Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes

Ingo Wald, Heiko Friedrich, Aaron Knoll, and Charles D. Hansen, *Senior Member, IEEE*

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 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 volumes is a persistent challenge in 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 for dynamic changes in camera, lighting, isovalue and time step, without sacrifice in interactivity.

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 [14], 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 (BVHs), 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-trace 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 BVH 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.

- *Ingo Wald is with the SCI Institute, University of Utah, as well as with Intel Corp, Santa Clara, CA; E-mail: wald@sci.utah.edu.*
- *Heiko Friedrich is with the Computer Graphics Group at Saarland University, Saarbrücken, Germany; E-mail: heiko@graphics.cs.uni-sb.de.*
- *Aaron Knoll is with the SCI Institute, University of Utah; E-mail: knolla@sci.utah.edu.*
- *Charles D. Hansen is with the SCI Institute, University of Utah; E-mail: hansen@cs.utah.edu.*

Manuscript received 31 March 2007; accepted 1 August 2007; posted online 27 October 2007. Published 14 September 2007.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

2 Related Work

2.1 Isosurface Extraction

Marching cubes was first applied to isosurface extraction of structured data by Wyvill et al. [41], and Lorensen & Cline [21]. 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 [25] showed that the vertex processor can be utilized to create appropriate quadrilaterals for the isosurface within a tetrahedron. Similarly, Klein et al.[15] 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 [20], adaptive extraction [38], and implicit occlusion culling [26].

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 [29] 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 [40]. 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

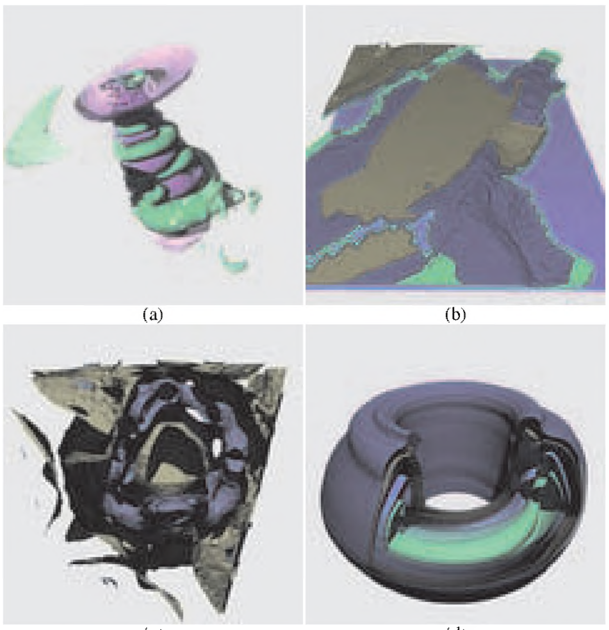

Fig. 1. Several samples of our interactive system running at 1024×1024 pixels: a) tjet (1m tets) with shadows, transparent depth-peeling, and multiple isosurfaces b) SF1 (14m tets) with four isosurfaces. c) buckyball with two a clip-box, multiple isosurfaces and shadows, d) Time step 60 of the time-varying fusion data set (3m tets, 116 time steps), rendered with four isosurfaces, clip box, shadows, and transparency. With a 1024×1024 frame buffer, these examples render at 2.0, 3.1 5.4, and 0.8 fps, respectively, on an Intel Core 1 Duo 2.33 GHz laptop with 1 GB RAM; and and 11, 18, 52, and 10 fps, respectively, on a 16-core 3.0 GHz Opteron workstation with 64 GB RAM.

of unstructured data at real-time rates, albeit with minor artifacts and delayed full visualization of large data. Bernardon et al. [1] modified HAVS to visualize isosurfaces. GPU fragment-program ray casting approaches, as first proposed by Weiler et al. [37] have also proven feasible. Georgii & Westermann [10] perform ray-casting through projected cells on the GPU, and demonstrate performance gains over [3]. However, for all rasterization-based GPU techniques, interactivity degrades significantly for larger datasets over 1 million tets.

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. [36], 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. [27], or Wald et al. [32]), 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 [32, 34, 19, 35]. Of particular interest to our approach is the *dynamic BVH traversal* proposed by Wald et al. [32].

2.4 Interactive Isosurface Ray Tracing

Isosurface ray tracing on the CPU has been explored before, particularly for large data applications. Parker et al. [24] employed a hierarchical grid to ray trace isosurfaces on a small supercomputer; DeMarle et al. [6] extended this implementation to clusters. Knoll et al. [16] proposed losslessly compressed octree volumes for rendering larger data. Wald et al. [33] 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 [22] 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 [33], 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 [33], 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 [32]. 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 16×16 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 are 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 $f(\vec{x}) = v$ 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

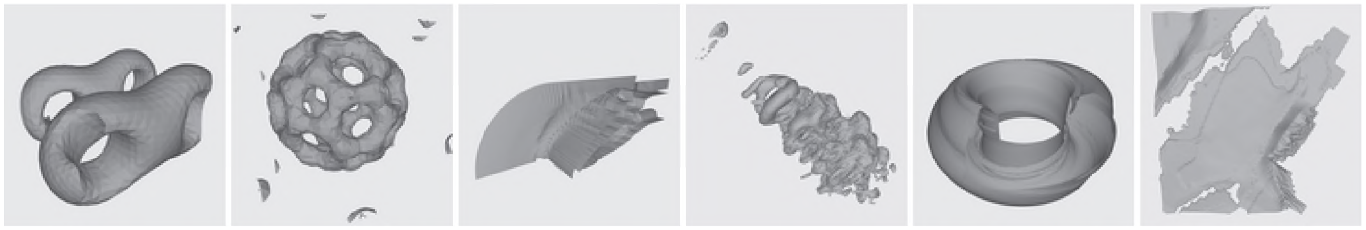

Fig. 2. From left to right: ell32P (149k tets), bucky ball (177k tets), bluntfin (225k tets, two isosurfaces), tjet (1m tets), timestep 50 of the fusion data (3m tets), and the sf1 seismic data (14m tets). With simple shading, these examples run at 14.2, 13.3, 18.9, 10.1, 4.0 and 3.3 frames per second (1024×1024 pixels) on an Intel Core 1 Duo 2.33 GHz laptop with 1GB RAM, and at 116, 112, 95, 66, 57, and 32 frames per second on a 16-core 3.0 GHz Opteron workstation.

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 how 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 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 the aforementioned table of 16 cases. Once we know which tet edges contain 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 [31] 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, rejecting 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. [7, 2]. 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 perform a distance test for our component rays.

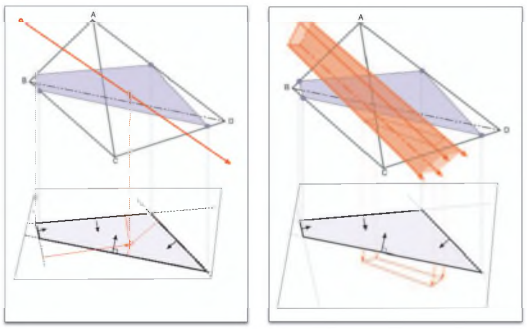

Fig. 3. *Ray-Isopolygon Intersection in an Isotetrahedron.* Knowing that the isosurface inside the tetrahedron is a plane, we first extract an isopolygon. We then compute the point where the ray pierces that polygon's supporting plane, and project both the polygon and that hit point to a 2D coordinate 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.

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. 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, though this ratio declines for larger models. Every time SIMD frustum culling rejects a packet test, all individual ray-isopolygon tests are avoided, e.g. 256 for a 16×16 ray packet.

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 all rays in the packet – 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 [33] 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 [39]). 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 [32, 30], and tets behave similarly to triangles in this regard.

Traditional bottom-up BVH builds (e.g. [11]) generally result in inefficient BVHs [13]. 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 [32], and Wächter et al.'s fast spatial median build as proposed in his bounding interval hierarchy (BIH) paper [30]. The SAH build employs a *surface area heuristic* [11, 13] 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 [32], 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. [32], 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 are also active, this strategy allows many ray-box tests to be skipped when numerous consecutive rays are active.

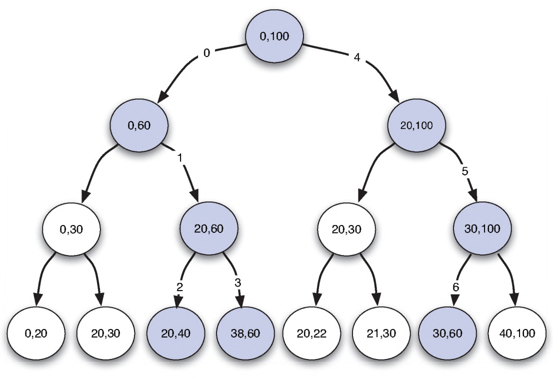

Fig. 5. *Implicit Culling.* The implicit BVH is a min-max tree containing only a subset of BVH nodes containing our desired isovalue(s). We can speculatively descend the min-max tree until we reach a leaf, or an intersection test fails. Only at bifurcation nodes (dark blue) must we resort immediately 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.

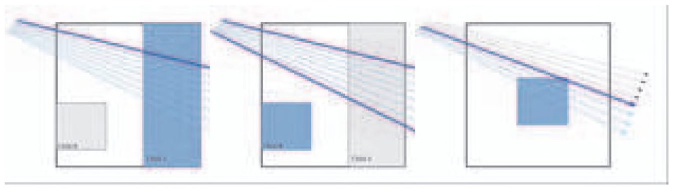

Fig. 6. *First-active descent, frustum test, and active ray tracking.* Given a BVH node, we speculatively test the first "active" ray in the packet against the bounding box, and immediately descend if it hits (left). If this test fails, we perform a frustum test to reject nodes completely outside the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

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. [27, 2]) 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 BVH construction, balancing performance and memory footprint. Results are analyzed in Sec. 8.6.

6.1 Schema I: Unique BVH Per-Step

The naïve way of accommodating time-varying data is to compute a unique BVH for each time step. 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 entail a considerable memory footprint.

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 BVH. This technique has been successfully applied to ray tracing dynamic triangle meshes [32, 19]. 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 BVH, this method works extremely well due to the continuous nature of tet deformations in FE

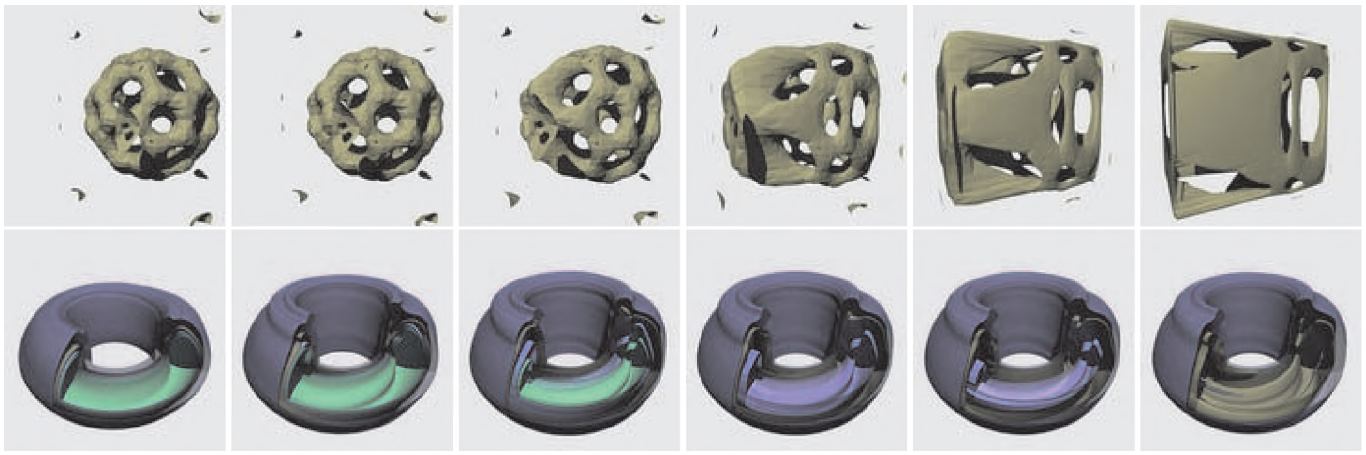

Fig. 4. Two examples of time-varying data sets, rendered at 1024×1024 pixels, using a 16-core 3.0 GHz Opteron workstation. Top: An artificially created deforming bucky ball that shows severe deformation of its 226K tets, running at 50+ frames per second including shadows from a point light source. Bottom: 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.

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. 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.

Shadows. Shadows add important visual cues in understanding shape (see Figure 7). 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. Fortunately, shadow packets may still employ the Reshetov et al. [27] frustum-culling technique at traversal, as this requires no actual geometric frustum. The overall speed impact of shadow rays varies, but is typically lower than $2\times$ (see Figure 7a-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 7.

Transparent Depth Peeling. Rendering transparent isosurfaces also provides better understanding of the dataset. Though straightforward to implement, transparency multiplies the complexity of rendering an image by the number of transparent hits required. Though it is possible to implement by recording multiple hits per ray, in our packet architecture it is more elegant to implement as a shader via secondary rays. By simply specifying a minimum hit distance for each transparency ray, we can re-use the origin, corner rays and frustum of the original ray packet. 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. As shading is performed front-to-back, shadows and transparency are always computed accurately (Figure 7).

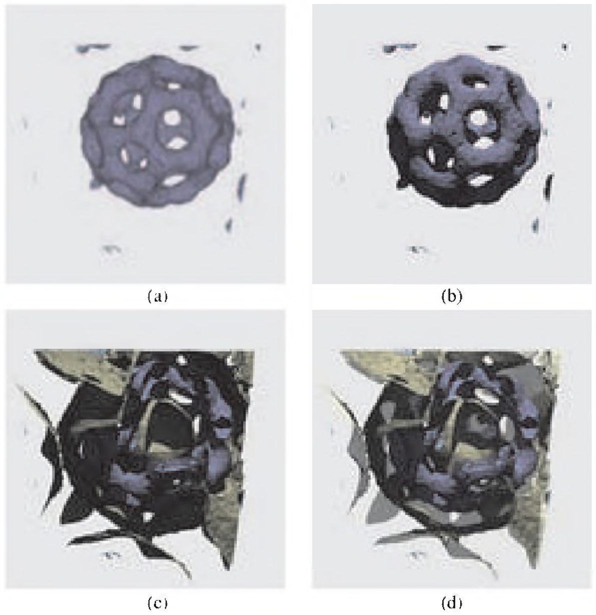

Fig. 7. *Impact of adding additional shading effects:* a) A bucky ball rendered with a single isosurface, and diffuse shading. b) After turning on diffuse shading with shadows. c) With a second isosurface and an interactive clip-box to expose the interior. d) Adding transparency as well. At 1024×1024 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 3.0 GHz workstation, they render at 90, 70, 42, and 19 frames per second, respectively.

8 Results and Discussion

In this section, we evaluate the system as a whole, and the overall success of coherent BVH ray tracing for tet-volume isosurfaces. For our benchmarks, we consider three representative machines: a laptop equipped with an Intel Core (1) Duo 2.33 GHz and 1 GB RAM; a 4-core dual Intel Xeon 2.33 GHz desktop with 4 GB RAM; and a 8-CPU dual-core (16 cores total) Opteron 3.0 GHz workstation with 64 GB RAM. Unless otherwise stated, all examples run at 1024×1024 pixels, and use packets of 16×16 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	bucky	blunt	tjet	fusion (t=50)	sf1
#tets	148,995	176,856	224,874	1.0m	3m x 116	14m
render performance (frames per second)						
BIH	48.0	39.4	53.8	28.5	11.8	13.1
SAH	43.7	39.5	57.1	27.7	12.3	13.1
build time (ms, dual Intel Xeon 2.33 GHz)						
BIH	32	40	61	607	1402	4908
SAH	1647	1794	2710	20886	70119	311267

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

8.2 Rendering Performance

As seen in 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. Empirically, we found our application scales roughly linearly with respect to the number of pixels per frame. Thus, a frame buffer of 512×512 generally renders four times faster than at 1024×1024, enabling interactive rates for difficult scenes on the laptop.

	ell32p	bucky	blunt	tjet	fusion (t=50)	sf1
render performance (frames per second)						
laptop	14.2	13.3	18.9	10.1	4.0	3.3
desktop	48.0	39.4	53.8	28.5	11.8	13.1
workstation	116	112	95	66	57	32

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 Xeon 2.33 GHz, 4 GB RAM. *Workstation* is a 16-core cc-NUMA 3.0 GHz Opteron, with 64 GB RAM. Refer to Figure 2 for images.

8.3 Scalability in model size

Performance degrades gracefully when increasing model size, dropping only by 4x from from the smallest model (feok, 121k tets) to the most complex one (sf1, 14M 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 synthetically replicated a bucky ball $n \times n \times n$ times *without instancing*. As evident in Tab. 3, performance drops moderately even for hugely complex models of up to nearly a billion tets. Though they require workstation-class memory capacity, large unstructured data such as the STP bullet simulation (36m tets) render equally efficiently (Fig. 8).

# replications	1	2^3	4^3	8^3	16^3
# tets total	177k	1.4m	11.3m	90.4m	724m
frames per second	43	16.7	6.2	2.0	0.80

Table 3. Performance in frames per second on four Opteron 3.0GHz cores, for varying numbers of replication of the bucky ball scene (no instancing is used).

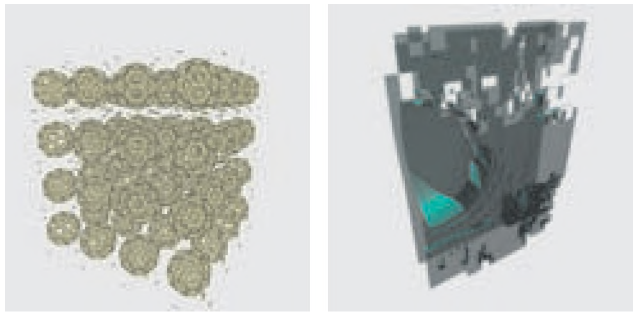

Fig. 8. Left: 4^3 replicated buckyballs with 11.3m tets. Right: STP dataset with 36m tets. With simple shading, these datasets perform at 27.8 and and 26.9 fps respectively on a 16-core 3.0 GHz Opteron workstation with 64 GB RAM.

8.4 Traversal Efficiency

The key to this interactive performance lies in the aggressive large-packet traversal scheme, as seen in 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 2×2 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×, 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	ell32P	bucky	bluntfin	tjet	fusion (t=50)	sf1
number of individual packet-box tests						
2x2	56.75	93.84	48.05	44.67	175.83	33.21
16x16	1.11	1.8	0.94	1.20	4.32	1.69
ratio	52×	52×	51×	37×	41×	20×
number of individual ray-isopolygon tests						
2x2	8.0	13.52	8.90	6.8	29.35	9.37
16x16	3.39	4.42	3.19	3.95	16.47	7.64
ratio	2.4×	3.0×	2.4×	1.7×	1.8×	1.22×
number of total packet isorange tests						
2x2	99.89	152.31	76.75	135.32	279.75	77.10
16x16	1.88	2.84	1.45	3.00	6.48	2.72
ratio	53×	54×	51×	45×	43×	28×
number of total isopolygon extractions (×1000)						
2x2	1908	354	2216	1154	7285	1943
16x16	64	10	69	110	296	373487
ratio	29×	34×	32×	10.3×	25×	5.2×

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

8.5 Multiple Isosurfaces, Shadows, and Transparency

Rendering 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 7 is due to the 2× higher projected area of the model after adding the outer isosurface. However, as mentioned in Section 7, advanced shading bears a significant cost due to the higher number of rays traced. Shadows usually increase the render cost by about 2x if the rendered object covers the entire screen, and somewhat less, otherwise (also see Figure 7). Transparency similarly increases to the total number of rays traced per-frame, and thus increases the render cost. We typically limit the number of transparency rays to a user-specified maximum (2 by default), which can be changed interactively. All these effects can be supported simultaneously, even for large time-varying data sets (see Figures 4 and 7).

8.6 Time-Varying Data Sets

Precomputing a BVH and replicating vertex arrays for each timestep, as in Sec. 6.1, is only practical for small data or workstations with copious memory. For the fusion dataset this requires over 22 GB in memory footprint. Nevertheless, this scheme remains desirable, as moving across timesteps incurs no noticeable penalty in frame rate. Conversely, by employing a single BVH and refitting it per-frame (Sec. 6.2), the BVH and all 116 time steps of the fusion data occupy only 538 MB, allowing us to render 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. This update is fully parallelized, but scales poorly due to intensive and asymmetrical memory access on our workstation's cc-NUMA architecture. Effectively, refitting adds a significant per-frame cost that limits its maximum performance to 3.5 fps on the workstation. Moreover, precomputation and refitting offer a classical trade-off between performance and memory consumption.

8.7 Memory Overhead

The bounding volume hierarchy structure occupies a significant footprint in main memory. In our implementation, the BVH requires two arrays: one for BVH nodes, at 32 bytes per node, and another for storing the lists of tet IDs that the leaf nodes refer to. The tetID list uses a constant amount of memory, requiring exactly 4 bytes per tet. The size of the node array depends on how many nodes are allocated, which in turn depends on the data and build strategy. In the worst case, a BVH would always split until each tet is contained in exactly one leaf, in which case a total of $2N-1$ nodes, (i.e., roughly $64 \times N$ bytes) would be allocated for the node array. In practice, the optimal BVH is much shallower, and uses only a fraction of that memory ($^1/_4{}^{th}$–$^1/_6{}^{th}$).

For that worst-case assumption, however, table 5 shows that for static scenes, memory overhead is around $4\times$ that of the raw input data. For the time-varying deformed bucky and fusion data sets, this overhead increases to a significant $18\times$ and $20\times$ if a separate BVH is stored per time step. If the BVH is shared over time, the overhead drops to 92% for the deformed bucky while for the fusion data set the overhead is only 18%. In general, more time steps reduce the relative overhead, as they amortize input tet data footprint.

scene	number of			raw	BVH per step		shared BVH	
	tets	verts	steps	mem	mem	ratio	mem	ratio
ell32p	149k	33k	1	2.8MB	9.6MB	4.1×	–	–
bluntfin	225k	41k	1	4.1MB	18MB	4.2×	–	–
SF1	13.9m	2.5m	1	251MB	906MB	3.6×	–	–
TJet	1m	163k	1	17.7MB	64.9MB	3.6×	–	–
bucky $\times 4^3$	11.3m	2.1m	1	205MB	734MB	4.2×	–	–
STP	36m	6.3m	1	1.7GB	7.2GB	4.2×	–	–
bucky. def.	176k	32k	20	12.7MB	234MB	18×	11.7MB	0.92×
fusion	3.0m	622k	116	1.1GB	22GB	20×	194MB	0.18×

Table 5. *Memory usage and BVH overhead.* Note that we report a worst-case upper bound on BVH memory ($2 \times N - 1$ nodes for N tets), as this is what our system actually pre-allocates memory for. In practice, only about one fourth to one sixth of that pre-allocated memory is actually used (i.e., memory overhead could be reduced rather easily should that ever become an issue).

8.8 Comparison to Existing CPU based Approaches

Our results compare favorably to the performance achieved by Marmitt et al.'s Plücker-based tet marching algorithm [22], which reported 1.67 and 0.92 fps at 512×512 on a dual-Opteron for isosurfaces on the bluntfin and buckyball, respectively. On comparable hardware and frame buffer size, our system performs around 40 times faster. However, it is important to note that the Marmitt et al. method also supports semi-transparent volume ray-casting, which ours does not.

8.9 Comparison to Existing GPU based Approaches

GPU hardware is continually changing, so comparing to previously published results would be an unfair comparison to already-outdated hardware. For that reason, we have decided to base our comparisons mainly on HAVS [4] and its isosurface extension [1], running on a state-of-the-art nVidia 8800 GTX. HAVS is well-known and freely available, thus an appropriate system for benchmarking GPU performance. As seen in Table 6, when isosurfacing small and moderate-sized datasets (less than 1M), ray tracing achieves roughly equivalent performance on a 4-core Xeon as rasterization on the nVidia 8800 GTX in the same desktop. For larger data sets, however, our method can outperform HAVS significantly, even for models that fit comfortably in GPU memory.

For small data such as the bluntfin, isosurfacing via the GPU ray-casting method of Georgii & Westermann [10] reports 175 fps at 512×512 on an nVidia 7900 GTX; our system achieves 160 fps on the 4-core Xeon desktop at the same resolution. However, their performance degrades significantly for larger datasets over 1M tets. We refrain from absolute comparison, but our system achieves similar performance for small data, and is substantially faster for large data. Again, it should be noted these GPU methods are designed for object-order volume rendering without acceleration structures, whereas our technique relies on logarithmic-order BVH traversal and is restricted to isosurface visualization. Nonetheless, these results suggest that CPU ray tracing is roughly competitive in performance with GPU methods for isosurface visualization of unstructured grids, and exhibits better overall scalability.

scene	ell32P	bucky	bluntfin	tjet	fusion	SF1
# Tetrahedra	149k	177k	225k	1m	3m	14m
BVH	48	39.4	53.8	28.5	11.8	13.1
HAVS	50	50	30	3.0	1.5	0.3

Table 6. *GPU Performance Comparison*, in frames per second, with HAVS [4, 1], running on an nVidia 8800 GTX, and our method on a 4-core Intel Xeon 2.33 GHz, at 1024×1024 resolution.

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 per time step, or by building a single BVH that is updated on the fly.

Compared to existing GPU methods, our system exhibits better scalability to large data, and is not limited by the GPU memory capacity. However, our current system is limited to isosurfacing, whereas existing GPU methods support direct volume rendering. Moreover, multi-core CPUs are increasingly mainstream, and future GPUs will likely evolve to run a ray-tracing system similar ours. Ultimately, the question is not one of GPU vs CPU, but rather which rendering algorithm is used.

Our approach opens several avenues for future work. We could extend BVH 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 BVH could still be useful for space-skipping when the transfer function is sufficiently sparse, as in [17]. Another intriguing extension would be support for higher-order finite elements in the spirit of Nelson & Kirby [23] or Rössl et al. [28]. This would require a completely different intersection routine, but the BVH 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 [12]. Finally, investigating scalable build algorithms could allow for rendering even complex data with arbitrary deformations without precomputation.

Acknowledgements

This work was supported by the U.S. Department of Energy through the Center for the Simulation of Accidental Fires and Explosions, under grant W-7405-ENG-48, and by the National Science Foundation under CISE grants number CRI-0513212, CCF-0541113, and SEII-0513212. It was also supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-FC02-06ER25781 through the SciDAC Visualization and Analytics Center for Enabling Technologies (VACET), as well as through a visiting professorship by Intel Corp. The fusion simulation was originally performed by Kruger et al. [18]; the data set was made available to us by Allen Sanderson from the SCI Institute. We would like to thank Steve Callahan for his assistance in comparing performance with GPU rendering using HAVS, and the anonymous reviewers for their constructive comments.

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*, 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*, 24(5), 1990.

[10] J. Georgii and R. Westermann. A Generic and Scalable Pipeline for GPU Tetrahedral Grid Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1345–1352, 2006.

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

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

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

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

[15] 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.

[16] 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.

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

[18] S. E. Kruger, D. D. Schnack, and C. R. Sovinec. Dynamics of the major disruption of a DIII-D plasma. *Physics of Plasmas*, 12, 2005.

[19] 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.

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

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

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

[27] 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).

[28] 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.

[29] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *ACM Computer Graphics (Proceedings San Diego Workshop on Vlume Visualization 1990*, 24(5):63–70, 1990.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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).

[35] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley. State of the Art in Ray Tracing Animated Scenes. In *Eurographics 2007 State of the Art Reports*.

[36] 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).

[37] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings IEEE Visualization 2003*, pages 333–340, 2003.

[38] 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.

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

[40] 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, 2002.

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