Faster Isosurface Ray Tracing using Implicit KD-Trees

Ingo Wald, Heiko Friedrich, Gerd Marmitt, Philipp Slusallek and Hans-Peter Seidel

Abstract— The visualization of high-quality isosurfaces at interactive rates is an important tool in many simulation and visualization applications. Today, isosurfaces are most often visualized by extracting a polygonal approximation that is then rendered via graphics hardware, or by using a special variant of pre-integrated volume rendering. However, these approaches have a number of limitations in terms of quality of the isosurface, lack of performance for complex data sets, or supported shading models.
An alternative isosurface rendering method that does not suffer from these limitations is to directly ray trace the isosurface. However, this approach has been much too slow for interactive applications unless massively parallel shared-memory supercomputers have been used. In this paper, we implement interactive isosurface ray tracing on commodity desktop PCs by building on recent advances in real-time ray tracing of polygonal scenes, and using those to improve isosurface ray tracing performance as well. The high performance and scalability of our approach will be demonstrated with several practical examples, including the visualization of highly complex isosurface data sets, the interactive rendering of hybrid polygonal/isosurface scenes including high-quality ray traced shading effects, and even interactive global illumination on isosurfaces.

Index Terms— Ray tracing, real-time rendering, isosurface, visualization, global illumination

1 Introduction

MANY disciplines require the interactive or real-time visualization of three-dimensional scalar fields $\rho_{iso} = \rho(x,y,z)$, which are most commonly given as 3D rectilinear grids of discrete density values ρ_{ijk}. Such data sets typically originate from measurement equipment such as CT and MRI scanners, or from simulations like computational fluid dynamics (CFD).

One particularly important method for visualizing such data sets is to display one or more *isosurfaces* of certain density values, i.e. the surface $\rho(x,y,z) = \rho_{iso}$. In particular by interactively browsing through the range of possible isovalues the user can gain a good understanding of the three-dimensional structure of the data set.

Traditionally, isosurfaces of such data sets have been displayed by first *extracting* a polygonal approximation of the isosurface – usually via a variant of the marching cubes (MC) algorithm [1], [2] – and then rendering the resulting polygons using graphics hardware. This approach however has several drawbacks: First, even moderately complex data sets result in millions of polygons, which are still challenging for current rasterization hardware; large-scale data sets then easily result in a prohibitive amount of triangles. For example, a single time step of the LLNL volume data set [3] results in roughly half a *billion* triangles after tessellation.

Second, the polygonal approximation is only valid for one single isovalue, and has to be regenerated every time the user changes the isovalue. Since extracting the isosurface is quite costly for non-trivial data sets, the whole procedure is problematic for interactive applications in which the users tend to 'browse' the data by modifying the isovalue interactively.

Finally, the set of visual effects that can be simulated with the standard approaches is limited. In particular, it is not easily possible to display an isosurface with advanced lighting effects such as smooth shadows or global illumination. This is quite unfortunate, as such global effects add important visual cues that are important for a good understanding of the data set (see Figure 1).

1.1 Isosurface Ray Tracing

An alternative to isosurface extraction is to *directly* compute the isosurface, either by some form of pre-integrated direct volume rendering [4], [5], or by ray tracing, i.e., by computing the intersection of rays with the implicit function $\rho(x,y,z) = \rho_{iso}$. Due to the high computational cost, isosurface ray tracing was first realized on supercomputers by Parker et al. [6], [7], but is nowadays also feasible on modern GPUs as well [8]. Compared to extracting an explicit tessellation of the isosurface, direct ray tracing has several advantages: First of all, ray tracing directly supports global effects like shadows, reflections, or global illumination, and can be used for realizing any desired shading effect.

Second, ray tracing does not rely on a polygonal approximation of the density function, as it computes an intersection with the tri-linearly interpolated sample points themselves. Thus, compared to MC-style algorithms the surface is always smooth, can be a highly curved cubic function in each voxel (as compared to a set of flat polygons), none of the topological ambiguities of MC can appear, and even higher-order interpolations with improved quality features are possible [9].

Ingo Wald and Hans-Peter Seidel are with the MPII Saarbrücken, Germany, {wald,hpseidel}@mpi-sb.mpg.de.
Heiko Friedrich, Gerd Marmitt, and Philipp Slusallek are with the Computer Graphics Group at Saarland University, Saarbrücken, Germany, {heiko,marmitt,slusallek}@graphics.cs.uni-sb.de.

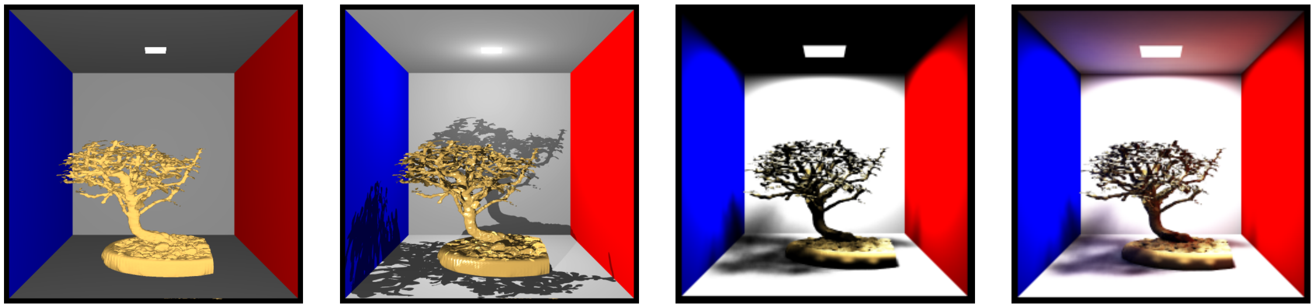


Fig. 1. Impact of ray traced lighting effects. a.) Bonsai model in a simplified Cornell box, with direct ray casting only (corresponding to an isosurface extracted via Marching Cubes and rendered via graphics hardware). b.) Ray traced, with (hard) shadows from two point lights. c.) With smooth shadows from an area light source, d.) including indirect illumination from the surrounding walls. As can be seen, shadows and global effects add important visual cues that can significantly improve the impression of shape and depth of a model.

Third, ray tracing scales nicely with scene complexity [7], and thus can efficiently cope with the currently observable explosion in data set complexity. Additionally, software ray tracing is not restricted by the the size of GPU memory, but can rather make use of the usually much larger main memory of the host computer(s).

Finally, not using a polygonal approximation avoids any need for extracting a new approximation every time the isovalue changes, and thus allows for arbitrary modifications to the isovalue at any time.

Despite all these advantages, ray tracing is also quite costly, and is commonly considered too slow for interactive use except on massively parallel shared-memory supercomputers [6], [7]. The massive compute power of modern GPUs now also allows for interactive isosurface ray casting performance on commodity GPUs [8], [10], but these approaches are not yet flexible enough for full-featured ray tracing.

With the recent advances in both ray tracing technology [11], [12] and CPU compute power, real-time performance for full-featured ray tracing has recently become possible also on commodity PCs. However, these advancements in ray tracing performance so far have been limited to polygonal ray tracing. In this paper, we present a combination of algorithms and data structures that allow for interactive isosurface ray tracing even on commodity CPUs. Being integrated into the OpenRT real-time ray tracing engine [12], our approach allows for interactive isosurface and polygonal rendering on individual PCs or small PC clusters, including support for secondary lighting effects, global illumination, and efficient support for highly complex data sets.

2 Previous Work

For visualizing 3D scalar data sets, there are two fundamentally different approaches; direct volume rendering [13], [14] on one side, and isosurfacing on the other. For the remainder of this paper, we will not consider direct volume rendering, but instead only concentrate on visualizing isosurfaces on 3D rectilinear grids.

The interactive visualization of isosurfaces was first achieved by the seminal work of Lorensen et al.'s "Marching Cubes" algorithm [1]. Since then, their approach has been extended in many forms, e.g., in the better handling of topological ambiguities [2], higher efficiency for larger data sets [15], [16], view-dependent methods [17], and adaptive or multi-resolution methods [18]. Even though all these methods allow for handling larger data sets, they still have to deal with cracks and reconstruction artifacts, and are usually still far from interactive for high-resolution data sets.

One way of avoiding to explicitly extract the isosurface is to exploit novel features of modern GPUs (2D and 3D texturing, as well as programmable shading) using a special form of pre-integrated direct volume rendering [5], [14], [10]. Applying compression [19] and distributed rendering across GPU clusters [20] also allows for visualizing large or time-dependent data set at interactive rates. Nevertheless, the interactive visualization of large data sets, in particular with advanced shading effects, is still problematic with standard approaches.

Interactive Ray Tracing

Another way of avoiding an explicit isosurface extraction is to directly ray trace the isosurface. In particular for interactive applications, this approach has first been followed by Parker et al. [6], who exploited the inherent parallelism of ray tracing to achieve interactive performance even for highly complex data sets. Additionally, their system allowed for rendering multiple isosurfaces at the same time, combination with polygonal geometry, and high-quality shading effects including shadows and transparency. Recently, DeMarle et al. [21], [22] have ported this system to also run on PC clusters.

In the related field of interactive *polygonal* ray tracing, Wald et al. [23], [12] have explicitly targeted commodity CPUs, which nowadays offer floating point performance of several GigaFlops per CPU. However, in order to get but close to this peak performance, one has to explicitly optimize for the architectural strengths and limitations of such CPUs. In particular, this includes optimizing for memory and caching effects and to exploit SIMD extensions such as Intel's SSE, which is now also available on AMDs 64-bit Opteron CPUs [24], [25], [12].

3 Efficient Isosurface Ray Tracing using Implicit KD-Trees

In this paper, we investigate how the ideas of "Coherent Ray Tracing" for polygonal ray tracing can also be applied to isosurface data sets. In particular, this requires

- an acceleration data structure that adapts to the data set and minimizes the number of operations to be performed, if possible in a hierarchical way,
- efficient and CPU-friendly algorithms, if possible using processor-specific (SIMD) extensions, and
- special care for a cache- and memory-friendly data-layout and implementation.

In Coherent Ray Tracing, SIMD extensions could be successfully exploited by traversing *packets* of rays in a parallel. This however can be realized efficiently *only* with kd-trees, which require only one binary decision per ray in each traversal step. For grid-like data structures – which seem much more intuitive for volumetric data sets – this approach cannot easily be realized.

3.1 Kd-Trees for Isosurface Ray Tracing

Kd-trees are well-known for representing polygonal scenes, where they often outperform other data structures, as they can much better adapt to the scene's geometry [26]. This is particularly the case for scenes with highly varying primitive density, as these usually contain large regions of empty space that a well-built kd-tree can traverse with very few traversal steps.

For highly regular scenes such as a 3D volume grid of density values however these advantages cannot be exploited, and the large amount of inner nodes in a kd-tree is usually detrimental in both memory overhead and number of traversal steps. Thus, grid-like data structures are usually better traversed from cell to cell with a voxel walking algorithm such as by Amanatides et al. [27]. For example, just finding the starting voxel of a ray incurs logarithmic cost for a kd-tree, while in a grid it can be found in constant time. Similarly, regular data sets are also badly suited for the afore-mentioned data-parallel packet traversal, as all of the voxels in a regular data set are small. This makes it likely that the rays in a packet diverge and traverse/intersect different voxels, thus offering little traversal coherency.

While these arguments are undoubtedly true for regular (volume) data sets, they do not necessarily hold for the specific task of rendering *isosurfaces* defined by such a volume data set: While the set of data points in fact *is* a regular 3D grid, the isosurface defined by that data set is only located within a small subset of all cells. These cells – which in the following we will call "boundary cells" – again share many properties of primitives in polygonal ray tracing: They are irregularly distributed, sparse, and are enclosed by large regions of "empty space". This once again is the environment for which a kd-tree is ideally suited (also see Figure 2).

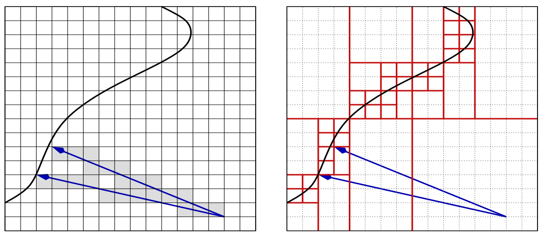


Fig. 2. The small, regular voxels in a regular volume data set offer few potential for exploiting the advantages of kd-trees and packet traversal, as rays have to perform many traversal steps, and diverge quickly. However, considering only the boundary cells of an isovalue, a kd-tree allows for quickly skipping large regions of space, and SIMD packet traversal can still be applied.

3.2 The Implicit Kd-Tree

This only leaves the question how to best use a kd-tree for representing the isosurface. If one were willing to restrict oneself to rendering a fixed isovalue only, one could identify all boundary cells in advance, build a kd-tree only over those "primitives", and expect to reap all the benefits of kd-trees also for isosurface ray tracing. Unfortunately this would no longer allow for interactively browsing the isovalue. Instead, we build a kd-tree that contains all possible isosurfaces at the same time, annotate each kd-tree node with information on what isosurfaces it contains, and perform the classification *implicitly* during traversal. Knowing which isovalues are contained within a subtree allows to easily skip entire subtrees of cells that do not contain the requires isovalue, and thus – implicitly – traverse a kd-tree only containing boundary cells of the current isovalue.

In order to realize this "implicit kd-tree", only two ingredients are required: First, a kd-tree in which each node maintains information on what isosurfaces are contained within its subtree. Second, a modified traversal algorithm that traverses a kd-tree, but implicitly classifies each visited node for whether it can actually contain the queried isovalue, and skips it if that is not the case. As the tree still encodes the whole data set, the isovalue can still be changed on the fly. As a side effect, this approach also allows for searching for several different isosurfaces concurrently within the same traversal operation, as one can trivially base the culling operation on multiple isovalues at the same time. This is particularly important for scenes in which multiple isosurfaces – e.g., both skin and bone – are of equal interest (see Figure 3).

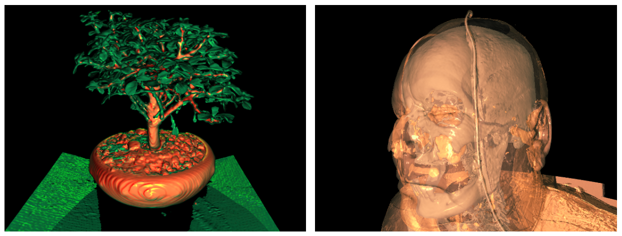


Fig. 3. Implicitly culling non-contributing branches of the implicit kd-tree during traversal also allows for rendering multiple isosurfaces at the same time. Left: The bonsai tree, with a green isosurface for the leaves, and a brown one for the trunk. Right: The Visible Female's head, with bones shining through a semitransparent skin surface.

3.3 Building the Implicit Kd-Tree

Building the implicit kd-tree in fact is quite easy. First, we build a kd-tree over all the voxels of the entire data set. This is done in a way that a kd-tree split plane always coincides with the cell boundaries of the volume cells, yielding a one-to-one mapping between the volume's cells and the voxels of the kd-tree. Currently, we always split the volume at the cell boundary that is closest in the center of the largest dimension.

Second, we annotate each cell with the range of isosurfaces contained within its subtree. In fact, this can be shown to simply be the minimum and maximum of all the densities contained within the subtree, which can be computed recursively: Each leaf node stores the min/max values of its corner densities, and each inner node stores the min/max values of its children.

Note that this data structure is similar to the one used by Wilhelm and van Gelder [28], except for that we use a kd-tree instead of an octree, and for that we use it for efficient ray traversal instead of for isosurface extraction.

4 Efficient Traversal and Intersection

Once we have defined our basic data structure and overall approach, we need to take a closer look at the core algorithms, namely ray traversal and intersection.

4.1 Single-Ray Traversal

The data structure outlined above is – except for the min/max values stored per node – very similar to the polygonal case. Thus, the already-existent polygonal traversal code requires only minimal modifications: During each traversal step we first test whether the current isovalue lies in the min/max range specified by the current node. If this is not the case, we immediately cull this subtree, and jump to the next one on the traversal stack. Otherwise, we perform exactly the same operations as in the polygonal case: Compute the distance t_d to the splitting plane, and compare the current ray segment $[t_{near}, t_{far}]$ for whether it a) overlaps the node's split plane ($t_{near} < t_d < t_{far}$); b) lies entirely in front of it ($t_{far} \leq t_d$), or c) entirely behind it ($t_{near} \geq t_d$). Just as in the polygonal case we then traverse both sides, front side only, or back side only, respectively, and update t_{near}, t_{far} accordingly for the next traversal step (see e.g. [12], [26]).

The culling can be realized by two simple integer-compares ("$\rho_{iso} \geq \rho_{min}(node)$" and "$\rho_{iso} \leq \rho_{max}(node)$") and one additional branch in each traversal step. Although these tests have to be performed in every traversal step, they are still quite affordable (see below).

4.2 Voxel Intersection

While traversing the data structure is almost the same as in polygonal ray tracing, when reaching a voxel the situation changes. While in polygonal ray tracing a voxel contains a list of triangle IDs, in isosurface ray tracing each voxel contains exactly one "primitive", i.e., a cell with a density function $\rho(x,y,z)$ that is trilinearly interpolated among its eight voxel corner densities $\rho_{ijk}, i, j, k \in 0, 1$. As this ray-voxel intersection is considerably more costly than a ray-triangle intersection we have to take special care to implement this operation efficiently.

Our system supports several different ray-voxel intersection algorithms: linear interpolation, Schwarzes's method, Neubauer iteration, and the correct root finding method by Marmitt et al. These methods are described in detail in a related publication [29], and will not be discussed here in detail.

In practice, the performance impact of the voxel intersection is usually less than 10%. Though each intersection is much more costly than a traversal step, traversals are much more common, and usually dominate the time spent on voxel intersection. Even for a closeup onto the Bonsai data set, profiling revealed only 9% of compute time spent on the voxel intersection, 67% on traversal, and the rest on shading and normal calculation.

4.3 SIMD Traversal

The same argument as for single-ray traversal holds similarly for SIMD packet traversal: Since we eventually have a kd-tree, we can again take the ideas of Coherent Ray Tracing [23], [12], and also traverse packets of rays through the kd-tree in SIMD-style.

Again, the existing code for the traversal loop (from [23]) could be mostly reused, with the same small modifications for culling as in the single-ray traversal step. In fact, the culling overhead for the SSE version is even less than for the single-ray version, as the culling overhead can be amortized over all rays in the packet.

As already discussed in the Coherent Ray Tracing paper [23], the efficiency of the SIMD packet traversal code to a large degree depends on the average utilization of the SIMD units, i.e., on the average number of rays that are active in a packet. As the voxels of a volume data set are often quite small in comparison to the screen

Screen Res.		512x512			1024x1024		
Data Set	Res.	"C"	SIMD	Ratio	"C"	SIMD	Ratio
Aneurism	256^3	19.7	5.73	3.41	78.9	21.5	3.66
Bonsai	256^3	14.7	4.73	3.10	58.7	16.8	3.49
ML	32^3	7.93	2.17	3.64	31.7	8.33	3.81
ML	128^3	11.8	3.73	3.15	47.2	13.4	3.51
ML	512^3	15.5	6.39	2.42	61.9	2.71	2.95
Female	$512^2 * 1734$	5.57	2.82	1.97	22.3	9.56	2.33
" (zoom)	$512^2 * 1734$	8.33	2.08	3.99	33.3	8.36	3.99
LLNL	$2048^2 * 1920$	15.9	9.47	1.68	65.5	31.9	1.99
" (zoom)	$2048^2 * 1920$	13.3	3.32	4.00	53.2	13.3	4.00

Tab. I. Number of traversal steps (in millions) for both single-ray and SIMD traversal, for various scenes and resolutions. While SIMD effiency is quite low (down to an average of 1.99 active rays out of 4 rays) for distant views of high-resolution data sets, larger screen resolutions or zooms into an object result in a reduction of 3.5-4.

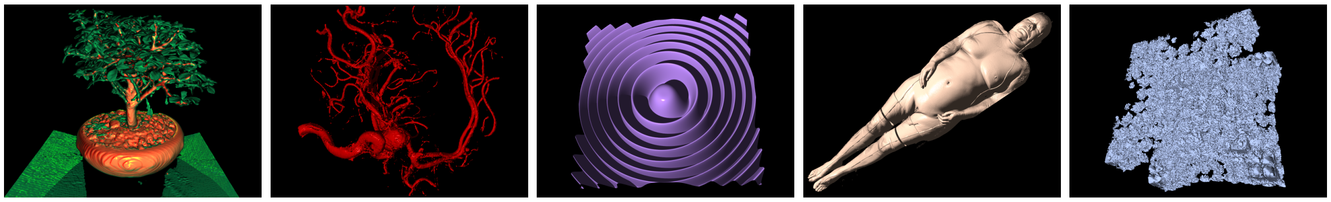


Fig. 4. The data sets used for our experiments: The bonsai tree (256^3), the Aneurism (256^3), various resolutions of the synthetic Marschner-Lobb data set (from 32^3 to 1024^3), the Visible Female ($512^2 \times 1734$), and the Lawrence-Livermore (LLNL) Richtmyer-Meshkov simulation ($2048^2 \times 1920$) These data sets have been carefully selected to cover a wide range of different data, from low (ML) to high surface frequency (bonsai, LLNL), from medical (aneurism and female) to scientific data (LLNL), and from very small (ML32) to extremely large data sets (female, LLNL).

resolution one could naïvely expect the SIMD efficiency to be small as well, as different rays may traverse different voxels. Using a kd-tree however most of the traversal operations are performed in the upper levels of the kd-tree, for which the rays still stay together (see Table I). As expected, the SIMD traversal suffers from a lack of coherence for distant views of high-resolution data sets, in particular for low screen resolutions. For less extreme settings however, the SIMD code allows for reducing the number of traversal steps by up to a factor of 4, and in practice works quite well.

4.4 SIMD Voxel Intersection

After SIMD traversal allows for significantly reducing the number of traversal steps, it would be highly beneficial to use a SIMD variant for voxel intersections as well. Due to the high computational density of the ray-voxel intersection, a SIMD variant that intersects a packet of four rays in parallel can be implemented quite efficiently, and achieves good speedups [29]. Nonetheless, there are two (related) issues that at first made the SIMD voxel intersection problematic in practice.

First, the SIMD efficiency for voxel intersection is usually much lower than for packet traversal, as – in contrast to traversal – voxel intersection is always performed at the level where the rays are most incoherent. Therefore, "intersection coherence" is usually much lower than "voxel coherence", and often very few rays are actually still active in a ray packet when reaching the leaf cells (see Table II). Unfortunately SIMD code often bears some overhead as compared to a single-ray variant, which only pays off if the SIMD code can be used for many rays in parallel. If however only a single ray is active, any potential overhead of the SIMD implementation may even result in a reduction of the overall performance.

Second, upon successful intersection a large number of values have to be stored to update the current hitpoint information: hit flag, hit distance, local cell coordinates, and normal – for a total of 8 values per ray (128 bytes total). As we may store these values only for those rays that actually had an intersection, each of these stores in SIMD mode has to be realized with several masking operations to implement "conditional moves" [25]. These turned out to consume a significant portion of compute time in the original implementation [29], and led to significant overhead, which – combined with the low intersection coherence described above – made the original implementation quite problematic for certain settings.

Therefore, we have split the voxel intersection code into its computational core and into a result storage phase. The computational core is implemented entirely in SIMD mode using SSE "intrinsics" [30]. Due to the high computational density of this code, combined with the high floating point efficiency of intrinsics-code, this part of the code never gets slower than the single-ray "C" code, and thus can be safely used even if only a low degree of coherence is present.

The high cost of the result storage phase can be significantly reduced as well. Even if the overall SIMD efficiency is quite low, it often happens that either none, or all four of the rays had an intersection. Though checking these special cases separately is very much unlike typical SIMD coding, it significantly reduces the amount of the costly conditional moves. In combination, these two measures make the current SIMD intersection code well applicable in practice.

5 Efficient Memory Representation

So far, we have only discussed the structure of our kd-tree on an abstract level, but have not yet discussed its memory-efficient realization. In a naïve implementation, one would simply use the same optimized node layout as in [23], and simply add the two min/max values (either 8-bit or 16-bit unsigned ints) to each node. Assuming a default of 16-bit density values this naïve approach however requires 12 bytes for each node: 8

Screen Res.		512x512			1024x1024		
Data Set	Res.	"C"	SIMD	Ratio	"C"	SIMD	Ratio
Aneurism	256^3	307	186	1.65	1229	545	2.25
Bonsai	256^3	544	360	1.51	2184	1027	2.12
ML	32^3	215	79	3.20	3451	927	3.55
ML	128^3	786	381	2.06	786	381	2.06
ML	512^3	680	646	1.05	2718	1863	1.46
Female	$512^2 * 1734$	179	177	1.01	716	708	1.01
" (zoom)	$512^2 * 1734$	384	98	3.99	1535	390	3.99
LLNL	$2048^2 * 1920$	631	632	0.98	2523	2520	1.02
" (zoom)	$2048^2 * 1920$	907	228	3.98	3630	909	3.99

Tab. II. Number of surface intersection tests (in thousands) for both single-ray and SIMD traversal code, for various scenes and screen resolutions. The "ratio" column reveals the average number of active rays in a 4-ray packet. Due to the loss of coherency at the leaves, for extreme settings the number of intersections can even be slightly *higher* than in the single-ray code. For less extreme setting however the SIMD code can still achieve reasonable reductions in the the number of voxel intersections computed.

bytes for specifying the plane and pointers, plus 4 bytes for the min/max values. As a kd-tree of N leaves has an additional $N-1$ inner nodes, for N 16-bit data points we require $(2N-1) \times 12$ bytes for the kd-tree. At 2 bytes per input data value, the size of the acceleration structure would then be 12 times the size of the input data. Obviously, this 12-fold memory overhead is too high except for small data sets. For 8-bit values, the relative overhead would be even worse (20 bytes per 1 byte input data). This of course is not practical.

5.1 Reducing Node Storage

Fortunately the memory overhead can be significantly reduced: If we assume for a moment that the number of voxels in each dimension is a power of two (we will relax that constraint below), the resulting kd-tree would be a balanced binary tree, i.e., all its leaves are on the same level. In a balanced binary tree however it is easy to show that *all* the nodes in the same level l will use the same splitting dimension d_l. Therefore, we do not have to store that value once for each node, but rather can store a single dimension-value per level.

Similarly, we do not have to store the split plane position in each node, either: If level l splits $R_{x,l} \times R_{y,l} \times R_{z,l}$ voxels in the $d_l = x$ dimension, then there are only $R_{x,l} - 1$ possible split locations, and each node (i,j,k,l)[1] will use the split plane $x = x_{i,l}$. Thus, instead of storing a split in each node, we only have to store $R_{x,l}$ floats per level l. The same argument holds for $d_l = y$ and $d_l = z$.

Finally, having a balanced tree allows for performing all address computations without pointers: The address of node (i,j,k,l) is $base_l + (x + R_{x,l}(y + R_{y,l}))$, and the children of (i,j,k,l) (for splits in $d = x$ dimension) will be $(2i,j,k,l+1)$ and $(2i+1,j,k,l+1)$, respectively.

As a side effect of not storing any pointers, this approach will work unmodified (and even without any additional memory) also on a 64-bit architecture, and can thus handle extremely large data sets. In summary, we can get rid of all node description data except for the min/max values, thus save two thirds of our kd-tree data, and reduce the memory overhead from 12 to 4 (respectively from 16 to 4 for 8-bit densities).

5.2 Getting Rid of Leaves

Additionally to these savings, we can avoid storing the min/max values for leaf nodes as well, and instead compute the leaf's min/max values on the fly from the cell's corner densities. This on the fly computation of the leaves is quite tolerable, as min/max operations can be implemented quite efficiently with both C Code and SIMD extensions. Furthermore, these min/max operations have to be performed *only* for leaf traversals, which are much less common than inner node traversals. Finally, as our min/max values allow for effienctly culling non-boundary cells, almost all visited leaves also require a ray/voxel intersection, whose cost totally dominates the cheap min/max computations. As in a binary tree half of all nodes are leaves, getting rid of the leaves allows for reducing our memory requirements by another factor of two, reducing the total overhead from 12 (respectively 20 for 8-bit values) to a mere 2. Of course, a memory overhead of two is still significant, in particular when compared to the 0.5% overhead achieved by Parker et al [7]. Nonetheless, we believe a factor of two to be quite tolerable already, in particular as our hierarchical traversal scheme touches only a fraction of the overall data.

The memory overhead could be further reduced by discretizing the min/max values to, e.g., 4 bits only. So far however this compression scheme has not been investigated in full detail, and is not used in our framework.

5.3 Relaxing the "power-of-two" Constraint

As mentioned before, our memory reduction scheme requires that each level of the tree has 2^i cells, i.e. that the original data set has a resolution of $2^i + 1$ in each dimension. One simple method of making arbitrary data sets comply to this constraint would be to "pad" them to a suitable resolution.

Instead, a better solution is to *imagine* that all nodes were embedded in a larger, *virtual* grid of a suitable size that exceeds the scene's original bounding box of $[0..1]^3$, and to build the kd-tree over that virtual grid (see Figure 5).

By properly assigning the split-plane positions, we can make sure that all virtual nodes lie outside the "real" scene's bounding box of $[0..1]^3$. As the kd-tree traversal code always first clips the ray to that bounding box (see [12]), we know that rays will never be traversed outside that box, and thus can *guarantee* that no ray will ever touch any of these virtual nodes. As such, we do not have to store them, either. Obviously, the same argument also holds for nodes on inner levels as well.

All that has to be done to use this scheme for a data set

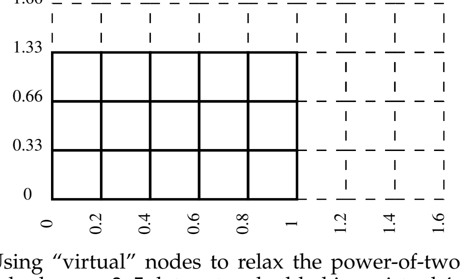


Fig. 5. Using "virtual" nodes to relax the power-of-two constraint: This example shows a 3x5 data set embedded in a virtual 4x8 grid with a balanced kd-tree. By cleverly choosing the split plane positions we can make sure that virtual voxels lie outside the scene bounds $[0..1]^2$, and thus will never be traversed by a ray. Thus, these nodes do not have to be stored, and thus do not consume any memory, either.

[1] (i,j,k,l) denotes the node (i,j,k) in level l.

of $R_x \times R_y \times R_z$ cells is to find $R'_{x,y,z} = min\{2^i | R_{x,y,z} \leq 2^i\}$, and just build the kd-tree over this virtually padded volume $R'_x \times R'_y \times R'_z$, while still doing the address computations and memory allocation with the (unpadded) original resolutions of R_x, R_y, and R_z.

Using our compression scheme, we can reduce the memory overhead of our kd-tree from 12–20 to a mere 2 (see Table III), independent of the data set's resolution. This has shown to be quite tolerable for being able to use our fast, kd-tree based algorithms.

Scene	Data	Raw	Fat	Slim			
				w Leaves		w/o Leaves	
	Bits	Data	Mem	Mem	Ratio	Mem	Ratio
Bonsai	8	16MB	316MB	64MB	5	32MB	10
Aneurism	8	16MB	316MB	64MB	5	32MB	10
ML 32^3	16	65KB	680KB	220KB	3	110KB	6
ML 128^3	16	4MB	46MB	15MB	3	7.8MB	6
ML 512^3	16	256MB	3GB	1GB	3	509MB	6
Female	12(16)	900MB	–	3.4GB	–	1.7GB	–
LLNL	8	8GB	–	36GB	–	18GB	–

Tab. III. Memory savings of the compressed ("slim") vs. the naïve ("fat") implementation. The slim representation can achieve memory reductions of up to a factor of 10. Note that both the female and LLNL data sets cannot be rendered at all with the naïve representation, as the address bits in the fat node layout do not suffice for addressing as large data sets. The slim variant does not use any pointers at all, and thus can be used for arbitrarily sized data sets.

5.4 Traversal Overhead of the Compressed Kd-Tree

Unfortunately, such significant memory savings rarely come for free: Whereas the uncompressed "fat" variant can use almost exactly the same traversal code as the original implementation [23], the compressed "slim" variant requires additional operations in each traversal step for the address computations. In particular, it requires tracking and updating the four (i, j, k, l) indices of the current node, as well as several integer multiplications and additions for computing the childrens' address. Additionally, tracking a voxel by four indices instead of only one address requires additional stack operations. As these additional operations have to be performed for each traversal step, they can have a notable impact on total rendering performance.

As can be seen in Table IV, the slim variant shows an overhead of roughly 40 to 60 percent as compared to the fat variant. As expected, the overhead is slightly less for the SIMD code, as the latter allows for amortizing address computation overhead over all rays in the packet.

Overall, an overhead of at most 68 percent is quite a reasonable price for a memory reduction by a factor of up to 10. In particular for large models such as the visible female or the LLNL data set, the slim representation is the only reasonable alternative, as the high memory requirements of these scenes did not allow for rendering using the fat node layout at all. Therefore, we usually use the slim variant, except for very small data sets.

Scene	C			SIMD		
	Fat	Slim	Overhead	Fat	Slim	Overhead
Aneurism	1.57	0.99	1.59	3.44	2.24	1.54
Bonsai	1.79	1.14	1.57	2.91	2.1	1.39
ML 32^3	2.47	1.47	1.68	4.92	3.41	1.44
ML 128^3	1.86	1.14	1.63	2.93	2.14	1.37
ML 512^3	1.30	0.91	1.43	1.62	1.24	1.31

Tab. IV. Performance (in fps) of the compressed ("slim") vs. the naïve ("fat") kd-tree, for both single rays and SIMD code, measured at 512×512 pixels. Larger scenes (such as female and LLNL) could not be rendered with the fat kd-tree, due to too high memory requirements.

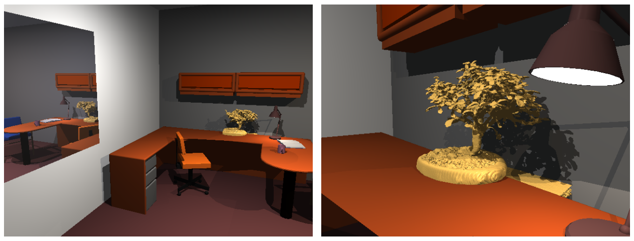


Fig. 6. Ray tracing in a hybrid polygonal/isosurface scene, showing the "bonsai" isosurface data set in the polygonal "office" scene. Note how shadows and reflections are computed correctly between both isosurfaces and polygons. a.) Overview. b.) Zoom onto the bonsai tree. On a single PC, these scenes render at 0.9 and 1.4 fps including shadows and reflections at 640×480 pixel, and higher frame rates can be achieved by using the parallelization features of OpenRT.

6 Integration into the OpenRT Engine

Using similar algorithms and data structures as the original coherent ray tracing system, the implicit kd-tree can be seamlessly integrated into the RTRT/OpenRT framework [12]: In order to support dynamically changing scenes, the OpenRT system uses a two-level hierarchy in which the lower levels of the hierarchy represent polygonal meshes, which have then been efficiently organized in an upper-level kd-tree [12]. This two-level structure has been modified to also support isosurface "objects" in the lower hierarchy level. Most core data structures (e.g. ray, hit info, shader and scene access) are shared between the polygonal and the isosurface part. Similarly, both parts share exactly the same external interface, e.g. for shooting secondary rays. This allowed the integration to be minimally intrusive, and most parts of the overall system (e.g., shaders and application frontend) do not know about different object types at all.

Obviously, a tight integration implies that all aspects of the OpenRT framework continue to function as before: Picking, parallelization, indirect effects like shadows and reflections, occlusion culling and early ray termination, all kinds of shaders (including even global illumination) etc. all continue to function on isosurfaces as they did on polygons. In particular, isosurfaces and polygons fit seamlessly together, i.e., a polygon may be reflected off of an isosurface, and an isosurface may cast a shadow on any other kind of geometry (see Figure 6).

7 Experiments and Results

Once all the ingredients of our real-time isosurfacing system are available, we can start evaluating its performance. In particular, we are interested in its absolute performance, its scalability behavior, and in its applicability for practical applications. If not mentioned otherwise, for the following experiments we use a *single* dual-1.8 GHz AMD Opteron 246 desktop PC with 6 GB RAM, rendering at a default resolution of 512×512 pixels.

7.1 Overall Performance Data

First of all, we quantify the overall performance of our system for different data set. As can be seen from Table V, at a default resolution of 512×512 pixels we can achieve interactive performance for all tested scenes even on a single PC. Note that this PC is not even state of the art any more, as 2.4 GHz Opterons and quad-PCs are already available, and even dual-core CPUs are about to enter the market soon.

Additionally, higher performance can be achieved by running our framework on multiple PCs in parallel. To this end we have built a "mini-cluster" of 5 dual-1.8 GHz Opteron PCs of 2GB RAM each, linked via Gigabit Ethernet. Unfortunately, scalability could not be measured beyond that number, as only 5 such dual-Opterons have been available for testing. As can be seen from Table V, this setup allows for frame rates of up to 39 frames per second, even including the most complex data sets. Note that this compares quite favorably to previous approaches (e.g. [31], [32], [21], [22]).

Scene		Single PC			5-Node Cluster		
		C	SIMD	Ratio	C	SIMD	Ratio
Bonsai	fat	3.4	5.2	1.5	16.2	24.6	1.5
Aneurism	fat	3.0	6.2	2.0	14.6	29.8	2.0
ML 64^3	fat	4.3	7.8	1.8	20.1	35.7	1.7
ML 512^3	slim	1.2	2.3	1.8	6.1	11.3	1.8
Female	slim	2.7	4.2	1.5	13.6	20.7	1.5
″ (zoom)	slim	2.3	7.9	3.5	11.2	39.1	3.5
LLNL	slim	0.9	1.3	1.5	–	–	–
″ (zoom)	slim	1.6	5.4	3.9	7.6	28.7	3.8

Tab. V. Overall rendering performance data when running our framework in various scenes including diffuse shading, for both a single (dual-CPU) PC, as well as with a 5-node dual-Opteron cluster. The overview of the LLNL data set could not be rendered, because the memory footprint at this view was larger than the 2GB RAM per client in the cluster setup.

7.2 SIMD Speedup

From Table V, we can also deduce the average speedup achieved through the SIMD variants of our algorithms. As expected, this speedup varies significantly, and strongly depends on the average projected cell size. Nonetheless, the SIMD variant for suitable configurations achieves speedups of up to 3.9, and still achieves noteable speedups even for the most extreme settings.

7.3 Scalability in Data Set Complexity

In polygonal ray tracing, one of the biggest advantages of ray tracing is its sublinear (i.e., logarithmic) scalability in model size, which is due to the use of *hierarchical* data structures such as kd-trees [26], [7].

As we now use such a hierarchical data structure as well, the same properties should also apply to our isosurface ray tracing framework. To verify this, we have generated various resolutions of the synthetic Marschner-Lobb data set, and measured both number of traversal steps and overall rendering performance. As expected, Figure 7 shows that our implicit kd-tree exhibits roughly logarithmic scalability in model size – the slight rise of the curve beyond 256^3 is most likely due to caching effects for such large models, as can be seen by the number of traversal steps (also given in Figure 7) which exhibits a perfectly logarithmic behavior. This logarithmic scalability makes our method highly suitable for extremely complex datasets: Even for an increase in data set complexity from 32^3 (3.2×10^4 cells) to a full 1024^3 (10^9 cells) – corresponding to $4\frac{1}{2}$ orders of magnitude in scene complexity – the performance only drops by a mere factor of 2.1.

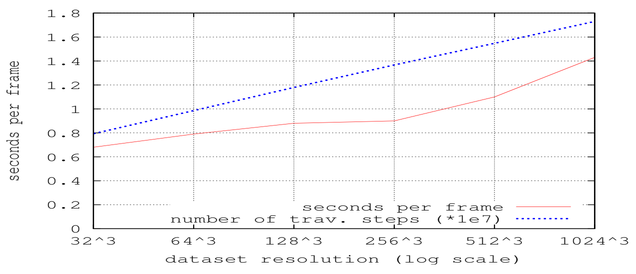


Fig. 7. Scalability of our implicit kd-tree with increasing data set resolution, measured with various resolutions of the synthetic Marschner-Lobb data set. Note the exponential scale (2^{3x}) on the x-axis, which for a roughly linear graph implies a logarithmic curve (the slight rise of the curve beyond 256^3 is most likely due to caching effects, as can be seen by the almost perfectly logarithmic number of traversal steps). Due to this logarithmic scalability, performance drops by a mere factor of 2.1 for an increase in data of $4\frac{1}{2}$ orders of magnitude. .

8 Applications

After having both described our framework and analyzed its performance, we want to briefly discuss some of the practical applications that it allows for.

8.1 Interactive Exploration of Complex Isosurfaces

Due to the logarithmic scalability in data set size (see Section 7.3), one obvious application of our framework is the interactive visualization of highly complex data sets. For example, Figure 8 shows the $512 \times 512 \times 1920$ "Visible Female" data set, rendered with different shader configurations. Except for the transparent skin example, we can use the fast SIMD code for visualizing the model, and achieve frame rates of 8.6, 5.0, and 4.0 frames per second at 640×480 pixels, respectively, even on a single PC. Due to splitting up of the rays, for the transparent skin example we had to use the single-ray code, but still – including all secondary rays – achieve 0.8 frames per second per PC. Using the distribution features of OpenRT, higher frame rates can easily be achieved by running the system in parallel (see Table V).

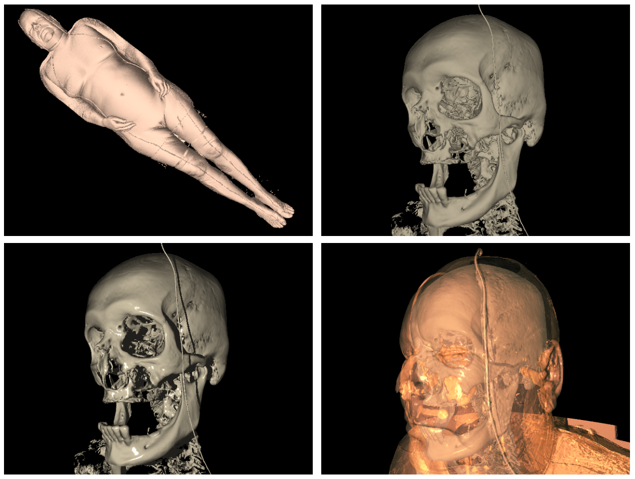


Fig. 8. The “Visible Female” (512 × 512 × 1920), rendered at 640 × 480 pixels on a single dual-1.8 GHz Opteron PC a) Overall model with direct display of the skin isosurface (8.6fps/1PC). b) Zoom onto the head with bones isovalue (5FPS/1PC). c) with additional shadows (4FPS/1PC). d.) The same, plus semi-transparent skin (0.8fps/1PC).

8.1.1 Interactive Out-of-Core Ray Tracing

As an even more interesting example, we have also rendered several slices of the Lawrence Livermore (LLNL) data set, a time dependent simulation of a Richtmyer-Meshkov instability [3], with a resolution of 2048 × 2048 × 1920 voxels for each time step.

Figure 9 shows the slice for time step 250. In tessellated form, this time step alone corresponds to roughly 470 *million* triangles, which cannot easily be rendered interactively even on the most up-to-date graphics hardware. For our implicit kd-tree, the logarithmic scalability makes data sets of even this size quite tractable. However, even with the compressed form of the kd-tree each slice still consumes 24GB of memory, which is usually not available on off-the-shelf PCs.

Fortunately, here again we can reuse features of the polygonal system: Using the same techniques developed for polygonal out-of-core ray tracing [33], we can also render this massively complex data set on a single PC with 6 GB RAM only. At 640 × 480 pixels with pure ray casting, we achieve 0.9 frames per second for the complete overview shown in Figure 9a, and 2.1 FPS when zooming in. Even when turning on shadows, we still achieve 0.3 and 1.1 frames per second, respectively.

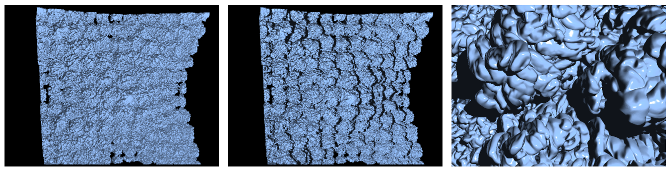


Fig. 9. The LLNL data set, a 2048 × 2048 × 1920 simulation of a Richtmyer-Meshkov Instability. a) entire data set, with ray casting only. b) Same view with shadows. c) Zoom onto the surface, to show the effect of shadows. Even such a complex data set of 24GB total can still be rendered interactively on a single PC. At 640 × 480 pixels, these images render at 0.9, 0.3, and 1.1 FPS, respectively, on a single 1.8 GHz dual-Opteron with 6 GB RAM.

8.1.2 Comparison to Graphics Hardware

In order to fully appreciate this level of performance for the complex data sets, one must compare to the standard approach of extracting a polygonal isosurface to be rendered via graphics hardware. For example, a GForce FX4000 currently delivers a theoretical peak performance of 133 million shaded and lit triangles per second. However, for the LLNL data set the tessellated isosurface consists of 470 million triangles, which would require several seconds to rasterize even under best-case assumptions. Additionally, by directly ray tracing the isosurface we can still interactively adjust the isovalue, which is not easily possible using a pre-tessellated model.

Also note that this level of performance clearly outperforms previously published isosurface ray tracing results on similar hardware [21].

8.2 Interactive Global Illumination on Iso-Surfaces

Once being able to handle shadows and reflections, it is an obvious next step to also support global illumination on isosurfaces. For that purpose, we use the “Instant Global Illumination” technique [34], [35], [12], in which the illumination in a scene is approximated using “Virtual Point Lights” generated by tracing light particles into a scene and using those for illuminating the scene.

The Instant Global Illumination algorithm is completely independent of geometry, only requires the ability to shoot rays, and thus is ideally suited for our hybrid polygon/isosurface setting. As we support four-ray packet-traversal, even the fast implementation of Benthin et al. [35] could be used without major modifications.

Figure 10 once again shows the bonsai model on the desk of the office scene, now with global illumination from three area lights turned on. As can be seen, all indirect interactions between the polygonal scene and the isosurface data set work as expected. Note that we have chosen the relatively simple bonsai model only because it best fitted the rest of the scene. More complex data sets could have been used just as well.

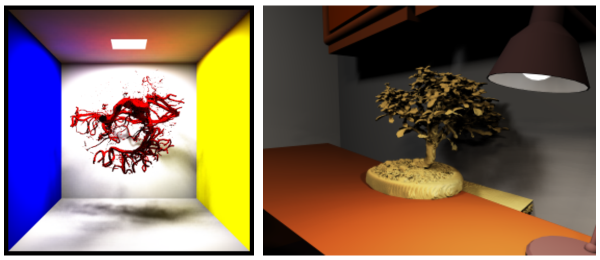


Fig. 10. Instant global illumination on isosurfaces. a) The aneurism data set in a Cornell box. Note the slight color bleeding on the ceiling, as well as the smooth shadows on the walls and the floor. b) Bonsai tree in the office scene, with smooth shadows and indirect illumination. As our method is tightly integrated into the RTRT/OpenRT system, the Instant Global Illumination implementation can be applied to our isosurfaces just as easily as originally proposed for polygons.

9 Summary and Conclusions

In this paper, we have shown how the recent advancements in polygonal ray tracing performance can be leveraged to also significantly increase interactive isosurface ray tracing performance on off-the-shelf PCs.

To this end, we have proposed using an "implicit kd-tree" for storing the data set in a hierarchical way that is well suited for efficient ray traversal, and have discussed an efficient realization of that data structure as well as the corresponding algorithms for traversing and intersecting both single rays and packets of rays.

The presented data structure and its corresponding algorithms together allow for achieving interactive isosurface ray tracing performance on individual PCs, and furthermore allow for scaling performance by running in parallel on multiple PCs. Even for highly non-trivial data sets, interactive performance can be achieved on a single dual-processor desktop PC only. Due to the good scalability in data set complexity, this level of performance can be maintained even for massively complex data sets of several Gigabytes. The new hierarchical, kd-tree based data structure, as well as the thereby-enabled processor-friendly implementation thus allow our approach to clearly outperform previously published isosurface ray tracing approaches.

While on a single PC GPU-based methods can achieve higher frame rates for small datasets, they usually do not easily scale to larger datasets, and for datasets as used in our system are often not applicable at all. Note however that our proposed methods are not limited to a CPU implementation only, but should similarly benefit GPU-based ray tracing approaches (e.g., [8]) as well.

Having never made any assumption on the isovalue, the isovalue can be interactively changed any time, and even multiple isosurfaces of the same data set can be easily rendered concurrently.

Finally, being tightly integrated into the OpenRT engine, the presented framework allows for augmenting isosurfaces with ray traced lighting effects such as transparency, shadows, reflections, refraction, and even global illumination. At the given level of performance, all these effects can be fully recomputed every frame even under interactive changes to camera, isovalue(s), or scene.

Future Work

As next steps, we want to investigate how to further reduce the memory overhead. We will also investigate further low-level optimizations. In particular, the exact caching behavior requires closer attention, in particular for complex data sets.

Finally, it is an obvious next challenge to investigate complex time-varying data sets such as the full 1.5TB LLNL dataset. In particular the hierarchical nature of our approach seems promising for this specific application.

Acknowledgments

We want to thank Ken Moreland and Randy Frank for providing access to the LLNL's Richtmyer-Meshkov data set, and the Visible Human Project for the Visible Female data. Dave DeMarle and Steven Parker have provided help in preparing the Visible Female and LLNL data.

References

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

[2] G. Nielson and B. Hamann, "The Asymptotic Decider: Removing the Ambiguity in Marching Cubes," in *Proceedings of Visualization '91*, G. Nielson and L. Rosenblum, Eds. IEEE Computer Society Press, 1991, pp. 83–91.

[3] A. A. Mirin, R. H. Cohen, B. C. Curtis, W. P. Dannevik, Andris, M. Dimits, M. A. Duchaineau, D. E. Eliason, D. R. Schikore, S. E. Anderson, D. H. Porter, and P. R. Woodward, "Very High Resolution Simulation Of Compressible Turbulence On The IBM-SP System," in *Proceedings of SuperComputing*, 1999, (Also available as Lawrence Livermore National Laboratory technical report UCRL-MI-134237).

[4] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, "Interactive Volume Rendering on Standard PC Graphics Hardware using Multi-textures and Multi-stage Rasterization," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. ACM Press, 2000, pp. 109–118.

[5] K. Engel, M. Kraus, and T. Ertl, "High-quality pre-integrated volume rendering using hardware-accelerated pixel shading," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*. ACM Press, 2001, pp. 9–16.

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

[7] S. Parker, P. Shirley, Y. Livnat, C. Hansen, and P.-P. Sloan, "Interactive Ray Tracing," in *Proceedings of Interactive 3D Graphics*, 1999, pp. 119–126.

[8] R. Westermann and B. Sevenich, "Accelerated Volume Ray-Casting using Texture Mapping," in *IEEE Visualization 2001*, 2001.

[9] 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*, vol. 10, no. 4, pp. 397–409, 2004.

[10] J. Krueger and R. Westermann, "Acceleration Techniques for GPU-based Volume Rendering," in *Proceedings IEEE Visualization 2003*, 2003.

[11] I. Wald, T. J. Purcell, J. Schmittler, C. Benthin, and P. Slusallek, "Realtime Ray Tracing and its use for Interactive Global Illumination," in *Eurographics State of the Art Reports*, 2003.

[12] I. Wald, "Realtime Ray Tracing and Interactive Global Illumination," Ph.D. dissertation, Computer Graphics Group, Saarland University, 2004, available at http://www.mpi-sb.mpg.de/~wald/PhD/.

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

[14] M. Hadwiger, "High-Quality Visualization and Filtering of Textures and Segmented Volume Data on Consumer Graphics Hardware," Ph.D. dissertation, Techniche Universität Wien, 2004.

[15] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno, "Speeding Up Isosurface Extraction Using Interval Trees," *IEEE Transactions on Visualization and Computer Graphics*, vol. 3, no. 2, pp. 158–170, 1997.

[16] Y. Livnat, H.-W. Shen, and C. R. Johnson, “A Near Optimal Isosurface Extraction Algorithm Using the Span Space,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 2, no. 1, pp. 73–84, 1996.

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

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

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

[20] M. Strengert, M. Magallon, D. Weiskopf, S. Guthe, and T. Ertl, “Hierarchical Visualization and Compression of Large Volume Datasets Using GPU Clusters,” *Parallel Graphics and Visualization*, 2004.

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

[22] D. E. DeMarle, C. Gribble, and S. Parker, “Memory-Savvy Distributed Interactive Ray Tracing,” in *Eurographics Symposium on Parallel Graphics and Visualization*, 2004.

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

[24] *IA-32 Intel Architecture Optimization – Reference Manual*, Available from http://www.intel.com/design/pentium4/manuals/, Intel Corp., 2001.

[25] Advanced Micro Devices, “Software Optimization Guide for AMD Athlon(tm) 64 and AMD Opteron(tm) Processors,” available from http://www.amd.com/us-en/Processors/TechnicalResources/.

[26] V. Havran, “Heuristic Ray Shooting Algorithms,” Ph.D. dissertation, Czech Technical University in Prague, 2001.

[27] J. Amanatides and A. Woo, “A Fast Voxel Traversal Algorithm for Ray Tracing,” in *Eurographics ’87*, 1987, pp. 3–10.

[28] J. Wilhelms and A. Van Gelder, “Octrees for faster isosurface generation,” *ACM Transactions on Graphics*, vol. 11, no. 3, pp. 201–227, July 1992.

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

[30] *Intel C/C++ Compilers*, Intel Corp, 2002, http://www.intel.com/software/products/compilers.

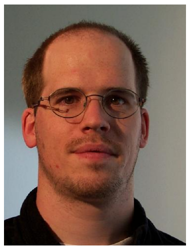
[31] A. Neubauer, L. Mroz, H. Hauser, and R. Wegenkittl, “Cell-based first-hit ray casting,” in *Proceedings of the Symposium on Data Visualisation 2002*, 2002, pp. 77–ff.

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

[33] I. Wald, A. Dietrich, and P. Slusallek, “An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models,” in *Rendering Techniques 2004, Proceedings of the Eurographics Symposium on Rendering*, June 2004, pp. 81–92.

[34] I. Wald, T. Kollig, C. Benthin, A. Keller, and P. Slusallek, “Interactive Global Illumination using Fast Ray Tracing,” *Rendering Techniques*, pp. 15–24, 2002, (Proceedings of the 13th Eurographics Workshop on Rendering).

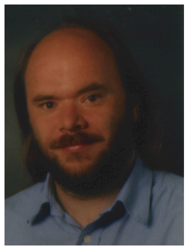
[35] C. Benthin, I. Wald, and P. Slusallek, “A Scalable Approach to Interactive Global Illumination,” *Computer Graphics Forum*, vol. 22, no. 3, pp. 621–630, 2003, (Proceedings of Eurographics).



Ingo Wald got a PhD in engineering from Saarland University in 2004, and is currently a post-doctoral researcher at the Max-Planck-Institut für Informatik in Saarbrücken, Germany. His work concentrates on realtime ray tracing on PCs and PC clusters, photorealistic rendering, scientific visualization, efficient parallel rendering, and massive model visualization. He is coordinating the OpenRT Realtime Ray Tracing Project and is co-founder and CEO of *in*Trace Realtime Ray Tracing Technologies GmbH.

Heiko Friedrich is currently a PhD student at the computer graphics group at Saarland University in Saarbrücken, Germany. In 2004, he finished his masters on realtime iso-surface ray tracing. Before doing his masters degree, he received a Bachelor from Saarland State University in 2002, and worked as an intern at the Imaging and Visualization Department of Siemens Corporate Research in Princeton/USA. His main research interest are realtime volume rendering and ray tracing.

Gerd Marmitt received his Masters Degree in computer science from Clemson University, South Carolina (USA) in 2003 after a one-year Fulbright scholarship. In the same year he received his Bachelors Degree at the University of Applied Sciences in Trier, including an internship at the Max-Planck-Institute for computer science in Saarbrücken. He is currently at the Computer Graphics Group at Saarland University, where he is holder of a scholarship funded by the German Science Fondation (DFG). His research interests are interactive volume rendering, isosurface visualization, and ray tracing.



Philipp Slusallek is full professor at the computer graphics lab of Saarland University. In 1998/99 he was visiting assistant professor at the Stanford University graphics lab. His current research activities focus on realtime ray tracing on off-the-shelf computers and on designing a hardware architecture for realtime ray tracing. Other research topics include design of a network-aware multi-media infrastructure, consistent illumination in virtual environments, physically-based and realistic image synthesis, and object-oriented software design.

Hans-Peter Seidel Hans-Peter Seidel studied mathematics, physics and computer science at the University of Tübingen, Germany. He received his Ph.D. in mathematics in 1987, and his habilitation for computer science in 1989, both from University of Tübingen. From 1989, he was an assistant professor at the University of Waterloo, Canada. In 1992, he was appointed at the University of Erlangen-Nürnberg, Germany. Since 1999 he has been director of the Computer Graphics Group at the Max-Planck-Institute for Computer Science and cross-appointed to Saarland University in Saarbrücken, Germany. In his research, Seidel investigates algorithms for 3D Image Analysis and Synthesis. This involves the complete processing chain from data acquisition over geometric modeling to image synthesis. In 2003 Seidel was awarded the ‘Leibniz Preis’, the most prestigious German research award, from the German Research Foundation (DFG). Seidel is the first computer graphics researcher to receive this award. In 2004 he was selected as founding chair of the Eurographics Awards Programme.