CPU Volume Rendering of Adaptive Mesh Refinement Data

Ingo Wald
Intel

Carson Brownlee
Intel

Will Usher
SCI Institute, University of Utah

Aaron Knoll
SCI Institute, University of Utah

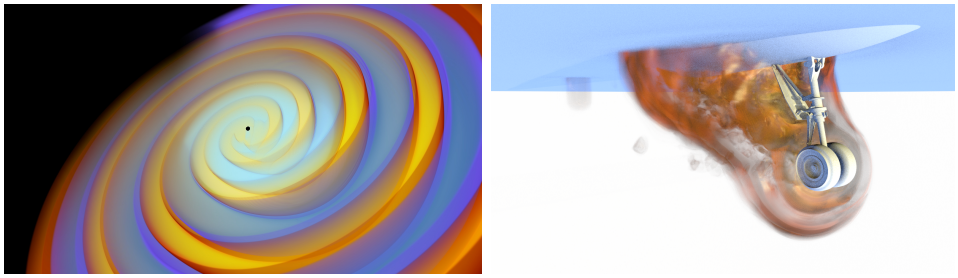


Figure 1: Two examples of our method (integrated within the OSPRay ray tracer): Left: 1.8 GB `Cosmos` AMR data, rendered in ParaView. Right: a 57 GB NASA Chombo simulation, rendered with ambient occlusion and shadows alongside mesh geometry.

ABSTRACT

Adaptive Mesh Refinement (AMR) methods are widespread in scientific computing, and visualizing the resulting data with efficient and accurate rendering methods can be vital for enabling interactive data exploration. In this work, we detail a comprehensive solution for directly volume rendering block-structured (Berger-Colella) AMR data in the OSPRay interactive CPU ray tracing framework. In particular, we contribute a general method for representing and traversing AMR data using a kd-tree structure, and four different reconstruction options, one of which in particular (the basis function approach) is novel compared to existing methods. We demonstrate our system on two types of block-structured AMR data and compressed scalar field data, and show how it can be easily used in existing production-ready applications through a prototypical integration in the widely used visualization program ParaView.

CCS CONCEPTS

• **Computing methodologies** → **Ray tracing**; • **Human-centered computing** → *Scientific visualization*;

KEYWORDS

Ray Tracing, Volume Ray Tracing, Adaptive Mesh Refinement (AMR), Berger Colella AMR Scheme

ACM Reference format:
Ingo Wald, Carson Brownlee, Will Usher, and Aaron Knoll. 2017. CPU Volume Rendering of Adaptive Mesh Refinement Data. In *Proceedings of SA '17 Symposium on Visualization, Bangkok, Thailand, November 27-30, 2017,* 8 pages.

SA '17 Symposium on Visualization, November 27-30, 2017, Bangkok, Thailand

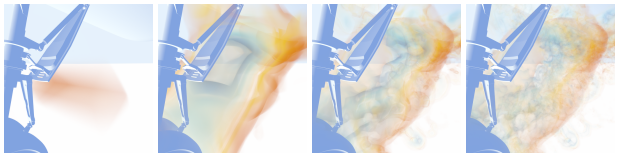
ACM ISBN 978-1-4503-5411-0/17/11.
https://doi.org/10.1145/3139295.3139305

https://doi.org/10.1145/3139295.3139305

1 INTRODUCTION

Adaptive Mesh Refinement (AMR) describes a family of techniques allowing meshes to represent computational domains in an adaptive fashion, using higher resolution meshes to effectively compute multiscale phenomena with specific spatial regions of interest. One of the most popular and general approaches for rectilinear grid data is that of Berger and Colella [Berger and Colella 1989], commonly referred to as "block-structured AMR" (for our purposes, denoted BC AMR). BC AMR forms the basis for many codes including *LAVA* [Kiris et al. 2014], *Chombo* [Colella et al. 2000], *GR-Chombo*, used for the COSMOS gravitational waves simulation *grchombo*, and *Enzo* [O'shea et al. 2005], a radiation-hydrodynamics cosmic web code. Though other forms of AMR exist, block-structured AMR is perhaps the most general in that its structure could be used for octree and recursive-grid AMR with few modifications.

Despite the widespread application of AMR methods in high performance computing, methods for visualizing AMR data remain either inefficient or special-purpose. Widely used visualization packages such as VTK, ParaView, and VisIt have some level of support for BC AMR data, but rendering such data at interactive frame rates, with high visual quality (without cracks, holes, or artifacts), remains a challenge. As most visualization software pipelines are designed for non-adaptive data, data readers in these tools often "flatten" AMR data into regular grids for rendering, at the expense of significant time and memory. Moreover, visualizing specific coarser AMR levels, as shown in Figure 2, may be desirable for validation. However, existing direct rendering mechanisms for AMR data rely on their own internal data formats, and are often built for specific architectures (i.e., GPUs). While numerous GPU AMR implementations exist, they have not by and large been integrated into production visualization pipelines, chiefly due to the special-purpose nature of both the hardware and data they were designed to support. Efficient



(a) Level 0-1 (b) Level 0-2 (c) Level 0-3 (d) Level 0-4

Figure 2: Different AMR levels on the Landing Gear, using the *blend* method (Section 4.3).

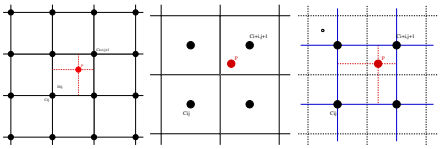


Figure 3: Trilinear interpolation for regular grids: For node-centered data (left) one simply interpolates between the cell vertices. For cell-centered data (center) one interpolates the respective dual grid (right).

and general means of directly rendering AMR data on the CPU remain desirable – due to the prevalence of CPU architectures in HPC and their large on-board memory, which helps in visualizing larger data on fewer resources.

In this paper, we detail a CPU-based approach for direct volume rendering of AMR data. Our contributions are:

- Four strategies for reconstruction of BC AMR data, one of which in particular (basis functions) is non-trivial and novel in that it interpolates across all AMR levels, remedies stitching artifacts, and does not rely on interpolation of non-rectilinear grids.
- A general data structure for efficient query, sampling and direct rendering of Berger-Colella AMR data, adapted for Chombo and VTK hierarchical grid data, suitable for any block-structured, recursive-grid or octree data.
- Implementation of these methods in the OSPRay CPU ray tracing framework and integration with ParaView for use in production.

We show that our implementation is competitive in performance with existing approaches, presents novel sampling methods, requires relatively modest hardware resources, supports full ray tracing of AMR volume data (including volumetric lighting, ambient occlusion, and proper integration with geometric surfaces, Fig. 8), and moreover provides a general framework for AMR visualization with a variety of reconstruction options.

2 RELATED WORK

Adaptive mesh refinement was first introduced by Berger and Oliger [Berger and Oliger 1984], using binary decomposition structures (quadtree, octree) in which refinement levels advanced by one step. More general block structured AMR data, consisting of nested grids of arbitrary dimension, was later proposed by Berger and Colella [Berger and Colella 1989]. This has subsequently been used in computational packages such as Chombo [Colella et al. 2000], GR-Chombo [Clough et al. 2015], Enzo [O'shea et al. 2005], etc.

The earliest AMR volume rendering system was introduced by Ma and Crockett [Ma and Crockett 1997] and was based on cell projection [Max 1993]. This approach was later extended to support MPI parallel rendering [Ma 1999]. Park et al. [Park et al. 2002] then used splatting to render interactively on a single workstation. Though efficient, resampling and rasterizing the AMR field into polygonal data could result in interpolation artifacts. Later GPU approaches employed special-case ray casting solutions to achieve interactive performance. Kahler and Hege [Kähler and Hege 2002] used slice-based volume rendering of 3D textures on the GPU to locally represent and resample AMR blocks, overlapping coarse and fine-resolution AMR levels. This system was later extended to use GPU ray casting, first by compositing multiple passes [Kähler et al. 2006] and then in a single pass [Kähler and Abel 2013]. Later GPU approaches employed more sophisticated and special-purpose techniques. Gosink [Gosink et al. 2008] devised an out-of-core query mechanism for resampling AMR data into structured volumes for rendering on the GPU. Marchesin and de Verdiere [Marchesin and De Verdiere 2009] employed a special-case solution for analytical ray casting of hexahedral cell data using piecewise-polynomial approximations, subdividing coarse AMR cells at junctions. More recently, Leaf et al. [Leaf et al. 2013] showed an efficient method for rendering AMR data in distributed parallel settings, using a method similar to that of Ljung [Ljung et al. 2006].

The problem of interpolating or "stitching" between adjacent AMR cells of different resolution recurs in the literature. Weber et al. [Weber et al. 2012, 2003] subdivide lower-resolution hexahedral cells into multiple stitch cells and reformulate the interpolant accordingly. Moran and Ellsworth [Moran and Ellsworth 2011] extend this to AMR cells differing in resolution by more than one level. As with [Beyer et al. 2008], the latter performs reconstruction at boundaries by subdividing cell-centered data into non-rectilinear simplicial elements (i.e., tet, hex). Our basis function method addresses this issue without subdividing.

Our actual implementation builds upon OSPRay [Wald et al. 2017], a general-purpose framework for ray-tracing based visualization. The approaches in this work are not specific to OSPRay or SPMD ray tracing, and would generalize to other rendering systems and architectures.

3 BACKGROUND

AMR data are used to compute and represent multiscale continuous phenomena. While AMR attributes may represent vector or tensor fields and overlaid particle data, they are most commonly used to represent scalar fields. For the purposes of this work we are concerned with three-dimensional data evolving over time, i.e. $f(\vec{x})$ for some $\vec{x} = (x, y, z) \in \mathbb{R}^3$. Reconstruction of data from scalar fields requires a filter, which depends on an underlying grid or mesh topology of discrete data. For structured rectilinear data the most common filter kernel is the trilinear interpolant.

In a structured grid, data can be either cell-centered or node-centered. For single-level structured data this distinction is not significant, since any cell-centered grid can be treated as the dual of a node-centered grid (Figure 3). However for multilevel AMR data, interpretation as node or cell-centered has significant implications, as discussed in Section 3.2.

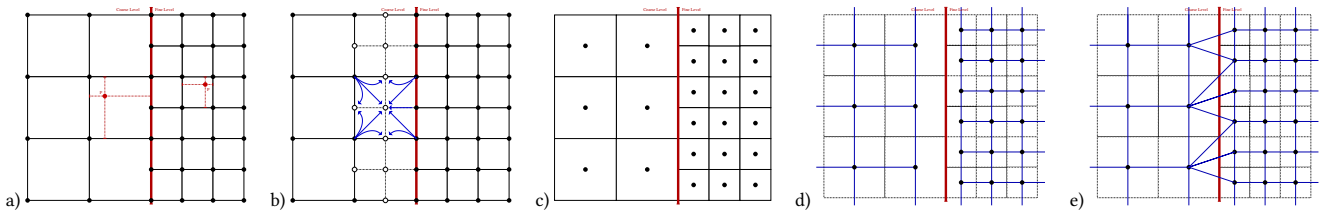
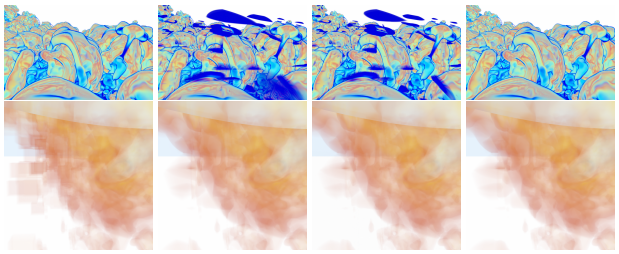


Figure 4: a) For *node*-centered AMR, both sides of a level boundary form structured node-centered grids that naturally lend to trilinear interpolation. b) If any stitching is necessary between the different levels' boundary values, this can easily be done by introducing one layer of ghost cells, which will always align nicely with both sides (top right). c-e) For *cell*-centered AMR (c) there are still obvious dual grids in regions of same refinement level (d), but stitching across the boundary requires unstructured mesh geometry (i.e., tets or hexes) (e).



(a) Finest **(b) Current** **(c) Blend** **(d) Basis**

Figure 5: Reconstruction quality for the four different methods described in this paper, on the example of an AMR representation of the well-known Richtmyer-Meshkov data set (top), and the Landing Gear (bottom). The *current* and *blend* methods can in some cases lead to artifacts we refer to as "ghosting". Our highest-quality reconstruction method, *basis* , does not use inner nodes, and thus never exhibits this artifact.

3.1 Berger-Colella AMR Data

Berger-Colella AMR data, as shown in Figure 4, is specified as set of *bricks* $B^{(i)}$ ($i = 0..N_{BRICKS}$) of typically around $16 \times 16 \times 16$ data cells each (bricks can have other sizes, but 16 seems a common value), with a data value specified for the center of each cell. Each brick lies on a specific level $L^{(i)}$; bricks on the same level do not overlap, but do overlap other bricks on coarser levels. Where finer bricks overlap coarser ones they do so in a way that the finer-level brick aligns with cell boundaries on the coarser level, and such that each coarser cell is covered by exactly $R \times R \times R$ finer cells, where R is called the *refinement factor*. The root level bricks together generally form a structured grid; finer levels are typically sparse.

Reconstruction Goals. There is no single "correct" solution to reconstruction from multilevel AMR data; nearest-neighbor, trilinear interpolation, and higher-order basis functions are all valid options for reconstructing structured data, but with different properties. Likewise, AMR may differ in how blocks are laid out, differences in resolution between blocks, and desired interpolation behavior and performance. Generally, we desire a reconstruction technique that is simple, performant, and smooth. We seek approaches which are interpolating (i.e. for any point $\vec{x}$ for at which a data value v exists, $F(\vec{x}) = v$), and moreover, that obey partition-of-unity. Approaches requiring only local support are preferred for efficiency – as well as methods that are locally trilinear, but continuous across level boundaries. The methods described in Section 4 each provide tradeoffs with respect to these criteria.

3.2 Structure of AMR Data

In this section we characterize the structure of BC AMR data, defining a common terminology which can then be used when describing our reconstruction strategies.

Logical Nested Grid Space: In order to abstract from the actual input data layout (individual bricks on different levels) we refer to cells in what we call *logical nested grid space*. We view the AMR data as a succession of levels $L^{(l)}$ ($l = 0..maxLevel$), where each level is logically a structured grid (an AMR *block*, or *patch*) of $N_x^{(l)} \times N_y^{(l)} \times N_z^{(l)}$ *logical* cells $C_{i,j,k}^{(l)}$, with the caveat that some of these cells actually exist, and others do not (see Figure 4). We refer to those logical cells that do exist in the input as *actual* cells, while those that do not as *virtual*. Each cell C has a center C_p (C.P in pseudocode below), a level C_l, etc. Actual cells also have a scalar field value C_v , while virtual ones do not. We can logically extend this grid to infinity by considering all cells outside the root level's bounding box as virtual. For any level l and 3D point $\vec{P}$ we define an operator $C^{(l)}(\vec{P})$ that maps $\vec{P}$ to the logical cell it lies in.

Closest Existing Cell: For each logical cell C we can further define what we call the *closest existing cell* (CEC) $\hat{C}$ of C: if C is an actual cell, its CEC is just that; if it is not, we define $\hat{C}$ as the finest-level cell for that location that a vertex of that cell (denoted C_p) lies in. We assume that this CEC-operator will, for any vertex C_p outside this bounding box, move C_p to its respectively closest position inside the domain. Similarly, we can define the closest existing level-l cell $\hat{C}^{(l)}(\vec{P})$ as the CEC of $C^l(\vec{P})$; and the *closest existing leaf* (CEL) $\hat{C}(\vec{P})$, where a leaf cell is the lowest level actual cell at point P with scalar values.

Dual Cell: Using our logical grid terminology allowed us to more easily reason about logical cells $C_{i,j,k}^{(l)}$. In a similar fashion, we can now define logical dual cells $D_{i,j,k}^{(l)}$, spanned by $C_{i,j,k}^{(l)}$ and $C_{i+1,j+1,k+1}^{(l)}$. As with logical cells, we assume that there is a kernel $D^{(l)}(\vec{P})$ that computes the coordinates of the dual cell, as well as the CEC of each of its 8 corner vertices. For each of these corners we have access to both its logical coordinates C as well as the *actual* coordinates (and value) of its CEC, $\hat{C}$. This allows for determining which of the corners C actually exist ($C = \hat{C}$), and which ones are virtual ($C \neq \hat{C}$).

Cell Location: In all our methods we assume that it is possible to efficiently query cells, dual cells, etc. In our pseudo-codes we assume a kernel `findLeaf(P)` that finds the CEL of $\vec{P}$, a `findCell(l,P)` that finds the CEC $C^{(l)}(\vec{P})$, a `findDual(l,P)` that finds the level-l

dual cell $D^{(l)}(\vec{P})$ (and the CECs of its corners), and a `D.lerp(P)` that computes trilinear interpolation inside D.

4 RECONSTRUCTION STRATEGIES

Similar to nearest-neighbor filtering for textures or structured data, the simplest reconstruction strategy is to look up the leaf cell containing the query point, and return its value:

```
float nearest(P)
  C = findLeaf(P)
  return C.v
```

This is fast and simple, but not continuous even in same-level regions, which rather limits is usefulness.

4.1 Finest-Level Interpolation

Thanks to our logical grid abstraction we can view each specific level as a structured grid, with values for non-existing cells defined through the CEC operator. In particular, this allows for picking any logical level l, and trilinearly interpolating on it. This approach is cheap and continuous, but not adaptive. The logical grid is cell-centered, but can be interpolated using its dual:

```
float lerpOnLevel(l,P)
  D = findDual(l,P)
  return D.lerp(P)
```

4.2 Current-Sample Leaf-Level Interpolation

Rather than use a fixed level, we can also, for each sample, look up that sample's leaf level, and then interpolate on that level. This *current* method is no longer continuous across level boundaries, but otherwise quite useful: it is adaptive, locally trilinear, simple, and fast. In particular, in all regions except boundaries it is the same as trilinear interpolation on that given region's leaf level.

```
float leafLevelLerp(P)
  C = findLeafCell(p)
  return lerpOnLevel(C.l,P)
```

4.3 Blending Between Levels

The cause of the previous method's discontinuities is that, though each level's interpolant is continuous, what we used for selecting the level is not. One way of fixing this is to smoothly *blend* between levels. In particular, we can view virtual cells as transparent, and actual ones as opaque. We can then assign opacities of 1 and 0 to the centers of actual and virtual cells respectively, and trilinearly interpolate between these. This yields a continuous blending function that we can use to blend between any level l and the coarser one(s) below:

```
float blendNaive(P)
   float f = 0
   for (level=0,1,..MAXLEVEL)
      (f_l,a_l) = lerpOnLevel(l,P)
      if (a_l == 0) break
      f = a_l*f_l+(1-a_l)*f
   return f
```

In homogeneous regions, this method is the same as trilinear interpolation on that level; across boundaries it smoothly blends between the adjoining regions' interpolants. This method is thus adaptive and continuous; and (though less obviously so) interpolating. It is however expensive, as it would have to perform a dual-cell look-up on every level. In practice, however, very few levels will contribute to any point $\vec{x}$: many fine levels will be completely transparent at $\vec{x}$, and everything below a completely opaque level would be weighted by 0. This suggests an optimization where we either start at the coarsest level and blend "upwards" until we reach a completely transparent level (at which point no finer levels can contribute); or start at the finest one and blend "downwards" until we reach a completely opaque one. In fact, we can start at the leaf level of $\vec{x}$, and then blend both upwards *and* downwards until the finest and coarsest contributing levels have been found. This faster *blend* approach, is implemented as follows:

```
float blendFast(P)
  // find leaf level, and lerp
  C = findLeafCell(P)
  D = findDualCell(C.l,P)
  f = D.lerp()
  // blend towards finer
  D' = D
  for (l = C.l+1 ... MAXLEVEL)
     if (all vertices of D' are leaves) break
     D' = findDualCell(l,P)
     (f_l,a_l) = D'.lerp(P)
     f = a_l*f_l + (1-a_l)*f
  // blend towards coarser
  if (any vertices in D are virtual) {
    a = D.lerpAlpha(P)  //lerp only alpha
    f = f*a
    for (l = C.l-1 ... 0)
       D' = findDualCell(l,P)
       f += (1-a) * D'.lerp(P)
       if (all vertices in D' exist) break
       a += (1-a)*D'.lerpAlpha(P)
  return f;
```

4.3.1 Ghosting Artifacts. Both the *current* and *blend* approaches can lead to an unwanted artifact we refer to as *ghosting*, shown in Figure 5. This occurs when a given coarse cell has a neighbor that is further refined, and whose fine values undergo a sharp transition from one value to the next. In this case, for reconstruction methods that involve the neighboring cell's *inner* value (as happens for both *current* and *blend*) the value used by this interpolant is not representative of the fine cells' values at the boundary.

Note in particular, this effect is *not* the same as a discontinuity, and can appear even when smoothly and continuously blending across levels. That said, the artifact is rather rare, and limited to strongly varying data sets rendered with spiky transfer functions. It can also be avoided by employing filters that never user inner-node values, such as the *basis* method presented in the following section.

4.4 Reconstruction via Basis Functions

Though often seen as a form of "blending" between extremal values, regular trilinear interpolation can also be viewed as the sum of 8

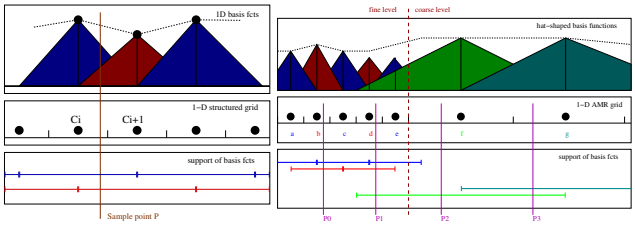


Figure 6: 1D-illustration of reconstruction via basis functions. Left: trilinear interpolation as a sum of linear (hat) basis functions. In this case, the basis functions sum to 1, and only one basis function has non-zero support at any data point. Right: The same for our generalization to BC AMR data: the function is still smooth, continuous, and adaptive, but at least at level boundaries the basis functions no longer sum to 1, and may overlap multiple fine-level cells.

hat-shaped basis functions located at the dual cell's corners:

$$\text{lerp}(\vec{P}, D) = \sum_{C \in corners(D)} \hat{H}_C(\vec{P}) C_v, \quad (1)$$

using the hat-shaped basis functions:

$$\hat{H}_C(\vec{P}) = \hat{h}\left(\frac{|\vec{C_{p,x}} - \vec{P}_x|}{C_w}\right)\hat{h}\left(\frac{|\vec{C_{p,y}} - \vec{P}_y|}{C_w}\right)\hat{h}\left(\frac{|\vec{C_{p,z}} - \vec{P}_z|}{C_w}\right), \quad (2)$$

with $\hat{h}(t) = \max(1-t, 0)$. For each cell C this basis function would be centered at C_p and have a support width of $\pm C_w$.

Borrowing some concepts from *scattered data interpolation* techniques (see, e.g. [Franke and Nielson 1980]) we can now view our AMR data points as a sort of scattered data points $(C_i)_{i=0}^{N}$ with basis functions $\hat{H}_C$, and—using Franke-style scattered data interpolation—can reconstruct via a weighted and re-normalized sum of those basis functions:

$$AMR_{basis}(\vec{P}) = \frac{\sum_{C_i} \hat{H}_{C_i}(\vec{P}) C v_i}{\sum_{C_i} \hat{H}_{C_i}(\vec{P})} \quad (3)$$

Though Franke's scattered-data interpolation method does not specify which basis functions to use (and is often used with Gaussian or other basis functions), our choice of $\hat{H}_C$ is deliberate: they are easy to compute, and in most regions will automatically yield exactly the same interpolant as trilinear interpolation. Around boundaries we get the superposition of basis functions from different levels, which will smoothly blend between levels; and as we never use virtual or inner cells we see significantly reduced ghosting.

For any point $\vec{P}$, computing this interpolant requires finding all leaf cells C that have non-zero contribution at $\vec{P}$. For our choice of basis function, on each level L only the eight corners of $\vec{P}$'s dual cell can possibly contribute, leading to a very simple implementation:

```
float AMR_basis(P) =
  float sum_weights = 0
  float sum_weightedValues = 0
  for (l = 0 ...)
    D=findDualCell(P)
    foreach corner cell C of D
      if (C is a leaf cell)
        sum_weights += H_hat(P,C)
        sum_weightedValues += H_hat(P,C)*C.v
    if (none of the C in D are inner nodes)
      break
  return sum_weightedValues / sum_weights
```

The basis function method is illustrated in Figure 6. On the upside, this interpolant is easy to implement, smooth, continuous, and produces good image quality. On the downside, it is no longer strictly interpolating, and also no longer obvious how to do implicit ray-isosurface intersection. In terms of performance, we can easily determine the *finest* level that contributes, but have not yet found a good way of knowing the coarsest one that does; this means that in deeply refined regions we potentially have to perform many dual-cell look-ups, which is costly.

5 IMPLEMENTATION

This section details the implementation of the AMR data structure, traversal and rendering methods in OSPRay.

5.1 AMR KD-Tree and Cell Location

Regardless of the context of ray tracing where we want to use our reconstruction strategies, we need efficient methods to compute the cell location kernels they are built on. Our initial approach used a multi-octree data structure, whose traversal was efficient, but required completely reformating the input data; and thus could not directly operate on native VTK data, instead requiring at least one copy of the data.

To remedy this, we adopted an alternative approach similar to a median-split kd-tree build: we start by looking at all bricks across all levels, and compute the world space bounding box of these bricks. We then recursively partition this space as follows: First, we determine a list of all block boundaries that intersect the space we want to partition (each such boundary defines an axis-aligned plane). We then pick one of those as a kd-tree partitioning plane—we currently use the one closest to the spatial median—and use this to partition the current domain into left and right halves. We then go over all blocks in the current region and sort them into those overlapping the left half, and those overlapping the right; those that overlap both (which is perfectly possible in BC AMR) go into both. Finally, we recurse on the left and right halves until no more boundary plane overlaps the current region, in which case we make a leaf. We can easily build this data structure over an existing `vtkHierarchicalBoxDataSet` structure as used by ParaView's and VitIt's Chombo readers, without replicating any of the voxels. An illustration of this data structure and its construction is given in Figure 7.

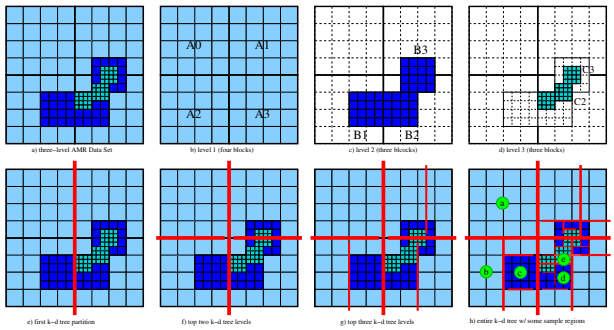


Figure 7: Illustration of the kd-tree we build over unique block regions. a-d) A sample BC AMR data set consisting of 10 data blocks across 3 levels e-g) The first three subdivions performed by the k-d tree builder. h) The full k-d tree, with some sample regions 'a'-'e'. Any brick may be referenced by multiple leaves, and leaves typically cover many cells; but each leaf refers to a spatial region that contains exactly one brick per level it covers.

5.2 OSPRay Integration

Though the reconstruction strategies and kernels are generally applicable, for the remainder of this paper we consider an implementation within the OSPRay ray tracing framework [Wald et al. 2017]. OSPRay already comes with a ray tracer, a renderer that supports volume ray casting, and with ready abstractions to implement new volume types. To make it support our methods, we had to set up the data structures and build the kd-tree; and implement the cell location kernels and reconstruction strategies described previously. Once the respective `ospray::ChomboVolume` data type was implemented, it worked out-of-the-box with OSPRay's exisiting renderers.

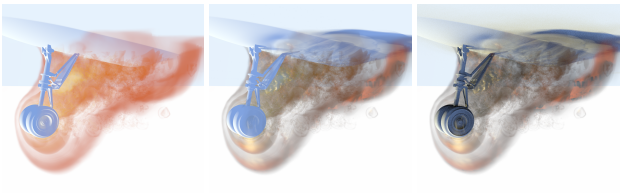


Figure 8: Left to Right: The landing gear with volume ray marching of the AMR data, volumetric shadows, ambient occlusion and volumetric shadows.

5.3 Fast Cell/Dual-Cell Queries

The key factor determining rendering performance is how quickly we can perform cell and dual cell queries. We implemented our reconstruction kernels in ISPC, performing N reconstructions (and correspondingly, N cell/dual cell queries) in parallel (where N is the vector width of the underlying CPU architecture). Rather than operating on logical integer IDs for cells and levels we reference cells by their (float) center point and levels by their (float) center width, which better utilizes floating point units, minimizes register pressure and stack space, avoids costly int-to-float conversions when traversing the data structure, and allows for utilizing modern vector units' multiply-add capabilities (even some of the address computations inside each brick can be done in floating point!). In addition, we implemented cell location in a "packet" paradigm [Wald et al. 2001] in which all N queries stay together while going down the tree, and realized the recursion with a software-maintained stack. These optimized kernels were an order of magnitude faster than the scalar reference code.

For dual-cell query we implemented a special variant built on the packetized cell location kernel which actually performs all 8 corner queries in a single traversal. To do this, we view the 8 vertex locations as the intersection of three sets of parallel planes (the planes that form the boundary of the dual cell), and traverse these *planes* down the kd-tree. At each kd-tree node we check on which side of the node's partitioning plane the respective query planes lie, and can consequently track, with only 6 state variables, which of the planes are active in a given subtree. Once a leaf is reached, which of the 6 planes are active defines which of the 8 corner values are active, and those that are can be filled from the brick stored in the given leaf.

5.4 VTK and ParaView

OSPRay was officially integrated into VTK 7.1 and ParaView 5.0 with support for surface rendering and volume rendering of regular grids. Before our framework existed, ParaView handled AMR data by loading it into a `vtkAMR` data structure (which implements a Berger Colella layout), and resampling this into a structured volume, from which it could be rendered using the standard volume renderer. This also worked with ParaView's OSPRay volume renderer, but incurred the cost in memory and resampling time, as well as the quality issues of resampling. In order to directly volume render the `vtkAMR` structure, we created a `vtkAMRVolumeMapper` that does not resample the data, and which instead directly maps the `vtkAMR` structure to OSPRay's internal AMR representation, using zero-copy memory sharing where possible. This results in a direct implementation in ParaView with little memory overhead, which is also able to render with OSPRay geometry and composite with ParaView's OpenGL framebuffer. A still from a resulting movie rendered with our implementation in ParaView is shown in Fig 1.

6 RESULTS

We evaluate the presented techniques using two machines representative of typical HPC resources for scientific visualization: a dual-socket workstation, and some of the Intel Xeon Phi "Knights Landing" nodes on the "Stampede 2" supercomputer. The Xeon workstation has two Intel Xeon E5-2699 v4 CPUs, with a total of 44 cores (88 threads) running at 2.2 GHz, with 256 GB of RAM and a Matrox MGA G200e for display. The Stampede KNL nodes are compute nodes within the Stampede 2 supercomputer at the Texas Advanced Computing Center. In this setup we used VNC for remote rendering, using OpenSWR-enabled Mesa to drive the viewer's user interface. The KNL nodes use Intel Xeon Phi 7250 "Knights Landing" (KNL) processors with 68 cores (4 hardware threads per core), and 16 GB of MCDRAM in cache mode, as well as 96 GB of DDR4 RAM.

For benchmarking we use three AMR data sets: `LandingGear` is a Chombo [Colella et al. 2000] data set provided by NASA AMES with 9 refinement levels; `BHM` (black hole merger) is a GR-Chombo [Clough et al. 2015] simulation of gravitational waves resulting from the collision of two black holes with 3 refinement levels, provided by Cambridge's COSMOS team; and `spheres` is a direct numerical simulation of flow around a sphere by Trebotich et al. [Trebotich and Graves 2015], with 3 levels that each have a refinement factor of 4.

In addition, we wrote a tool that converts a structured input volume into Chombo-style AMR by first bricking it into 16^3 bricks, then eliminating all bricks whose value does not vary beyond a given threshold. We applied this process to various structured data sets such as the $2k^3$ Richtmyer-Meshkov instability `LLNL`, and the $2048 \times 512 \times 1536$ voxel `DNS` data set of turbulent channel flow [Lee et al. 2013].

6.1 Rendering Quality

We show comparisons of the `LandingGear` dataset from levels 0-4 in Figure 2. The coarsest level shows an unrecognizable section of flow around the landing gear, while finer levels reveal increasingly smaller details. These images used the *blend* method to smoothly interpolate across all levels. To demonstrate the differences between the sampling strategies Figure 5 shows zoomed-in sections of the `LandingGear` and `LLNL` datasets: In those images the *finest* method produced artifact-free images for the converted `LLNL`, for which it actually produces exactly the same output as the original structured model would have yielded before converting it to AMR. For "true" AMR, however, *finest* can yield artifacts at course cell boundaries, caused by high-frequency transitions between two neighboring

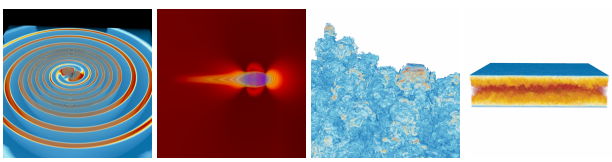


Figure 9: Images of the benchmarking scenes: `BHM`, `Sphere Flow`, `LLNL`, and `DNS`.

Table 1: Rendering performance on a dual-Xeon workstation in frames per second for varying reconstruction filters and maximum AMR levels. For reference, the last column on the right shows performance on a single KNL node on Stampede 2.

LandingGear 57GB	lvl0	lvl1	lvl2	lvl3	lvl4	knl4
Cells	4	8	52	560	2056	2056
Current	13.12	13.09	8.44	5.35	4.29	2.36
Basis	11.40	11.34	5.79	2.54	1.83	0.99
Blend	12.44	12.12	7.78	4.84	3.88	2.08
Finest	14.65	13.91	9.67	6.31	5.28	2.94
BHM 28GB	lvl0	lvl1	lvl2	lvl3	lvl4	knl4
Cells	4096	4098	4106	4114	-	4114
Current	25.00	24.96	25.26	25.58	-	14.52
Basis	29.57	28.59	28.52	27.49	-	15.86
Blend	24.94	25.64	25.36	24.32	-	14.32
Finest	30.48	29.17	26.36	26.25	-	14.93
Sphere 6GB	lvl0	lvl1	lvl2	lvl3	lvl4	knl4
Cells	1024	1034	1214	-	-	1214
Current	3.79	3.64	3.57	-	-	2.22
Basis	3.87	3.89	3.76	-	-	2.43
Blend	3.76	3.59	3.47	-	-	2.15
Finest	4.36	4.27	4.19	-	-	2.77
LLNL 8GB	lvl0	lvl1	lvl2	lvl3	lvl4	knl4
Cells	31k	56k	29M	-	-	29M
Current	4.89	4.31	3.36	-	-	1.833
Basis	5.51	4.35	2.76	-	-	1.53
Blend	4.77	4.07	3.06	-	-	1.68
Finest	5.74	5.72	4.65	-	-	2.61
DNS 7GB	lvl0	lvl1	lvl2	lvl3	lvl4	knl4
Cells	6144	399k	24M	-	-	24M
Current	44.51	22.55	5.67	-	-	2.88
Basis	50.71	14.08	2.75	-	-	1.12
Blend	45.13	22.31	5.52	-	-	2.78
Finest	52.81	24.71	6.53	-	-	3.49

coarse cells' values. *Current* and *blend* usually perform well in practice (e.g., in the `LandingGear`).

It is difficult to compare quality and performance of AMR data with that of flattened finest-level resolution equivalent structured grids. For example, for the `LandingGear` the width of a cell on the finest level is only 0.00024 times that of the coarsest: sampling the entire domain this would require roughly 100,000 voxels per axis, or 10^{15} total voxels.

6.2 Performance Evaluation

Table 1 displays performance for each reconstruction method across multiple refinement levels (level #0 containing only the coarsest level, level #4 containing all 5 levels). Performance is measured in frames per second at 1024×1024 pixels, and with OSPRay's adaptive sampling and preintegration enabled. We measure performance across levels on the 44-core dual-Xeon workstation.

As expected, performance strongly depends on the occurrence of samples taken in finer (and thus, costlier) regions: The `LandingGear` dataset has a high coverage of finer levels in the chosen view, so increasing the maximum reconstruction level results in more samples taken per ray at increasingly smaller ray steps, and with deeper refinement levels, resulting in a bigger performance impact. The `BHM` dataset has finer levels concentrated at the center of the dataset, with the majority of the waves being at coarser levels. This results in finer levels only affecting a small portion of the rays, and thus performance is almost oblivious to the maximum refinement level. *Finest* achieves highest performance but the worst quality, while *basis* is most expensive. The *current* method is a good trade-off; it has the second highest performance, and though it *can* produce ghosting (see Fig. 5) in practice these artifacts are rare.

In terms of hardware, we found a KNL node to be roughly equivalent to a single socket of the dual-Xeon node. This is worse than our experience with other rendering methods, and likely caused by the code- and data-divergent nature of our reconstruction kernels' SPMD implementation.

A full exploration of OSPRay's distributed rendering performance is out of the scope of the paper; however, we perform a preliminary scaling benchmark with the `LandingGear` on up to 32 nodes of Stampede 2. Rendering performance started at 2.36 FPS with 2 nodes (with one node used for display); 4.39 FPS with 3 nodes; 16.79 FPS with 17 nodes; and 21.89 FPS with 32 nodes.

6.3 Comparison to Other Approaches.

Overall, our AMR rendering system shows competitive performance, ranging anywhere from 2–50 MRays/s on a dual-Xeon workstation, depending on data set, reconstruction strategy and chosen level(s). Previous mixed-resolution single-GPU approaches [Beyer et al. 2008; Ljung et al. 2006] suggested interactive performance (15 MRays/sec and above for the method of Beyer et al. [Beyer et al. 2008] on an NVIDIA 280 GTX), but with the limitations that they generally interpolated only between two levels per sample, and involved generally smaller data. The more recent single-pass GPU approach of Kahler and Abel [Kähler and Abel 2013] reported frame rates of 0.5–4 MRays/s, using a variety of strategies. The performance achieved by Leaf et al. [Leaf et al. 2013] on a GPU cluster varies with the type of dataset; their strong scaling results suggest a render time of 2 MRays/s (0.5 fps for a 4 MP frame buffer) on 128 GPUs, with a constant sample rate of 8 samples per voxel.

7 CONCLUSION AND FUTURE WORK

We have presented a framework for efficient CPU-based volume rendering of Berger-Colella AMR data. We achieve consistently interactive performance for gigascale AMR data sets, even using the costliest reconstruction strategies (*basis*, *blend*) across all levels. Our approach is competitive with similar approaches on single GPUs, and even GPU clusters, while scaling on CPU clusters. As an OSPRay module with a standalone integration in ParaView, our system could be easily adopted for production use. AMR continues to be vital in large-scale simulations, and this method for Berger-Colella block-structured data can be applied to other multiresolution rectilinear data (e.g., octrees).

Though our framework already achieves both good image quality and interactive performance, some work remains to be done. The hierarchical nature of the input data, and our use of the kd-tree, suggests space skipping could be tailored to the AMR heirarchy for better performance and quality. Extending this work to direct ray-isosurface intersection would also be desirable [Wald et al. 2005]. Data-distributed AMR is another extension worth exploring for time series. Finally, performance could likely be improved by manually coded, low-level reconstruction kernels written in SIMD wrappers, as opposed to the current SPMD code in ISPC.

ACKNOWLEDGMENTS

Patrick Moran from NASA Ames graciously offered use of the Landing gear dataset. We would like to thank Juha Jaykka and Paul Shellard from the Stephen Hawking Centre for Theoretical Cosmology for use of their cosmos AMR dataset. David Trebotich and Gunther Weber from Lawrence Berkeley NL provided the sphere flow dataset.

REFERENCES

Marsha J Berger and Phillip Colella. Local adaptive mesh refinement for shock hydrodynamics. *Journal of computational Physics* 82, 1 (1989), 64–84.

Marsha J Berger and Joseph Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of computational Physics* 53, 3 (1984).

Johanna Beyer, Markus Hadwiger, Torsten Möller, and Laura Fritz. 2008. Smooth mixed-resolution GPU volume rendering. In *Proceedings of the Fifth Eurographics/IEEE VGTC conference on Point-Based Graphics*. 163–170.

Katy Clough, Pau Figueras, Hal Finkel, Markus Kunesch, Eugene A. Lim, and Saran Tunyasuvunakool. Numerical Relativity with Adaptive Mesh Refinement. *Classical and Quantum Gravity* 32, 24 (2015).

P Colella, DT Graves, TJ Ligocki, DF Martin, D Modiano, DB Serafini, and B Van Straalen. 2000. Chombo software package for AMR applications-design document. (2000).

Richard Franke and Greg Nielson. Smooth interpolation of large sets of scattered data. *Numerical Methods in Engineering* 15 (1980). Issue 11.

Luke J Gosink, John C Anderson, E Wes Bethel, and Kenneth I Joy. Query-driven visualization of time-varying adaptive mesh refinement data. *IEEE transactions on visualization and computer graphics* 14, 6 (2008), 1715–1722.

Ralf Kähler and Tom Abel. 2013. Single-pass GPU-raycasting for structured adaptive mesh refinement data. In *IS&T/SPIE Electronic Imaging*. 865408–865408.

Ralf Kähler and Hans-Christian Hege. Texture-based volume rendering of adaptive mesh refinement data. *The Visual Computer* 18, 8 (2002), 481–492.

Ralf Kähler, John Wise, Tom Abel, and Hans-Christian Hege. 2006. GPU-assisted raycasting for cosmological adaptive mesh refinement simulations.. In *Volume Graphics*. 103–110.

Cetin C Kiris, Michael F Barad, Jeffrey A Housman, Emre Sozer, Christoph Brehm, and Shayan Moini-Yekta. The LAVA Computational Fluid Dynamics Solver. *52nd Aerospace Sciences Meeting, AIAA SciTech Forum* 70 (2014).

Nick Leaf, Venkatram Vishwanath, Joseph Insley, Mark Hereld, Michael E Papka, and Kwan-Liu Ma. 2013. Efficient parallel volume rendering of large-scale adaptive mesh refinement data. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization*. 35–42.

Myoungkyu Lee, Nicholas Malaya, and Robert D. Moser. 2013. Petascale Direct Numerical Simulation of Turbulent Channel Flow on Up to 786K Cores. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. Article 61.

Patric Ljung, Claes Lundström, and Anders Ynnerman. Multiresolution interblock interpolation in direct volume rendering. (2006), 259–266.

Kwan-Liu Ma. 1999. Parallel rendering of 3D AMR data on the SGI/Cray T3E. In *The 7th Symposium on the Frontiers of Massively Parallel Computation*. 138–145.

Kwan-Liu Ma and Thomas W Crockett. 1997. A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data. In *Proceedings of the IEEE symposium on Parallel rendering*.

Stéphane Marchesin and Guillaume Colin De Verdiere. High-quality, semi-analytical volume rendering for AMR data. *IEEE Transactions on Visualization and Computer Graphics* 15, 6 (2009).

Nelson Max. 1993. Sorting for polyhedron composition. In *Focus on Scientific Visualization*. 259–268.

Patrick Moran and David Ellsworth. Visualization of AMR data with multi-level dual-mesh interpolation. *IEEE transactions on visualization and computer graphics* 17, 12 (2011), 1862–1871.

Brian W O'shea, Greg Bryan, James Bordner, Michael L Norman, Tom Abel, Robert Harkness, and Alexei Kritsuk. 2005. Introducing Enzo, an AMR cosmology application. In *Adaptive mesh refinement-theory and applications*.

Sanghun Park, Chandrajit L. Bajaj, and Vinay Siddavanahalli. 2002. Case Study: Interactive Rendering of Adaptive Mesh Refinement Data. In *Proceedings of the Conference on Visualization '02 (VIS '02)*. IEEE Computer Society, Washington, DC, USA, 521–524. http://dl.acm.org/citation.cfm?id=602099.602186

David Trebotich and Daniel Graves. An adaptive finite volume method for the incompressible Navier–Stokes equations in complex geometries. *Communications in Applied Mathematics and Computational Science* 10, 1 (2015), 43–82.

Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek, and Hans-Peter Seidel. Faster Isosurface Ray Tracing using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics* 11, 5 (2005).

I Wald, GP Johnson, J Amstutz, C Brownlee, A Knoll, J Jeffers, J Günther, and P Navratil. OSPRay-A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics* 23, 1 (2017).

Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum* 20, 3 (2001), 153–164. (Proceedings of Eurographics 2001).

Gunther H Weber, Hank Childs, and Jeremy S Meredith. 2012. Efficient parallel extraction of crack-free isosurfaces from adaptive mesh refinement (AMR) data. In *2012 IEEE Symposium on Large Data Analysis and Visualization*.

Gunther H Weber, Oliver Kreylos, Terry J Ligocki, John M Shalf, Hans Hagen, Bernd Hamann, and Kenneth I Joy. 2003. Extraction of crack-free isosurfaces from adaptive mesh refinement data. In *Hierarchical and Geometrical Methods in Scientific Visualization*.