DOI: 10.1111/cgf.12158
Eurographics Symposium on Rendering 2013
Nicolas Holzschuch and Szymon Rusinkiewicz
(Guest Editors)

Volume 32 (2013), Number 4

Sorted Deferred Shading for Production Path Tracing

Christian Eisenacher   Gregory Nichols   Andrew Selle   Brent Burley

Walt Disney Animation Studios

Abstract

Ray-traced global illumination (GI) is becoming widespread in production rendering but incoherent secondary ray traversal limits practical rendering to scenes that fit in memory. Incoherent shading also leads to intractable performance with production-scale textures forcing renderers to resort to caching of irradiance, radiosity, and other values to amortize expensive shading. Unfortunately, such caching strategies complicate artist workflow, are difficult to parallelize effectively, and contend for precious memory. Worse, these caches involve approximations that compromise quality. In this paper, we introduce a novel path-tracing framework that avoids these tradeoffs. We sort large, potentially out-of-core ray batches to ensure coherence of ray traversal. We then defer shading of ray hits until we have sorted them, achieving perfectly coherent shading and avoiding the need for shading caches.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing

1. Introduction

Path-traced GI promises many benefits for production rendering: it can produce richer, more plausible results with far fewer lights, and avoids the data-management burden associated with huge point clouds and deep shadow maps. While this often results in better images and higher productivity, rendering performance degrades significantly as unique geometric and texture detail scales beyond the capacity of main memory. Hence, existing production renderers focus on tracing fewer rays and shading fewer points, and encourage the use of lighting and shading caches and geometric instancing. Such measures compromise artistic intent.

As ray-tracing research improves raw geometry intersection, incoherent shading becomes a relatively larger problem [CFLB06]. In particular, incoherent access to production-scale texture data incurs frequent cache-misses, leading to memory, disk and network delays that dominate run-time. Irradiance and radiosity caches [KTO11, CHS*12] attempt to mitigate this, though their inherent non-directionality makes them suitable only for perfectly diffuse surfaces. Consequently, common practice separates integration strategies for diffuse rays, specular rays, and caustics, re-complicating artist workflow. Worse, these approaches are difficult to scale to multi-core architectures because of synchronization, NUMA effects and load-balancing. Thus, it is desirable to avoid caches and instead seek to ensure coherent shading.

In this paper, we present a streaming ray-tracer, capable of performing multi-bounce GI on production-scale scenes without resorting to shading caches or instancing. To achieve this, we introduce a novel two-stage ray sorting framework. First, **we sort large, potentially out-of-core ray batches** to ensure coherence. Working with large batches is essential to extract coherent ray groups from complex scenes. Second, **we sort ray hits for deferred shading with out-of-core textures**. For each batch we achieve perfectly coherent shading with sequential texture reads, eliminating the need for a texture cache. Our approach is simple to implement and compatible with most scene traversal strategies. We demonstrate the effectiveness of our two-stage sorting with out-of-core ray batches and compare our results with two state-of-the-art production ray tracers, Pixar's PhotoRealistic RenderMan and Solid Angle's Arnold.

2. Related Work

Much research focuses on efficient recursive traversal for both coherent and incoherent rays. Wald et al. [WSBW01] introduced packets which were improved by others [RSH05, BWB08, ORM08]. Several researchers proposed eliminating packets in favor of SIMD traversal of hierarchies with a branching factor equal to the SIMD width [DHK08, EG08, WBB08]. Benthin et al. [BWW*12] proposed a hybrid single-ray/packet approach. However, incoherent rays and

© 2013 The Author(s)
Computer Graphics Forum © 2013 The Eurographics Association and Blackwell Publishing Ltd. Published by Blackwell Publishing, 9600 Garsington Road, Oxford OX4 2DQ, UK and 350 Main Street, Malden, MA 02148, USA.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

Figure 1: *Scene rendered with production shaders in 35 minutes (5 ray sorting, 12.5 traversal, 1.5 hit point sorting, 15 shading, 1 system overhead). 1920×900, 512 SPP, max. path length 5; 133 M triangles and 15.6 GB unique textures.*

small packets still incur high memory latency costs during traversal, restricting scenes to fit in memory.

To improve coherence, many researchers use ray queues to reorder intersection tests. This is referred to variously as breadth-first ray tracing [Han86, LMW90], ray reordering [PKGH97], and ray streaming [GR08, Tsa09]. Rays have been queued at grid boundaries [PKGH97, NO97, RCJ99, WSBW01, PFHA10, Bik12] or scene hierarchy nodes [BBS*09, Tsa09, NFLM07] to amortize object access cost. Ray queues have also been used for distributed rendering to leverage more compute resources or handle scenes that don't fit in memory on a single computer [WSBW01, KS02]. Ray queues have been scheduled in approximate front-to-back traversal order [Tsa09], and prioritized by generation [KS02] or for optimal data cache usage [BBS*09].

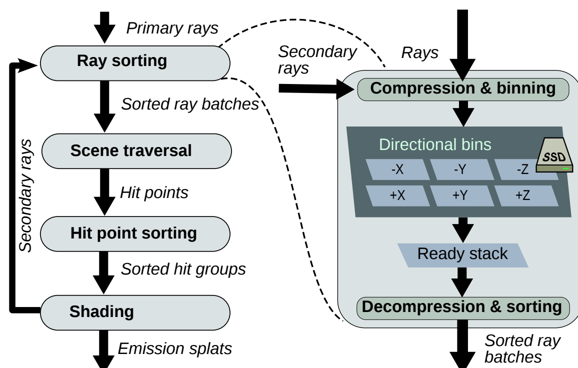

Figure 2: *Left: We ensure coherent path tracing using two sorting stages. Right: An exploded view of ray sorting.*

To extract more coherence, other researchers reorder large ray batches, either full generations or fractions thereof as limited by memory. Arvo and Kirk [AK87] organize rays into a 5D spatio-directional beam tree and then intersect them against an object stream. Garanzha and Loop [GL10] use a uniform, user-specified grid to group rays in 5D bins that require scene-specific calibration. Moon et al. [MBK*10] reorder rays based on approximate hit points traced against a carefully constructed, simplified version of the scene. Hanika et al. [HKL10] record the N nearest patch bounding box hits for each ray in a large batch then intersect each patch with its candidate rays. Top-level traversal remains incoherent, which may be an issue for scenes with many patches; the fixed depth complexity means that if more than N patch bounding boxes overlap then the correct intersection point may not be found immediately.

We are interested in efficient shading for raytraced global illumination. While many researchers demonstrate impressive ray-per-second performance or consider out-of-core occlusion queries for production scale geometry [PFHA10], as far as we know only a few [PKGH97, CFLB06, BBS*09] consider out-of-core texture access, and many do not mention textures at all. Hanika et al. [HKL10] sort shading points by material, but only present results for in-core measured BRDF data where the sorting was shown to be unnecessary. Hoberock et al. [HLJH09] compact, reorder and schedule a stream of deferrred shader invocations on a GPU. This reduces SIMD divergence by calling a small number of shaders coherently, but does not address production scale geometry, textures, or the number and complexity of shaders.

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

3. Sorted Path-Tracing

An overview of our path-tracing pipeline is given on the left of Figure 2. Starting with primary rays we perform **ray sorting**: We bin rays by direction and group them into large, sorted ray batches of fixed size, typically about 30-60 M rays per batch. The OS streams inactive ray batches to a local SSD until the system is ready to sort and trace the next batch.

Next, we perform **scene traversal**, one sorted ray batch at a time. Any hierarchical depth- or breadth-first traversal strategy will likely benefit from ray reordering. We currently use a two-level quad-BVH [Tsa09] with streaming packet-traversal [GL10] in the top level, and naïve single-ray traversal in the bottom. We exploit the fact that our ray batches are directionally coherent to perform approximate front-to-back traversal at each node. The result of traversal is a list of hit points (one per ray).

Next, **hit point sorting** organizes ray hits by shading context. Each subdivision mesh is associated with one or more texture-files containing a per-face texture [BL08] for each layer; thus, a full shading context consists of a mesh ID and a face ID. We group hit points by mesh ID, and then sort each group by face ID for coherent texturing and shading.

Shading happens in parallel with each thread processing a different mesh. If a shading task has many hit points, it is partitioned into sub-tasks, further increasing parallelism. If an object or hit point is found to be emissive, its emission is splatted into the image buffer. The shader also feeds secondary rays back into ray sorting to continue ray paths.

3.1. Ray Sorting

As illustrated on the right of Figure 2, we compress rays after they are created, and add them to the end of one of six cardinal direction bins in a parallel and lock-free manner. Each bin holds a single batch of rays in a memory-mapped file of fixed capacity. For each thread we first add rays to one of six small local buffers based on the major ray direction. When a local buffer is full, we atomically increment the size of the corresponding bin and `memcpy()` the local rays.

As we fill the bins, the operating system (OS) streams the rays to a local solid state disk (SSD). When the incremented size exceeds a bin's capacity we lock on a mutex, close the file, add its filename to a ready stack, and start a new empty bin in its place. When the current batch has finished traversal and shading, we pop the next batch filename from the ready stack, and decompress, sort, and place the rays in a global active ray buffer. To minimize I/O waits we stream in the next batch as we process the current one.

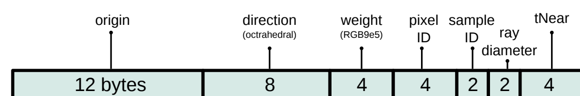

Figure 3: *Layout of a compressed ray with 36 bytes per ray.*

To sort ray batches we perform recursive median partitioning along the longest axis of the current subset at each step. We first partition based on ray origins until we reach subset of no more than 4096 rays. Then we partition based on ray directions until we obtain groups of 64 rays. During traversal each 64-ray leaf group forms a coherent ray packet.

Each ray represents the last segment of a path starting on the image plane. We store a minimal amount of information per path, and further compress it, using lossless octahedral normals [MSS*10] for ray directions, and the shared exponent RGB9e5 format for the aggregate path weight. Overall we store 36 bytes per ray as shown in Figure 3. We store compressed rays in an array-of-structs layout, which is convenient for streaming them to disk as they are generated. During decompression we restore the ray data to floats, convert it to a more efficient structs-of-arrays layout, and precompute additional values that are beneficial for traversal.

3.2. Deferred Shading

After traversal, each ray in the active ray buffer has a corresponding ray hit point consisting of differential geometry, mesh ID, face ID, and face UV. We first group hit points by mesh ID using a CPU parallel radix sort [SHG09]. Then we dispatch one shading task for each group of hit points to run in a separate thread. At the start of each shading task, we pre-sort the hit points by face ID, so that the shading order exactly matches the on-disk order of the per-face textures.

Because each mesh face is touched at most once when shading a ray batch, all shader inputs, including texture maps, are only accessed once. This amortizes per-file texture costs (opening the file over the network), and per-face texture costs (reading and decompressing a block of texels) perfectly for each batch. Further this access is coherent and sequential, so prefetching of subsequent per-face textures is trivial—completely eliminating the need for texture caches.

This is beneficial, as it frees memory for use in streaming additional active rays or storing more unique in-core geometry. Moreover, maintaining a large cache across multiple ray batches does not necessarily improve performance anyway; in production-scale scenes with many large textures, subsequent batches are unlikely to use the same textures at the same resolutions. In these cases, the overhead of managing the cache can actually decrease performance.

4. Results

We validate our method on test scenes with different levels of geometric and shading complexity. Each example limits paths to length 4, 5, or 6 (i.e. 2, 3, or 4 "bounces of indirect"). Our shader creates one reflection ray, except on the first reflection, where it creates four BRDF samples and one

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

Figure 4: *Interior scene: 818×580, 1024 SPP, path length 6; 70.5 M triangles, 13.6 GB unique textures; 68 minutes.*

light sample. Textures are stored compressed on a texture server. We render using 12 threads on a 12-core Xeon 5675 system with 48 GB memory.

4.1. Interior Scene

Figure 4 shows a production interior scene lit by a single area light, demonstrating rich global illumination with a maximum path length of six. Each material consists of four texture layers totaling 13.6 GB - small compared to production assets that are often authored with dozens of texture layers and masks. To focus on texturing cost, we use simple Lambertian reflectance and simple geometry. Specifically, we uniformly subdivide 2710 Catmull-Clark subdivision surfaces with 551 k faces into 70.5 M triangles. Figure 4 is rendered with 1024 samples per pixel (SPP), but timings use only 64 SPP to keep the unsorted render times tractable.

In Figure 5, we analyze the impact of the two sorting stages. Disabling both, our performance is similar or worse than other production renderers (see Section 4.3). Sorting just the rays improves both traversal and shading performance. Sorting just the hit points during deferred shading dramatically improves texturing performance, but yields no benefits for traversal. Sorting both rays and hit points produces the best performance, with negligible cost for sorting.

Figure 6: *Top: an artistic view of the city scene, highlighting its texture detail. Bottom: the scene as rendered in the tests. The red circle highlights the rooftop from the top image.*

Figure 7 explores how larger ray batches extract more coherence and thus improve rendering performance of the scene. While increasing the ray batch size from 2^{15} to 2^{25} yields a significant 5× speed improvement in traversal alone, the relatively more expensive shading component improves 20×.

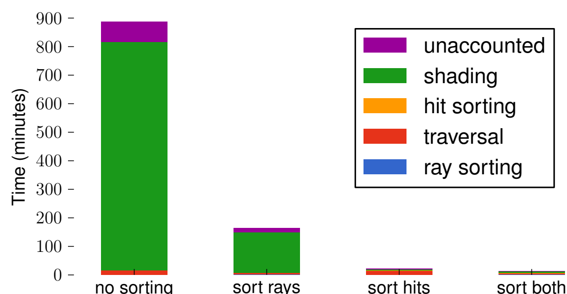

Figure 5: ***Interior scene with a single texture layer.*** *Sorting significantly reduces shading time (the dominant cost) due to more coherent texture lookups.*

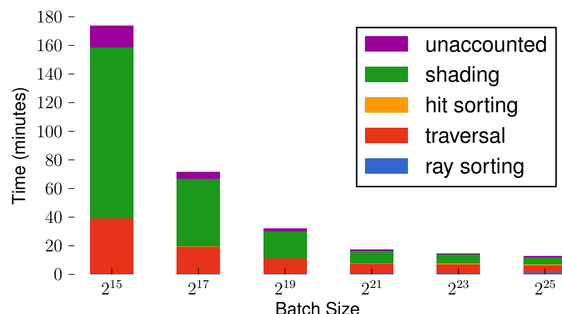

Figure 7: ***Interior scene with single texture layer.*** *Larger ray batches allow our method to discover more coherence through sorting, leading to better performance.*

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

4.2. City Scene

Figure 6 depicts a procedural city scene lit by a sky dome and a single area light. It consists of 51.6 k meshes with a total of 53 M cage faces (106 M triangles), where building templates are duplicated on disk rather than instanced. It is also more interesting than the interior scene because of its more complex base geometry (53 M cage faces instead of 551 k), increased depth complexity, and its larger resolution - significantly taxing our out-of-core batch system. For increased challenge, we also use our full production shader with eight color layers (totaling 21.6 GB of high-resolution texture data).

This setup increases traversal cost, shader computation per hit, and texture reads. Moreover, this setup features many more objects and hence many more texture files, which we must re-open over the network for each batch.

As Figure 8 illustrates, texture access dominates render time even more, emphasizing the importance of coherent texture access in production scale rendering. As in the previous example, both traversal and render time improve considerably with sorting and larger batch sizes. With the additional workload in our system, we observe significant improvements for larger batch sizes: even the apparently small improvement between 2^{23} and 2^{25} is actually a 33% reduction in overall render time. This suggests that increasing batch size even more could be beneficial.

4.3. Comparison with Production Renderers

In Figure 9, we compare render times of the interior scene for our method vs two leading production renderers. Ray counts and sampling strategies were matched as closely as possible, and all advanced features such as adaptive sampling, radiosity caching, etc., were turned off to isolate raw intersection and shading performance. The built-in *ptexture* shadeop was used in PRMan and a comparable shader node was implemented in Arnold using the same *libPtex* library. Both were configured to cache 1000 texture files and 100MB per thread, a generous cache size for 2710 texture files per layer. Notably, our method does not use a persistent texture cache, and requires reopening each texture file over the network each time a given surface is shaded.

Rendering performance without textures was dominated by ray traversal time and was very similar between the renderers (9.8, 10.7, 8.7 minutes for Arnold, PRMan, and our method, respectively). When textures were added, the renderers without sorting performed poorly, exhibiting a superlinear increase in cost for each additional texture layer. With our method, each additional texture layer added a modest linear cost of roughly 52 seconds. For four texture layers, render times were 1094, 214, and 11.2 minutes for Arnold, PRMan, and our method. Without sorting, our render times degenerated to 819 minutes, around 70× slower.

4.4. Resource Usage

Figure 10 explores system resource usage. With 46.2 GB available on our test machine (left column), the geometry of the interior scene, its four texture layers, and all ray batches fit entirely in core, yielding good CPU utilization. Although the OS speculatively writes all ray batches to the SSD, it can cache them in memory and never has to read them back. Similarly it is able to serve most texture files from cache after some initial network traffic to the texture servers.

To force our renderer out-of-core, we launch a simple “memhog” application that allocates and locks the majority of memory, leaving only 16.2 GB available (right column). Without sufficient space for OS caches we now observe additional SSD reads as we must stream the ray batches back into memory. Similarly, while the OS caches some of the early texture reads, we observe more traffic to the texture servers once the renderer reaches its maximum memory footprint.

In the out-of-core scenario, CPU utilization is interrupted by bursts of low utilization: the OS cannot buffer and sched-

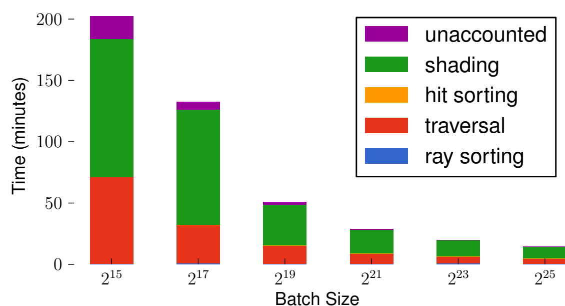

Figure 8: ***City scene with eight texture layers.*** *Large ray batches allow our method to extract more coherence and significantly improve shading and traversal times.*

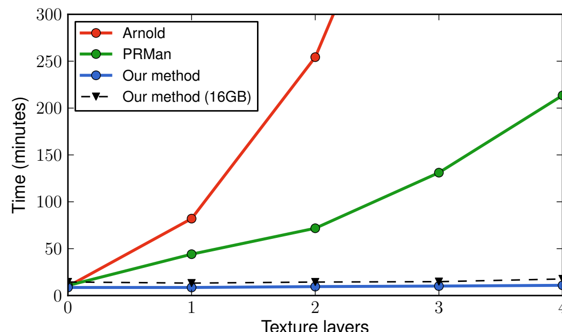

Figure 9: *Comparing texture performance of our method against production renderers on the interior scene.*

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

ule the slower SSD writes as effectively. The interior scene is so fast to trace and shade that we cannot hide the I/O latency of streaming ray batches from disk entirely, and there is room for improvement prefetching textures from the servers. However, we observe only a modest increase in render time: from 8.7 to 14.5 minutes for a single layer and from 11.2 to 17.7 minutes for four texture layers. With only one additional GB available, the latter reduces to 14 minutes, suggesting that even a small amount of OS cache is beneficial.

5. Discussion and Future Work

Larger ray batches increase efficiency thanks to added coherence discovered by sorting. However, there is a tradeoff between performance gained from more coherency, and performance lost by ray memory usage starving other parts of the system, such as the geometry or OS cache. For our scenes and test system, 2^{25} was the largest practical batch size; larger batch sizes may be beneficial for systems with more memory. Perhaps the most significant benefit of shading rays in such massive batches is that it enables the use of high detail textures during path tracing. However, to handle large ray batches our method traces rays one segment-at-a-time, making certain existing techniques more challenging to apply.

Currently our renderer employs naïve forward path tracing, a well-understood and widely used method. This requires many samples to resolve difficult light paths, so other researchers employ bidirectional techniques. Such methods typically require full light paths, precluding their use in our framework without modification. However, two recent techniques [GKDS12, HPJ12] combine sampling of light paths with eye paths in a way that, as illustrated on Figure 6 in [GKDS12], can be performed a segment at a time, without access to full paths.

Similar tradeoffs exist in shading, where post-hit techniques like radiance closures fit well with recursive ray tracing but would be difficult to apply in our system without modification. Kato and Saito [KS02] address this by saving and restoring shader state at significant memory cost, but it is not clear whether we could do the same without fine-grained ray queues, which would likely destroy our coherence. Even so, we believe our performance gains will motivate research into compatible approaches to shading.

Not all of our choices imply compromise. For example, though all the tests in this paper use Ptex [BL08], this is not required. Our method can be formulated equally well in the context of conventional texture storage methods. In particular, atlased models could be accessed in groups sorted by subimage and UV order.

Finally, due to the coherence afforded by our method, we believe it lends itself particularly well to out-of-core rendering of scenes with massive amounts of geometry. While Figure 10 shows that there is room for improvement in resource utilization, our preliminary experiments are encouraging: we rendered the procedural city scene with 260 k unique meshes (530 M triangles) in 2:43 hours for 64 SPP, and in 16 hours for 512 SPP (shown in Figure 11). An even larger scene with 2.6 M meshes (5.4 G triangles on disk) rendered 64 SPP in 15 hours. In both cases we only used a single texture layer, and though our current approach is fairly naïve, we observed a very modest 2× increase of total render time for going out-of-core with geometry. These initial experiments are a very promising basis for future work.

6. Conclusion

We have presented a novel approach to efficiently path-trace global illumination on production-scale scenes that use out-of-core texture maps. We have demonstrated that sorting large batches of rays using out-of-core ray streaming extracts a much greater degree of coherence than is possible with smaller ray batches, and that sorting ray hit points before shading avoids the need for a texture cache. Moreover, we believe that the pairing of these two sorting strategies is the key to adoption of GI without artistic compromise.

Acknowledgements

We would like to thank Chuck Tappan for his enthusiastic support of this work, and for lighting and texturing all the scenes in the paper. Thanks also to John Huikku and Konrad Lightner for providing data for the city scene, and Lawrence Chai for his input on the paper. Finally, we would like to thank our anonymous reviewers for their insightful and constructive comments, which greatly improved the final version of this paper.

References

[AK87] Arvo J., Kirk D.: Fast ray tracing by ray classification. In *Proc. of SIGGRAPH* (1987).

[BBS*09] Budge B., Bernardin T., Stuart J. A., Sengupta S., Joy K. I., Owens J. D.: Out-of-core data management for path tracing on hybrid resources. In *Computer Graphics Forum* (2009).

[Bik12] Bikker J.: Improving data locality for efficient in-core path tracing. In *Computer Graphics Forum* (2012).

[BL08] Burley B., Lacewell D.: Ptex: per-face textures for production rendering. In *Proc. of EGSR* (2008).

[BWB08] Boulos S., Wald I., Benthin C.: Adaptive ray packet reordering. In *Proc. of Interactive Ray Tracing* (2008).

[BWW*12] Benthin C., Wald I., Woop S., Ernst M., Mark W. R.: Combining single and packet-ray tracing for arbitrary ray distributions on the Intel MIC architecture. *Trans. on Visualization and Computer Graphics* (2012).

[CFLB06] Christensen P. H., Fong J., Laur D. M., Batali D.: Ray tracing for the movie Cars. In *Proc. of Interactive Ray Tracing* (2006).

[CHS*12] Christensen P. H., Harker G., Shade J., Schubert B., Batali D.: *Multiresolution radiosity caching for efficient preview and final quality global illumination in movies*. Tech. rep., Pixar, 2012.

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

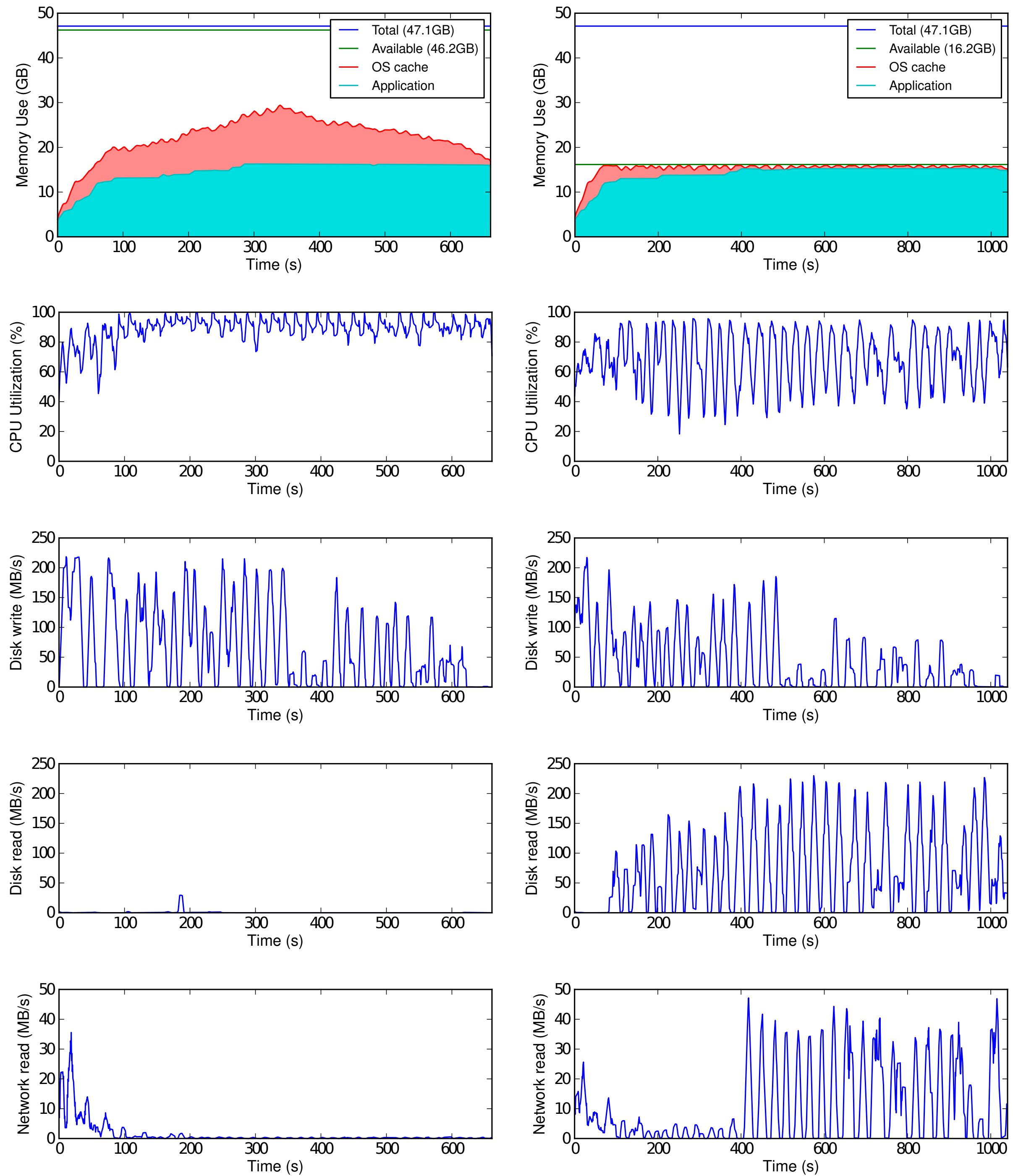

Figure 10: *System profile during rendering of the interior scene with 4 textures. Left column: with 46.2 GB of available memory, ray batches and textures are cached by the OS, and minimal disk/network reads are required. Right column: restricting available memory to 16.2 GB prevents OS caching, requiring local reads for ray batches and network reads for texture data. Spikes correspond to ray batches being processed.*

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.

Figure 11: *Out-of-core city scene: 4096×1716, 512 SPP, path length 4; 530 M triangles, no instancing, 16 hours.*

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In *Computer Graphics Forum* (2008).

[EG08] ERNST M., GREINER G.: Multi bounding volume hierarchies. In *Proc. of Interactive Ray Tracing* (2008).

[GKDS12] GEORGIEV I., KŘIVÁNEK J., DAVIDOVIČ T., SLUSALLEK P.: Light transport simulation with vertex connection and merging. *ACM Trans. on Graphics* (2012).

[GL10] GARANZHA K., LOOP C.: Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Computer Graphics Forum* (2010).

[GR08] GRIBBLE C. P., RAMANI K.: Coherent ray tracing via stream filtering. In *Proc. of Interactive Ray Tracing* (2008).

[Han86] HANRAHAN P.: Using caching and breadth first traversal to speed up ray tracing. In *Proc. of Graphics Interface* (1986).

[HKL10] HANIKA J., KELLER A., LENSCH H. P. A.: Two-level ray tracing with reordering for highly complex scenes. In *Proc. of Graphics Interface* (2010).

[HLJH09] HOBEROCK J., LU V., JIA Y., HART J. C.: Stream compaction for deferred shading. In *Proc. of High Performance Graphics* (2009).

[HPJ12] HACHISUKA T., PANTALEONI J., JENSEN H. W.: A path space extension for robust light transport simulation. *ACM Trans. on Graphics* (2012).

[KS02] KATO T., SAITO J.: Kilauea: parallel global illumination renderer. In *Proc. of Eurographics Workshop on Parallel Graphics and Visualization* (2002).

[KTO11] KONTKANEN J., TABELLION E., OVERBECK R. S.: Coherent out-of-core point-based global illumination. In *Computer Graphics Forum* (2011), vol. 30.

[LMW90] LAMPARTER B., MUELLER H., WINCKLER J.: *The Ray-z-Buffer – an approach for ray tracing arbitrarily large scenes*. Tech. rep., Univ. Freiburg Inst. für Informatik, 1990.

[MBK*10] MOON B., BYUN Y., KIM T.-J., CLAUDIO P., KIM H.-S., BAN Y.-J., NAM S. W., YOON S.-E.: Cache-oblivious ray reordering. *ACM Trans. on Graphics* (2010).

[MSS*10] MEYER Q., SÜSSMUTH J., SUSSNER G., STAMMINGER M., GREINER G.: On floating-point normal vectors. In *Computer Graphics Forum* (2010).

[NFLM07] NAVRÁTIL P. A., FUSSELL D. S., LIN C., MARK W. R.: Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Proc. of Interactive Ray Tracing* (2007).

[NO97] NAKAMARU K., OHNO Y.: Breadth-first ray tracing utilizing uniform spatial subdivision. *Trans. on Visualization and Computer Graphics* (1997).

[ORM08] OVERBECK R., RAMAMOORTHI R., MARK W. R.: Large ray packets for real-time Whitted ray tracing. In *Proc. of Interactive Ray Tracing* (2008).

[PFHA10] PANTALEONI J., FASCIONE L., HILL M., AILA T.: PantaRay: fast ray-traced occlusion caching of massive scenes. *ACM Trans. on Graphics* (2010).

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. In *Proc. of SIGGRAPH* (1997).

[RCJ99] REINHARD E., CHALMERS A., JANSEN F. W.: Hybrid scheduling for parallel rendering using coherent ray tasks. In *Proc. Symp. on Parallel Visualization and Graphics* (1999).

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. In *ACM Trans. on Graphics* (2005).

[SHG09] SATISH N., HARRIS M., GARLAND M.: Designing efficient sorting algorithms for manycore gpus. In *IEEE International Symp. on Parallel & Distributed Processing* (2009), pp. 1–10.

[Tsa09] TSAKOK J. A.: Faster incoherent rays: Multi-BVH ray stream tracing. In *Proc. of High Performance Graphics* (2009).

[WBB08] WALD I., BENTHIN C., BOULOS S.: Getting rid of packets: efficient SIMD single-ray traversal using multi-branching BVHs. In *Proc. of Interactive Ray Tracing* (2008).

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive distributed ray tracing of highly complex models. In *Proc. of Eurographics Workshop on Rendering Techniques* (2001).

© 2013 The Author(s)
© 2013 The Eurographics Association and Blackwell Publishing Ltd.