Cache-Oblivious Ray Reordering

Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio
KAIST
Hye-sun Kim, Yun-ji Ban, Seung Woo Nam
Electronics and Telecommunications Research Institute (ETRI)
and
Sung-eui Yoon
KAIST

We present a cache-oblivious ray reordering method for ray tracing. Many global illumination methods such as path tracing and photon mapping use ray tracing and generate lots of rays to simulate various realistic visual effects. However, these rays tend to be very incoherent and show lower cache utilizations during ray tracing of models. In order to address this problem and improve the ray coherence, we propose a novel *hit point heuristic* (HPH) to compute a coherent ordering of rays. The HPH uses the hit points between rays and the scene as a ray reordering measure. We reorder rays by using a space-filling curve based on their hit points. Since a hit point of a ray is available only after performing the ray intersection test with the scene, we compute an approximate hit point for the ray by performing an intersection test between the ray and simplified representations of the original models. Our method is a highly modular approach, since our reordering method is decoupled from other components of common ray tracing systems. We apply our method to photon mapping and path tracing and achieve more than an order of magnitude performance improvement for massive models that cannot fit into main memory, compared to rendering without reordering rays. Also, our method shows a performance improvement even for ray tracing small models that can fit into main memory. This performance improvement for small and massive models is caused by reducing cache misses occurring between different memory levels including the L1/L2 caches, main memory, and disk. This result demonstrates the cache-oblivious nature of our method, which works for various kinds of cache parameters. Because of the cache-obliviousness and the high modularity, our method can be widely applied to many existing ray tracing systems and show performance improvements with various models and machines that have different cache parameters.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Raytracing*

General Terms: Performnace, Algorithms

Additional Key Words and Phrases: ray coherence, reordering, cache utilization, ray tracing

1. INTRODUCTION

Ray tracing has been widely used as the main rendering engine of various global illumination methods (e.g., path tracing and photon mapping). Typically, ray tracing generates lots of primary, secondary, and shadow rays, in order to simulate realistic rendering effects (e.g., soft shadows, reflections, caustics, motion blur, etc.). However, ray tracing has been still known to be slow to provide these realistic visual effects.

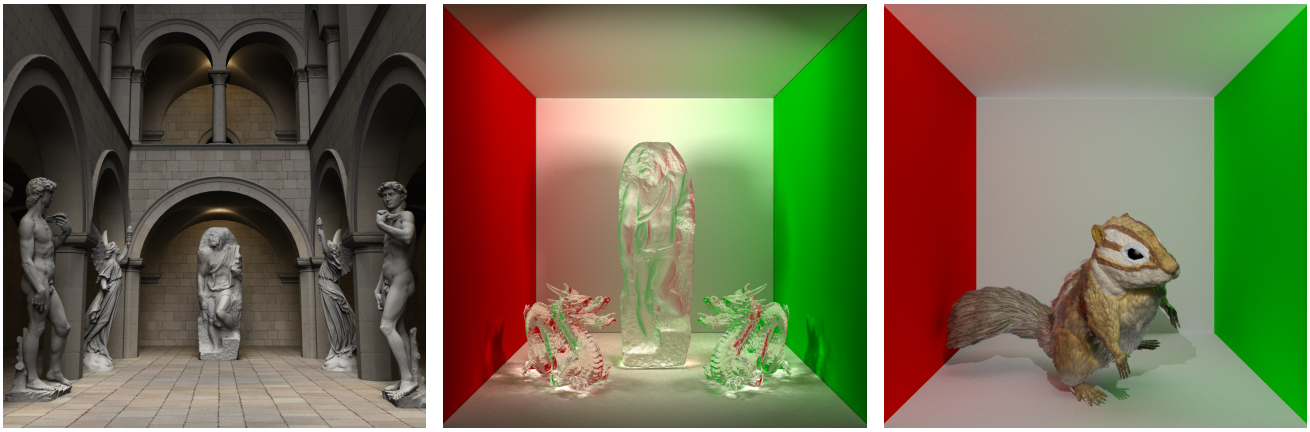
In order to improve the performance of ray tracing, a lot of studies have been done on designing efficient intersection tests, constructing efficient acceleration hierarchies, and exploiting data-level parallelism using the SIMD functionality and GPUs [Shirley and Morley 2003; Pharr and Humphreys 2004; Wald et al. 2007]. Most research has focused on improving the performance of ray tracing with primary rays. However, the focus has been recently shifted towards efficiently handling secondary rays that can provide realistic visual effects.

It has been widely known that secondary rays generated for simulating realistic visual effects show a low ray coherence and thus low cache utilizations during processing of these rays with meshes and their acceleration hierarchies. One of the main challenges to the efficient handling of secondary rays, therefore, is to achieve a high ray coherence and cache utilizations during processing of rays. This problem of achieving high cache utilizations is becoming more important, since there is the widening gap between the data access speed and the data processing speed [Hennessy et al. 2007].

In order to achieve a high cache coherence for ray tracing, two orthogonal and complementary approaches, *layout reordering* and *ray reordering*, have been studied. Layout reordering methods [Sagan 1994; Yoon et al. 2008] aim to compute cache-coherent layouts of meshes and hierarchies such that data elements (e.g., vertices, triangles, and nodes) that are close in meshes and hierarchies are also closely stored in their one dimensional data layouts in main memory and external drive.

Although meshes and hierarchies are stored coherently in their layouts, the data access pattern on these layouts should be coherent as well, in order to design cache coherent ray tracing. A few ray reordering techniques [Pharr et al. 1997; Navratil et al. 2007; Budge et al. 2009] have been proposed. The seminal ray reordering method proposed by Pharr et al. [1997] does not process each ray as it is generated. Instead, the method queues rays into ray buffers associated with regions of the mesh and processes these regions in a coherent manner to reduce the number of expensive disk I/O accesses. Most other ray reordering methods are based on variations of this ray reordering framework. The original method proposed by Pharr et al. uses a scheduling grid and sorts rays into each grid cell during the scene traversal. Other techniques have extended this method to use an acceleration hierarchy and sort rays into nodes of the hierarchy during the hierarchy traversal, while considering available cache information.

These methods essentially exploit the information about whether a data is cached or not given a cache and sort rays depending on the data access pattern during the scene or hierarchy traversal. Although this kind of approaches can achieve high cache utilizations during ray tracing of models, it complicates the ray tracing system by coupling the traversal and the ray reordering algorithm. Furthermore, all of these prior methods focused only on either reducing L1/L2 caches for small models or reducing the disk I/O accesses



1: *The left image shows the result of our method applied to path tracing of a Sponza model with a St. Matthew model, two Lucy, and two David models. This Sponza scene consists of 104 million triangles, requiring 12.8 GB for the original meshes and their acceleration hierarchies. The middle and right images show photon mapping results of a transparent St. Matthew model consisting of 128 M triangles in the Cornell box with two transparent dragon models, and a furry squirrel modeled with 32 million hair strands in the Cornell box. The St. Matthew and squirrel scenes take 15.7 GB and 8.2 GB respectively. These two global illumination methods generate many incoherent rays to render these images. By reordering such rays, we achieve more than one order of magnitude performance improvement in a machine with 4 GB main memory, compared to without reordering rays. This performance improvement is caused by the improved ray coherence.*

for massive models that cannot fit into main memory, because of the cache-aware nature of these methods.

Main contributions: In this paper, we present a cache-oblivious ray reordering method to achieve high cache utilizations during ray tracing of models for global illumination methods. Our approach decouples the ray reordering method from the hierarchy traversal to achieve high modularity. In order to reorder rays, we propose a novel *hit point heuristic*, which uses hit points between rays and the scene as a ray reordering measure (Sec. 4). Since the hit point of a ray is only available once the ray is processed by traversing the hierarchy, we approximate the hit point by using a simplified model of the original model. We then use a space-filling curve to reorder rays based on their approximate hit points. This enables our method to work with different cache parameters and to achieve high cache utilizations for various memory levels. We apply our ray reordering method to path tracing and photon mapping (Sec. 5). By reordering rays, we achieve more than an order of magnitude performance improvement compared to rendering without reordering rays for massive models that cannot fit into main memory. Moreover, our method shows a performance improvement for small models that fit into main memory. We verify that the reduced L1/L2 cache misses result in the performance improvement, by simulating L1/L2 caches and measuring L1/L2 cache misses. These results demonstrate the benefits of the cache-oblivious nature of our ray reordering method. We also discuss various factors affecting the performance of our algorithm and compare our method with prior methods (Sec. 6). We conclude in Sec. 7 with future work.

2. RELATED WORK

Ray tracing and global illumination methods have been well studied. Also, good surveys and books are available [Shirley and Morley 2003; Pharr and Humphreys 2004; Wald et al. 2007]. In this section, we review prior work related directly to our problem.

2.1 Computation Reordering

Computation reordering strives to achieve a cache-coherent order of runtime operations in order to improve program locality and reduce the number of cache misses. Computation reordering methods can be classified into either *cache-aware* or *cache-oblivious*. Cache-aware algorithms utilize the knowledge of cache parameters, such as cache block size [Vitter 2001]. On the other hand, cache-oblivious algorithms do not assume any knowledge of cache parameters [Frigo et al. 1999]. There is a considerable amount of literature on developing cache-efficient computation reordering algorithms for specific problems and applications [Arge et al. 2005; Vitter 2001]. In computer graphics, out-of-core algorithms [Silva et al. 2002], which are cache-aware methods, have been designed to handle massive models.

2.2 Cache-Coherent Ray Tracing

There has been extensive research on exploiting the coherence in ray tracing. These can be classified into packet methods, layout reordering, and ray reordering methods.

Packet ray tracing: Neighboring rays can exhibit spatial coherence and utilizing this coherence can improve the performance of ray tracing. Earlier attempts include beam tracing [Heckbert and Hanrahan 1984]. Wald et al. [2001] exploited the coherence of primary and shadow rays by grouping rays into packets and utilizing the SIMD functionality of modern processors. Reshetov *et al.* [2005] proposed an algorithm to integrate beam tracing with the kd-tree spatial structure and were able to further exploit coherence of primary and shadow rays. There have been a few ray reordering methods that can utilize the SIMD functionality for secondary rays [Boulos et al. 2008; Gribble and Ramani 2008]. These ray reordering methods for the SIMD utilization can be performed on rays reordered by our method.

Layout reordering: The order of data stored in memory or external drives can affect the performance of ray tracing, given the widely used block-fetching caching scheme [Yoon et al. 2008]. In

this caching scheme, blocking related nodes in a cluster can reduce the number of cache misses. The *van Emde Boas* layouts of trees [van Emde Boas 1977] are constructed by performing a recursive blocking to nodes. Havran analyzes various layouts of hierarchies in the context of ray tracing and improves the performance by using a compact layout representation of hierarchies [Havran 1997]. Yoon and Manocha [2006] developed cache-efficient layouts of hierarchies for ray tracing. Also, there are a few cache-coherent mesh layouts [Yoon et al. 2005; Yoon and Lindstrom 2006; Sagan 1994].

Ray reordering: To reorder primary rays, space-filling curves like Z-curves [Sagan 1994] have been used. Mansson et al. [2007] showed coherence among secondary rays based on their proposed ray coherence measures. However, it was not demonstrated to achieve a higher runtime performance based on their proposed ray reordering heuristics. Pharr et al. [1997] proposed a ray reordering method for ray tracing massive models that cannot fit into main memory. Their method uses a scheduling grid for queueing rays and processes rays in a coherent manner, while considering the available cache information. Steinhurst et al. [2005] reorder kNN searches of photon mapping to reduce the memory bandwidth. Navratil et al. [2007] presented a ray scheduling approach that improves a cache utilization and reduces DRAM-to-cache bandwidth usage. Budge et al. [2009] employed a ray reordering method to utilize hybrid resources such as multiple CPUs and GPUs. These techniques are based on Pharr et al.'s ray reordering method, which couples the ray reordering and the scene traversal. By doing so, these methods can easily know which parts of meshes and hierarchies are accessed and cached during processing of rays. A downside of these techniques is that by coupling the ray reordering and scene traversal, the modularity of these methods is lowered.

2.3 Ray Tracing Massive Models

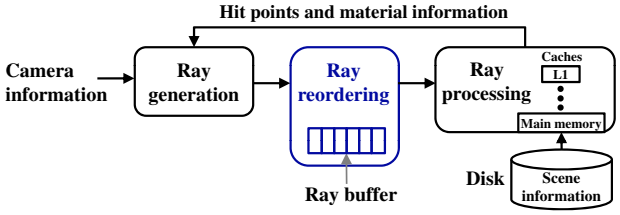
Ray tracing massive models has been studied well. In-core techniques exist to perform the ray tracing of massive datasets [DeMarle et al. 2004; Stephens et al. 2006] by using large, shared memory systems. There are also out-of-core techniques including latency hiding [Wald et al. 2004]. There are different approaches aiming at designing compact representations, by applying the quantization on acceleration hierarchies [Cline et al. 2006], reducing costs of representing meshes and hierarchies [Lauterbach et al. 2008; Kim et al. 2010], or efficient culling techniques [Reshetov 2007]. These methods can be combined with our proposed method to further improve the performance of ray tracing massive models.

3. OVERVIEW

In this section, we discuss the ray coherence of different types of rays and briefly explain the overall approach of our method.

3.1 Ray Coherence

Ray tracing generates a lot of rays to simulate various visual effects. These rays can be classified as primary, shadow, and secondary rays. Primary rays are known to show a high coherence during the hierarchy traversal and mesh accesses. Space-filling curves such as Z-curves have been used to reorder primary rays [Pharr et al. 1997], based on positions of primary rays in the image plane. Once a primary ray has intersected with an object, shadow rays to lights and secondary rays (e.g., reflection rays), depending on the material property of the intersected object, are generated. Since light positions can be arbitrary and the intersected geometry can have an arbitrary normal, shadow and secondary rays generally have a lower coherence than primary rays. If rays are incoherent, then the data access pattern on the acceleration hierarchies and meshes can be incoherent. This incoherence may result in a high number of cache misses in various memory levels and lower the runtime performance. Therefore, processing rays in a cache-coherent manner is critical to design cache-coherent ray tracers.



2: *This figure shows different modules of our ray reordering framework. Our main contribution is the hit point heuristic (HPH) based ray reordering method employed in the ray reordering module.*

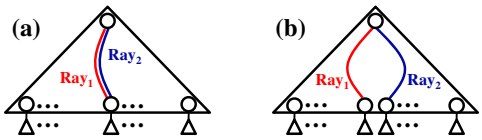
3.2 Ray Reordering Framework

In order to reorder rays, we use a ray reordering framework (see Fig. 2) extended from typical ray tracing systems. This framework consists of ray generation, ray reordering, and ray processing modules. The ray generation module constructs rays including primary, secondary, and shadow rays. The ray processing module takes each ray and finds a hit point between the ray and the scene by accessing acceleration hierarchies and the meshes of the scene. Also, the ray processing module performs shading based on the hit point and its corresponding material information. If we have to generate shadow and secondary rays, the ray processing module sends the hit points and material information to the ray generation module. Typical ray tracing systems consist of only these two modules and process rays as they are generated without reordering rays.

In addition to these modules, we also use the ray reordering module. The ray reordering module maintains a *ray buffer* that can hold a user defined number of rays. Once the ray generation module constructs rays, these rays are stored in the ray buffer and then reordered in a way such that meshes and hierarchies are accessed in a cache-coherent manner during processing of reordered rays in the ray processing module. Note that our ray reordering framework is similar to previous ray reordering methods [Pharr et al. 1997; Navratil et al. 2007; Budge et al. 2009]. The main difference of our framework over these prior methods is that we decouple the ray reordering module from other modules, thereby achieving high modularity.

Given this ray reordering framework, the key component that governs the performance improvement is the ray reordering method. To maximize the benefits of the reordering method, the overhead of reordering should be kept small. We propose a simple cache-oblivious reordering method that has a low reordering overhead, increases the cache coherence, and improves the performance of ray tracing models that have different model complexities.

Cache-coherent layouts of meshes and hierarchies: Our ray reordering method works on the assumption that geometrically close mesh data (e.g., vertices or triangles) and topologically close hierarchy data (e.g., nodes) are also stored closely in their corresponding mesh and hierarchy layouts respectively. There are many layouts satisfying such a property for meshes [Sagan 1994; Diaz-Gutierrez et al. 2005; Yoon and Lindstrom 2006] and for hierarchies [van Emde Boas 1977; Havran 1997; Yoon and Manocha 2006]. In our implementation, we use cache-oblivious layouts of meshes and hierarchies [Yoon and Lindstrom 2006; Yoon and Manocha 2006]



4: *These two figures show data access patterns on the hierarchy during processing of two different rays, whose hit points are close to each other. The difference between the left and right figures is that two rays' directions are similar in the left, but different in the right.*

4. CACHE-OBLIVIOUS RAY REORDERING

In this section we introduce our cache-oblivious ray reordering method.

4.1 Hit Point Heuristic

To reorder rays, we propose a *hit point heuristic* (HPH). A hit point of a ray is defined as the first intersection point computed between the ray and the scene, starting from the ray's origin. The main idea of the HPH method is to reorder rays based on their hit points using a space-filling curve (e.g., Z-curve).

The rationale why we use the hit point of a ray as a reordering measure is twofold. First, if the hit points of rays are geometrically close to each other, then the mesh regions accessed during processing of these rays are likely to be close too. Second, suppose that a hierarchy is decomposed into lower and upper regions. Lower regions of the hierarchy are closer to leaf nodes and upper regions of the hierarchy are closer to the root node of the hierarchy. Then, the lower regions of the hierarchy accessed during processing of rays whose hit points are close are likely to be close too because of the same reason that were for meshes (see Fig. 4). Although hit points of rays are close to each other, these rays' directions may be very different. In this case, their access patterns on upper regions of the hierarchy may be very different (see Fig. 4-(b)). However, the size of these upper regions of the hierarchy is relatively small compared to those of lower regions of the hierarchy. Also, the upper regions of the hierarchy are accessed by almost all the rays and thus are unlikely to be unloaded from the cache. Therefore, we may not get additional cache misses during processing of rays with the upper regions of the hierarchy.

To empirically verify the second rationale, we simulate a 6MB wide 24-way set-associative L2 cache of our test machine and measure L2 cache misses that occur in the upper and lower regions of a hierarchy during photon mapping of the Armadillo model in the Cornell box scene (Fig. 3). The number of L2 cache misses occurring in the lower regions of the hierarchy is significantly higher (e.g., 141.4 times higher than that occurring in the upper region of the hierarchy. As a result, we conclude that hit points between rays and the scene are equally or more important features to our problem than ray directions and ray origins, which have been widely considered as reordering measures in most prior works.

3: *Photon mapping of an Armadillo (346 K triangles and 43.5 MB) in the Cornell box.*

4.2 Approximate Hit Points

An issue of the HPH method is that it requires hit points between rays and the scene to reorder rays. However, computing these hit points requires processing of rays by traversing the hierarchy and accessing the mesh, which may cause a high number of cache misses that we attempted to avoid by reordering. To address this problem, we compute approximate hit points efficiently by performing the intersection tests between rays and simplified representations of the original models.

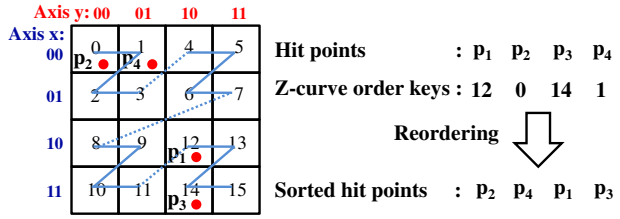
We compute a simplified representation of the original model using an out-of-core mesh simplification method [Yoon et al. 2008]. This simplification method decomposes an input model into a set of clusters, each of which can be stored in main memory. Then, we simplify each cluster one by one. In order to compute approximate hit points that are close to the exact hit points, the simplified model should be geometrically similar to the original model. We use quadrics and choose edge collapses in an increasing order of simplification errors for each cluster by using a heap [Garland and Heckbert 1997] within each cluster. While simplifying each cluster, we also allow simplifying edges that span multiple clusters. For a simplified representation, we set the bounding box of the simplified model to be the bounding box of the original model. Therefore, if a ray does not intersect with the bounding box, it is guaranteed that the ray does not intersect with the original model.

Although simplification techniques including ours that rely on quadrics and edge collapses have been known to work well for various polygonal models [Luebke et al. 2002], we found that it does not work well with models with lots of small objects including a furry squirrel model (shown in the right image of Fig. 1) in our benchmark models. Fortunately, we found that a recent stochastic simplification technique [Cook et al. 2007] works quite well for such models that have aggregate detail. We use this stochastic simplification method only for the furry squirrel model, given our out-of-core simplification framework described above.

For each simplified representation, we build a hierarchy in the same manner as building the hierarchy for the original model. In order to reduce the overhead of computing hit points with the simplified representations at runtime, we drastically simplify the models. In our tests, we use simplified models consisting of 2% of the complexity of the original models. We found that this strikes a good balance between the overhead of our method and the approximation quality and thus achieves the best performance improvement of using our ray reordering method (see Sec. 5.3).

To compute approximate hit points of rays, we perform intersection tests between the rays and the simplified models of the scene. If a ray intersects with one primitive of the simplified models, we use the hit point for the ray reordering. If the ray does not intersect with any primitives of simplified models, but one of the bounding boxes of the original models, we use the intersection point between the ray and the bounding boxes as a *virtual hit point* and use it for the ray reordering. For other rays that do not intersect with any of the bounding boxes, we terminate the processing of these rays, since it is guaranteed that they do not intersect with the original models of the scene. We use the computed approximate hit points only for reordering, not for other computations (e.g., shading).

One may consider to use virtual hit points as approximate hit points even for rays intersected with the scene, instead of using high quality simplified representations. However, we found that using only virtual hit points produces rather low-quality approximation results and using high quality simplified representations shows much higher (e.g., up to 12.1 times) performance improvements in our benchmark scenes.



5: *This figure shows an ordering of hit points with the Z-curve ordering of cells in the uniform grid.*

4.3 Space-Filling Curve based Reordering

Once we compute approximate hit points for rays stored in the ray buffer, we reorder these rays by using a Z-curve, a simple space-filling curve. Since a Z-curve is defined in a uniform structure, we place hit points in a grid and compute ordering keys for these hit points by using a Z-curve ordering of cells in the grid structure.

We define the grid to enclose the bounding volume of the scene and to have $2^k \times 2^k \times 2^k$ cells. Then, we quantize each of three coordinates of a hit point into a k-bit integer. It has been known that the z-curve ordering key of such a point is computed by simply interleaving bits of three k-bit integers of the quantized three coordinates of the point [Lauterbach et al. 2009]. For example, suppose that $x_k \cdots x_1$, $y_k \cdots y_1$, and $z_k \cdots z_1$ are three k-bit integers of the quantized three coordinates. Then, the z-curve ordering key of such point is defined by a 3k-bit integer of $x_k y_k z_k \cdots x_1 y_1 z_1$. An example of the Z-curve ordering keys of hit points is shown in Fig. 5.

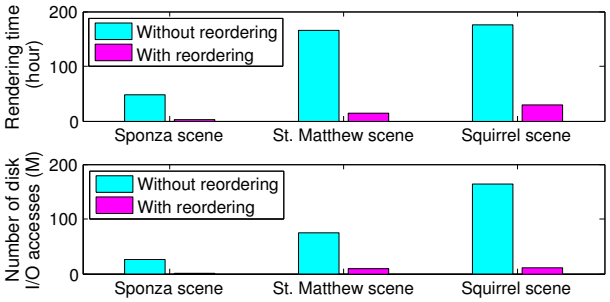
In our current implementation, we choose k to be 20. Therefore, the ordering key for each hit point is represented with 60 bits that can be stored in an 8-byte integer. Also, our grid structure decomposes the bounding volume of the scene into 2^{60} uniform-sized cells. Therefore, most final ordering keys computed from rays are likely to be unique with models that we can have in practice. We also tried Hilbert-curves [Sagan 1994], but found that Z-curves are easier to implement and have less computations, while having only minor performance degradation (e.g., 2%) over Hilbert-curves.

Once we compute the ordering keys for rays, we sort rays based on the ordering keys. We use the 2-way merge sort due to its simplicity. After sorting rays using their associated approximate hit points, sorted rays are processed in the ray processing module.

5. IMPLEMENTATIONS AND RESULTS

We have implemented and integrated our ray reordering module in a CPU-based out-of-core ray tracing system. Our ray tracing system uses bounding volume hierarchies (BVHs) with axis-aligned bounding volumes for models [Wald et al. 2007; Lauterbach et al. 2006]. Also, to design an out-of-core ray tracing system, we employ an out-of-core data access framework for meshes and hierarchies [Kim et al. 2010]. This framework maintains a memory pool that consists of pages, each of which holds 4 MB of data. The size of the memory pool is determined by the available main memory. We also employ a simple memory management method based on the least-recently used (LRU) replacement policy. To implement the LRU replacement policy, we maintain a LRU list containing pages that have been accessed during the mesh and hierarchy traversal for ray tracing.

We use 512 by 512 image resolutions and perform various tests with a 32 bit Windows machine consisting of a 3.0 GHz processor, a disk that supports a sequential reading performance of 101 MB per second, and 4 GB memory, unless mentioned otherwise. Although the machine has 4 GB main memory, all the programs in the 32 bit Windows can use only up to 3.25 GB. Also, the Windows OS in our test machine uses about 0.2 GB. Therefore, our ray tracer can use up to about 3.05 GB and we use this as the maximum size of the memory pool for our out-of-core data access framework.



6: *These figures show the overall rendering time and the number of the disk I/O accesses that occurred during rendering of the Sponza, the St. Matthew, and the squirrel scenes.*

Ray processing throughputs: Our out-of-core ray tracer does not have a high ray processing throughput that is comparable to those of the-state-of-the-art ray tracers. When we test our ray tracer generating only primary rays with small models (e.g., Stanford bunny) that fit into main memory, our single-threaded ray tracer can process 1 million rays per second. Also, when we test our ray tracer for path tracing the Sponza scene with enough main memory (e.g., 16 GB) that can hold all the data, our ray tracer can process 82.3 K rays per second; detailed rendering configurations will be given in Sec. 5.1. Our method uses out-of-core abstractions, which have high overheads. Also, our method does not use any packet tracing methods; if we implement recent packet tracing methods, we expect that our ray tracer can have higher ray processing throughputs.

Ray buffer: Our ray buffer consists of in-core and out-of-core parts. We allocate only 88 MB of the main memory space to an *in-core* ray buffer. Once the in-core buffer is full, we push these rays into an *out-of-core* ray buffer on the disk and then store the next rays in the in-core ray buffer. We do not pose any restriction on the size of the out-of-core ray buffer. If there are no more rays that we can generate, we sort the rays stored in the in-core and out-of-core ray buffers.

We test our method with two global illumination methods: path tracing and photon mapping, both of which generate many incoherent rays to produce realistic visual effects. We generate primary rays in Z-curves for all the tests.

5.1 Path Tracing

The left image of Fig. 1 shows an unbiased rendering image of the St. Matthew, two Lucy, and two David models in the Sponza scene using a path tracing method [Shirley and Morley 2003; Pharr and Humphreys 2004]. This scene consists of 104 M triangles; we do not use any instancing for duplicate models. BVHs and meshes of models in the scene take 12.8 GB. Since our ray tracer with the 32 bit test machine can use only 3.05 GB, main memory of the machine can cache 23.8% of all the data for our out-of-core ray tracer. To illuminate the scene, we use 8 area lights. We generate 100 primary rays (i.e., paths) per pixel and use simple importance sampling by generating shadow rays to the lights. In this configuration, we generate 361 M rays at each frame; 309 M and 26 M rays among all the generated rays are shadow and secondary rays respectively. We use the Russian roulette method to determine the path length.

In-core ray buffer size	22MB	44MB	88MB	176MB
Rendering time (sec.)	10,541	10,460	10,314	10,459
Overhead (sec.)	1,039	879	754	693

I: *This table shows the overall rendering time and the total overhead of our method as a function of the in-core ray buffer size in the Sponza benchmark with 4 GB main memory.*

In this scene, our method achieves a 16.83 times performance improvement over rendering without reordering rays. We also measure the number of the disk I/O accesses occurring during the access of meshes and BVHs (Fig. 6), by using the Windows built-in performance monitor tool, *perfmon*. By reordering rays, we reduce the number of the disk I/O accesses that occurred without reordering rays by 93.6%. We also measure the average disk I/O access performance (MB/sec.) per disk I/O access. We found that reordering rays improves the disk I/O access performance by 208.5%. This is because the disk I/O accesses become more coherent and the disk can process these I/O accesses with a higher reading performance during the random accesses on BVHs and meshes. Because of these two factors, the reduction of disk I/O accesses and the improvement of disk I/O performance, we achieve more than an order of magnitude performance improvement when caching only 23.8% of all the data in main memory.

5.2 Photon Mapping

The middle image of Fig. 1 shows a rendering of the transparent St. Matthew and two transparent dragon models in the Cornell box scene using the photon mapping method [Jensen 2005]. This St. Matthew scene consists of 128 M triangles and takes 15.7 GB for its meshes and BVHs; therefore, the machine can cache only 19% of the total model size. We use 4 area lights, generate 25 primary rays per pixel and 10 final gathering rays, and use 100 samples for the irradiance estimation; 26 M shadow and 91 M secondary rays are generated among all the generated 124 M rays at each frame. In this configuration, our method achieves a 12.28 times improvement compared to rendering without reordering rays. By reordering rays, we reduce 88% of the disk I/O accesses and improve the disk I/O performance by 141%.

We also test a furry squirrel model that has 32 M hair strands. Each hair strand is represented as 8 cylinders. This model, shown in the right image of Fig. 1, consists of 256 M cylinders and takes 8.2 GB for its cylinders and BVHs. This model has lots of small hairs and thus is considered as a difficult benchmark for computing a high-quality simplification. Moreover, since there are a lot of complex occlusion among furs, our approximation method for hit points using simplified representations may not work well in this squirrel scene. Also, we have to recursively generate many secondary rays until the accumulated opacity is higher than a threshold (e.g., 0.9), because of the semi-transparent property of furs. We use a single point light, generate 25 primary rays per pixel, and 10 final gathering with 300 samples for the irradiance estimation; 8 M shadow and 67 M secondary rays are generated among all the generated 81 M rays.

In this configuration, our method that uses HPH achieves 3.77 times performance improvement and reduces 85% of the disk I/O accesses over rendering without reordering rays. Note that reordering rays based on HPH shows a relatively low performance improvement in this scene, compared to other scenes. Although there are small differences among the approximate hit points computed from the simplified fur model, there can be big differences among the exact hit points, lowering the ray coherence in the sorted rays. However, we found that considering ray origins or directions in addition to approximate hit points can further improve the performance. For example, when we consider approximate hit points as well as ray directions within our ray reordering method, it achieves a higher improvement, 5.9 times improvement, over rendering without reordering rays.

Complexity of simplified model	0.0125%	0.05%	2%	8%
Rendering time (sec.)	10,644	10,342	10,314	13,789
Overhead (sec.)	637	648	754	964

II: *This table shows the overall rendering time and the total overhead of our method as a function of model complexity of simplified models, represented in the percentage of the original model complexity, in the Sponza benchmark.*

5.3 Analysis

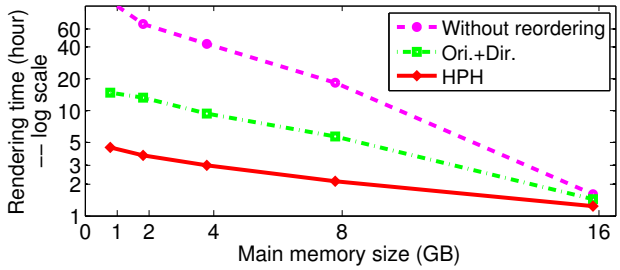
We discuss various factors that affect the performance of our method with the path tracing benchmark of the Sponza scene in this section, unless mentioned otherwise.

Performance vs. complexity of simplified models: The complexity of simplified models can affect the performance improvement of our ray reordering method. We measure the performance improvement caused by our reordering method with different complexities of simplified models (Table II). We achieve the highest performance when we use simplified models whose model complexities are 2% of original models. Moreover, we also found that the performance of our method does not decrease much as we use drastically simplified models (e.g., 0.0125% of the original models for the simplified models).

Overhead: We also measure the total overhead of our method, which consists of computing approximate hit points and sorting rays stored in the ray buffer. We found that the total overhead of our method is 7% of the total rendering time when we use 2% of the original model complexity for the simplified models; sorting rays takes 65% of the total overhead. Also, about 55% of the total rendering time with reordering rays is spent on reading data from the disk, compared to 98% of the total rendering time measured without reordering rays.

Performance vs. ray buffer size: The performance improvement can be affected by the size of the in-core ray buffer. We measure the rendering time as a function of the size of the in-core ray buffer (Table I). As the size is increased, we found that the overhead of our method is decreased. However, as we allocate more memory space for the in-core ray buffer, less memory space is used for other data such as meshes and BVHs. Therefore, we achieve the highest performance when we allocate 88 MB for the in-core ray buffer. However, the performance variation is rather minor in the tested range of the buffer size.

Cache-oblivious nature of our method: Our method uses Z-curves for reordering rays and has the cache-oblivious property caused by using the space-filling curve [Yoon and Lindstrom 2006] that works with different cache parameters. Therefore, it can reduce cache misses occurring between different memory levels including L1/L2 caches, main memory, and disk. To demonstrate the cache-oblivious property of our method, we test our method with photon mapping of the Armadillo model consisting of 346 K triangles in the Cornell box (Fig. 3). The whole data of this small scene takes 43.5 MB, which fits into main memory. In this scene, we reorder rays when our in-core ray buffer is full, instead of dumping rays stored in the ray buffer to the out-of-core ray buffer. In this case, our method shows a 21% overall performance improvement by reordering rays. This improvement is caused by the improved ray process-



7: *This graph shows the rendering time of path tracing in the Sponza scene with different physical main memory sizes, when we use our method or not. We also show the rendering time measured with a reordering method that considers ray origins as well as ray directions together (***Ori.+Dir.***).*

ing throughput, although our method has an overhead of computing approximate hit points and sorting rays, which take about 14% of the overall rendering time. The ray processing throughputs is improved from 135 K rays per second (RPS) to 164 K RPS. We also measure the L2 cache miss ratios by simulating the 6MB wide 24-way set-associative L2 cache of our test machine. We observe more than two times cache miss reduction by reordering rays compared to without reordering rays.

Performance vs. cache size: The performance improvement of our method depends on how much portion of the data of a scene can be stored by different caches. To shed light on this factor, we measure the overall rendering time, as a function of the available memory size with and without using our ray reordering method (Fig 7). For this test, we use a 64 bit machine; note that the OS uses 0.2 GB space from the physical main memory. When we use 16 GB main memory, the whole data of the scene can be uploaded into main memory. Even in this case, our method improves the performance by 31% over rendering without reordering rays, because our method improves the cache utilizations of L1/L2 caches. As we decrease the memory size, the performance of ray tracing also decreases. Nonetheless, the performance with our ray reordering method decreases more gracefully. When caching 1.8 GB, 14.1% of the whole data, in main memory, our method shows a 17.8 times improvement. Even when the available memory size is 0.8 GB, 6.2% of the whole data, our method can render the Sponza scene without I/O thrashing. Also, as we reduce the memory size, the performance improvement of our method increases, since data access time takes a larger portion in the whole rendering time, which gives more room for improvements to our method.

Performance vs. layout: We have used cache-efficient layouts for meshes and hierarchies in this paper, to maximize the benefits of our ray reordering method. Also, the depth-first layout of a BVH has been also widely used in many ray tracers [Yoon and Manocha 2006]. We also measure the performance of our ray tracer with the depth-first layout, to see how much performance degradation our method can have. Even if we use the depth-first layout, we observe only 14% performance degradation over using the cache-efficient layout.

Multi-core architectures: We also test our method in the 32 bit machine with a quad-core CPU. Our reordering method can be easily parallelized since ordering keys of hit points is easily computed by a few simple bit operations. Also, the 2-way merge sort method that we used for sorting rays is easily parallelized. We measure the performance improvement by reordering rays when we use four threads for ray tracing and our reordering method. By reordering rays, we achieve 10.5 times improvement over without reordering rays when we use four threads in the quad-core CPU machine.

Scene	**HPH**	**Ori.**	**Ori.+Dir.**	**HPH+Ori.**	**HPH+Dir.**	**Pharr97**
Sponza scene	16.65	5.16	5.19	7.97	6.36	13.69
St. Matthew scene	12.28	1.7	2.29	4.03	5.36	28.61
Squirrel scene	3.77	0.89	3.49	4.35	5.91	N/A
Small scene	1.21	0.97	1.23	1.18	1.17	N/A

III: *This table shows performance improvements (times) of tested sorting measures over the overall rendering time without reordering rays in our benchmark scenes.* **Ori.**, **Ori.+Dir.**, **HPH+Ori.**, **HPH+Dir.**, *and* **Pharr97** *represent sorting rays based on ray origins, ray origins combined with ray directions, hit points combined with ray origins, hit points with ray directions, and using the cache-aware method of Pharr et al. [1997] respectively.*

6. COMPARISONS AND DISCUSSIONS

We compare the performance of our method (**HPH**) with those of other reordering methods that include a seminal ray reordering method (**Pharr97**) proposed by Pharr et al. [1997] and simple ray reordering methods that sort rays based on ray origins (**Ori.**) and ray origins with ray directions (**Ori.+Dir.**). We also test two variations of our method that sort rays based on hit points with ray origins (**HPH+Ori.**) and hit points with ray directions (**HPH+Dir.**). Note that **Pharr97** is a cache-aware method, while all the other methods including ours are cache-oblivious.

All the techniques except for **Pharr97** are implemented within our space-filling curve based reordering framework described in Sec. 4.3; we use 5 dimensional grids with $k = 12$ for **Ori.+Dir.** and **HPH+Dir.**, and use 6 dimensional grids with $k = 10$ for **HPH+Ori.**. For **Pharr97**, we divide the scene into to a set of chunks, each of which takes about 32 MB and has its own ray queue. We also construct a higher level kd-tree whose leaf node contains a single chunk. We choose to use the kd-tree for the higher level hierarchy instead of a BVH, since the kd-tree can provide early terminations of rays. Then, we construct a low-level BVH for each chunk, to perform a fair comparison with our method that uses BVHs. We follow the scheduling method for chunks as proposed by Pharr et al. [1997].

Reordering methods based on HPH show higher performance improvements over simple cache-oblivious ray reordering methods based on ray origins or ray directions (see Table III). **HPH** shows higher performances over **Ori.** and **Ori.+Dir.** in all the tested scenes, except for the small Armadillo scene (Fig. 3). Even in the small scene, **HPH** has a shorter traversal time than **Ori.+Dir.**. However, the overhead of **HPH**, especially computing approximate hit points, is higher than that of **Ori.+Dir.**. As a result, **HPH** shows a slightly lower performance than **Ori.+Dir.**. **HPH** also shows higher performances than other sorting methods that consider **HPH** as well as **Ori.** (or **Dir.**) in all the tested scenes, except for the squirrel scene that has complex occlusions.

Our method shows about two times slower performance than **Pharr97** in the St. Matthew scene. However, our method shows a slightly higher (e.g., 23%) performance improvement than **Pharr97** in the Sponza scene. **Pharr97** like most previous cache-aware methods [Navratil et al. 2007; Budge et al. 2009] reorders rays as they traverse their scenes or acceleration hierarchies, because the data access patterns of rays are known during the scene or hierarchy traversal. The main benefit of these methods is that since the data access patterns of rays to the hierarchies and meshes are known during the traversal, sorting rays with this information

can result in a low number of cache misses and even higher performances than our method, as demonstrated in the St. Matthew scene. However, this approach requires a tight integration between the ray reordering module and the ray processing module, causing a complication to the overall ray tracing system and a major restructuring of existing systems in order to use these reordering methods. Although our method shows a lower performance over **Pharr97** in the St. Matthew scene, our method did not show a drastically lower performance and shows even higher performance in the Sponza scene. Therefore, we argue that our method can be useful and widly applied to many existing ray tracing system, since it is simple to implement, highly modular, and cache-oblivious in addition to showing comparable performances with **Pharr97**.

Multi-resolution subdivision techniques: Some of the tested scanned models in our benchmark scenes are highly tessellated. One can simplify these highly tessellated models and render them by using multi-resolution subdivision techniques [Christensen et al. 2003; Tabellion and Lamorlette 2004; Djeu et al. 2007] without significant image quality degradations. We would like to point out that our ray reordering can further improve the performance of these multi-resolution methods. More specifically speaking, even though multi-resolution approaches can reduce the number of triangles that have to be processed, we usually have to deal with a huge number of subdivided triangles to provide high quality rendering. Therefore, typical multi-resolution approaches rely heavily on smart caching schemes. Our ray reordering method can maximize the benefits of these kinds of caching schemes by improving the ray coherence. Therefore, our method is not competing with the multi-resolution methods, but complementary to each other.

Optimized ray tracers: Our ray tracer does not have a high ray processing throughput. However, we would like to point out that the performance improvement caused by reducing the number of cache misses can be higher with more highly optimized ray tracing systems. This is because in these optimized systems, the data access time caused by cache misses will take relatively higher portions in the overall ray tracing time. Therefore, our method that reduces this data access time can achieve higher improvements with more optimized ray tracers. To support this argument, we implement multi-BVHs [Ernst and Greiner 2008] that provide an efficient SIMD-based triangle packet and thus improve the ray processing throughput, especially for incoherent secondary rays. We choose to have four child nodes for an intermediate node and four triangles in the leaf node of the multi-BVH. Also, in the 32 bit machine, only 39% of the whole data of the Sponza scene is cached in main memory. Given the multi-BVH, we use a single-ray and single-triangle traversal by disabling to use the SIMD-based triangle packet, as a lower performance ray tracer. In this ray tracer, the ray processing throughputs is improved from 5.49 K rays per second (RPS) to 27.4 K RPS, resulting in a 4.99 times improvement, by reordering rays. Then, we enable the SIMD-based triangle packet traversal for a higher performance ray tracer. In this case, by reordering rays, the ray processing throughput is improved from 5.5 K RPS to 28.92 K RPS, leading to a 5.26 times improvement, a higher improvement than that with the lower performance ray tracer.

Limitations: Our method has certain limitations. Our method inherits drawbacks of existing reordering methods: to use our reordering method in a ray tracer, the ray tracer should be decomposed into separate ray generation and processing modules, and processing rays are performed iteratively by using these modules. Also, our ray reordering method like other ray reordering methods may not work with shaders that do not allow deferred shading, though most general shaders work with the deferred shading. Although we have demonstrated performance improvements over other cache-oblivious reordering methods that use ray origins or ray directions with all the tested benchmarks, there is no guarantee that our method will improve the performance of ray tracing because of the overhead of our method. Also, our reordering method requires simplified representations of original models. Computing such simplified representations require extra implementation efforts. Moreover, for certain class of models such as forest scenes and furry models, computing high-quality simplified representations may be very difficult, thus lowering the performance benefits of our method in such scenes. Also, as demonstrated in the St. Matthew scene, our method can show a lower performance than optimized cache-aware ray reordering methods.

7. CONCLUSION AND FUTURE WORK

We have presented a cache-oblivious ray reordering method that achieves the performance improvement for various models. Our method reorders rays by computing approximate hit points and efficiently sorting them with the Z-curve. We have demonstrated the performance benefits and the high modularity of our method with path tracing and photon mapping, both of which require lots of incoherent rays to generate realistic visual images. By reordering these rays, we have achieved significant performance improvements over other simple cache-oblivious ray reordering methods that use ray origins or ray directions for massive models. Moreover, our method shows a performance improvement for small models that can fit into main memory. This performance improvement is caused by reducing the cache misses of the L1/L2 caches. Also, our method shows comparable performances even with the optimized cache-aware ray reordering method proposed by Pharr el al. [1997].

There are many exciting future directions lying ahead. Currently, we have tested our ray reordering method only with the CPU architecture. It will be very interesting to see how our method can be extended to handle incoherent rays and improve GPU cache utilizations in GPU-based global illumination methods. It will be also interesting to apply our method to hybrid ray tracers that run on both CPUs and GPUs. Finally, although we have presented a cache-oblivious computational reordering method for ray tracing, its idea can be applied to many different applications whose main bottleneck is in the data access time. Therefore, we would like to extend our current method to different computer graphics applications.

Acknowledgements

We would like to thank Brian Budge for the discussion on implementing the cache-aware method. We also thank Stanford University and Marko Dabrovic for providing scanned models and Sponza Atrium model. We are thankful to Christian Lauterbach, the members of SGLab., and the anonymous reviewers for their helpful feedbacks. In particular, we appreciate one of reviewers who suggested to combine the HPH with other ray information (e.g., ray origins) for ray reordering. This work was supported in part by MKE/IITA u-Learning, KRF-2008-313-D00922, MKE/MCST/IITA[2008-F-033-02], MKE digital mask control, MCST/KOCCA-CTR&DP-2009, KMCC, DAPA/ADD (UD080042AD), and the MKE project of semi-realtime renderer.

REFERENCES

ARGE, L., BRODAL, G. S., AND FAGERBERG, R. 2005. *Cache-Oblivious Data Structures in Handbook of Data Structures*. CRC Press.

BOULOS, S., WALD, I., AND BENTHIN, C. 2008. Adaptive ray packet reordering. In *IEEE Symp. on Interactive Ray Tracing*. 131–138.

BUDGE, B., BERNARDIN, T., STUART, J. A., SENGUPTA, S., JOY, K. I., AND OWENS, J. D. 2009. Out-of-core data management for path tracing on hybrid resources. *Computer Graphics Forum (EG) 28,* 2, 385–396.

CHRISTENSEN, P. H., LAUR, D. M., FONG, J., WOOTEN, W. L., AND BATALI, D. 2003. Ray differentials and multiresolution geometry caching for distribution ray tracing in complex scenes. *Computer Graphics Forum 22,* 3 (Sept.), 543–552.

CLINE, D., STEELE, K., AND EGBERT, P. K. 2006. Lightweight bounding volumes for ray tracing. *Journal of Graphics Tools 11,* 4, 61–71.

COOK, R. L., HALSTEAD, J., PLANCK, M., AND RYU, D. 2007. Stochastic simplification of aggregate detail. *ACM Trans. Graph. 26,* 3, 79.

DEMARLE, D. E., GRIBBLE, C. P., AND PARKER, S. G. 2004. Memory-savvy distributed interactive ray tracing. In *EGPGV*. 93–100.

DIAZ-GUTIERREZ, P., BHUSHAN, A., GOPI, M., AND PAJAROLA, R. 2005. Constrained Strip Generation and Management for Efficient Interactive 3D Rendering. In *Computer Graphics International*. 115–121.

DJEU, P., HUNT, W., WANG, R., ELHASSAN, I., STOLL, G., AND MARK, W. R. 2007. Razor: An architecture for dynamic multiresolution ray tracing. Tech. Rep. TR-07-52, The Univ. of Texas at Austin, Dept. of Computer Sciences. January 24.

ERNST, M. AND GREINER, G. 2008. Multi bounding volume hierarchies. In *Interactive Ray Tracing. IEEE Symp. on*. 35–40.

FRIGO, M., LEISERSON, C., PROKOP, H., AND RAMACHANDRAN, S. 1999. Cache-oblivious algorithms. In *Foundations of Computer Science*. 285–297.

GARLAND, M. AND HECKBERT, P. 1997. Surface simplification using quadric error metrics. In *SIGGRAPH 97 Proceedings*. 209–216.

GRIBBLE, C. P. AND RAMANI, K. 2008. Coherent ray tracing via stream filtering. In *IEEE Symposium on Interactive Ray Tracing*. 59–66.

HAVRAN, V. 1997. Cache sensitive representation for the bsp tree. In *Proc. of Compugraphics*.

HECKBERT, P. S. AND HANRAHAN, P. 1984. Beam tracing polygonal objects. In *SIGGRAPH*. ACM Press, New York, NY, USA, 119–127.

HENNESSY, J. L., PATTERSON, D. A., AND GOLDBERG, D. 2007. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann.

JENSEN, H. W. 2005. *Realistic Image Synthesis Using Photon Mapping*. AK Peters.

KIM, T.-J., BYUN, Y., KIM, Y., MOON, B., LEE, S., AND YOON, S.-E. 2010. HCCMeshes: Hierarchical-culling oriented compact meshes. *Computer Graphics Forum (Eurographics) 29,* 2. To appear.

KIM, T.-J., MOON, B., KIM, D., AND YOON, S.-E. 2010. RACBVHs: Random-accessible compressed bounding volume hierarchies. *IEEE Trans. on Visualization and Computer Graphics 16,* 2, 273–286.

LAUTERBACH, C., GARLAND, M., SENGUPTA, S., LUEBKE, D., AND MANOCHA, D. 2009. Fast bvh construction on gpus. *Computer Graphics Forum (EG) 28,* 2, 375–384.

LAUTERBACH, C., YOON, S.-E., TANG, M., AND MANOCHA, D. 2008. ReduceM: Interactive and memory efficient ray tracing of large models. *Computer Graphics Forum (EG Symp. on Rendering) 27,* 4, 1313–1321.

LAUTERBACH, C., YOON, S.-E., TUFT, D., AND MANOCHA, D. 2006. RT-DEFORM: Interactive ray tracing of dynamic scenes using bvhs. In *IEEE Symp. on Interactive Ray Tracing*. 39–46.

LUEBKE, D., REDDY, M., COHEN, J., VARSHNEY, A., WATSON, B., AND HUEBNER, R. 2002. *Level of Detail for 3D Graphics*. Morgan-Kaufmann.

MANSSON, E., MUNKBERG, J., AND AKENINE-MOLLER, T. 2007. Deep coherent ray tracing. In *IEEE Symp. on Interactive Ray Tracing*. 79–85.

NAVRATIL, P., FUSSELL, D., LIN, C., AND MARK, W. 2007. Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *IEEE Symposium on Interactive Ray Tracing*. 95–104.

PHARR, M. AND HUMPHREYS, G. 2004. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

PHARR, M., KOLB, C., GERSHBEIN, R., AND HANRAHAN, P. 1997. Rendering complex scenes with memory-coherent ray tracing. In *ACM SIGGRAPH*. 101–108.

RESHETOV, A. 2007. Faster ray packets - triangle intersection through vertex culling. In *IEEE Symp. on Interactive Ray Tracing*. 105–112.

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level ray tracing algorithm. *ACM Trans. Graph. 24,* 3, 1176–1185.

SAGAN, H. 1994. *Space-Filling Curves*. Springer-Verlag.

SHIRLEY, P. AND MORLEY, R. K. 2003. *Realistic Ray Tracing*, Second ed. AK Peters.

SILVA, C., CHIANG, Y.-J., CORREA, W., EL-SANA, J., AND LINDSTROM, P. 2002. Out-of-core algorithms for scientific visualization and computer graphics. In *IEEE Visualization Course Notes*.

STEINHURST, J., COOMBE, G., AND LASTRA, A. 2005. Reordering for cache conscious photon mapping. In *Graphics Interface*. 97–104.

STEPHENS, A., BOULOS, S., BIGLER, J., WALD, I., AND PARKER, S. G. 2006. An Application of Scalable Massive Model Interaction using Shared Memory Systems. In *EGPGV*. 19–26.

TABELLION, E. AND LAMORLETTE, A. 2004. An approximate global illumination system for computer generated films. *ACM Trans. Graph. 23,* 3, 469–476.

VAN EMDE BOAS, P. 1977. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett. 6*, 80–82.

VITTER, J. S. 2001. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv. 33,* 2, 209–271.

WALD, I., BOULOS, S., AND SHIRLEY, P. 2007. Ray Tracing Deformable Scenes using Dynamic Bounding Volume Hierarchies. *ACM Transactions on Graphics 26,* 1, 6.

WALD, I., DIETRICH, A., AND SLUSALLEK, P. 2004. An Interactive Out-of-Core Rendering Framework for Visualizing Massively Complex Models. In *EG Symp. on Rendering*. 82–91.

WALD, I., MARK, W. R., GÜNTHER, J., BOULOS, S., IZE, T., HUNT, W., PARKER, S. G., AND SHIRLEY, P. 2007. State of the Art in Ray Tracing Animated Scenes. In *Eurographics State of the Art Reports*.

WALD, I., SLUSALLEK, P., AND BENTHIN, C. 2001. Interactive distributed ray tracing of highly complex models. In *EG Workshop on Rendering*. 277–288.

YOON, S.-E., GOBBETTI, E., KASIK, D., AND MANOCHA, D. 2008. *Real-Time Massive Model Rendering*. Morgan & Claypool Publisher.

YOON, S.-E. AND LINDSTROM, P. 2006. Mesh layouts for block-based caches. *IEEE Trans. on Visualization and Computer Graphics (Proc. Visualization) 12,* 5, 1213–1220.

YOON, S.-E., LINDSTROM, P., PASCUCCI, V., AND MANOCHA, D. 2005. Cache-Oblivious Mesh Layouts. *ACM Transactions on Graphics (SIGGRAPH) 24,* 3, 886–893.

YOON, S.-E. AND MANOCHA, D. 2006. Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum (Eurographics) 25,* 3, 507–516.