The Visual Computer manuscript No.
(will be inserted by the editor)

Aaron M. Knoll · Ingo Wald · Charles D. Hansen

Coherent Multiresolution Isosurface Ray Tracing

the date of receipt and acceptance should be inserted later

Abstract We implement and evaluate a fast ray tracing method for rendering large structured volumes. Input data is losslessly compressed into an octree, enabling residency in CPU main memory. We cast packets of coherent rays through a min/max acceleration structure within the octree, employing a slice-based technique to amortize the higher cost of compressed data access. By employing a multiresolution level of detail (LOD) scheme in conjunction with packets, coherent ray tracing can efficiently render inherently incoherent scenes of complex data. We achieve higher performance with lesser footprint than previous isosurface ray tracers, and deliver large frame buffers, smooth gradient normals and shadows at relatively lesser cost. In this context, we weigh the strengths of coherent ray tracing against those of the conventional single-ray approach, and present a system that visualizes large volumes at full data resolution on commodity computers.

Keywords ray tracing · isosurfaces · volume data · compression · level of detail

1 Introduction

Interactive rendering of large volumes is an ongoing problem in visualization. Adaptive isosurface extraction techniques are CPU-bound, and render a piecewise linear mesh that locally differs from the implicit interpolating surface on the source data. GPU direct volume rendering (DVR) delivers consistently real-time frame rates for moderate-size data; but GPU memory imposes a limit on the volume size. Although large data can be accessed asynchronously through out-of-core methods and progressive rendering, rasterization algorithms nonetheless have object-order complexity, which breaks down for sufficiently large data. Given a limited number of slices sampling a high-resolution volume, GPU DVR methods have difficulty rendering a precise surface, which is often desirable in scientific and medical analysis.

Scientific Computing and Imaging Institute,
University of Utah,
Salt Lake City, Utah 84112,
Tel.: (801)585-1867 Fax: (801)585-6513
E-mail: : {knolla|hansen|wald}@sci.utah.edu

Isosurface ray tracing of large volume data overcomes many of these issues. As it is not limited to polygonal geometry, it can directly render implicit surface patches as base primitives, and render exact piecewise-smooth isosurfaces in this manner. More significantly, ray tracing scales well to large data, particularly when scene complexity is high relative to the number of rays that must be cast to fill a frame. Although ray tracing is increasingly feasible on the GPU, rendering on the CPU allows for direct access to expandable mainboard memory, and greater control over hierarchical data structures than with current GPU hardware. This flexibility enables use of an adaptive-resolution octree, which we can use as both a natively compressed data format and an acceleration structure for rendering. Previous work ray-traced large octree volumes interactively, but required substantial workstation hardware [13]. In this paper, we optimize isosurface ray tracing with a coherent octree traversal technique, then employ a multiresolution level of detail (LOD) scheme to ensure coherence and hence performance. The resulting system allows for faster and improved-quality rendering on modest CPU hardware, and retains overall scalability to large data for which single-ray tracing methods have proven effective. Our paper is organized as follows: the next section reviews related work; Section 3 discusses motivation for the proposed system, and a technical overview. Section 4 illustrates octree volume construction. Section 5 details the coherent octree traversal algorithm and related optimizations for large volume data. Section 6 discusses the traversal-time multiresolution scheme. Section 7 covers shading modalities; and Section 8 analyzes our results.

2 Related Work

Mesh Extraction and Direct Volume Rendering

The conventional method for isosurface rendering has been extraction via marching cubes [20] or some variant; paired with rasterization of the resulting mesh. Wilhelms and Van Gelder [31] proposed a min/max octree hierarchy that allowed the extraction process to only consider cells containing the surface. This concept has been extended with frustum and per-ray visibility culling [18,17] and multiresolution volume data [30]. Livnat & Tricoche [19] effectively combined mesh extraction with point-based splatting for efficient isosurface rendering. Direct volume rendering (DVR) [16] is a popular alternative to isosurfacing, and efficient for moderate-size data on GPU's [3]. LaMar et al. [15] proposed a multiresolution sampling of octree tile blocks according to view-dependent criteria. Boada et al. [2] proposed a coarse octree built upon uniform sub-blocks of the volume, and a memory paging scheme. Large data has been addressed via block-based adaptive texture schemes (e.g. Kraus & Ertl [14]), and an octree hierarchy of wavelet-compressed blocks (e.g. Guthe et al. [9]).

Volumetric Isosurface Ray Tracing

Interactive isosurfacing of large volumes was first realized in a ray tracer by Parker et al. [23], using a hierarchical grid of macrocells as an acceleration structure. A single ray was tested for intersection inside a cell of eight voxel vertices, solving a cubic polynomial to find where the ray intersects the interpolant surface in that local cell. DeMarle et al. [5] extended this approach to clusters, allowing arbitrarily large data to be accessed via distributed shared memory. Coherent ray tracing [24,26,28] combined highly-optimized coherent traversal with SIMD primitive intersection to deliver up to two orders of magnitude increase in frame rate, allowing interactive ray tracing on a single processor. Marmitt et al. [21] adapted the trilinear interpolant path intersection test to a SIMD SSE architecture. Wald et al. [27] implemented a coherent SIMD isosurface ray tracer employing implicit kd-trees. This system was extended by Friedrich et al. [7] with an multiresolution volume hierarchy for efficient out-of-core progressive rendering. Knoll et al. [13] implemented a single-ray traversal scheme for rendering octree-compressed volume data. By employing one structure for both the min/max acceleration tree and the voxel data itself, this system rendered large volumes given limited main memory. While octree volume traversal incurred some penalty from looking up compressed data within the octree, it performed competitively with the best-known techniques employing either single rays or packets.

LOD Ray Tracing

Level of detail methods have already been employed in ray tracing. Igehy et al. [10] proposed ray differentials as a LOD metric for improved mipmap texture filtering. Yoon et al. [32] explored hierarchical splatting as a method of rendering massive mesh models. Djeu et al. [6] employed ray differentials in conjunction with subdivision surfaces for ray tracing LOD geometry.

3 Coherent Ray Tracing of Volume Data using LOD

The primary goal of this work is to optimize ray tracing of octree volumes, and ideally to deliver interactivity on commodity CPU's. Our main vehicle for such performance gains is coherence. The general premise is to assemble neighboring rays into groups, or packets, with common characteristics. Then, rather than computing traversal and intersection per ray, we perform these computations per packet. High coherence occurs when rays in a packet behave similarly, intersecting common nodes in the efficiency structure or common cells in the volume. Thus, coherence depends on scene complexity as defined by the dataset and camera position.

Coherence via Level of Detail

Successful coherent systems have been optimized for relatively small dynamic polygonal data [28,26] in which many rays intersect common primitives. In contrast, large volume data exhibit low spatial coherence, particularly from far-away camera positions. Isosurface ray tracing of large data using conservative 2x2 ray packets [27] has suggested performance generally on par with a single-ray system [13]. Coherent traversal may induce more intersection tests than a single-ray traversal; and without optimizations, actually perform worse than a single-ray tracer. To remedy this, we employ a multiresolution level of detail scheme: when data is sufficiently complex to hamper coherence, we render a coarser-resolution representation with higher coherence. The octree volume is inherently suited as a multiresolution LOD structure; coarser-resolution voxel data can be stored in interior nodes, allowing the original data, acceleration structure and all LOD's to be stored for a fraction of the original uncompressed data footprint. To render a coarser LOD, one simply specifies a cut of octree at a specified depth. The ray tracer then omits traversal and intersection of subtrees below that depth, and instead intersects coarser, larger cells at termination depth. As more rays intersect a common cell, coherence, and thus speedup, is achieved.

Overview

As shown in Fig. 1, our system consists of offline construction of the multiresolution octree structure from the original data (A); followed by rendering of this octree using a thread-parallel SSE-optimized packet ray tracer (B), e.g. [27]. The latter distributes ray packets to worker threads (C), which then perform per-packet coherent traversal, SSE isosurface intersection, and shading in that order. Our main contributions involve extending the static-resolution octree volume to multiresolution (Section 4); devising a coherent traversal technique for the octree (Section 5); and leveraging the

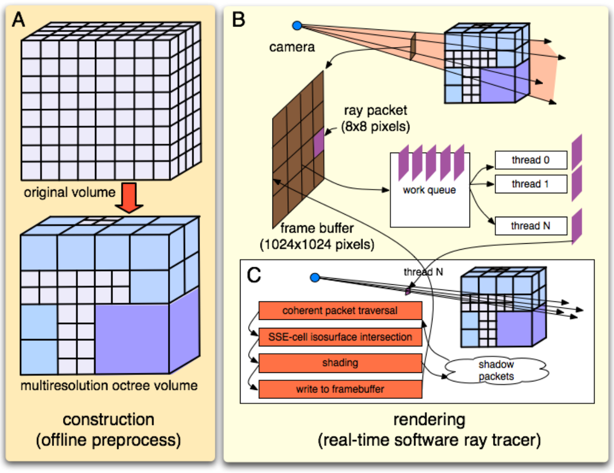


Fig. 1 *System overview.*

traversal technique to reduce the cost of compressed data access (Section 6). Ultimately, our system delivers interactive ray tracing on a desktop CPU while preserving image quality, and enables shading techniques that would be expensive in a conventional non-coherent octree volume ray tracer. Moreover, it allows for scalable rendering of large data that would be difficult for object-order volume rendering on single-GPU systems.

4 Multiresolution Octree Volume Construction

An octree volume is an hierarchically compressed scalar field. Scalar values are stored at leaf nodes. At maximum octree depth, these correspond to the finest available data resolution. Scalars at less than maximum depth store coarser resolutions, by factors of 8 per depth level. Interior nodes maintain pointers from parents to children. In our multiresolution LOD application, they also contain coarser-resolution representations of each of their children.

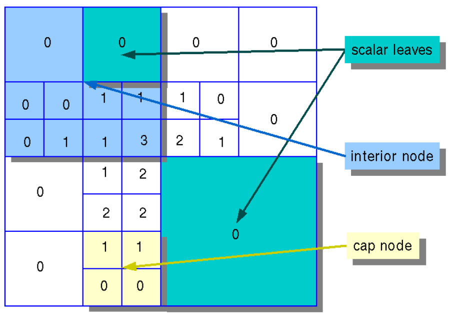


Fig. 2 *Octree volume format illustrated,* showing examples of an interior node, a cap node, and scalar leaves.

4.1 Construction Algorithm

Volume data can be natively computed and stored in the adaptive octree format. Alternately, the octree can be built from a scalar field in a 3D array. Such a construction is detailed by Knoll et al. [13]; this paper only discusses extensions to the construction technique that allow for multiresolution. In brief, construction is a bottom-up procedure in which identical or similar voxels are merged together into a single voxel within a parent node. Voxels are logically leaves of the octree. However, rather than store each voxel in a separate memory structure, we store every voxel within its immediate parent. This yields two distinct structures: *cap* nodes consisting of eight voxels at the finest resolution; and *interior* nodes consisting of pointers to other nodes, which can optionally be single *scalar leaf* voxels of a coarser resolution. As shown in Fig. 2, A scalar leaf is not a separate structure, but a value embedded inside its parent interior node. Similarly, cap nodes are not leaves themselves but contain eight scalars at the maximal depth of the octree.

Extension to multiresolution

In multiresolution octree volume construction, coarser-resolution consolidated voxels are *always* computed and stored in interior nodes, regardless of whether or not they are leaves. Theoretically, a static-resolution octree volume could use a single array to contain *either* a pointer to a child subtree or a coarser-resolution scalar leaf. In practice however, the memory savings of this approach were too small to justify the added computation. Multiresolution octree volumes are thus constructed exactly as in the static-resolution implementation [13]: nodes store eight-value arrays for child pointers and scalar leaves. The only difference is that multiresolution rendering actually uses non-leaf scalar data.

Min/Max tree computation

The only significant difference between multiresolution and static-resolution construction lies in computing the min/max tree. Static-resolution data requires the min/max pair of a given voxel to reflect the minimum and maximum of eight scalar vertices constituting the cell that maps to this voxel (Fig. 3). We do not store a min/max pair for each finest-level voxel due to the prohibitive 3x footprint. Instead, we compute them for the immediate parents of the finest voxels (cap nodes in Fig. 2). As shown in Fig. 4 (top), each leaf node must compute the minimum and maximum of its cell, hence account for the values of neighbors in the positive X and Y dimensions (left). This yields a min/max pair for the leaf node (right). Neighbors can potentially exist at different depths of the octree, as is the case for at the blue leaf node.. For multiresolution data, cells may have any power-of-two width, and we accordingly consider forward-neighbors at each depth of the min/max tree (Fig. 4, bottom). As a result, the min/max tree for a multiresolution octree volume is looser than that of static-resolution data. In practice, the impact on performance is negligible for the data we test.

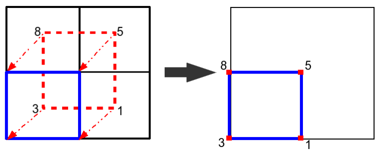


Fig. 3 *Voxel-cell mapping.* Given a scalar-centered voxel, we construct its dual *cell* by mapping the scalar to the lower-most vertex, and assigning forward-neighboring scalars to the remaining vertices.

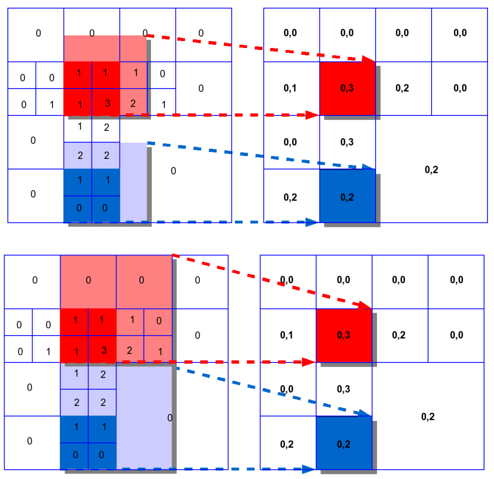


Fig. 4 *Min/max tree construction from forward neighbors.*

5 Coherent Octree Volume Ray Tracing

Having constructed a compact octree volume with an embedded min/max acceleration structure, we now turn to the task of building a coherent ray tracing system. In general, we seek to optimize for coherence as aggressively as possible, namely by implementing a vertical SSE packet architecture and a frustum-based octree traversal similar to the coherent grid traversal of Wald et al. [28].

5.1 SSE Packet Architecture

A coherent ray tracer achieves its performance by operating on groups of neighboring or similar rays in packets. To exploit coherence during primitive intersection, we perform computations on SIMD groups of four rays (frequently referred to as *packlets*) and mask differing hit results as necessary. Performing these SIMD computations requires that we store ray information vertically within a packet. For example, ray directions are stored as separate arrays of X,Y,Z components, as opposed to a single horizontal array of 3-vectors. These vertical arrays are 16-byte-aligned, permitting us to access a packlet of four rays at a time in a single SSE register. Similarly, the packet structure stores aligned SSE arrays of hit results, such as hit position and normals.

5.2 Coherent Traversal Background

As an efficiency structure for ray tracing, the octree affords several different styles of traversal. With coherent ray tracing, we are given the choice between depth-first traversal similar to a kd-tree [29] or BVH [26]; or a breadth-first coherent grid traversal (CGT) approach [28]. We choose the latter for several reasons. Our primitives are regular, non-overlapping cells, similar to large spherical particle data sets for which CGT has proven effective by Gribble et al. [8]. More significantly, the breadth-first nature of the CGT algorithm allows for a clever slice-based technique that amortizes voxel look-up from the octree when reconstructing the vertices of multiple cells.

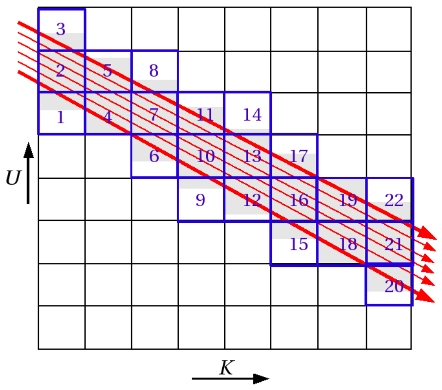


Fig. 5 *Coherent Grid Traversal.* The CGT algorithm [28] traverses a packet of rays through a grid slice by slice along a major march axis **K**, iteratively incrementing slice extents by the differential of the bounding frustum along the non-major axis **U** (and a third axis **V** in 3D).

Coherent Grid Traversal Algorithm

The original CGT algorithm departs from single-ray grid traversal in that it considers full slices of cells contained within a ray packet's bounding frustum, as opposed to marching across individual cells. The algorithm first determines the dominant X,Y,Z axis component of the first ray in each packet. This is denoted **K**, and the remaining axes are denoted **U** and **V**. Then, we consider the minimum and maximum u and v coordinates at the $k = 0$ slice, and note that the increment *du,dv* for a single unit along the march axis **K** is constant. We store this increment in a single SSE packed floating point unit, $d_{uv} = [du_{min}, dv_{min}, du_{max}, dv_{max}]$. Next, we determine the first and last k slice where the packet frustum intersects the volume. We begin at the u,v extents, $e_{uv} = [u_{min}, v_{min}, u_{max}, v_{max}]$, the minimum and maximum of enter and exit points on that slice of cells. To intersect primitives, we truncate these values to integers and iterate over all cells in that given **U**,**V** range. To march to the next slice, we add the constant increment. Thus, a non-hierarchical grid march is accomplished with a single SIMD addition and a SIMD float-to-integer truncation. Unlike a single-ray DDA grid algorithm [1], cells may be traversed in arbitrary **U**,**V** order; however the **K** order is invariably front to back, permitting early termination. The 2D analog of this algorithm is illustrated in Fig. 5.

Macrocell Hierarchical CGT

The original CGT paper [28] implemented a two-level hierarchy, with a single layer of macrocells each corresponding to 6 grid cells. For small polygonal data, this was generally sufficient. As the smallest volume we test is 302^3, a more robust hierarchy could be desirable for our application. We extended the CGT algorithm to arbitrary number of macrocell layers similarly to Parker et al. [22], and found that a recursive 2^3 macrocell hierarchy – equivalent to a full octree – consistently yielded the best performance for volumes larger than 256^3. The macrocell traversal employs an array stack structure to avoid recursive function calls: this stores the u,v slice and increment for all macrocell levels, the current slice within the current macrocell level, and the next slice at which to return to parent macrocell traversal. When all rays in a packet have intersected or the packet exits the root macrocell level, traversal terminates. The approach is that of a recursive grid sharing common coordinate space on the given volume dimensions, in which each macrocell block is a multiple M of its children. Thus, child coordinates are always an M-multiple of parent macrocell coordinates. Child macrocells, or the volume cells themselves, are traversed when *any* macrocell in a given slice is non-empty – specifically, when our desired isovalue is within that macrocell's min, max range. Then, the packet frustum traverses full slices of that macrocell level's children. As shown in Fig. 6, our hierarchical grid employs recursively superimposed macrocell blocks, with each parent containing 2^3 children, for alignment with the octree volume. We depict a 3-deep hierarchy, with blue, yellow and green extents corresponding to macrocell layers from coarsest to finest.

Macrocells are only traversed when they contain our desired isovalue, as illustrated by the “surface” at the dotted line. With an octree, macrocells are implicit; min/max pairs are retrieved from the octree nodes via hashing.

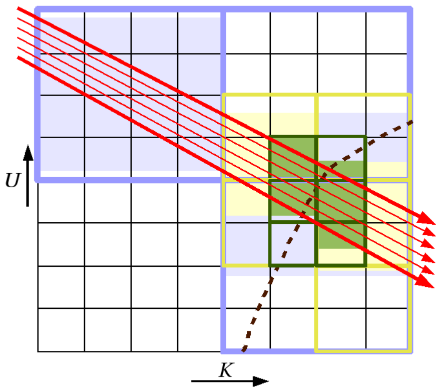


Fig. 6 *Coherent Octree Traversal via Implicit Macrocells.*

5.3 Implicit Macrocell Grid Traversal of Octree Volumes.

Our octree volume traversal is effectively coherent grid traversal of an implicit macrocell hierarchy, in which min/max pairs are retrieved from octree interior nodes instead of macrocells. Rather than repeatedly multiplying grid coordinates by the macrocell width M, octree nodes at all depths share a common coordinate space $[0, 2^{d_{max}}]$, where d_{max} is the maximum depth of the tree. Some macrocell traversal computation can be optimized for the binary subdivision of the octree. When recursing from a parent to traversing children, the macrocell grid multiplies the k-slice by the macrocell width M; in the octree $M = 2$, a bitwise left-shift. Computing the next macrocell slice requires a simple $+2$ addition.

Mapping Macrocells to Octree Nodes

Traversing implicit macrocells over an octree requires particular attention, as a single coarse scalar leaf node in the octree may may cover multiple finer-level implicit macrocells. Given an implicit macrocell coordinate, we seek the deepest octree child that maps to it. We then use the min/max pair in the parent node, corresponding to that child, to perform the isovalue culling test. As lookup is costly, we store the path from the octree root to the current node along the u,v-minimal ray of the frustum. We then use neighbor-finding as detailed in [13] to inexpensively traverse from one node to the next. Hierarchically recursing from a parent node to a child requires a single lookup step in the octree.

Default Slice-Based Traversal

At shallow levels of the octree, the packet frustum typically traverses a single common macrocell. At deeper levels, the u,v extents encompass multiple macrocells, so we must neighbor-find numerous octree nodes. By default, macrocell CGT stops iterating over a slice when *any* node is non-empty, and proceeds to traverse slices of children nodes. This ensures that traversal is performed purely based on the packet frustum as opposed to individual rays, and preserves the breadth-first coherent nature of the algorithm. Unchecked, it also causes numerous unnecessary octree lookups and ray-cell intersection tests. To mitigate this, we implement the two following optimizations.

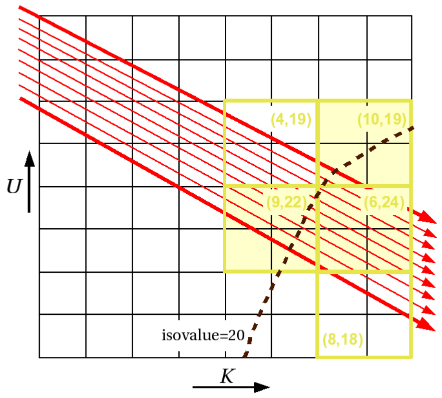


Fig. 7 *Culling empty macrocells from cap-node slices.*

Culling empty cap-level macrocells. To avoid unnecessary intersections and octree hashing, we clip the u,v slice corresponding to the deepest-level macrocells, one level above actual cell primitives. To do this, we iterate over the min/max pairs corresponding to the finest *available* octree depth. When traversing at maximum resolution, the deepest macrocells correspond to cap nodes (Fig. 2). Within this iteration, if a macrocell contains our isovalue, we compute new slice extents based on the minimum and maximum u,v coordinates. If the macrocell is empty, we omit it from extent computation. The effect is to clamp the u,v slice so that it more tightly encloses nodes with the desired isovalue. Fig. 7 illustrates this where we first clip slices of deepest macrocells, corresponding to cap nodes of the octree at depth $d_{max} - 1$. We narrow the u,v slice extents by omitting macrocells with ranges outside our value; only the shaded cells containing our isovalue are considered.

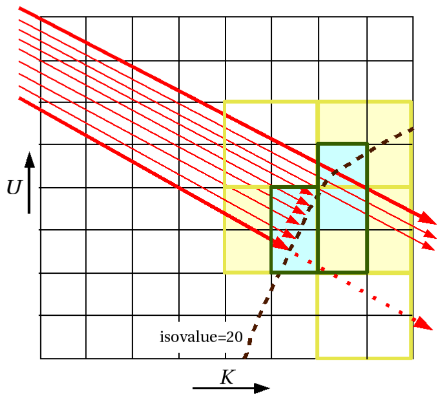


Fig. 8 *Clipping cell slices to fit active rays.*

Clipping the cell-level slice to active rays. To further reduce the number of cell primitives in a slice, we intersect individual rays with the world-space bounding box formed by the current u,v slice. When rays have already successfully hit a cell, they are "inactive" and can be safely ignored even if they intersect the slice bounding box. As shown in Fig. 8, this enables us to considerably shrink the *u,v* extents before intersecting a **K**-slice of cells by simply by computing the minimum and maximum of the enter and exit hit coordinates of active rays.

5.4 Cell Reconstruction from Cached Voxel Slices

Having clipped the primitive-level slice to as small a *u,v* extent as possible, we are ready to perform ray-cell intersection. Our ray-tracing primitive is a cell with eight scalar values; one at each vertex. However, the data primitives in our octree volume are voxels. Using the same duality employed by min/max tree construction, we map octree voxels to the lower-most vertex of each cell (Fig. 3). Our task now is to reconstruct cells efficiently from the octree, exploiting coherence whenever possible.

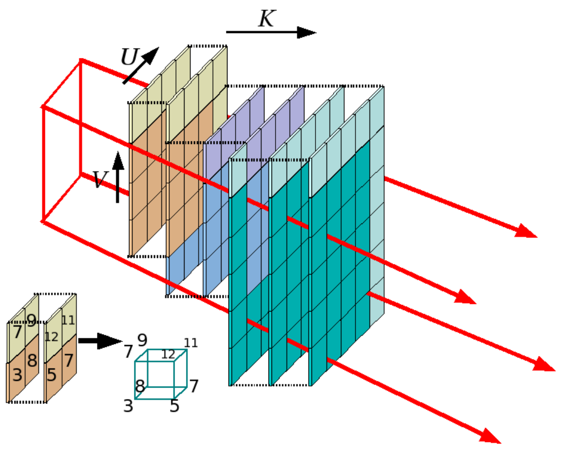


Fig. 9 *Slice-based cell reconstruction algorithm.*

U,V Voxel Slice Filling

In single-ray and depth-first traversals, cells are constructed independently, given a lower-most voxel from traversal, and using neighbor-finding to look up the remaining seven voxels. However, adjacent cells share vertices – much neighbor-finding effort is duplicated. With our octree CGT, we can iterate over an entire slice of adjacent *u,v* cells, access each voxel once, and store the results in a 2D array buffer. We add 1 to the maximal *u,v* slice extent to account for forward cell vertices in those directions. Then, we iterate over the *u* and *v* components of the slice, performing neighbor-finding from one coordinate to the next. By iterating in a scanline, the neighbor-finding algorithm need only find a common ancestor along one axis, and is slightly cheaper. We store the voxel results for this slice in a 2D array buffer, and look up values from this buffer to reconstruct four vertices of each cell in the slice. The remaining four vertices can be reconstructed in the same fashion by filling in a second buffer for the *k+1* slice. Thus, to find the eight vertices of each cell, rather than neighbor-find seven forward-neighbors per voxel, we exploit our slice-based traversal to look up and cache **K**-slices of voxels, amortizing and reducing the cost of data access. Fig. 9 illustrates filling of five successive slices, with like colors representing where cached voxels are used to avoid repeat neighbor-finding.

Copying the Previous-Step ***K****-Slice*

In cell reconstruction, we also exploit voxel coherence along the **K** axis. For this, we note that vertices on either the front (*k*) or back (*k+1*) slice of each cell are shared from one traversal step to the next, depending on whether the **K** march direction is positive or negative. In either case, we can copy an advancing slice buffer from the previous traversal step into a posterior buffer of the current traversal step (Fig. 9). We must account for the traversal offset in the minimum *u,v* coordinates between the two buffers; and perform neighbor-finding for voxels not buffered from the previous step, due either to that offset or different maximal *u,v* extents.

5.5 Ray-Cell Intersection

With our cached slice buffers, we can iterate over cell primitives and reconstruct cell vertices. To compute the ray-isosurface intersection, we iterate over all SIMD packlets, discarding packlets that are inactive (have already intersected) according to the per-packlet hit mask. For each packlet, we first check that each at least one actually intersects the bounding box of the cell in question, and then proceed to compute the ray intersection with the implicit isosurface.

For ray-cell intersection, we seek a surface inside a three-dimensional cell with given corner values (Fig. 3), such that trilinear interpolation of the corners yields our desired isovalue. This entails solving a cubic polynomial for each ray; the hit position is given at the first positive root. Our implementation uses the Neubauer iterative root finder proposed by Marmitt et al. [21]. Computation is performed per-packlet. If any ray in the packet intersects successfully, we compute the gradient normals for that packlet. We do not defer normal computation due to the prohibitive cost of reconstructing cell vertices twice.

6 Multiresolution Level of Detail System

Our optimized coherent traversal algorithm significantly outperforms single-ray traversal on simple scenes; and due to the lower data lookup cost even exhibits a factor-of-two speedup moderately incoherent scenes in which more than one ray per packlet seldom intersects the same cell (Table 2). However, coherence breaks down on highly complex scenes, where rays are separated by multiple cells that are never intersected. This pathological case is common with far views of large data sets. (Table 3). This behavior is detailed more fully in Section 8. The purpose of the multiresolution system is to manage pathological cases posed by large data, and preserve coherence with only minor sacrifice in quality.

6.1 Resolution Heuristic

Stop depth. The general vehicle for the multiresolution scheme is determining an effective depth at which to stop traversing children, and instead reconstruct cells to intersect. Coarser-resolution voxels are explicitly stored in the scalar leaf fields of interior nodes, regardless of whether a finer-resolution subtree exists. When the traversal algorithm stops, cell reconstruction proceeds exactly as it would at the finest resolution, except given a stop depth d_{stop} it increments the u,v coordinates by $2^{d_{stop}}$ instead of simply 1 at the finest resolution. Thus, the octree hash scheme operates on canonical octree space $[0, 2^{d_{max}}]$, regardless of LOD depth.

Pixel-to-voxel width ratio. A more difficult problem in formulating the multiresolution scheme is determining which parts of the scene should be rendered at which resolution. Generally, we note that when multiple voxels project to the same pixel, a coarser level of resolution is desirable. LOD techniques for volume rendering often use a view-dependent heuristic to perform some projection of voxels to screen-space pixels, and identify distinct regions of differing resolutions [15]. In the case of ray-casting with a pinhole camera, the number of voxels that project to one pixel varies quadratically with the distance from the camera. As aspect ratio is constant, we may simply consider the linear relation along one axis **U**, namely the increment between each primary ray along **U**, du. Then, we can render the coarser resolution at d_{stop} when $du = Q_{stop} * dV$, where dV is the **U**-width of a voxel, and Q_{stop} is some constant threshold. As the **U**-width of a single pixel, dP, is simply a multiple of du, we can simply reformulate our constant as a ratio of pixel width to voxel width dP/dV, where $Q_{stop} = (du/dP) * (dP/dV)$.

Packet extents metric. Ideally, our LOD metric should be evaluated per packet. An obvious choice would be the du width of the packet, given by the aforementioned u,v slice extents. One could render a coarser resolution whenever the number of cells in a slice at the current resolution surpassed some threshold. Unfortunately, at the same k-slice, the du_{packet} could vary between packets, causing neighboring rays to intersect different-resolution cells, hence resulting in seams. We desire a similar scheme that allows us to perform transitions consistently between packets.

LOD Mapping via **K** *Transition Slices.* To ensure consistent transitions from one resolution to the next, we compute a view-dependent map from resolution levels to world-space regions along the major traversal axis **K**. We note that the width of a pixel corresponds to the distance between primary rays along the **U** and **V** axes, which increases with greater t, as we move farther from the camera origin. If we consider a major march direction **K**, we can find the exact k slice coordinate where any given number of voxels corresponds to exactly one pixel. This is similar to the per-ray metric approach, except it solves where $du = Q_{stop} * dV$ at a discrete **K**-slice, k. As packets traverse the octree one **K**-slice at a time, we have a constant world-space LOD function that can be computed on a per-packet basis.

We multiply the ratio of pixel width to voxel width, dP/dV, by the power-of-two unit width corresponding to each depth d of the octree. Then, we solve for the t parameter where this voxel width is equal to the distance between viewing rays, du_{camera}. Finally, we evaluate **K**-component of the direction ray to compute the **K**-slice where our fixed dP/dV ratio occurs, $k_{transition}[d]$. These mark the transition slices from each resolution to its coarser parent. The array is computed once per frame, using Algorithm 1. The dP/dV constant is thus our base quality metric; Fig. 13 shows the same scene rendered using varying dP/dV.

Algorithm 1 Transition Array Computation

```
Require: Pixel-width to voxel-width ratio, $dP/dV$
  Per-ray camera offset along U axis, $du_{camera}$
Ensure: Array of K-transition slices, $k_{transition}[]$
  for all octree depths $d \in \{0..d_{max}-1\}$ do
    $voxelWidth[d] \Leftarrow 2^{d_{max}-d} * dP/dV$
    $t_{transition}[d] \Leftarrow voxelWidth[d] \ / \ du_{camera}$
    $k_{transition}[d] \Leftarrow k_{origin} + t_{transition} k_{direction}$
  end for
```

6.2 Multiresolution Traversal

Rather than determining the major march axis **K** per packet, we decide it once per frame based on the direction vector of the camera. While this causes some packets to perform CGT on a non-dominant axis, in practice there is no appreciable loss in performance with a typical 60-degree field of view.

The traversal algorithm determines the initial transition slice when it computes the first k-slice of a packet, by finding the first $k_{transition}[d] < k$. Then, before recursively traversing a child slice at the current resolution depth, we check if $k_{child} >= k_{d-1}$, the slice corresponding to transition to the *next* coarser resolution. When that occurs, we omit traversal of the child and perform cell reconstruction. The current resolution depth is then decremented, so the traverser seeks the subsequent coarser-resolution transition slice. This process is illustrated in Fig. 10 (left).

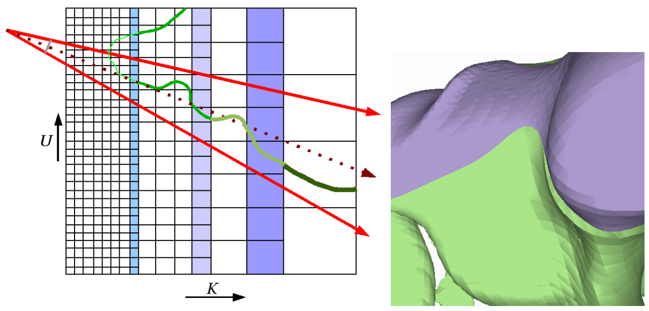


Fig. 10 **Left:** multiresolution transition slices along the **K** axis. **Right:** transitions are smoothed by substituting coarse-LOD values at fine-level cell vertices at the transition slice.

6.3 Smooth Transitions

Isosurfaces are piecewise patches over their respective cells, and can vary both topologically and locally from one resolution to the next. As such, discontinuities arise at transition slices between finer and coarser isosurfaces. While these discontinuous surfaces are technically "correct" with respect to each resolution, it is frequently desirable to mask the multiresolution transition and render a single smooth surface. To accomplish this, our slice-based reconstruction algorithm checks if each **K**-slice is equal to the next k_d transition slice. If it is, we look up voxel data from the octree at coarser depth $d-1$ as opposed to the current default depth d. This guarantees identical voxel values on either side of the transition, and thus continuous surfaces (Fig. 10, right). Exceptions may occur in cases of gross disparity between each resolution of the scalar field, where topological differences cause a surface to *exist* at one resolution but not the other This is common in highly entropic regions of the Richtmyer-Meshkov data. In these cases, it is desirable to omit smooth transitions and expose levels of detail via color-coding (Figs. 13, 16).

7 Shading

Our technique affords better flexibility in shading the isosurface. One limitation of the octree volume is that data access for cell reconstruction is expensive, discouraging techniques such as central-differences gradients that require additional neighbor-finding. With slice-based coherent traversal, we are able to amortize the cost of cell reconstruction as shown previously. Multiresolution allows us to simplify the casting of shadow rays and illustrate depth cues with less performance sacrifice.

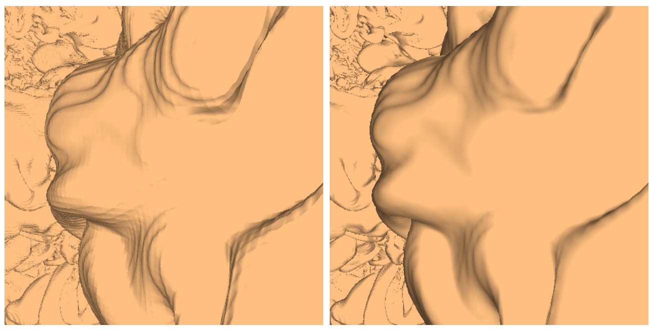


Fig. 11 *Gradient normals,* computed on a forward differences stencil yielding 5.5 FPS (**left**), and a central differences stencil at 4.7 FPS (**right**) on an Intel Core Duo 2.16 GHz with a 512^2 frame buffer.

7.1 Smooth Gradient Normals

By default, normals are computed using the forward-differences gradient at the intersection point within the given cell. The disadvantage of this method is that such gradients are continuous only within each cell. The isosurface itself is formed from piecewise trilinear patches with C_0 continuity at cell edges. For a more continuous normal vector field, and better visual quality, we can compute gradients on a central differences stencil to ensure C_1 continuity along cell edges.

To compute the central differences gradient, we use a stencil of three cells along each axis; thus 64 cell vertices (voxels) must be found during reconstruction. Reconstructing a 4^3 voxel neighborhood per-ray is costly in non-coherent octree volume isosurface ray tracing [13]. Coherent reconstruction with cached slices allows for smooth normals with far lesser penalty. In a non-coherent ray tracer this entails eight times the lookup cost of forward differences, causing worse than half the forward-differences performance. In our coherent system, we return to the slice-based cell reconstruction technique to amortize that cost of neighbor-finding. We simply retrieve two additional rows and columns of voxels, corresponding to $u_{min}-1, v_{min}-1$ and $u_{min}+2, v_{min}+2$ coordinates. In addition to our existing 2D array buffers for the k and $k+1$ slices, we store two additional buffers corresponding to the $k-1$ and $k+2$ slices. We then use this four-wide kernel with a central-differences stencil to compute the gradient: $\frac{1}{2}(V_{X-1,Y,Z}) - V_{(X+1,Y,Z)})$ along the X axis, and similarly for the Y and Z axes. Performance with central differences is typically 15%-30% slower than with forward differences. Given the improvement in visual quality, smooth normals are arguably worth the trade (Fig. 11).

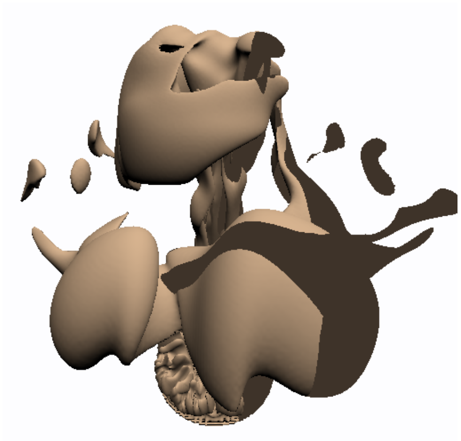


Fig. 12 *Shadows.* With centrally-differenced gradient normals, the above shadowed scene renders at 3.9 FPS on an Intel Core Duo 2.16 GHz with a 512^2 framebuffer, as opposed to 5.1 FPS without shadows.

7.2 Shadows

An oft-cited advantage of ray tracing is that shadows can be computed trivially without adding geometric complexity or implementing sophisticated multi-pass texturing techniques. In practice, tracing shadows doubles the cost of casting each ray that successfully hits an object. Computing shadow rays in a coherent packet system is more complicated than for a single-ray tracer, as individual rays must be masked and shadow packets generated based on the hit results of the primary rays. Fortunately, point-light shadows may be cast from the light to the primary hit point, thus they share a common origin and benefit from coherent optimizations. Our primary goal being interactivity, we are interested in hard shadows that may not appear photorealistic, but adequately provide depth cues to the viewer. As such, we can exploit the LOD system to cast faster coherent shadow rays through a coarser-resolution representation of our volume – for example, using a shadow ray *dP/dV* of twice the viewing ray *dP/dV*. By coherently casting shadow rays through a coarser resolution, we can achieve higher performance and provide similar depth cues. This yields framerates only 20%-30% slower with shadows than without (Fig. 12).

8 Results

We first note the impact of octree volumes on compression and render-time memory footprint. We then evaluate performance of our system by first considering coherent octree traversal alone, and then analyzing the performance of the multiresolution system.

8.1 Octree Construction Results

Octree volumes are remarkable not in the overall compression ratios they achieve, but in their ability to provide respectable lossless compression, spatial hashing, and effective ray traversal in a single structure. Table 1 shows compression achieved for various structured data. Generally, a factor of 4:1 is common with lossless consolidation, but actual compression depends enormously on the overall entropy of the volume. Fluid dynamics simulations such as the Richtmyer-Meshkov and heptane compress well, but noisy medical data can actually occupy more space in an octree. Segmentation allows us to meet memory constraints, and isolate data ranges of interest.

DATA	ISO-RANGE	TIME STEP	SIZE original	SIZE octree	%
heptane	full	70	27.5M	3.96M	14
	full	152	27.5M	9.5M	33
	full	0-152	4.11G	678M	16
RM	full	50	8.0G	687M	8.5
	full	150	8.0G	1.89G	25
	full	270	8.0G	2.48G	30
	64-127	270	8.0G	1.81G	22
CThead	full		14.8M	12.4M	84
femur	full		162M	163M	101
	100-163		162M	9.0M	5.5

Table 1 *Compression achieved for various structured data* when converted to octree volumes. The second column represents iso-ranges. Clamping all values outside a given range delivers additional octree compression, and preserves lossless compression for values within that range. "Full" indicates the full 0-255 range for 8-bit quantized scalars. Data sizes are in bytes, and include all features of the octree, including overhead of the embedded min/max tree.

Further Compression

Generally, our goal is simply to compress a single data timestep into a manageable footprint for limited main memory. Sometimes losslessly compressed data will be slightly too large to meet this constraint. One option is lossy compression via a non-zero variance threshold, which behaves similarly to quantization. A more attractive method, for our purposes, is segmenting data into interesting ranges of isovalues, and clamping scalars outside those values to the minimum and maximum of the range. This allows for lossless-quality rendering of isovalues within that range. For example, compressing only the 64-127 value range of timestep 270 of the Richtmyer-Meshkov data allows us to render that range on a machine with 2 GB RAM (Table 1). This method is even better suited for medical data such as the visible female femur, when the user is specifically interested in bone or skin ranges. The full original CT scan has highly-variant, homogeneous data for soft tissue isovalues from 0-100, causing the octree volume to actually exceed the original data in footprint. However, considering only the bone isovalues 100-163, we achieve nearly 20:1 compression (Table 1). Not coincidentally, such "solid" data segments are best suited for visualization via isosurfacing (Fig. 15).

Construction Performance and Filtering

The bottom-up octree build algorithm is $O(N)$ with regard to the total number of voxels; nonetheless N can be quite large. On a single core of a 16-core 2.4 GHz Opteron workstation, building a single timestep of the 302^3 heptane volume requires a mere 8 seconds and negligable memory footprint; whereas a timestep of the Richtmyer-Meshkov data requires 45 minutes and a footprint of nearly 40 GB. The build itself creates an expanded full octree structure that occupies a footprint of four times the raw volume size. Thus, building octree volumes from large data requires a 64-bit workstation. Although an offline process, parallelizing and optimizing the build would be both desirable and feasible as future work. In addition, the current construction algorithm effectively samples coarser resolutions via recursive clustered averaging. Superior LOD quality could be achieved with bilinear or higher-order filtering.

Memory Footprint Comparison

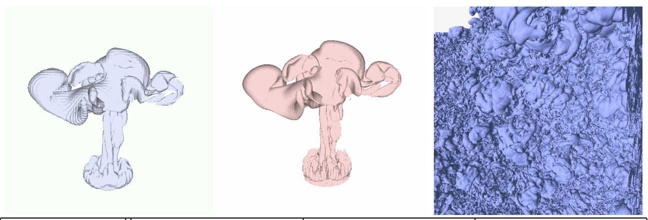
Octrees generally occupy 20%-30% the memory footprint of the uncompressed grid data, including both the multiresolution LOD structure and min/max acceleration tree. Conversely, storing a full 3D array for each power-of-two LOD volume would approach twice the footprint of the original uncompressed volume. Other ray-tracing efficiency structures such as implicit kd-trees [27] could require up to twice the full data footprint, often with an additional overhead of around 15% for cache-efficient bricking [22]. Thus, octrees compare favorably to other volume ray tracing structures.

8.2 Coherent Traversal Analysis

The main purpose of our slice-based algorithmic enhancements, and indeed of traversal itself, is to minimize the number of cells that must be intersected. By employing packets and the breadth-first CGT frustum algorithm, we are able to dramatically reduce both the computational and memory access costs of traversal. Finally, when multiple rays in a SSE packlet intersect the same object, we may effectively perform up to four intersections for the price of one. For these reasons, we are able to achieve significant speedups on highly coherent simple scenes. Even with moderately complex scenes where a pixel seldom contains more than one voxel, and SIMD intersection yields little speedup, slice-based reconstruction effectively doubles performance (Table 2). Moreover, rendering time is strongly correlated with the number of ray-cell intersections. Performance profiling reveals that only 5%-15% of CPU time is spent in traversal, compared to over 70% in reconstruction and intersection.

TRAV.	LOOKUPS	ISECS	L/RAY	I/RAY	FPS
single	314707	166719	1.2	0.64	2.3
packet	1187798	469560	4.5	1.8	.78
+*slice*	1187798	469560	4.5	1.8	2.2
+*mcell*	561889	124221	2.14	0.47	3.9
+*cell*	270123	120514	1.0	0.47	4.6
+*mulres*	98055	44419	0.37	0.17	7.6

Table 2 *Results from clipping optimizations* when ray-casting a moderately complex scene with low primitive-level coherence, from the heptane fire dataset (HEP302, Table 3). We compare single-ray traversal and 8x8 octree-CGT packet traversal with and without optimizations. +*slice:* use slice-based cell reconstruction. +*mcell:* clip the deepest macrocell slice extents to discard nodes not containing the isovalue. +*cell:* clip the cell slice extents to the set of active rays. +*mulres:* multiresolution scheme, with $dP/dV = 1$. Tests at 512^2 using one core of an Intel Core Duo 2.16 GHz.



SCENE	HEP64		HEP302		RM	
	I/RAY	FPS	I/RAY	FPS	I/RAY	FPS
single	0.70	1.9	0.64	2.3	3.58	0.57
coherent						
2x2	0.26	4.3	0.5	2.84	5.65	0.38
4x4	0.11	9.7	0.45	4.54	7.94	0.33
8x8	0.058	14.4	0.47	4.6	12.4	0.22
16x16	0.041	14.5	0.50	2.81	20.5	0.08

Table 3 *Results with coherent packets*, showing the net number of intersections per ray and frames per second with a single-ray tracer, and our coherent system with varying packet sizes. We examine three scenes of increasing complexity. Leftmost (HEP64) is the 64^3 down-sampled heptane data, which has high intersection-level coherence. The full 302^3 heptane data (HEP302) has low intersection-level coherence, but benefits from coherent traversal. The 2048^3 RM data yields even less coherence, and is a pathological case for packet traversal. Benchmarks on a single core of an Intel Core Duo 2.16 GHz, with a 512^2 frame buffer and multiresolution disabled.

Packet size

For performance reasons, our implementation chooses a static packet size for traversal. This is appropriate for our application, as we seek to render isosurfaces with constant complexity. Later, we enforce this via the pixel to voxel width ratio in the LOD scheme. Empirically, we find that packets of 8x8 work best for scenes where one to 4 rays intersect a common cell. 16x16 packets yield little benefit even for simple data, and perform poorly on complex scenes of large data (Table 3).

Incoherent behavior without multiresolution

Complex scenes reveal the shortcoming of coherent traversal. Because traversal is not computed on a per-ray basis, but solely from the packet frustum corners, it frequently looks up cells that would have been correctly ignored by a more expensive single-ray traverser. Our clipping optimizations (Figs. 7, 8) noticeably alleviate this, as we can see in Table 2. However, for complex scenes such as far views of large data, rendering cost is totally bound by intersection (Table 3). Ultimately, frustum-based traversal causes large numbers of cells to be looked up, though no rays in the packet actually intersect them. This in turn causes many unnecessary intersection tests to be performed. Successful intersection tests are no less expensive, as packlet-cell intersection degenerates to single-ray performance without primitive-level coherence. These higher costs eventually overwhelm any gains made by more efficient traversal, and cause the coherent ray tracer, without multiresolution, to perform worse than a single-ray algorithm on sufficiently complex scenes.

8.3 Multiresolution Results

The combination of multiresolution level of detail and coherence enables frame rates up to an order of magnitude faster for coherent scenes. With large volume data and small frame buffers, coherence is less common; but in general it is possible to decrease *dP/dV* to achieve interactive frame rates and interesting, albeit coarser-quality, representations of the data. For highly entropic large volume data, this behavior is frequently useful as coarser LODs inherently possess less variance, thus manifest less aliasing. However, coarser LOD rendering are also less correct with respect to the original resolution, as shown in Fig. 13.

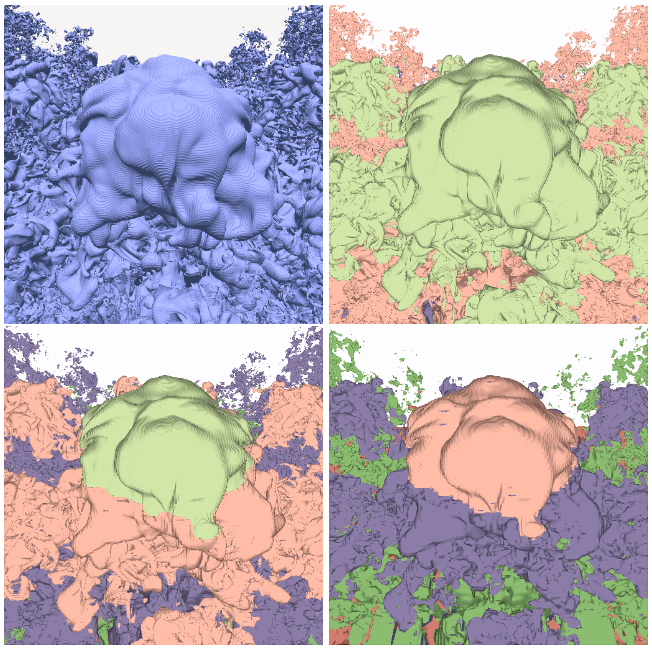


Fig. 13 *Qualitative impact of multiresolution* on the Richtmyer-Meshkov data at t=270, isovalue 20. Top left to bottom right: single-ray, then coherent multiresolution with dP/dV of 1,2 and 4. On an Intel Core Duo 2.16 GHz with a 512^2 frame buffer, these render at 0.92, 1.0, 1.9, and 3.6 FPS respectively. To illustrate LOD transitions, like colors indicate the same resolution.

Quality Comparison

Fig. 16 in the Appendix compares quality with and without multiresolution at $dP/dV = 1$, with forward and central differences gradients. The bottom images show per-pixel differences (computed using $1 - \mathrm{abs}(reference - image)$ per color channel), comparing multiresolution and non-multiresolution results, both rendered as white surfaces on black background. In reproducing major features, renderings with and without multiresolution look essentially identical. However, the difference images reveal that though most features remain intact, actual isosurfaces are slightly offset when rendered at varying resolution – this accounts for the black pixels (shown in closeup) where the surface exists at one resolution but not another. Otherwise, most differences lie in the intensity of the gradient at different resolutions, as evidenced by grayscale pixels in the difference images.

Overall Performance

In best-case scenarios, our system significantly outperforms the single-ray tracer. With close camera views of the RM data and $dP/dV = 1$, we see order-of-magnitude improvement (Table 4). The coherent technique usually yields modest improvements even for scenes with generally poor coherence. For sufficiently far camera angles viewing complex data, the single-ray system may actually outperform the coherent method, when using a LOD $dP/dV = 1$. For these pathological cases, we recommend relaxing dP/dV for exploration, or resorting to single-ray traversal for quality.

Coherent traversal handles a difficult scenario for the single-ray system: a close-up scene deep within the volume, with an isovalue for which the min/max tree is particularly loose. Such is the case in the last example of Table 4. While single-ray suffers from data access demand, coherent traversal largely amortizes these costs and performs comparably to other scenes with similar complexity.

Another substantial advantage of coherence is that large frame buffers can be rendered relatively faster. Doubling the frame buffer dimensions generally causes a factor of four slowdown in a single-ray tracer; by comparison the packet system frequently experiences a factor of two or better performance decrease, particularly when higher resolution leads to improved intersection-level coherence. For the dataset in Fig. 15, coherent ray tracing scales well to large frame buffers. This dataset renders at 6.0 FPS at 512^2, versus 3.1 FPS at 1024^2 on an Intel Core Duo 2.16 GHz, with central differences and shadows.

8.4 Comparison to Existing Systems

Table 4 shows performance for the Richtmyer Meshkov dataset with our coherent multiresolution system with $dP/dV = 1$; and the single-ray implementation [13] with no multiresolution scheme. In the best-case scenario we achieve a factor of 23 faster than single-ray performance, and even in worst cases the coherent multiresolution implementation does not exhibit substantially inferior performance. These numbers compare favorably to other implementatations.

SCENE	C.Duo,512^2		Xeon,512^2		Xeon,1024^2	
	single	8x8	single	8x8	single	8x8
50, far	2.5	3.5	14.5	19.5	4.5	6.5
150, far	1.9	2.5	11.1	15.0	3.4	4.9
270, far	1.1	1.1	7.2	12.4	2.2	2.2
50, close	2.0	6.9	12.1	39.8	3.6	14.2
150, close	1.7	8.1	12.0	44.2	3.5	14.6
270, close	0.2	4.7	1.4	38.6	0.4	9.2

Table 4 *Framerates of various time steps of the Richtmyer-Meshkov data,* on an 2-core Intel Core Duo 2.16 GHz laptop (2 GB RAM) and an 8-core dual 3 GHz Intel Xeon (Clovertown) with 4 GB RAM; with our coherent multiresolution method with 8x8 packets and $dP/dV = 1$, and single-ray without multiresolution [13]. Refer to Fig. 14 for images.

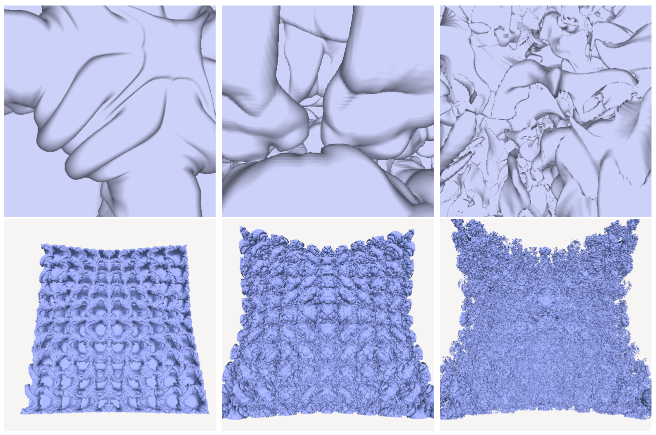


Fig. 14 *Richtmyer-Meshkov results.* From left to right, timesteps 50, 150 (isovalue 20), and 270 (isovalue 160). **Top:** various close-up camera views, illustrating highly coherent scenes. **Bottom:** far views exhibiting generally poor coherence. We use $dP/dV = 1$.

For similar camera positions, we achieve the same 2 FPS RM data performance on an two-core Intel Core Duo as DeMarle et al. [5] report on a 64-processor cluster with a distributed shared memory layer. We are competitive with Wald et al. [27] for far views, and perhaps faster for close-up scenes, while generally requiring an order of magnitude lesser memory footprint. The performance of our system is also on par with that of Friedrich et al. [7]; however such comparison is not completely fair as that system employs LOD for progressive as opposed to dynamic rendering.

Comparison with state-of-the-art GPU methods is more difficult. Clearly, slice-based direct-volume rasterization on the GPU outperforms our method by well over an order of magnitude for small data (less than 512^3). For larger data, this gap is less pronounced, but GPU DVR methods can equally employ multiresolution compression schemes on blocks [9] and space-skipping and culling optimizations [25]. These techniques are still limited by bus latency, and to our knowledge data the size of the Richtmyer-Meshkov has yet to be visualized at original data resolution on a GPU. Out-of-core streaming and progressive rendering, as well as multi-GPU distributed systems, are clearly valid approaches to large-scale volume visualization [4]. However, multicore workstations are increasingly inexpensive commodities, and share a more straightforward and scalable programming model. Ultimately in rendering large data, performance is bound more by memory access than by computation. To that end, multicore CPU's, with hierarchical caches that directly access expandable mainboard memory, are increasingly attractive. Ray tracing algorithms are well-suited for both this application and platform.

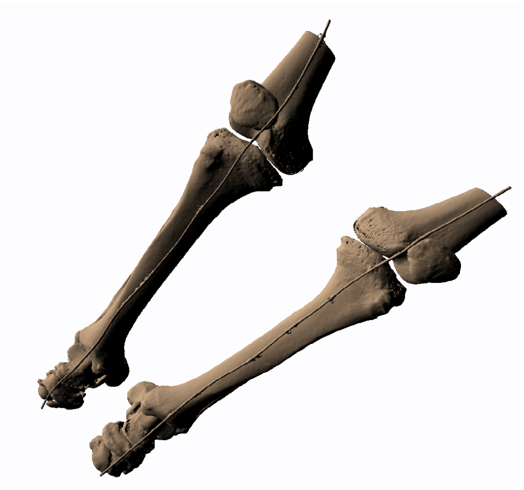


Fig. 15 *The visible female femur.*

9 Conclusions

We have presented a method for coherent ray tracing of large octree volume data using a multiresolution level of detail scheme to improve performance. Octree volume ray tracing allows for interactive exploration of large structured data on multicore computers using a fraction of the original memory footprint. While other spatial structures might deliver greater compression or faster traversal, the octree strikes a particularly good balance of these goals. With multiresolution and coherent traversal, we are able to trade quality for performance and render at interactive rates. Coherent traversal amortizes the cost of cell lookup, which allows for faster intersection and improved shading techniques.

As future work, the multiresolution octree could trivially be employed for out-of-core progressive rendering similar to Friedrich et al. [7], using the same compressed structure for LOD. Equally intriguing would be adapting the slice-caching reconstruction algorithm to perform volume rendering. Though computationally demanding, it could be implemented to take advantage of SIMD vector instructions [11], and would exhibit similar overall complexity to isosurfacing if the transfer function were sufficiently sparse. Also of interest would be employing generalized higher-order implicit surfaces [12] as intersection primitives, which could yield higher-quality reconstructions. Finally, generating coarser LOD's with improved filtering, as well as smoothly blending between LOD levels as opposed to only interpolating at transitions, could improve visual quality.

An overarching concern is that LOD may not be an ideal solution for high-quality rendering, and ultimately performance gains from improved coherence may not justify the increase in code complexity and loss in visual quality. One of the major advantages of ray tracing, when compared to rasterization, is that performance depends logarithmically, not linearly, on geometric complexity. The single-ray tracer renders both simple and complex data at roughly equal, though slow, frame rates. Coherent multiresolution essentially forfeits this advantage; it instead opts to improve best-case performance of simple scenes, while attempting to simplify complex scenes to mitigate worst-case performance. In a way, coherent ray tracing behaves similary to rasterization in that its performance depends on LOD.

Nonetheless, for the purposes of large volume visualization, multiresolution isosurface ray tracing presents clear benefits. The main goal of our optimizations was to overcome limitations single-ray octree volume ray tracing [13] and to ensure general interactivity. Overall, we accomplish that: our system is generally faster than single-ray non-coherent methods, allows for improved shading at reduced cost, and permits the user to trade visual quality for speed when interactivity is marginal. Moreover, as multicore CPU's increase in power and availibility, techniques such as these become increasingly interactive, while retaining their scalability to large data and more cores in the long term.

10 Acknowledgments

This work was supported by the U.S. Department of Energy through CSAFE grant W-7405-ENG-48, and by the National Science Foundation under CISE grants CRI-0513212, CCF-0541113, and SEII-0513212. It was also supported by the US Deparment of Energy SciDAC VACET, Contract No. DE-FC02-06ER25781 (SciDAC VACET), and a visiting professorship sponsored by Intel Corp. Thanks to Mark Duchaineau at Lawrence Livermore National Laboratory for use of the Richtmyer-Meshkov dataset, and to Steve Parker and Heiko Friedrich for their support and insights.

References

1. Amanatides, J., Woo, A.: A Fast Voxel Traversal Algorithm for Ray Tracing. In: Proceedings of Eurographics, pp. 3–10. Eurographics Association (1987)
2. Boada, I., Navazo, I., Scopigno, R.: Multiresolution Volume Visualization with a Texture-Based Octree. The Visual Computer **17**(3) (2001)
3. Cabral, B., Cam, N., Foran, J.: Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In: VVS '94: Proceedings of the 1994 symposium on Volume visualization, pp. 91–98. ACM Press, New York, NY, USA (1994). DOI http://doi.acm.org/10.1145/197938.197972
4. Castanie, L., Mion, C., Cavin, X., Levy, B.: Distributed shared memory for roaming large volumes. IEEE Transactions on Visualization and Computer Graphics **12**(5), 1299–1306 (2006). DOI http://doi.ieeecomputersociety.org/10.1109/TVCG.2006.135. Proc. IEEE Visualization 2006
5. DeMarle, D.E., Parker, S., Hartner, M., Gribble, C., Hansen, C.: Distributed Interactive Ray Tracing for Large Volume Visualization. In: Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG), pp. 87–94 (2003)
6. Djeu, P., Hunt, W., Wang, R., Elhassan, I., Stoll, G., Mark, W.R.: Razor: An architecture for dynamic multiresolution ray tracing. Tech. rep., The University of Texas at Austin (2007). (Cond. accepted to ACM Transactions on Graphics)

7. Friedrich, H., Wald, I., Slusallek, P.: Interactive Iso-Surface Ray Tracing of Massive Volumetric Data Sets. In: Proceedings of the 2007 Eurographics Symposium on Parallel Graphics and Visualization (2007)
8. Gribble, C., Ize, T., Kensler, A., Wald, I., Parker, S.G.: A coherent grid traversal approach to visualizing particle-based simulation data. Tech. Rep. UUSCI-2006-024, SCI Institute, University of Utah (conditionally accepted at ACM Transactions on Graphics, 2006) (2006)
9. Guthe, S., Wand, M., Gonser, J., Straßer, W.: Interactive Rendering of Large Volume Data Sets. In: Proceedings of the conference on Visualization '02, pp. 53–60. IEEE Computer Society (2002)
10. Igehy, H.: Tracing Ray Differentials. In: Computer Graphics (Proceedings of ACM SIGGRAPH), pp. 179–186 (1999)
11. Knittel, G.: The ULTRAVIS System. In: Proceedings of the 2000 IEEE symposium on Volume visualization, pp. 71–79. ACM Press (2000). DOI http://doi.acm.org/10.1145/353888.353901
12. Knoll, A., Hijazi, Y., Wald, I., Hansen, C., Hagen, H.: Interactive Ray Tracing of Arbitrary Implicit Functions with SIMD Interval Arithmetic. In: Proceedings of the 2007 Eurographics/IEEE Symposium on Interactive Ray Tracing (2007)
13. Knoll, A., Wald, I., Parker, S., Hansen, C.: Interactive Isosurface Ray Tracing of Large Octree Volumes. In: Proceedings of the IEEE Symposium on Interactive Ray Tracing (2006)
14. Kraus, M., Ertl, T.: Adaptive Texture Maps. Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware (2002)
15. LaMar, E., Hamann, B., Joy, K.I.: Multiresolution Techniques for Interactive Texture-based VolumeVisualization. In: Proceedings IEEE Visualization 1999 (1999)
16. Levoy, M.: Efficient Ray Tracing for Volume Data. ACM Transactions on Graphics **9**(3), 245–261 (1990)
17. Liu, Z., Finkelstein, A., Li, K.: Improving Progressive View-Dependent Isosurface Propagation. Computers & Graphics **26**(2), 209–218 (2002)
18. Livnat, Y., Hansen, C.D.: View Dependent Isosurface Extraction. In: Proceedings of IEEE Visualization '98, pp. 175–180. IEEE Computer Society (1998)
19. Livnat, Y., Tricoche, X.: Interactive Point-based Isosurface Extraction. In: Proceedings of IEEE Visualization 2004, pp. 457–464 (2004)
20. Lorensen, W.E., Cline, H.E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. Computer Graphics (Proceedings of ACM SIGGRAPH) **21**(4), 163–169 (1987)
21. Marmitt, G., Friedrich, H., Kleer, A., Wald, I., Slusallek, P.: Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In: Proceedings of Vision, Modeling, and Visualization (VMV), pp. 429–435 (2004)
22. Parker, S., Parker, M., Livnat, Y., Sloan, P.P., Hansen, C., Shirley, P.: Interactive Ray Tracing for Volume Visualization. IEEE Transactions on Computer Graphics and Visualization **5**(3), 238–250 (1999)
23. Parker, S., Shirley, P., Livnat, Y., Hansen, C., Sloan, P.P.: Interactive Ray Tracing for Isosurface Rendering. In: IEEE Visualization, pp. 233–238 (1998)
24. Reshetov, A., Soupikov, A., Hurley, J.: Multi-Level Ray Tracing Algorithm. ACM Transaction of Graphics **24**(3), 1176–1185 (2005). (Proceedings of ACM SIGGRAPH)
25. Ruijters, D., Vilanova, A.: Optimizing GPU Volume Rendering. Winter School of Computer Graphics, Pilzen (2006)
26. Wald, I., Boulos, S., Shirley, P.: Ray tracing deformable scenes using dynamic bounding volume hierarchices. Tech. Rep. UUSCI-2006-023, SCI Institute, University of Utah (conditionally accepted at ACM Transactions on Graphics, 2006) (2006)
27. Wald, I., Friedrich, H., Marmitt, G., Slusallek, P., Seidel, H.P.: Faster Isosurface Ray Tracing using Implicit KD-Trees. IEEE Transactions on Visualization and Computer Graphics **11**(5), 562–573 (2005)
28. Wald, I., Ize, T., Kensler, A., Knoll, A., Parker, S.: Ray tracing animated scenes using coherent grid traversal. In: Proceedings of ACM SIGGRAPH 2006) (2006)
29. Wald, I., Slusallek, P., Benthin, C., Wagner, M.: Interactive Rendering with Coherent Ray Tracing. Computer Graphics Forum **20**(3), 153–164 (2001). (Proceedings of Eurographics)
30. Westermann, R., Kobbelt, L., Ertl, T.: Real-time Exploration of Regular Volume Data by Adaptive Reconstruction of Iso-Surfaces. The Visual Computer **15**(2), 100–111 (1999)
31. Wilhelms, J., Gelder, A.V.: Octrees For Faster Isosurface Generation. ACM Transactions on Graphics **11**(3), 201–227 (1992)
32. Yoon, S.E., Lauterbach, C., Manocha, D.: R-lods: Fast lod-based ray tracing of massive models. The Visual Computer (Proc. Pacific Graphics 2006) **22**(9-11), 772–784 (2006)

A Coherent Octree Traversal Algorithm

In this pseudocode, d_{uv} and e_{uv} are SSE vector variables, and k is an integer. Cap depth is $d_{cap} = d_{max} - 1$. For multiresolution, the algorithm is similar except we may intersect slices at lesser stop depth than d_{max}. Also refer to Figs. 6,8, and 9 for illustration of this algorithm.

Algorithm 2 Octree CGT algorithm

```
Require: $axes\ \mathbf{K},\mathbf{U},\mathbf{V};\ packet\ P;\ octree\ volume\ OV; isovalue$
Ensure: $compute\ P\ intersection\ with\ OV$
  for all $depths\ i \in \{0..d_{max}\}$ do
    $d_{uv}[i] \Leftarrow [du_{min}, dv_{min}, du_{max}, dv_{max}]\ /\ 2^{d_{max}-i}$
    $k_0[i] \Leftarrow (P\ enters\ OV)_{\mathbf{K}}\ /\ 2^{d_{max}-i}$
    $k_1[i] \Leftarrow (P\ exits\ OV)_{\mathbf{K}}\ /\ 2^{d_{max}-i}$
    $e_{uv}[i] \Leftarrow [u_{min}, v_{min}, u_{max}, v_{max}]\ at\ k_0[i], k_1[i]$
    $k[i] \Leftarrow k_0[i]$
    $k_{nextMC}[i] \Leftarrow k[i] + 2$
  end for
  $d \Leftarrow 0$
  while $k[d] \le k_1[d]$ do
    if $k[d] = k_{nextMC}[d]$ then
      $d \Leftarrow d - 1$
      continue
    end if
    $traverseChild \Leftarrow false$;
    for all $u \in [u_{min}, u_{max}], v \in [v_{min}, v_{max}]$ of $e_{uv}$ do
      $node \Leftarrow OV.lookup(vec3(k,u,v), d)$
      if $isovalue \in [node.min, node.max]$ then
        $traverseChild \Leftarrow true$
        break
      end if
    end for
    if $d = d_{cap}$ then
      clip $e_{uv}$ to non-empty cap-level macrocells
    end if
    if $traverseChild = true$ then
      if $d = d_{max}$ then
        clip cell slice $e_{uv}$ to active rays
        intersect P with slice $k[d_{cap}]$ at $e_{uv}[d_{cap}]$
        if all rays in P hit then
          return
        end if
      else
        $e_{uv}[d] \Leftarrow e_{uv}[d] + d_{uv}[d]$
        $k_{new}[d+1] \Leftarrow 2 * k[d]$
        $k[d+1] \Leftarrow k_{new}[d+1]$
        $k_{nextMC}[d+1] \Leftarrow k[d+1] + 2$
        $d \Leftarrow d + 1$
        continue
      end if
    end if
    $e_{uv}[d_{cap}] \Leftarrow e_{uv}[d_{cap}] + d_{uv}[d_{cap}]$
  end while
```

B Quality Comparison

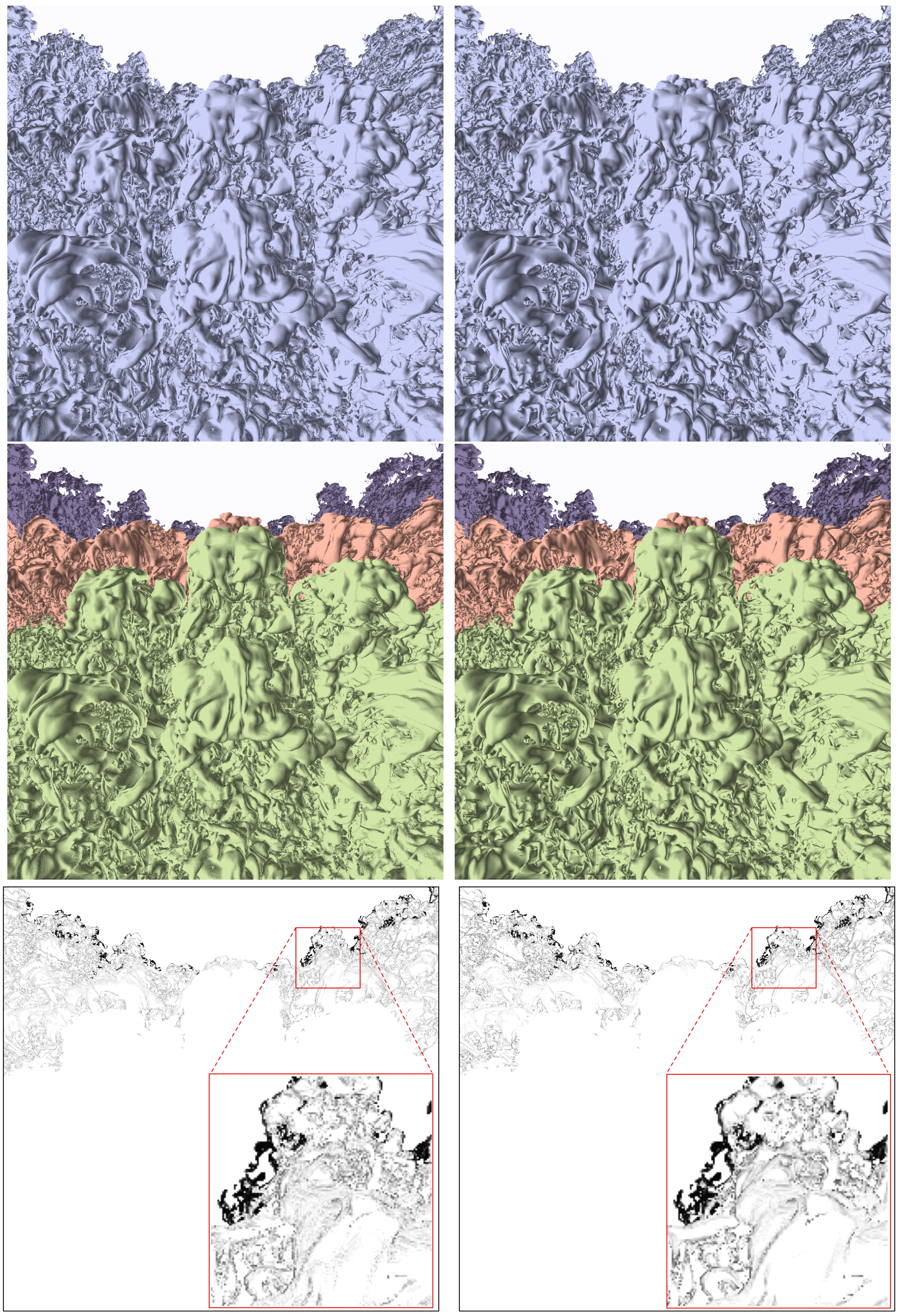


Fig. 16 *Qualitative comparison of results on the RM data, t=270,* 1024^2 *framebuffer, using 8x8 packets.* **Top row:** without multiresolution, with forward differences (left) and central differences (right), rendering at 1.8 and 1.3 fps, respectively. **Middle row:** color-coded multiresolution with $dP/dV = 1$, rendering at 4.2 and 3.1 fps for forward and central differences respectively. **Bottom row:** inverted differences between the results with and without multiresolution. Benchmarks performed on an 8-core dual Intel Xeon 3 GHz desktop with 4 GB RAM.