SCI INSTITUTE
TECHNICAL REPORT

Coherent Multiresolution Isosurface Ray Tracing

Aaron Knoll, Charles Hansen and Ingo Wald

UUSCI-2007-001

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

January 18, 2007

Abstract:

We implement and evaluate a fast ray tracing method for rendering large structured volumes. Input data is 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 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.

THE UNIVERSITY OF UTAH

Coherent Multiresolution Isosurface Ray Tracing

Aaron Knoll Charles Hansen Ingo Wald

Scientific Computing and Imaging Institute, University of Utah
{knolla|hansen|wald}@sci.utah.edu

Abstract

We implement and evaluate a fast ray tracing method for rendering large structured volumes. Input data is 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 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.

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 delivers consistently real-time frame rates for moderate-size data; but GPU memory imposes a limit on the volume size. Although large data access can be achieved through out-of-core techniques, for complex scenes the direct volume rendering algorithm has difficulty rendering a precise isosurface, given a limited number of slices for sampling a high-resolution scalar field.

Ray tracing, though also dependent on the CPU, is not limited to polygonal geometry, and can render implicit surfaces that are locally correct with respect to input data. Ray tracing also 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. Finally, rendering on the CPU allows for access to full system 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 on substantial workstation hardware [1]. In this work, we optimize isosurface ray tracing with a coherent octree traversal technique, then employ a multiresolution level of detail scheme to ensure coherence and hence performance.

2 Related Work

Extraction The conventional method for isosurface rendering has been extraction via marching cubes [2] or some variant; paired with z-buffer rasterization of the resulting mesh. Wilhelms and Van Gelder [3] proposed a min/max octree hierarchy that allowed the extraction process to only consider cells containing the surface. Livnat & Hansen [4] improved this concept with a view-dependent frustum culling technique. Westermann et al. [5] further extended it to adaptive extraction of multiresolution (though not compressed) octree volume data. Liu et al. [6] cast rays through an octree to determine visible "seed" cells for isosurface extraction. Livnat & Tricoche [7] combined extraction with point-based rendering, allowing high-frequency regions of voxels to be represented by splats, and delivering smooth results without relying on adaptive LOD methods.

Direct Volume Rendering An alternative to isosurfacing is direct volume rendering (DVR), e.g. Levoy [8], commonly implemented on graphics hardware by compositing slices of a 3D texture, e.g. Cabral et al. [9]. LaMar et al. [10] proposed a multiresolution sampling of octree tile blocks according to view-dependent criteria. Boada et al. [11] proposed a coarse octree built upon uniform subblocks of the volume, and a memory paging scheme. Large data has been addressed via block-based adaptive texture schemes (e.g. Kraus & Ertl [12]), and out-of-core processing of an octree hierarchy of wavelet-compressed blocks (e.g. Guthe et al. [13]).

Ray Tracing Volumes Interactive isosurfacing of large volumes was first realized in a ray tracer by Parker et al. [14], 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. Parallel ray tracing allowed for full use of main memory on supercomputers or workstations. DeMarle et al. [15] extended the system to clusters, allowing arbitrarily large data to be accessed via distributed shared memory.

Recent works in coherent ray tracing [16, 17, 18] 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. For faster isosurface ray tracing, the ray-cell intersection test was adapted to a SIMD SSE architecture by Marmitt et al. [19]. Then, using implicit kd-trees, Wald et al. [20] implemented a coherent isosurface ray tracing system.

Knoll et al. [1] implemented a single-ray traversal scheme for rendering compressed octree volume data. By employing one structure for both the min/max acceleration tree and the voxel data itself, the authors were able to render 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.

3 Coherent Ray Tracing of Volume Data using Level of Detail

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 rays into groups, or packets, when they share common characteristics. In the case of ray casting, a packet consists of a group of neighboring rays with common origin. 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. In our context, this behavior depends on both dataset and camera. Since our application employs constant-width packets, coherence is a function of scene complexity.

Large Data and Incoherence Coherent ray tracing poses a significant caveat: large volume data are more complex, and thus less coherent, than small volumes. Successful coherent systems have been optimized for relatively small dynamic polygonal data [18, 17] in which many rays intersect common primitives. For sufficiently complex scenes, where each ray intersects a different primitive, intersection costs at least as much as it would in the single-ray case. Worse yet, coherent traversal may induce more intersection tests than a single-ray traversal. In this scenario, a coherent system may perform worse than a single-ray tracer. On scenes with poor coherence, coherent isosurface ray tracing using conservative 2x2 ray packets [20] has produced performance generally on par with a single-ray system [1].

Coherence via Level of Detail Our solution to the problem of poor coherence in complex scenes is a multiresolution level of detail scheme. The premise is simple: when data is sufficiently complex to hamper coherent ray tracing, we render a coarser-resolution representation. This is accomplished by lazy enforcement of a fixed ratio of voxels to pixels during octree traversal. The octree volume is inherently suited as a multiresolution LOD structure; coarser-resolution voxel data can be stored in interior nodes at practically no extra cost. Moreover, one can build a multiresolution volume with an embedded efficiency structure and multiple levels of detail, all for a fraction of the footprint of the original uncompressed data. To render a coarser level of detail, one simply specifies a "stop depth", or a cut, that is less than maximum octree depth. Then, the ray tracer omits both traversal and intersection of subtrees below this stop depth, instead intersecting larger cells at the coarser depth. As more rays intersect a common, wider cell, coherence, and therefore speedup, is achieved.

The main contributions of this work involve extending a static-resolution octree volume to multiresolution; devising a coherent traversal technique for the octree; and leveraging the traversal technique to reduce the cost of compressed data access. Ultimately, we present a coherent multiresolution traversal that delivers interactive ray tracing on a single processor with some tradeoff in quality; provides faster rendering with larger frame buffers on current multicore workstation hardware; and allows for improved shading techniques that would be expensive in a conventional non-coherent octree volume ray tracer.

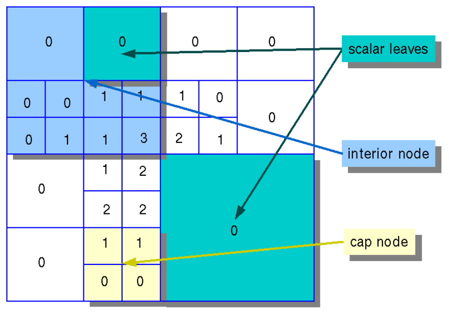


Figure 1: *Octree volume format illustrated*, showing examples of an interior node, a cap node, and scalar leaves. A scalar leaf is not a separate structure, but rather a single value embedded inside its parent interior node. Similarly, cap nodes are not leaves themselves but contain eight scalars at the maximal depth of the octree. Thus, nodes in this structure are the parents of nodes in the logical octree.

4 Multiresolution Octree Volume Construction

An octree volume is an adaptive hierarchical 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.

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 build is detailed by Knoll et al. [1]; 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. This format is detailed in Fig. 1.

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. Our multiresolution system is constructed identically to the static-resolution implementation [1], with separate eight-value arrays for both child pointers and scalar leaves. The only difference is that non-leaf scalars are actually employed in multiresolution rendering.

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. 2). 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. 1), as shown in Fig. 3 (top). 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. 3, 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.

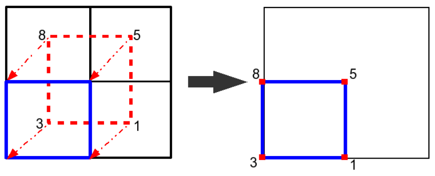


Figure 2: *Voxel-cell mapping.* Given a scalar-centered voxel (blue outline), we construct a cell at that location by mapping the scalar to the lower-most vertex, and assigning forward neighboring scalars to the remaining vertices. This is equivalent to the dual, but spatially offset to align with the coordinate frame of the voxel.

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

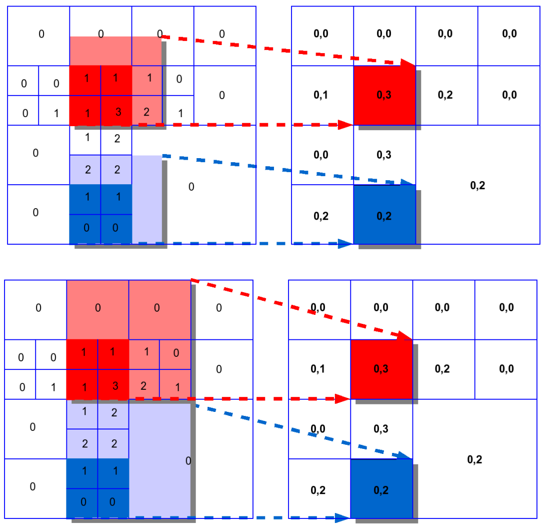


Figure 3: *Min/max tree construction from forward neighbors.* **Top:** Each leaf node must compute the minimum and maximum of its cell, hence account for the values of neighbors in the positive X and Y dimensions (left). This yields a min/max pair for the leaf node (right). Neighbors can potentially exist at different depths of the octree, as is the case for at the blue leaf node. **Bottom:** For multiresolution data, we must include wider neighbors at coarser resolutions into the min/max computation.

and a frustum-based octree traversal similar to the coherent grid traversal of Wald et al. [18].

5.1 SSE Packet Architecture

A coherent ray tracer achieves its performance by operating on groups of neighboring or similar rays in packets. Traversal iself is in fact fairly orthogonal to the chosen packet structure, generally requiring only a packet bounding frustum and a method of iterating over member rays. 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. When packet traversal and intersection have completed, we iterate over each SIMD packlet and shade using the deferred hit information and a given material. Example pseudocode of a packet structure is given below.

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 [21] or BVH [17]; or a breadth-first coherent grid traversal (CGT) approach [18]. We choose the latter for several reasons. Our primitives are regular, non-

Algorithm 1 Ray Packet Structure

```
const int NUM_PACKLETS = NUM_RAYS / 4;
struct RayPacket
{
  simd orig[3][NUM_PACKLETS];
  simd dir[3][NUM_PACKLETS];
  simd inv_dir[3][NUM_PACKLETS];

  simd hit_t[NUM_PACKLETS];
  simd hit_pos[3][NUM_PACKLETS];
  simd normal[3][NUM_PACKLETS];

  simd hit_mask[NUM_PACKLETS];
};
```

overlapping cells, similar to large spherical particle data sets for which CGT has proven effective by Gribble et al. [22]. 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.

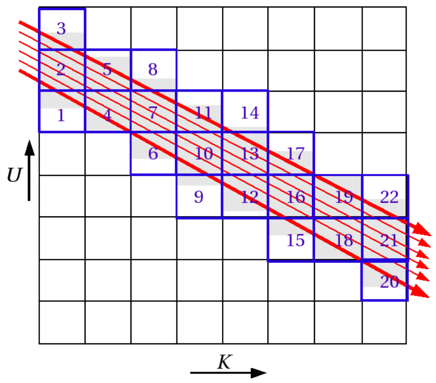


Figure 4: *Coherent Grid Traversal.* The classic CGT algorithm traverses a packet of rays through a grid slice by slice along a major march axis $\vec{K}$, incrementing each subsequent slice extents by the differential of the bounding frustum along the non-major axis $\vec{U}$, as well as the third axis $\vec{V}$ in the 3D case. This illustration numbers the grid cells in the order of their traversal. Unlike the single-ray DDA grid algorithm, cells may be traversed in arbitrary $\vec{U}$,$\vec{V}$ order; however the $\vec{K}$ order is invariably front to back.

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 $\vec{K}$, and the remaining axes are denoted $\vec{U}$ and $\vec{V}$. 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 $\vec{K}$ 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 $\vec{U}$,$\vec{V}$ 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. The 2D analog of this algorithm is illustrated in Fig. 4.

Macrocell Hierarchical CGT The original CGT paper [18] 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. [23], 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. This algorithm is illustrated in Fig. 5.

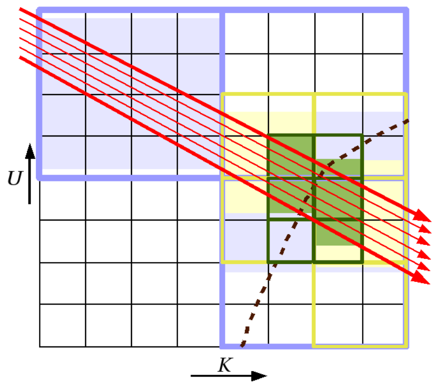


Figure 5: *Coherent Octree Traversal via Implicit Macrocells.* Our hierarchical grid employs recursively superimposed macrocell blocks, with each parent containing 2^3 children, for alignment with the octree volume. We depict a 3-deep hierarchy, with blue, yellow and green extents corresponding to macrocell 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, and their min/max pairs are retrieved from the octree volume via hashing.

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. Figs. 5, 6, and 7 illustrate traversal; refer to Algorithm 3 in the Appendix for pseudocode.

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 [1] 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.

Clipping the Cap-Level Macrocell Slice 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. 1). 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. 6, top).

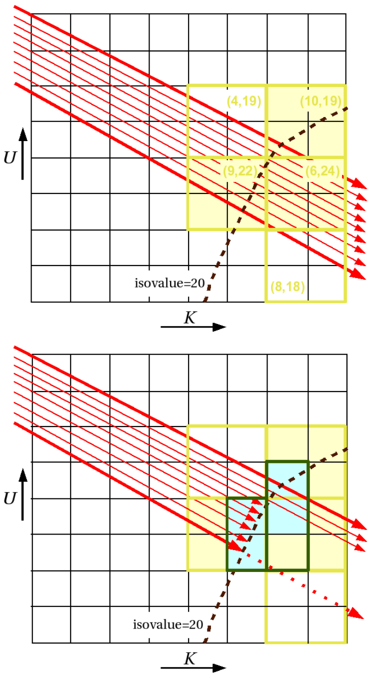


Figure 6: *Slice extent clipping optimizations.* **Top:** We first clip slices of deepest macrocells, corresponding to cap nodes of the octree at depth $d_{max} - 1$. We narrow the u,v slice extents by omitting macrocells with ranges outside our value; only the shaded cells containing our isovalue are considered. **Bottom:** Having done this, we intersect individual rays in the packet with the bounding box of each finest-level slice of cells. "Inactive" rays that have already hit the surface are omitted. This allows us to further constrict the u,v slice extents before intersecting a $\vec{K}$-slice of cells.

Clipping the Cell 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. This enables us to considerably shrink the u,v extents, simply by computing the minimum and maximum of the enter and exit hit coordinates of active rays (Fig. 6, bottom).

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. 2). Our task now is to reconstruct cells efficiently from the octree, exploiting coherence whenever possible.

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

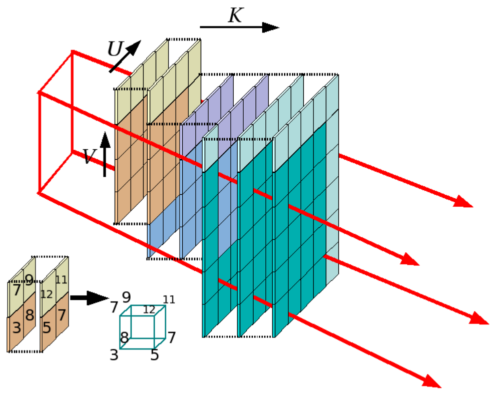


Figure 7: *Slice-based cell reconstruction algorithm.* To find the eight vertices of each cell, rather than neighbor-find seven forward-neighbors per voxel, we exploit our slice-based traversal to look up and cache $\vec{K}$-slices of voxels. Thus we exploit coherence to reduce the overall cost of data access. Above, we illustrate five successive slices, with like colors representing where voxel-caching can be used to avoid repeat neighbor-finding. As evident from lighter-colored voxels at the upper right of each slice, we require the voxels at the $u_{max}+1, v_{max}+1$ extents to reconstruct a cell. Similarly, we can re-use the previous cell's $k+1$ slice to at least partially construct the subsequent $\vec{K}$-slice.

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 (Fig. 7).

Copying the Previous-Step $\vec{K}$-Slice In cell reconstruction, we also exploit voxel coherence along the $\vec{K}$ 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 $\vec{K}$ 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. 7). 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 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. 2), 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. [19]. 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, where more than one ray in packlet seldome intersects the same cell (Tab. 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. (Tab. 3). This behavior is detailed more fully in Section VIII. The purpose of the multiresolution system is to manage pathological cases posed by large data, and preserve coherence while sacrificing quality as little as possible.

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. Moreover, the octree hash scheme operates on canonical octree space $[0,2^{d_{max}}]$, regardless of resolution level.

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 [10]. 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 $\vec{U}$, namely the increment between each primary ray along $\vec{U}$, du. Then, we can render the coarser resolution at d_{stop} when $du = Q_{stop} * dV$, where dV is the $\vec{U}$-width of a voxel, and Q_{stop} is some constant threshold. As the $\vec{U}$-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.

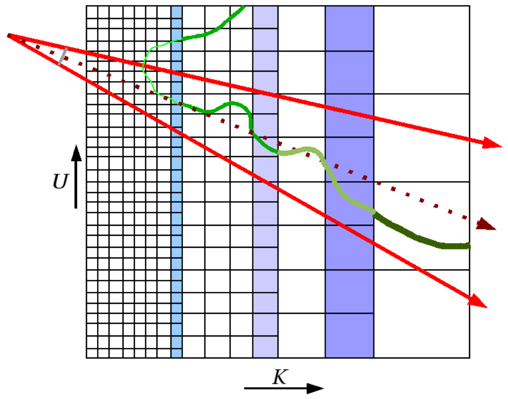


Figure 8: *Multiresolution transition slices.* We determine transition slices along the major march axis $\vec{K}$, transitioning from finer to coarser resolution as slices progress from the camera origin. For the transitions, we determine the last k-slice where each resolution level corresponds to a fixed voxel-to-pixel ratio.

LOD Mapping via $\vec{K}$ 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 $\vec{K}$. We note that the width of a pixel corresponds to the distance between primary rays along the $\vec{U}$ and $\vec{V}$ axes, which increases with greater t, as we move farther from the camera origin. If we consider a major march direction $\vec{K}$, 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 $\vec{K}$-slice, k. As packets traverse the octree one $\vec{K}$-slice at a time, we have a constant world-space LOD map 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 $\vec{K}$-component of the direction ray to compute the $\vec{K}$-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, as in Algorithm 2. The dP/dV constant is thus our base quality metric; Fig. 12 shows the same scene rendered using multiresolution and varying dP/dV.

Algorithm 2 Transition Array Computation

```
Require: Pixel-width to voxel-width ratio, $dP/dV$
  Per-ray camera offset along $\vec{U}$ axis, $du_{camera}$
Ensure: Array of $\vec{K}$-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 $\vec{K}$ 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, and the traversal algorithm seeks the subsequent coarser-resolution transition slice.

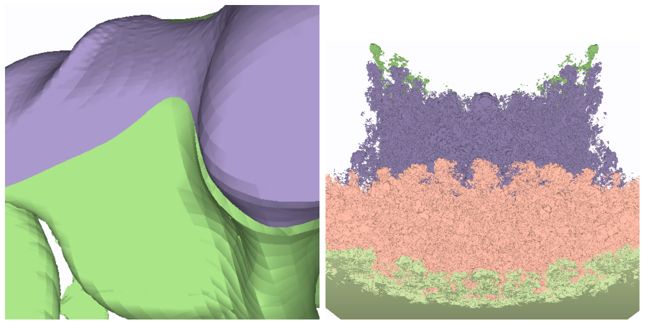


Figure 9: *Color-coded multiresolution.* **Left:** transitions between isosurfaces are smoothed by substituting coarser-detail voxel values into finer-detail cell vertices at the transition slice. **Right:** three transitions along the $\vec{K}$ axis, from finer to coarser levels of detail, on the Richtmyer-Meshkov data.

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 $\vec{K}$-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. 9, left). 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 (Fig. 9, right).

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.

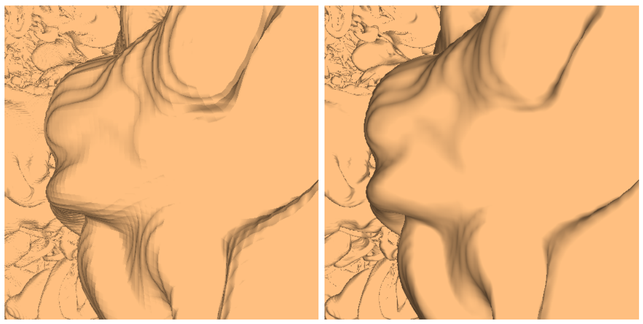


Figure 10: *Gradient normals*, computed on a forward differences stencil yielding 5.5 FPS (**left**), and a central differences stencil at 4.7 FPS (**right**) on an Intel Core Duo 2.16 GHz with a 512^2 frame buffer. The lookup overhead of a 4^3 neighborhood of voxels makes central differences extremely costly in a single-ray or depth-first system. The slice-based coherent scheme delivers smooth normals with a far lesser penalty.

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. 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. 10).

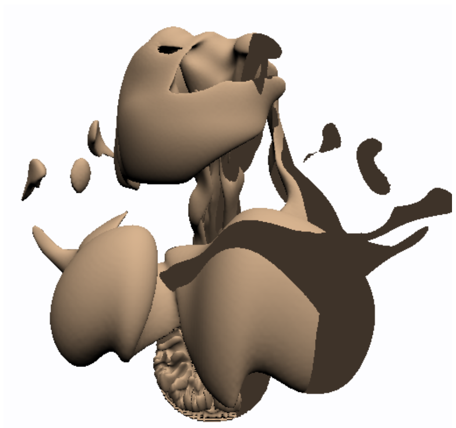


Figure 11: *Shadows.* By coherently casting shadow rays through a coarser-resolution version of our data, we achieve higher performance while providing similar spatial depth cues as shadows cast on the full-resolution volume. With centrally-differenced gradient normals, the above shadowed scene renders at 3.9 FPS on an Intel Core Duo 2.16 GHz with a 512^2 framebuffer; only slightly slower than without shadows at 5.1 FPS.

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 level of detail 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*. As shown in Fig. 11, this often yields framerates only 20%-30% slower with shadows than without.

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. Tab. 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 (Tab. 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 (Tab. 1). Not coincidentally, such "solid" data segments are best suited for visualization via isosurfacing (Fig. 14).

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. Building a single timestep of the 302^3 heptane volume requires a mere 8 seconds; whereas a timestep of the Richtmyer-Meshkov data takes 45 minutes. 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 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 [20] could require up to twice the full data footprint, often with an additional overhead of around 15% for cache-efficient bricking [23]. Thus, octrees compare quite favorably to competing volume ray tracing structures.

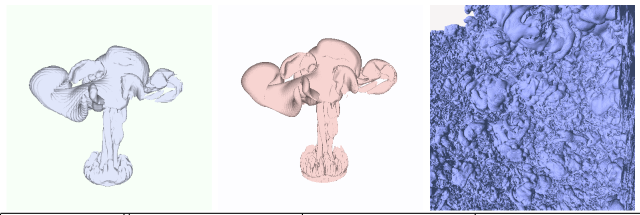
8.2 Coherent Traversal Results

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 (Tab. 2). Moreover, rendering time is strongly correlated with the number of ray-cell intersections. Code profiling reveals that traversal occupies a mere 5%-15% of CPU time, compared to well over 70% spent in cell 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	0.78
+*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
+*multires*	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, Tab. 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. *+multires:* multiresolution scheme, with $dP/dV = 1$. Tests at 512^2 using one core of an Intel Core Duo 2.16 GHz.

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 (Tab. 3).



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 coherence*, 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. The leftmost (HEP64) is the 64^3 downsampled heptane data, and has high coherence at the primitive level. (HEP302) is the same as in Tab. 2: a moderate case in which few gains are made from intersection-level coherence, but coherent traversal is beneficial. (RM) is a pathological case for packet traversal, in which neighboring rays in a packet are separated by numerous cells. Allowing large data such as this to benefit from coherence requires a multiresolution scheme. Tests run on a single core of an Intel Core Duo 2.16 GHz, with a 512^2 frame buffer and multiresolution disabled.

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 (Fig. 6) noticeably alleviate this, as we can see in Tab. 2. However, for complex scenes such as far views of large data, rendering cost is totally bound by intersection (Tab. 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 the less expensive traversal, and cause the coherent ray tracer, without multiresolution, to perform worse than a conventional single-ray algorithm on sufficiently complex scenes.

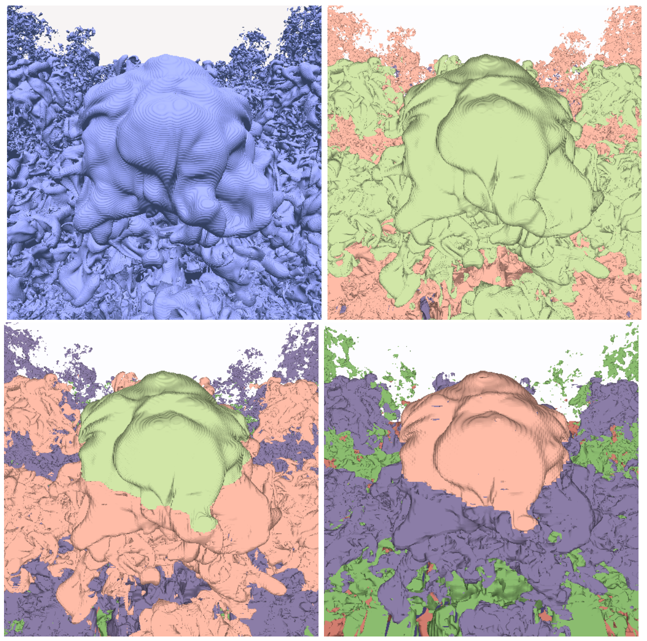


Figure 12: *Qualitative impact of multiresolution* on the Richtmyer-Meshkov data at t=270, isovalue 20. Top left to bottom right: single-ray, then coherent multiresolution with dP/dV of 1,2 and 4. On an Intel Core Duo 2.16 GHz with a 512^2 frame buffer, these render at 0.92, 1.0, 1.9, and 3.6 FPS respectively. To illustrate LOD transitions, like colors indicate the same resolution.

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. Fig. 12 illustrates behavior of the RM data with our LOD system with varying *dP/dV*.

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 (Tab. 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 Tab. 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 as in Fig. 14.

SCENE	C.DUO,512^2		NUMA,512^2		NUMA,1024^2	
	single	8x8	single	8x8	single	8x8
50, far	2.5	3.5	17.9	25.1	4.9	7.1
150, far	1.9	2.5	13.6	17.9	3.7	5.8
270, far	1.1	1.1	8.1	7.8	2.4	3.5
50, close	2.0	6.9	14.3	48.5	4.0	16.1
150, close	1.7	8.1	14.2	57.5	4.0	16.7
270, close	0.2	4.7	1.48	33.6	0.5	10.5

Table 4: *Frame rates of various time steps of the Richtmyer Meshkov data,* on an Intel Core Duo 2.16 GHz laptop (2 GB RAM) and a 16-core NUMA 2.4 GHz Opteron workstation (64 GB RAM). Refer to Fig. 13 for images.

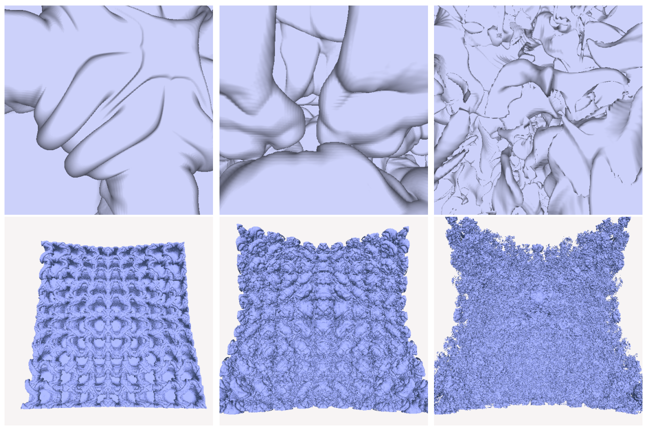


Figure 13: *Richtmyer-Meshkov results.* From left to right, timesteps 50, 150 (isovalue 20), and 270 (isovalue 160). **Top:** various close-up camera views, illustrating highly coherent scenes. We use $dP/dV = 1$. **Bottom:** far views with the same camera position, exhibiting generally poor coherence.

8.4 Comparison to Existing Systems

Tab. 4 demonstrates performance results on the Richtmyer Meshkov dataset in comparison to our single-ray implementation [1]. In the best-case scenario we achieve a factor of 23 faster than single-ray performance, and even in worst cases the coherent implementation does not exhibit substantially inferior performance. These numbers compare favorably to other implementatations. For similar camera positions, we achieve the same 2 FPS RM data performance on an two-core Intel Core Duo as DeMarle et al. [15] report on a 64-processor cluster with a distributed shared memory layer. We are competitive with Wald et al. [20] for far views, and perhaps faster for close-up scenes, while generally requiring an order of magnitude lesser memory footprint.

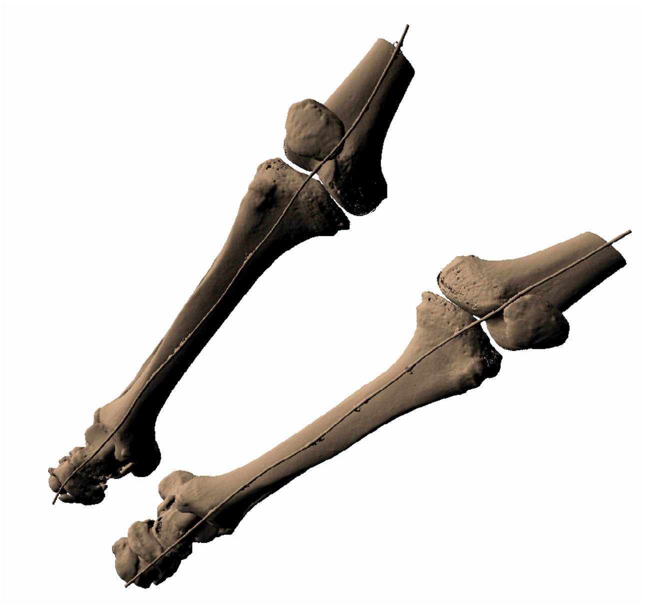


Figure 14: *The visible female femur.* The original, full 617x512x512 volume occupies more space as an octree than uncompressed, due to the entropic nature of soft tissues. Bone, which is more appropriately visualized as an isosurface, can be represented by 100-163 isovalue segments, and compressed into an octree volume with a 20:1 ratio, including the multiresolution data and min/max acceleration structure. For this scene, coherent ray tracing scales well to large frame buffers. The image 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.

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.

Octree ray tracing is not ideal for all volume rendering applications. For smaller volume data with uniformly high isovalue variance, an octree can actually occupy more space than a 3D array, and explicit macrocell-grid or kd-tree traversal might perform slightly better. However, for small data with simple shading models a GPU volume renderer would generally be preferable to an interactive ray tracing solution. Thus, our method is primarily useful for large volumes, or medium volumes with numerous timesteps. As large data is the impetus for ray tracing in the first place, the octree is well suited to this particular application.

On current dual-core computers, single-ray octree volume ray tracing [1] performs sub-interactively, albeit at multiple frames per second. The main goal of this work was to devise a system that would consistently allow for interactivity. Overall, we accomplish that: it is always possible to relax the pixel to voxel width ratio to the point where performance is interactive. However, doing so often requires visualizing coarser resolutions than would be ideal. We find a better application of coherent techniques to be high-quality rendering of large frame buffers on multicore workstations. This exploits coherent traversal without resorting to overly aggressive coarse LODs, and will be interactive on commodity hardware in the near future. Future improvements to our system could explore this path. Implementing super-sampled filtering, and intersecting higher-order implicit primitives defined on wider cell kernels, could result in extremely high-quality isosurfaces of large data, for applications where local detail and feature correctness are critical.

An overarching concern is that level of detail may not be an ideal solution for such high-quality rendering, and ultimately performance gains from improved coherence may not justify the loss in quality. One of the major advantages of ray tracing, when compared to rasterization, is that rendering is less bound by geometric complexity. Effectively, complex scene geometry can be rendered linearly with respect to the number of rays cast, as opposed to the number of objects in the scene. As shown by Knoll et al. [1], a 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. Thus, coherence *depends* on level of detail. Without it, complex scenes can easily perform worse than under a naive single-ray traversal.

The problem of applying coherent techniques to incoherent, complex scenes is a major obstacle for rendering large data with packet architectures, and for ray tracing in general. While the primary focus of our work was a visualization system, we have sought to at least identify this issue, if not address it. Multiresolution volumes make an effective testbed for this problem, as complexity can be easily measured in number of voxels, and surface geometry retains similar features between levels of detail. Future work in ray tracing should improve the behavior of coherent traversal on inherently incoherent scenes. While difficult, this would be possible if the overall cost of primitive intersection were substantially reduced.

References

[1] A. Knoll, I. Wald, S. Parker, and C. Hansen, "Interactive Isosurface Ray Tracing of Large Octree Volumes," in *Proceedings of the IEEE Symposium on Interactive Ray Tracing*, 2006.

[2] W. E. Lorensen and H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," *Computer Graphics (Proceedings of ACM SIGGRAPH)*, vol. 21, no. 4, pp. 163–169, 1987.

[3] J. Wilhelms and A. V. Gelder, "Octrees For Faster Isosurface Generation," *ACM Transactions on Graphics*, vol. 11, no. 3, pp. 201–227, July 1992.

[4] Y. Livnat and C. D. Hansen, "View Dependent Isosurface Extraction," in *Proceedings of IEEE Visualization '98*. IEEE Computer Society, Oct. 1998, pp. 175–180.

[5] R. Westermann, L. Kobbelt, and T. Ertl, "Real-time Exploration of Regular Volume Data by Adaptive Reconstruction of Iso-Surfaces," *The Visual Computer*, vol. 15, no. 2, pp. 100–111, 1999.

[6] Z. Liu, A. Finkelstein, and K. Li, "Improving Progressive View-Dependent Isosurface Propagation," *Computers & Graphics*, vol. 26, no. 2, pp. 209–218, 2002.

[7] Y. Livnat and X. Tricoche, "Interactive Point-based Isosurface Extraction," in *Proceedings of IEEE Visualization 2004*, 2004, pp. 457–464.

[8] M. Levoy, "Efficient Ray Tracing for Volume Data," *ACM Transactions on Graphics*, vol. 9, no. 3, pp. 245–261, July 1990.

[9] B. Cabral, N. Cam, and J. Foran, "Accelerated volume rendering and tomographic reconstruction using texture mapping hardware," in *VVS '94: Proceedings of the 1994 symposium on Volume visualization*. New York, NY, USA: ACM Press, 1994, pp. 91–98.

[10] E. LaMar, B. Hamann, and K. I. Joy, "Multiresolution Techniques for Interactive Texture-based VolumeVisualization," in *Proceedings IEEE Visualization 1999*, 1999.

[11] I. Boada, I. Navazo, and R. Scopigno, "Multiresolution Volume Visualization with a Texture-Based Octree," *The Visual Computer*, vol. 17, no. 3, 2001.

[12] M. Kraus and T. Ertl, "Adaptive Texture Maps," *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 2002.

[13] S. Guthe, M. Wand, J. Gonser, and W. Straßer, "Interactive Rendering of Large Volume Data Sets," in *Proceedings of the conference on Visualization '02*. IEEE Computer Society, 2002, pp. 53–60.

[14] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan, "Interactive Ray Tracing for Isosurface Rendering," in *IEEE Visualization*, October 1998, pp. 233–238.

[15] D. E. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen, "Distributed Interactive Ray Tracing for Large Volume Visualization," in *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, 2003, pp. 87–94.

[16] A. Reshetov, A. Soupikov, and J. Hurley, "Multi-Level Ray Tracing Algorithm," *ACM Transaction of Graphics*, vol. 24, no. 3, pp. 1176–1185, 2005, (Proceedings of ACM SIGGRAPH).

[17] I. Wald, S. Boulos, and P. Shirley, "Ray tracing deformable scenes using dynamic bounding volume hierarchices," SCI Institute, University of Utah (conditionally accepted at ACM Transactions on Graphics, 2006), Tech. Rep. UUSCI-2006-023, 2006.

[18] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. Parker, "Ray tracing animated scenes using coherent grid traversal," in *Proceedings of ACM SIGGRAPH 2006)*, 2006.

[19] G. Marmitt, H. Friedrich, A. Kleer, I. Wald, and P. Slusallek, "Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing," in *Proceedings of Vision, Modeling, and Visualization (VMV)*, 2004, pp. 429–435.

[20] I. Wald, H. Friedrich, G. Marmitt, P. Slusallek, and H.-P. Seidel, "Faster Isosurface Ray Tracing using Implicit KD-Trees," *IEEE Transactions on Visualization and Computer Graphics*, vol. 11, no. 5, pp. 562–573, 2005.

[21] I. Wald, P. Slusallek, C. Benthin, and M. Wagner, "Interactive Rendering with Coherent Ray Tracing," *Computer Graphics Forum*, vol. 20, no. 3, pp. 153–164, 2001, (Proceedings of Eurographics).

[22] C. Gribble, T. Ize, A. Kensler, I. Wald, and S. G. Parker, "A coherent grid traversal approach to visualizing particle-based simulation data," SCI Institute, University of Utah (conditionally accepted at ACM Transactions on Graphics, 2006), Tech. Rep. UUSCI-2006-024, 2006.

[23] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley, "Interactive Ray Tracing for Volume Visualization," *IEEE Transactions on Computer Graphics and Visualization*, vol. 5, no. 3, pp. 238–250, 1999.

We provide abbreviated pseudocode for coherent grid traversal of an octree volume. In our implementation, d_{uv} and e_{uv} are SSE datatypes, and k is an integer. Cap depth is given as $d_{cap} = d_{max} - 1$. For multiresolution, the algorithm is similar except we may intersect slices at lesser stop depth than d_{max}. Also refer to Figs. 5, 6, and 7 for illustration of this algorithm.

Algorithm 3 Octree CGT algorithm

```
Require: $axes\ \vec{K}, \vec{U}, \vec{V};\ packet\ P;\ octree\ volume\ OV;\ isovalue$
Ensure: $compute\ P\ intersection\ with\ OV$
  for all $depths\ i \in \{0..d_{max}\}$ do
    $d_{uv}[i] \Leftarrow [du_{min}, dv_{min}, du_{max}, dv_{max}]\ /\ 2^{d_{max}-i}$
    $k_0[i] \Leftarrow (P\ enters\ OV)_{\vec{K}}\ /\ 2^{d_{max}-i}$
    $k_1[i] \Leftarrow (P\ exits\ OV)_{\vec{K}}\ /\ 2^{d_{max}-i}$
    $e_{uv}[i] \Leftarrow [u_{min}, v_{min}, u_{max}, v_{max}]\ at\ k_0[i], k_1[i]$
    $k[i] \Leftarrow k_0[i]$
    $k_{nextMC}[i] \Leftarrow k[i] + 2$
  end for
  $d \Leftarrow 0$
  while $k[d] \leq k_1[d]$ 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
```