Eurographics Conference on Visualization (EuroVis) 2020
M. Gleicher, T. Landesberger von Antburg, and I. Viola
(Guest Editors)

Volume 39 (2020), Number 3

CPU Ray Tracing of Tree-Based Adaptive Mesh Refinement Data

Feng Wang[†1], Nathan Marshak[1], Will Usher[1,2], Carsten Burstedde[3], Aaron Knoll[2], Timo Heister[4] and Chris R. Johnson[1]

[1]SCI Institute, University of Utah [2]Intel Corp. [3]Institute for Numerical Simulation, University of Bonn
[4]School of Mathematical and Statistical Sciences, Clemson University

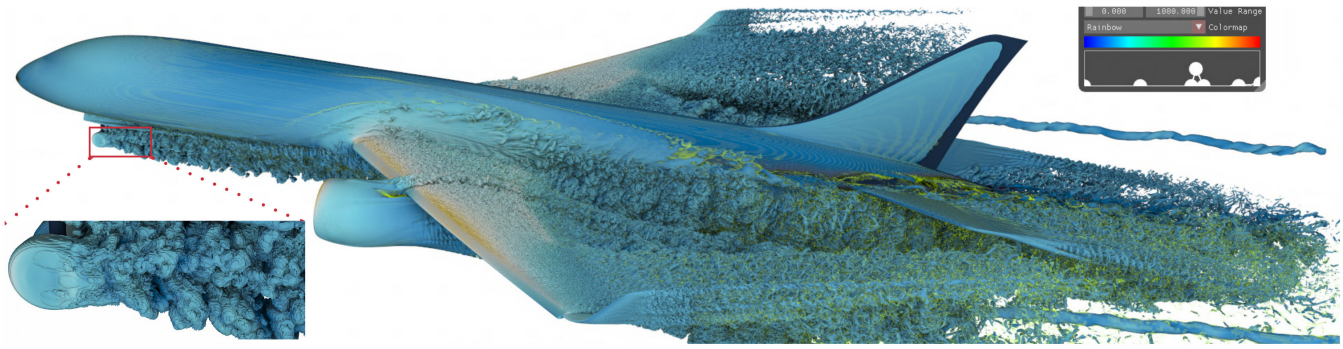


Figure 1: *High-fidelity visualization (volume and implicit isosurface rendering) of the NASA ExaJet dataset (field: vorticity) [Exa98]. This dataset contains 656M cells (1.31B after instancing) of adaptive resolution and 63.2M triangles (126M after instancing). This* 2400×600 *image is rendered on a workstation with four Intel Xeon E7-8890 v3 CPUs (72 cores, 2.5 GHz) at a framerate of 6.64 FPS.We show that our system has the capability of ray tracing TB-AMR data in combination with advanced shading effects like ambient occlusion and path tracing.*

Abstract

Adaptive mesh refinement (AMR) techniques allow for representing a simulation's computation domain in an adaptive fashion. Although these techniques have found widespread adoption in high-performance computing simulations, visualizing their data output interactively and without cracks or artifacts remains challenging. In this paper, we present an efficient solution for direct volume rendering and hybrid implicit isosurface ray tracing of tree-based AMR (TB-AMR) data. We propose a novel reconstruction strategy, Generalized Trilinear Interpolation (GTI), to interpolate across AMR level boundaries without cracks or discontinuities in the surface normal. We employ a general sparse octree structure supporting a wide range of AMR data, and use it to accelerate volume rendering, hybrid implicit isosurface rendering and value queries. We demonstrate that our approach achieves artifact-free isosurface and volume rendering and provides higher quality output images compared to existing methods at interactive rendering rates.

1. Introduction

AMR techniques have been broadly adopted to solve complex large-scale simulation problems in high-performance computing. By providing an adaptive, hierarchical representation of the computational domain, they allow the simulation to focus computation and memory in regions of interest and to better resolve fine detail features of interest [DAB*14]. Since its introduction by Berger and Oliger [BO84], AMR has been widely adopted in numerous simulation frameworks, e.g., AMReX [ZAB*19], LAVA [KBH*14], Enzo [OBB*05], Chombo [CGL*00], Paramesh [MOM*00] and p4est [BWG11]. However, native support for AMR formats in current visualization tools remains limited, and visualizing the output of these simulations remains a challenge.

A key issue when visualizing AMR data is performing correct interpolation of values across level boundaries to reconstruct the field. Although recent works have sought to address this issue, they require either introducing unstructured elements to stitch across the boundaries [WCM12, ME11], turning the AMR rendering problem into an unstructured mesh rendering problem [NSLD99], or do not always ensure a smoothly reconstructed field [WBUK17, WWW*19], leading to artifacts when rendering isosurfaces. Although capable of visualizing Block-Structured AMR data, current visualization packages (e.g., VTK [Kit03], VisIt [CBW*12], ParaView [Aya15], OSPRay [WJA*16]) provide only limited support for interactive ray tracing of Tree-Based AMR data. A common practice is to flatten Tree-Based AMR data out to a larger single-level structured or unstructured mesh, which is then rendered without any form of interpolation across boundaries. This practice is highly undesirable, as it requires significantly more memory to store and poorly repre-

† feng@sci.utah.edu

sents the original data. Although ongoing work has begun exploring a method to provide a native representation of TB-AMR data in VTK [HLP17b, HLP17a], how to perform interactive ray tracing over the presented data structure remains a challenge.

To address these challenges, we propose an efficient solution for interactive volume and implicit isosurface ray tracing of cell-centered tree-based AMR. Our visualization approach is built on a novel reconstruction strategy for interpolating across level boundaries, which we call Generalized Trilinear Interpolation (GTI). GTI ensures the reconstructed field is smooth and enables visualizations that more accurately represent ground-truth than the existing AMR reconstruction approaches. To support arbitrary AMR formats, we reorganize the leaf cells into a sparse octree. We implement vectorized tree traversal kernels for fast data query and sampling, along with empty space skipping and adaptive sampling for volume rendering, and active cell filtering for isosurface rendering. We integrate our reconstruction method and volume layout into the OSPRay ray tracing framework [WJA*16] as a module, allowing us to support high-quality combined volume and isosurface visualization. Our contributions are:

1. A novel reconstruction strategy for cell-centered AMR data, GTI, that enables artifact-free volume and isosurface rendering;
2. An efficient high-fidelity visualization solution for both AMR and other multiresolution grid datasets on multicore CPUs, supporting empty space skipping, adaptive sampling and isosurfacing;
3. Integration of our method into the OSPRay ray tracing library and open-source release[†] to make it widely available to domain scientists.

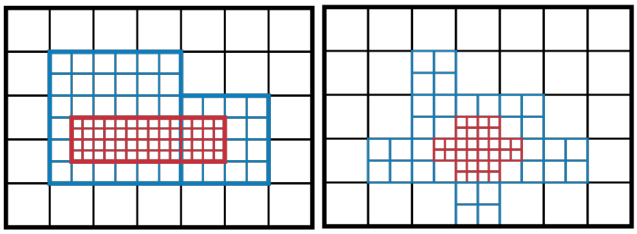
2. Background

AMR simulation techniques can be divided into two categories: *Block-Structured* (BS-AMR), also known as *patch-based* AMR, and *Tree-Based* (TB-AMR), also known as *point-wise structured* AMR. The fundamental difference between the two approaches is a trade-off between algorithm complexity and memory footprint [HLP17b].

BS-AMR grids are stored as a hierarchy of overlapping and successively finer uniform grids (see Figure 2a). The successive grids in BS-AMR data overlap, with data existing at all levels of refinement. In some packages, BS-AMR meshes also allow for more than one level of difference across level boundaries. In contrast, TB-AMR data are represented as trees, in which nodes can be refined when more details are needed in a region (see Figure 2b). For example, p4est encodes the computational domain using multiple linear octrees [BWG11]. The refinement levels of neighboring cells differ by one. A TB-AMR mesh typically requires less memory than its BS-AMR equivalent [HLP17b], since no redundancy lies at the finer region in the domain; however, TB-AMR meshes can be more complex and time consuming to perform visualization and analysis on [HLP17a].

AMR data can be further classified as vertex-centered or cell-centered, according to the relative location of the data points. Although designing an effective reconstruction strategy for vertex-centered AMR data is easier than for cell-centered data [WWW*19],

[†] https://github.com/ethan0911/TB-AMR



(a) *Block-Structured AMR* **(b)** *Tree-Based AMR*

Figure 2: *Examples of different AMR grids. (a) The mesh has four overlapping uniform grids; (b) arbitrary mesh cells can be refined as needed.*

the bulk of existing AMR frameworks operate on cell-centered grids [WCM12]. Unless otherwise specified, throughout this work we focus on cell-centered TB-AMR data, with a refinement factor of two. Although larger refinement factors have been used [DL19], a factor of two is most frequently found in practice [ME11, MOM*00, BWG11, AN14, Exa98]. Our approach assumes that the TB-AMR mesh employs a refinement factor of two.

3. Related Work

Generally, rendering volume data can be done using explicit isosurface mesh extraction [LC87], direct isosurface ray casting [PSL*98] or direct volume rendering (DVR) [DCH88]. Our work provides a ray-tracing based solution for the latter two. Ma and Crockett [MC97] are credited with the first AMR volume rendering approach, based on cell projection [Max93]. Kähler and Hege [KH02] employed a 3D slicing method and 3D texture proxies on the GPU, resampling AMR blocks while overlapping coarse- and fine-resolution AMR levels. This approach, which discards the original AMR grid hierarchy and maintains nonoverlapping blocks that contain only same-level cells, was further extended to direct ray casting (see, e.g., [KWAH06, KA13]). Gosink et al. [GABJ08] used out-of-core methods to query and resample AMR data into structured volumes for rendering on the GPU. Marchesin and de Verdiere [MDV09] performed special-case analytical ray casting for hexahedral cell data using piecewise-polynomial approximations.

To provide high-quality volume rendering of multiresolution datasets, Ljung et al. [LLY06] proposed interblock interpolation and used GPU fragment shaders to sample between level boundaries without replication. Beyer et al. [BHMF08] introduced a multilevel interpolation scheme based on a trapezoid and wedge decomposition. Although bearing some similarities to AMR, these approaches were designed for multiresolution rectilinear volume data as opposed to AMR grids, and require either explicit normalization of weights [LLY06] or texture coordinate remapping [BHMF08]. In contrast, our GTI method requires only a small set of fixed weights that are derived a priori and implicitly guaranteed to be normalized. Leaf et al. [LVI*13] showed a solution for rendering AMR data in distributed parallel settings, using an interpolation method similar to that of Ljung et al. [LLY06].

Correctly handling level boundaries of multiresolution data such as AMR remains a challenging problem, requiring data structures that faithfully represent the underlying data and efficient methods for stitching across levels. Van Gelder and Wilhelms [VGW94] con-

ducted a survey on this problem and introduced multiple solutions to address it. Specifically for AMR, Weber et al. [WKL*01, WKL*03] first proposed an approach that allows for extracting crack-free isosurfaces by introducing dual cells and unstructured mesh elements, extracting the isosurface from the produced unstructured mesh. Weber et al. [WCM12] later extended their work to run in parallel [WCM12]. Moran and Ellsworth [ME11] extended Weber et al.'s approach, and proposed a reconstruction method that allows for processing a Block-Structured AMR grid in which brick resolutions at level boundaries can differ by more than one level. To avoid constructing unstructured elements to stitch across boundaries, Wald el al. [WBUK17] explored several different options for smooth volume rendering of Block-Structured AMR data in a large-scale interactive CPU-based rendering framework. Wang et al. [WWW*19] further extended this work, and proposed a continuous and adaptive reconstruction method that allowed for implicit isosurface ray tracing of Block-Structured AMR data. Although these prior techniques can produce artifact-free visualizations, they are designed specifically for Block-Structured AMR data, and do not necessarily extend well for Tree-Based AMR.

Recently, Harel et al. [HLP17b, HLP17a] contributed a general approach for supporting Tree-Based AMR data in VTK (the vtkHyperTreeGrid). Similarly, Dubois and Lekien [DL19] demonstrated applications of the hypertree grid in processing and rendering large-scale AMR data. However, it is unclear how to directly perform interactive ray tracing on top of the presented structures for volume ray casting or implicit isosurface rendering. In this work, we provide an efficient solution that supports interactive high-quality visualization of Tree-Based AMR data in the widely used open-source ray tracing library, OSPRay.

4. Adaptive Octree Representation

Our approach entails representing Tree-Based AMR data as adaptive-resolution octrees to preserve the AMR hierarchy information. By preserving this information, we are able to achieve faster and higher quality rendering with less overhead than the alternatives commonly used in practice, e.g., flattening the mesh to an unstructured one, resampling it to a rectilinear grid or converting it to other proxy representations. In this section, we describe the adaptive octree layout that we use to store TB-AMR data to accelerate ray traversal and sample computation.

Broadly speaking, hierarchical data representation lends itself to logarithmic-time ray traversal and is key to interactively rendering large-scale datasets [WWJ19]. Moreover, a large number of TB-AMR simulations use an adaptive-resolution octree as the simulation mesh, and employing the same data structure for rendering enables us to easily support such simulations. We employ a compact sparse octree, somewhat similar to a Sparse Voxel Octree [KWH09, LK10], with the key difference that we support storing voxels of different resolutions, which is required to support AMR data. Our choice of data structure is motivated by the following:

1. Popular TB-AMR frameworks (e.g., Paramesh [MOM*00] and p4est [BWG11]) and solvers (e.g., Cart3D [AN14] and PowerFLOW [Exa98]) work on octrees or octree-like meshes with a 2:1 refinement ratio. Thus, an adaptive octree representation inherently fits such simulations well.

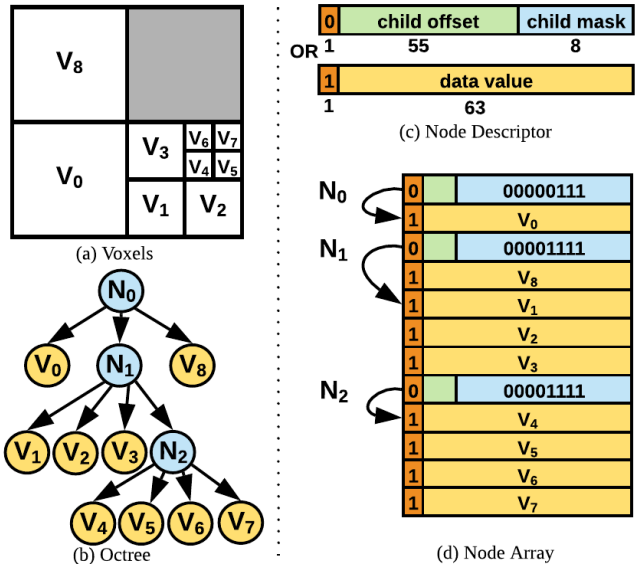


Figure 3: *The encoding of our octree hierarchy. Each leaf node corresponds to one input cell in the AMR mesh. Each inner node encodes which children exist in the tree and the offset to them using a 64-bit descriptor. The children are stored contiguously in the array.*

2. Other AMR meshes, such as BS-AMR, can be adapted to an octree hierarchy.
3. Octree traversal maps well to parallel ray traversal on multicore CPUs, and ensures the renderer can dynamically adjust the sampling rate to avoid over- and under-sampling.

The encoding used to store the adaptive octree representation is illustrated in Figure 3. Each node is represented using a 64-bit descriptor and the value range of its children, if any, and are stored in an array. For leaf nodes, we pack the single precision data value of the voxel directly in the descriptor. For inner nodes, we store information about which of its eight potential children exist, as the tree may be sparse, and the offset to its child nodes in the array. The children of each inner node are stored contiguously in the array, allowing them to be referenced by a single offset to reduce memory use. Each inner node's 64-bit descriptor is divided into an 8-bit mask indicating which children exist, a 55-bit offset to the children in the array and a 1-bit flag indicating whether the node is a leaf. The 8-bit child mask stores a 1 in the i'th bit if the i'th child exists, and a 0 if not. A particular child node can be fetched by incrementing the inner node's child offset based on the number of existing children before the desired one in the mask.

To accelerate both implicit isosurface rendering and volume ray tracing, we also store the value range as a pair of floats for each node, for an additional 64 bits. Although storing the value range requires additional memory, it enables empty-space skipping (Section 6.1) and active-cell filtering (Section 6.2.2), which significantly improve performance. We note that, for simplicity, our current approach redundantly stores these ranges for leaf nodes; however, this is not required and more optimized layouts are possible.

We construct the octree in an offline process using a simple top-down serial builder, which outputs the array of nodes to a binary file for use by the renderer. Although faster parallel octree builds can be

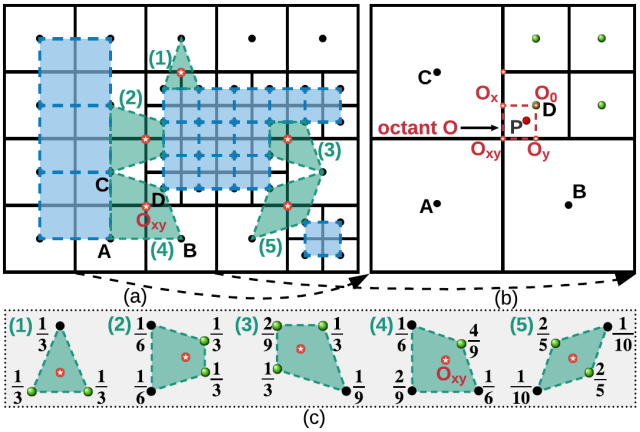


Figure 4: *A 2D illustration of the GTI method. (a) Five unstructured element configurations are used to tile the level boundary. (b) To compute the value of P, we use the GTI method to initialize the vertices of its containing octant, after which we can interpolate within the octant. (c) Weights for the five 2D cases. Points colored in green lie in the finer region, and red points denote the octant vertex being computed.*

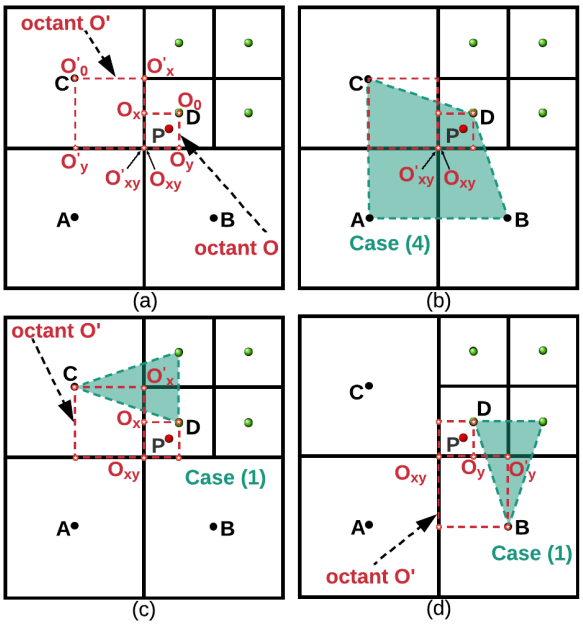


Figure 5: *A 2D illustration of interpolating the value of octant O's vertices. (a) A vertex in the finer side octant (O) asks the value from the coarser side octant (O'). (b) The computation of the value for O_{xy} fits into case (4). (c) The computation of the value for O'_x fits into case (1). (d) The computation of the values for O_x and O_y is symmetric.*

implemented, we did not find the builder to be a significant bottleneck relative to file IO. For example, the builder could construct the octree representation of the 656M cell ExaJet dataset in 160s.

5. Generalized Trilinear Interpolation (GTI)

Trilinear interpolation is widely used in the visualization of uniform grid scalar fields. When interpolating cell-centered data, dual cells with vertices at cell centers are introduced. Trilinear interpolation can then be performed within these dual cells. However, trilinear interpolation does not easily extend to AMR data, where interpolating across refinement level boundaries poses a challenge. At level boundaries, the dual cells do not line up, resulting in T-junctions [ME11, WCM12]. To reconstruct AMR data across level boundaries, a reconstruction method is needed to "stitch" across the boundary. Although in general there is no single "correct" solution for stitching, prior works [ME11, WCM12, WBUK17, WWW*19] have proposed several approaches that provide C^0 continuous results. However, current approaches can still result in artifacts. Our GTI method ensures C^0 continuity, adapts to the AMR mesh resolution, and enables artifact- and crack-free visualization.

Figure 4 illustrates our GTI method in 2D. Given a cell-centered AMR grid (Figure 4a), dual cells (colored in blue) are introduced in the reconstruction process. Samples that fall within these dual cells can be computed by simply performing trilinear interpolation using the values at the cell vertices. However, if the sample point is located at a level boundary between the dual cells, additional care must be taken to interpolate smoothly across the boundary.

Our GTI reconstruction process builds on top of the *octant* method [WWW*19], which subdivides the AMR cells into octants within which trilinear interpolation can be performed (octant O in Figure 4b). The key remaining challenge is in how to initialize each octant's vertices to achieve smooth interpolation. We achieve this in our approach by introducing "virtual unstructured elements" to stitch across the boundary, which we use to compute the octant vertex values. We note that these elements are not explicitly computed or stored. Following from our 2:1 refinement ratio assumption, each octant vertex falls into a fixed location within a limited number of such virtual unstructured elements: 5 in 2D (Figure 4c) and 20 in 3D (Figure 6). Thus, we can precompute the weights of each case, and when reconstructing an octant vertex, can determine the case and interpolate the four (in 2D) or eight (in 3D) vertices of the AMR cells forming the unstructured stitching element using the precomputed weights.

5.1. Weight Computation in 2D

To provide an intuitive understanding of the GTI method, we begin with a 2D example of computing the value at the point P located at the level boundary shown in Figure 5a. We step through the initialization of the octant vertices and explicitly derive the weights for case (4).

The point P is contained within an octant (a quadrant in 2D) in the finer level cell, within which we wish to perform bilinear interpolation. To do so, we must compute the values of the octant's vertices O_0, O_x, O_y and O_{xy}. The octant vertex O_0 is always located at the parent cell's center, and its value is known. The vertices O_x, O_y and O_{xy} lie on the level boundary and must be set to interpolate smoothly with the data on the coarser side. We use a combination of the octant method and our GTI method to set their values appropriately.

The vertex O_{xy} lies at the corner of the octant O, and is contained within the polygon formed by the cell centers A, B, C, and D. As mentioned, the value of O_{xy} is set from the coarser side octant (O'_{xy}).

To compute the value at O'_{xy} using GTI, we introduce the virtual unstructured element shown in case (4) (Figure 4), whose vertices are placed at A, B, C, and D. Following from our 2:1 refinement ratio assumption, the relative position of the vertices of the polygon and the location of O'_{xy} within the polygon are fixed, and we can precompute the interpolation weights for combining the known cell values at A, B, C, and D.

To derive the weights for case (4), we begin by assuming the field values at the polygon's vertices are related by a bilinear function f, where a, b, c, and d are real constants.

$$f(x,y) = ax + by + cxy + d \tag{1}$$

We express the interpolated value I at (x,y) as a weighted average (using weights c_i) of the known data values at the cell centers:

$$\begin{aligned} I(x,y) &= c_A A + c_B B + c_C C + c_D D \\ &= \sum_{i=A,B,C,D} c_i f(x_i, y_i) \end{aligned} \tag{2}$$

The above is subject to a constraint, namely, the interpolation property must be satisfied. For $p \in \{A,B,C,D\}$, the following must hold:

$$f(x_p, y_p) = I(x_p, y_p) \tag{3}$$

Since the above equations hold for all f, they must hold for any particular f. Thus we can derive the following constraints:

$$\begin{aligned} 1 &= \sum_i c_i && \leftarrow && f(x,y) = d, \quad d \neq 0 \\ x_p &= \sum_i c_i x_i && \leftarrow && f(x,y) = x \\ y_p &= \sum_i c_i y_i && \leftarrow && f(x,y) = y \\ x_p y_p &= \sum_i c_i x_i y_i && \leftarrow && f(x,y) = xy \end{aligned}$$

We can compute the weights for each case by substituting the coordinates of the vertices A, B, C, D into the above constraints and solving the resulting system of equations. To compute the weights for case (4), we set O'_{xy} as the origin and, following from the 2:1 refinement ratio, can set the coordinates of the polygon vertices as $A = (-2,-2)$, $B = (2,-2)$, $C = (-2,2)$, $D = (1,1)$. These relative positions are the same regardless of the actual AMR level, as the ratio of the coarser to finer cells is always 2:1, and thus any constant scaling factor cancels out.

$$c_D = 2c_A, \quad c_B = c_C, \quad 1 = 3c_A + 2c_B, \quad 0 = 2(3c_A - 4c_B) \tag{4}$$

Finally, we solve algebraically for the weights required to reconstruct the field value at O'_{xy} within case (4):

$$c_A = \frac{2}{9}, \quad c_B = \frac{1}{6}, \quad c_C = \frac{1}{6}, \quad c_D = \frac{4}{9} \tag{5}$$

Notice D has the highest weight, which is intuitive, given that it is closer to P than the other vertices. The weights of other cases can be derived following the same principles.

The computation of the values for O_x and O_y is symmetric (Figure 5c,d). Both vertices lie along the edge with the coarser side at the middle of the parent cell's edge, i.e., at the midpoint of the octant on the coarser side. To ensure the values computed for these vertices interpolate smoothly from the coarser side, we set their values using the coarser side octant. To do so, we must compute a subset of the coarser octant's vertices. Specifically, we must find the value for O'_x to compute O_x (Figure 5c), and O'_y to compute O_y (Figure 5d). The value of the finer side octant vertex is then the average of O_{xy} and O'_x or O'_y, for O_x and O_y respectively. As shown in Figure 5c,d, the computation of O'_x and O'_y falls into case (1). We can fetch the cell values corresponding to the triangle's vertices and use the precomputed weights for the case to determine the value at O'_x and O'_y.

5.2. Weight Computation in 3D

Stitching across level boundaries is more complex in 3D; however, the GTI method extends directly from the intuition given in the 2D example to 3D. To reconstruct the field at the sample point P located in the right-up-far octant of the cell C (Figure 6a), we must compute the values of the vertices of the right-up-far octant containing the point, after which we can perform trilinear interpolation in the octant. In 3D, this requires setting the values of the eight vertices forming the octant: the cell center O_0; the side vertices O_x, O_y, and O_z; the edge vertices O_{xy}, O_{yz}, and O_{xz}; and the corner O_{xyz}. The vertex O_0 is located at the cell center, and its value is known. The remaining seven vertices are computed analogously to the 2D case, using a combination of the octant and GTI methods to achieve smooth interpolation across the boundary.

When the neighboring cells to the side are refined, the corresponding side vertex (O_x, O_y, O_z) is shared by five cells (one coarser cell and four finer cells). The side vertex can be enclosed in a pyramid, after which the interpolation can be reduced to the 2D case (1) by first interpolating along the base of the pyramid to form case (1) (Figure 7a). Similarly, each edge vertex (O_{xy}, O_{yz}, O_{xz}) is shared by four neighboring cells (e.g., cell 0, 2, 4 and 6 for O_{yz}). Depending on which neighbors are refined, the interpolation can be simplified to 2D cases (2)-(5) (Figure 7b).

The corner vertex O_{xyz} is shared by eight neighboring cells, and 20 cases emerge for computing its value depending on which of those neighbors are refined (Figure 6). Here we provide a brief overview of the derivation and how it follows from the 2D example. For the full derivation, please see the appendix.

Equation (1) from the 2D case, extended to trilinear interpolation in 3D, can be seen as a linear combination where $\mathbf{x} = (x,y,z)^T$:

$$f(x,y,z) = f(\mathbf{x}) = \sum_{i=1}^{N} k_i \phi_i(\mathbf{x}) \tag{6}$$

where we choose N and the basis functions ϕ_i, with unknown coefficients k_i. In 3D, $N = 8$, and $\phi_1(\mathbf{x}) = 1$, $\phi_2(\mathbf{x}) = x$, …, and $\phi_8(\mathbf{x}) = xyz$. We assume a coordinate frame that places our "target vertex" O_{xyz} at the origin (Figure 6). Given N known pairs $(\mathbf{x}_i, I(\mathbf{x}_i))$, we require that Equation (3) holds, which leads to the following system, where V is a multivariate Vandermonde matrix [HK]:

$$\mathbf{I} = V\mathbf{k}, \quad V_{ij} = \phi_j(\mathbf{x}_i) \tag{7}$$

Let $\mathbf{x}_{O_{xyz}}$ denote the coordinates of O_{xyz}. As we have set $\mathbf{x}_{O_{xyz}}$ as the origin, $f(\mathbf{x}_{O_{xyz}}) = f(\mathbf{0})$. Thus:

$$f(\mathbf{x}_{O_{xyz}}) = \mathbf{e}_1{}^T\mathbf{k} = (\mathbf{e}_1{}^T V^{-1})\mathbf{I} = \mathbf{c}^T\mathbf{I} \tag{8}$$

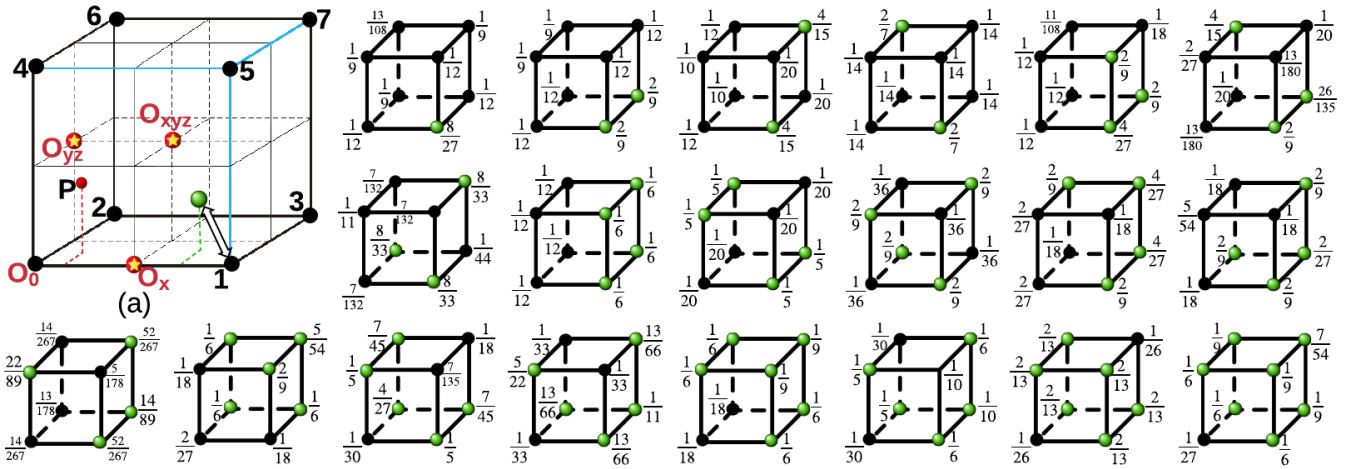


Figure 6: *Illustration (not to scale) of the 20 cases for reconstructing the corner vertex O_{xyz} of an octant that lies on the coarser side of the boundary. The left-down-near point in each case is coincident with O_0 in (a). Green points denote data points on the finer side of the boundary. The derived weights for interpolating O_{xyz} in each case are listed in the figure.*

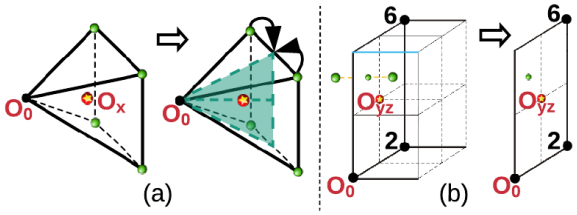


Figure 7: *The sampling of a side vertex (O_x) or an edge vertex (O_{yz}) in 3D can be simplified to a 2D case: (a) can be simplified to 2D case (1), (b) can be simplified to 2D case (4).*

With $\mathbf{e}_1$ denoting the canonical basis vector $(1,0,\ldots,0)^T$, $\mathbf{k}$ denoting the vector of coefficients $(k_1,k_2,\ldots,k_N)^T$, and $\mathbf{c}$ being the solution of the system:

$$V^T\mathbf{c} = \mathbf{e}_1 \tag{9}$$

When interpolating between eight cell centers as in Figure 6, there are 20 different configurations of the cell centers ($\mathbf{x}_i$), modulo symmetry. Each case defines a different V and $\mathbf{c}$ by Equations (7) and (9). For each case, once $\mathbf{c}$ is known, the field value at O_{xyz} can be computed as follows:

$$\mathbf{I}(\mathbf{x}_{O_{xyz}}) = \mathbf{c}\cdot\mathbf{I} \tag{10}$$

In other words, $\mathbf{c}$ can be thought of as a vector of interpolation weights. As before, the configurations are fixed and the weights for each case can be precomputed. The weights for each case were found numerically. Their solutions are demonstrated in the SymPy notebook provided in the supplemental material and the detailed derivation provided in the appendix.

6. Rendering of TB-AMR Data

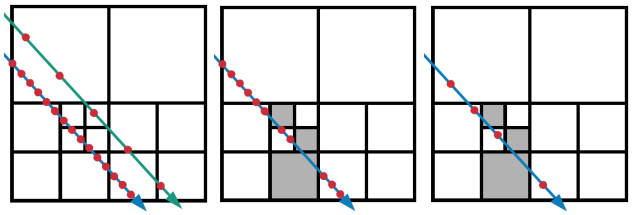
We implement our TB-AMR volume as a module for the OSPRay [WJA*16] CPU ray tracing framework, using OSPRay version 1.8. OSPRay is an interactive CPU ray tracing API and engine for scientific visualization and photorealistic rendering. Internally, OSPRay builds on Embree [WWB*14] for ray traversal, Intel's Thread Building Blocks (TBB) for multithreading and the Intel SPMD Program Compiler (ISPC) [PM12] for vectorization. Our module can be used for interactive high-quality rendering of TB-AMR data in our layout with both direct volume rendering (Section 6.1) and implicit isosurface rendering (Section 6.2).

To perform interactive direct volume ray tracing and enable fast determination of cells containing a desired isosurface, we must perform many top-down queries to look up dual and leaf cells containing a query point. Given the desired point(s) and optional transfer function or isovalue, we traverse the octree top-down using a software-maintained stack. The optional transfer function and isovalue are used in combination with the node's value range to determine whether a node is fully transparent and can be skipped.

When interpolating within a nonboundary region, we must find the values of the eight corners of the dual cell to interpolate between. As the corners of the cell are likely to exist down similar subtrees of the octree, it is beneficial to traverse all eight points at once, as opposed to performing eight top-down traversals. To do so, *findDualCell* uses a bitmask to track which of the eight query points require traversal of each child node being considered, and stores this information along with the node ID in the stack. When interpolating at the level boundary, we must find the corresponding leaf cells that enclose the virtual unstructured element for our GTI method. To find these cells, the *findLeafCell* traversal takes a query point and traverses it to the bottom of the tree, returning the leaf cell containing the point.

6.1. Direct Volume Rendering (DVR)

Volume types in OSPRay need to implement only a `sample` function (e.g., the GTI method) to sample the continuous field at a 3D point in the volume, and a `stepRay` function to advance the ray to the next sample point. However, OSPRay 1.8's volume rendering is built on taking a fixed step size (see Figure 8a) and does not support adjusting the opacity when changing the step size dynamically along the ray, as is required to adaptively sample an AMR volume. These limitations have been addressed in OSPRay 2.0, but were not available for our



(a) *Fixed step size* **(b)** *Skip empty space* **(c)** *Adaptive sampling*

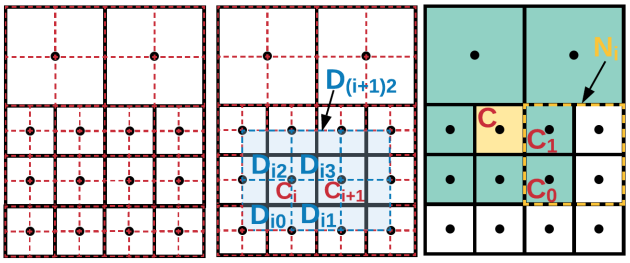
Figure 8: *Ray traversal through a TB-AMR volume. (a) A fixed step size will lead to severe over-sampling (blue ray) or under-sampling (green ray). We perform a ray-octree traversal with (b) empty space skipping and (c) adaptive sampling, to both skip empty space and sample the data based on the underlying data resolution.*

implementation. Without proper support for adaptively sampling the data, our renderer would either produce artifacts or have to severely over- or under-sample the data.

Therefore, in our current implementation, we modified OSPRay's scientific visualization renderer to support traversing and integrating the TB-AMR volume over a given ray interval. We employ a standard front-to-back octree ray traversal, where inner and leaf nodes are skipped if their contained value range is entirely transparent, and the integration step size is adjusted to the width of the leaf cell being sampled (Figure 8). To determine if a node and its children are entirely transparent, we use OSPRay's built-in preintegrated transfer function. OSPRay's preintegrated transfer function supports arbitrary 1D opacity configurations and provides a method to query the maximum opacity over a given value range. If the returned max opacity is 0, we can skip the node and its children. To compute samples for the ray interval, we employ a user-selected reconstruction kernel to compute the sample value (e.g., finest [WBUK17], octant [WWW*19] or our GTI method). Our octree traversal ensures that each cell can be sampled accurately, and our GTI method ensures the samples reconstructed in the cells are continuous, even at level boundaries.

6.2. Implicit Isosurface Ray Tracing

A standard approach for isosurface rendering is to perform explicit isosurface extraction, producing a set of triangles forming the surface. In the case of TB-AMR data, explicit isosurface extraction could be done by treating the octants as the primitives containing the surface, where our GTI method can be used to set the values at the vertices of each octant (Figure 9a). Marching cubes could then be performed within each octant, for example. However, the number of triangles that would be produced would consume a large amount of memory and be expensive to render. An alternative is to use implicit isosurface ray tracing, where we perform an implicit ray-isosurface intersection, rather than explicitly extracting surface geometry (see, e.g., [PSL*98, MKW*04, WFM*05, KWH09, WWW*19]. In our approach, we adopt the hybrid implicit isosurface method of Wang et al. [WWW*19], which was used in combination with their octant reconstruction method for BS-AMR data. Hybrid implicit isosurface ray tracing extracts a list of active octants as a set of vertex-centered primitives that are placed into an Embree [WWB*14] bounding volume hierarchy (BVH) to accelerate traversal. Within each primitive, a ray-implicit isosurface intersection is performed [MKW*04]. Two



(a) *Tiling with octant* **(b)** *Apply dual cell* **(c)** *Active cell filtering*

Figure 9: *Optimizations on extracting the active cubic primitives. (a) The naïve approach tiling the entire domain with octant (red cube) and resampling the octant's vertex value. (b) Applying dual cells (blue cube) at the nonboundary region rather than the octant. (c) Filter active cell by checking if the value range of cell C and its neighboring cells (green) overlap with the isovalue.*

processes are required to perform hybrid implicit isosurface ray tracing: 1) extracting a list of vertex-centered primitives from the cell-centered TB-AMR data and computing the values of the primitive's vertices and 2) filtering the active primitives (a primitive is an active primitive if its value range overlaps with the user-specified isovalue) and performing the ray-isosurface intersection over those active primitives to render the isosurface. We use our GTI method for step 1 to ensure a continuous surface is rendered, and our octree layout to accelerate the cell filtering in step 2.

In step 2, OSPRay uses Embree to build a BVH over the active primitives and traverse rays through the BVH. The Embree BVH builder is quite efficient (achieving $\sim$ 110 million primitives per second), and the ISPC intersection within each primitive is effectively vectorized. The main performance bottleneck lies in the first process where resampling is required. A naïve approach involves using an octant as the primitive (Figure 9a). To extract the primitives, we iterate all the input cells, generate eight octants for each cell and compute the value of the octants' vertices using the GTI method. Although simple and straightforward, this approach incurs a massive amount of redundant computation. For example, each vertex of an octant is recomputed eight times, since it is shared by eight neighboring octants. To improve the performance of the first process, we propose two optimizations: 1) using dual cells at nonboundary regions (Section 6.2.1) and 2) filtering out input cells that do not contain the isovalue before extracting dual cells and octants (Section 6.2.2).

6.2.1. Optimization 1: Using Dual Cells at Same-level Regions

One optimization that allows for significantly reducing the number of primitives that must be traversed by the hybrid implicit isosurface renderer involves using dual cells in same-level regions instead of octants. A dual cell in a nonboundary region is enclosed with data points of known values, for which no costly reconstruction kernel or boundary stitching is required. More importantly, Figure 9b shows that an octant of a cell remains an octant of the corresponding dual cell, which ensures that the final isosurface does not change when eight octants are merged into a dual cell in the same-level region. With this optimization, we achieve an approximate 4$\times$ speed-up in extraction time. A detailed benchmark is done in Section 7.3.

Although Wang et al. applied the same optimization for BS-AMR data in [WWW*19], the implementation for TB-AMR data is more complicated due to the different data layout. BS-AMR data are stored as a list of uniform grids, with each grid storing a set of cells at the same AMR level. For BS-AMR data, it is relatively easy to construct the dual cells, as we know all the cells in a brick are at the same level. However, for TB-AMR data, the cells are stored in octree, and we do not readily have access to information about the neighboring cells. This makes the decision of whether to construct a dual cell or an octant complicated, because we will need to check if the current cell lies in a same-level region

Algorithm 1: *Process of extracting cubic primitives (dual cell and octant) with TB-AMR data (See Figure 9b.)*

```
1  Mesh genActiveCubicPrimitives(Mesh input, float isovalue)
2    Mesh res;
3    for (int i=0; i < input.num_cells(); i++)
4      C_i = input.GetCell(i);
5      for (int j=0; j < 8; j++)
6        D_ij = findDualCell(C_i,j)
7        bool inSamelvl = isDualCellInSameLevel(D_ij)
8        if(inSamelvl) //dual cell's vertices are located in same level
9          if(j == 7 && D_ij.ValueRange().Contains(isovalue))
10           res.insert_cell(D_ij);
11       else // parent cell is at a level boundary
12         O_ij = initOctant(C_i, j)
13         GTI_method(O_ij)
14         if(O_ij.ValueRange().Contains(isovalue))
15             res.insert_cell(O_ij);
16   return res;
```

When extracting the list of active primitives (octants or dual cells), we iterate through the cells in the input mesh to create primitives corresponding to each cells' octants or dual cells (see Algorithm 1). In 3D, each cell C_i (Figure 9b) can produce up to eight octants (O_{ij}), or be shared by up to eight dual cells (D_{ij}, $j \in \{0 \dots 7\}$), where its center is a vertex of the octant or dual cell, respectively.

As each dual cell D_{ij} contains the octant O_{ij}, for each octant we must first determine if the dual cell could be emitted instead; otherwise, the emitted octant and dual cell would overlap. To do so, we compute the dual cell's vertices and traverse them through the octree to determine if each vertex lies at the same AMR level (lines 6-7, Algorithm 1). If this is the case, we know a dual cell can be constructed, and we skip emitting the octant. Each cell center is shared by up to eight dual cells. Thus, to avoid emitting redundant dual cells, each cell is responsible for emitting only its last dual cell, D_{i7}. If the vertices of the dual cell are not all on the same level, the cell lies at a level boundary and an octant must be emitted instead. The octant's vertex values are initialized using our GTI method (lines 12-13, Algorithm 1). Finally, if the computed dual cell or octant does not contain the isosovalue, it is discarded.

6.2.2. Optimization 2: Filtering Active Cells

Optimization 1 allows for dramatically reducing the amount of output active primitives that are fed into Embree for intersection. However, this optimization as well as the naïve approach (pure octant) performs the extraction over the list of all input cells. This approach is fairly inefficient, since we will construct dual cells or octants for all input cells, even if the isosurface does not pass through the input cell. What makes matters worse is that the number of input cells is usually very large in HPC simulations (e.g., 656M cells for

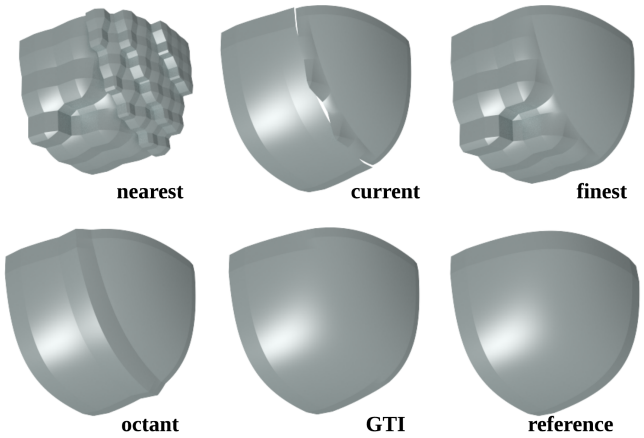


Figure 10: *Isosurface extracted on the synthetic data with different reconstruction methods (isovalue = 6.5). The nearest and current method is incontinuous; the finest method is nonadaptive; the octant method is both continuous and adaptive, but produces artifacts at the boundary; our GTI method shows an artifact-free result that is closest to the reference result.*

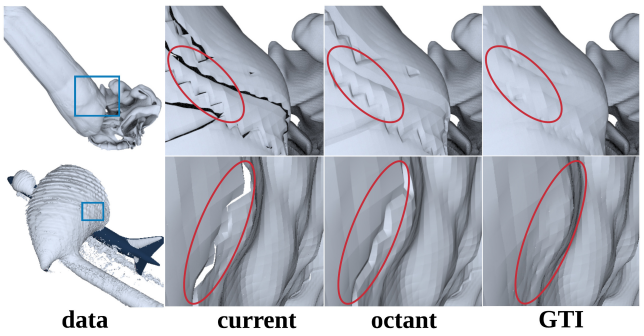
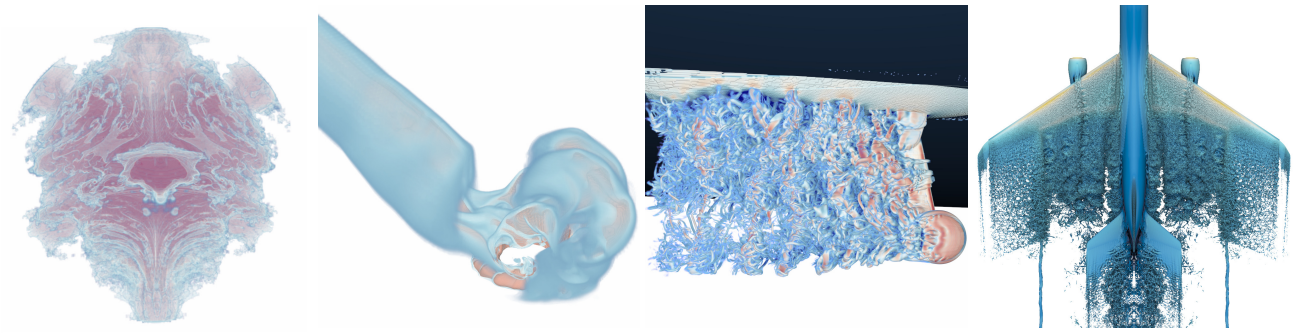


Figure 11: *Isosurface extracted on real datasets with three reconstruction methods. Top: LANL meteor-20060 (field: tev). Bottom: the NASA ExaJet data (field: density). Our GTI method achieves the highest quality isosurface.*

the NASA ExaJet data). A desirable additional optimization that would significantly improve performance would be to consider only those input cells that do contain the isosurface. The input cell C (see Figure 9c) is labeled an "active cell" when the value range of C and its neighbors (colored in green) contains the specified isovalue. If it can be determined that C is not active, i.e., the value range does not contain the isovalue, we can discard it from the input.

To check if the input cell C is an active cell, we traverse the octree from the root and return *true* if we find any neighbor of C whose value range unioned with C's contains the isovalue. A naïve approach would traverse to the leaves of the tree, querying each neighbor's value and testing if the value range overlaps the isovalue one by one. However, we can leverage the value range stored at each inner node to terminate traversal early, when it is found that the inner node's value range does not contain the isovalue. Given two neighboring cells that are descendants of an inner node N_i, we know that if the value range of N_i does not contain the isovalue, its children cannot contain the isovalue and do not need to descend the



(a) *p4est-Mandelbulb, 2.6M cells* **(b)** *LANL Asteroid Impact ($t = 20060$), 158M cells* **(c)** *NASA Landing Gear, 262M cells, 1.59M tris* **(d)** *NASA ExaJet + mirrored instance, 1.31B cells, 126M triangles*

Figure 12: *High-fidelity visualization of the different datasets (image resolution: 1000×1000, our GTI reconstruction method is applied). (a) DVR of Mandelbulb only (0.31 fps); (b) combined DVR and isosurface of meteor (0.94 fps); (c) combined DVR and isosurface of NASA LandingGear (5.02 fps); (d) isosurface raycasting with ambient occlusion of NASA exajet (14.28 fps).*

tree further. We show that the extraction time can be considerably reduced with this optimization in Section 7.3.

7. Results

In this section, we evaluate three key aspects of our system: the final image quality of the GTI method (Section 7.1); rendering performance (Section 7.2); isosurface extraction time (Section 7.3).

Evaluation Hardware. We perform our benchmarks on a quad-socket workstation, equipped with four Intel Xeon E7-8890v3 CPUs (2.5 GHz base clock), with a total of 72 physical cores (144 threads) and 3 TB RAM.

Data Description. We evaluate our GTI method on five datasets, ranging from small to large and covering both TB-AMR and BS-AMR data:

- **Synthetic**: is a small dataset (288 cells) that is generated on the fly from a 4^3 uniform grid by refining the right-half cells. Of particular interest for this dataset is that we can define a test function over the mesh and get a reference result to compare with.
- **p4est-Mandelbulb**: is a 3D fractal, constructed using spherical coordinates by White and Nylander [WN]. This data is created in p4est and demonstrates our system's capablities for rendering TB-AMR data.
- **Meteor**: is a simulation of an asteroid impact in deep ocean water from LANL [PG17]. Multiple timesteps of this data are available for the benchmark. We use timestep 20060 in our benchmarks.
- **LandingGear**: was originally BS-AMR data that was produced by NASA for simulating the air flow around an aircraft's landing gear assembly. It is also used in [WBUK17, WWW*19]. Here, we load it into OSPRay and convert it into the TB-AMR format for our use. 262 M cells are stored in the exported file.
- **Exajet**: is a Cartesian grid AMR dataset produced by NASA using PowerFlow [Exa98], simulating the air flow around a jet. This model contains 656M cells across four levels along with 63.2M triangles, representing half of the jet. To create a visualization of the entire jet, we create a mirrored instance of the input jet using OSPRay's instancing functionality, for a total of 1.31B cells and 126M triangles.

7.1. Image Quality with Different Interpolants

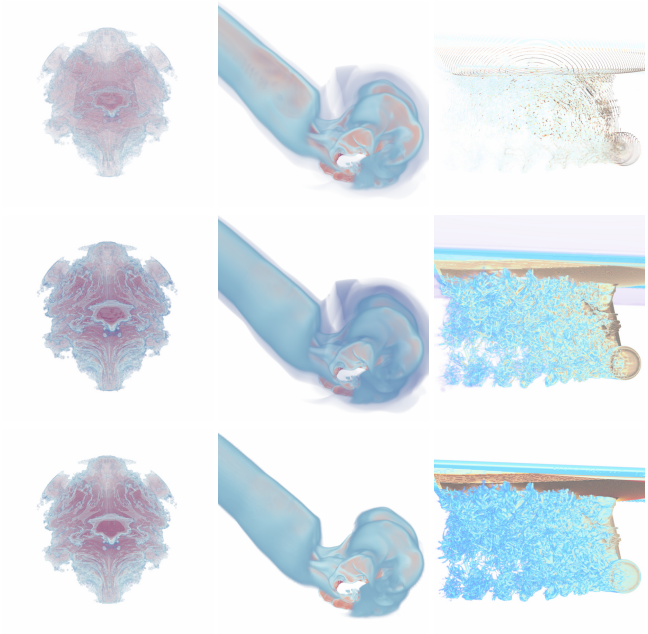
Different reconstruction methods result in isosurfaces of varying smoothness and quality. To evaluate the functionality of our GTI method, we compare it with four strategies proposed in the related research [WBUK17, WWW*19]. We perform the benchmark on synthetic data with each cell's value computed with a test function $f(x,y,z) = xyz$. We run this experiment on synthetic data, so that we can generate a reference result for comparison. The reference image shows an isosurface generated with a 4^3 uniform grid using trilinear interpolation (Figure 10). In contrast, isosurfaces in other images are extracted from an adaptive mesh, which is generated from the uniform grid by refining voxels at the right side of the boundary ($x = 2$).

Figure 10 shows the isosurface generated with $isovalue = 6.5$. We observe that the *nearest* method produces severe artifacts in the visualization; the *current* method is adaptive but yields cracks at the boundary. The *finest* method stitches at the boundary but loses adaptivity, and thus we see some artifacts at the coarser side. The *octant* is continuous and adaptive, but it results in some unsmooth artifacts at the boundary. Our *GTI* method produces a crack-free isosurface that is visually closest to the underlying data.

We then test the reconstruction methods with the LANL meteor and NASA ExaJet dataset (Figure 11). We compare three methods: *current*, *octant* and *GTI*. As with the synthetic data result, we find that the GTI method produces the highest quality crack-free isosurface.

7.2. Rendering Performance

Interactive rendering of BS-AMR data in OSPRay has previously been demonstrated (see, e.g., [WBUK17, WWW*19]). However, support for direct volume rendering or implicit isosurface rendering of TB-AMR was limited prior to our work. A typical way to render TB-AMR data with OSPRay or other off-the-shelf tools is to represent the data as an unstructured mesh. In this section, we compare rendering performance and memory consumption when visualizing four datasets with OSPRay's unstructured mesh renderer and our approach. We perform a comparison only for direct volume rendering, since OSPRay's built-in isosurface raycasting functionality for



Mandelbulb **Meteor** **LandingGear**

Figure 13: *Output images for benchmarks in Table 1. Top row: volume rendering using OSPRay's unstructured renderer with* `samplingRate` *= 5. Middle row: rendering using OSPRay unstructured renderer with* `samplingRate` *> 5, tuned such that the image best matches the output of our method. Bottom row: volume rendering using our approach and the GTI interpolation strategy.*

		OSPRay Unstructured			Ours	
Data	Cells	Low SR	High SR	Mem	FPS	Mem
p4est-mandel	2.6M	3.31	1.40	0.83	0.33	0.46
Meteor-20060	158M	1.42	0.83	40.66	0.96	4.62
LandingGear	262M	1.72	0.06	51.85	3.10	7.41
ExaJet	656M	N/A	N/A	N/A	2.25	33.72

Table 1: *Volume rendering benchmarks on the quad-socket workstation. "Mem" denotes the peak resident memory size in GB. FPS denotes the number of frames per second. For "low SR",* `samplingRate` *was set to 5 for all datasets. For "high SR",* `samplingRate` *was set to 10, 9 and 160 for Mandel, Meteor and LandingGear, respectively.*

the unstructured mesh failed on three of the four tested datasets. However, our approach performs well when conducting implicit isosurface ray tracing—isosurface rendering can be performed at 14.28 FPS (average) for the largest dataset tested (the NASA ExaJet, Figure 12d).

OSPRay's existing unstructured volume renderer advances the ray using a single fixed increment, which is determined by $samplingStep \times samplingRate$. The `samplingStep` is roughly the cell's width, whereas the `samplingRate` is a user-defined value that is approximately the number of samples per cell. When rendering data on an uniform grid, a lower value of `samplingRate` will result in higher rendering performance at the cost of image quality. For an adaptive mesh, choosing an appropriate `samplingRate` is a difficult task. For example, in the LandingGear dataset, the ratio between coarsest and finest cell width can be as high as 4096:1. A `samplingRate` that is high enough to sample finer cells well will severely over-sample coarser cells.

				Isosurface extraction time (s)		
Data	Cells	ISO	Active	Naive	Opt. 1	Opt. 2
Synthetic	288	6.5	56.61%	0.02	0.03	0.02
p4est-mandel	2.6M	0.9	64.84%	3.68	8.98	5.63
Meteor-20060	158M	0.2	1.44%	114.08	56.82	6.68
LandingGear	262M	99K	7.08%	243.21	59.31	14.46
ExaJet	656M	1.2	0.83%	505.96	109.81	19.13

Table 2: *The isosurface extraction time of different approaches. For each dataset, we show the input cell number, isovalue, active cell percentage (active cell / input cell) and extraction time (s).*

As part of our benchmark, we measured the performance of OSPRay's unstructured renderer with low and high values of `samplingRate` ("low SR" and "high SR", respectively, in Table 1.) For "low SR", `samplingRate` was set to constant for all datasets, whereas for "high SR", `samplingRate` was adjusted in order for the result to best match the output of our tree-based algorithm. See the caption of Table 1 for specific sampling rate values.

Performance and memory consumption of both methods are demonstrated in Table 1, and the output images are shown in Figure 13. OSPRay's unstructured renderer crashed when rendering the ExaJet dataset, likely due to the enormous number of primitives. Thus, we report only the result of our approach for that dataset. When the unstructured renderer is set to "high SR", it produces images with similar quality to our approach, yet our framerates are comparable or higher. By contrast, when using "low SR" with the unstructured renderer, our approach can render images of higher quality, while maintaining comparable framerates. The Mandelbulb dataset (which is the smallest) is the lone exception; for all real-world large datasets, our approach is significantly faster than OSPRay's unstructured mesh renderer.

7.3. Isosurface Extraction Time

Besides rendering performance, it is worth considering the time cost of isosurface extraction. In this section, we benchmark the processing time of extracting the active cubic primitives, and the scalability of our system as the chosen isosurface grows in complexity (higher active-cell percentage).

Table 2 shows the extraction time when applying the naïve approach and two optimizations on five datasets. From this table, we observe that the optimizations hamper the performance for a small dataset (e.g., synthetic and p4est). Performing these optimizations does take additional compute time, which on small datasets may not pay off compared to a brute force approach. However, the optimized approaches performed much better than the naïve approach when it comes to medium or large datasets. The speed-ups of the two optimizations are 2×-5× and 16×-27×, respectively. Furthermore, our system is capable of processing the LandingGear dataset (262 million cells) in 15 s, which is similar to the performance demonstrated in [WWW*19] by using KD-tree.

In terms of scalability, we benchmark extraction time with increasing active cell percentage. Theoretically, a higher active cell

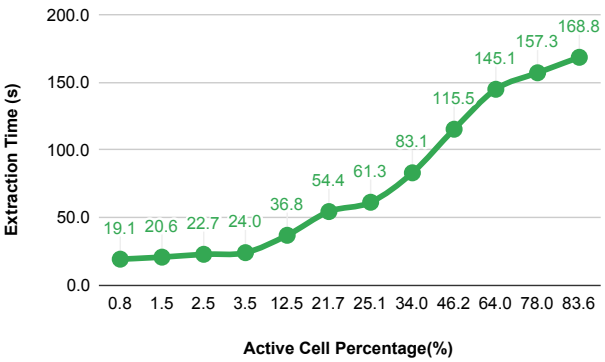


Figure 14: *The processing time (with both optimizations applied) of extracting isosurface for the NASA Exajet dataset vorticity field for different isovalues. A different isosurface will result in various active cell percentages.*

percentage, along with a more complex isosurface, indicates more data to process and thus a longer processing time. Particularly in our experiment, we specify different isovalues to the NASA exajet data (field: vorticity) and depict the result in Figure 14. We conclude that the extraction time positively correlates with the active cell percentage. Our extraction time increases only by $8.2\times$ ($20.6s \rightarrow 168.8s$) when the active cell percentage goes up by $55.7\times$ ($1.5\% \rightarrow 83.6\%$), likely due to better memory access patterns despite the increased workload.

8. Conclusion

In this work, we have presented an efficient solution for interactive high-fidelity visualization of cell-centered tree-based AMR data. To correctly handle interpolation at level boundaries, we proposed a novel reconstruction strategy—Generalized Trilinear Interpolation—-which is continuous and adaptive, and allows for producing higher quality visualizations than the current state of the art [WBUK17, WWW*19]. Besides TB-AMR data, GTI can be applied to other AMR or multiresolution grids to produce a smooth reconstruction field. To organize the "flattened" TB-AMR data, our solution employs a sparse octree to which arbitrary AMR formats can be easily adapted and coupled. In addition, we implemented vectorized traversal kernels on top of the hierarchy to support fast data query on multicore CPU architectures. Along with empty-space skipping, adaptive sampling and isosurface extraction optimizations, our system is capable of performing interactive high-fidelity visualization of large-scale TB-AMR data with volume and isosurface ray tracing and advanced shading effects.

We integrated our approach into the OSPRay ray tracing library as a module and released it as an open-source module. As OSPRay is integrated into ParaView and VisIt [WBUK17, WUP*18], our module can be leveraged by visualization practitioners and domain scientists for rendering TB-AMR data, especially within the p4est community. In the future, we hope to investigate better implementations of volume rendering traversal and sampling using neighbor-finding techniques to locate dual cells and reduce redundant traversal. In addition, a more compact representation of our TB-AMR data structure should be possible and would be desirable. Lastly, although our work was done on OSPRay 1.8, OSPRay 2.0 brings significant changes that we will incorporate into our module to make it available through the new OpenVKL library and OSPRay 2.0 API.

Acknowledgement

This project was supported in part by the Intel Graphics and Visualization Institutes of XeLLENCE and the National Institute of General Medical Sciences of the National Institutes of Health under grant number P41 GM103545-18. This work is supported in part by NSF: CGV Award: 1314896, NSF:IIP Award: 1602127, NSF:ACI Award: 1649923, DOE/SciDAC DESC0007446, CCMSC DE-NA0002375 and NSF:OAC Award: 1842042. The authors wish to thank Patrick Moran for the NASA ExaJet and Landing Gear datasets, and Ingo Wald for assistance processing the NASA LandingGear dataset. We would also like to thank the reviewers for assistance improving the manuscript.

References

[AN14] AFTOSMIS M. J., NEMEC M.: Cart3D simulations for the first AIAA sonic boom prediction workshop. In *52nd Aerospace Sciences Meeting* (2014).

[Aya15] AYACHIT U.: *The ParaView guide: A parallel visualization application*. Kitware, Inc., 2015.

[BHMF08] BEYER J., HADWIGER M., MÖLLER T., FRITZ L.: Smooth mixed-resolution GPU volume rendering. In *Proceedings of the Fifth Eurographics/IEEE VGTC conference on Point-Based Graphics* (2008), pp. 163–170.

[BO84] BERGER M. J., OLIGER J.: Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of computational Physics 53*, 3 (1984), 484–512.

[BWG11] BURSTEDDE C., WILCOX L. C., GHATTAS O.: p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing 33*, 3 (2011), 1103–1133.

[CBW*12] CHILDS H., BRUGGER E., WHITLOCK B., MEREDITH J., AHERN S., PUGMIRE D., BIAGAS K., MILLER M., WEBER G. H., ET AL.: *VisIt: An end-user tool for visualizing and analyzing very large data*. Tech. rep., Lawrence Berkeley National Laboratory, 2012.

[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.

[DAB*14] DUBEY A., ALMGREN A., BELL J., BERZINS M., BRANDT S., BRYAN G., COLELLA P., GRAVES D., LIJEWSKI M., LÖFFLER F., O'SHEA B., SCHNETTER E., VAN STRAALEN B., WEIDE K.: A survey of high level frameworks in block-structured adaptive mesh refinement packages. *Journal of Parallel and Distributed Computing* (2014).

[DCH88] DREBIN R. A., CARPENTER L., HANRAHAN P.: Volume rendering. *ACM SIGGRAPH Computer Graphics (Proceedings of the 15th annual conference on Computer graphics and interactive techniques - SIGGRAPH '88) 22*, 4 (1988).

[DL19] DUBOIS J., LEKIEN J.-B.: Highly efficient controlled hierarchical data reduction techniques for interactive visualization of massive simulation data. In *EuroVis 2019 - Short Papers* (2019), The Eurographics Association. doi:10.2312/evs.20191167.

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

[GABJ08] GOSINK L. J., ANDERSON J. C., BETHEL E. W., JOY K. I.: Query-driven visualization of time-varying adaptive mesh refinement data. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008).

[HK] HARDER D. W., KHOURY R.: *Multivariate Interpolation*. ch. 5.5. URL: https://ece.uwaterloo.ca/~dwharder/NumericalAnalysis/05Interpolation/multi/ [cited 2019-12-05].

[HLP17a] HAREL G., LEKIEN J.-B., PÉBAŸ P. P.: Two new contributions to the visualization of AMR grids: I. interactive rendering of extreme-scale 2-dimensional grids ii. novel selection filters in arbitrary dimension. *ArXiv abs/1703.00212* (2017).

[HLP17b] HAREL G., LEKIEN J.-B., PÉBAŸ P. P.: Visualization and analysis of large-scale, tree-based, adaptive mesh refinement simulations with arbitrary rectilinear geometry. *ArXiv abs/1702.04852* (2017).

[KA13] KÄHLER R., ABEL T.: Single-pass GPU-raycasting for structured adaptive mesh refinement data. In *IS&T/SPIE Electronic Imaging* (2013).

[KBH*14] KIRIS C. C., BARAD M. F., HOUSMAN J. A., SOZER E., BREHM C., MOINI-YEKTA S.: The LAVA computational fluid dynamics solver. In *52nd Aerospace Sciences Meeting* (2014).

[KH02] KÄHLER R., HEGE H.-C.: Texture-based volume rendering of adaptive mesh refinement data. *The Visual Computer 18*, 8 (2002).

[Kit03] KITWARE, INC.: *The Visualization Toolkit User's Guide*, January 2003. URL: http://www.kitware.com/publications/item/view/1269.

[KWAH06] KÄHLER R., WISE J., ABEL T., HEGE H.-C.: GPU-assisted raycasting for cosmological adaptive mesh refinement simulations. In *Volume Graphics* (2006).

[KWH09] KNOLL A. M., WALD I., HANSEN C. D.: Coherent multiresolution isosurface ray tracing. *The Visual Computer 25*, 3 (2009), 209–225.

[LC87] LORENSEN W. E., CLINE H. E.: Marching Cubes: A high resolution 3D surface construction algorithm. In *International Conference on Computer Graphics and Interactive Techniques* (1987).

[LK10] LAINE S., KARRAS T.: Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics 17*, 8 (2010), 1048–1059.

[LLY06] LJUNG P., LUNDSTRÖM C., YNNERMAN A.: Multiresolution interblock interpolation in direct volume rendering. In *EUROVIS - Eurographics /IEEE VGTC Symposium on Visualization* (2006), The Eurographics Association. doi:10.2312/VisSym/EuroVis06/259-266.

[LVI*13] LEAF N., VISHWANATH V., INSLEY J., HERELD M., PAPKA M. E., MA K.-L.: Efficient parallel volume rendering of large-scale adaptive mesh refinement data. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)* (2013), IEEE, pp. 35–42.

[Max93] MAX N.: Sorting for polyhedron compositing. In *Focus on Scientific Visualization*. Springer-Verlag, Berlin, 1993.

[MC97] MA K.-L., CROCKETT T. W.: A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. In *Proceedings of the IEEE symposium on Parallel rendering* (1997).

[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.

[ME11] MORAN P., ELLSWORTH D.: Visualization of AMR data with multi-level dual-mesh interpolation. *IEEE transactions on visualization and computer graphics* (2011).

[MKW*04] MARMITT G., KLEER A., WALD I., FRIEDRICH H., SLUSALLEK P.: Fast and accurate ray-voxel intersection techniques for iso-surface ray tracing. In *VMV* (2004), vol. 4, pp. 429–435.

[MOM*00] MACNEICE P., OLSON K. M., MOBARRY C., DE FAINCHTEIN R., PACKER C.: Paramesh: A parallel adaptive mesh refinement community toolkit. *Computer physics communications 126*, 3 (2000), 330–354.

[NSLD99] NORMAN M. L., SHALF J., LEVY S., DAUES G.: Diving deep: data-management and visualization strategies for adaptive mesh refinement simulations. *Computing in Science Engineering 1*, 4 (July 1999), 36–47. doi:10.1109/5992.774839.

[OBB*05] O'SHEA B. W., BRYAN G., BORDNER J., NORMAN M. L., ABEL T., HARKNESS R., KRITSUK A.: Introducing Enzo, an AMR cosmology application. In *Adaptive mesh refinement-theory and applications*. 2005.

[PG17] PATCHETT J., GISLER G.: Deep water impact ensemble data set. *Los Alamos National Laboratory, LA-UR-17-21595, available at http://dssdata. org* (2017).

[PM12] PHARR M., MARK W. R.: ispc: A SPMD compiler for high-performance CPU programming. In *2012 Innovative Parallel Computing (InPar)* (2012), IEEE, pp. 1–13.

[PSL*98] PARKER S., SHIRLEY P., LIVNAT Y., HANSEN C., SLOAN P.-P.: Interactive ray tracing for isosurface rendering. In *Proceedings Visualization'98 (Cat. No. 98CB36276)* (1998), IEEE, pp. 233–238.

[VGW94] VAN GELDER A., WILHELMS J.: Topological considerations in isosurface generation. *ACM Transactions on Graphics (TOG) 13*, 4 (1994), 337–375.

[WBUK17] WALD I., BROWNLEE C., USHER W., KNOLL A.: CPU volume rendering of adaptive mesh refinement data. In *SIGGRAPH Asia 2017 Symposium on Visualization* (New York, NY, USA, 2017), SA '17, ACM, pp. 9:1–9:8. doi:10.1145/3139295.3139305.

[WCM12] WEBER G. H., CHILDS H., MEREDITH J. S.: Efficient parallel extraction of crack-free isosurfaces from adaptive mesh refinement (AMR) data. In *IEEE Symposium on Large Data Analysis and Visualization (LDAV)* (2012), IEEE, pp. 31–38.

[WFM*05] WALD I., FRIEDRICH H., MARMITT G., SLUSALLEK P., SEIDEL H.-P.: Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics 11*, 5 (2005), 562–572.

[WJA*16] WALD I., JOHNSON G. P., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J., GÜNTHER J., NAVRÁTIL P.: OSPRay-A CPU ray tracing framework for scientific visualization. *IEEE transactions on visualization and computer graphics 23*, 1 (2016), 931–940.

[WKL*01] WEBER G. H., KREYLOS O., LIGOCKI T. J., SHALF J., HAGEN H., HAMANN B., JOY K. I., MA K.-L., COMPUTERGRAPHIK A.: High-quality volume rendering of adaptive mesh refinement data. In *VMV* (2001), vol. 1.

[WKL*03] WEBER G. H., KREYLOS O., LIGOCKI T. J., SHALF J. M., HAGEN H., HAMANN B., JOY K. I.: Extraction of crack-free isosurfaces from adaptive mesh refinement data. In *Hierarchical and Geometrical Methods in Scientific Visualization*. 2003.

[WN] WHITE D., NYLANDER P.: Hypercomplex fractals. URL: http://www.bugman123.com/Hypercomplex/index.html [cited 11.20.2019].

[WUP*18] WU Q., USHER W., PETRUZZA S., KUMAR S., WANG F., WALD I., PASCUCCI V., HANSEN C. D.: VisIt-OSPRay: Toward an exascale volume visualization system. In *Eurographics Symposium on Parallel Graphics and Visualization* (2018), Childs H., Cucchietti F., (Eds.), The Eurographics Association. doi:10.2312/pgv.20181091.

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: A kernel framework for efficient CPU ray tracing. *ACM Trans. Graph. 33*, 4 (July 2014), 143:1–143:8. doi:10.1145/2601097.2601199.

[WWJ19] WANG F., WALD I., JOHNSON C. R.: Interactive rendering of large-scale volumes on multi-core CPUs. In *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV)* (2019), IEEE, pp. 27–36.

[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 25*, 1 (2019), 1142–1151.

[ZAB*19] ZHANG W., ALMGREN A., BECKNER V., BELL J., BLASCHKE J., CHAN C., DAY M., FRIESEN B., GOTT K., GRAVES D., KATZ M., MYERS A., NGUYEN T., NONAKA A., ROSSO M., WILLIAMS S., ZINGALE M.: AMReX: A framework for block-structured adaptive mesh refinement. *Journal of Open Source Software 4*, 37 (May 2019), 1370.