Interactive Isosurface Ray Tracing of Large Octree Volumes

Aaron Knoll Ingo Wald Steven Parker Charles Hansen

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

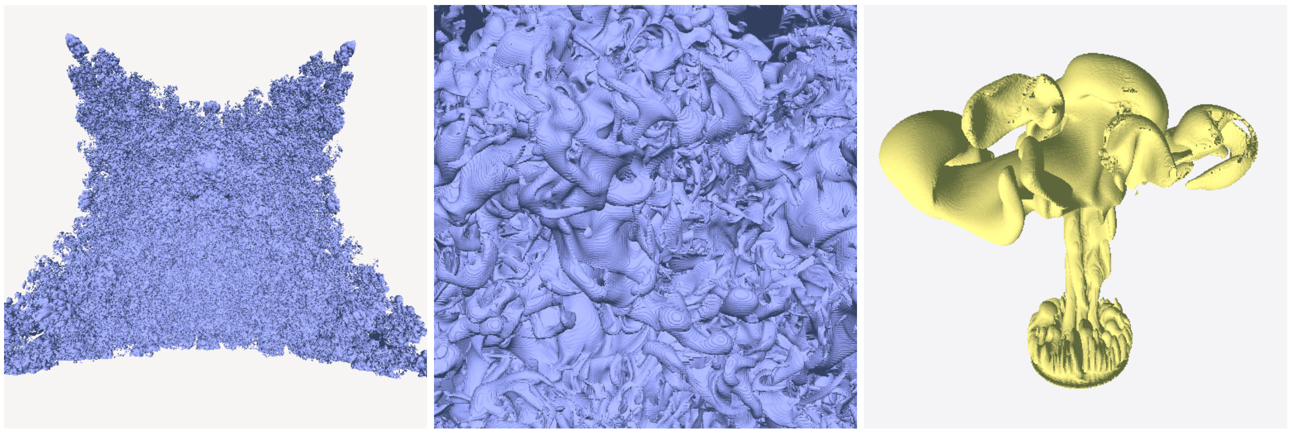

Figure 1: *Large volume data ray-traced at* 512^2 *using octrees for compression and acceleration.* From left to right: (1) LLNL Richtmyer-Meshkov instability field (shown at timestep 270, with an isovalue of 100). (2) Closer view of the previous scene. (3) Utah CSAFE heptane simulation (timestep 152, isovalue 42). Data is losslessly compressed into an octree volume to occupy less than one quarter the size of the original 3D array. Our approach permits storage of large data such as the LLNL simulation, and full sequences of medium-size data such as the heptane, in main memory of consumer machines. Frame rates on an Intel Core Duo 2.16 GHz laptop with 2 GB RAM are 2.4, 1.3, and 3.3 fps respectively. On a 16-node NUMA 2.4 GHz Opteron workstation, these images render at 17.9, 9.8, and 22.0 fps.

Abstract

We present a technique for ray tracing isosurfaces of large compressed structured volumes. Data is first converted into a lossless-compression octree representation that occupies a fraction of the original memory footprint. An isosurface is then dynamically rendered by tracing rays through a min/max hierarchy inside interior octree nodes. By embedding the acceleration tree and scalar data in a single structure and employing optimized octree hash schemes, we achieve competitive frame rates on common multicore architectures, and render large time-variant data that could not otherwise be accomodated.

Keywords: isosurface, ray tracing, octree, compression, volume

1 Introduction

Interactive rendering of large volumes is a difficult problem in visualization. With direct volume rendering, GPU memory imposes an absolute limit on the volume size, and the video bus restricts real-time rendering of time-variant data. Adaptive isosurface extraction techniques are fast, but depend on effective processing and streaming of large data to the CPU. Furthermore, they render a piecewise linear mesh that may be topologically different from the true isosurface as defined by the source data. Ray tracing, though traditionally slower, is not limited to rendering polygonal geometry, and can guarantee continuous isosurfaces that locally interpolate the 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 provided by current GPUs. This flexibility enables the use of an adaptive-resolution octree, which we can use as both a natively compressed data format and an acceleration structure for rendering.

Previous works have applied octrees as acceleration structures for ray tracing geometry. In modern interactive ray tracers, however, octrees are unpopular compared to kd-trees, bounding volume hierarchies or hierarchical grids. For general ray tracing, octrees lack the nonrecursive traversal of grids, or ability of kd-trees and BVHs to adapt to overlapping polygonal scene geometry. Volume rendering, however, guarantees regularly-spaced, non-overlapping voxels, which are directly used to construct cell intersection primitives. Moreover, one can potentially extract cache savings from traversing the same hierarchical data structure that encapsulates volume data. Thus, octrees are worth revisiting in the context of volume ray tracing. Our work involves compressing volumes into an octree structure, and employing that for ray traversal.

2 Related Work

Mesh Extraction. With the widespread availability of GPU hardware, the most common trend has been towards isosurface extraction via marching cubes [12] paired with z-buffer rasterization of the resulting mesh. Much work has been done in this area; one of the first applications of an octree for extraction was by Wilhelms and Van Gelder [24], though the structure was used only for acceleration and not compression. Velasco and Torres [20] propose using the octree structure itself to contain, and thus compress, the actual scalar data.

With extraction, it is generally desirable to implement an adaptive scheme that generates a view-dependent mesh per frame, e.g. Livnat et al. [11]. Westermann et al. [23] use an octree for multiresolution adaptive mesh extraction. A major advantage of extraction techniques is that geometry can be effectively streamed from CPU to GPU, e.g. Mascarenhas et al. [14], and extended to remote client-server visualization of large datasets.

Direct Volume Rendering. An alternative to rendering a mesh is direct volume rendering (DVR), e.g. Levoy [10], which integrates rays intersecting a volume. While this process is slow in ray tracing, it is effective on current GPUs by storing the volume as slices of 2D textures and computing gradients across sequential cutting planes. This no longer restricts the viewer to rendering an isosurface, though choosing a singular transfer function yields a surface if desired.

To address the issue of size, Boada et al. [1] propose a coarse octree built upon uniform sub-blocks of the volume, and a memory paging scheme. This enables a DVR system to access larger data, at high cost in performance. Kniss et al. [9] implement an efficient similar structure for mesh painting on the GPU, though do not apply it to large volume data.

Ray Tracing Volumes. Interactive volume isosurfacing was first realized in a ray tracer by Parker et al. [15], using a hierarchical grid of macrocells as an acceleration structure. A single ray is tested for intersection inside a cell of 8 voxels using a cubic root solver to find the intersection point on the implicit. Ray tracing permits the full use of large main memory on supercomputers or workstations. Parallel isosurface ray tracing was extended by DeMarle et al. [3] to clusters, allowing arbitrarily large data to be accessed by a distributed shared memory scheme.

The recent trend of coherent packets [16, 22] has brought interactive ray tracing to the commodity desktop. The ray-cell intersection test was adapted to exploit SIMD and packets by Marmitt et al. [13]. Then, using coherent kd-tree traversal, Wald et al. [21] applied packet ray tracing to isosurface rendering.

Ray Tracing Octrees. Our choice of octree as a container for volume data is convenient for ray tracing. We can use the same hierarchy as an acceleration structure; octrees have been well-studied as structures in ray tracing.

Octrees are, in fact, theoretically optimal in terms of fewest traversal steps, assuming objects are contained uniformly within cells of the acceleration structure, with no overlap [2]. The combination of regular, hierarchical nature of the structure affords many different styles of traversal algorithm. The original Glassner implementation [6] proposed top-down point location testing along successive octree nodes hit by the ray. Samet [17] modified this marching procedure to incoporate a neighbor-finding algorithm, delivering dramatic speedups. Sung [19] proposed a DDA traversal similar to a hierarchical grid. Finally, Gargantini and Atkinson [5] implemented a traversal similar to a kd-tree where the ray intersection with each octant mid-plane is ordered.

Due to their high memory consumption and lack of a clearly optimal traversal implementation [7], octrees were overtaken by hierarchical grids as general-purpose ray tracing structures [8]. With coherent ray tracing, kd-trees have in turn come into favor [16, 22]. Nonetheless, the ability to use a single structure for both ray traversal and scalar storage is tempting, and recommends the octree as an acceleration structure for our application.

Octree Hashing. Of final but important note is previous work in octree hashing. The general goal is *point location*: given (x,y,z) coordinates and the root node of the octree, retrieve a leaf node of the octree at that location. A related problem is *neighbor-finding*, in which we are given a leaf node and asked to find an adjacent neighboring leaf. Incidentally, these two algorithms were pioneered by Glassner [6] and Samet [17] for use in ray tracing; however their application extends beyond ray tracing to the use of any regular binary tree (quadtree, octree, etc.). Frisken and Perry [4] propose an efficient and concise hashing scheme using binary arithmetic on integer coordinates. We build upon their work to create our own fast, general-purpose hashing scheme.

3 Ray Tracing Octree Volumes

An original goal of this work was to render data already in octree form from the simulation. However, as evident from related work, storing and rendering large 3D array volumes is difficult for commodity machines with limited memory. We propose to compress scalar data from a 3D grid into an octree, similar to the approach of Velasco and Torres [20]. Then, rather than extracting a mesh and streaming geometry to the GPU, we seek to ray trace the octree-encapsulated volume directly.

We draw inspiration from previous works that use a min/max tree to simplify extraction and rasterization [20, 23, 24]. The same min/max tree can be used as an acceleration structure for ray tracing, similar to the macrocell grid employed by Parker et al. [15] and implicit kd-tree of Wald et al. [21]. The crux of our work is employment of a single octree structure, used simultaneously as an acceleration structure for ray tracing, and as a hierarchical compression structure for scalar volume data.

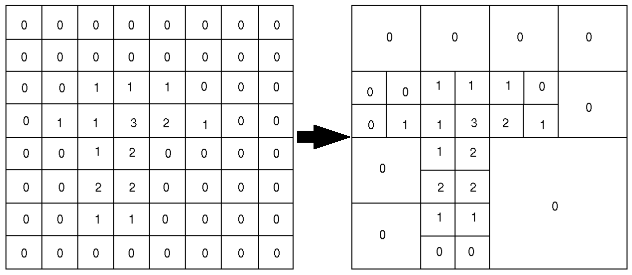

Figure 2: *Generation of a quadtree from a 2D grid* by consolidating pixels with zero inter-pixel variance. The same principle extends to 3D with our octree, a 3D grid and inter-voxel variance.

3.1 Octree Volume Definition

An octree volume is an adaptive-resolution, 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 need not contain scalar data unless we desire a multiresolution representation. Invariably, however, interior nodes define the structure of the octree by maintaing pointers from parents to children.

Volume data could be natively computed and stored in this format; however for our purposes it is desirable to build an octree volume from a scalar field in a 3D array. The process of creating an octree volume is conceptually simple: given input data in the form of a 3D array, we group regions with low variance and output a hierarchically compressed octree volume. Specifically, we consider groups of 8 voxels nested within a parent node of the octree. If these voxels are identical (in lossless compression), or have a combined variance below a desired threshold (lossy compression), we compute their average and consolidate them into a single node at the previous depth level of the octree (Figure 2). By recursively consolidating nodes with low inter-voxel variance, we can build an octree volume in bottom-up fashion.

3.2 Ray Tracing and Voxel-Cell Duality

The crucial technique of our application is ray intersection with the octree data itself, thus using the same structure encapsulating the data to accelerate traversal. Moreover, we wish to use the octree structure in the same manner in which other isosurface ray tracers [15, 21] employed grids and kd-trees: avoiding traversal and

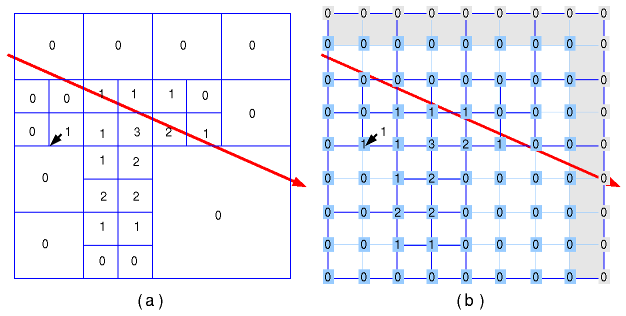

Figure 3: *Ray traversal of the octree.* While the octree volume (a) is given with voxels at the center of each node, we actually seek to ray trace a field of cells with voxel values at the corners (b). To accomplish this, we observe a duality between voxels and cells, by mapping each voxel to the lower-left corner of a cell. Values outside the octree data (in gray) are defined to be zero. Thus, the ray traverses interior nodes of the octree, and intersects with a well-defined cell primitive composed of 8 voxels.

intersection in regions of space that do not contain our desired isovalue within their min/max range.

Our choice to use the same structure for data and acceleration comes with a caveat: though our volume data consists of voxels, we ray trace an isosurface that is defined within *cells* of 8 voxels. Fortunately, there exists a dual relationship between voxels and cells. By logically shifting the position of all scalars backward by half a unit of voxel width, we re-map our scalar field to cells (Figure 3).

Two options exist to accomplish this mapping in memory. One could expand each voxel to contain its forward neighbors, thus store each cell completely. While this would require no additional searching through a structure to retrieve cell corner values, it requires 8 times the storage of the original volume. With our goal of ray tracing compressed data, we instead turn to the approach of Parker et al. [15] which simply retrieves the 7 forward neighbors of a voxel at intersection time. This permits us to traverse interior nodes of the octree volume, and intersect with an 8-voxel (Figure 4), even though the data stored at each leaf node is actually a single scalar value.

With a volume stored in a 3D array, querying the values of these neighbors is trivial: simply an array index into memory that is typically already in cache. For the octree, the process is more intensive. Here, we must employ point location to retrieve the voxel values of the forward neighbors. Full top-down point location from the root would result in a $O(log(N))$ algorithm. However, with neighbor-finding techniques we can significantly reduce this lookup cost. The worst-case complexity of neighbor-finding is $O(log(N))$, but in practice the algorithm skews heavily toward the best-case of $O(1)$, when neighboring voxels lie within the same parent. Even then, neighbor-finding on octree data must perform competitively with the $O(1)$ complexity of lookup on uncompressed 3D arrays. It is readily apparent that octree hashing, specifically neighbor-finding, is a fundamental algorithmic component of our work.

3.3 Computing the Min/Max Tree

Ray tracing cells defined by forward-neighbors (Figure 4) directly impacts the construction of our min/max tree. Specifically, a parent node in the octree must compute the minimum and maximum based not only its own children, but on voxels forward-adjacent to its children as well (Figure 5). Knowing this, one can compute a min/max pair for that leaf node based on the cell corner values. The min/max tree is then computed recursively, by finding for each parent node the minimum and maximum of all children min/max pairs. As we

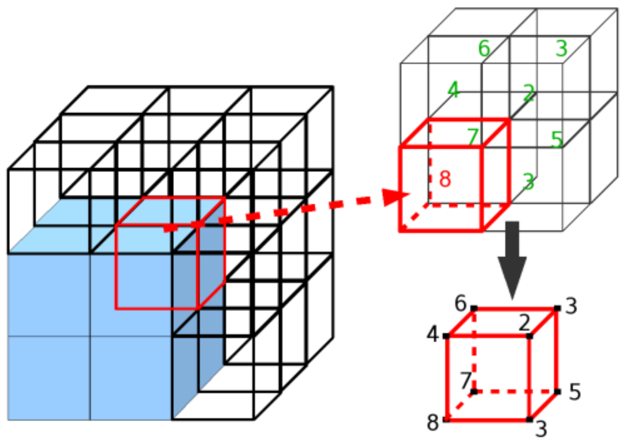

Figure 4: *Retrieving a cell from a neighborhood of voxels.* Given an octree interior node composed of eight voxels (solid blue), we seek to intersect a leaf node (red outline) consisting of a single scalar value. We perform neighbor-finding on the octree structure to retrieve the forward-neighboring voxels (green). This yields a cell of 8 voxels, which we then use as the intersection primitive from which we reconstruct the isosurface.

are only concerned with cells at the finest depth of the octree, it suffices to account for forward-neighbors once at the deepest leaf level, and thereafter compute each parent's min/max pair based on the pairs of the 8 children.

Clearly, storing the min/max tree within the octree data structure entails some overhead. As compression is a major goal of our work, it would be unwise to store the min/max pairs of each scalar voxel, which would demand over three times the storage of the raw octree data. Instead, one could compute the extrema temporarily at leaf nodes, and begin storing the min/max tree at depth $d_{max}-1$. Omitting the min/max pair at leaf nodes would seem to generate a looser tree and hurt performance; but in practice, it simply forces us to compute the minimum and maximum of forward voxels while we are looking them up via neighbor-finding. Logically, this approach entails an overhead of one min/max pair for 8 voxels, plus pairs for other interior nodes of the tree. This suggests approximately a 22% additional footprint on top of raw scalar octree-compressed data. While not insubstantial, that seems acceptable given the acceleration capabilities of the min/max structure.

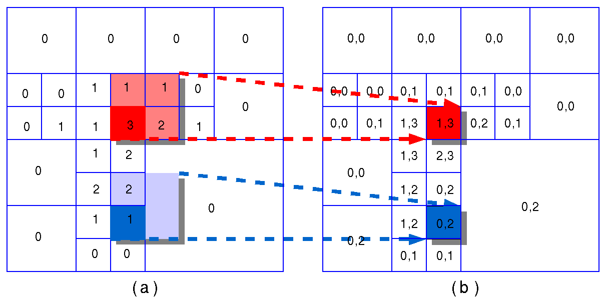

Figure 5: *Min/max tree construction from forward neighbors.* In the quadtree case, each leaf node must compute the minimum and maximum of its cell, hence account for the values of neighbors in the positive X and Y dimensions (a). This yields a min/max pair for the leaf node (b). Neighbors can potentially exist at different depths of the octree, as is the case for at the blue leaf node.

4 Implementation

Our implementation builds on the theoretical foundation laid in the previous section, with details provided for the octree data format, point location and neighbor-finding, and the octree traversal itself. Pseudocode of these algorithms is provided in the appendices; however it is not necessary to understand our approach.

We chose not to employ SIMD or packets. Given our focus on large data, we would expect highly-variant scenes and at best modest speedups from coherent techniques. Wald et al. [21] reported little performance gains from coherent techniques on large data. Specifically, with comprehensive scenes of large volumes, a pixel can frequently cover multiple voxels. With agressively coherent techniques such as frustum-based traversals, this entails much unnecessary work and potentially a performance decrease over single-ray techniques. Moreover, we are first interested in how an optimized single-ray octree algorithm behaves compared to known techniques, and the relative performance of octree volumes versus uncompressed structured data. Coherent octree traversal will likely be explored in future work, however.

4.1 Data Format

To avoid explicitly storing a full node for each leaf of the octree volume, we store nodes corresponding to the parent. In this scheme, at the maximum depth of the octree, all children are guaranteed to be leaves. Thus, at depth $d_{max}-1$ of the octree, we employ a separate structure called a *cap*, consisting simply of 8 scalar values. All other interior nodes contain the values, min/max pairs, and pointers for 8 children. Though a slight misnomer (cap scalars are logically leaves of the tree), we denote any scalar value at non-cap depth a *scalar leaf*. These are stored directly within their parents, and are indicated by a null child pointer.

Rather than store full pointers, we store a 32-bit child_start and a single-byte offset per child. In early implementations, we used binary arithmetic masks and bit-counting to determine which nodes were leaves; in practice however this requires computation (specifically left-shifting by a non-constant) that hampered performance. Ultimately, we use an array to indicate the offset of each child, or -1 if that child is a leaf. We use a second array, child_scalars, to contain the value of each child. In this application we only care about this value when the octant is a scalar leaf at sub-maximum depth; however future implementations could take advantage of this inherently multi-resolution approach to provide a level-of-detail scheme. Details of the structure are provided in Appendix A.

To build our structure, we use a 3D array of rectilinear grid data as input. We determine N, the smallest power of 2 that encompasses the largest dimension of that volume, and choose the maximum depth $d_{max} = log_2(N)$. We then proceed from the bottom-up, assigning groups of 8 voxels from the original structured grid to the caps. Groups of 8 identical voxels are consolidated into a single scalar leaf of the parent. Pointers from interior nodes to children are subsequently filled in, until the root node completes the tree.

The min/max tree is computed simultaneously alongside bottom-up consolidation. As explained in the previous section and in Figure 5, we must consider not only the 8 child voxels of each parent, but their forward-neighbors as well. As a result, we compute the minimum and maximum of 27 voxel values, and store these in our min/max tree. Similarly to how we store scalar values as child_scalars within the parent node, we found it preferable to store the minimum and maximum values of children within the parent nodes, as opposed to the child nodes themselves. This allows us to reject children without actually traversing them, sparing us cache misses.

Values are retrieved from the original data only for cap nodes, and used to compute the min/max tree. Afterwards, parent nodes are computed solely based on the values of their children. When child voxels consolidate into a parent, the corresponding child nodes are removed. This process continues recursively until the root node of the octree is completed.

4.2 Octree Data Lookup

Voxel-cell mapping manifests the need for a fast neighbor-finding routine, which brings us to octree hashing. As mentioned before, we adopt a scheme like that of Frisken and Perry [4], in which octree cells are defined on the interval $[0,2^{d_{max}}]$. Then, given a vector in this coordinate space, we simply cast its components to integers and perform point-location from the root node of the octree.

Point Location. Point location is simply top-down search through the octree; given an initial node, that node's current coordinates, and the coordinates of the desired destination. With full point location, the initial node would be the root, with all-zero coordinates. However, in conjunction with neighbor-finding it is desirable to start point-location deep in the tree. Frisken and Perry [4] propose creating a single-bit mask corresponding to the current depth, applying it to both origin and destination coordinates, and comparing the results. Though this is an elegant algorithm, repeatedly left-shifting bits by arbitrary integers proved expensive. By precomputing child_bit_depth[d] = 1 << (max_depth - depth - 1), and left-shifting by constants, we experience noticeably better performance. Then, we compute the target child octant with binary & and integer inequality operations. We &-mask this value with the destination coordinates and bit-shift by constants corresponding to the X,Y and Z components. This yields the 0-7 octant offset of the child, and hence its index. We return the scalar value when we encounter a leaf; either a scalar leaf in an interior node, or a voxel within a cap node. Details can be found in Appendix B.1.

Neighbor Finding. Given an origin node and a coordinate direction to a desired destination node, neighbor-finding entails recursion up the octree until we find a parent containing both nodes. Frisken and Perry [4] propose using the aforementioned depth mask with an exclusive "or" to determine immediately adjacent neighbors. However, we find that this no cheaper than simply &-masking both source and destination, and performing integer equality on the results. When a common parent is found, the neighbor-finding function then relies on point location to find the leaf at the given destination. The result is an arbitrary neighbor-finding algorithm for potentially non-adjacent neighbors. To minimize the memory footprint of the octree, we chose to omit parent pointers from the nodes of our octree. Effectively, recursing up the octree requires knowledge of parent indices. We provide this in the ray traversal algorithm itself, which fills a index_trace[] array containing the indices of all parents nodes for a given cap. Pseudocode for neighbor-finding is given in Appendix B.2.

4.3 Ray-Octree Traversal

Finally, we approach the problem of adapting a ray traversal scheme to our octree structure and its given hashing scheme. After experimenting with the methods of Sung [19] and Samet [17], the fastest traversal that emerged most resembled the technique of Gargantini and Atkinson [5]. The traversal is similar to that of a kd-tree, with splits along the X,Y and Z mid-planes of each node. Gargantini and Atkinson proposed fully sorting child octants by the order of their traversal; this is the approach we take (Figure 6). We optimize it to exploit binary arithmetic on integer octree-space coordinates, similar to our neighbor-finding and point-location implementations.

Rays are generated in canonical octree space on $[0,2^{d_{max}}]$, so no additional transform is required. We first perform a standard ray-bounding box test to discard rays that never intersect the volume. This test yields a tenter and a texit for the root node of the octree, which we pass to our recursive traversal algorithm.

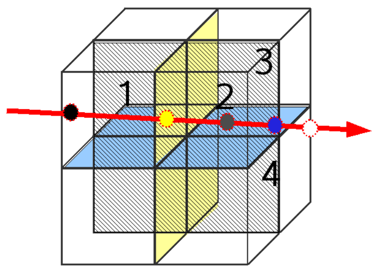

Figure 6: *Single ray traversing an octree node.* The traversal algorithm, finds the order of intersection of the X (yellow), Y (blue), and Z (gray) mid-planes. As we already know the entry (black) and exit (white) intersection points, we have the exact order of traversal of child octants.

Interior nodes. The single-ray traversal first retrieves the octree node given by depth and node_index. Then, it computes the octree-space coordinates of the mid-planes (Figure 6) that divide the child octants of this node. The computation-heavy section of the traversal involves evaluating penter and sorting the tcenter intersection distances in a separate array axis_isects[]. We use that array to sequentially march across the child octants in the correct order of their traversal. The algorithm has moderate initial cost associated with computing and sorting the mid-plane intersections; afterwards traversing the child octants is trivial. The first child octant is computed using the same constant shifting and binary-or as point location; aftwards moving from one octant to the next merely requires inversion of the bitmask (axisbit) along the corresponding mid-plane axes traversed. Pseudocode is provided in Appendix C.

Our structure requires special traversal routines for scalar leaves and cap nodes. Exact details are left as an exercise for the reader; however, both are similar to interior node traversal in Appendix C.

Cap nodes. Cap intersection is identical to that of interior nodes, except for the block of code checking the isovalue against the min/max range and recursively calling the child traversal routine (Appendix C). In its place, we determine the values eight forward-neighbor voxels (Figure 4). Before resorting to neighbor finding, we observe that given a voxel of interest intersected by a ray octree structure, anywhere from 1 to 8 voxels in this neighborhood will lie within the same cap node. Specifically, given the 0-7 child octant child, and a 0-7 direction "dir" to a desired neighbor, we simply check if (child & dir). If this evaluates false, the neighbor is simply cap.scalars[dir_idx]. If it is true, we proceed with neighbor-finding to retrieve the value.

Scalar leaves. A scalar leaf is traversed recursively to the same depth as caps, even though it has no children and homogeneous value. When the traversal reaches cap depth, if the traversal encounters a neighborhood of identical voxels within the scalar leaf, we know that no isosurface is encountered. Otherwise, at the borders of the scalar leaf node, we perform neighbor-finding as we do for cap nodes.

Once we have the eight voxel values, we check that our isovalue lies within their minimum and maximum. If it does, we perform the isosurface intersection with the 8 voxels as corners of the cell.

4.4 Isosurface Intersection

To compute the ray-isosurface intersection, we seek a surface inside a three-dimensional cell with given corner values (Figure 4), such that trilinear interpolation of the corners yields our desired isovalue. We can find where a ray instersects this surface by solving a cubic polynomial. Specifically, the hit position is given by evaluating the ray at the first positive root of that ray's polynomial. While the same recipe is generally used to generate the four coefficients of the polynomial, various techniques exist for finding the root.

Our implementation uses the same approach as the Neubauer iterative root finder proposed by Marmitt et al. [13]. Here, a ray is iteratively re-parameterized into sub-intervals within the cell in question, until a sign change is detected within the sub-interval and a root is found. Compared to the analytical root finder based on Schwarze's cubic solver [18] used by Parker et al. [15], it is slightly faster and yields single-precision, numerically stable results.

4.5 Shading and Filling the Frame Buffer

While ray tracing delivers great flexibility in per-pixel shading methods, we are mostly interested in fast ray casting of the isosurface. Thus, our results show simple Lambertian shading with no shadows.

The traversal itself does not employ packets, however we use a packet architecture for ray generation and shading. We do not defer normal computation due to the prohibitive cost of storing each cell per ray, or repeating the neighbor-finding process. However, the packet architecture allows diffuse shading to be performed in batch, which likely delivers some speedup over a more conventional single-ray tracer.

5 Results

5.1 Data Compression

Lossless octree compression by consolidating voxels with zero variance commonly yields a compression factor of 3 to 5, depending on the spread of isovalues within the data. In general, large data yield higher compression benefit than small data (Table 1). Additional compression can be achieved by segmenting the data into iso-ranges of interest. For example, if we are mostly interested in isovalues from 64 to 127 in the Richtmyer Meshkov data, we can clamp scalars outside that range to those limiting values. As we see in Table 1, this allows us to losslessly compress a sizeable range of isovalues from the LLNL data into under 2 GB. Furthermore, if lossy compression is acceptable, one can more aggressively consolidate inter-voxel variance. This could be desirable for large data that varies gradually in space. The effect would be to further quantize isovalues, and deliver extra compression.

DATA	ISO-RANGE	TIME STEP	SIZE original grid	SIZE octree volume	%
heptane	full	70	27.5M	3.96M	14
heptane	full	152	27.5M	9.5M	33
heptane	full	0-152	4.11G	678M	16
llnl	full	50	8.0G	687M	8.5
llnl	full	150	8.0G	1.89G	25
llnl	full	270	8.0G	2.48G	30
llnl	64-127	270	8.0G	1.81G	22
CThead	full		14.8M	12.4M	84

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 computed in bytes, and include all features of the octree, including the embedded min/max tree overhead.

The compression achieved by the octree depends entirely on the inter-voxel variance of the volume at large. When large regions of a volume are uniform in value, and "interesting" isosurfaces lie within a relatively narrow spatial region, octree compression delivers impressive results. Conversely, volumes with uniformly high variance yield little consolidation; due to the overhead of the octree hierarchy they could potentially occupy greater space than the original 3D array. The latter is the case with the UNC CTHead data, which has inherent measuring noise. Fortunately, large volume data from fluid or mechanical simulations behave more like the former, thus benefit greatly from octree volume compression (Table 1).

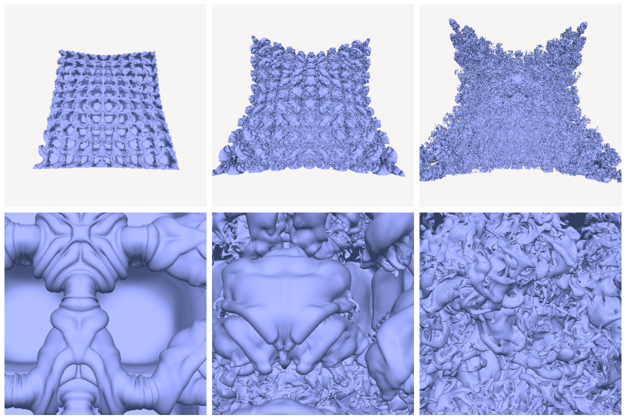
Figure 7: *The LLNL Richtmyer-Meshkov data.* Various scenes with an isovalue of 20. Top row, from left to right: timesteps 50, 150, and 270. Bottom: same timesteps, with a closer camera.

5.2 LLNL Richtmyer Meshkov

We consider the frame-rate performance across several timesteps of the LLNL Richtmyer-Meshkov instability field, a 2048x2048x1920 fluid dynamics dataset. Using octree compression we are able to render this volume at multiple frames per second on a 32-bit laptop; however for an indicator of performance on future multicore CPUs we benchmark fully interactive rates on a 16-node non-uniform memory access (NUMA) workstation of 8 dual-core 2.4 GHz AMD Opterons. For volumes as complex as the LLNL data, it is perhaps preferable to render a 1024^2 frame.

SCENE	CORE DUO-512^2	NUMA-512^2	NUMA-1024^2
50, far	3.6	25.8	7.4
150, far	2.8	20.0	5.7
270, far	2.4	17.5	4.7
50, close	2.1	15.4	4.3
150, close	1.8	14.2	3.6
270, close	1.7	13.6	3.5

Table 2: *Frame rates of various time steps of the LLNL 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 Figure 7 for images.

The results on the LLNL data are competitive: even on the Core Duo laptop, frame rates remain above 2 fps for most camera positions. Results on the Core Duo at timestep 270 actually exceed those achieved by DeMarle et al. [3] on a cluster of 32 PC's, albeit with a distributed shared memory system. They also perform on par with the Wald et al. [21] coherent kd-tree system, which reported around 1 fps on a dual 1.8 GHz Opteron at 640x480 for scenes similar to our far camera image.

5.3 Comparison to Hierarchical Grid

To gauge the performance of our octree traversal algorithm, we compare it to the performance of the Parker hierarchical grid on the same data. We first consider the performance of each as an acceleration structure only, with both methods retrieving their data directly from the uncompressed original 3D array. The octree performs fairly well, albeit not as fast as the grid. Next, we compare grid and octree performance when looking up octree data via neighbor-finding. The octree surprisingly performs better than it did on array data, likely due to cache behavior on large data. The grid performs top-down point location for the first lookup, and subsequently uses neighbor-finding; its results on octree data are noticeably slower. The results demonstrate that for rendering octree data, traversing the same min/max octree encapsulating that data yields a distinct advantage.

DATA	FPS	
	macrocell grid	octree
3D array	18.9	15.7
octree volume	8.0	**17.5**

Table 3: *Octree-grid comparison.* Frame-rates for the same scene, traversed by our octree or a 5-deep hierarchical macrocell grid; using either uncompressed 3D array data or compressed octree data. Tests performed on the LLNL data at timestep 270, on a 16-core NUMA 2.4 GHz Opteron workstation. The octree traversal with octree data performs nearly as fast as the hierarchical grid with uncompressed array data.

5.4 Scalability

While ray tracing is inherently parallel, complicated memory access could potentially compromise scalability on a shared memory or NUMA architecture. Thus, it is worth demonstrating that our technique scales well to multiple processors. Figure 8 demonstrates an efficiency of 91% with 16 processors, which behaves similarly to uncompressed 3D array volumes using the macrocell grid. Once again, the macrocell grid performs slightly faster, but without the benefit of compressed data.

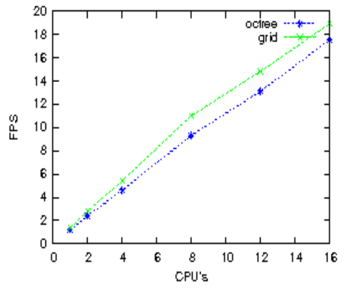

Figure 8: *Scalability.* Scalability of our technique on 1,2,4,8,12 and 16 threads, on a 2.4 GHz Opteron NUMA workstation with the LLNL 270 far scene at 512^2. The slight change in slope at 8 threads corresponds to the use of local NUMA memory by two cores instead of one. This demonstrates that our octree technique scales as well as the hierarchical grid with uncompressed data.

5.5 Time-Variant Volumes

One limitation of GPU volume rendering is that, for time-variant volumes, GPU memory restricts the number of timesteps that can be stored and rendered in-core. Bus bandwidth prevents a GPU from streaming textures as effectively as geometry from the CPU. With octree volumes, we can compress full sequences of medium-sized (less than 512^3) time variant data to fit within limited CPU main memory. The dataset in Figure 9 contains 153 timesteps, each of which would occupy 27.5 MB for a total of 4.11 GB. With octree compression, we compress the entire dataset in 678 MB, and render at multiple frames per second on a laptop (Table 4).

Octree volumes are useful in that they allow data such as the LLNL to be visualized on machines with limited main memory. However, even in a workstation with 64 GB RAM, memory is a precious commodity. Compression would permit multiple timesteps of the LLNL data to be stored and rendered interactively in sequence.

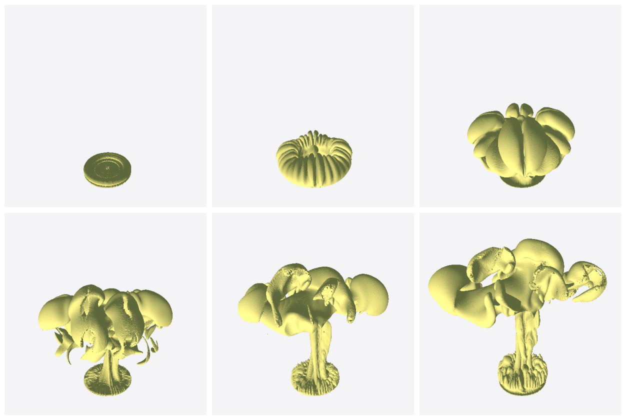

Figure 9: *Time-variant volume data.* Utah CSAFE heptane simulation, a 302^3 volume. The full sequence of 153 timesteps is stored in 678 MB as opposed to 4.1 GB uncompressed, permitting residency in main memory. We illustrate six timesteps from this sequence, at an isovalue of 42.

TIME STEP	CORE DUO-512^2	NUMA-512^2	NUMA-1024^2
25	17.0	87.1	29.2
50	11.1	60.3	18.0
75	5.7	36.7	9.6
100	4.1	26.6	6.6
125	3.5	28.0	7.1
150	3.2	23.1	6.3

Table 4: *Frame rates for the CSAFE heptane 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 Figure 9 for images.

6 Conclusion and Future Work

We have presented an octree volume format and traversal technique that allows for accelerated ray tracing of compressed data. Our method allows for interactive exploration of large structured data on multicore computers using a fraction of the original memory footprint. Compressing volumes into octrees allows us to visualize data locally with the same quality as uncompressed arrays. While other spatial structures could deliver greater compression or faster traversal, the octree strikes a particularly good balance of these goals.

Our traversal is highly dependent on a fast octree hashing scheme. Our contributions in ray traversal and min/max tree construction are designed for this application alone; however, the point location and neighbor-finding implementations extend to general use of a binary hash tree. While benchmarking other applications of octree hashing falls outside the scope of this paper, our routines showed speedups over the code proposed by Frisken and Perry [4].

Octree ray tracing is not necessarily the ideal solution for general-purpose volume rendering. For smaller volume data with uniformly high isovalue variance, an octree can actually occupy more space than a 3D array; moreover, the uniform grid and coherent kd-trees would likely outperform the octree for such scenes. However, in these cases 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. Moreover, as large data is often the impetus for ray tracing volumes in the first place, this method is highly appropriate for its particular application.

Future work will involve exploiting the multiresolution nature of the octree to provide a dynamic, view-adaptive level of detail scheme. Such a system would reduce the complexity and variance of the overall scene. In conjunction with coherent packet traversal, this could deliver dramatic speedups, as coherent methods have shown order-of-magnitude better performance than single-ray on low-variance scenes. Rather than isosurfacing, we might experiment with simplified direct volume rendering techniques to achieve smoother results.

Overall, hardware trends favor ray tracing large volumes using methods similar to this. Doubling each dimension of a 3D grid entails a factor of eight increase in memory footprint; this all but guarantees that main memory will continue to be a scarce resource in large volume rendering. Moreover, as multicore CPUs become increasingly prevalent, the degree of interactivity on mobile machines will rise to the levels delivered by today's SMP workstations.

References

[1] Imma Boada, Isable Navazo, and Roberto Scopigno. Multiresolution Volume Visualization with a Texture-Based Octree. *The Visual Computer*, 17(3), 2001.

[2] Herve Bronnimann and Marc Glisse. Cost-optimal trees for ray shooting. In *Proceedings of the Latin American Symposium on Theoretical Informatics*, 2004.

[3] David E. DeMarle, Steve Parker, Mark Hartner, Christiaan Gribble, and Charles Hansen. Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE Symposium on Parallel and Large-Data Visualization and Graphics (PVG)*, pages 87–94, 2003.

[4] Sarah F. Frisken and Ronald N. Perry. Simple and Efficient Traversal Methods for Quadtrees and Octrees. *Journal of Graphics Tools*, 7(3), 2002.

[5] Irene Gargantini and H.H. Atkinson. Ray Tracing an Octree: Numerical Evaluation of the First Interaction . *Computer Graphics Forum*, 12(4):199–210, 1993.

[6] Andrew S. Glassner. Space Subdivision For Fast Ray Tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[7] Vlastimil Havran. A Summary of Octree Ray Traversal Algorithms. *Ray Tracing News*, 12(2), 1999.

[8] Ben Hutchison, Eric Haines, Hanan Samet, and Erik Jansen. Octree Traversal and the Best Efficiency Scheme. *Ray Tracing News*, 12(1), 1999.

[9] Joe M. Kniss, Aaron Lefohn, Robert Strzodka, Shubhabrata Sengupta, and John D. Owens. Octree Textures on Graphics Hardware. In *Proceedings of ACM SIGGRAPH 2005 Conference Abstracts and Applications*, August 2005.

[10] Marc Levoy. Efficient Ray Tracing for Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[11] Yarden Livnat and Charles D. Hansen. View Dependent Isosurface Extraction. In *Proceedings of IEEE Visualization '98*, pages 175–180. IEEE Computer Society, October 1998.

[12] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 21(4):163–169, 1987.

[13] Gerd Marmitt, Heiko Friedrich, Andreas Kleer, Ingo Wald, and Philipp Slusallek. Fast and Accurate Ray-Voxel Intersection Techniques for Iso-Surface Ray Tracing. In *Proceedings of Vision, Modeling, and Visualization (VMV)*, pages 429–435, 2004.

[14] Ajith Mascarenhas, Martin Isenburg, Valerio Pascucci, and Jack Snoeyink. Encoding Volumetric Grids For Streaming Isosurface Extraction. In *3D Data Processing, Visualization and Transmission*, pages 665–672, September 2004.

[15] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization*, pages 233–238, October 1998.

[16] Alexander Reshetov, Alexei Soupikov, and Jim Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transaction of Graphics*, 24(3):1176–1185, 2005. (Proceedings of ACM SIGGRAPH).

[17] Hanan Samet. Implementing ray tracing with octrees and neighbor finding. *Computers and Graphics*, 13(4):445–60, 1989.

[18] Jochen Schwarze. Cubic and Quartic Roots. In Andres Glassner, editor, *Graphics Gems*, pages 404–407. Academic Press, 1990.

[19] Kelvin Sung. A DDA octree traversal algorithm for ray tracing. In Werner Purgathofer, editor, *Eurographics '91*, pages 73–85. North-

Holland, September 1991.

[20] Francisco Velasco and Juan Carlos Torres. Cell Octree: A New Data Structure for Volume Modeling and Visualization. *VI Fall Workshop on Vision, Modeling and Visualization*, pages 665–672, 2001.

[21] Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek, and Hans-Peter Seidel. Faster Isosurface Ray Tracing using Implicit KD-Trees. *IEEE Transactions on Visualization and Computer Graphics*, 11(5):562–573, 2005.

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

[23] Rüdiger Westermann, Leif Kobbelt, and Tom Ertl. Real-time Exploration of Regular Volume Data by Adaptive Reconstruction of Iso-Surfaces. *The Visual Computer*, 15(2):100–111, 1999.

[24] Jane Wilhelms and Allen Van Gelder. Octrees For Faster Isosurface Generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

.

A OCTREE VOLUME STRUCTURE

An octree volume consists of the following structure: interior nodes are stored in an array indexed by depth, from root depth 0 to depth $d_{max}-2$. "Cap" nodes exist at $d_{max}-1$. For the hashing scheme, we cache an array, child_bit_depth[d] $= 1 <<$ max_depth - d - 1.

```
struct OctreeData
{
  OctNode* nodes[max_depth];
  OctCap* caps;
  int child_bit_depth[max_depth];
};

struct OctNode
{
  T child_scalars[8];   //scalar leaves
  T child_mins[8];      //min/max tree
  T child_maxs[8];
  unsigned int child_start;  //base pointer to children
  char child_offset[8];      //offset from base
};

struct OctCap
{
  T scalars[8];
};
```

B OCTREE HASHING

Our octree hash scheme consists of acclerated routines for point location and neighbor finding in canonical octree coordinates, $[0, d_{max}]$. While binary arithmetic on integers is not a new hashing scheme [6, 4], we propose caching the depth masks to avoid costly arbitrary left shifts, and then shifting by constants.

B.1 Point Location

Point location algorithm. We use a cached table, child_bit_depth[], to avoid arbitrary left-shift operations.

```
T point_locate(Vec3i dest, int depth, int index)
{
  for(;;)
  {
    OctNode& node = octdata.nodes[depth][index];
    int child_bit = octdata.child_bit_depth[depth];
    int child = (dest.x & child_bit!=0) << 2
             || (dest.y & child_bit!=0) << 1
             || (dest.z & child_bit!=0);

    if (node.child_offset[child] == -1)
    {
      return node.child_scalars[child];
    }
    else if (depth == octdata.max_depth  2)
    {
      index = node.child_start + node.child_offset[child];
      child = (dest.x & 1)<<2 |
              (dest.y & 1)<<1 |
              (dest.z & 1);
      return caps[index].child_scalars[child];
    }
    index = node.child_start + node.child_offset[child];
    depth++;
  }
  return 0;
}
```

B.2 Neighbor-Finding

Neighbor finding algorithm. Given start coordinates, destination coordinates, and the octree depth of the start coordinates, we back-trace up the octree and then perform point location to retrieve a neighbor. index_trace contains pointers to nodes, so we only need store 1-way pointers in our tree.

```
T neighbor_find(Vec3i start, Vec3i dest, int depth,
                int parent_trace[])
{
  for(int up=depth; up >= 0; up--)
  {
    int child_bit = octdata.child_bit_depth[up];
    if ((dest.x & child_bit) == (cell.x & child_bit)
         && (dest.y & child_bit) == (cell.y & child_bit)
         && (dest.z & child_bit) == (cell.z & child_bit)
      return point_locate(dest, up, parent_trace[up]);
  }
  //root node
  if ((dest.x & child_bit) == (cell.x & child_bit)
       && (dest.y & child_bit) == (cell.y & child_bit)
       && (dest.z & child_bit) == (cell.z & child_bit)
    return point_locate(dest, 0, 0);
  return 0;
}
```

C RAY-OCTREE TRAVERSAL

Pseudocode for a ray traversal through an interior node of an octree volume. For brevity, some operations are omitted; those are bracketed with a brief description. Traversals of scalar leaf nodes and cap nodes operate similarly.

```
bool traverse(Ray ray,
     int depth, uint node_index,
     int parent_trace[], Vec3f cell,
     float tenter, float texit)
{
  OctNode& node = octdata.nodes[depth][node_index];
  parent_trace[depth] = node_index;
  int child_bit = octdata.child_bit_depth[depth];
  Vec3f center = Vec3f( cell | Vec3i(child_bit) );
  Vec3f tcenter = (center  ray.orig) / ray.dir;
  Vec3f penter = ray.orig + ray.dir * tenter;
  Vec3i child_cell = cell;
  Vec3i tc;
  tc.x = (penter.x >= center.x);
  tc.y = (penter.y >= center.y);
  tc.z = (penter.z >= center.z);

  int child = tc.x << 2 | tc.y << 1 | tc.z;
  child_cell.x |= tc.x ? child_bit : 0;
  child_cell.y |= tc.y ? child_bit : 0;
  child_cell.z |= tc.z ? child_bit : 0;

  Vec3i axis_isects;
  {perform 3-way minimum of tcenter such that axis_isects
  contains the sorted intersection of with the X,Y,Z
  octant mid-planes}

  const int axis_table[] = {4,2,1};

  float child_tenter = tenter;
  float child_texit;
  for( {all valid axis_isects[i] while tcenter < texit} ; i++)
  {
    child_texit = min(tcenter[axis_isects[i]], texit);
    if (isovalue >= node.child_mins[child] ||
        isovalue <= node.child_maxs[child]){
       //traverse scalar leaf, cap or node
       if (node.child_offset == -1)
         if (traverse_scalar_leaf(...)) return true;
       else if (depth == octdata.max_depth  2)
         if (traverse_cap(...)) return true;
       else
         if (traverse(ray,depth+1,parent_trace,
            child_cell, child_tenter, child_texit))
            return true;
    }
    if (child_texit == texit)
       return false;
    child_tenter = child_texit;
    axisbit = axis_table[axis_isects[i]];
    if (child & axisbit){
       child &= ~axisbit;
       child_cell[axis_isects[i]] &= ~child_bit;
    }
    else{
       child |= axisbit;
       child_cell[axis_isects[i]] |= child_bit;
    }
  }
  return false;
}
```