction.mapColor(sampledScalar)
        if sampledColor.density ≥ rand() × majorant then
            return sample
        end if
    end while
    return backgroundColor
end function
```

The performance of Woodcock tracking is defined by the number of null-collisions sampled during the traversal of a ray, which depends on the majorant extinction used for the volume. Instead of using a single majorant extinction over the entire volume, Kalos et al. [35] use a structured grid of majorants to bound the local maximum densities tightly, thereby reducing the number of null collisions. Recent works have used coarse-structured grid macrocells to obtain tighter majorants for parts of a volume [1,36,37].

2.2.3. View-dependent rendering

View-dependent rendering in direct volume visualization modifies the rendering based on the camera's location, which can be achieved in various ways. Kähler et al. [38] exploit level-of-detail-based methods that stop traversing into higher refinement levels and use data at lower detail when a threshold like the size of elements on screen is crossed, improving performance with minimal visual loss, while importance-based approaches [39,40] adapt sampling according to view-dependent relevance.

3. Method overview

This work addresses the problem of direct volume rendering of tree-based tetrahedral adaptive mesh refinement data (Tet-AMR). Although AMR frameworks like `t8code` support hybrid AMR meshes with various element types, we focus on AMR meshes composed solely of tetrahedral elements.

Rather than explicitly storing the entire hierarchy, Tet-AMR structures typically maintain a compact description of the leaf elements that enables reconstruction of the full refinement forest. Enumerating the leaf elements according to the recursive refinement pattern yields a space-filling curve (SFC). Using this SFC, all elements within a refinement tree are assigned integer indices and stored linearly. Explicit storage of element coordinates or adjacency is unnecessary, as both can be reconstructed from the SFC.

During preprocessing, we first reconstruct missing interior (parent) elements of the refinement trees. We then reorder sibling elements from the SFC order to a preorder arrangement for efficient GPU traversal. Next, we convert cell-centered scalars to vertex-centered scalars. We do not store vertices of refined elements; instead, we reconstruct them on the fly from coarse elements during refinement tree traversal.

We implement two volume rendering techniques: ray marching and Woodcock tracking. Rendering requires us to sample at specific query points. To sample the volume, we first locate the coarse element. To accelerate this process, we construct a hardware-accelerated BVH over the faces of the coarse elements. After finding the refinement tree rooted at the containing coarse element, we start the refinement tree traversal process. We traverse the refinement tree one level at a time, generating the geometry of the current level on the fly, testing the query point against the geometry, and identifying the containing element for that level. This process continues until either the finest element containing the query point is found or a screen space area threshold is met.

In Section 4, we describe the preprocessing pipeline and the construction of our GPU-friendly rendering data structure for Tet-AMR meshes. Section 5 details the volume rendering process, including coarse-element traversal and refinement tree traversal for both ray marching and Woodcock tracking.

4. Preprocessing for rendering Tet-AMR

We use the VTK unstructured mesh format (VTU) as a generalized interface for Tet-AMR data. This conveniently allows for loose coupling at the interface between our backend and other application frontends. Because we assume our input only contains the leaf nodes of the data structure, we first have to reconstruct the AMR hierarchy as part of the conversion process.

During preprocessing, we convert the input data into a GPU-friendly representation. Our preprocessing starts with removing duplicate vertices. Tet-AMR output usually contains duplicate vertices; we remove them and store the unique shared vertices along with an index array containing the vertex IDs for each element, thereby reducing the data size. Moreover, it allows us to work with the indices during the later steps of the data preparation process.

The subdivision of the parent tetrahedron into eight child tetrahedra relies on Bey's red-refinement rule [41–43]. Each tetrahedron is defined by four vertices. In this work, we refer to these vertices as the top, left, right, and back vertices, denoted by T, L, R, and B, respectively. We further define the midpoints of these vertices as follows:

TL=(T+L)/2	TB=(T+B)/2	LR=(L+R)/2
TR=(T+R)/2	LB=(L+B)/2	RB=(R+B)/2

A tetrahedron is divided into eight child tetrahedra, four outer (those having a vertex of the original parent tetrahedron) and four inner (those not having a vertex of the original parent tetrahedron), and they are defined with a combination of the original corner vertices and the previously defined midpoints. All of them have one-eighth of the volume of the parent element. Fig. 2 depicts the child tetrahedra, and we define them as:

- Top Outer={T,TL,TR,TB}
- Top Inner={LB,TB,TR,TL}
- Left Outer={L,TL,LB,LR}
- Left Inner={TR,LR,LB,TL}
- Right Outer={R,RB,TR,LR}
- Right Inner={LB,LR,TR,RB}
- Back Outer={B,RB,LB,TB}
- Back Inner={TR,TB,LB,RB}

As our input is composed only of the leaf elements of each refinement tree, we generate the missing parent elements in a bottom-up

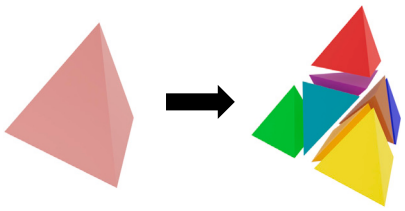


Fig. 2. Parent tetrahedron refined into eight child tetrahedra using Bey's red-refinement rule.

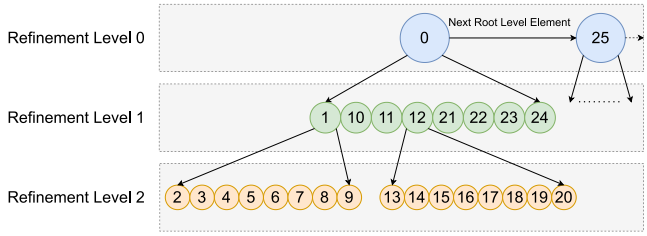


Fig. 3. Refinement tree element order with preorder traversal.

process to reconstruct the full hierarchy. Starting from the leaf elements, we process every group of eight elements subdivided from the same parent. The parent element uses the four outer corner vertices of its children as its vertices. We calculate the cell data at the centroid of this new element using the cell data of its children. We also calculate the child counts for each element, which we use during reordering and traversal. The child count for an element is the number of its descendants. Once we calculated all the necessary values, the child elements are removed from our working set, and the new parent element takes their place. We continue the process until the root element of the current tree has been generated. The process then continues with the next tree until all trees have been processed.

The elements in each refinement tree must be ordered to facilitate easy access. We accomplish this by reordering them so that every element will have all its descendants before the next sibling element of the parent; this corresponds to elements being given IDs ordered according to the preorder traversal of each tree, an example of which can be seen in Fig. 3.

The first child of an element is the "Top Outer" child, and is given the ID $parentID + 1$. The next child of the element is the "Top Inner" child, given the ID $topOuterChildID + 1 + childCounts[topOuterChildID]$. The rest of the siblings' IDs are similarly offset by the previous sibling's child count plus one. The sibling pairs follow the order *Top* → *Left* → *Right* → *Back*, and for each pair, the outer one is ordered before the inner. During this process, we also reorder the vertices of each child relative to its parent vertices, ensuring each child's vertex ordering is consistent and has a positive signed volume. After all elements are reordered, we can use the child counts to find the IDs of the children or siblings of a given element.

We compute vertex-centered scalars for all vertices in the forest to enable smooth interpolation of data values. We traverse all leaf-level elements, and for each vertex of an element, we accumulate the scalar contributions from the cell scalar. We use inverse distance weighting, incorporating the scalars and centroid distances of neighboring cells, to calculate the final value of the vertex-centered scalar. These computed values are stored in a flat 1D array indexed by the unique vertex IDs, which is then passed to our GPU rendering data structure. For each element face, we store a neighbor ID. Since an adjacent element may be refined to a higher level than the element itself, we store only a single neighbor ID per face, corresponding to a neighbor at the same or a coarser level. Faces on the mesh boundaries do not have neighbors; we indicate these boundary faces with a neighbor ID whose bits are all set to 1. At the end of preprocessing, we discard the vertices of the non-coarse elements, since they will be generated on the fly during traversal. Our GPU rendering data structure is given in Listing 1. For the coarse mesh, we store the `coarse_index` and `coarse_vertex` buffers; the former stores the triangular faces of the coarse tetrahedra, with indices pointing to the `coarse_vertex` buffer. We use the `coarse_index` and `coarse_vertex` buffers to construct a hardware BVH for the coarse-mesh faces. `tree_root_ids` and `macrocell_values` arrays store the element IDs of the roots for each tree and the minimum and maximum macrocell densities for each coarse element. `cell_data` and `vertex_data` buffers store the per-cell and per-vertex scalars.

Listing 1: The global GPU rendering data structure. It stores coarse-mesh geometry, macrocell density ranges, and refinement-tree data (including child counts, neighbor IDs, and scalars) in flattened arrays for rendering.

```
struct Render_Data {
  uint64_t num_coarse_elements;
  uint64_t num_elements;

  vec3i *coarse_index;
  vec3f *coarse_vertex;
  uint64_t *tree_root_ids;
  vec2f *macrocell_values;

  uint64_t *child_counts;
  uint64_t *neighbors;
  float *cell_data;
  float *vertex_data;
};
```

5. Rendering

At the start of rendering, we construct a ray for each pixel. We then traverse the coarse mesh tetrahedra one at a time along the ray using a hardware-accelerated BVH. We sample scalar values along the ray by traversing the refinement tree associated with the coarse element. In this section, we describe the coarse-mesh traversal, the tetrahedral-refinement-tree traversal, and the rendering methods.

5.1. Coarse mesh traversal

We use a triangle BVH to traverse the coarse tetrahedral mesh. The BVH is built over the coarse mesh by inserting four triangular faces facing outside, for every coarse element. Our coarse mesh traversal method is similar to shell-to-shell traversal [15]. However, we trace two forward-facing rays for traversal rather than one forward-facing and one backward-facing ray. We found that the two-forward-facing-ray method yields a simpler traversal scheme without sacrificing the robustness of the shell-to-shell traversal.

We trace two rays into the BVH, one with front-face culling enabled, and the other with back-face culling enabled. If the front culling enabled ray misses, we know we are outside the coarse mesh and that there are no elements along the ray; if there were, they would intersect the ray. If the front culling enabled ray hits, there are three cases of the coarse mesh traversal algorithm (see Fig. 4). If both rays intersect with the same tetrahedral element, we know that the ray is currently outside the coarse mesh, and the next element is a boundary element (Case 1). If both rays intersect with different tetrahedral elements (Case 2), which is the case for any neighboring tetrahedra, or if there is a hole in the volume. If the back-culling-enabled ray has missed, we know we are inside the last element that intersects the ray on the coarse mesh (Case 3). Using the miss case and the three distinct cases, we can handle traversing coarse meshes that are non-convex or have holes.

If none of the rays hit, we return a value indicating we are outside and beyond the coarse mesh. For any other case, we return the coarse element ID, the maximum t value for the element, and the t value for

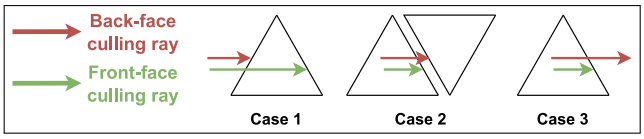


Fig. 4. Possible cases for the two rays traced into the coarse mesh. Case 1: The ray is entering the coarse mesh from outside. Case 2: the ray is inside the coarse mesh and intersects another coarse mesh element. Case 3: the ray is inside a boundary element of the coarse mesh, and will exit the volume.

the next coarse element on the ray. Since we already use the coarse mesh to identify coarse elements, using it as a macrocell [37] mesh also allows us to perform performance optimizations, such as skipping empty space by checking the maximum density value associated with a coarse element before processing it. If it is zero, we continue to the next coarse element, skipping the current one. It can also allow the Woodcock renderer to obtain a tighter majorant extinction for each coarse element, reducing the chance that a sample will be rejected and allowing us to sample longer free-path distances in that region of the volume. This, in turn, reduces the number of samples taken and improves performance.

We treat each coarse mesh element as a macrocell, utilizing an approach similar to the Adaptive Sampling Clusters in Morrical et al. [36]. For each coarse element, we store the precomputed minimum and maximum scalar values of all its descendant elements, which is similar to Wilhelms and Van Gelder's work [44]. As in [36], whenever the transfer function is edited, we recalculate the maximum density for each macrocell in parallel on the GPU by performing a linear search over the transfer function within the precomputed scalar range. Since the number of coarse elements is typically small, this GPU operation is highly efficient and is performed only once per change in transfer function to maintain interactive rendering performance [37].

5.2. Tetrahedral refinement tree traversal

Tetrahedral refinement tree traversal starts with a query point, the ordered vertex set of the coarse tetrahedron, and the coarse tetrahedron's element ID. The process traverses the tetrahedral refinement tree from the coarse root element to the leaf tetrahedron containing the query point. It returns the leaf element's ID and its four corner vertices. At each step, the current tetrahedron's vertices are updated on the fly, avoiding storage of lower-level elements. Traversal terminates upon reaching a leaf, a maximum depth, or when the projected screen-space size falls below a threshold (Section 5.4), and a sample is taken from the last visited level. Fig. 5 illustrates this process. The tetrahedral refinement trees need to be queried repeatedly during rendering, which requires an efficient method to locate the specific child element containing the query point. Instead of testing all eight child elements, we can take advantage of the regular structure of red-refined tetrahedra. We explored two alternative methods: *point–plane-based* and *barycentric-coordinate-based* tests.

5.2.1. Point–plane-based test

We use half-space tests to classify the query point with respect to the plane defined by the points (*TR, LB, TB*) and (*TR, LB, TL*). If the point is in front of the plane (*TR, LB, TB*), we know that it is either in the right or the back quadrant, and similarly for the plane (*TR, LB, TL*), the top and back quadrants (see Fig. 6). Since both tests omit the left quadrant, we can test whether the values are the same. If they are, we are either in the left or back quadrants. We also know which half we are in from the first test: either the top-left or the bottom-right. Lastly, we perform the in-quadrant plane test to determine whether the point lies within the inner or outer child. We do not need to perform all four in-quadrant tests as specified in Algorithm 3, once we determine the quadrant the point belongs to, a point–plane test is sufficient to identify the child tetrahedron.

Algorithm 3 Point plane test method for refinement tree traversal.

```
1: function TraversePointPlane(tetID, childCounts, maxDepth, point)
2:     rLevel ← 0
3:     V ← GetCoarseVertices(tetID)
4:     while childCounts[tetID] > 0 and rLevel ≤ maxDepth do
5:         t, l, r, b ← V
6:         RB_half ← halfSpace(point, (t+r)/2, (l+b)/2, (t+b)/2)
7:         TB_half ← halfSpace(point, (t+r)/2, (l+b)/2, (t+l)/2)
8:         inner ← halfSpace(point, (t+l)/2, (t+b)/2, (t+r)/2) &
                   halfSpace(point, (l+t)/2, (l+r)/2, (l+b)/2) &
                   halfSpace(point, (r+t)/2, (r+b)/2, (r+l)/2) &
                   halfSpace(point, (b+t)/2, (b+l)/2, (b+r)/2)
9:         localChildNo ← (RB_half × 4) +
                          ((RB_half == TB_half) × 2) + inner
10:        V ← CalculateTetrahedronVertices(V, localChildNo)
11:        tetID ← CalculateChildID(tetID, localChildNo, childCounts)
12:        ++rLevel
13:    end while
14:    return tetID, V
15: end function
```

After the local child index has been found, the CalculateChildID function uses the child counts of the current element's children to find the element ID of the child with the given local child number (cf. Algorithm 4). The CalculateTetrahedronVertices function uses the given set of vertices and the local child no, indicating which of the eight children it is, to create the vertices of the child element using the corresponding child tetrahedron representation (e.g., top outer, top inner) defined in Section 4. The process repeats until one of the termination conditions is met, after which we return the ID of the located element and its ordered vertex set.

Algorithm 4 Calculates the child tetrahedron ID from the parent's ID and child number

```
1: function CalculateChildID(tetID, childNo, childCounts)
2:     tetID ← tetID + 1
3:     for i ← 0 … childNo do
4:         tetID ← tetID + childCounts[tetID] + 1
5:     end for
6:     return tetID
7: end function
```

5.2.2. Barycentric coordinate-based test

An alternative to the point–plane tests is to use the barycentric coordinates of our sample point, calculated for the current element we are inside. The logic is similar to the point plane test traversal algorithm. However, instead of explicitly using planes, we can determine which child element we are in by checking different combinations of barycentric coordinates. (u, v, w, x) correspond to the barycentric coordinates of the point associated with the currently traversed element's top, left, right, and back vertices, respectively. The first check $(u + v) < 0.5$ corresponds to the plane (*TR, LB, TB*), the vertices of which have the following barycentric coordinates: *TR*=(0.5, 0, 0.5, 0), *LB*=(0, 0.5, 0, 0.5), and *TB*=(0.5, 0, 0, 0.5).

The plane defined by these three points is the $u+v = 0.5$ plane, which divides the tetrahedron volume into two halves. Any point closer to the top or left vertices will have $u > 0.5$ or $v > 0.5$, moving away from the plane. The other check, $(v + w) < 0.5$, corresponds to the plane (*TR, LB, TL*) in the point–plane test version and works similarly. The plane tests that check whether the point is in an inner or outer element are similarly translated to their barycentric-coordinate versions. Using these tests, we can identify which child element we are in, relative to the current traversed element. After each iteration, we divide the volume by eight, since the child tetrahedra volume is one-eighth of the

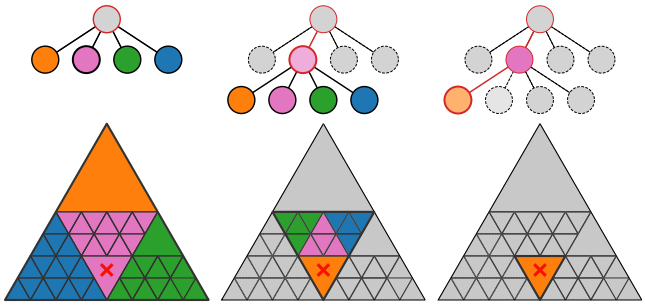
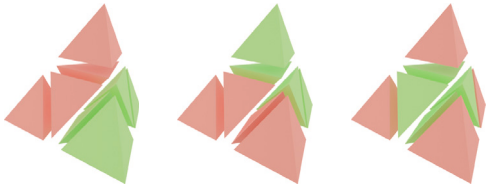


Fig. 5. Tetrahedral refinement tree traversal logic, shown as a simplified 2D version with four triangles instead of eight child tetrahedra. Starting from the root, the figure illustrates the sampling process from left to right. The sample point is marked with a red cross.



(a) RB_half (b) TB_half (c) Inner

Fig. 6. Tests performed for identifying the child element.

parent element. Algorithm 5 shows the tree refinement tree traversal with the barycentric coordinates. To optimize the algorithm, we can skip normalizing the barycentric coordinates. Instead of calculating the volume, we can compute the half-volume at the beginning, so comparisons against 0.5 become comparisons against the half-volume. Any volume calculation will have a positive signed value, as the elements' vertices were ordered for that. We therefore do not need to check the sign of the volume upon comparison.

Algorithm 5 Barycentric-based refinement tree traversal.

```
function TraverseBarycentric(tetID, childCounts, maxDepth, point)
    rLevel ← 0
    V ← GetCoarseVertices(tetID)
    vol ← volume(t, l, r, b)
    while childCounts[tetID] > 0 and rLevel ≤ maxDepth do
        t, l, r, b ← V
        u ← volume(point, l, r, b)/vol
        v ← volume(point, t, b, r)/vol
        w ← volume(point, t, l, b)/vol
        x ← 1 − (u + v + w)
        RB_half, TB_half ← (u + v) < 0.5, (v + w) < 0.5
        inner ← (u < 0.5) & (v < 0.5) & (w < 0.5) & (x < 0.5)
        localChildNo ← (RB_half × 4) +
                       ((RB_half == TB_half) × 2) + inner
        V ← CalculateTetrahedronVertices(V, localChildNo)
        tetID ← CalculateChildID(tetID, localChildNo, childCounts)
        vol ← vol/8
        ++rLevel
    end while
    return tetID, V
end function
```

5.3. Rendering algorithms

We implemented two volume renderers, one using ray marching and the other using Woodcock tracking. Only emission and absorption were implemented in the renderers, without any secondary effects. Each renderer has an accompanying coarse mesh traversal method that calls the actual rendering method, which processes each element traversed on the coarse mesh. The algorithms provided utilize macrocell density optimizations; however, we have also implemented modified versions that do not employ macrocells for the experiments. Appendix A provides the detailed algorithms for both renderers.

5.3.1. Ray marching

Our ray-marching method starts by identifying the coarse element for sampling. If the maximum macrocell density is zero, we skip processing that element and move to the next one. We perform a two-level nested march across the coarse element and its descendant elements. Once we retrieve the vertices and ID of the target element, we determine when the ray exits by intersecting it with the element's faces and selecting the smallest forward t value. This avoids repeated tree traversals within the same element. We then march, take samples, apply the transfer function, and composite. Once outside the coarse element, we resume coarse-mesh traversal. We repeat the process until we are outside the volume or when the opacity threshold of 0.99 is exceeded. The method uses a constant step size, which impacts rendering performance at smaller sizes and requires more samples.

5.3.2. Woodcock tracking

As an alternative to fixed-step ray marching, we implement a Woodcock tracking renderer that samples interaction events based on majorant. We employ the coarse mesh as a macrocell structure to provide majorant estimates for the volume. Processing each coarse element begins by computing a majorant extinction coefficient for its volume, obtained by scaling the maximum macrocell density with the transfer function's opacity. If this value is zero, the coarse element is skipped.

Using the majorant, we sample a free-flight distance and advance the ray accordingly. While the ray remains inside the current coarse element, we evaluate the volume density at the sampled position by traversing the refinement tree to determine the local scalar value. This value is mapped through the transfer function. We interpret the color component as albedo, and opacity as extinction. The extinction is compared against a random threshold, and if the sample is accepted, the corresponding color component is returned as the sample's albedo. Otherwise, the algorithm continues by sampling a new free-flight distance.

When using macrocell-based majorants, the free-flight distance may advance the ray beyond the boundary of the current coarse element due to overestimation of the local density. To ensure correct traversal, after finishing Woodcock tracking within a coarse element, we clamp the ray position to just before the entry point of the next coarse element. This correction is not required when macrocell majorants are not used.

5.4. View-dependent early tree traversal termination

We implemented view-dependent early termination when traversing the tree to find elements to sample from—if their projected screen-space size is below a certain threshold. For that, while we traverse each level of a refinement tree to find the element to sample, whether we

proceed to the next level is determined by comparing the screen-space area against a user-controlled refinement criterion.

Assuming viewing rays originate outside a convex polytope, any intersecting ray enters and exits the volume exactly once, producing two surface intersections. Consequently, the front-facing and back-facing facets project onto the identical image-space region. Therefore, the screen-space silhouette area is half the sum of the projected areas of all faces. The calculation is described in Algorithm 6. We use cross products to calculate the area of a triangle. Since they give the area of the wedge defined by the two vectors, we divide the sum by four. We modified Algorithms 3 and 5 given in Section 5.2, only calculating the screen space size to avoid unnecessary computation if the refinement criterion is greater than zero.

Algorithm 6 Screen space area calculation.

```
function CalculateScreenSpaceArea(T, L, R, B, camera)
    T ← camera.projectPoint(T)
    L ← camera.projectPoint(L)
    R ← camera.projectPoint(R)
    B ← camera.projectPoint(B)
    area ← |(L − T) × (R − T)| + |(R − T) × (B − T)|
    area ← area + |(B − T) × (L − T)| + |(L − B) × (R − B)|
    return area / 4
end function
```

6. Experimental results

We run our experiments on a NVIDIA GeForce RTX 4070 Ti desktop GPU, using OWL [45] and OptiX 7.7 [46] to implement renderers. We implemented two Tet-AMR renderers, a ray marcher, and a Woodcock tracker. To compare with them, we also implemented the renderers' unstructured-mesh versions, which use the same coarse-mesh traversal process. However, all leaf-level elements are inserted into the BVH, eliminating the need for refinement tree traversal. All tests in this section were performed using vertex-centric sampling; a comparison with cell-centric sampling is provided in Appendix B. Unless otherwise stated, the tests were run using barycentric traversal for the Tet-AMR renderers, and the images were rendered at 1920 × 1080 resolution. The transfer functions, step sizes for the ray marcher, and opacities were chosen per dataset but remained constant throughout the experiments. All renderers were executed for several frames prior to benchmarking to exclude initialization overheads from the measurements. The reported rendering times are average per-frame render times computed over 1000 frames.

If the memory required for the acceleration structure building process could be accommodated within the GPU device memory, we retained it as a single bottom-level acceleration structure (BLAS). Otherwise, we constructed a top-level acceleration structure (TLAS)+BLAS hierarchy. If we group elements into BLASs without regard for geometry, we end up with more TLAS tests per ray, thereby decreasing performance. The data must be carefully divided into smaller, non-overlapping groups for optimal performance. To this end, we split the tetrahedra into four groups, with a split along the center x-value and another along the center y-value of the bounding box of the geometry. We keep the chunks' bounding boxes as separate as possible to minimize performance loss and ensure a fair comparison. We only had to do this for the unstructured renderer with the Engine dataset, which contains 34M leaf elements (see below).

6.1. Datasets

Due to a lack of publicly available Tet-AMR datasets, we synthesized test data to evaluate various scenarios. We generated these datasets using `t8code` to convert existing data to tetrahedral AMR or to generate them directly. Data generation with `t8code` begins with a coarse tetrahedral mesh, which can be optionally refined to a specified level. At each step, `t8code` adapts the mesh by coarsening or refining all leaf-level elements according to user-provided callback functions. If the sample taken from the element's centroid and the average of the samples taken from the vertices of the element differ by a threshold, we refine the element. We use 1% of the maximum scalar as the threshold. The data is sampled from the input volume being processed, whether a structured grid or an unstructured mesh. The datasets we converted or created are as follows:

- *Engine*, *Carp*, *MRI*, and *Heptane*: These raw structured grids from Open Scientific Visualization Datasets [47] were processed with the above-mentioned method.
- *Engine Uniform*: The same Engine dataset as above, but refined to a uniform refinement level of 7 throughout the mesh, with 6×8^7 leaf elements.
- *Torus*: Sampled from the implicit torus equation, refined to a uniform refinement level of 5, with 48×8^5 leaf elements.
- *Advection*: Comes from the `t8_advection` example on the `t8code` GitHub repository [27].
- *Agulhas*: Unlike the ones above, the Agulhas unstructured mesh was used as the input coarse mesh and was refined until the threshold was met up to a maximum refinement level of 3. Creating many refinement trees that are shallower than the other datasets. The Agulhas dataset is courtesy of Dr. Niklas Röber.

Due to a lack of publicly available Tet-AMR datasets, we synthesized test data to evaluate various scenarios. `t8code` initializes coarse meshes by generating a hypercube or by using an existing mesh. While the *Engine Uniform* and *Torus* datasets were generated from 1 and 8 hypercubes, respectively, each hypercube decomposed into 6 tetrahedra, the resulting elements have axis-aligned faces, which are insufficient for evaluating our traversal algorithms and view-dependent early tree-traversal method. To generate suitably irregular coarse meshes, we utilized TetGen [48] for the *Engine*, *Carp*, *MRI*, and *Heptane* datasets. By extracting the boundaries and 100 high-gradient vertices from the original structured grids, we constructed complex Delaunay tetrahedralizations for `t8code` to refine. Finally, for the *Advection* dataset, we utilized its provided coarse mesh. For the unstructured *Agulhas* dataset, we subdivided each wedge into 14 tetrahedra by inserting vertices at the wedge and face centroids. This forms two tetrahedra from the original triangular faces and twelve from the partitioned quadrilateral faces, all sharing the central vertex. The resulting mesh was then simplified using VTK quadric decimation.

6.2. Tet-AMR vs. unstructured mesh rendering

We compare our rendering method with unstructured-mesh rendering without using the coarse mesh as a macrocell and without view-dependent early tree traversal termination. The results are given in Table 1. Except for the Advection and Agulhas datasets, the Tet-AMR renderers outperform the unstructured ones. We attribute this to the Advection dataset having a large number of elements that are refined to a high level, with 5,246,424 elements at refinement level 8, causing the refinement tree traversal process to take longer on average than the other datasets. Combined with the BVH overhead, this results in worse performance. It is also similar for Agulhas; this time, the tree traversal process takes, on average, less time than for the other datasets. However, the overhead of the increased number of coarse elements in the BVH and the constant function overhead from the refinement tree traversal, which is called many times during shallow tree traversals over many elements, causes performance to decrease.

We note that these results reflect an equal-quality comparison without view-dependent refinement. Although not enabled here, the Tet-AMR renderer supports view-dependent early tree traversal termination, which can be used in practice to reduce rendering cost with little noticeable change in visual quality. This capability is not available in unstructured-mesh renderers.

Table 1
Dataset statistics and Tet-AMR vs. unstructured mesh rendering times (ms) for the Engine, Engine Uniform, Torus, Agulhas, Carp, MRI, Heptane and Advection datasets. The VRAM usage is in Mebibytes (MiB).

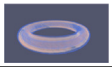
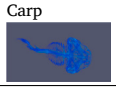
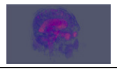
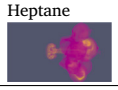
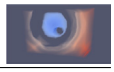
Datasets	Engine	Engine Uniform	Torus	Agulhas
				
No. elements	38,921,369	14,380,470	1,797,552	6,181,711
No. leaf elements	34,056,845	12,582,912	1,572,864	5,643,944
No. trees	5177	6	48	1,879,575
Avg. refinement level	5.56	7	5	1.53
Traversal method	Tet-AMR Rendering times			
Ray marcher	**42.55**	**73.39**	**58.04**	**32.2**
Woodcock	**22.08**	**42.95**	**61.93**	5.73
VRAM Usage	2768	1344	624	1146
Traversal method	Unstructured Mesh Rendering times			
Ray marcher	126.42	124.31	164.74	32.77
Woodcock	42.87	45.11	86.13	**5.43**
VRAM Usage	6078	2352	762	1524
Datasets	Carp	MRI	Heptane	Advection
				
No. elements	8,733,614	17,119,323	26,744,665	7,156,556
No. leaf elements	7,642,366	14,979,878	23,402,138	6,262,248
No. trees	3630	3763	4449	2092
Avg. refinement level	4.61	4.65	4.75	7.77
Traversal method	Tet-AMR Rendering times			
Ray marcher	**52.2**	**39.39**	**52.59**	54.73
Woodcock	**13.41**	**10.34**	**9.66**	3.79
VRAM Usage	1022	1502	2056	932
Traversal method	Unstructured Mesh Rendering times			
Ray marcher	125.1	110.01	75.02	**48.85**
Woodcock	21.32	18.77	12.75	**3.17**
VRAM Usage	1774	2986	4330	1512

Table 2
With and without macrocell rendering times (ms) for the Engine, Engine Uniform, Torus, Agulhas, Carp, MRI, Heptane and Advection datasets.

Traversal method	Engine	Engine Uniform	Torus	Agulhas
Ray marcher with macrocell	**42.29**	**72.35**	**57.31**	32.26
Ray marcher without macrocell	42.55	73.39	58.04	**32.2**
Woodcock with macrocell	**8.31**	**17.13**	**14.29**	57.23
Woodcock without macrocell	22.08	42.95	61.93	**5.73**
Traversal method	Carp	MRI	Heptane	Advection
Ray marcher with macrocell	**51.78**	**39.1**	**46.28**	**44.05**
Ray marcher without macrocell	52.2	39.39	52.59	54.73
Woodcock with macrocell	**8.95**	**4.72**	**5.33**	**3.78**
Woodcock without macrocell	13.41	10.34	9.66	3.79

We also evaluate GPU VRAM usage during rendering. Memory consumption increases with the number of elements stored in the BVH, which correspond to coarse elements for Tet-AMR and leaf elements for unstructured meshes. As expected, Tet-AMR uses less VRAM because it has significantly fewer BVH elements.

6.3. Coarse mesh macrocell density optimizations

We performed tests by enabling empty-space skipping in the ray marcher and using the coarse mesh as a macrocell structure for Woodcock tracking. The results are given in Table 2. With the Woodcock renderer, enabling local majorant optimization typically improves performance by allowing us to obtain tighter majorants, thereby reducing the number of sampled collisions that are not accepted. However, with the Agulhas dataset, the performance of Woodcock tracking tends to decline when employing macrocell optimizations, as we need to start processing each element from the beginning of the next element when using macrocell values, to avoid overstepping into a denser region and missing possible collisions.

It is important to note that the performance improvements depend on the transfer function, as the macrocell values are derived from the transfer function densities; this is particularly true for the ray marcher. With the ray marcher, we observe that using macrocell densities to skip empty spaces yields performance gains when the number of skipped elements is higher. However, extra memory accesses to retrieve macrocell values and checks to determine whether an element should be skipped during marching may decrease performance. Using the same transfer functions from our earlier experiments, we obtain 11, 0, 601, and 564 zero-density macrocells for the Engine, Agulhas, Heptane, and Advection datasets, respectively.

With the ray marcher, we observe notable performance improvements for Heptane and Advection, as many cells can be skipped. Performance improvements for the Engine dataset are negligible, due to the small number of zero-density macrocells. Empty-space skipping does not improve performance for Agulhas, as no zero-density macrocells exist. Instead, performance is degraded by the additional checks.

Table 3
Rendering performance (ms) of the ray marcher with macrocell for MRI dataset under transfer functions with progressively reduced opacity values. Starting from opacity scale 1.0, all opacity values are reduced by 0.15 at each step until reaching 0.10.

Maximum Opacity	1.00	0.85	0.70	0.55	0.40	0.25	0.10
Number of skipped macrocells	0	331	653	805	947	1060	1325
Ray marcher with macrocell	28.27	28.49	22.48	21.49	21.17	21.88	21.23
Transfer Function							

By modifying our transfer function to increase the number of skipped cells, we observe performance improvements. We demonstrate this on the MRI dataset using the ray tracer. At each step, we increment the number of macrocells with zero density by modifying the transfer function. Table 3 gives the results for this experiment.

6.4. Varying refinement levels of coarse elements

We evaluated different coarse mesh configurations by inserting elements from different refinement levels. For each refinement tree, traversal continues until the target level or a leaf is reached, and the resulting elements are added to the coarse mesh. Since macrocells correspond to coarse mesh elements, increasing coarse mesh resolution also increases the number of macrocells. We tested the renderers with and without macrocell optimizations to assess their performance impact; results are shown in Table 4.

The performance varies depending on the refinement level of the elements inserted into the coarse mesh. Increasing the number of elements in the coarse mesh reduces BVH efficiency, thereby lowering the maximum depth traversed during the refinement tree traversal. We do not see any improvement for the Engine dataset. However, inserting them at a higher level into the coarse mesh may be beneficial for refinement trees with many elements and a similar average tree height, as seen with Uniform Engine at refinement level 1, using the ray marcher. Although the BVH overhead increases slightly, decreasing the refinement tree traversal depth yields a slight performance improvement. We observe this behavior over multiple runs of experiments. We also observe a slight performance increase with the Woodcock tracker when using macrocell optimizations on Uniform Engine at first and second refinement levels. As macrocells shrink, the macrocell majorants become tighter, enabling the Woodcock tracker to perform more effectively.

6.5. Barycentric coordinates vs. point–plane tests

Comparing the optimized versions of the traversal methods, we see that both perform similarly, with differences within the margins of measurement error. Table 5 shows the results. The optimized barycentric method requires 3 or 4 midpoint calculations when constructing child vertices, whereas the optimized point–plane method performs 4 midpoint calculations for initial plane tests and up to 2 additional midpoint computations, depending on the selected child. Although this results in slightly more arithmetic operations for the point–plane method, the difference does not measurably affect rendering performance.

6.6. Refinement criteria

We evaluated varying refinement criteria using the view-dependent early tree-traversal termination method on the Engine dataset. Fig. 7 shows the rendering speedups, and Fig. 8 shows the rendered images.

Since we only calculate the screen space area when the refinement threshold is set to a value greater than zero during traversal, we incur an initial performance hit at startup. However, we begin to see improved performance at the expense of visual fidelity as we refine the criteria further.

Some of the performance gains are due to view-dependent early termination of tree traversal, while most are from the skipped tree traversal calls that would have been performed if we had not used a sample element from a coarser level. Because the ray marcher uses a fixed step size, it benefits more from a larger refinement threshold, since the number of samples it takes inside the sampled element depends on the step size. However, because Woodcock samples free-paths randomly, we see less of a decrease in rendering time, as it might be exiting the element much sooner, compared to the ray marcher, when we sample a large free-path.

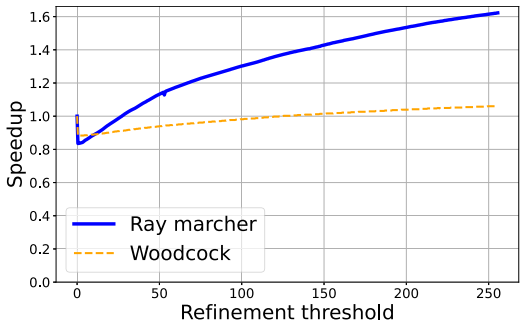


Fig. 7. Rendering performance speedups of the ray marcher and Woodcock tracking methods for different thresholds on the Engine dataset. Speedup is measured relative to rendering without view-dependent early tree-termination. The refinement threshold is defined by pixel area. For example, a refinement threshold of 128 means we do not refine an element if it covers fewer than 128 pixels on the screen.

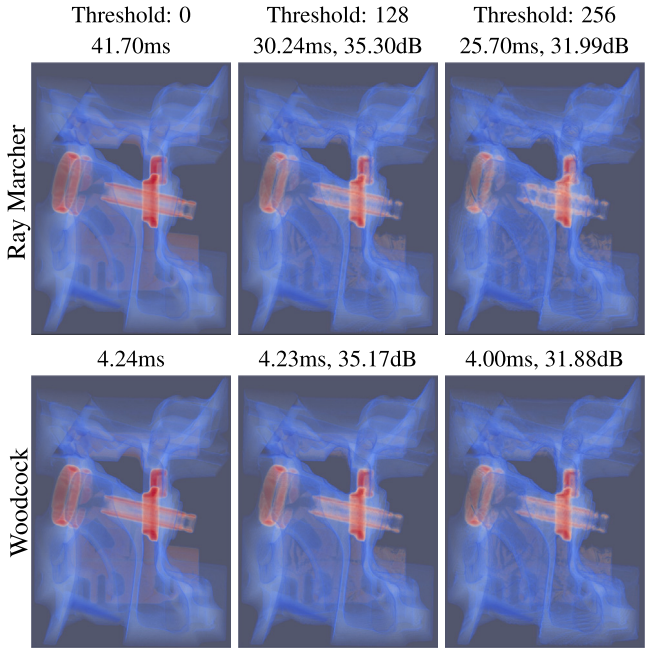


Fig. 8. Engine dataset rendered with the ray marcher and Woodcock tracking at three different refinement thresholds. The reported values indicate rendering time in milliseconds (ms) and image quality measured in peak signal-to-noise ratio (PSNR, dB).

Table 4
Rendering times (ms) at varying coarse mesh levels, with four faces per element inside the BVH. Bold values indicate the refinement levels with the fastest rendering times.

Traversal method	Refinement Levels					
	0	1	2	3	4	5
	Engine Rendering Times					
Ray marcher with macrocell	**41.75**	45.79	49.80	52.92	62.53	89.89
Ray marcher without macrocell	**40.65**	44.63	49.04	52.89	63.15	90.86
Woodcock with macrocell	**8.04**	10.56	16.25	28.37	48.22	81.37
Woodcock without macrocell	**20.85**	22.47	23.66	25.39	29.22	36.32
No. elements in BVH	5177	27,822	179,106	989,118	4,263,774	14,402,140
	Uniform Engine Rendering times					
Ray marcher with macrocell	76.9	**74.09**	79.18	89.34	97.74	96.21
Ray marcher without macrocell	74.17	**71.48**	76.31	86.20	94.50	94.11
Woodcock with macrocell	17.25	14.63	**13.24**	14.69	21.44	39.96
Woodcock without macrocell	41.74	**41.30**	46.36	51.54	51.2	47.14
No. elements in BVH	6	48	384	3072	24,576	196,608

Table 5
Traversal methods rendering times (ms) for the Engine, Engine Uniform, Torus, Agulhas, Carp, MRI, Heptane, and Advection datasets.

Traversal method	Engine	Engine Uniform	Torus	Agulhas
Ray marcher plane	44.43	79.37	62.58	32.25
Ray marcher barycentric	**42.55**	**73.39**	**58.04**	**32.20**
Woodcock plane	22.31	43.35	63.72	5.75
Woodcock barycentric	**22.08**	**42.95**	**61.93**	**5.73**
Traversal method	Carp	MRI	Heptane	Advection
Ray marcher plane	54.56	41.63	54.51	55.94
Ray marcher barycentric	**52.20**	**39.39**	**52.59**	**54.73**
Woodcock plane	13.58	10.47	9.78	**3.79**
Woodcock barycentric	**13.41**	**10.34**	**9.66**	**3.79**

7. Conclusions and future work

Our experiments demonstrated that Tet-AMR data can be rendered more efficiently in terms of memory usage and computational cost by utilizing coarse elements within a BVH and traversing the refinement tree associated with those elements, rather than rendering it as an unstructured mesh. We observe a significant reduction in memory usage when datasets with numerous leaf elements are refined from coarse meshes containing significantly fewer elements. As the maximum refinement level increases, these reductions can become even more extreme.

Although our current implementation does not consistently achieve real-time performance across all datasets, several cases already achieve interactive frame times. Further improvements in GPU implementation efficiency and traversal performance could help bring Tet-AMR rendering closer to real-time visualization.

We acknowledge that the current approach to determining when to terminate rendering, which relies solely on projected screen-space area, can result in reduced visual quality in exchange for modest performance gains in certain situations. Employing a sophisticated refinement criterion based on additional features, color, and geometric errors, light intensity, affected volume, and distance to the camera [49] may improve the balance between quality and performance. Another possible approach would be to allow the user to specify a "focus area" and render the model based on that selection [50].

Maintaining a small BVH and leveraging the structure of the tetrahedral refinement trees yields better performance. For deep, dense refinement trees, such as the uniformly refined datasets we used, inserting higher refinement levels as coarse elements can yield slight performance benefits. A heuristic could be developed to better leverage this behavior by modifying the coarse mesh before rendering starts, thereby increasing the BVH overhead while decreasing the depth of the refinement tree traversal. Future research directions include extending the Tet-AMR rendering method to hybrid AMR meshes. Additionally, data-parallel methods such as Ray Queue Cycling [51] or the approach proposed by Sahistan et al. [16] could be used to visualize larger datasets across multiple GPUs.

CRediT authorship contribution statement

Musa Ege Ünalan: Writing – review & editing, Writing – original draft, Validation, Software, Investigation, Formal analysis, Data curation. **Serkan Demirci:** Writing – review & editing, Writing – original draft, Validation, Software, Investigation, Formal analysis, Data curation. **Stefan Zellmann:** Writing – review & editing, Validation, Formal analysis. **Uğur Güdükbay:** Writing – review & editing, Validation, Supervision, Formal analysis, Conceptualization.

Declaration of generative AI usage

During the preparation of this work, the authors used Grammarly and its Generative AI module to enhance the clarity and style of the writing. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the content of the published article.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Appendix A. Rendering algorithms

A.1. Coarse mesh traversal

The pseudo-code of coarse mesh traversal is given in Algorithm 7. We first trace a ray with front-face culling enabled; if it misses, we know it is outside the volume; otherwise, we check the three cases described in Section 5.1. We use the *nextafterf* function to carry the t values to the next representable floating point number, just over the previous hit's t value to avoid hitting the same face in the next iteration.

A.2. Ray marching

Coarse-mesh traversal is the first stage of our ray-marching process. It iterates over the coarse elements intersected by the ray and uses macrocell data to determine if they should be processed. Algorithm 8 details the traversal logic for the ray marcher. As we traverse each coarse element, we perform a nested march along the ray path within that element, as detailed in Algorithm 9.

Algorithm 7 Coarse mesh traversal.

```
function TraverseCoarseMesh(ray)
    rayRecord ← traceRay(ray, CULL_FRONT_FACING_TRIANGLES)
    if rayRecord.missed then
        return -1, 0, 0
    end if
    tBack ← rayRecord.t
    coarseID ← rayRecord.coarseTetID
    rayRecord ← traceRay(ray, CULL_BACK_FACING_TRIANGLES)
    tFront ← rayRecord.t
    if !rayRecord.missed then
        if coarseID == rayRecord.coarseTetID then
            ray.t ← nextafterf(tFront, ∞)
            tNextFace ← nextafterf(tBack, ∞)
        else
            tNextFace ← nextafterf(tFront, ∞)
        end if
    else
        tNextFace ← nextafterf(tBack, ∞)
    end if
    return coarseID, tBack, tNextFace
end function
```

Algorithm 8 Ray marching for the coarse mesh.

```
function RayMarchCoarse(ray, macrocellValues, stepSize, tf,
                        childCounts)
    color ← {0, 0, 0}
    α ← 0
    while α < 0.99 do
        coarseID, tBack, tNextFace ← TraverseCoarseMesh(ray)
        if coarseID == −1 then
            break
        end if
        skipElement ← macrocellValues[coarseID] ≤ 0
        if !skipElement then
            color, α ← RayMarchElement(ray, color, α, tBack,
                       coarseID, stepSize, tf, childCounts)
        end if
        ray.t ← max(tNextFace, ray.t)
    end while
    return (1 - α) × bgColor + color
end function
```

A.3. Woodcock tracking

Algorithm 10 describes the Woodcock tracking version of coarse-mesh traversal. We traverse the coarse mesh one element at a time, applying Woodcock tracking within each coarse element using its local majorant to efficiently sample free-flight distances. This process repeats until a sample is accepted or the ray exits the volume. The inner tracking loop, which samples free-flight distances and validates them against the refinement tree data, is detailed in Algorithm 11.

Appendix B. Cell sampling vs. vertex sampling

We compared the renderers' performance using two sampling methods: cell sampling and vertex sampling. The cell sampling method uses the neighbor IDs stored during preprocessing to compute the mean of neighbors' scalars, yielding a per-element scalar. In contrast, the vertex sampling method duplicates vertex scalars per element and uses barycentric interpolation to compute the scalar at the sampled point.

Table B.6 presents the rendering times, and a visual comparison is shown in Fig. B.9. The vertex sampling method considers the location of each sample point, thereby enhancing visual fidelity. In terms of performance, both methods achieve similar rendering times.

Algorithm 9 Ray marching inside every coarse element.

```
function RayMarchElement(ray, color, α, tMax, coarseTetID,
                         stepSize, tf, childCounts)
    tExit ← 0
    sampleTetID ← 0
    tetVertices ← {}
    while ray.t < tMax do
        samplePoint ← ray.org + ray.t × ray.dir
        if ray.t > tExit then
            sampleTetID, tetVertices ← traverse(coarseTetID,
                                       childCounts, ∞, samplePoint)
            tExit←intersectFaces(ray.t, samplePoint, ray.dir, tetVertices)
        end if
        scalar ← sampleData(samplePoint, sampleTetID, tetVertices)
        ray.t ← ray.t + stepSize
        if tf.inRange(scalar) then
            tfColor ← tf.getColor(scalar)
            tfColor.opacity = tfColor.opacity × tf.opacity
            tfColor.opacity = 1 − pow(1 - tfColor.opacity, stepSize)
            color ← color + (1 - α) × tfColor.opacity + tfColor
            α ← α + tfColor.opacity × (1 - α)
        end if
    end while
    return color, α
end function
```

Algorithm 10 Woodcock tracking for the coarse mesh.

```
function WoodcockCoarse(ray, macrocellValues, tf, childCounts)
    while true do
        coarseID, tBack, tNextFace ← TraverseCoarseMesh(ray)
        if coarseID == −1 then
            return bgColor
        end if
        macrocellMajorant ← macrocellValues[coarseID]
        if WoodcockElem(ray, color, tBack, coarseID, macrocellMajorant,
                        tf, childCounts) then
            return color
        end if
        ray.t ← tNextFace
    end while
end function
```

Table B.6
Vertex sampling and cell sampling rendering times (ms) for the Engine, Engine Uniform, Torus, Agulhas, Carp, MRI, Heptane, and Advection datasets.

Traversal method	Engine	Engine Uniform	Torus	Agulhas
Ray marcher cell	**41.63**	75.53	60.39	32.30
Ray marcher vertex	42.55	**73.39**	**58.04**	**32.20**
Woodcock cell	22.39	45.59	61.98	5.86
Woodcock vertex	**22.08**	**42.95**	**61.93**	**5.73**
Traversal method	Carp	MRI	Heptane	Advection
Ray marcher cell	**48.16**	**38.35**	**51.87**	**46.97**
Ray marcher vertex	52.20	39.39	52.59	54.73
Woodcock cell	13.51	10.50	9.76	**3.43**
Woodcock vertex	**13.41**	**10.34**	**9.66**	3.79

Appendix C. Supplementary data

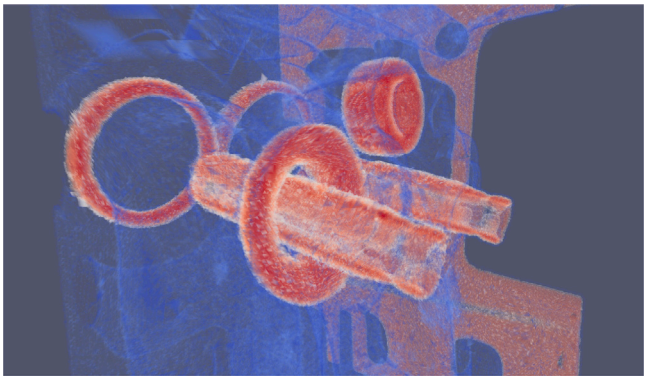
Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cag.2026.104638.

Data availability

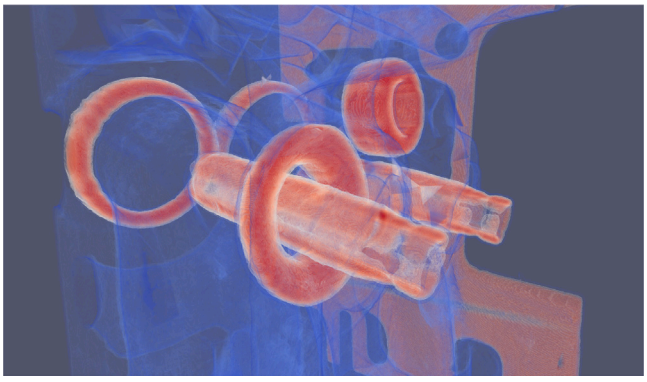
Most of the datasets used in the experiments are from public resources. The data and source code are available in the GitHub repository https://github.com/Bilkent-ModVis/tet-amr-rendering.

Algorithm 11 Woodcock tracking inside coarse elements.

```
function WoodcockElem(ray, color, tMax, coarseTetID,
                      macrocellMajorant, tf, childCounts)
    majorant ← tf.opacity × macrocellMajorant
    if majorant ≤ 0 then
        return false
    end if
    tExit ← nextafterf(ray.t, -∞)
    sampleTetID ← 0
    tetVertices ← {}
    while true do
        ray.t ← ray.t - log(1 - rand()) / majorant
        if ray.t ≥ tMax then
            return false
        end if
        samplePoint ← ray.org + ray.t × ray.dir
        if ray.t > tExit then
          sampleTetID, tetVertices←traverse(coarseTetID,
                                            childCounts, ∞, samplePoints)
          tExit←intersectFaces(ray.t, samplePoint, ray.dir, tetVertices)
        end if
        scalar ← sampleData(samplePoint, sampleTetID, tetVertices)
        if !tf.inRange(scalar) then
            continue
        end if
        color ← tf.getColor(scalar)
        if color.opacity × tf.opacity ≥ rand() × majorant then
            return color
        end if
    end while
end function
```



(a)



(b)

Fig. B.9. Engine dataset with cell sampling (a) and vertex sampling (b).

References

[1] Sarton J, Zellmann S, Demirci S, Güdükbay U, Alexandre-Barff W, Lucas L, Dischler JM, Wesner S, Wald I. State-of-the-art in large-scale volume visualization beyond structured data. Comput Graph Forum 2023;42(3):491–515. http://dx.doi.org/10.1111/cgf.14857.

[2] Greiner G, Grosso R. Hierarchical tetrahedral-octahedral subdivision for volume visualization. Vis Comput 2000;16(6):357–69. http://dx.doi.org/10.1007/PL00007214.

[3] KitWare, Inc. The Visualization Toolkit (VTK). 2022, Available at https://vtk.org/, [Accessed 29 January 2026].

[4] KitWare, Inc. ParaView. 2009, Available at https://www.paraview.org, [Accessed 29 January 2026].

[5] Childs H, Brugger E, Whitlock B, Meredith J, Ahern S, Pugmire D, Biagas K, Miller M, Harrison C, Weber GH, Krishnan H, Fogal T, Sanderson A, Garth C, Bethel EW, Camp D, Rübel O, Durant M, Favre JM, Navrátil P. VisIt: An end-user tool for visualizing and analyzing very large data. In: High performance visualization–enabling extreme-scale scientific insight. New York, NY: Chapman and Hall/CRC; 2012, p. 357–72.

[6] Berk H, Aykanat C, Güdükbay U. Direct volume rendering of unstructured grids. Comput Graph 2003;27(3):387–406.

[7] Wald I, Zellmann S, Morrical N. Faster RTX-accelerated empty space skipping using Triangulated Active Region boundary geometry. In: Proceedings of the eurographics symposium on parallel graphics and visualization. EPGV '21, The Eurographics Association; 2021, p. 37–44.

[8] Zellmann S, Schulze JP, Lang U. Binned k-d tree construction for sparse volume data on multi-core and GPU systems. IEEE Trans Vis Comput Graphics 2021;27(3):1904–15.

[9] Museth K. NanoVDB: A GPU-friendly and portable VDB data structure for real-time rendering and simulation. In: ACM SIGGRAPH 2021 talks. New York, NY, USA: Association for Computing Machinery; 2021, http://dx.doi.org/10.1145/3450623.3464653, Article no. 1, 2 pages.

[10] Schaefer S, Warren J. Dual marching cubes: primal contouring of dual grids. In: Proceedings of the 12th Pacific conference on computer graphics and applications. PG 2004, 2004, p. 70–6. http://dx.doi.org/10.1109/PCCGA.2004.1348336.

[11] Ljung P, Lundström C, Ynnerman A. Multiresolution interblock interpolation in direct volume rendering. In: Santos BS, Ertl T, Joy K, editors. Proceedings of the eurographics /IEEE vGTC symposium on visualization. The Eurographics Association; 2006, http://dx.doi.org/10.2312/VisSym/EuroVis06/259-266.

[12] Wang F, Wald I, Johnson CR. Interactive rendering of large-scale volumes on multi-core CPUs. In: Proceedings of the IEEE 9th symposium on large data analysis and visualization. LDAV '19, 2019, p. 27–36. http://dx.doi.org/10.1109/LDAV48142.2019.8944267.

[13] Maria M, Horna S, Aveneau L. Efficient ray traversal of constrained Delaunay tetrahedralization. In: Proceedings of the 12th international joint conference on computer vision, imaging and computer graphics theory and applications. VISIGRAPP 2017, vol. 1, Porto, Portugal; 2017, p. 236–43, URL: https://hal.science/hal-01486575.

[14] Aman A, Demirci S, Güdükbay U. Compact tetrahedralization-based acceleration structures for ray tracing. J Vis 2022;25(5):1103–15.

[15] 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 IEEE visualization conference. VIS '21, 2021, p. 91–5.

[16] Sahistan A, Demirci S, Wald I, Zellmann S, Barbosa J, Morrical N, Güdükbay U. Visualization of large non-trivially partitioned unstructured data with native distribution on high-performance computing systems. IEEE Trans Vis Comput Graphics 2025;31(9):5000–14. http://dx.doi.org/10.1109/TVCG.2024.3427335.

[17] 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 the conference on high-performance graphics. Goslar, DEU: Eurographics Association; 2022, p. 7–13. http://dx.doi.org/10.2312/hpg.20191189.

[18] Morrical N, Wald I, Usher W, Pascucci V. Accelerating unstructured mesh point location with RT cores. IEEE Trans Vis Comput Graphics 2022;28(8):2852–66.

[19] Berger MJ, Oliger J. Adaptive mesh refinement for hyperbolic partial differential equations. J Comput Phys 1984;53(3):484–512.

[20] Dubey A, Antypas K, Calder AC, Daley C, Fryxell B, Gallagher JB, Lamb DQ, Lee D, Olson K, Reid LB, Rich P, Ricker PM, Riley KM, Rosner R, Siegel A, Taylor NT, Weide K, Timmes FX, Vladimirova N, ZuHone J. Evolution of FLASH, a multi-physics scientific simulation code for high-performance computing. Int J High Perform Comput Appl 2014;28(2):225–37.

[21] Wang F, Wald I, Wu Q, Usher W, Johnson CR. CPU isosurface ray tracing of adaptive mesh refinement data. IEEE Trans Vis Comput Graphics 2019;25(1):1142–51.

[22] Weber GH, Childs H, Meredith JS. Efficient parallel extraction of crack-free isosurfaces from adaptive mesh refinement (AMR) data. In: IEEE symposium on large data analysis and visualization. 2012, p. 31–8. http://dx.doi.org/10.1109/LDAV.2012.6378973.

[23] Zellmann S, Wu Q, Ma K-L, Wald I. Memory-efficient GPU volume path tracing of AMR data using the dual mesh. Comput Graph Forum 2023;42(3):51–62. http://dx.doi.org/10.1111/cgf.14811.
[24] Wald I, Brownlee C, Usher W, Knoll A. CPU volume rendering of adaptive mesh refinement data. In: Proceedings of SIGGRAPH Asia 2017 symposium on visualization. SA '17, New York, NY, USA: Association for Computing Machinery; 2017, Article no. 9, 8 pages.
[25] Wang F, Wald I, Johnson CR. Interactive rendering of large-scale volumes on Multi-Core CPUs. In: Proceedings of the IEEE 9th symposium on large data analysis and visualization. LDAV '19, 2019, p. 27–36.
[26] Holke J. Scalable algorithms for parallel tree-based adaptive mesh refinement with general element types [Ph.D. thesis], Rheinische Friedrich-Wilhelms-Universität Bonn; 2018, URL: https://hdl.handle.net/20.500.11811/7661.
[27] Holke J, Markert J, Knapp D, Dreyer L, Elsweijer S, Boeing N, Lilikakis I, Fußbroich J, Leistikow T, Becker F, Ünlü V, Albers O, Burstedde C, Basermann A, Hergl C, Julia W, Schoenlein K, Ackerschott J, Evgenii A, Csati Z, Dutka A, Geihe B, Kestener P, Kirby A, Ranocha H, Schlottke-Lakemper M. T8code - modular adaptive mesh refinement in the exascale era. 2024, http://dx.doi.org/10.5281/zenodo.14418226.
[28] Holke J, Burstedde C, Knapp D, Dreyer L, Elsweijer S, Ünlü V, Markert J, Lilikakis I, Böing N, Ponnusamy P, Basermann A. t8code v. 1.0 - modular adaptive mesh refinement in the exascale era. In: SIAM international meshing round table. 2023, URL: https://elib.dlr.de/194377/.
[29] Burstedde C, Holke J. Coarse mesh partitioning for tree-based AMR. SIAM J Sci Comput 2017;39(5):C364–92. http://dx.doi.org/10.1137/16M1103518.
[30] Burstedde C, Holke J. A tetrahedral space-filling curve for nonconforming adaptive meshes. SIAM J Sci Comput 2016;38(5):C471–503. http://dx.doi.org/10.1137/15M1040049.
[31] Max N. Optical models for direct volume rendering. IEEE Trans Vis Comput Graphics 1995;1(2):99–108.
[32] Levoy M. Display of surfaces from volume data. IEEE Comput Graph Appl 1988;8(3):29–37. http://dx.doi.org/10.1109/38.511.
[33] Fong J, Wrenninge M, Kulla C, Habel R. Production volume rendering: SIGGRAPH 2017 Course. In: ACM SIGGRAPH courses. SIGGRAPH '17, New York, NY, USA: Association for Computing Machinery; 2017, http://dx.doi.org/10.1145/3084873.3084907, Article no. 2, 79 pages.
[34] Woodcock E, Murphy T, Hemmings P, Longworth T. Techniques used in the GEM code for Monte Carlo neutronics calculation in reactors and other systems of complex geometry. Technical Report, Argonne National Laboratory; 1965.
[35] Szirmay-Kalos L, Tóth B, Magdics M, Csébfalvi B. Efficient Free Path Sampling in Inhomogeneous Media. In: Hast A, Viola I, editors. Eurographics 2010 - posters. The Eurographics Association; 2010.
[36] Morrical N, Sahistan A, Güdükbay U, Wald I, Pascucci V. Quick clusters: A GPU-parallel partitioning for efficient path tracing of unstructured volumetric grids. IEEE Trans Vis Comput Graphics 2023;29(1):537–47.
[37] Zellmann S, Wu Q, Sahistan A, Ma K, Wald I. Beyond ExaBricks: GPU volume path tracing of AMR data. Comput Graph Forum 2024;43(3). http://dx.doi.org/10.1111/CGF.15095, Article no. e15095, 12 pages.
[38] Kähler R, Hege H-C. Texture-based volume rendering of adaptive mesh refinement data. Vis Comput 2002;18(8):481–92.
[39] Lerzer N, Dachsbacher C. View-dependent visibility optimization for Monte Carlo volume visualization. Comput Graph Forum 2025;44. http://dx.doi.org/10.1111/cgf.70064, Article no. e70064, 11 pages.
[40] Viola I, Kanitsar A, Groller ME. Importance-driven volume rendering. In: Proceedings of the conference on visualization '04. Washington, DC, USA: IEEE Computer Society; 2004, p. 139–46.
[41] Bey J. Tetrahedral grid refinement. Computing 1995;55:355–78.
[42] Bey J. der BPX-vorkonditionierer in 3 dimensionen: gitter-verfeinerung, parallelisierung und simulation [Ph.D. thesis], Universität Heidelberg, Interdisziplinäres Zentrum für Wissens. Rechnen (IWR); 1992.
[43] Gmeiner B, Rüde U, Stengel H, Waluga C, Wohlmuth B. Towards textbook efficiency for parallel multigrid. Numer Math Theory Methods Appl 2015;8(1):22–46.
[44] Wilhelms J, Van Gelder A. Octrees for faster isosurface generation. ACM Trans Graph 1992;11(3):201–27. http://dx.doi.org/10.1145/130881.130882.
[45] Wald I, Morrical N, Haines E. OWL: A Productivity Library for OptiX 7 and 8. 2020, Available at https://github.com/owl-project/owl, [Accessed 29 January 2026].
[46] NVIDIA Corp. NVIDIA OptiX Ray Tracing Engine. 2009, Available at https://developer.nvidia.com/optix, [Accessed 29 January 2026].
[47] Klacansky P. Open SciVis datasets. 2017, [Accessed 29 January 2026], http://klacansky.com/open-scivis-datasets/.
[48] Si H. TetGen, a Delaunay-Based Quality Tetrahedral Mesh Generator. ACM Trans Math Software 2015;41. http://dx.doi.org/10.1145/2629697.
[49] Okuyan E, Güdükbay U, İşler V. Dynamic view-dependent visualization of unstructured tetrahedral volumetric meshes. J Vis 2012;15(2):167–78.
[50] Cignoni P, Montani C, Puppo E, Scopigno R. Multiresolution representation and visualization of volume data. IEEE Trans Vis Comput Graphics 1997;3(4):352–69. http://dx.doi.org/10.1109/2945.646238.
[51] Wald I, Jaroš M, Zellmann S. Data parallel Multi-GPU path tracing using ray queue cycling. Comput Graph Forum 2023;42(8). http://dx.doi.org/10.1111/cgf.14873, Article no. e14873, 11 pages.