The Visual Computer manuscript No.
(will be inserted by the editor)

Sung-Eui Yoon · Christian Lauterbach · Dinesh Manocha

R-LODs: Fast LOD-based Ray Tracing of Massive Models

Abstract We present a novel LOD (level-of-detail) algorithm to accelerate ray tracing of massive models. Our approach computes drastic simplifications of the model and the LODs are well integrated with the kd-tree data structure. We introduce a simple and efficient LOD metric to bound the error for primary and secondary rays. The LOD representation has small runtime overhead and our algorithm can be combined with ray coherence techniques and cache-coherent layouts to improve the performance. In practice, the use of LODs can alleviate aliasing artifacts and improve memory coherence. We implement our algorithm on both 32bit and 64bit machines and able to achieve up to 2–20 times improvement in frame rate of rendering models consisting of tens or hundreds of millions of triangles with little loss in image quality.

1 Introduction

In recent years, there has been a renewed interest in real-time ray tracing for interactive applications. This is due to many factors: firstly, processor speed has continued to rise at exponential rates as predicted by Moore's Law and is approaching the raw computational power needed for interactive ray tracing. Secondly, ray tracing algorithms can be parallelized on shared memory and distributed memory systems. Therefore, the current hardware trend towards desktop systems with multi-core CPUs and programmable GPUs can be used to accelerate ray tracing. Finally, recent algorithmic improvements that exploit ray coherence can achieve a significant improvement in rendering time [22, 31].

Our goal is to perform interactive ray tracing of massive models consisting of tens or hundreds of millions of triangles on current desktop systems. Such gigabyte-sized models are the result of advances in model acquisition, computer-aided design (CAD), and simulation technologies and their complexity makes interactive visualization and walk-throughs a challenging task. In the context of rendering massive models, ray tracing has an important property: its asymptotic performance is logarithmic in the number of primitives for a given resolution. This is due to the use of hierarchical data structures such as bounding volume hierarchies or kd-trees. The asymptotic complexity makes ray tracing an attractive choice, especially for rendering of massive models.

Fig. 1 St. Matthew Model: *We use our LOD-based algorithm to accelerate ray tracing of the St. Matthew model with shadows and reflections. We render the* $128M$ *triangle model at* 512×512 *resolution with* 2×2 *anti-aliasing and pixels-of-error (PoE)* $= 4$*. We are able to achieve* $2 - 3$ *frames per second on two dual-core Xeon processors with 4GB of memory. We observe a* $2 - 20$ *times increase in the frame rate due to R-LODs with very little loss in image quality.*

The logarithmic growth, however, continues only as long as the system has sufficient main memory to contain the entire model and hierarchical data structures. As models grow much larger, the size of the hierarchical structure also increases linearly and the underlying ray tracer performs its computations in an out-of-core manner, slowing down drastically. A major trend in computing hardware has been the increasing gap between processor speed and memory speed. Moreover, disk I/O accesses are in general more than three orders of magnitude slower than main memory accesses. Because of these gaps, hardware advances are not expected to provide an efficient solution to the problem of ray tracing massive models.

Main Contributions: We present a new algorithm to accelerate ray tracing of massive models using geometric levels-of-detail (LODs). Our approach computes simple and drastic

Sung-Eui Yoon
Lawrence Livermore National Laboratory
E-mail: sungeui@llnl.gov

Christian Lauterbach · Dinesh Manocha
University of North Carolina at Chapel Hill
E-mail: cl,dm@cs.unc.edu

simplifications, called *R-LODs*, of the polygonal model. The R-LODs have a compact representation and are tightly integrated with the kd-tree. We present a simple and efficient LOD error metric to bound the error for primary and secondary rays. Additionally, we use techniques based on ray coherence and cache-oblivious layouts to improve the performance of our LOD based ray tracing algorithm. R-LODs also alleviate the temporal aliasing that can arise during rendering of highly tessellated models.

We have implemented and tested our system on two machines running Windows XP 32-bit and 64-bit with two dual-core Xeon CPUs and have evaluated its performance on different kinds of models with $20 - 128M$ triangles. The performance gain of our LOD based ray tracer is proportional to the reduction in the working set size and the number of intersection tests. The frame rate improvement varies from 2 times on models with small working set size to almost $20 - 50$ times on models with very large working set size.

Our ray tracing algorithm offers the following *benefits*:

1. **Simplicity:** R-LODs are very easy to implement and their representation has small runtime overhead. Our algorithm maintains the simplicity, coherence, and performance of the kd-tree data structure.
2. **Interactivity:** The LOD based ray tracer provides a framework for interactive ray tracing due to the fact that we can trade off image quality for improved frame rate.
3. **Front size:** R-LODs reduce the size of the front traversed in the kd-tree. This results in fewer ray intersection tests and decreases the size of the working set.
4. **Coherence:** R-LODs make memory accesses more coherent and reduce the number of L1/L2 cache misses and page faults. Furthermore, they can also improve the performance of ray coherence techniques.
5. **Generality:** Our algorithm is applicable to a wide variety of polygonal models, including scanned and CAD models.

Organization: The rest of the paper is organized in the following manner: Section 2 gives a brief summary of prior work in interactive rendering. We give an overview of our approach in Section 3 and present the R-LOD representation and computation algorithm in Section 4. Section 5 shows acceleration techniques based on cache-coherent layouts and ray coherence. We describe the implementation of our ray tracer and analyze its performance on different models in Section 6. Finally, Section 7 compares our algorithm with other approaches.

2 Related work

In this section, we give a short overview of interactive ray tracing and the use of LODs for interactive rendering.

2.1 Interactive Ray Tracing

Ray tracing was introduced by Appel [3] and Whitted [36] and is a very well studied field. In this section, we just briefly survey some recent techniques used to accelerate ray tracing, but a detailed description is available in [27]. At a broad level, we classify prior approaches into four categories:

Exploiting ray coherence: The underlying idea here is not to trace each ray by itself, but to utilize the fact that neighboring rays exhibit spatial coherence. Earlier attempts to exploit this concept were beam tracing [11], pencil tracing [26] and cone tracing [2]. More recently, Wald *et al.* [31] group rays into bundles and use them to accelerate traversal and intersection with primitives for all rays simultaneously by taking advantage of SIMD instructions. Reshetov *et al.* [22] propose an algorithm to integrate beam tracing with the kd-tree spatial structure and were able to further exploit ray coherence.

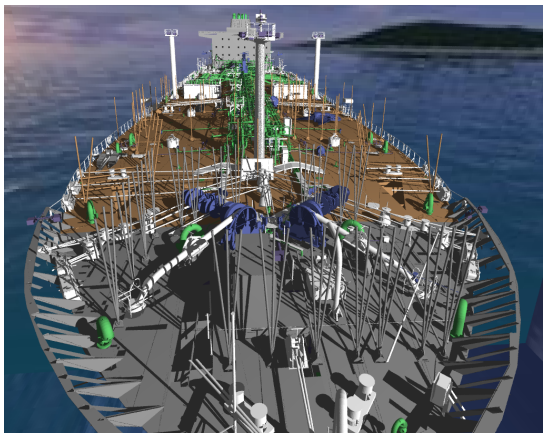


Fig. 2 Double Eagle Tanker: *The deck of the Double Eagle tanker with shadows is shown using ray tracing. We are able to achieve* 1-3*fps at* 512 *by* 512 *image resolution with* $2x2$ *super-sampling and PoE* $= 4$ *on a dual Xeon workstation. In this model, the working set of the ray tracer is low and we are able to achieve up to* 2 *times improvement in the frame rate.*

Hardware acceleration: Another trend has been to use hardware support to accelerate ray tracing. Purcell *et al.* [21] show that ray tracing could be adapted to the streaming model of current programmable GPUs, which are mainly designed for rasterization. Schmittler *et al.* [25] and Woop *et al.* [38] present prototypes for a complete and programmable ray tracing hardware architecture to run at interactive rates.

Parallel computing: Ray tracing is easily parallelizable due to the fact that all rays can be traced independently. Parker *et al.* [19], DeMarle *at al.* [7], and Dietrich *et al.* [8] describe an interactive ray tracer for rendering large scientific or CAD datasets running on shared memory or distributed architectures. Wald *et al.* [34] built a ray tracer to run on clusters of commodity hardware machines and were able to achieve interactive frame rates for large architectural and CAD models. Both of these systems are mainly intended for models that could be kept in the main memory of a shared memory system or of PCs used in the cluster.

Large datasets: Many algorithms have been presented to improve the performance of ray tracing on large datasets [7, 10, 20, 32]. Our approach is complimentary to these methods and can be combined with them to further improve the performance.

2.2 Interactive Rendering using LODs and Out-of-Core Techniques

LODs have been widely used to accelerate rasterization of large polygonal datasets [16]. At a broad level, prior algorithms can be classified into static LODs, view-dependent simplification, image-based representations and hybrid combinations of geometric and image-based representations. Out-of-core algorithms are an active area in computer graphics and visualization with the goal to efficiently handle large datasets [4]. LOD algorithms can be combined with out-of-core techniques to rasterize large polygonal datasets composed of tens or hundreds of millions of polygons at interactive rates on commodity PCs [23, 6, 42, 9].

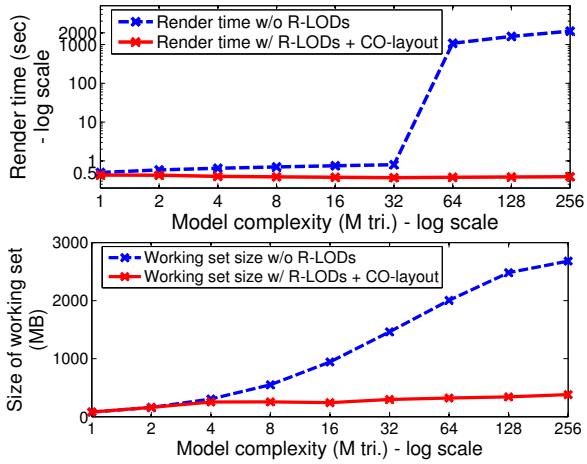


Fig. 3 Performance of Ray Tracing: *We precompute simplified versions of the St. Matthew model and ray trace each simplification separately from the same viewpoint. We measure the frame time and working set size of the ray-tracer, with and without R-LODs by using different simplified models on a* 64*-bit machine with* 2*GB RAM. Notice the big jump in frame time for ray tracing without R-LODs, as the working set of ray tracing with massive models exceeds the available RAM. On the other hand, our R-LOD based ray tracing combined with cache-oblivious layouts (CO-layout) achieves near-constant performance in terms of frame rate and the working size. The cache-oblivious layouts increase the performance of our LOD ray tracer by* $10-60\%$ *over the depth-first layout.*

LOD-based based algorithms can also be applied to accelerate ray tracing. Christensen *et al.* [5] introduce a LOD approach for an offline ray tracer based on ray differentials [12]. Wand and Straßer [35] propose an algorithm for multi-resolution ray tracing of point-sampled geometry based on ray-differentials. Another approach is to integrate the LODs into the hierarchical structure [37]. Recently, Stoll *et al.* [28] proposed a novel architecture for dynamic multiresolution ray tracing. They proposed a watertight multiresolution method by interpolating between discrete LODs for each ray. Their discrete LODs are computed from choosing proper tessellation levels for subdivision meshes. Also, efficient algorithms based on depth images can be used to accelerate ray tracing [15, 1].

3 Overview

In this section, we discuss many issues that govern the performance of ray tracing and give an overview of our approach.

3.1 Ray tracing of massive models

In this paper, we restrict ourselves to triangulated models, though our approach can also be extended to other primitives such as point clouds. All efficient ray tracers employ hierarchical data structures to avoid testing each ray with every primitive. We use the kd-tree, which is a special case of the general BSP tree and has recently become a popular choice due to its simplicity and performance [10, 27]. Each node of the kd-tree represents one subdivision of the parent's space and contains information about the axis-aligned plane used for the split as well as pointers to its child nodes. We use the optimized representation proposed by Wald *et al.* [30] and extend it to efficiently handle LODs.

Out-of-core ray tracing: Ray tracers taking advantage of hierarchical data structures should exhibit a logarithmic growth rate as a function of the model complexity [31]. We measured the performance of a coherent ray tracer during rendering different simplification levels of the St. Matthew model, as shown in Fig. 3. Our experiment indicates that ray tracing performance increases as a logarithmic function of model complexity as long as the kd-tree and primitives of a model can fit in the main memory. However, once the model size and the working set size of the kd-tree exceeds the available main memory of the machine, the disk I/O significantly affects the performance of the ray tracer.

Ray coherence: Recent approaches that exploit spatial and ray coherence decrease the number of memory accesses and therefore also the number of disk accesses for large models [31, 22]. These algorithms perform traversals and intersections for multiple spatially-coherent rays in a group at the same time. In general, rays in a group exhibit higher coherence at the higher levels of the kd-tree (that usually are in main memory) because each ray in the group is likely to follow same path in the tree as other rays. However, accesses to the nodes deeper in the tree are incoherent and, thus, result in disk cache misses, especially when dealing with massive models, since bounding box of those nodes become smaller compared to width of the ray group. Therefore, in order to accelerate out-of-core ray tracing, we need to reduce the number of accesses made to the nodes deeper in the tree.

3.2 Our Approach

We mainly address the problem of ray tracing massive models. If models have high depth complexity, current traversal algorithms based on kd-trees can efficiently handle such kinds of models. In this case, the working set size is proportional to the number of visible primitives from the primary and secondary rays. Therefore, we primarily deal with the problem of fast ray tracing when the number of visible primitives is high.

We assume that each ray or ray bundle is represented by a pyramidal beam or frustum. As described in the multi-level ray tracing of Reshetov *et al.* [22], during traversal the frustum is checked for intersection with the bounding box of the current kd-tree node by using an inverse frustum culling approach. This results in two interesting cases:

1. Models with large primitives: If the bounding box of the node is larger than the frustum, it is likely that the node intersects with the whole beam, i.e. we can exploit spatial coherence. Typically, architectural models or CAD models result in such cases whenever the model is coarsely tessellated, has large planar primitives or is viewed at close range.

2. Highly tessellated models: In this case, the bounding box is much smaller than the frustum. This implies that the beam needs to be split into smaller sub-beams. However, if the beam represents just one ray, then further subdivision is not possible, even though the sub-tree represented by the node has a high number of descendants and, thus, there is high local geometric complexity. Therefore, ray coherence approaches like multi-level ray tracing and ray packet tracing fall back to normal ray tracing and may not offer much benefit. For example, consider ray tracing a St.Matthew model consisting of 128M triangles at a resolution of 1024^2 primary rays. Assume that every ray hits the model and half of the model's triangles are visible to the eye. In this case, fewer than 1% of the actual triangles are hit by one of the rays. Moreover, each of these triangles is sampled as a representative of several triangles in the subtree. This has two consequences: first, the memory accesses may be *incoherent* because each triangle may lie in a different part of memory.

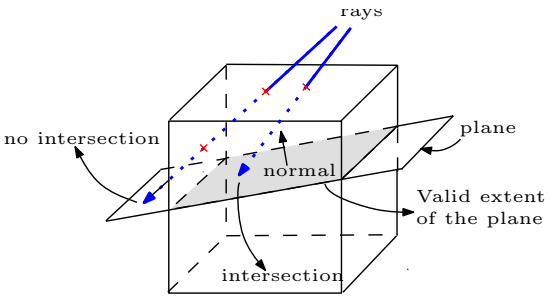


Fig. 5 LOD Representation: *A R-LOD consists of a plane with material attributes. It serves as a drastic simplification of triangle primitives contained in the bounding box of the subtree of a kd-tree node. Its extent is implicitly given by its containing kd-node. The plane representation makes the intersection between a ray and a R-LOD very efficiently and results in a compact representation.*

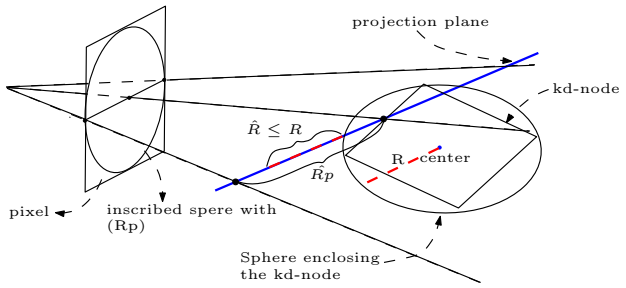


Fig. 6 Projection-based LOD Metric: *We place a projection plane at the intersection point between a ray and the kd-node. The plane is orthogonal to the ray. Based on this projection plane, we conservatively check whether the R-LOD satisfies the error metric.*

Secondly, *temporal aliasing* can occur between frames since it is likely that a different triangle will be chosen in successive frames.

Our novel LOD-based ray tracing algorithm handles this second case by choosing our precomputed R-LOD representation when traversal determines that a LOD metric is satisfied. This means that traversal can stop before reaching the deep levels of the tree, reducing the number of incoherent accesses and the size of the working set, while maintaining ray coherence so that related techniques still work well. As a result, we obtain significant improvements in rendering speed.

4 LOD-Based Ray Tracing

In this section we present the R-LODs that are used to accelerate ray tracing. We first describe our R-LOD representation and the modified traversal algorithm. Then we present our LOD error metric and the R-LOD construction algorithm.

4.1 R-LOD Representation for Ray Tracing

Our goal for interactive ray tracing is to design a LOD representation that retains the benefits of kd-tree based acceleration algorithms, i.e. simplicity, efficiency and low runtime overhead.

A R-LOD consists of a plane with material attributes (e.g. color), which is a drastic simplification of the descendant triangles contained in an inner node of the kd-tree, as shown in Fig. 5. Each R-LOD is also associated with a surface deviation error which is used to quantify the projected screen-space error at runtime.

Let us assume that the original tree has height h, where $h \approx \log_2(n)$, and n is the number of triangles in the original model. The R-LOD associated with a kd-tree node at height k is a simplification into a plane of the 2^k descendant triangles. Our choice to use such a representation is motivated by the following goals:

Simple and efficient LOD representation: Current ray object intersection algorithms based on the kd-tree are highly optimized for interactive ray tracing. We use simple representations for LODs to minimize storage and traversal overhead. Each R-LOD adds 4 bytes to an inner node of the kd-tree. We also use a simple and fast LOD selection algorithm to reduce the traversal overhead.

Drastic model simplification: The computational workload of ray tracing is a logarithmic function of the model complexity. If the model size is reduced by a factor of m, the tree traversal overhead reduces by only $\log(m)$. As a result, m has to be a significant number, say 2^3or 2^4.

Error-controllable LOD rendering: In order to control the errors caused by using R-LODs, we associate a deviation error metric and compute a screen-space projection in terms of *pixels-of-error* (PoE) deviation. Also, we assume that our drastically simplified LOD representations are mainly used given small PoE values (e.g., 1–4 pixels at image resolution 1024×1024) for high-quality rendering. However, one can efficiently explore a model with a high PoE; once a desired view is found, high-quality image can be acquired by setting a low PoE.

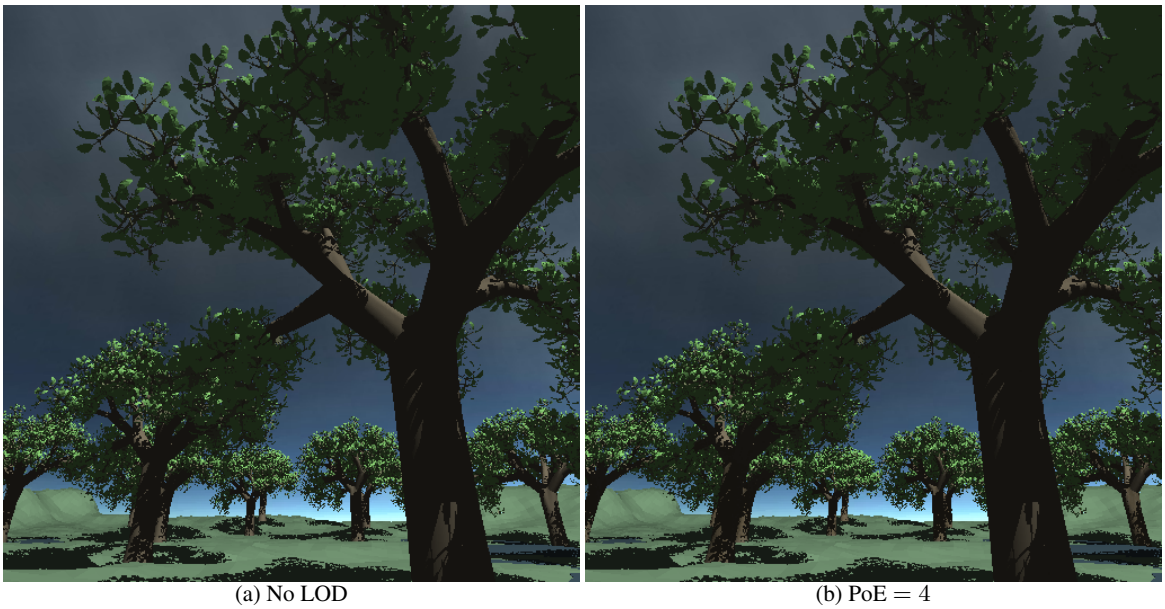
4.2 Runtime Traversal with R-LODs

Our new traversal algorithm is a modification of the efficient traversal algorithm described in Wald's thesis [30] and [29]. We recursively traverse the kd-tree from the root node or the entry-point that is computed using multi-level ray tracing. When we reach an intermediate node associated with a R-LOD, we check whether we can use the R-LOD based on our LOD error metric. If the current R-LOD satisfies the LOD error metric, we perform an intersection test between a R-LOD and the ray. If there is an intersection, we stop the traversal and return the intersection data of the R-LOD to compute shading and shoot secondary rays, if necessary. If there is no intersection, the algorithm does not traverse the child nodes of the intermediate node associated with the R-LOD. Each R-LOD is bounded by a kd-node and therefore, the extent of the plane of the R-LOD is implicitly bounded by the kd-node during tree traversal. The implicit extent of the plane results in a compact R-LOD representation.

4.3 LOD Error Metric Evaluation

We use a projection-based algorithm integrated with surface deviation error to select appropriate LODs for ray tracing.

Conservative projection algorithm: We use a projection method to efficiently compare the screen-space area of the R-LOD after the perspective projection with the PoE in the screen-space. Conceptually, we position a projection plane at the intersection between the ray and the kd-tree node. The plane is set to be orthogonal to the ray, as shown in Fig. 6. We enclose the R-LOD (and its corresponding simplified geometry) in a sphere. The area of the R-LOD projected onto the projection plane is conservatively measured by computing πR^2, where R is the radius of the sphere. Let R_p be the radius of a sphere inscribed in a rectangular shape pixel of the image screen. In this case, R_p is simply half of the width of the pixel. Then, the projected area of a pixel in the pro-



(a) No LOD (b) PoE $= 4$

Fig. 4 Forest Model: *We render the forest model consisting of* 32 *million triangles with shadow rays using PoE* $= 0$ *and PoE* $= 4$*. The image resolutions are* 512×512 *without anti-aliasing to highlight image quality differences. We are able to render the model given the viewpoint at the* 1.6 *frames per second and achieve* 5 *times improvement by using R-LODs.*

jection plane satisfies the following relationship:

$$\frac{d_{near}}{R_p} = \frac{d_{min}}{\hat{R}_p} \Rightarrow \hat{R}_p = d_{min}\frac{R_p}{d_{near}} = d_{min}C, \quad (1)$$

where $\hat{R}_p$ is the projected radius of R_p, d_{near} is the distance from the viewer to the image plane, and d_{min} is the distance from the viewer to the intersection point between the ray and the kd-node. Since $\frac{R_p}{d_{near}}$ $(= C)$ is a constant, the projected radius, $\hat{R}_p$, is a simple linear function of the distance, d_{min}, along the ray from the eye to the intersecting node. We select an R-LOD if $\hat{R}_p$, is bigger than the radius, R, associated with the R-LOD. Our LOD metric is very efficient as it requires only one multiplication and d_{min} is already known during the tree traversal.

Surface deviation: The error metric described above conservatively measures the projected screen-space area of the R-LOD. We augment the metric to take into account the surface deviation of a R-LOD. For this we first measure the surface deviation between the plane of the R-LOD and all the contained triangles. We combine the surface deviation and the projected screen-space area of the R-LOD in the following geometric formulation. We compute the volume of the surface deviation along the plane and add this volume to the volume of the sphere enclosing the R-LOD. We then treat the summed volume as a volume of an imaginary sphere and use its radius as the error bound of the R-LOD. In this geometric formulation, these two seeming different error bounds can be treated in a uniform manner.

Error quantization: The exact representation of the plane and associated materials takes 32 bytes. Instead of directly associating this information with each node of the kd-tree, we quantize the error bounds associated with the R-LODs and store the quantized error bound as well as an R-LOD index in a 4 byte structure as the part of the kd-node in order to reduce the working set size during traversals. Therefore, only if the error bound of an R-LOD is satisfied within the PoE bound, we load the exact R-LOD representation by using the R-LOD index. When considering a path from a leaf node of the kd-tree to the root node, the error bounds associated with the nodes increase as a geometric series. Therefore, we use a geometric distribution equation to quantize the error values associated with the R-LODs. We found that 5 bits are enough (i.e. 10%–20% quantization error) to conservatively quantize the error bound of the R-LODs in our benchmarks; therefore, each R-LOD index is stored in 27 bits, which are enough to indicate all the R-LODs in our tested models.

Secondary rays: Our LOD metric based on conservative projection also extends to secondary rays. These include *reflection* (in which a ray reflects at an intersection point with a reflective triangle) and *shadow* rays. This is mainly because these secondary rays can be expressed as a linear transformation [11]. In the case of reflection, the radius, R_p, of the sphere inscribed in the pixels of the image space increases linearly based on the sum of the distance from the viewer to an intersecting reflective triangle, and to another intersecting object along the primary or reflective secondary rays. Similarly, our metric also works well for shadow rays and again we use a linear transformation. One issue with using LODs for shadow rays is that they can result in self-shadowing artifacts when different versions of the R-LODs are selected by the primary ray and the shadow ray. We overcome this problem by ignoring the intersections between the shadow ray and the primitives that are within the LOD error bounds associated with the R-LOD selected by the primary ray.

Our projection-based method does not work with *refraction*, since refraction is not a linear transformation [11]. In this case, we can use a more general, but expensive method based on ray differentials [12], to decide whether an R-LOD satisfies the PoE bound after refraction.

4.4 R-LOD Construction

Our goal is to compute a plane that approximates the triangles that are contained in the subtree of an intermediate kd-node and also their material properties. If a triangle contained in the subtree is not fully contained in the bounding box of the node, we clip the triangle against the box and do not consider the clipped portion of the triangle. We use principal component analysis (PCA [13]), to compute the plane. PCA computes the eigenvectors that provide a statistical description of input points. We perform PCA computation based on the vertices of the triangles, but also take into account the size of the triangles by associating the area of the triangle as a weight for each vertex. The plane is computed based on the eigenvector associated with the largest eigenvalue and this eigenvector represents the normal to the plane[1]. We compute material properties that are mean values of the contained triangles and associate them with the R-LOD. The surface deviation of the plane against the geometric primitives is computed based on the smallest eigenvalue, which corresponds to a variance of geometry along the normal of the plane.

Hierarchical R-LOD computation: We can compute the R-LODs associated with each node of the tree in a bottom-up manner. However, a naive algorithm would compute the R-LOD for each node independently and this can result in a $O(n \log n)$ algorithm.

Instead, we present a R-LOD computation algorithm that has linear time complexity and is well suited for out-of-core computation. Each element, σ_{ij}, of (i,j)th component of a covariance matrix for PCA is defined as the following:

$$\sigma_{ij} = \sum_{k=1}^{n} (V_i^k - \mu_i)(V_j^k - \mu_j), \tag{2}$$

where V_i^k is the ith component (e.g. x, y, and z) of kth vertex data, μ_i is the mean of V_i^k, and n is the number of vertices. This equation can be reformulated as:

$$\sigma_{i,j} = \sum_{k=1}^{n} V_i^k V_j^k - \frac{2}{n} \sum_{k=1}^{n} V_i^k \sum_{k=1}^{n} V_j^k + \frac{1}{n^2} \sum_{k=1}^{n} V_i^k \sum_{k=1}^{n} V_j^k, \tag{3}$$

It follows that if we can compute and store the sums of V_i^k, V_j^k, $V_i^k V_j^k$, and n, we can compute the covariance with these sums and n for any intermediate node. In order to compute the covariance matrix of a parent node, we simply add these variables as a weighted sum of the number of vertices contained in each child node. This property is particularly useful to compute the R-LODs of inner nodes in the kd-tree in an out-of-core manner. Our algorithm has linear time complexity and its memory overhead is a function of the height of the tree. In practice, the memory overhead in our benchmarks is less than 1MB.

4.5 C^0 Discontinuity between R-LODs

Our LOD computation algorithm computes a drastic simplification. Therefore, if the underlying triangles have high curvature, the PCA-based approximation can have high surface deviation. In this case, it is possible that our algorithm does not maintain C^0 continuity between R-LODs, which can result in some holes in the resulting image (see Fig. 7). This kind of problem has been well-studied in the LOD and point-based rendering literature. Particularly, many techniques in the LOD literature have been proposed to patch these holes using precomputed data structures or runtime algorithms [6, 42]. However, those approaches can increase the storage and runtime overhead of ray tracing algorithms. In our implementation, we do not use any patching techniques.

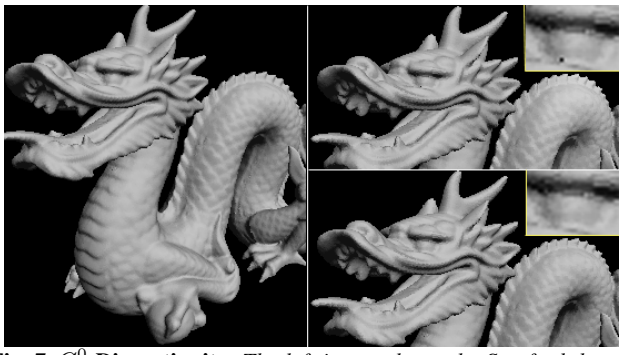


Fig. 7 C^0 **Discontinuity**: *The left image shows the Stanford dragon model as rendered by our approach with PoE* $= 0$*, i.e. using original triangles. The top right image was acquired by setting PoE* $= 5$ *at* 512×512 *image resolution with no expansion of R-LODs. As can be seen in the area of the dragon's eye, there is a hole caused by* C^0 *discontinuity of our LOD representation. By allowing a small amount of expansion of R-LODs, we can avoid having holes in the final image as shown in the bottom right image. Close-ups of the eye are shown in boxes with yellow borders.*

Instead, we ameliorate this problem through our R-LOD selection algorithm. A very low PoE bound should be used to limit the error introduced by the R-LODs. The low PoE bound also minimizes temporal popping that can arise when we switch between the R-LODs of parent and children nodes during successive frames. Moreover, we assign higher weight to surface deviation computation as part of the error metric computation; therefore, higher resolutions are used in the region with high curvature.

Expansion of R-LODs: In addition to these two heuristics, we also expand the extents of R-LODs to remove holes caused by C^0 discontinuity between R-LODs. Please note that as the surface deviation increases, it is likely that gaps become larger. Therefore, we increase the extent of a R-LOD as a function of the surface deviation associated with the R-LOD. This expansion is efficiently considered during the plane and ray intersection as an additional numerical tolerance. In practice, we found that combining these heuristics work well to remove holes caused by C^0 discontinuity without introducing any noticeable visual artifact given low PoE error bounds (see Fig. 7).

5 Utilizing Coherences

In this section, we describe approaches to improve the performance of our ray tracing algorithm using ray coherence and cache-coherent layouts.

5.1 Ray Coherence

We define ray coherence as the coherence of rays in tree traversal and intersection, i.e. rays may take a similar path in the tree and may hit the same triangles. For primary rays, our ray tracer starts out by assuming there is ray coherence

[1] The direction of the normal is chosen to be closer to the average normal of triangles.

Model	Vert. (M)	Tri. (M)	Node (M)	Size (GB)	R-LOD Comp. (min)
Forest	19	32	105	4.1	10
Double eagle	77.7	81.7	173	9.1	32
St. Matthew	128	256	378	26	124

Table 1 Benchmark models

Model complexity, the number of kd-nodes, the total size of kd-tree, geometry, and R-LODs, and the construction time of R-LODs are shown.

and shoots a beam using the algorithm presented in [22]. We compute a common entry point in the tree for all rays in the beam, at which the beam is split into either sub-beams or ray packets depending on its size. For the latter case, we use the coherent ray tracing algorithm [31] which works on a 2×2 packet of rays in parallel using current processors' SIMD instructions. During all traversal, we check whether we need to use R-LODs that have appropriate resolution based on our LOD metric. If so, we stop traversal of that subtree and intersect with the simplified representation. If the given model is highly tessellated, beam tracing and the use of SIMD instructions may not work well and can even lead to a decrease in performance (as explained in Section 3.2). However, we found that the use of R-LODs alleviates this problem, as we generally do not traverse as deep into the tree and therefore execute less overhead intersections. Secondary rays can be also handled in a similar manner.

5.2 Cache Coherence

Recently, there has been efforts to compute cache-coherent layouts of meshes [40, 39]. Since a kd-tree is more frequently accessed than its input mesh at runtime, it is more highly desirable to maintain cache coherence during run-time tree traversals to help to achieve good performance. To achieve this goal, it is critical to predict the runtime access behavior and closely store kd-nodes that are likely to be accessed in the succession. In order to predict the runtime behavior on kd-nodes during tree traversal, we use a simple method to compute the probability that a node will be accessed given that its parent node has been accessed before, based on their geometric relationship [41]. The ray tracing algorithm traverses the child nodes from the parent node when there is an intersection between a ray and the bounding box of the parent node. Therefore, we estimate that the probability that the child node is accessed increases as its surface area compared to its parent node increases. This property is already well exploited by the kd-tree construction algorithms by using the surface areas of the bounding boxes of the kd-nodes [17]. The layouts can increase the performance of the ray tracer by $10 - 60\%$ on massive models. This is in addition to the speedups obtained by R-LODs.

6 Implementation and Results

In this section, we describe our implementation and highlight the performance of our ray tracer on different benchmarks.

6.1 Implementation

We have implemented our R-LOD construction algorithm and ray tracer on both 32-bit and 64-bit machines that have two dual-core Xeon processors running 32-bit and 64-bit Windows XP, respectively. For runtime ray tracing, we use memory mapped files to efficiently access large files of geometry and kd-tree. However, in the 32-bit OS, we can only map up to 3GB total memory. To deal with larger data, we have implemented explicit out-of-core memory access management. This is not necessary in the 64-bit OS where we just use implicit OS memory mapping functionality.

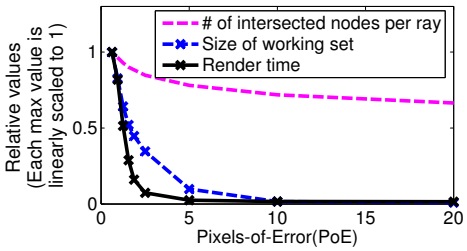


Fig. 9 Performance variation as a function of PoE: *We show the relative benefit of R-LODs on different aspects of overall performance of ray tracing St. Matthew model. We measured the rendering time, average number of processed node per ray, and size of working set on a 32-bit Xeon machine with 2GB RAM. All these values are shown in a scale-invariant manner by linearly scaling their maximum values to 1. The performance of our LOD-based ray tracer drastically decreases as we linearly increase the PoE. Moreover, the graph indicates that there is high correlation between the performance of the ray tracer and the size of working set. Image shots generated by tested PoE values can be seen in Fig. 8.*

In order to construct the kd-tree for a model that does not fit into main memory, we first subdivide the model into voxels in an out-of-core manner and then build the kd-tree for each of these voxels individually in core [42]. This step can also be performed in parallel on different voxels for speeding up the construction. Afterwards, the kd-tree for each voxel is merged into the global tree, which is used for ray tracing.

Since we have found that the quality of the kd-tree is the most important factor for fast ray tracing, we build the kd-tree using the surface-area heuristic [17, 10] and some further improvements as presented by [22]. Especially important is to introduce extra splits for empty areas in order to bound the geometry more tightly for our R-LOD representation.

6.2 Results

Benchmarks: We have applied our LOD-based ray tracing algorithm to different benchmarks as shown in Table 1. We computed different paths through these models and measured the performance of the ray tracer with and without LODs using a small PoE metric. We use a resolution of 512×512 pixels for interactive rendering. We also use 2×2 super-sampling per pixel; therefore, we effectively shoot $1K \times 1K$ rays from the eye for each frame. We are able to render most of these models at $5 - 12$ frames a second with primary rays and $1 - 8$ frames a second when we include reflections and shadow rays. These results are shown in the video.

Preprocessing: We only compute R-LODs for a subset of the nodes in the kd-tree to avoid excessive memory overhead. Our current implementation selects every third node on the path from the root node to the leaf node. Our unoptimized R-LOD construction implementation can process 2–3 million triangles per minute; most of the processing time is spent on reading data from the disk. The size of R-LODs associated with each node takes less than 10% of the total storage. However, if we consider the additional 4 bytes for R-LOD index and quantized error bound in the kd-nodes, total storage overhead of our R-LOD nodes is roughly 33% compared to the optimized kd-tree representation[30].

Performance variation as a function of PoE: We vary the PoE metric for the St. Matthew model (256M triangles) and measure its benefit on the rendering time, average num-

(a) No LOD (b) PoE $= 2.5$ (c) PoE $= 5$

Fig. 8 *Images of the St. Matthew model with different PoE values are shown at* 512×512 *image resolutions. We do not use anti-aliasing to highlight image quality difference. Please note that the original St. Matthew model has many holes. The use of R-LODs can alleviate aliasing artifacts and improve the performance of massive models.*

ber of processed nodes per ray, and size of working set per frame. The working set is measured at a granularity of 4KB. In order to show the relative benefit, we linearly scale each value into $[0, 1]$ by scaling the maximum value of each item to 1. The min and max values of each item are as follows: rendering time (ms)$(160, 11914)$, size of the working set(MB) $(2, 1565)$, and average number of processed nodes per ray $(13.6, 22.42)$. As can be seen in Fig. 9, the performance of the ray tracer increases drastically as we linearly increase the PoE values.

Runtime performance: The benefit of LODs varies with the reduction in the working set size. For a highly tessellated St. Matthew model with 128M triangles, we achieve more than one order of magnitude reduction in the size of the working set and almost two orders of magnitude improvement in the frame rate. This model has low depth complexity and more than half the primitives are visible from the eye. We show the frame rates obtained during rendering of the St. Matthew model with and without R-LODs and cache-oblivious layouts in Fig. 10. Moreover, we are able to achieve 2.6 frames per second while rendering the model with shadow and reflection with little loss of image quality (see. Fig. 1). For the forest model shown in Fig. 4, we are able to achieve more than five times improvement by using R-LODs.

In the case of the Double Eagle tanker, we get 10%–200% improvement. This model has high depth complexity and is not highly tessellated. As a result, the performance improvement due to LODs is limited. An image of the tanker with shadows is shown in Fig. 2.

7 Analysis and Comparison

In this section, we analyze the performance of our ray tracing algorithm and also compare its performance to prior approaches. We also discuss some limitations of our approach.

7.1 Analysis

We first examine different aspects of our R-LOD representation.

R-LOD overhead: Our algorithm introduces 4 bytes of additional storage for each kd-node. We also measure the additional computational overhead of evaluating our LOD metric during traversal by comparing the runtime performance on the Stanford scanned dragon model (870K triangles) of the standard ray tracer using 8 byte sized kd-nodes and of our ray tracer, which uses 12 byte-sized nodes with stored R-LODs. In order to measure the overhead of R-LODs, we set our PoE metric to 0 during LOD tree traversal; consequently, the image quality is the same in both cases. We found that the R-LOD overhead for storage and traversal reduces the performance by 2%–5%, as compared to ray tracing as described in [30].

Performance gains: The use of R-LODs reduces both computational workload and memory requirements. A major benefit of R-LODs is the reduction of the working set size and cache miss ratios of the runtime algorithm. This size decreases almost as a exponential function of the PoE as shown in Fig. 9. As a result, we get fewer L1/L2 cache misses and page faults and our new ray tracing algorithm is more cache coherent.

7.2 Comparison to other approaches

Our algorithm integrates R-LODs with the kd-tree representation for ray tracing. The idea of using an integrated hierarchical representation for traversal, visibility and simplification has been used by other algorithms for interactive rendering. These include the QSplat system [23], which uses a hierarchy of spheres and a screen space PoE metric to stop the tree traversal at a node. However, QSplat is mainly designed for point datasets or dense meshes arising from scanned models. Moreover, our LOD computation and error metric evaluation algorithms are different from QSplat as we take into account primary and secondary rays. The Quick-VDR system [42] uses a two-level multiresolution hierarchy called CHPM for view-dependent simplification and visibility culling of large polygonal models. However, the CHPM representation has a high memory overhead and does not lend itself well to ray tracing.

Several other ray tracing algorithms based on LODs have been proposed. The algorithm that is closest to our approach is the out-of-core ray tracer described in [32]. While we use R-LODs to perform fewer node and triangle intersections, Wald *et al.* use a simplified version only when the data is not in main memory in order to hide the latency incurred by loading data from the disk. This approach works well when the working set is smaller than main memory. Our LOD based algorithm is complimentary to their work and uses a different representation to reduce the size of the working set and perform fewer ray intersections.

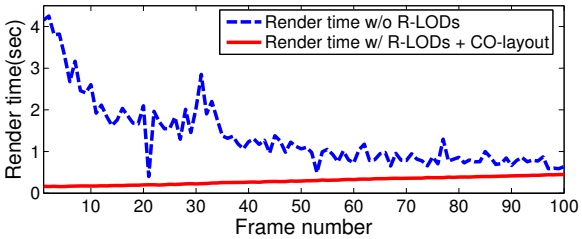


Fig. 10 Frame time with and without R-LODs: *The graphs shows frame times while rendering the* 128*M St. Matthew model with/without R-LODs and cache-oblivious (CO) layout. We measure frame time when we approach the model starting from the viewpoint shown in Fig. 8. The path is also shown in the video.*

Pharr *et al.* [20] describe an algorithm to optimize memory coherence in ray tracing. In their approach, the rays are reordered so that they access the scene data in a coherent manner. Their prime application was accelerating ray tracing for offline rendering. Our LOD based approach is quite complimentary to their algorithm. The LOD-based renderer described by Christensen *et al.*[5] differs from ours in two respects. Firstly, it uses subdivision meshes. Therefore, it is primarily useful for computing appropriate tessellation levels from the coarsest resolution. On the other hand, we compute the R-LODs from the original mesh. Secondly, Christensen *et al.* use ray differentials, which is expensive for real-time ray tracing. In contrast, our LOD metric is very efficient and optimized for interactive rendering.

7.3 Limitations

Our approach has certain limitations. First of all, any LOD-based acceleration technique can result in visual artifacts. We minimize these artifacts by using a low PoE bound and combining the projected screen-space error and surface deviation error of an R-LOD. If we use a high PoE bound, the R-LODs may result in holes on the simplified representation. This visual artifact can be removed by employing implicit surfaces[33, 14] as a LOD and thereby sacrificing some of the efficiency of our LOD representation. Moreover, our current R-LOD representation is a drastic simplification of the underlying geometric primitives and their material properties. As a result, the R-LOD representation may not provide high quality simplification for surfaces that have highly varying BRDF. One possibility is to use a more complex reflectance representation [18] in such cases. Also, our LOD metric does not give guarantees on the errors in the path traced by the secondary rays and the illumination computed at each pixel. However, we indirectly reduce the differences by reducing errors associated with the R-LODs. Finally, our efficient projection-based LOD error metric can currently handle planar reflections and shadow rays, but not refraction nor non-planar reflection.

8 Conclusion and Future Work

We have presented a novel LOD-based ray tracing algorithm to improve the performance of ray tracing massive models. We use the R-LOD representation as a drastic simplification of geometric primitives contained in the subtree of a kd-node and select the LODs based on our projection-based LOD error metric. We have described a hierarchical R-LOD construction algorithm that has linear time complexity and is well suited for out-of-core computation. The use of R-LODs results in fewer intersection tests and can significantly improve the memory coherence of the ray tracing algorithm. We have observed more than an order of magnitude speedup on massive models, and most of these gains are due to improved memory coherence and fewer cache misses.

There are many avenues for future work. In addition to addressing current limitations of our approaches, we would like to extend our current R-LOD representation to support smooth implicit surfaces to improve the rendering quality, and still have a compact representation. Moreover, we would like to extend our approach to handle other kinds of input model types such as point clouds [24] and higher order primitives. It might be useful to integrate approximate visibility criteria within our efficient LOD metric to further improve the performance ray tracing on massive models with high depth complexity. Also, we would like to consider visibility issues during construction of R-LODs in order to have better visual quality. Furthermore, we are interested in evaluating our ray tracer on other complex datasets and measuring the performance benefit. LODs could also be potentially useful in the context of designing future hardware for interactive ray tracing.

Acknowledgments

We would like to thank Eric Haines, David Kasik, Matt Pharr, Ingo Wald, and anonymous reviewers for providing helpful feedback on the draft of this paper. Also, we would like to thank Dawoon Jung and Peter Lindstrom for their support. Simplified versions of St. Matthew model are made by the streaming simplification tool of Martin Isenburg and Peter Lindstrom. The St. Matthew and Dragon model is courtesy of Stanford University. The double eagle tanker is courtesy of Newport News Shipbuilding. This work was supported in part by ARO Contracts DAAD19-02-1-0390 and W911NF-04-1-0088, NSF awards 0400134 and 0118743, ONR Contract N00014-01-1-0496, DARPA/RDECOM Contract N61339-04-C-0043, Intel, and, LOCAL LLNL LDRD project (05-ERD-018). Some of the work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under Contract No. W-7405-Eng-48.

References

1. Agrawala, M., Ramamoorthi, R., Moll, A.: Efficient image-based methods for rendering soft shadows. In: ACM SIGGRAPH, pp. 375–384 (2000)
2. Amanatides, J.: Ray tracing with cones. In: Computer Graphics (SIGGRAPH '84 Proceedings), vol. 18, pp. 129–135 (1984)
3. Appel, A.: Some techniques for shading machine renderings of solids. In: AFIPS 1968 Spring Joint Computer Conf., vol. 32, pp. 37–45 (1968)
4. Chiang, Y.J., El-Sana, J., Lindstrom, P., Pajarola, R., Silva, C.T.: Out-of-core algorithms for scientific visualization and computer graphics. IEEE Visualization 2003 Course Notes (2003)
5. Christensen, P.H., Laur, D.M., Fong, J., Wooten, W.L., Batali, D.: Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. Computer Graphics Forum **22**(3), 543–552 (2003)
6. Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., Scopigno, R.: Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. ACM Trans. Graph. **23**(3), 796–803 (2004)
7. DeMarle, D.E., Gribble, C.P., Parker, S.G.: Memory-savvy distributed interactive ray tracing. In: EGPGV, pp. 93–100 (2004)
8. Dietrich, A., Wald, I., Slusallek, P.: Large-Scale CAD Model Visualization on a Scalable Shared-Memory Architecture. In: Proceedings of 10th International Fall Workshop - Vision, Modeling, and Visualization (VMV) 2005, pp. 303–310 (2005)
9. Gobbetti, E., Marton, F.: Far voxels: A multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. ACM Trans. Graph. **24**(3), 878–885 (2005)
10. Havran, V.: Heuristic ray shooting algorithms. PhD thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague (2000)

11. Heckbert, P.S., Hanrahan, P.: Beam tracing polygonal objects. In: SIGGRAPH '84, pp. 119–127 (1984)
12. Igehy, H.: Tracing ray differentials. In: ACM SIGGRAPH, pp. 179–186 (1999)
13. Jolliffe, I.: Principle component analysis. In: Springer-Veriag (1986)
14. Levin, D.: Mesh-independent surface interpolation. In: Geometric Modeling for Scientific Visualization, pp. 37–49 (2003)
15. Lischinski, D., Rappoport, A.: Image-based rendering for non-diffuse synthetic scenes. In: Eurographics Rendering Workshop 98, pp. 301–314 (1998)
16. Luebke, D., Reddy, M., Cohen, J., Varshney, A., Watson, B., Huebner, R.: Level of Detail for 3D Graphics. Morgan-Kaufmann (2002)
17. MacDonald, J.D., Booth, K.S.: Heuristics for ray tracing using space subdivision. Visual Computer (1990)
18. Neyret, F.: Modeling, animating, and rendering complex scenes using volumetric textures. IEEE Transactions on Visualization and Computer Graphics (1998)
19. Parker, S., Martin, W., Sloan, P., Shirley, P., Smits, B., Hansen, C.: Interactive ray tracing. Symposium on Interactive 3D Graphics (1999)
20. Pharr, M., Kolb, C., Gershbein, R., Hanrahan, P.: Rendering complex scenes with memory-coherent ray tracing. In: Proc. of ACM SIGGRAPH, pp. 101–108 (1997)
21. Purcell, T., Buck, I., Mark, W., Hanrahan, P.: Ray tracing on programmable graphics hardware. ACM Trans. on Graphics (Proc. of SIGGRAPH'02) **21**(3), 703–712 (2002)
22. Reshetov, A., Soupikov, A., Hurley, J.: Multi-level ray tracing algorithm. ACM Trans. Graph. **24**(3), 1176–1185 (2005)
23. Rusinkiewicz, S., Levoy, M.: Qsplat: A multiresolution point rendering system for large meshes. Proc. of ACM SIGGRAPH pp. 343–352 (2000)
24. Schaufler, G., Jensen, H.W.: Ray tracing point sampled geometry. In: Rendering Techniques, pp. 319–328 (2000)
25. Schmittler, J., Woop, S., Wagner, D., Paul, W.J., Slusallek, P.: Realtime ray tracing of dynamic scenes on an FPGA chip. In: Proc. on Graphics hardware, pp. 95–106 (2004)
26. Shinya, M., Takahashi, T., Naito, S.: Principles and applications of pencil tracing. In: SIGGRAPH, vol. 21, pp. 45–54 (1987)
27. Shirley, P., Slusallek, P., Mark, B., Stoll, G., Wald, I.: Introduction to real-time ray tracing. SIGGRAPH Course Notes (2005)
28. Stoll, G., Mark, W.R., Djeu, P., Wang, R., Elhassan, I.: Razor: An Architecture for Dynamic Multiresolution Ray Tracing. Tech. Rep. TR-06-21, Dept. of CS, Univ. of Texas (2006)
29. Sung, K., Shirley, P.: Ray tracing with the BSP tree. Graphics Gems III pp. 271–274 (1992)
30. Wald, I.: Realtime Ray Tracing and Interactive Global Illumination. Ph.D. thesis, Computer Graphics Group, Saarland University (2004)
31. Wald, I., Benthin, C., Wagner, M., Slusallek, P.: Interactive rendering with coherent ray tracing. In: Computer Graphics Forum (EUROGRAPHICS), vol. 20, pp. 153–164 (2001)
32. Wald, I., Dietrich, A., Slusallek, P.: An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In: Proc. of the Eurographics Symp. on Rendering (2004)
33. Wald, I., Seidel, H.P.: Interactive Ray Tracing of Point Based Models. In: Proc. of 2005 Symp. on Point Based Graphics (2005)
34. Wald, I., Slusallek, P., Benthin, C.: Interactive distributed ray tracing of highly complex models. In: Proc. of EUROGRAPHICS Workshop on Rendering, pp. 277–288 (2001)
35. Wand, M., Straßer, W.: Multi-resolution point-sampled raytracing. In: Graphics Interface (2003)
36. Whitted, T.: An improved illumination model for shaded display. Commun. ACM **23**(6), 343–349 (1980)
37. Wiley, C., A.T. Campbell, I., Szygenda, S., Fussell, D., Hudson, F.: Multiresolution BSP trees applied to terrain, transparency, and general objects. In: Graphics Interface, pp. 88–96 (1997)
38. Woop, S., Schmittler, J., Slusallek, P.: RPU: a programmable ray processing unit for realtime ray tracing. ACM Trans. Graph. **24**(3), 434–444 (2005)
39. Yoon, S.E., Lindstrom, P.: Mesh Layouts for Block-Based Caches. Tech. Rep. UCRL-TR-220368-DRAFT, Lawrence Livermore National Lab. (2006)
40. Yoon, S.E., Lindstrom, P., Pascucci, V., Manocha, D.: Cache-Oblivious Mesh Layouts. Proc. of ACM SIGGRAPH (2005)
41. Yoon, S.E., Manocha, D.: Cache-Efficient Layouts of Bounding Volume Hierarchies. In: Computer Graphics Forum (Eurographics) (2006). To appear
42. Yoon, S.E., Salomon, B., Gayle, R., Manocha, D.: Quick-VDR: Interactive View-dependent Rendering of Massive Models. IEEE Visualization pp. 131–138 (2004)

Sung-Eui Yoon is currently a post-doctoral scholar at Lawrence Livermore National Laboratory. He received the B.S. and M.S. degrees in computer science from Seoul National University in 1999 and 2001 respectively. He received his Ph.D. degree in computer science from the University of North Carolina at Chapel Hill in 2005. His research interests include visualization, interactive rendering, geometric problems, and cache-coherent algorithms and layouts.

Christian Lauterbach is currently a Ph.D. student at the University of North Carolina at Chapel Hill. He received the Diplom in computer science from the University of Bremen, Germany in 2005. His research interests include interactive ray tracing, global illumination algorithms and massive model rendering.

Dinesh Manocha is currently the Phi Delta Theta/Matthew Mason Distinguished professor of computer science at the University of North Carolina at Chapel Hill. He received his B.Tech. degree in computer science and engineering from the Indian Institute of Technology, Delhi in 1987; and his M.S. and Ph.D. degrees in computer science from the University of California at Berkeley in 1990 and 1992, respectively. He has published more than 200 papers in the leading conferences and journals in computer graphics, geometric modeling, computational geometry, robotics and symbolic computation. He was selected an Alfred P. Sloan Research Fellow, received NSF Career Award in 1995 and Office of Naval Research Young Investigator Award in 1996, Honda Research Initiation Award in 1997, and Hettleman Prize for scholarly achievement at UNC Chapel Hill in 1998. He has also received 8 best paper and panel awards at the ACM SuperComputing, ACM Multimedia, IEEE VR, ACM Solid Modeling, IEEE Visualization, Pacific Graphics and Eurographics Conferences.