*SCI Instituite, Technical Report No UUSCI-2007-003*

# Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes

Ingo Wald
SCI Institute
University of Utah

Heiko Friedrich
Computer Graphics Group
Saarland University

Aaron Knoll
SCI Institute
University of Utah

Charles D Hansen
SCI Institute
University of Utah

**Abstract**— We describe a system for interactively rendering isosurfaces of tetrahedral finite-element scalar fields using coherent ray tracing techniques on the CPU. By employing state-of-the art methods in polygonal ray tracing, namely aggressive packet/frustum traversal of a bounding volume hierarchy, we can accomodate large and time-varying unstructured volume data. In conjunction with this efficiency structure, we introduce a novel technique for intersecting ray packets with tetrahedral primitives. Ray tracing is flexible, allowing for dynamic changes in isovalue and time step, visualization of multiple isosurfaces, shadows, and depth-peeling transparency effects. The resulting system offers the intuitive simplicity of isosurfacing, guaranteed-correct visual results, and ultimately a scalable, dynamic and consistently interactive solution for visualizing unstructured volumes.

**Index Terms**—Ray Tracing, Isosurfaces, Unstructured meshes, Tetrahedra, Scalar Fields, Time-varying data.

## 1 Introduction

Visualization of large unstructured data is a persistent challenge for data analysis. Due to its adaptive nature and simplicity, finite element (FE) analysis has experienced widespread adoption in simulations for numerous computational scientific and engineering disciplines such as CFD, meteorology, geology, and astronomy. With increasingly sophisticated simulation techniques and powerful parallel computing environments, the effective size of finite element fields is quickly outpacing the memory capacity of commodity graphics processors (GPUs). Nonetheless, scientists generally desire accurate visualization of these data sets in their entirety, with few, if any, compromises. Ideally, the visualization system should allow the user to interactively change the viewpoint, light sources, isovalues, time steps, etc.

A conventional method of rendering isosurfaces of volume data has been extraction via marching cubes or marching tetrahedra, followed by Z-buffer rasterization on GPU hardware. While more than adequate for small data, this approach faces difficulties for large, high-frequency volumes, where significant amounts of geometry must be extracted to faithfully reproduce a surface. View-dependent and multiresolution extraction methods can reduce the amount of geometry, but ultimately extraction is bound by geometric complexity.

Recent techniques for rendering unstructured data have leveraged the power of GPU hardware, applying direct volume rendering (DVR) techniques to depth-sorted tetrahedra. Large data has been addressed through multiresolution and progressive rendering techniques, as well as out-of-core mechanisms. While powerful, these methods incur limitations, as interactivity is realized through simplification or temporary omission of the full data set. Conversely, ray tracing methods on CPU workstations can directly address large memory, and are inherently scalable to multiple processors and large data.

Multi-core CPU's are increasingly prevalent. Large-scale multi-core architectures, such as Terascale [13], are clearly on the horizon. Current cc-NUMA workstations support 16 to 32 cores, and can directly address nearly two orders of magnitude more memory than a GPU. Algorithmic flexibility and SIMD instructions on the CPU encourage coherent ray tracing techniques, which amortize the costs of acceleration structure traversal and primitive intersection across multiple rays. Unstructured tetrahedral volumes encourage adaptive acceleration structures, such as bounding volume hierarchies, that have proven efficient for dynamic triangle mesh ray tracing. Isosurfaces for first-order FE are inherently polygonal, allowing for fast ray tracing via simple geometric intersection tests.

In this paper, we propose a new approach to directly ray tracing isosurfaces defined over tetrahedral domains by combining recent advancements in polygonal ray tracing with existing techniques for unstructured isosurface extraction. We detail a novel packet-tetrahedron intersection algorithm inspired by marching tetrahedra, and its integration with a coherent implicit bounding volume hierarchy traversal. We extend this technique to practical shading and visualization features such as multiple transparent isosurfaces and dynamic shadows. Ultimately, we find that ray tracing unstructured data on the CPU allows for interactive performance on current laptop hardware, flexible and correct visualization of isosurfaces, and the ability to render large time-varying unstructured data, limited only by the size of CPU main memory.

## 2 Related Work

### 2.1 Isosurface Extraction

Marching cubes were first applied to isosurface extraction of structured data by Wyvill et al. [37], and Lorensen & Cline [19]. Doi & Koide [8] developed a similar and arguably simpler algorithm based on marching tetrahedra for isosurfacing unstructured scalar fields. Nonetheless, naïve extraction of surfaces is bound by data complexity, and often slow. Recent works have accelerated marching tet extraction on the GPU. Pascucci [23] showed that the vertex processor of can be utilized to create appropriate quadrilaterals for the isosurface within a

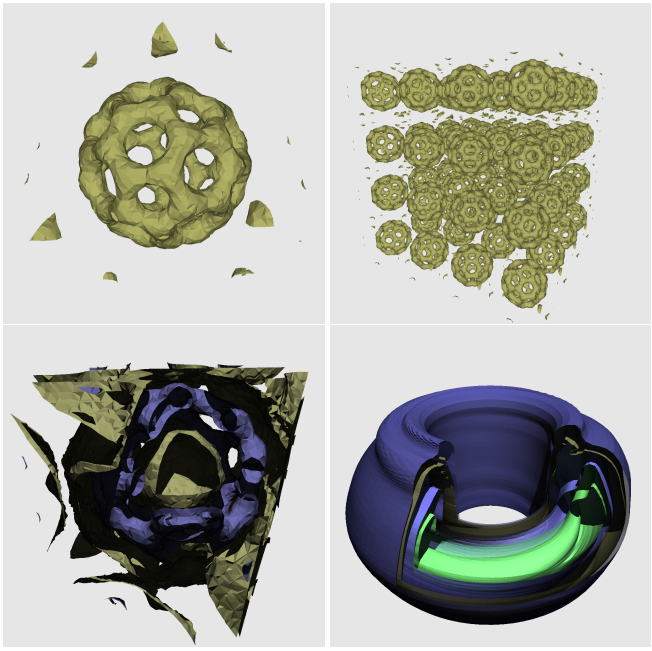

Fig. 1. Several samples of our interactive system running at $1024 \times 1024$ pixels: a) bucky ball (225K tets) with simple shading. b) bucky cube (11.3m tets). c) buckyball with two isosurfaces, clip box, and shadows, d) Time step 60 of the time-varying fusion data set (3m tets, 116 time steps), rendered with four isosurfaces, clip box, shadows, and transparency. With a $1024 \times 1024$ frame buffer, these examples render at 12.2, 2.7, 5.4, and 0.8 fps, respectively, on a Intel Duo 2.33 GHz laptop with 1 GB RAM; and and 90, 19, 42, and 7 fps, respectively, on a 16-core 2.4 GHz Opteron workstation with 64 GB RAM.

tetrahedron. Similarly, Klein et al. [14] exploit fragment programs for their quadrilateral computation. These GPU approaches yield overall rendering frame rates from 1 fps for million-tet data to 60 fps for smaller data sets. Though not implemented for dynamic unstructured extraction, techniques exist to improve performance on complex geometry, such as view-dependent frustum culling [18], adaptive extraction [34], and implicit occlusion culling [24].

### 2.2 Unstructured Volume Rendering

Garrity [9] first applied ray casting to unstructured meshes, by computing the entry and exit points of each ray with a face of the tet mesh, and accumulating opacity as in volume ray casting. Shirley & Tuchman presented an approach similar to splatting, based on rasterization of depth-sorted projected tetrahedra (PT). Due to the power of rasterization hardware, methods involving projection and sorting have become popular, such as vertex shader methods for performing PT classification [36]. Nonetheless, GPU ray casting approaches such as Weiler et al. [33], and Bernardon et al. [1] have proven feasible, thanks to clever mapping of geometry structures and the ray-tet marching algorithm to GPU fragment shaders. However, these implementations deliver barely-interactive frame rates for even moderate datasets of 1 million tets. Callahan et al. [5] proposed an extremely efficient GPU method of partially ordering projected tet fragments by depth in both image and object space. The HAVS method has been extended to handle large data using LOD [4], progressive rendering, and out-of-core streaming [3]. Their system allows for direct volume rendering of unstructured data at real-time rates, albeit with minor artifacts and delayed full visualization of large data.

### 2.3 Interactive Ray Tracing on the CPU

Instead of using rasterization techniques, our system builds on fast ray tracing. Interactive ray tracing was first proven feasible on commodity CPU's by Wald et al. [32], using SIMD instructions on coherent ray packets in a kd-tree. More aggressive coherent methods involve culling geometry outside the packet bounding frustum (e.g. Dimitriev et al. [7]), or frustum traversal of wide packets (e.g. Reshetov et al. [25], or Wald et al. [29]), both of which ideas we will employ. Ray tracing today can easily trace millions of rays on desktop PCs, and animated scenes (the counterpart to time-varying data) have successfully been addressed [17, 29, 31]. Of particular interest to our approach is the *dynamic BVH traversal* proposed by Wald et al. [29].

### 2.4 Interactive Isosurface Ray Tracing

Isosurface ray tracing on the CPU has been explored before, particularly for large data applications. Parker et al. [22] employed a hierarchical grid to ray trace isosurfaces on a small supercomputer; DeMarle et al. [6] extended this implementation to clusters. Knoll et al. [15] proposed losslessly compressed octree volumes for rendering larger data. Wald et al. [30] showed how coherent optimizations could be applied to ray trace isosurfaces interactively on small workstations, using implicit min-max kd-trees; our method is heavily inspired by this work. Marmitt & Slusallek [20] proposed a new ray marching algorithm for directly traversing tet meshes using Plücker coordinates. Optimized coherent ray tracing has not yet been applied to unstructured isosurfacing.

## 3 Coherent Ray Tracing of Tetrahedral Isosurfaces

Our core approach to ray tracing unstructured scalar fields is an implicit dynamic bounding volume hierarchy in the spirit of implicit kd-trees [30], combined with aggressive large-packet coherent ray traversal; and a specially designed packet-isopolygon intersection technique inspired by fast packet-triangle intersectors and the Marching Tetrahedra algorithm.

In unstructured grids, the scalar field is defined through linear interpolation over tetrahedral primitives; each such isotetrahedron can then contain one or more more isosurfaces given user-specified iso values. As with implicit kd-trees [30], we build a hierarchical data structure over these primitives such that each node in the hierarchy contains the minimum and maximum of the scalar field below that node's subtree; these isoranges can then be used during traversal to discard subtrees that cannot contain the isovalue. Instead of kd-trees, we opt for bounding volume hierarchies. In practice, they are at least as fast, equally efficient for time-varying data, and better suited to the irregular, overlapping geometry of unstructured volumes.

The implicit bounding volume hierarchy encourages a variation of the aggressive packet-frustum BVH traversal that was recently proposed for polygonal ray tracing [29]. This operates on much larger packets (typically 8x8 or 16x16 rays) than the 4-ray SIMD traversal proposed for implicit kd-trees, and uses frustum culling and speculative descent to minimize the number of ray-node traversal steps. Larger packets also imply better amortization of per-packet costs, and thus help in hiding the overhead induced through implicit culling. Since the implicit BVH is built over the space of all isovalues, the isovalue(s) of interest can be changed interactively any time, and even multiple isovalues can be trivially supported. A BVH also allows for easily updating the data structure once the scalar field or even vertex positions change, and thus allows for naturally supporting time-varying data.

When a packet reaches a leaf of the BVH, we intersect the isotetrahedra contained in that leaf using a new technique inspired both by marching tetrahedra [8] and fast packet-polygon tests. In both intersection and traversal, we will make heavy use of large-packet/frustum techniques recently developed in polygonal ray tracing. Unless otherwise specified, both intersection and traversal are assumed to operate on packets of $16 \times 16$ rays.

## 4 Isosurface Intersection

An isosurface is the implicit surface $f(\vec{x}) = v$ where a scalar field $f(\vec{x})$ takes on a given isovalue $v$. For conventional first-order finite elements, the scalar field is given as a tetrahedral mesh in which the scalar values specified at the vertices $A$, $B$, $C$, and $D$; the scalar field inside each *isotetrahedron*, or *isotet*, is defined by linear interpolation

$$f(\vec{x}) = f(\alpha, \beta, \gamma, \delta) = \alpha A + \beta B + \gamma C + \delta D,$$

where $\alpha, \beta, \gamma, \delta$ are the barycentric coordinates of $\vec{x}$.

To intersect a ray $\vec{x}(t) = \vec{o} + t\vec{d}$ with any isosurface $f(\vec{x}) = \mathbf{v}$ one can immediately substitute the ray equation into the linear interpolation, solve a linear system for $t$, and check that the solution lies within the isotet. However, we can also observe that for linear interpolation the isosurface must be planar. This plane is bounded by line segments along the edges of the isotet in which it exists, forming either a triangular or quadrilateral polygon as shown in the various cases of Marching Tetrahedra, and illustrated in Figure 3. We denote this polygon an *isopolygon* (or *isopoly*), as it represents the base geometric primitive we seek to ray-trace. Unlike solving the ray-parametrized implicit, this isopolygon must only be computed once per isotet traversed; that cost is amortized over all rays in the packet, and the full array of fast ray-polygon techniques can be applied.

### 4.1 Extracting the Isopolygon

To compute the plane equation and bounding edges of the isopolygon, we turn to the Marching Tetrahedra algorithm [8]. Vertices of the isopolygon lie on edges of the isotet, and isopolygon edges lie on the tet faces. Polygon vertices will lie only on those tet edges for which one vertex is greater and one is smaller than the isovalue. Having four vertices, there are only 16 cases for which a given vertex is either larger or smaller than the isovalue. For each of these cases, we can store however many vertices the resulting polygon will have, and the indices of the two tet vertices that span the edge on which that polygon vertex must lie. In SSE, this lookup is particularly simple: after loading the four vertices' isovalues into a SIMD register, an SSE

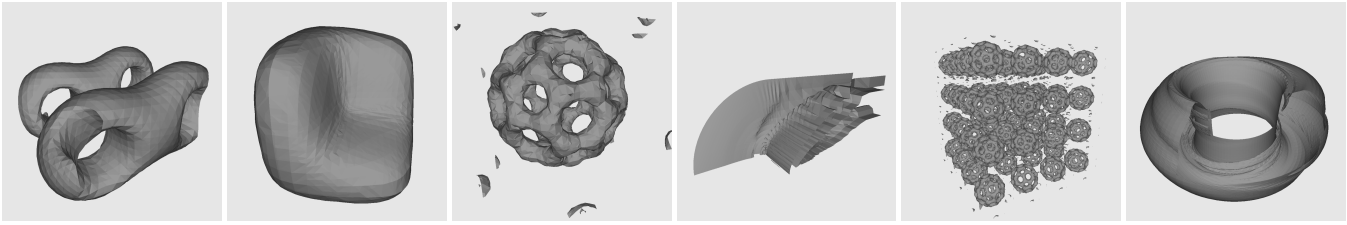

Fig. 2. From left to right: ell32P (149K tets), feok (122K), bucky ball (177K), bluntfin (225K tets, two isosurfaces), bucky cube (4x4x4 bucky balls, for a total of 11.3m tets), and time step 50 of the fusion data set. With simple shading, these examples run at 14.2, 12.6, 13.3, 18.9, 2.8, and 3.3 frames per second ($1024 \times 1024$ pixels) on a Intel Core 1 duo laptop with 1GB RAM, and at 95, 93, 90, 94, 19.1, and 26.1 frames per second on a 16-core 2.4 GHz Opteron workstation.

comparison followed by a `movemask` operation will return the desired case. The result is conveniently returned in a 4-bit integer (one bit for each comparison) that can be directly used to index into aforementioned table of 16 cases. Once we know which tet edges contains an isopolygon vertices, each isopoly vertex can be computed by linear interpolation along the two vertices of the corresponding tet edge.

### 4.2 Ray-Isopolygon Intersection

Once the vertices of our polygon are known, we can use an extension of Wald's triangle test [28] to intersect it. As shown in Figure 3 (left), ray-isopolygon intersection first computes the distance to the precomputed plane, then projects the ray hit point onto a suitable 2D coordinate plane. Here, each of the edges defines a (2D) half-space, which we orient to point towards the inside of the isopolygon. Since the isopolygon must be convex, we can then take the projected hit point and perform a 2D half-space test with each of the edges, and can reject the hit point as soon as any of these tests fails. This test can be performed efficiently for four rays in SSE for both triangle and quad cases.

### 4.3 SIMD Frustum Culling

In addition to fast SIMD intersection, we also apply conservative "full miss" and "full hit" tests for the entire packet, using packet frustum culling, e.g. [2, 7]. These tests require computation of the four corner rays bounding the packet frustum in SSE. For a given isopolygon, we can forgo individual ray intersections when all four bounding rays fail for the *same* 2D half-space test (Figure 3, right). Similarly, if all four rays pass all half-space tests, the entire packet passes through the triangle, and we must only compute perform a distance test for our component rays. Thus, intersection tests for individual rays are only required when the frustum neither fully misses nor fully hits.

The efficiency of frustum culling depends on the relative areas of the frustum and isopolygon within the plane. For complex scenes, tets are too small to have full hits, and frustum culling rarely succeeds.

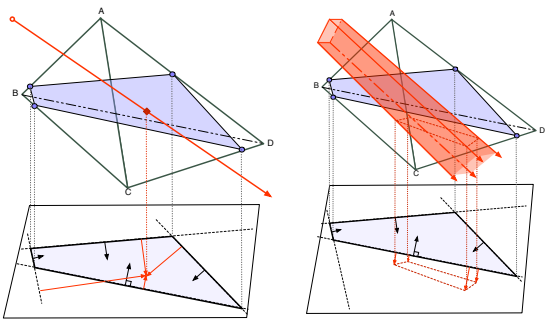


Fig. 3. *Ray-Isopolygon Intersection in an Isotetrahedron.* Knowing that the isosurface inside the tetrahedron is a plane, we first extract an isopolygon. We then compute the point where the ray pierces that polygon's supporting plane, and project both the polygon and that hit point to a 2D coordinate plane. In 2D, we then perform a point in (convex) polygon test by considering if the point is on each of the edges' positive half-spaces. The test can trivially be extended to support frustum culling: If all corner rays of the bounding frustum fail at the *same* edge, all the rays inside the frustum must fail.

However, full misses are quite common due to the loose nature of the implicit BVH, making this test highly effective overall. Typically, frustum culling can reject 40–60% of the packet-isopolygon tests, tough this ratio declines for larger models. Every times SIMD frustum culling rejects a packet test, 256 individual ray-isopolygon tests are avoided.

### 4.4 Isopolygon Pre-Computation

Isopolygon computation can be executed in three ways:

1. *Full pre-computation.* Pre-compute all isopolys every time the user changes the isovalue(s) of interest.
2. *On-the-fly computation* from scratch on demand.
3. *On-the-fly computation with caching.* Compute isopolys only when needed, but keep a cache of already computed isotets; clear the cache every time the user changes the isovalue(s) or time step.

Full precomputation maximizes performance for navigation with static isovalues, but requires larger memory footprint and incurs delays when the user changes isovalue or time step. On-the-fly computation is slower during rendering, but offers greater flexibility with scene interaction. Caching in theory offers a compromise, but in practice is quite complicated in a multi-core environment, as it requires the resolution of cache conflicts in a thread-safe manner, requiring significant synchronization overhead. We therefore opt for pure on-the-fly computation by default. Due to the use of large packets – which allow for amortizing the on-the-fly computations over 64 rays – the overhead is in the range of 5–8%, which we believe is a tolerable price for the ability to arbitrarily change the time step or isovalue.

## 5 The Implicit Bounding Volume Hierarchy

The concept of the implicit BVH is similar to that of the implicit kd-tree [30] in that the acceleration structure is not built for a single isovalue, but rather as a tree of min-max isovalue ranges (e.g. Wilhelms & Van Gelder [35]). Each node stores the minimum and maximum of all scalar field values contained within that subtree. During traversal, we can consequently cull all BVH nodes that do not contain our desired isovalue. Once built, the implicit BVH structure is valid for all isovalues, and thus allows for simultaneously rendering multiple isosurfaces from the entire range of isovalues. As subtrees that do not contain the isovalue are never traversed, the only effective cost of supporting arbitrary isovalues is a slightly looser-fitting BVH.

### 5.1 Building the BVH

Building an implicit BVH for tets in fact is similar to building a BVH for triangle meshes. Most mesh-BVH builds rely on bounding boxes or centroids of their primitives as construction metrics [27, 29], and tets behave similarly to triangles in this regard.

Traditional bottom-up BVH builds (e.g. [10]) generally result in inefficient BVHs [12]. Recent BVH literature has favored top-down builds, which recursively partition primitives into two subgroups. Two partitioning strategies are of particular interest: Wald et al.'s sweep surface

area heuristic (SAH) build [29], and Wächter et al.'s fast spatial median build as proposed in his bounding interval hierarchy paper [27]. The SAH build employs a *surface area heuristic* [10, 12] to select a partition with lowest expected cost, but is costly to build. The BIH-style build is closer in spirit to spatial median builds and, as it requires no cost function evaluation, it builds significantly faster than SAH methods. In both constructions, nodes are partitioned until leaves contain 12 or fewer tet primitives. Empirically, we have found this fixed value to work best.

**BVH Structure.** Our BVH node employs the same structure as [29], with a crucial modification: we interpret the isovalue $v$ as a 4th dimension of the bounding volume, leading to 4D bounds $\{x, y, z, v\}$. This can then be stored and processed as SSE vectors. Integers for the child node index and traversal bookkeeping follow, padded to ensure SSE-friendly 16-byte alignment. Storing isovalues alongside geometric extents allow all dimensions to be processed simultaneously in SSE.

### 5.2 Implicit BVH Traversal

Having constructed the implicit BVH, we now proceed to traversal. As previously mentioned, we employ the coherent traversal algorithm of Wald et al. [29], and extend it to implicit iso range culling. In general, this algorithm operates on large packets of rays, and tracks both a bounding frustum and the first "active" ray in the packet that intersects a current BVH node. Instead of intersecting each traversed node with *all* the rays in the packet, it employs optimizations such as speculative descent and frustum culling of nodes. With the implicit BVH, nodes not containing an isovalue in their min-max range are culled.

**I) Implicit culling.** At the heart of implicit BVH traversal lies the concept of culling subtrees that are known to be *inactive* – those whose isorange does not contain an isovalue. As this test is very cheap, we naturally perform it first. In addition, we observe that each active node must have at least one active child, and if the first child is inactive, we can proceed to its active sibling. Only at *bifurcation nodes* - where both children are active - do we actually revert to the geometric tests outlined below. In the worst case, this behavior causes us to descend several times into a subtree that is not actually visible. Since these speculative descents are fast, however, this is still quicker than testing all the nodes for visibility; and even *if* the fast descent led to a subtree that is outside the packet's bounding frustum, this node would be immediately rejected by the frustum test outlined below.

**II) Speculative first-active descent.** For our first geometric traversal test, we examine the first *active* ray in the packet. If that hits the current node, we can immediately descend without performing any more ray-box tests, as illustrated in Figure 6(a). Since we never test whether any of the other rays actually hit the current node, this test is speculative. Though it may cause modest extra work when few rays in the packet

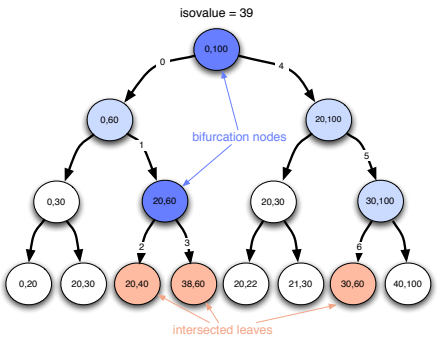


Fig. 5. *Implicit Culling.* The implicit BVH is a min-max tree containing only a subset of BVH nodes containing our desired isovalue(s). We can speculatively descend the min-max tree until we reach a leaf, or an intersection test fails. Only at bifurcation nodes (dark blue) must we resort immediately to geometric packet-BVH traversal computation. Thus, geometric tests are performed as if the BVH had only been built over active nodes for a single isovalue.

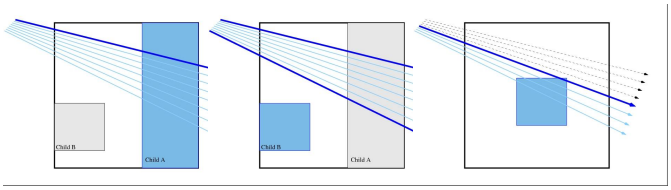


Fig. 6. *First-active descent, frustum test, and active ray tracking in BVH Traversal.* Given a BVH node, we speculatively test the first "active" ray in the packet against the, and immediately descend if it hits (a). If the first hit test failed, we perform a frustum test to reject nodes completely outside the frustum (b). If neither of these tests proved successful, we test all rays sequentially in a packet until one hits; rays that missed are deactivated for future traversal steps (c).

are also active, this strategy allows many ray-box tests to be skipped when numerous consecutive rays are active.

**III) Frustum test.** If the first active test fails, we know that the packet at least partially misses the box, and can perform a frustum test to conservatively determine if the entire packet misses. Technically we employ an interval arithmetic (e.g. $[2, 25]$) test instead of a geometric frustum test, but the effect is similar in behavior. If the full packet missed, we reject the current node and go to the next node on the stack (see Figure 6(b)).

**IV) First-active ray tracking.** If both the speculative descent and frustum tests fail, we test all remaining rays until we find the first active one that hits the current node. Those rays that failed the test are marked inactive by tracking the index of the first active ray in the packet (all rays with a smaller index are known to be inactive). If no active ray could be found, we reject the node and pop the next subtree from the stack. Rays with indices higher than the first active one we found are not tested, and are speculatively descended into the subtree as well.

**V) Leaf traversal.** When encountering a leaf, we first perform a frustum test as for all other nodes. If that test passes, we iterate over all the tets referenced in that node, then determine that tet's isorange (which may be smaller than the node's isorange), test that range, and finally either reject the tet or intersect it as described above.

## 6 Time-Varying Data

Time-varying data is extremely common in FE simulations. In the simplest time-varying tet meshes, geometry remains constant and only scalar values change. More complex scenarios include changing geometry and topology, and potentially dynamic addition and removal of elements from one time step to the next. To address these possibilities, we propose two schema for dynamic IBVH construction, balancing performance and memory footprint. Results are analyzed in Section 8.5.

### 6.1 Schema I: Unique BVH Per-Step

The naïve way of accommodating time-varying data is to compute a unique IBVH for each individual time step. Thus, no render-time computation is necessary to progress from one time step to the next, regardless of changes in geometry or scalar element values. As we operate completely in host memory, this approach is in fact very efficient. However, for large data sets with many time steps such as the fusion data set in Figure 4, this approach may require considerable amounts of memory.

### 6.2 Schema II: Dynamic Refitting

Fully computing a new BVH on-the-fly during rendering is too costly for large data, even using the fast BIH-style build. However, we observe that when tet mesh vertices change position but connectivity remains constant, the BVH structure will not change between time steps. Thus, simply refitting the nodes' bounding extents will yield a correct

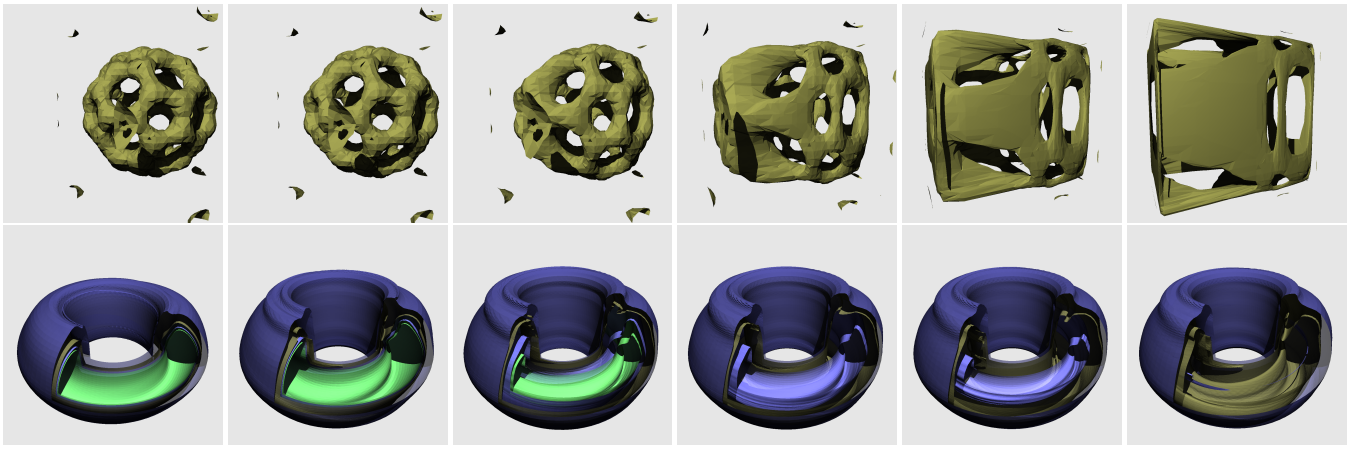

Fig. 4. Two examples of time-varying data sets, rendered at $1024 \times 1024$ pixels, using a 16-core 2.4 GHz Opteron workstation. Top: An artificially created deforming bucky ball that shows severe deformation of its 226K tets, running at 50+ frames per second including shadows from a point light source. Bottom: The fusion data set with a time-varying scalar field (3m tets, 116 time steps), rendered with four layers of isosurfaces, a crop box, shadows, and transparency, running at 7 to 15 frames per second. Camera and light positions, time step, and number and parameters of the isosurfaces can be changed interactively.

BVH. This technique has been successfully applied to ray tracing dynamic triangle meshes [17, 29]. The main drawback is that, particularly in cases of extreme geometric deformation, the refit BVH may perform worse than a BVH built from scratch for that particular time step. Fortunately, for tet meshes and our IBVH, this method works extremely well due to the continuous nature of tet deformations in FE simulation, particularly for rigid bodies. Moreover, when vertices remain constant but the scalar field changes, the BVH is identical for all time steps, as only the min-max isovalues must be updated.

As previously mentioned, minimum and maximum geometric bounds and isovalues are stored adjacently in 4D SSE vectors. Refitting the 4D extents can thus be accomplished with one SSE min and one SSE max per BVH node. Tet vertices and scalars are also stored as 4D points; thus computing the 4D bounds of a tet is also extremely efficient, requiring only 3 SSE min and max operations each per tet. With multi-core CPU's, it is straightforward to parallelize the update process. After the initial BVH has been built we find all the subtrees for a given level in the BVH hierarchy, and store their indices. During a refit, we can then update these subtrees in parallel. Once all subtrees are updated, a single thread refits the remaining few nodes close to the root node.

## 7 Shading and Interaction Modalities

Having leveraged these algorithms for efficient unstructured volume ray tracing, we describe several visualization modalities that can assist in understanding our data sets.

**Smooth Normals.** Since linear interpolation in tetrahedral meshes leads to piecewise-planar isosurfaces, the rendered isosurface has normal discontinuities where different tets abut, resulting in a faceted surface appearance. Instead of using the geometric surface normal for shading, we can achieve a smooth surface appearance by precomputing and interpolating vertex normals, with little additional cost. Though visually more pleasing, most scientists prefer seeing the data "as computed", so we disable this feature by default.

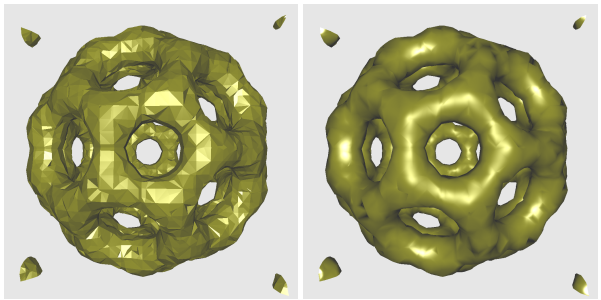

Fig. 7. Instead of shading with the surface normal, a smoother appearance can be achieved by precomputing and interpolating the tets' vertex gradients.

**Shadows.** A far more useful effect that a ray tracer can support is shadows, which can add important visual cues over an object's shape (see Figure 8). In casting shadow packets, rays are generally coherent and share a common origin in the case of point lights. Unlike primary rays, shadow rays do not inherently form a regular beam, and thus have no concept of "corner rays" for SIMD frustum culling. Though bounding corner rays could easily be computed from a packet [2], this is not yet implemented. Fortunately, frustum culling during BVH traversal still works for shadow rays using the Reshetov et al. [25] technique, which requires no actual geometric frustum. In theory, shadow rays are simpler than other rays, as they can be terminated as soon as *any* valid intersection is detected. These special cases, however, are not yet exploited. The overall speed impact of shadow rays varies, but is typically lower than $2\times$ (see Figure 8a-b).

**Multiple Isosurfaces.** Supporting multiple isosurfaces in an implicit BVH is straightforward, by simply testing whether a BVH subtree overlaps *any* of the isovalues before descending it. To follow the SIMD paradigm, we currently support up to four different isosurfaces, though it would be trivial to add more. Keeping the four isovalues in a SIMD vector, we can test when a BVH node's or isotetrahedron's iso range contains any of these four isovalues in parallel. These are in turn intersected with all the rays that actually hit the leaf node. Though rendering multiple surfaces can require tracing more rays per image, particularly when transparency is enabled, it causes no significant computation penalty in and of itself.

**Clipping Planes and Boxes.** While isosurfaces provide an intuitive way of visualizing a data set, one of their drawbacks is that the surface often occludes the data set's interior. For that reason, visualization systems often employ clipping planes (or boxes) that allow for cropping certain parts of the model to expose its interior. We currently allow for a single box that may or may not extend to infinity (to simulate a plane), and use this to clip BVH sub-trees. During traversal, if a node's subtree is completely enclosed in the crop box, we skip the subtree just as if it was out of the isorange. In SIMD, a box-in-box test is very cheap and can be amortized per packet, incurring negligible cost. An example of this feature is shown in Figure 8.

**Transparent Depth Peeling.** Another effect that allows one to see through an isosurface is to render it with transparency. Though straightforward to implement, transparency multiplies the complexity of rendering an image by the number of transparent hits required. Depth peeling could also be handled by storing multiple hit points in a

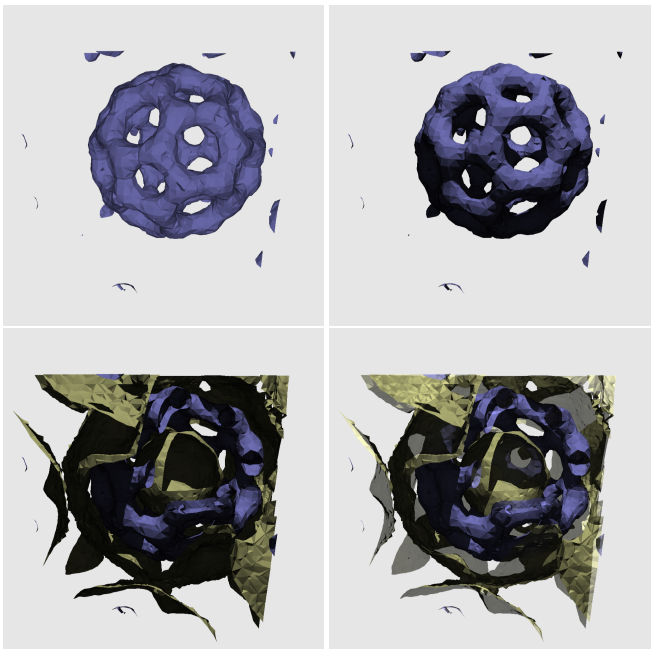

Fig. 8. *Impact of adding additional shading effects:* a) A bucky ball rendered with a single isosurface, and diffuse shading. b) After turning on diffuse shading with shadows. c) With a second isosurface and an interactive clip-box to expose the interior. d) Adding transparency as well. At $1024 \times 1024$ pixels on a Intel Core 1 duo laptop, these screenshots render at 15.6, 10.2, 5.4, and 2.6 frames per second, respectively. On our 16-core Opteron 2.4 GHz workstation, they render at 90, 70, 42, and 19 frames per second, respectively.

ray packet, but in our ray tracing architecture it is more elegant to implement via secondary rays in the shader. Rather than generating a set of completely new rays at the first surface, we can re-use the original ray packet by specifying a minimum hit distance for each ray. Thus, the secondary packet has exactly the same (common) origin, corner rays and frustum as the primary packet, allowing for all of the aforementioned optimizations. Rays that do not require a transparency ray are disabled, sometimes leading to partially-filled packets, but incurring no additional traversal steps or isopolygon intersections. Note that even though a BVH can have overlapping subtrees, shading will always be performed front-to-back, so both shadows and transparency are always computed accurately (Figure 8).

## 8 Results and Discussion

Thus far, we have addressed performance tradeoffs of individual algorithmic components in their respective sections. In this section, we consider benchmarks for the system as a whole, and evaluate the overall success of coherent IBVH ray tracing for tet-volume isosurfaces. For our experiments, we consider three representative machines: a laptop equipped with an Intel Core (1) Duo 2.33 GHz and 1 GB RAM; a Mac Pro desktop PC with a dual Intel Core 2 Duo 2.66 GHz and 4 GB RAM; and a 8-CPU dual-core (16 cores total) Opteron 2.4 GHz workstation with 64 GB RAM. In general we found the desktop frequently performed on par with the workstation, except in the case of multiple transparent isosurfaces on large data where the large L2 cache of the workstation had a major impact. If not mentioned otherwise, all examples run at $1024 \times 1024$ pixels, and use packets of $16 \times 16$ rays. The data sets and scenes we used for our comparisons are depicted in Figures 2 and 4.

### 8.1 Build Time and Performance

Because a tetrahedral mesh has far less geometric variation than a polygonal model (i.e., tets form a partition of space, and never overlap or self-intersect), the qualitative difference between a SAH and a BIH build is virtually nonexistent (Table 1). Because of the lower build times, we default to the BIH-style build. With the fast BIH-style build, most of the smaller data sets could in fact be rebuilt from scratch per frame.

| | ell32p | feok | bucky | blunt | buckycube | fusion (t=50) |
|---|---|---|---|---|---|---|
| #tets | 148,955 | 121,668 | 224,874 | 176,856 | 11.3m | 3m x 116 |
| render performance (frames per second) | | | | | | |
| BIH | 27.0 | 23.6 | 18.8 | 28.4 | 6.23 | 11.47 |
| SAH | 26.3 | 23.6 | 18.9 | 28.5 | 6.30 | 12.13 |
| build time (ms, dual Intel Core 2 Duo 2.66 GHz) | | | | | | |
| BIH | 46 | 42 | 61 | 87 | 4988 | 1495 |
| SAH | 2432 | 1854 | 2932 | 3887 | 312620 | 70689 |

Table 1. BIH-style build vs SAH for building the Implicit BVH. Because the tetrahedra are distributed over space for more evenly than triangles in a polygonal model, the render performance between BIH-style build and SAH build is very similar, but executing the BIH-style build is much faster.

### 8.2 Rendering Performance

As can be seen from Table 1 and Figure 2, all of the static examples can be rendered at multiple frames per second even on the dual-core laptop. For static scenes, performance is typically linear in the number of CPU cores, but with an upper bound of 50–60 fps due to the cost of writing ray-traced pixels to the GPU frame buffer. Empirically, we find our application scales roughly linearly with respect to number of pixels per frame. Thus, a frame buffer of $512 \times 512$ generally renders four times faster than at $1024 \times 1024$, allowing for quite interactive rates even when rendering difficult scenes on the laptop.

| | ell32p | feok | bucky | blunt | buckycube | fusion (t=50) |
|---|---|---|---|---|---|---|
| render performance (frames per second) | | | | | | |
| laptop | 14.2 | 12.6 | 13.3 | 18.9 | 2.8 | 3.3 |
| desktop | 29.4 | 25.9 | 27.3 | 45.5 | 7.21 | 8.47 |
| workstation | 95 | 93 | 90 | 94 | 19.1 | 26.1 |

Table 2. *Performance in frames per second* for various data sets and platforms. *Laptop* is an Intel Core Duo 2.33 GHz, 1 GB RAM. *Desktop* is a 4-core dual Intel Core 2 Duo 2.66 GHz, 4 GB RAM. *Workstation* is a 16-core cc-NUMA 2.4 GHz Opteron, with 64 GB RAM. Refer to Figure 2 for images.

**Scalability in model size.** Performance degrades quite gracefully when increasing model size, dropping at most by 4x when going from the smallest model (feok, 121k tets) to the most complex one (buckycube, 11.3m tets), even though the latter has nearly 100 times the number of tets. This is largely due to the logarithmic complexity of ray tracing efficiency structures, and the packet-amortized cost of memory access. To further evaluate scalability to large models, we have generated several example scenes where we replicated a bucky ball $n \times n \times n$ times *without instancing*. As evident in Table 3, performance drops moderately even for hugely complex models of up to nearly a billion tets.

| # replications | 1 | $2^3$ | $4^3$ | $8^3$ | $16^3$ |
|---|---|---|---|---|---|
| # tets total | 177k | 1.4m | 11.3m | 90.4m | 724m |
| frames per second | 34 | 13.5 | 5.0 | 1.8 | 0.66 |

Table 3. Performance in frames per second on four Opteron 2.4GHz cores, for varying numbers of replication of the bucky ball scene (no instancing is used).

**Comparison to existing approaches.** Our results compare quite favorably to the isosurface ray tracing performance achieved by Marmitt et al.'s Plücker-based tet marching algorithm [20], which reported 1.67 and 0.92 fps at $512 \times 512$ on a dual-Opteron for isosurfaces on the bluntfin and buckyball, respectively. On comparable hardware (and scaled to same viewport size), our system performs approximately 40 times faster. However, it is important to note that the Marmitt et al. method also supports semi-transparent volume ray casting, which ours doesn't. Comparison with GPU isosurfacing methods is more difficult, due to completely different and continually changing hardware and programming models. We therefore restrain from any absolute comparisons, but believe that the frame rates we achieve are sufficiently interactive to compete with most GPU based methods for large data sets, while offering more flexibility and unconditional accuracy.

### 8.3 Traversal Efficiency

The key to this interactive performance lies in the aggressive large-packet traversal scheme, as can be seen from Table 4. Speculative descent and frustum culling greatly reduce the number of individual ray-box tests during traversal by roughly a factor of 18–51 compared to tracing $2 \times 2$ packets (the smallest an SSE-based system can trace). Using packets allows for traversal and intersection code in SSE, which is crucial to realizing the performance potential of modern CPU's.

Because we have transformed the ray-isotet intersection to a polygonal problem, the same frustum culling techniques can also be used to significantly reduce the number of individual ray-isopolygon tests, by about 2–3×, even though for the most complex scene the number of ray-isopolygon tests actually increases (see Table 4). Finally, larger packets allow for amortizing per-packet operations like isorange culling and isotet extraction over the entire packet, thus reducing the total number of these operations per frame. As evident in Table 4, this reduces the number of isopolygon generations by about 6–40×, and the number of culling tests by 22–55×.

| # scene | bluntfin | buckyball | ell32P | fe_ok | fusion50 | bucky cube |
|---|---|---|---|---|---|---|
| number of individual ray-box tests | | | | | | |
| 2x2 | 48.05 | 93.84 | 56.75 | 57.42 | 175.83 | 95.89 |
| 16x16 | 0.94 | 1.8 | 1.11 | 1.10 | 4.32 | 5.44 |
| ratio | 51× | 52× | 52× | 52× | 41× | 18× |
| number of individual ray-isopolygon tests | | | | | | |
| 2x2 | 8.90 | 13.52 | 8.0 | 12.45 | 29.35 | 15.51 |
| 16x16 | 3.19 | 4.42 | 3.39 | 3.86 | 16.47 | 23.91 |
| ratio | 2.8× | 3.0× | 2.4× | 3.5× | 1.8× | 0.65× |
| number of total packet isorange tests | | | | | | |
| 2x2 | 76.75 | 152.31 | 99.89 | 95.96 | 279.75 | 181.48 |
| 16x16 | 1.45 | 2.84 | 1.88 | 1.79 | 6.48 | 8.29 |
| ratio | 51× | 54× | 53× | 53× | 43× | 22× |
| number of total isopolygon extractions (×1000) | | | | | | |
| 2x2 | 2216 | 354 | 1908 | 4436 | 7285 | 3468 |
| 16x16 | 69 | 10 | 6429 | 109 | 296 | 616 |
| ratio | 32× | 34× | 29× | 41× | 25× | 5.6× |

Table 4. Traversal statistics of using our aggressive packet-frustum traversal scheme (using $16 \times 16$ rays) vs. standard $2 \times 2$ packet traversal.

**Isopolygon caching vs on-the-fly recomputation.** Because the large packets reduce the number of isopolygon extractions, caching the isopolygons has a relatively low impact. Even when using only a single CPU and a large enough cache (so no conflicts occur, and all synchronization can be disabled), caching only increases total frame rate by 5–8% over on-the-fly recomputation, thus we opt for the on-the-fly recomputation by default.

### 8.4 Multiple Isosurfaces, Shadows, and Transparency

As mentioned in Section 7, more advanced shading bears a significant cost, mostly due to the higher number of rays traced in the scene. Shadows usually increase the render cost by about 2x if the rendered object covers the entire screen, and somewhat less, otherwise (also see Figure 8). Of course, adding more shadow-casting light sources–or even soft shadows or global illumination–would further increase the cost per image, making these effects infeasible on low-end hardware.

Transparency, too, adds to the number of rays traced per image, and correspondingly increases the render cost, particularly if the object has a high depth complexity. For this reason, we typically reduce the number of transparency levels to a user-specified maximum (2 by default), which can be changed interactively. All these effects can be supported simultaneously, even for the complex time-varying data sets (see Figures 8 and 4).

With diffuse shading, supporting multiple isosurfaces in itself does not significantly raise the cost of an image, due to the ray tracer's implicit occlusion culling (the 2× drop in framerate in Figure 8 is entirely due to the 2× higher projected area of the model after adding the outer isosurface). Adding the clip-box in Figure 8 is virtually cost-free.

### 8.5 Time-Varying Data Sets

With isopolygon caching disabled by default, the performance for handling time-varying data depends entirely on the cost of retrieving the BVH and geometry for the proceeding frame. When BVH and vertex positions are precomputed for each frame, switching to a new BVH has no measurable performance impact, as switching requires only changing a few pointers, and models are too large to remain resident in L2 cache anyway. On the other hand, precomputation requires a lavish amount of main memory: for the fusion data set, storing a precomputed BVH and vertices for each time step currently requires a total of 21 GB of memory. Though we believe this could be significantly reduced, this memory footprint is still significant.

Without replicating the vertex arrays and precomputing the BVHs, all 116 time steps of the 3 million tet fusion data set can be fit into 538 MB (including one shared BVH that is refit per frame), allowing us to render even that model on the laptop. However, refitting requires updating the vertex array, all the BVH nodes, and some precomputed shading data (e.g., per-tet gradients) per frame, adding a significant per-frame cost that limits maximum performance. The update is fully parallelized, but – unlike rendering – scales poorly due to intensive and asymmetrical memory access on that particular workstation's cc-NUMA architecture.

In short, precomputation and refitting offer a classical trade-off between performance and memory consumption. For the fusion data set shown in Figure 4 with all effects turned on, precomputation results in 7–15 fps on the 16-core Opteron, but requires 21 GB or memory. Refitting requires only 538 MB of memory, but is limited to 3.5 fps if we switch to a new time step every frame.

For smaller models, interactive refitting is not an issue, and for model sizes of 100K–250K tets even per-frame rebuilds are feasible (see Table 1). This would even allow for applications where neither scalar field, nor number of tets, nor mesh topology are known in advance. For models as large as the fusion data set, this is currently not possible at interactive rates. However, as the serial BIH build is sufficiently fast that an efficiently implemented distributed build could permit fully dynamic rebuilding.

## 9 Conclusion

In this paper we have shown it is possible to ray trace isosurfaces of tetrahedral scalar fields at interactive to real-time frame rates, purely on the CPU. In doing so, we are able to correctly visualize large unstructured volumes, interactively manipulate isovalues and shader modalities, and handle time-varying data with hundreds of steps.

The main algorithmic contributions of this paper are the fast packet-isotetrahedron intersection test and extension of the coherent BVH to an implicit min-max tree over the tetrahedral volume. Our implementation naturally supports multiple isosurfaces, on-the-fly clipping, semi-transparent depth peeling, and shadows. Accommodation of large data is limited only by host memory capacity, though the overhead of the BVH must be taken into consideration. Time-varying data can be handled by either precomputing an implicit BVH, or by building a single IBVH that is updated on the fly. In the former case, one can jump immediately between arbitrary time steps – a feat that would be difficult for streaming GPU methods. Overall, we present a practical tool for visualizing large tetrahedral data sets.

Our approach opens several avenues for future work. We could extend IBVH traversal to direct volume rendering methods, such as maximum intensity projection (MIP) or full transfer-function methods. Though the latter suffer from high traversal complexity, the IBVH could still be useful for space-skipping when the transfer function is sufficiently sparse, as in [16]. Another intriguing extension would be support for higher-order finite elements in the spirit of Nelson et al. [21] or Rössl et al. [26]. This would require a completely different intersection routine, but the IBVH traversal would remain unchanged. Also of interest

would be more advanced lighting effects such as soft shadows, ambient occlusion, or global illumination, which can significantly improve understanding of data sets [11]. Finally, investigating scalable build algorithms could allow for rendering even complex data with arbitrary deformations without precomputation.

## REFERENCES


[1] F. F. Bernardon, S. P. Callahan, J. L. D. Comba, and C. T. Silva. An adaptive framework for visualizing unstructured grids with time-varying scalar fields. *Parallel Computing*, 2007. to appear.

[2] S. Boulos, I. Wald, and P. Shirley. Geometric and Arithmetic Culling Methods for Entire Ray Packets. Technical Report UUCS-06-010, SCI Institute, University of Utah, 2006.

[3] S. P. Callahan, L. Bavoil, V. Pascucci, and C. T. Silva. Progressive volume rendering of large unstructured grids. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Visualization / Information Visualization 2006)*, 12(5):1307–1314, Sept/Oct 2006.

[4] S. P. Callahan, J. L. D. Comba, P. Shirley, and C. T. Silva. Interactive rendering of large unstructured grids using dynamic level-of-detail. In *IEEE Visualization '05*, pages 199–206, 2005.

[5] S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[6] D. DeMarle, S. Parker, M. Hartner, C. Gribble, and C. Hansen. Distributed Interactive Ray Tracing for Large Volume Visualization. In *Proceedings of the IEEE PVG*, pages 87–94, 2003.

[7] K. Dmitriev, V. Havran, and H.-P. Seidel. Faster Ray Tracing with SIMD Shaft Culling. Research Report MPI-I-2004-4-006, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 2004.

[8] A. Doi and A. Koide. An efficient method of triangulating equi-valued surfaces by using tetrahedral cells. *IEICE Trans Commun. Elec. Inf. Syst*, E-74(1):213–224, 1991.

[9] M. P. Garrity. Raytracing irregular volume data. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24(5):35–40, November 1990.

[10] J. Goldsmith and J. Salmon. Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987.

[11] C. Gribble. *Interactive Methods for Effective Particle Visualization*. PhD thesis, University of Utah, 2006.

[12] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001.

[13] Intel. http://www.intel.com/go/terascale/, 2006.

[14] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. In *Proceedings of Pacific Graphics '04*, pages 186–195, 2004.

[15] A. Knoll, I. Wald, S. G. Parker, and C. D. Hansen. Interactive Isosurface Ray Tracing of Large Octree Volumes. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 115–124, 2006.

[16] J. Krüger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings IEEE Visualization 2003*, pages 257–292, 2003.

[17] C. Lauterbach, S.-E. Yoon, D. Tuft, and D. Manocha. RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing*, pages 39–45, 2006.

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

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

[20] G. Marmitt and P. Slusallek. Fast Ray Traversal of Tetrahedral and Hexahedral Meshes for Direct Volume Rendering. In *Eurographics/IEEE-VGTC Symposium on Visualization (EuroVIS)*, pages 235–242, 2006.

[21] B. Nelson and R. M. Kirby. Ray-tracing polymorphic multi-domain spectral/hp elements for isosurface rendering. *IEEE Transactions on Visualization and Computer Graphics (Proceedings IEEE Visualization 2005)*, 12(1):114–125, 2005.

[22] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan. Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization '98*, pages 233–238, October 1998.

[23] V. Pascucci. Isosurface Computation Made Simple: Hardware Acceleration, Adaptive Refinement and Tetrahedral Stripping. In *Eurographics - IEE TCVG Symposium on Visualization (2004)*, pages 293–300, 2004.

[24] S. Pesco, P. Lindstrom, V. Pascucci, and C. T. Silva. Implicit Occluders. In *IEEE/SIGGRAPH Symposium on Volume Visualization*, pages 47–54, 2004.

[25] A. Reshetov, A. Soupikov, and J. Hurley. Multi-Level Ray Tracing Algorithm. *ACM Transaction on Graphics*, 24(3):1176–1185, 2005. (Proceedings of ACM SIGGRAPH 2005).

[26] C. Rössl, F. Zeilfelder, G. Nürnberger, and H.-P. Seidel. Reconstruction of Volume Data with Quadratic Super Splines. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):397–409, 2004.

[27] C. Wächter and A. Keller. Instant Ray Tracing: The Bounding Interval Hierarchy. In *Rendering Techniques 2006 – Proceedings of the 17th Eurographics Symposium on Rendering*, pages 139–149, 2006.

[28] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[29] I. Wald, S. Boulos, and P. Shirley. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics*, 26(1):1–18, 2007.

[30] 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*, 11(5):562–573, 2005.

[31] I. Wald, T. Ize, A. Kensler, A. Knoll, and S. G. Parker. Ray Tracing Animated Scenes using Coherent Grid Traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006. (Proceedings of ACM SIGGRAPH).

[32] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).

[33] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, pages 333–340, Washington, DC, USA, 2003. IEEE Computer Society.

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

[35] J. Wilhelms and A. Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.

[36] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral Projection using Vertex Shaders. In *Proceedings of IEEE Volume Visualization and Graphics Symposium*, pages 7–12, October 2002.

[37] G. Wyvill, C. McPheeters, and B. Wyvill. Data structure for soft objects. *The Visual Computer*, 2:227–234, 1986.