CPU Isosurface Ray Tracing of Adaptive Mesh Refinement Data

Feng Wang, Ingo Wald, Qi Wu, Will Usher and Chris R. Johnson

Fig. 1: High-fidelity isosurface visualizations of gigascale block-structured adaptive mesh refinement (BS-AMR) data using our method. Left: a 28 GB GR-Chombo [7] simulation of gravitational waves resulting from the collision of two black holes. Middle and Right: a 57 GB AMR dataset computed with LAVA [17] at NASA, simulating multiple fields over the landing gear of an aircraft. Middle: isosurface representation of the vorticity, rendered with path tracing. Right: a combined visualization of volume rendering and advanced shading effects like transparency, ambient occlusion and path tracing.

Abstract—Adaptive mesh refinement (AMR) is a key technology for large-scale simulations that allows for adaptively changing the mesh resolution, resulting in significant computational and storage savings. However, visualizing such AMR data poses a significant challenge due to the difficulties introduced by the hierarchical representation when reconstructing continuous field values. In this paper, we detail a comprehensive solution for interactive isosurface rendering of block-structured AMR data. We contribute a novel reconstruction strategy—the *octant* method—which is continuous, adaptive and simple to implement. Furthermore, we present a generally applicable hybrid implicit isosurface ray-tracing method, which provides better rendering quality and performance than the built-in sampling-based approach in OSPRay. Finally, we integrate our *octant* method and hybrid isosurface geometry into OSPRay as a module, providing the ability to create high-quality interactive visualizations combining volume and isosurface representations of BS-AMR data. We evaluate the rendering performance, memory consumption and quality of our method on two gigascale block-structured AMR datasets.

Index Terms—AMR, Isosurface, Ray tracing, Reconstruction strategy, OSPRay

1 INTRODUCTION

Adaptive mesh refinement (AMR) techniques are used to solve a range of complex problems in numerical analysis. By providing an adaptive, hierarchical resolution representation of the computational domain, AMR techniques allow the simulation to focus both computational effort and storage on regions of interest, enabling larger, more complex problems to be solved. Although other forms of AMR data exist (e.g., mesh distortion and tree-based), block-structured AMR (BS-AMR) [3, 4] is the most widely used in practice, as it can be easily coupled with octree or recursive-grid AMR. BS-AMR forms the basis for a number of scientific simulation frameworks, including BoxLib [2], LAVA [17], Chombo [9], GR-Chombo [7], Enzo [30], AMReX [1] and Uintah [32]. A detailed overview of these frameworks, and other BS-AMR-based simulations, can be found in Dubey et al.’s survey [10].

Although BS-AMR techniques have found wide adoption in current large-scale HPC simulations, visualization techniques for such data have struggled to keep up. Existing visualization solutions for large-scale AMR data remain either special purpose [13, 26] or have severe limitations [28]. General visualization frameworks such as VTK [34, 35], ParaView [36] and VisIt [6] provide limited support for direct visualization of AMR datasets, requiring the user to either down- or up-sample the data to a fixed resolution grid before rendering. Down-sampling the data clearly comes with an undesirable loss of resolution in regions of interest in the data, whereas up-sampling the data may require an exorbitant amount of memory.

A key challenge in directly rendering AMR data is reconstructing the data at level boundaries. Prior work has proposed to introduce unstructured mesh elements to stitch across level boundaries [11, 45], at the cost of requiring the rendering method to handle unstructured elements. GPU-based approaches for visualizing such data [13, 16] typically remain in special-purpose tools, and are limited by the size of the GPU memory, requiring data-parallel rendering [12, 21, 24] to support the large datasets produced by current simulations. Thus, an efficient approach for direct isosurface visualization of AMR data on CPUs remains desirable, due to both the prevalence of CPUs on current and upcoming HPC systems and the large amount of memory available.

In this paper, we propose an efficient solution for isosurface visualization of large-scale BS-AMR data. We build our approach on a novel reconstruction method for BS-AMR data, called the *octant* method, that allows us to construct crack-free implicit isosurfaces, even across level boundaries. To render these isosurfaces, we combine ideas from isosurface extraction and implicit isosurface ray tracing and present an efficient hybrid implicit isosurface ray-tracing approach, which allows for semi-interactive changes to the isovalue. Finally, we integrate our reconstruction method and hybrid implicit isosurface approach into the OSPRay ray-tracing framework [41] as a module, allowing us to trivially support multiple transparent isosurfaces, combined isosurface...
and volume rendering and advanced shading effects. Our contributions in detail are:

- A novel BS-AMR reconstruction strategy—the 
  octant method—
  applicable to both isosurface and direct volume rendering, that is
  locally rectilinear, adaptive and continuous, even across level
  boundaries.

- An efficient hybrid implicit isosurface ray-tracing approach that
  combines ideas from isosurface extraction and implicit isosurface
  ray-tracing, applicable to both non-AMR data and (using our
  octant method) BS-AMR data.

- The integration of the 
  octant method and hybrid isosurface ray-
  tracing approach within OSPRay and the evaluation of the system’s
  capabilities on two complex BS-AMR datasets.

2 RELATED WORK

AMR was first introduced by Berger and Oliger [4], who used a binary
decomposition (e.g., a quadtree or octree) to create a hierarchical represen-
tation of the simulation domain. Berger and Colella [3] extended this
approach and proposed a more general BS-AMR representation. BS-
AMR represents the simulation domain as a series of overlapping grids
of arbitrary dimension, where higher resolution grids are used only in
regions of interest. As discussed previously, this BS-AMR representation
has found wide adoption in the simulation community [10]. Similar
to nonadaptive grid approaches (Figures 2a and 2b), in BS-AMR meth-
ods the data can be stored either on the grid vertices (Figure 2c) or at
the cell centers (Figure 2d). In practice, most existing AMR simulation
frameworks use a cell-centered grid [42]. Unless otherwise specified,
throughout the text we will focus on cell-centered AMR data.

As AMR data becomes more widely used in scientific simulations,
visualization researchers have worked to address the corresponding
challenges encountered when visualizing such data. A key challenge in
visualizing BS-AMR data is how to reconstruct the data across level
boundaries to produce a continuous function. This reconstructed
function can then be visualized using volume rendering or by explicitly
extracting isosurfaces or rendering them implicitly.

2.1 Reconstruction Across Boundaries

Correctly reconstructing, or “stitching”, the BS-AMR data across ad-
jacent cells at different resolutions is a well-known and challenging
problem. A survey provided by Van Gelder and Wilhelms [37] intro-
duced various solutions to this problem, also sometimes referred to as
the T-junction problem in the literature [42]. Generally, the T-junction
problem produces discontinuities in the reconstructed field, leading to,
for example, holes in isosurfaces computed on the field or incorrect
colormapping when volume rendering. These errors in the reconstruc-
tion lead to incorrect interpretations of the simulation data. A desirable
reconstruction method should be able to interpolate a continuous func-
tion at any given point in the simulation domain, including across level
boundaries.

Weber et al. [42, 45] proposed a solution based on the dual grid
method and hybrid isosurface ray-tracing approach within OSPRay.

Fig. 2: (a) Trilinear interpolation is trivial on a vertex-centered single-
level grid. (b) A cell-centered single-level grid can be converted to a
vertex-centered grid by introducing dual cells. At the level boundaries
of vertex-centered AMR data (c), it is sufficient to introduce a layer
of ghost cells. (d) A cell-centered AMR grid can still be transformed using
dual cells; however, stitching across the boundary remains challenging.
Previous work has addressed the T-junction problem by introducing
unstructured elements at the boundary, shown in green.

2.2 Volume Rendering

Ma and Crockett [25] introduced the first high-quality AMR volume
rendering system, based on cell projection. Ma [24] further extended
this approach to support MPI parallel rendering, thereby achieving
better rendering performance. Norman et al. [29] proposed to leverage
the support of standard visualization tools for volume rendering finite-
element data to visualize AMR data by converting the AMR data
into finite-element hexahedral cells. However, this conversion incurs
both memory and computational costs. Park et al. [31] presented
a hierarchical multisresolution splatting technique to visualize AMR
data interactively on a single workstation. Wald et al. [38] recently
introduced an interactive method for GPU-based rendering of AMR
data within OSPRay.

On the GPU, Weber et al. [43, 44] presented an approach based on
cell projection for direct volume rendering of AMR data. Kahler and
Hege [15] introduced a 3D texture-based volume rendering algorithm
for AMR data that employs a space-partitioning scheme to decompose
the volume into axis-aligned regions of equal-sized cells. This approach,
although it achieves fast rendering performance, ignores the T-junction
problem at level boundaries. Kahler and Hege’s approach was further
extended to employ ray tracing in multiple rendering passes [16] and
finally in a single pass [14]. Gossink et al. [13] presented a visualization
system for time-varying AMR data on the GPU and designed an out-of-
core method to re-sample the data into nonadaptive grids. Marchesin
and de Verdiere [26] employed a special-case solution for high-quality
and semi-analytical volume rendering of hexahedral cell data. Recently,
Leaf et al. [21] used a reconstruction method similar to that of Ljung et
al. [22] and provided a cluster- and GPU-parallel rendering scheme to
visualize large-scale AMR data in a distributed parallel setting.

that it supports an arbitrary number of grids. Moreover, our method
can leverage the optimized data query approach of Wald et al. [38].

Our octant reconstruction method employs a similar approach to
that of Ljung et al. [22] and Beyer et al. [5], but it is less restrictive in
2.3 Isosurface Rendering

Whether extracting the isosurface to a mesh [23], or rendering it with an implicit method [33], the requirements placed on the reconstruction method used to sample the AMR data are much stricter than in volume rendering. Volume rendering tends to blur and smooth out features, hiding some artifacts; however, any small cracks or discontinuities will be readily apparent in a surface representation.

Marching cubes (MC) [23] has been applied to adaptive volumes with a variety of methods proposed for fixing cracks encountered at level boundaries. Shu et al. [35] extended the MC algorithm into the adaptive MC algorithm and patched cracks with polygons of the same shape. Westermann et al. [46] introduced an adaptive approach for isosurfacing regular volume data at arbitrary levels of detail and employed triangle fans to fill in the cracks at boundaries. Fang et al. [11] subdivided the lower resolution cell faces into pyramid elements to match the higher resolution faces. These approaches, although they yielded a crack-free isosurface, were applicable only for a vertex-centered multiresolution grid. For cell-centered AMR data, Weber et al. [42, 45] first transformed the cell-centered AMR grid into a vertex-centered grid by introducing dual cells and then stitched the boundary with an unstructured mesh (see Figure 2d). However, explicitly tessellating the isosurface can produce a large number of triangles, impacting rendering performance and the time it takes to change the isovalue.

An alternative approach that addresses these limitations is to directly ray trace an implicit representation of the isosurface [33]. A large body of work has investigated ray tracing implicit isosurfaces on regular grid volumes [19, 20, 33, 40]; however, relatively little work has explored implicit isosurface rendering of BS-AMR data. Co et al. [8] mention the applicability of their iso-splatting approach to AMR data, although this has not been explored. Wald et al.’s AMR reconstruction kernels [38] can be used for isosurface ray-tracing in OSPRay with the built-in sample-based isosurface method; however, this approach yields poor rendering quality and performance, as will be shown later.

3 Reconstructions of BS-AMR Data

In this section, we introduce a novel BS-AMR reconstruction strategy, called the octant method, which will take a given sample point \( p = (x, y, z) \) and map it to a scalar value \( F(p) \). BS-AMR data is specified as a set of data bricks, each a grid (typically of \( 16 \times 16 \times 16 \) cells) with a cell-centered data value associated with a refinement level \( L \). Bricks on the same level do not overlap, and on the coarsest level generally form a structured grid that fills the entire domain. However, finer level bricks overlap coarser ones, and the boundary of the finer level brick aligns with the coarser level cell boundary, such that each coarser cell is covered by exactly \( R \times R \times R \) finer cells, where \( R \) denotes the refinement factor.

3.1 Methodology Overview

A range of interpolants are available from numerical analysis for use in reconstructing a continuous field \( F(x) \) from a discrete set of data points. For example, the nearest neighbor, linear and higher order basis function interpolation methods have been widely used in visualization and computer graphics. However, these interpolation methods are highly dependent on the underlying topology of the data being reconstructed, making their application to BS-AMR grids more challenging than to nonadaptive grids (Figure 2), due to the grid topology change at the level boundaries. Computing a “correct” interpolant on BS-AMR data is made more challenging due to the variety of formats and layouts employed. While there is currently no gold standard interpolation method for AMR data, several key properties (ranked by importance) should be considered when designing an interpolant:

1. **Continuity**, in particular across-level boundaries, is a key concern, as discontinuities in \( F(x) \) can change the computed isosurface topology, resulting in undesirable artifacts.

2. **Adaptivity** denotes that it is desirable for the interpolant to have a higher frequency in finer regions and a lower frequency in coarser ones.

3. **Accuracy** requires that a reconstruction method should be interpolating, and, furthermore, that given an arbitrary sample point \( x \), the reconstructed value \( F(x) \) should be as close to the ground truth as possible.

4. **Locally Rectilinear** means that the approach will decompose the domain into a set of nonoverlapping rectilinear “cells” within which the interpolant is locally trilinear, allowing for fast implicit ray-isosurface intersections.

5. **Simplicity** indicates that the reconstruction kernel should be easy to implement. A simpler kernel is more likely to perform well.

3.2 Octant Method

A key challenge of reconstructing cell-centered AMR data is that the dual cells at different resolution levels do not line up and can even reach across the level boundaries (Figure 3a). Taking \( \bar{p} \) in Figure 3a as an example, using dual cell \( D_1, D_2 \) or \( D_3 \) to interpolate the value of \( \bar{p} \) will yield different results. We can address these issues by casting the problem in terms of interpolating within the octants of cells (Figure 3b). First, an octant does not extend beyond the bounds of its parent cell and thus will not cross level boundaries. Second, the entire set of octants completely tiles the domain, without gaps or overlap. Finally, octants are rectilinear, freeing us from requiring unstructured elements to stitch boundaries. Performing trilinear interpolation within each octant yields an adaptive and locally rectilinear interpolation scheme. Furthermore, we can achieve a continuous interpolant by taking some care in choosing the values at the octant vertices at level boundaries.

To better explain the octant method, let us consider a logical cell \( C \). The cell is evenly split into eight octants \( \{O_i, i \in 1, 2, ..., 8\} \), which lie along one of eight unit vectors \( \pm \hat{X}, \pm \hat{Y}, \pm \hat{Z} \) from \( C \)’s center (Figure 4b). Of the eight vertices of each octant, \( O^{(0)} \) coincides with the cell center, whereas the others lie on the cell’s boundary (faces, edges and corners). The boundary vertices are named based on the direction in which they can be reached from the cell center. For example, the vertices on \( C \)’s faces are labeled \( O^{(XY)}, O^{(YZ)}, O^{(ZX)} \); those on \( C \)’s edges are labeled \( O^{(XY)}, O^{(YZ)}, O^{(ZX)} \); and those on \( C \)’s corners \( O^{(XYZ)} \).
we take the vertices of the finer side and set their value to whatever when we use trilinear interpolation within the eight octants. Due to To compute \( O \), when we are in a nonboundary region, an octant of a cell is also an octant of a dual cell. Therefore, we will get the same interpolant as with dual cells when we use trilinear interpolation within the eight octants. Due to the symmetry between cells and dual cells, the vertex is the corner vertex of the dual cell, the cell edge vertices are the dual cell’s face vertices, etc. In nonboundary regions, an octant of a cell is also an octant of a dual cell. Thus, we can also compute the dual cell \( D \) (Figure 4a). We name the eight vertices of \( D \) following the same scheme as for the octant vertices: \( D^{(0)} \) coincides with the cell center; \( D^{(X)} \) lies along \( \vec{X} \) from \( D^{(0)}, D^{(XY)} \) lies along \( (\vec{X}, \vec{Y}) \) and \( D^{(XYZ)} \) along \( (\vec{X}, \vec{Y}, \vec{Z}) \). It is easy to see the cells and dual cells form a symmetric relationship: the cell center vertex is the corner vertex of the dual cell, the cell edge vertices are the dual cell’s face vertices, etc.

In nonboundary regions, an octant of a cell is also an octant of a dual cell. Therefore, we will get the same interpolant as with dual cells when we use trilinear interpolation within the eight octants. Due to the symmetry between cells and dual cells, the center of the face spanned by \( C \) and its neighbor cell along \( \vec{X} \), \( C^{(X)} \), and thus its value should be the average of those two cells’ values. From the symmetry of the cells and dual cells, the center of \( C^{(X)} \) is \( D^{(X)} \), and thus \( O^{(X)} = \frac{1}{2}(O^{(0)} + O^{(X)}) \). 3) The edge vertex \( O^{(XY)} \) lies exactly halfway between \( C^{(0)} \) and its neighbor cell along \( \vec{X}, C^{(X)} \), and thus is the average of the values for those four cells. 4) The corner vertex \( O^{(XYZ)} \) is set to the average of the eight vertices of \( D \).

We can achieve a continuous interpolation across the boundaries if we take the vertices of the finer side and set their value to whatever the octant’s interpolant produces on the coarse side. Even in a three-dimensional scenario, where the octant’s vertex may touch cells on multiple different levels, the finer level octant’s vertices always fall within the coarser level octant’s faces. Therefore, we will achieve continuity across the boundary as long as the coarser side octant defines the interpolant; however, this strategy will sacrifice some accuracy at the boundary.

**Octant Algorithm.** The above strategy leads to an algorithm that combines the stitching with the trilinear interpolant in coarse regions: For any point \( \vec{p} \), we first find the leaf cell \( C \) and octant \( O \) it is contained in, and its corresponding dual cell \( D \) on this level. In this octant, the value of vertex \( O^{(0)} \) is set to \( C \). For those vertices on the edge of the cell, there could be 2, 4 or 8 of \( D \)’s corners that are required to compute their value, if we are in a nonboundary region. Taking \( O^{(XYZ)} \) as an example, we would need to consider \( D^{(0)}, D^{(X)}, D^{(Y)} \) and \( D^{(Z)} \). If all these inputs exist and are at the same level as \( C \), then these vertices do not lie on a boundary, and thus can be computed as in the nonboundary case. Otherwise, if at least one of those inputs lies on a coarser level, we know that this vertex lies on at least one boundary, with a coarser cell on the other side. In this case, we will find the coarsest level neighbor and construct a continuous stitching by setting the vertex’s value with the interpolant from the coarser side. The searching of the coarsest level neighbor could be easily realized by recursively calling our sample function for the vertex position minutely moved along the direction away from the octant’s cell center. If both cases do not hit, we know that there exists at least one of those inputs that is involved for an octant’s vertex but is an inner node, and yet no other input is on a coarser level. Thus, we can infer that the vertex lies on a boundary but is on the coarser side and therefore can determine the interpolant. This description leads to the following algorithm:

```c
float octant(P)
Octant oct = findLeafOctant(P)
Dual D = findDualCell(oct)
/* center vertex */
oct[0].v = C.v;
/* edge vertex */
int lxMin = min(D[0].l,D[X].l)
int lxmax = max(D[0].l,D[X].l)
if (lxMin == lxmax) /* not a boundary */
oct[X].v = avg(D[0].v,D[X].v);
else if (lxMin < oct.l) /* we’re fine side */
"/ finer side: fill "FROM" coarse side/
oct[X].v = intersect(oct,X,p + eps * oct.dX)
else /* we’re coarse side */
(Compute Coarser Side Vertex)
/* face vertex */
int lXYmin = min(D[0].l,D[X].l,D[Y].l)
... /* symmetric to above*/
/* corner vertex */
int lXYZmin = min(D[0].l,D[X].l,D[X].l)
... /* symmetric to above*/
```

Figure 5 illustrates the above procedure. To determine the value of \( \vec{p} \) using the octant method, octant \( O \), and dual cell \( D \), shown as red and blue square, are initialized (Figure 5a). Unlike the simple calculation of \( O^{(0)} \)’s and \( O^{(X)} \)’s value by applying the previously mentioned rules, the calculation of \( O^{(X)} \)’s and \( O^{(XY)} \)’s value requires an additional stitching process since we detect that \( D^{(X)} \)’s level < \( O \)’s level. Then \( \vec{p} \) is computed...
by moving $O^{(X)}$ a bit to the coarser side and used for calculating octant $O^p$’s logical coordinates. Subsequently, the vertex value of $O^p$ is recursively initialized with the octant method. So far, the value of $O^{(X)}$ can be achieved by trilinearly interpolating the original point $O^X$ with octant $O^{p}$’s value, which is the same as the calculation of $O^{(X)}$’s value.

Computing the Coarser Side Interpolant. How exactly we compute the value for the coarser side octant $O^p$ is completely our choice. Fortunately, whatever we set to those vertices, the above rules will guarantee that our interpolant is continuous, adaptive, and locally rectilinear. In this paper, we will introduce four options: 

- coarsest level lerp, current-level lerp, basis function and finest level lerp.

Coarsest Level Lerp. The most obvious way of setting the coarser-side interpolant is to simply perform trilinear interpolation on the coarsest level involved for any of the inputs. In the logical grid abstraction of AMR data, we can still view each refinement level as a structure grid [38]. Hence, we could pick cells in any logical cell and provide an interface—lerpOnLevel—to trilinearly interpolate the value based on the cell. In this case, we can even forgo the epsilon-offsetting and directly call this trilinear interpolant for boundary vertices:

```cpp
// Compute Coarser Side Vertex

int lX' = min(D[0'], 1, D'[X']).l
if (lX' == oct.l)
    oct'[X'].v = lerpOnLevel(lX', oct'[X'].p)
else
    oct'[X'].v = avg(D[0'].v, D'[X'].v);
```

In most cases, the possibly multiple lerpOnLevel calls would all find the same dual cell $D$. This case could obviously be replaced by simply averaging the respective inputs in a performance-oriented implementation.

Current-Level Lerp. Given that it is easy to get the current level of a cell at a point using findLeafCell, we could perform the interpolation on the leaf cell, rather than the coarsest level cell. This strategy allows for the interpolant to be adaptive.

```cpp
// Compute Coarser Side Vertex

int lX' = min(D[0'], 1, D'[X']).l
if (lX' == oct.l)
    oct'[X'].v = avg(D[0'].v, D'[X'].v);
else
    lX'min = findLeafCell(oct'[X'].p).l
    int level = findLeafCell(oct'[X'].p).l
    oct'[X'].v = lerpOnLevel(level, oct'[X'].p)
```

Basis Functions. Setting the boundary to the above option is similar to the blending method described in [38], which involves some inner cell values at the boundary and therefore yields some ghosting. However, since we have full freedom on how exactly to set the coarser side boundary, we can also set the coarser side’s octant vertices using any other method. For example, we can compute these vertices using the basis function method described in [38], which employs a hat-shaped basis function to define the interpolant. This strategy will remove the ghost artifacts, since the calculation involves actual leaf cells only on the boundary; however, as described by Wald et al. [38], it is unclear how to perform ray-isosurface intersections with this interpolant.

Finest Level Lerp. Perhaps the best alternative for computing the coarser side interpolant is to use the finestLevelLerp. The vertex in question lies exactly on at least one boundary and always right in the center of any finest level logical dual cell. Therefore, the finest level lerp computes the weighted average of all leaf cells that touch at this point. For example, the value of $O^{(X)}$ in Figure 5b is filled with the weighted average of $V_2$, $V_3$, and $V_4$. This method, therefore, is not only fast and trivially simple to code but also qualitatively one of the best methods we have found so far, and it is used by default for calculating the coarser side octant’s value in our results. It is implemented as follows:

```cpp
// Compute Coarser Side Vertex

int lX'min = min(D[0'], 1, D'[X']).l
```

3.3 Potential Numerical Issue

Although the octant method provides a continuous interpolant across the level boundary in theory, it is still worth mentioning the potential numerical issue when using limited-precision floating-point arithmetic. The vertex value of the adjacent octant across the boundary might not exactly agree in practice due to intermediate round-off error when operations are performed on the same-source values in different orders, such as calculating the vertex values in a pre-computing step and then interpolating, as opposed to directly interpolating on the other side of an abutting face. Although the numerical issue is theoretically possible, we did not see it in practice in our experiments.

4 Ray Tracing Implicit Isosurfaces

Our octant reconstruction method is applicable to any use case that requires sampling of BS-AMR data. For example, our method could be used for explicit isosurface extraction by simply iterating over the octants, computing each octant’s vertex values with our octant method and applying marching cubes [23], treating each octant as a “voxel”. Although this approach would certainly work, it would generate a potentially very large number of triangles.

In a ray tracer, explicit tessellation can be avoided by employing an implicit isosurface ray tracing method [19, 33, 40]. The simplest approach is to march the ray through the volume with a fixed-step size, and at each step check if an intersection with the isosurface exists. OSPRay currently employs this ray marching approach to render implicit isosurfaces. However, this method is inherently nonadaptive, creating many unnecessary samples in coarse regions, and an insufficient number of samples in fine regions (Figure 6a), resulting in unnecessary high costs and poor rendering quality. Instead, one can build an implicit KD-tree [40] or implicit BVH [18, 39] over the voxels and use this acceleration structure to quickly locate voxels that contain the isosurfaces.
being rendered. The voxels containing the isosurface are referred to as “active voxels”. A similar approach could be implemented with our \textit{octant} method by treating each octant as a “voxel”.

Although we initially considered this approach, several issues arise when attempting to implement it within OSPRay. First, OSPRay heavily relies on Embree for BVH construction and ray traversal; however, Embree has no notion of implicit BVHs, requiring us instead to implement our own BVH construction and traversal kernels. Second, a naïve implementation of implicit BVHs usually has high memory requirements, because typically a BVH has at least one node per input voxel, which can significantly multiply the storage requirements. When this multiplication is coupled with the fact that each AMR cell would produce eight octants, the total memory cost of this approach becomes prohibitive.

To address these issues, we adopted two different and orthogonal strategies. First, we developed a “hybrid” implicit isosurface module for OSPRay that is able to use Embree for BVH construction and traversal and is applicable to general rectilinear volume data. Second, we derive a series of optimizations (e.g., active octant filtering and octant merging) specific to our \textit{octant} method to reduce the number of primitives we have to build the BVH over, further reducing memory overhead.

4.1 \textit{“Hybrid” Implicit Isosurface Ray Tracing}

The core idea of our hybrid implicit isosurface method is to combine ideas from both explicit isosurface extraction and implicit isosurface ray tracing. As in explicit isosurface extraction, we first extract a list of all the active voxels and consider only those active voxels; yet like implicit isosurface ray tracing, we then build a BVH over these active voxels (using Embree), traverse rays through this BVH and perform an implicit ray-iso-surface intersections within each voxel, without ever extracting any polygons (Figure 6b).

4.1.1 Voxels, Encoding and Active Voxel Sources

At the core of our method is an abstraction for viewing any structured volume (e.g., regular grids, rectilinear grids, BS-AMR), as a collection of logical voxels, where each voxel is a cube with trilinearly interpolated scalar values at each of its vertices. In this case, each voxel can thus be described by 12 values: three for its 3D coordinates, one for its width and eight for its vertex values. Note that for general rectilinear volumes we require two additional values to specify the height and depth of the voxel. These voxels can be, for example, dual cells in a structured volume or octants in a BS-AMR volume. Active voxels are those whose value range contains at least one of the isovalue(s), which we are interested in rendering.

With this abstraction, we can view any volume as simply a source of active voxels, by assuming that there is some kind of entity—a \textit{VoxelSource}—which can quickly generate a list of active voxels in the volume. This initial process works similar to the active voxel extraction of explicit isosurface extraction methods. We will describe later in Section 4.2.1 how we generate the voxels for our BS-AMR data.

Having to consider only the active voxels reduces memory usage considerably, as typically only a few of the total voxels are active. Nevertheless, explicitly storing a full 12 floats for even just these voxels would be prohibitively expensive. Therefore, our software abstraction further assumes that each active voxel can be encoded into a single 64-bit value (e.g., as 21:21:21 bit coordinates in a structured volume). The \textit{VoxelSource} then offers an interface to retrieve the complete voxel information from this 64-bit reference.

4.1.2 BVH Construction and Traversal

Since we now have to consider only the active voxels, we no longer need any special BVH construction or traversal kernel and can simply use Embree. To do so, we first use the \textit{VoxelSource} to produce a list of all active voxels, storing the 64-bit reference for each active voxel. We then create an Embree “user geometry” with as many primitives as active voxels, and within the geometry’s \textit{getBounds} callback query the \textit{VoxelSource} for the respective voxel’s bounding box to allow Embree to build a BVH over the voxels.

![Fig. 7: When generating the octants, we can merge “inner octants” (i.e., those not touching a boundary) into dual cells (shaded), significantly reducing memory consumption. We find that on the LandingGear, this optimization reduces the total number of octants by 70.6%.](image)

4.1.3 Ray Voxel Intersection

To perform the actual ray-voxel intersection, we implemented an ISPC version of the ray-iso voxel intersection technique proposed by Marmitt et al. [27] and used this as our Embree user geometry’s intersection routine. As with the bounding box callback, we first have to query the full voxel data for the 64-bit reference from the \textit{VoxelSource}.

Based on how ISPC and Embree’s intersection callbacks work, this ISPC implementation will always intersect the same voxel with either 4-, 8- or 16-wide ray “packets” in packet mode. Given the (very) small nature of each of our voxels, we are fully aware that the number of rays active during intersection will hardly ever be much larger than one, which is clearly wasteful. However, any alternative of intersecting eight different voxels would require significant changes to Embree, which is beyond the scope of this paper.

4.2 Application to Our Octant Method

As mentioned previously, to apply our hybrid implicit isosurface method to AMR data reconstructed using our \textit{octant} method, we can simply implement a \textit{VoxelSource} that encodes each octant as a “voxel”.

4.2.1 Octant Decomposition and Initialization

Although the core idea of our approach is straightforward, some care must be taken to efficiently extract the active octants from large AMR datasets. To allow efficient access to the AMR cells, we employ the AMR-KDTree introduced by Wald et al. [38]. This AMR-KDTree can be built over whatever external memory is used to store the brick’s cells, introducing little memory or compute overhead. The structure of the AMR-KDTree is as follows:

- A leaf in the tree represents a region where all cells come from the same brick. Note that the brick will likely stick out of the leaf’s bounding box, and the same brick may be listed in multiple leaves.
- A leaf node stores a pointer to the finest level brick along with pointers to the coarser bricks that overlap the region.
- A leaf node stores the value range of its finest level cells, which can be used for filtering leaves that do not contain the isovalue.

On top of this AMR-KDTree, the active octant extraction is particularly easy to implement. A naïve first approach could traverse all leaves of the tree, ignoring those that do not contain the isovalue, and decompose each cell of the finest brick in the leaf into eight octants using our \textit{octant} to compute the values of the octant’s vertices. Although this naïve approach will extract a correct crack-free isosurface, it will lead to a large amount of redundant computation. Specifically, the vertex values of “inner” octants will be re-computed eight times, as they are shared with eight other octants

6
Current Basis Finest Octant

INNER octant’s shared vertices (e.g., point A in Figure 7) can also be octant AMR volume to implement our AMR-KDTree structure presented by Wald et al. [38], allowing us to although our approach can be realized in any ray tracer, we evaluate our approach according to two criteria: rendering quality (Section 5.2) and performance (Section 5.3).

4.2.2 Optimized Octant Generation

In nonboundary regions, an “inner” octant is also an octant of the corresponding dual cell. Thus, we can reduce the number of octants we need to process by merging these “inner” octants into dual cells, without affecting the isosurface. We illustrate this optimization in Figure 7: the inner octants (shaded blue) can be merged into dual cells; however, octants touching a level boundary cannot be merged.

With this optimization, we reduce the number of octants processed on the LandingGear (Figure 1, right) by 70.6%, from roughly 2 billion to 616 million. Furthermore, the redundant computation of the “inner” octant’s shared vertices (e.g., point A in Figure 7) can also be avoided. The merged dual cell’s vertices coincide with the cell centers and can simply be set to the cell values. This optimization yields a 64.76% improvement in performance on the LandingGear data. Additional performance improvement can be achieved by computing the list of active octants in parallel; in our implementation we use TBB’s parallel_for. To encode our octants in the 64-bit reference used by the VoxelSource, we store them as 32:32 bits, with the first 32 bits encoding the AMR-KDTree leaf index and the second 32 encoding the octant ID within the leaf.

4.2.3 OSPRay Integration

Although our approach can be realized in any ray tracer, we evaluate our method implemented within the OSPRay ray tracing framework [41]. OSPRay already includes the previously discussed AMR volume and AMR-KDTree structure presented by Wald et al. [38], allowing us to easily re-use them. To integrate our approach, we extend OSPRay with a module implementing our hybrid implicit isosurface geometry, which can take any rectilinear volume as a VoxelSource and extend OSPRay’s AMR volume to implement our octant method.

5 Results

In this section, we first compare the quality of our reconstruction method with prior work [38] using a 2D visualization tool (Section 5.1). Next, we evaluate our approach according to two criteria: rendering quality (Section 5.2) and performance (Section 5.3).

Evaluation Hardware. We conduct our evaluation on three different systems. FSM is quad-socket workstation with four Xeon E7-8890 v3 CPUs, for a total of 72 physical cores at 2.5 GHz, along with 1.4 TB RAM. Lago is a Skylake Xeon workstation equipped with one Intel Xeon Skylake Processor (Gold 6136), for a total of 24 physical cores at 3.0 GHz, along with 256 GB RAM. Stampede2 is the largest supercomputer at the Texas Advanced Computing Center (TACC) and is composed of 4,200 Xeon Phi 7250 Knights Landing (KNL) nodes and 1,736 Skylake Xeon Platinum 8160 nodes (SKX). Each KNL node has 96 GB RAM and 68 physical cores, and each SKX node has 192 GB RAM and 48 physical cores over two sockets. The nodes are connected with an Intel Omni-Path network configured in a fat tree topology with six core switches.

Data Description. We use two BS-AMR datasets in our evaluation. The Black Hole Merger (BHM) is a GR-Chombo [7] simulation of the gravitational waves resulting from the collision of two black holes. The BHM is 28 GB, consisting of 4,114 data blocks and four refinement levels. The finer refinement levels are concentrated at the center of the domain where the black holes merge. The LandingGear (LG) is a dataset produced by NASA using LAVA [17] to simulate the air flow around a aircraft’s landing gear assembly. The LandingGear is 57 GB, consisting of 72,865 blocks and nine refinement levels.

5.1 2D Comparison of Reconstruction Methods

To demonstrate and compare the multiple reconstruction techniques discussed, we developed a 2D AMR reconstruction kernel visualization tool, which implements the five kernels proposed by Wald et al. [38] (the coarsest, current, finest, blend and basis methods), along with our octant method. We show a comparison on a simple case in Figure 8; here we compare on a two-level BS-AMR grid where cell values are 1 (blue, solid circle) or 0 (light green, open circle). To demonstrate the isosurface that would be reconstructed with these methods, we draw isocurves at values of 0.25, 0.5 and 0.75, which are shown in blue, green and white.

We observe that the coarsest method is not adaptive and loses data in refined regions, since it interpolates the value at the coarsest level. In contrast, the current method preserves the raw data but produces a discontinuity at the level boundary, leading to cracks in the surface. The finest method provides high-quality results, but it is not linearly interpolating in some regions (along AB) and is costly to compute. The blend method combines multiple levels but leads to “ghost” artifacts, as it involves interpolating the values of some inner cells. The basis method and our octant method provide similar quality and are both continuous and adaptive. However, the basis method is not locally rectilinear, and thus it is unclear how to formulate ray-iso surface intersections when using it.

5.2 Rendering Quality

Two factors affect the quality of the isosurfaces rendered by our approach: the choice of the reconstruction kernel and the choice of the implicit isosurface ray tracing strategy. We compare the previous sampling kernels of Wald et al. [38] that are applicable to isosurface rendering with our octant method and evaluate the quality of our hybrid implicit isosurface module against OSPRay’s current sample-based isosurface module.

5.2.1 Octant vs. Other Reconstruction Methods

To generate a crack-free isosurface, the reconstruction of the field produced by the sampling method must be continuous. In particular, the “stitching” strategy employed at the level boundaries must provide a continuous interpolation between the levels; otherwise, visible cracks will be produced in the surface at these boundaries. We compare our octant reconstruction method against current and nearest methods proposed by Wald et al. [38]. Compared to these prior reconstruction methods with two gigscale BS-AMR data, we find that only our octant method can reconstruct a correct, crack-free isosurface (see Figure 9).

Although Wald et al. [38] propose an additional three methods—the finest, blend and basis methods—these are either not applicable to isosurface rendering or not feasible to use for generating an isosurface. Although reported to provide good image quality [38], the lack of adaptivity in the finest method would require up-sampling the dataset.
We show our application has the capability of simultaneously direct visualizing multiple transparent isosurfaces simultaneously. While both approaches yield a crack-free isosurface at the boundary, our hybrid implicit isosurface ray tracing method yields a high-quality, crack-free isosurface, at better framerates.

To evaluate the quality of the isosurfaces produced by our proposed approach, we compare the rendering quality of the hybrid implicit isosurface method to the OSPRay’s built-in sample-based method on the three previously mentioned hardware platforms. The benchmarks were done by rendering to a 1024 × 768 framebuffer with OSPRay’s adaptive sampling enabled. We render a single warm-up frame and then take the average framerate over 100 frames. We report rendering performance on both the Black Hole Merger and LandingGear datasets, and we compare the current method [38] with our octant method and our hybrid implicit isosurface method against OSPRay’s built-in sample-based method in Table 1. Our comparisons are also done with two different renderers in OSPRay, the SciVis and pathtracer (pt) renderers. The SciVis renderer is a standard scientific visualization style renderer, supporting shadows and ambient occlusion, whereas the pathtracer is a photorealistic global illumination renderer.

We find that our octant method provides similar rendering performance to that of the current method, but produces a crack-free isosurface. When comparing the performance of our hybrid implicit isosurface module to the OSPRay’s sample-based method, we find a significant performance improvement of one to two orders of magnitude. In addition to the single node runs on Lago and FSM, we leverage OSPRay’s support for data-replicated rendering using MPI to run on 32 Stampede 2 Skylake Xeon nodes, and achieve interactive rendering with our proposed approach even in the most expensive rendering configurations (i.e., with path tracing).

Our approach is also capable of quickly recomputing the active octants, allowing for semi-interactive changes to the isosurface. On the Black Hole Merger dataset, our method takes 1.58s to generate and initialize the active octants, whereas on the LandingGear it requires 6.83s. The BVH is then built over these active octants using Embree, which can process approximately 110 million primitives per second. The BVH build time is less than a second in our experiments. Our approach allows for more interactive exploration of large data with fast isosurface updates, compared to explicit isosurface extraction approaches. Furthermore, by computing the active octants on the fly, and storing a minimal 64-bit reference for each such octant, we require only 10 GB of storage for the LandingGear isosurface.

Overall, we found that mid-gigascale BS-AMR data, such as the 57 GB LandingGear, can be rendered interactively on a single node with our approach. Larger AMR data could be handled with large-memory single node resources, or with parallel rendering on HPC platforms.

6 CONCLUSION AND FUTURE WORK

In this paper, we have presented an efficient solution for ray tracing implicit isosurfaces of BS-AMR data. Our method is based on a novel reconstruction method—the octant method—which allows us to reconstruct crack-free isosurfaces, even across refinement levels, without introducing unstructured elements at the boundaries. Combined with our hybrid implicit isosurface ray tracing method, we enable interactive, high-quality visualization of gigascale BS-AMR datasets, with relatively low memory overhead. Furthermore, our optimized octant extraction method enables semi-interactive isovalue changes. Finally, the hybrid implicit isosurface method presented is applicable to any rectilinear volume data, providing better quality and higher performance isosurface rendering than OSPRay’s built-in ray marching approach.

By integrating our approach into OSPRay as a geometry type, we can easily create combined visualizations, displaying the original volume and simulation mesh data to provide context. We can also leverage OSPRay’s support for transparent MPI-parallel data-replicated rendering to distribute work over multiple nodes. Our OSPRay module can also be leveraged by existing work integrating OSPRay into ParaView and VTK, to provide similar results to production visualization users.

Although our technique can produce high-quality isosurfaces of BS-AMR data, some issues remain to be addressed. First, we would like to investigate further optimizations of the active octant extraction, to provide faster isovalue updates. As isosurface exploration is a key mode of visualizing scientific data, the ability to quickly explore the
field is important. Additional work can be done to further reduce the memory consumption of our method. In addition to allowing for larger data to be explored on a single machine, this could also make our approach applicable to in situ use cases. Additional improvements can also be explored to improve our reconstruction method. While capable of computing crack-free isosurfaces, the computed surface normals can be discontinuous, producing some subtle shading artifacts. Finally, it would also be interesting to extend our work to apply for time-varying distributed AMR data, to allow for interactive visualization of large time-series datasets.

ACKNOWLEDGMENTS

This work was supported in part by the NIH (Grant P4I GM103545-18). Additional support comes from the Intel Parallel Computing Centers Program. The authors wish to thank Patrick Moran from NASA Ames for providing the LandingGear dataset, Juha Jaykka and Paul Shellard from the Stephen Hawking Center for Theoretical Cosmology for use of their COSMOS data. We would also like to thank Paul Navrátil from TACC for the use of Stampede2 and the reviewers for the feedback.

REFERENCES


Fig. 9: A comparison of the isosurfaces produced by two reconstruction kernels from Wald et al. [38] (a,b,d,e) and our method (c,f) on the Black Hole Merger (BHM) and LandingGear (LG) datasets. (a,d) Nearest is similar to nearest-neighbor filtering, resulting in discontinuities even within the same level. (b,e) Current provides better interpolation within a level but still has discontinuities at level boundaries. (c,f) Our Octant method provides a continuous stitching across level boundaries, producing a crack-free isosurface even between levels.