EUROGRAPHICS 2015 / O. Sorkine-Hornung and M. Wimmer
(Guest Editors)

Volume 34 (2015), Number 2

CHC+RT: Coherent Hierarchical Culling for Ray Tracing

O. Mattausch[1,2] and J. Bittner[3] and A. Jaspe[4] and E. Gobbetti[4] and M. Wimmer[5] and R. Pajarola[1]

[1]University of Zurich [2]ETH Zurich [3]Czech Technical University in Prague [4]CRS4, Italy [5]Vienna University of Technology

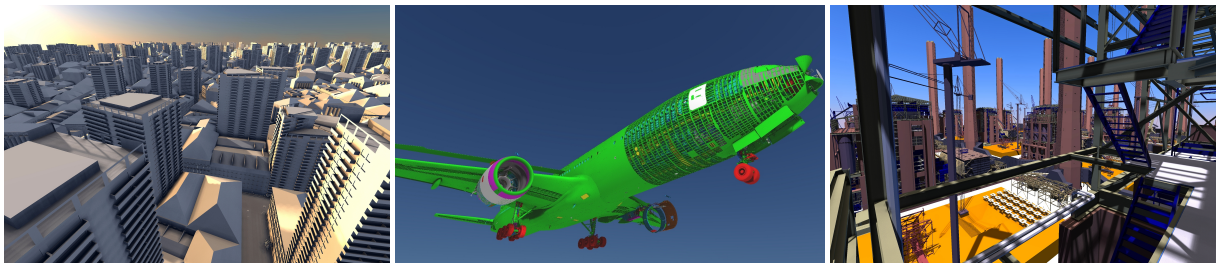


Figure 1: *Sample images from our interactive OpenGL ray tracer using the CHC+RT algorithm. From left to right: A City model with 138M triangles (7.89GB), the Boeing 777 model with 350M triangles (18.9GB) and 16 copies of the Powerplant model with 205M triangles (11.4GB). Our algorithm based on hierarchical occlusion culling allows a simple scheduling scheme for managing out-of-core scenes and also significantly accelerates OpenGL-based ray tracing in complex scenes.*

Abstract

We propose a new technique for in-core and out-of-core GPU ray tracing using a generalization of hierarchical occlusion culling in the style of the CHC++ method. Our method exploits the rasterization pipeline and hardware occlusion queries in order to create coherent batches of work for localized shader-based ray tracing kernels. By combining hierarchies in both ray space and object space, the method is able to share intermediate traversal results among multiple rays. We exploit temporal coherence among similar ray sets between frames and also within the given frame. A suitable management of the current visibility state makes it possible to benefit from occlusion culling for less coherent ray types like diffuse reflections. Since large scenes are still a challenge for modern GPU ray tracers, our method is most useful for scenes with medium to high complexity, especially since our method inherently supports ray tracing highly complex scenes that do not fit in GPU memory. For in-core scenes our method is comparable to CUDA ray tracing and performs up to 5.94× *better than pure shader-based ray tracing.*

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Picture/Image Generation—Line and curve generation

1. Introduction

Depth-buffered rasterization and ray tracing are nowadays the two dominant techniques in real-time rendering. In its basic form, rasterization is an object-order approach that determines visible surfaces by going through scene primitives, projecting them to screen and maintaining the nearest surface for each pixel. Ray tracing, on the other hand, is an image-order approach that determines visible surfaces by computing ray-primitive intersections for each pixel.

In principle, rasterization offers more code- and data-cache coherence, because switching primitives and rendering attributes occurs much less frequently, and most operations work on an object-by-object basis on data residing in local memory, without the need to access the entire scene. This explains the success of massively parallel GPU rasterization hardware based on streaming architectures. In contrast, for ray tracing, data is usually organized in space-partitioning data structures, and the traversal of these data structures results in non-streaming access patterns to the scene geometry. Even though current GPUs support general programming models and allow for programming acceleration data

structures and complex traversal algorithms, efficient memory management and computation scheduling is significantly harder than for rasterization, leading to performance problems and/or complications when trying to integrate rasterization and ray tracing within the same application, e.g., to compute complex global illumination.

We address these issues by proposing a ray-tracing technique that is designed to be integrated into the streaming rasterization pipeline. The core idea of the method is to exploit the rasterization pipeline together with occlusion queries in order to create coherent batches of work for GPU ray tracing. By combining hierarchies in both ray space and object space, and making use of temporal coherence, the ray-traversal overhead is minimized, and the method can concentrate on computing ray-object intersections for significantly reduced sets of rays and objects. This batched computation and memory-management approach makes it possible to use the same streaming schemes employed in current rasterization systems also for ray tracing. This opens the door to a flexible integration of rasterization and ray tracing, both for dynamic and out-of-core scenes. We show the efficiency of our method for several ray types like soft-shadow rays and diffuse interreflections. The main contributions of our paper are:

- Occlusion culling for ray tracing using the rasterization pipeline, which is up to 6 times faster than standalone OpenGL-based ray tracing.
- A means for scheduling visible parts of the scene hierarchy for ray-triangle intersection on the GPU that allows a simple and natural extension to out-of core ray tracing.

2. Related Work

Our work generalizes hierarchical occlusion culling, a technique traditionally used for accelerating *rasterization*, to incorporate *ray-tracing* effects. In the following, we discuss the most relevant work in these two well-studied fields, particularly those targeting the acceleration of both ray-tracing and rasterization techniques.

Ray tracing data structures and acceleration. Extensive research has been performed with the aim of accelerating the computation of intersections of rays with the scene. The commonly used acceleration data structures include uniform grids, octrees, kd-trees, and bounding-volume hierarchies (see established surveys for more details [WMG*09, HH11]). One of the keys to efficiency is the quality of the acceleration data structure, which, for the case of hierarchies, is usually constructed according to the Surface Area Heuristics (SAH) [GS87]. Related to our approach are the methods based on batched processing of rays, such as cone tracing [Ama84], beam tracing [HH84, LSLS09], or more generally the stream-ray architecture [RGD09]. Mora [Mor11] proposed a method which avoids organizing the scene in a spatial data structure, but instead sorts large groups of rays together with the scene geometry on the fly. The method of Bolous et al. [BWB08] uses coarse-grained visibility tests to reduce the active ray set for CPU packet tracing, which have a similar purpose as the hardware occlusion queries used by CHC+RT. While our method shares the idea of packet tracing, it differs particularly in the fact that it is designed for integration with GPU-based rasterization and does not use explicit ray bounding primitives or other per-packet information.

Recent advances in GPU programming make it possible to do real-time ray tracing on the GPU [AL09, AK10, PBD*10, ALK12]. While these methods are very fast, they usually require that the scene and the associated acceleration data structure is fully available in GPU memory, which makes it difficult to handle large scenes. Our technique, in contrast, naturally leads to more coherent data access patterns and to batch-based memory management.

Mixing ray tracing and rasterization. Several algorithms have tried to use the limited features of rasterization-based rendering for ray tracing. Most notably, Carr et al. [CHH02] proposed the Ray Engine, which achieves ray tracing effects by rendering a screen-sized quad and computing ray intersections for each scene triangle. The brute-force version of this process is inefficient and uses huge amounts of fill rate. Roger et al. [RAH07] improves on this method by building a hierarchy of cones over the rays and using them to reduce the number of computed intersections. In our algorithm, we conservatively cull those pairs of triangle batches and screen-space patches where the geometry is not intersected with respect to the screen-space patch. Novak and Dachsbacher [ND12] use rasterization to construct a hierarchy containing resampled scene geometry that can be processed by standard ray tracing methods. Davidovic et al. [DEG*12] proposed a 3D rasterization method designed for coherent rays. The authors show that there exists no fundamental difference between rasterization and ray tracing of primary rays, but a continuum of approaches that blend seamlessly between both paradigms. Our algorithm further explores the space between both paradigms by using the fixed-function pipeline and the z-buffer for arbitrary rays. Recently, Zirr et al. [ZRD14] proposed a method for ray tracing in a rasterization pipeline, using a voxel scene approximation to accelerate the traversal. A voxel representation is also used by Hu et al. [HHZ*14], using the A-buffer to search ray-triangle intersections in a shader. In contrast to these methods, we support casting arbitrary rays and out-of-core rendering.

Out-of-core ray tracing. Most of the work on rendering large scenes has focused on combining CPU techniques with out-of-core data-management methods (see a survey on massive-model rendering [GKY08]). Notable examples are methods using a scheduling grid for rays to improve the coherence of scene accesses (e.g., [PKGH97, MBK*10]) and methods exploiting level-of-detail representations [CLF*03, LYTM08, Áfr12]. More recent work also combined CPU/GPU computation using distributed computing approaches [BBS*09, KSY14]. In this context, Pantaleoni

et al. [PFHA10] proposed the PantaRay system, targeted at fast relighting of complex scenes based on occlusion caching. Garanzha et al. [GBPG11] used a complex data structure similar to PantaRay for CentiLeo, a commercial progressive out-of-core path tracer based on CUDA. Instead, our method subdivides the scene into adaptively sized batches of visible geometry by using occlusion culling, allowing simpler and more flexible data management that yields a natural out-of-core extension.

Visibility and rasterization methods. View-frustum and occlusion culling methods are commonly used to rasterize only the visible part of the scene and thus to make rendering output sensitive. In particular, hardware occlusion queries can be used to efficiently test the visibility of simple proxy objects, such as bounding boxes, against the depth buffer before rendering the real geometry [SBM03, BWPP04, GM05, GBK06, MBW08]. A general technique commonly used to compute complex effects in the rasterization pipeline is deferred shading, generalized by Saito and Takahashi [ST90], and used in several methods discussed above. Our novel algorithm also exploits this technique and generalizes the described culling methods by handling arbitrary primary and secondary rays with occlusion queries.

3. Overview

Figure 2 provides an overview of our method. We first render the scene either by rasterization or tracing primary rays. Then the method applies a number of additional ray tracing-based shading passes, which add the required illumination effects to the rendered image. In each pass, we first generate the rays to be cast and store them in a (full-screen) ray texture with one ray per pixel. Thus, the rays are directly associated with the pixels they should contribute to. We generate 3 ray types in this phase: soft-shadow rays, ambient-occlusion rays, and diffuse rays. Note that while a ray is stored in the pixel it will finally contribute to, it could start anywhere in the scene. Each pass uses one ray texture, and thus evaluates one ray contributing to the pixel.

The core part of our method is computing ray-triangle intersections in the rasterization pipeline using the given ray texture and a CPU-side scene hierarchy. The hierarchy can be a Bounding Volume Hierarchy (BVH) used by the CPU to perform view-frustum and occlusion culling. The basic operation that we use is determining whether rays corresponding to a certain screen-space tile intersect the bounding box of the given node of the BVH. We call this pair (the screen-space tile and the BVH node) a *query pair*. The algorithm starts with the query pair given by the tile representing the whole screen (all rays) and the bounding box corresponding to the root of the BVH (all triangles). The potential intersection of rays and the bounding box is evaluated in a shader that computes the nearest intersection of each ray with the given box. This distance is passed as a z-value to be compared with the already evaluated nearest distance using the hardware

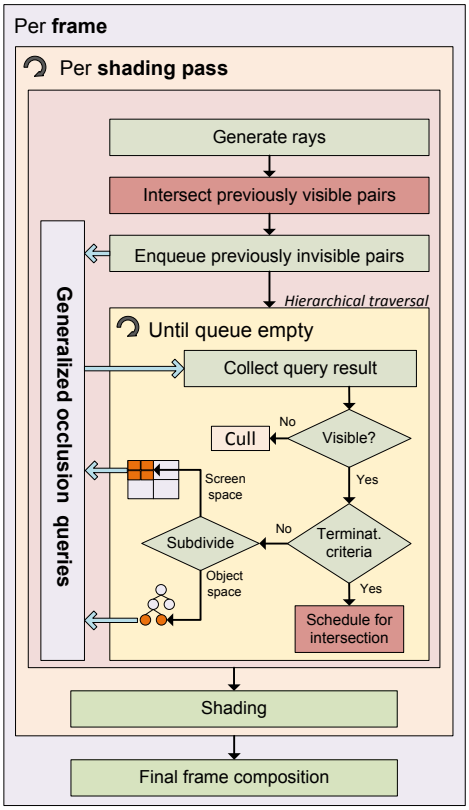


Figure 2: *Overview of the proposed algorithm, CHC+RT.*

z-buffer. We detect the rays intersecting the box by issuing an occlusion query that encapsulates the query pair processing. If the result of the occlusion query indicates a non-zero number of intersections, we either proceed by subdividing the screen-space tile or the BVH node and repeating the process for the newly created query pairs. This hierarchical traversal is indicated by the yellow box in Figure 2. The subdivision is terminated when reaching tiles of a certain minimum size and when meeting a termination criterion of the BVH. Then the actual ray-triangle intersections are computed.

We exploit temporal coherence by maintaining generalized visible and invisible fronts for the current ray set (stored as *previously visible pairs* and *previously invisible pairs*, as shown in Figure 2). Using this method we can reduce the number of intersection tests and also eliminate stalls caused by the latency occlusion queries. One key to the efficiency of our method is that the occlusion-culling phase computes a relatively coarse-grained cut in the query-pair hierarchy. We chose to use GLSL shader-based traversal to compute the fine-grained ray-triangle intersections of each visible subtree of the BVH. GLSL is very efficient in rendering small subtrees due to the high cache coherence of the traversal stack. The parts where query pairs are scheduled for intersection are shown in red in Figure 2. While our method is conceptually

similar to hierarchical occlusion culling for rasterization, the main difference is that the occlusion queries are generalized to arbitrary rays, and that we also maintain a hierarchy over screen space to localize the ray contributions.

4. Hierarchical Occlusion Culling for Ray Tracing

This section describes algorithmic and implementation details of the proposed method. We first describe the main components of the algorithm. Then we describe an optimized version of the method using temporal coherence.

4.1. Generalized Occlusion Queries

In our method, we use occlusion queries to cull those sets of rays and triangles that cannot intersect. The occlusion queries used in our method can be seen as a generalization of classical hardware occlusion queries [BWPP04]. Traditionally, occlusion queries handle visibility from the camera, and thus they deal with a well-defined set of primary rays enclosed in the viewing frustum. In ray tracing, we deal with arbitrarily distributed rays, and thus we have to be able to determine which rays intersect the given geometry using some other means than simple projection of the geometry and its rasterization. Similar to classical rasterization, we use a depth buffer to store the nearest intersection of each ray with the part of the scene processed so far (recall that rays are associated with pixels). We subdivide the screen into tiles corresponding to packets of rays. For each tile, we use an occlusion query to check if the bounding volume of an object intersects the rays enclosed by the tile. If there is at least one intersection, the query returns a non-zero value and we proceed by calculating the actual ray-triangle intersections. This step can be easily evaluated using a fragment shader in which we pass the bounding volume (axis-aligned box) as a shader parameter. The shader evaluates the ray/box intersection and returns the distance of the intersection as the depth value of the fragment. The query can thus count the number of fragments having nearer intersections than those stored in the z-buffer so far. So the main difference to classical occlusion queries is that the z-buffer values do not represent camera depth values, but distances along rays.

4.2. Shader-based Ray-Triangle Intersection

The visibility in the occlusion-query stage corresponds to a coarse cut in the BVH, making the method less sensitive to spatially incoherent ray packets. Once we reach a termination node in the BVH, the subtree is subsequently scheduled for intersection. The actual ray-triangle intersections are computed in a fragment shader executed for a given screen-space tile. The geometry (triangles) is stored in a texture buffer object and passed as a shader parameter. The actual shading is deferred to the moment when all scene geometry is processed and the final nearest intersections have been determined.

4.3. Hierarchical Occlusion Culling

The two above-described principles (generalized occlusion queries and shader-based ray triangle intersection) can be used together in an algorithm which processes both the rays and the triangles hierarchically. The hierarchy of rays is defined implicitly by a quadtree-based screen-space subdivision, the hierarchy of triangles is defined by a bounding-volume hierarchy (BVH). We propose a generalization of hierarchical occlusion culling, with the main difference that the query objects are not BVH nodes, but *query-pairs* consisting of a BVH-node and a screen-space tile.

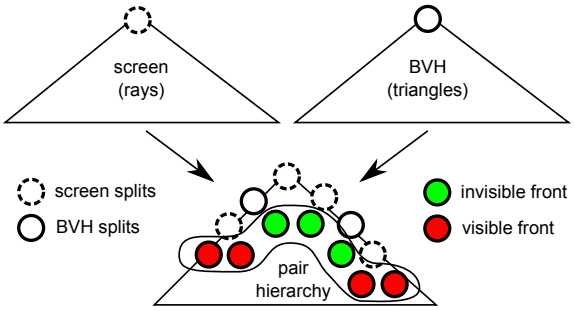


Figure 3: *Illustration of the query-pair hierarchy. The interior nodes of the hierarchy correspond to either screen-space splits or object-space splits. The leaf nodes belong to either the visible front (rays and triangles that can intersect) or the invisible front (rays and triangles that cannot intersect).*

4.4. Traversing the Query-Pair Hierarchy

When the result of the query indicates an intersection, we have to subdivide the query pair and construct new query pairs to refine the intersection results. Here, we have to decide between two choices – subdividing in screen space, and creating 4 new query pairs, or subdividing in object space, and creating two new query pairs (see Figure 3). This decision influences in how many steps a particular subtree of the query-pair hierarchy can be culled as being invisible, and hence it is important for the performance of the traversal algorithm. A split in object space can potentially reduce the *intersection cost*, while a split in screen space can potentially reduce the area and hence the *query cost*. Note that a split in object space can potentially double the overdraw and thus the query cost since both child nodes have to be queried for the same tile. We found that the best performance can be achieved by comparing the areas of the screen-space tile and the object-space node of a pair, which are connected to the query cost and intersection probability, respectively [GS87]. The areas are normalized by the area of the bounding box of the BVH root (A_{root}) and the full screen extent (A_{screen}). They are also weighted by a hardware-dependent factor t_q, which we set to 0.5 in all our comparisons (favoring object-space splits in the beginning). We always split in the domain where

the corresponding ratio is larger, i.e.:

$$\frac{A_{bvh}}{A_{root}} > t_q * \frac{A_{tile}}{A_{screen}} \begin{cases} \text{true:} & \text{split in object space} \\ \text{false:} & \text{split in screen space} \end{cases}$$

This heuristic aims to keep a rough balance between the extents of the screen-space and object-space domain within a query pair. Note that it would be more consistent to compare both areas in world space, but until we compute the intersections we do not know the world-space extent of the bounding volume of rays covered by a screen-space tile.

4.5. Exploiting Temporal Coherence

The hierarchical algorithm described above can be improved by exploiting temporal coherence among rendered frames. In particular, similar to occlusion-culling algorithms, we can initialize the content of the depth buffer by first evaluating intersections using all visible query pairs from the previous frame. Note that this assumption does not invalidate the correctness of the results since the actual ray-triangle intersections always use data for the current frame, i.e., rays generated for the current frame and triangles at correct positions for the current frame. After processing previously visible pairs, we issue queries on previously invisible pairs to verify if they stay invisible. If any previously invisible pair becomes visible, we process it hierarchically and collect all newly visible pairs for which ray-triangle intersections should be computed. At the end of the frame, these new intersections are evaluated, and finally the visibility front consisting of both visible and invisible pairs is updated.

5. CHC+RT

In this section we address the details regarding the actual OpenGL implementation of the method and its optimizations.

5.1. Hierarchical Traversal

The pseudocode of our traversal algorithm is shown in Algorithm 1. In analogy to occlusion culling [BWPP04, MBW08], we talk about visible/invisible nodes. For a node in the query-pair hierarchy, this means that the occlusion-query result is either positive (there are potential intersections, hence visible) or zero (there are no intersections, hence invisible). Most optimizations proposed in the CHC++ algorithm [MBW08] can also be used for CHC+RT for reducing the overhead of generalized occlusion queries. Our algorithm starts from the previous cut of visible leaf nodes and invisible (leaf or interior) nodes. It consists of three phases.

Phase 1. All previously visible leaf nodes are scheduled for ray-triangle intersection. This initializes the z-buffer with the intersections from those query pairs which have been visible in the previous frame and allows us to exploit ray occlusions.

Phase 2. The visibility status is queried for the previously visible and invisible pairs. First, all the previously invisible nodes are queried (line 4) and enqueued in the so-called *query queue*. Since the visibility status of the previously visible nodes could have changed from the previous frame, we lazily query them and update their visibility status at the end of the frame. Analogous to CHC++, we use a stratisfied sampling scheme by randomizing the first frame where the node is queried (between $1..n$ frames). Thereafter, the node is queried every n frames. In our tests, we set n to 3.

Phase 3. The actual hierarchical traversal is where newly visible pairs are detected and the invisible front is updated for the next frame. It starts by fetching the query results one at a time: If the visibility did not change from the previous frame, we are finished. If a node turned visible, we further subdivide the node and enqueue the child nodes, until either a node is found to be invisible or a termination criterion is met (in which case we set it to visible). During the traversal, whenever we have compiled more than m visible nodes (where $m = 16$ in our tests), we compute the ray-triangle intersections of the nodes found newly visible (line 22).

Each frame, invisibility information is pulled up in the hierarchy. This means, if all child nodes are invisible, the parent node is set to invisible and the child nodes can be deleted. This can be continued recursively until we encounter a visible child node, but we restricted it to at most one level per frame to avoid fluctuations. Note that for this purpose we maintain a query-pair hierarchy in order to recall the history of the subdivision and quickly determine which pairs to merge during the pull-up phase.

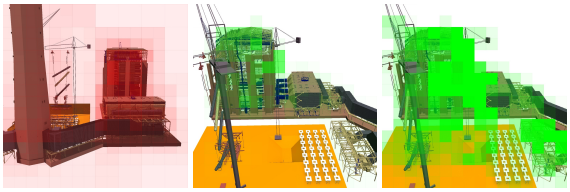


Figure 4: *Left: Visualization of the occlusion-query overdraw for primary rays, where reddish regions have high overdraw. Middle and right: Illustration of the reduced spatial coherence using diffuse rays with length* 10 *and* 1000*, respectively. An opaque green screen-space tile means that its rays potentially intersect* $> 2M$ *triangles.*

A useful optimization for previously visible node queries (line 6) are the so called *multi-queries* [MBW08]. Their purpose is to reduce the overdraw caused by queries overlapping in screen space, which we identified as the main source of performance overhead (see Figure 4-left). Multi-queries compile many previously invisible pairs projecting to the same screen-space tile into a single occlusion query over multiple bounding boxes. If this query is successful and all nodes stay invisible, many nodes have been handled in a single shader pass, which is more efficient than using a separate pass for each individual node. Note that in case the query fails (line 14), multi-queries have to be handled differently. In particu-

lar, all nodes have to be queried individually, since we don't know which of the nodes has become visible. We also exploit the tighter-bounds optimization of CHC++ by always querying the bounding boxes of the two children of a BVH node instead of the node itself.

```
   // Phase 1: intersect visible pairs
 1 sort previously visible pairs by tile
 2 for all previously visible pairs do
 3 |  compute ray-triangle intersections;
 4 end
   // Phase 2: query pairs
 5 sort previously invisible pairs by tile
 6 for all previously invisible pairs do
 7 |  compile (multi-)query and enqueue;
 8 end
 9 for previously visible leaves do
10 |  update visibility status every n frames;
11 end
   // Phase 3: recursive traversal
12 while not query queue empty do
13 |  fetch next query result;
14 |  if query result == visible then
15 |  |  if terminationReached(query pair) then
16 |  |  |  add node to newly visible leaves;
17 |  |  |  if newly visible leaves > m then
18 |  |  |  |  for newly visible leaves do
19 |  |  |  |  |  compute ray-triangle intersections;
20 |  |  |  |  |  clear newly visible leaves;
21 |  |  |  |  end
22 |  |  |  end
23 |  |  end
24 |  |  else
25 |  |  |  subdivide(query pair);
26 |  |  |  for all children do
27 |  |  |  |  issue occlusion query and enqueue;
28 |  |  |  end
29 |  |  end
30 |  end
31 end
   // intersect remaining visible leaves
32 for newly visible leaves do
33 |  compute ray-triangle intersections;
34 end
```

Algorithm 1: Traversal

5.2. Ray-Triangle Intersections

The ray-triangle intersection is the last stage of our algorithm. As we compute ray-triangle intersections for localized subsets of our scene geometry, we can achieve good data-access coherence and can employ streamlined acceleration data structures. The efficiency of the implementation of the proposed method greatly depends on the CPU/GPU data management, i.e., the way in which we pass the shader data, the actually used intersection algorithm, and how we organize the rendering calls of the shader kernels.

GLSL shader. We chose to use a GLSL shader-based traversal algorithm for the final intersections of the termination nodes in the BVH. The shader uses the speculative while-while ray traversal proposed by Aila and Laine [AL09] for CUDA-based ray tracing. The traversal always fetches both child nodes and traverses the nearer child first in order to exploit occlusion. We cache the intersected leaves for delayed coherent ray-triangle intersection. We observed that the optimal value for this leaf cache was 2 on our hardware. The cache size is small but nevertheless crucial, as omitting the cache causes a slowdown by approximately 30%.

CPU-GPU transfer. We pass the geometry to the shader using texture buffer objects in GLSL. For the in-core version, we simply allocate two texture buffers: one for the BVH and one for the geometry. Since we currently only allow diffuse materials, we store a diffuse color per triangle in the alpha channel of the RGBA texture used for storing the geometry.

Figure 5: *Left: Visualization of the BVH subtrees that will be scheduled for intersection in the fragment shader. Right: Visualization of the number of subtrees compiled in each batch (1 (red) – 24 (white)) for per-tile based batching.*

Termination criteria. We use 3 different termination criteria for the traversal of the query-pair hierarchy. One is connected to the termination in the screen-space hierarchy, the other two to the termination in the BVH hierarchy. In our experiments, the optimal size of a screen-space tile seemed independent of the chosen resolution of the actual render target. In our case, we set the minimum tile size to 200^2 pixels in all experiments, meaning that each tile covers less than 2% of the screen at Full-HD resolution. The key termination criterion in the BVH hierarchy turned out to be the maximum subtree height (i.e., the number of traversal steps until the farthest of the leaf nodes can be reached). The reason is that the GLSL shader-based traversal is very sensitive to the maximum stack size, which has to be at least as large as the maximum subtree height. The optimal granularity of the subtrees depends on various factors like the presence of occlusion. In our experiments, we found that setting the maximum subtree height to 24 levels works well in many cases. Figure 5 shows the subtrees induced by our termination parameters. Another termination criterion is the maximum number of triangles per subtree, which becomes important in the out-of-core scenarios. We set it to $1M$ triangles in all our tests.

By-Tile sorting. In our experiments it turned out to be inefficient to schedule the visible subtree nodes of the BVH separately for ray-triangle intersection. Instead, the GPU is better utilized if the contribution of several subtrees to the same screen-space tile is computed in a single shader call. For this purpose we sort the nodes scheduled for intersection by screen-space tile (line 2 in Algorithm 1) and then pass an array with the maximum number of 24 node ids to the shader, together with their bounding boxes. The shader then tests the bounding boxes for intersection and starts the traversal for all nodes that pass the intersection test. A visualization of this method is shown in Figure 5. The right image visualizes the number of subtrees that can be handled in a single shader pass, and how this number increases with distance. Apart from the better shader utilization, another benefit of this approach is that we can better exploit occlusion within the shader. For certain ray types like shadow or primary rays, this approach can be optimized further by passing the nodes in an approximate front-to-back order.

5.3. Ray Generation and Scheduling

In each render pass we generate a single ray direction for primary rays as well as shadow, ambient occlusion, and diffuse rays. The performance of our method benefits from *temporal coherence* and to a lesser degree *spatial coherence*. Less spatial coherence leads to less efficient pruning of invisible subtrees, as shown in Figure 4-right. We take this into account already during ray generation. An alternative possibility would have been to use ray sorting on the generated rays.

Spatial coherence. For both ambient occlusion (or diffuse rays, respectively) and shadow rays, we generate the samples in a stratified fashion. Using a Halton sequence, the same ray direction is generated for each pixel and perturbed with a random per-pixel offset. The degree of randomization depends on the number of rays shot. To achieve this for ambient occlusion and diffuse rays, we apply a random per-pixel rotation to the ray in tangent space, as proposed by Mittrig et al. for SSAO [Mit07]. The maximum angle is chosen so that the samples can cover the whole hemisphere.

Temporal coherence. For primary and shadow rays, ray directions are usually sufficiently coherent so that we maintain a single visibility status for all shadow rays. For ambient occlusion rays and diffuse rays, the ray directions exhibit more variation. We can nevertheless enforce temporal coherence by using a separate visibility status per ray direction, which contains all query pairs in the visibility front. Since we use a coarse hierarchy in screen space and object space, it is easy to keep track of many such cuts.

5.4. Out-of-Core Ray Tracing

Our method naturally allows for out-of-core ray tracing, with the possibility of rendering potentially unbounded scenes. This is difficult to achieve with current GPU ray-tracing architectures. In the best case, we assume that (most of) the working set required for computing a given frame fits in GPU memory, while the entire scene does not. By using a cache of recently used termination nodes on the GPU, we can avoid transferring to the GPU the geometry that is already in the cache. Note that this sort of memory management requires only minor modifications to the method shown in Algorithm 1, and can be managed inside the *compute ray-triangle intersections* function. In our current implementation, we use a simple round-robin style cache management for the BVH and geometry data of the visible subtrees. When caching an out-of-core node, the data is simply written in the next free slot in the texture buffer. When the end of the buffer is reached, we start overwriting the data from the beginning and mark the overwritten nodes as out-of-core. As the only extension to the core algorithm shown in Algorithm 5.2, we sort geometry that is scheduled for intersection by their out-of-core status, i.e., visible nodes that have their data currently cached on the GPU are scheduled for intersection first. Note that we still use per-tile sorting among cached nodes.

6. Discussion

At the core of our technique is a novel scheduler and memory manager for fine-grained ray-tracing computations that exploits coarse-grained hierarchies in both object space and screen space. The screen-space hierarchy significantly improves scheduling and speeds up rendering. E.g., for the Powerplant scene it results in a speedup by a factor of 3 for Full-HD. Moreover, a screen-space hierarchy makes the method well suited to the current trend towards larger resolution displays (4K and above). By working at a coarse grain, we can amortize the cost of taking decisions over a large number of ray-primitive intersection queries, and use an efficient and flexible adaptive-loading architecture working on optimized geometry batches.

Analysis of problem-domain pruning. A good insight into the principle of the method and its potential strengths and weaknesses can be obtained by analyzing the coverage of the whole ray-triangle intersection domain by the query pairs. For this analysis, we express this domain using a matrix in which each ray corresponds to a row in the matrix, while each triangle corresponds to a matrix column. When computing the nearest intersections of rays and triangles, there will be a single unique intersection in each row of the matrix, while there can be many intersections for each column (a triangle can define a nearest intersection for many rays). Our query pairs need to be constructed in a way that every potential intersection is correctly determined. In other words, the query pairs have to fully cover the whole matrix. We can observe an example of such a matrix and its coverage by query pairs in Figure 6. This matrix is generally very sparse. The coverage of the matrix by query pairs depends on two main factors: (1) the coherence of intersections and (2) on how densely the

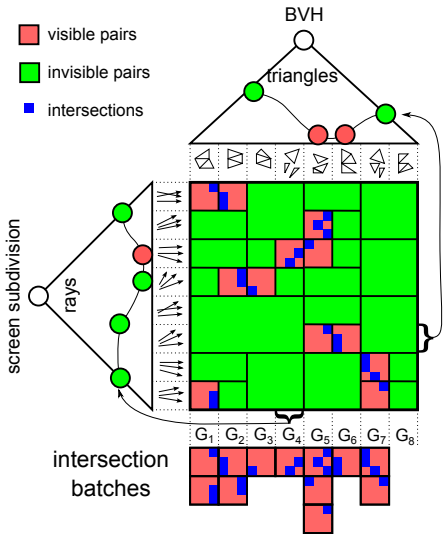


Figure 6: *Illustration of the coverage of the whole domain of ray-triangle intersection by constructed pairs. Pairs indicating potential ray-triangle intersections are shown in red, while the pairs for which the geometry bounding boxes do not intersect the rays are shown in green. The actual ray-triangle intersections are shown in blue. The figure shows batches of visible pairs (G_1-G_8) used for computing ray-triangle intersections. Note that the geometry G_8 is not used in any batch, meaning that it is not intersected by any ray and will not be scheduled for intersection. Note the two example cuts in the hierarchies: the cut in the screen-space subdivision shows query pairs corresponding to a BVH termination node, and the cut in the BVH shows query pairs corresponding to a given screen-space tile.*

rays sample the scene. The triangles are sorted in the BVH and therefore, for similar rays the intersections should cluster around similar triangles, creating a compact intersection cluster which can be covered by a few query pairs. However, if the rays are highly incoherent, the coverage by query pairs will become more complex. Note that even then, the matrix will be sparse, and at some point we will be able to prune most of the intersection domain if enough query pairs are used.

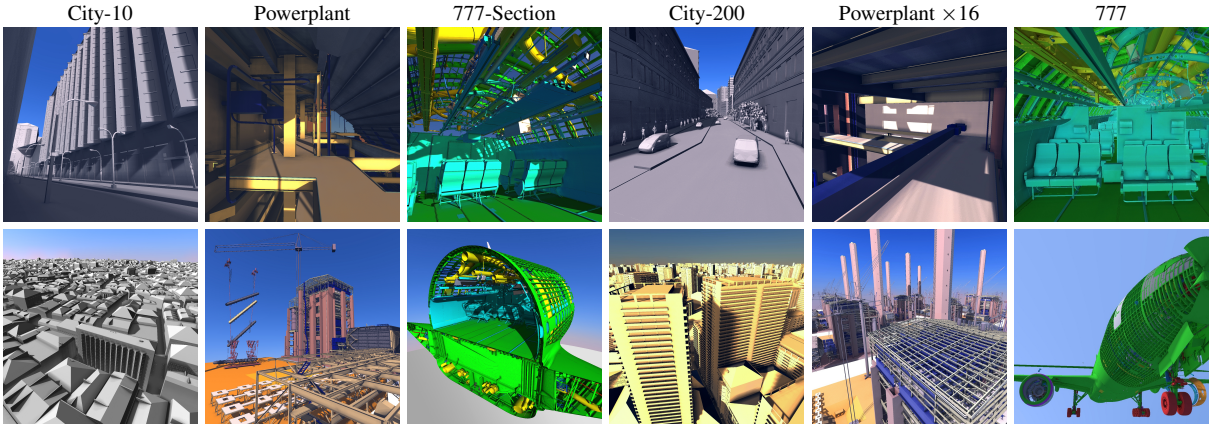
Another interesting observation follows from the analysis of rows and columns of the matrix. In particular, the number of query pairs covering a row of the matrix directly corresponds to the overdraw of the corresponding screen pixel. The visible query pairs (shown in red) will cause ray/triangle intersections to be executed for this pixel and passed to the z-buffer, while the invisible query pairs (green) will execute the ray/box intersection verifying the invisibility of the associated geometry by the given ray. Looking at the columns of the matrix, we can observe the number of query pairs (screen-space tiles) needed to handle the given geometry (triangle batch). The visible query pairs (red) are those for which the actual ray-triangle intersections are computed, while the invisible query pairs show the occlusion queries issued for rays which do not intersect the triangle batch. Note that the size of the rectangles shown in the matrix depends on the depth of the corresponding query pair in the two hierarchies. Thus the coverage of the matrix by query pairs also visualizes the double-hierarchical cut on which our algorithm operates.

GLSL rendering. Our method is not necessarily bound to a specific implementation, and a CUDA version of the algorithm is definitely possible. Nonetheless, focusing on GLSL in this paper provides specific advantages. First of all, by explicitly using a rasterization platform, we better convey the underlying idea that there is a continuum between rasterization and ray-tracing approaches. We aim to foster further research in the area of hybrid rendering by showing how techniques from the rasterization world, such as coherently scheduled visibility algorithms, batched computation and out-of-core rendering, can successfully improve ray tracing. Second, GLSL simplifies the implementation through features of the fixed-function pipeline. For instance, we can use automated shader scheduling instead of implementing explicit schemes, and, while Z-buffering and visibility queries can be realized in CUDA, using GLSL avoids the need to craft efficient synchronization methods using atomic operations. Finally, a GLSL implementation has the additional benefit to be less hardware dependent with respect to CUDA and to simplify integration into classic OpenGL rendering pipelines.

Limitations. Our framework currently only supports static scenes, but an extension to fully dynamic scenes would be possible without changes to the core of the algorithm. While different ray types are supported by our method and we present techniques for enforcing more coherence, it is still true that the method becomes less efficient for fully incoherent ray patterns. We focus on the overall method and its capability of handling large scenes using single-bounce illumination. For simple multi-bounce illumination, e.g., Whitted-style ray tracing, there is enough coherence even for secondary bounces, and our method can use a separate hierarchy cut for each such bounce. For a full path-tracing solution, our plan is to have a sorting step after each rendering pass that would reorder rays in a more coherent order using a space-filling curve based on scene hit points, along the lines of Moon et al. [MBK*10]. Other authors have already done this sorting on the GPU [GL10], so we are confident that real-time performance is possible. This pass would generate a linear order, and our screen-space hierarchy will become a ray-space hierarchy built on reordered rays.

7. Results

For our experiments we use an Intel i7-3770 CPU with 3.5GHz (using one core), 8 GB RAM, a resolution of 1920×1080 and an NVIDIA Titan GPU with 6 GB of video memory. However, as there seems to be a limitation at 4 GB for use with a single thread, we are only able to allocate a



	City-10	Powerplant	777-Section	City-200	Powerplant ×16	777
Triangles	11.7M	12.8M	21.5M	139M	205M	350M
Total size	677MB	710MB	1.19GB	7.89GB	11.4GB	18.9GB
%In-core	100%	100%	100%	**47%**	**32%**	**20%**
Bvh/Geo	140/537MB	126/584MB	244/984MB	1.54/6.35GB	2.02/9.34GB	2.80/16.1GB

Table 1: *Used models showing near-view and far-view.*

Scene	Ray type	Near-View (ms) Prim		AO		Diff		Far-View (ms) Prim		AO		Diff	
City-10	GLSL	18.22	(1.00×)	186	(1.00×)	493	(1.00×)	26.3	(1.00×)	**277**	(**1.00**×)	586	(1.00×)
	CHC+RT	**11.3**	(**1.61**×)	**153**	(**1.22**×)	**384**	(**1.28**×)	23.4	(1.12×)	286	(0.97×)	655	(0.89×)
	CUDA	12.5	(1.46×)	311	(0.60×)	615	(0.80×)	**14.9**	(1.77×)	278	(1.00×)	**422**	(**1.39**×)
Powerplant	GLSL	69.9	(1.00×)	772	(1.00×)	10064	(1.00×)	82.5	(1.00×)	588	(1.00×)	2722	(1.00×)
	CHC+RT	12.8	(5.46×)	**173**	(**4.46**×)	1700	(5.92×)	15.0	(5.50×)	**154**	(**3.82**×)	843	(3.23×)
	CUDA	**11.8**	(**5.92**×)	310	(2.49×)	**1152**	(**8.74**×)	**10.6**	(**7.78**×)	246	(2.39×)	**501**	(**5.43**×)
777-Section	GLSL	78.1	(1.00×)	516	(1.00×)	5464	(1.00×)	108	(1.00×)	781	(1.00×)	3752	(1.00×)
	CHC+RT	24.2	(3.23×)	**236**	(**2.19**×)	2729	(2.00×)	29.8	(3.62×)	**270**	(**2.89**×)	2382	(1.58×)
	CUDA	**12.5**	(**6.25**×)	277	(1.86×)	**1264**	(**4.32**×)	**16.4**	(**6.59**×)	316	(2.47×)	**809**	(**4.64**×)
City-200	CHC+RT	**25.9**	(-)	**325.7**	(-)	**2720**	(-)	**87.8**	(-)	**766.7**	(-)	**2492**	(-)
Powerplant×16	CHC+RT	**18.7**	(-)	**232**	(-)	**1855**	(-)	**270**	(-)	**1559**	(-)	**25975**	(-)
777	CHC+RT	**134**	(-)	**5175**	(-)	**30134**	(-)	**333**	(-)	**1961**	(-)	**16441**	(-)

Table 2: *Comparison of our method (CHC+RT) with shader-based ray tracing (GLSL) and CUDA-based ray tracing [ALK12] (CUDA) using a resolution of* 1080*p. We trace either primary rays or* 20 *samples per pixel of secondary rays. The numbers in bold identify the best method in terms of the overall frame time. The numbers in parenthesis show the speedup with respect to GLSL.*

maximum of 3.7 GB for the data (BVH, geometry, materials) of our out-of-core scenes.

Table 1 shows the models used in our experiments. We use 3 scenes for in-core ray tracing and 3 out-of-core scenes. City-10 and City-200 are a typical 2.5*D* city models, generated with the City Engine [MWH*06] in two levels of detail. They offer a high degree of occlusion for near views and a high degree of regularity. The Powerplant model is considerably less regular, and the created BVH is deep. For out-of-core ray tracing, we use 16 copies of the Powerplant. The complex 777 model is a standard scene for testing out-of-core methods. We also extracted a section of the 777 model for in-core use. Table 2 shows numerical results for our benchmarks using the described technique. In each scene we provide 2 walkthroughs roughly corresponding to a sequence of near-view points (e.g., on street level) and far-view points (e.g., bird's-eye view). We test the proposed method for primary rays, 20 short ambient-occlusion rays (0.5 units long) which sample the hemisphere, and 20 diffuse reflection rays where the maximum ray length is set to cover the full scene extent. These ray types cover many cases typically encountered in ray-tracing applications. Our method uses all optimizations described in Section 4.5 in order to fully exploit temporal and spatial coherence. The BVH in our tests is constructed on the CPU using SAH and optimized using the insertion-based BVH optimization [BHH13]. We compare our method against standalone GLSL shading-based traversal without occlusion culling (simply traversing the BVH from the root node) and the state-of-the-art CUDA ray tracer of Aila et al. [ALK12]. Note that we disabled ray sorting in CUDA tracing since the overhead significantly outperformed the gain in traversal

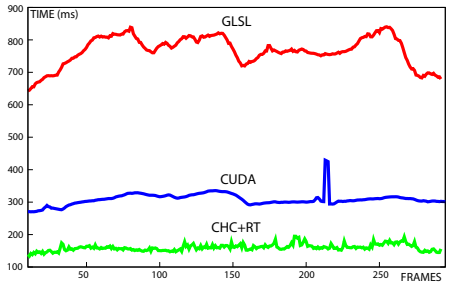


Figure 7: *Comparison of CHC+RT with GLSL and CUDA in Powerplant (near-view) for ambient-occlusion rays.*

time, while for our method the rays are already generated in a more coherent fashion.

CHC+RT usually works best for occluded walkthroughs (near views in City scenes and Powerplant). For the City near view, it is faster than both GLSL and CUDA tracing for all ray types. In the Powerplant model, CHC+RT is significantly faster than GLSL and mostly comparable with CUDA. This can also be observed in the frame-by-frame comparison shown in the plot of Figure 7. The sources of the speedup with respect to GLSL are that occlusion can be efficiently exploited in Powerplant, and that the deep BVH in Powerplant can be better handled by our method. In less occluded views, the overhead due to the occlusion queries can sometimes outweigh the benefit for CHC+RT (e.g., City far view). The 777 model is a challenging case for any rendering algorithm, and the 777 Section exhibits similar properties. Since parts of the hull have been removed, many complex details are visible most of the time. This is a good case for the CUDA ray tracer, which is indeed the best method for primary and diffuse rays. On the other hand, CHC+RT shows better overall frame times for ambient-occlusion rays due to the smaller setup time. Also note that CHC+RT is able to reduce the performance gap to CUDA in this scene by a large margin.

The CUDA ray tracer is generally faster in terms of pure traversal times than both GLSL and CHC+RT. But since a higher constant cost is involved in the setup of each frame for CUDA, GLSL is competitive for scene configurations where the ray traversal time is short (e.g., for AO rays and highly occluded scenes). We made the observation that GLSL is much more sensitive to the stack size than CUDA, and this becomes a bottleneck for deep hierarchies. On the other hand, CHC+RT does not suffer from this problem. Indeed, the scheduled subtrees have a bounded traversal height and hence the stack size can be bounded.

The method scales well to large, possibly out-of-core scenes if sufficient occlusion is available. The performance of the near view in City-200 is comparable to the near view in City-10 in spite of the over 12× larger scene and the out-of-core overhead, and similar to the performance in the 777 Section. The same is true for the near views in Powerplant

Figure 8: *Traversal-time comparison of CHC+RT with GLSL split into the different phases of the algorithm.*

Scene		Near-View			Far-View		
	Ray type	Prim	AO	Diff	Prim	AO	Diff
City-200	Mrays/sec	80.1	127	15.2	23.6	54.1	16.6
	Queries	822	13.6K	61.2K	2.46K	42.9K	117K
	Trans. MB	0	5.85	383	0.61	0.11	1.73
	BVH	0	3.00	94.9	1.96	0.30	2.04
	Geom	0	2.85	288	2.58	0.41	3.77
PP×16	Mrays/sec	111	179	22.4	7.68	26.6	1.60
	Queries	793	13.2K	32.3K	3.63K	61.7K	312K
	Trans. MB	0.11	0.03	0.55	170	43.6	2310
	BVH	0.04	0.01	0.13	85.8	34.7	663
	Geom	0.07	0.02	0.42	83.8	8.85	1647
777	Mrays/sec	15.4	8.01	1.38	6.23	21.1	2.52
	Queries	1.76K	123K	501K	6.31	85K	278K
	Trans. MB	11.7	419	1442	355	96.7	1073
	BVH	2.64	139	595	70.1	25.0	253
	Geom	9.14	281	847	285	71.7	820

Table 3: *Per-frame statistics for the out-of-core models.*

and Powerplant×16. As can be observed for the far view of Powerplant×16 and for 777, diffuse rays in open view-points in the large out-of-core scenes are quite challenging for our method, but can be improved using the aggressive version of our algorithm discussed below.

Figure 8 visualizes the timings of the different phases of the algorithm as listed in Algorithm 1. Phase 1 corresponds to the intersection of previously visible nodes, while Phase 2 and 3 correspond to the overhead caused by occlusion culling. Phase 2 evaluates the current visibility status using queries for previously visible and invisible nodes. Phase 3 traverses the hierarchy in response to a change in visibility. Interestingly, the time spent in Phase 3 relative to the other phases increases for the out-of-core scenes. The reason is that the overhead

		City-200		Powerplant×16		777	
Pixels	%Tile	Near	Far	Near	Far	Near	Far
0	0.000	2720	2492	1855	25975	30134	16441
20	0.005	780	2019	1682	9651	11544	4807
100	0.025	738	1751	1553	4727	8081	2712
200	0.050	715	1611	1489	3578	6804	2028

Table 4: *Timings for the aggressive version of CHC+RT using 20 diffuse reflection rays. %Tile shows the error in % of the termination size of the screen-space tiles.*

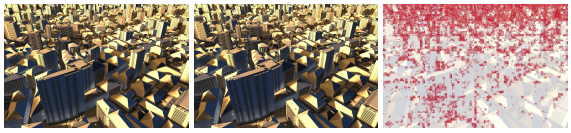


Figure 9: *Comparison of the conservative (left) with the aggressive version of our method (middle) allowing 20 pixels of error for diffuse rays. Right: Pixel differences are mostly in the background.*

of Phase 3 corresponds to changes in visibility. Even if the currently visible scene fits completely in-core and there are no node fetches during Phase 1, nodes that become newly visible will be uploaded to the GPU in this phase. Table 3 shows some interesting statistics for the out-of-core models. For all models, the primary ray rendering can be done predominantly in-core. Ambient occlusion and diffuse interreflections in particular require significantly larger transfer rates between CPU and GPU memory.

The proposed algorithm is conservative because the occlusion-query result (the number of visible pixels) is used for a binary decision. As an alternative, this number can be used for a simple LOD mechanism that culls all nodes whose contribution to a screen-space tile is less than a visible pixel threshold. As can be seen in Table 4, the aggressive algorithm is especially useful for reducing the computational complexity of diffuse reflections, where many nodes contribute to only a few pixels. As shown in Figure 9, allowing for example an error of 20 pixels per query can reduce the render time by a factor of 3 with only a minor decrease in accuracy, with a mean absolute pixel error of 9.04.

Table 5 shows the influence of spatial coherence during ray generation on the performance of our method. This is achieved by increasing the value for the maximum angle

Figure 10: *Effect of the random rotation on diffuse color bleeding using* 20 *samples for* 18° *(left),* 36° *(middle), and full randomization (right) (zoom in to see the differences).*

	Randomization	None	18°	36°	Full
AO	City-10 Near (ms)	136	145	156	193
	Powerplant Near (ms)	153	167	175	190
Diffuse	City-10 Near (ms)	228	323	384	805
	Powerplant Near (ms)	664	1334	1700	4125

Table 5: *Effect of the per-pixel random rotation of the diffuse sampling kernel on the coherence and frame time of 20 diffuse reflection rays in two selected models.*

for the random kernel rotation per-pixel. Diffuse reflections slow down by a factor of over 4–5× when going from no randomization to a fully randomized rotation, whereas the frame times for AO rays are affected much less. Note that the frame times using full randomization are still comparable to GLSL. The *temporal coherence* can be maintained by storing the visibility status for each ray direction. In our results we use a per-pixel rotation of 36° for 20 samples, which provides good quality and maintains a sufficient degree of spatial coherence (as shown in Figure 10). It also has the benefit to eliminate some temporal noise in moving frames.

8. Conclusion and Future Work

We presented a novel use of hierarchical occlusion culling for accelerating OpenGL-based ray tracing. Our method exploits the rasterization pipeline and hardware occlusion queries in order to create coherent batches of work for the GPU ray-tracing kernel. By generalizing occlusion culling to arbitrary rays through a combined hierarchy in both ray space and object space, we are able to share the intermediate traversal results among multiple rays, leading to a simple and efficient implicit parallelization using rasterization hardware. Through novel means for scheduling GLSL ray tracing kernels using the coarse-grained hierarchy over screen- and object-space, we are able to support rendering of out-of-core ray tracing using GPU memory as a cache. Our method narrows the gap between OpenGL-based ray tracing and CUDA ray tracing by a significant amount and is able to outperform CUDA ray tracing in some cases. In the future, we want to show the flexibility of our method on legacy hardware and other systems that support OpenGL (e.g., ATI GPUs).

Acknowledgments. This work is partially supported by the People Programme (Marie Curie Actions) of the European Union's Seventh Framework Programme FP7/2007-2013/ under REA grant n°290227 (DIVA), the Czech Science Foundation under research programs P202/11/1883 (Argie) and P202/12/2413 (Opalis), and the Czech Technical University in Prague grant n° SGS13/214/OHK3/3T/13. We also acknowledge the contribution of Sardinian Regional Authorities. We thank Fabio Marton for helpful comments and suggestions.

References

[AK10] AILA T., KARRAS T.: Architecture considerations for tracing incoherent rays. In *Proc. High-Performance Graphics* (2010), pp. 113–122. 2

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics* (2009), pp. 145–149. 2, 6

[ALK12] AILA T., LAINE S., KARRAS T.: *Understanding the Efficiency of Ray Traversal on GPUs – Kepler and Fermi Addendum*. Tech. Rep. NVR-2012-02, NVIDIA, June 2012. 2, 9

[Ama84] AMANATIDES J.: Ray tracing with cones. *SIGGRAPH Computer Graphics 18*, 3 (1984), 129–135. 2

[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. *Computer Graphics Forum 28*, 2 (2009), 385–396. 2

[BHH13] BITTNER J., HAPALA M., HAVRAN V.: Fast insertion-based optimization of bounding volume hierarchies. *Computer Graphics Forum 32*, 1 (2013), 85–100. 9

[BWB08] BOULOS S., WALD I., BENTHIN C.: Adaptive ray packet reordering. In *IEEE Symposium on Interactive Ray Tracing* (2008), pp. 131–138. 2

[BWPP04] BITTNER J., WIMMER M., PIRINGER H., PURGATHOFER W.: Coherent hierarchical culling: Hardware occlusion queries made useful. *Computer Graphics Forum 23*, 3 (2004), 615–624. 3, 4, 5

[CHH02] CARR N. A., HALL J. D., HART J. C.: The ray engine. In *Proc. ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (2002), pp. 37–46. 2

[CLF*03] 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 (2003), 543–543. 2

[DEG*12] DAVIDOVIC T., ENGELHARDT T., GEORGIEV I., SLUSALLEK P., DACHSBACHER C.: 3D rasterization: a bridge between rasterization and ray casting. In *Proc. Graphics Interface* (2012), pp. 201–208. 2

[Áfr12] ÁFRA A. T.: Interactive ray tracing of large models using voxel hierarchies. *Computer Graphics Forum 31*, 1 (2012), 75–88. 2

[GBK06] GUTHE M., BALÁZS A., KLEIN R.: Near optimal hierarchical culling: Performance driven use of hardware occlusion queries. In *Proc. Eurographics Symposium on Rendering* (2006). 3

[GBPG11] GARANZHA K., BELY A., PREMOZE S., GALAKTIONOV V.: Out-of-core GPU ray tracing of complex scenes. In *SIGGRAPH Talks* (2011), pp. 21:1–21:1. 3

[GKY08] GOBBETTI E., KASIK D., YOON S.-E.: Technical strategies for massive model visualization. In *Proc. ACM Symposium on Solid and Physical Modeling* (2008), pp. 405–415. 2

[GL10] GARANZHA K., LOOP C.: Fast ray sorting and breadth-first packet traversal for GPU ray tracing. *Computer Graphics Forum 29*, 2 (2010), 289–298. 8

[GM05] GOBBETTI E., MARTON F.: Far voxels: A multiresolution framework for interactive rendering of huge complex 3D models on commodity graphics platforms. *Transactions on Graphics 24*, 3 (2005), 878–885. 3

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (1987), 14–20. 2, 4

[HH84] HECKBERT P. S., HANRAHAN P.: Beam tracing polygonal objects. *SIGGRAPH Computer Graphics 18*, 3 (1984), 119–127. 2

[HH11] HAPALA M., HAVRAN V.: Review: Kd-tree Traversal Algorithms for Ray Tracing. *Computer Graphics Forum 30*, 1 (2011), 199–213. 2

[HHZ*14] HU W., HUANG Y., ZHANG F., YUAN G., LI W.: Ray tracing via GPU rasterization. *Visual Computer 30*, 6-8 (June 2014), 697–706. 2

[KSY14] KIM T.-J., SUN X., YOON S.-E.: T-rex: Interactive global illumination of massive models on heterogeneous computing resources. *Transactions on Visualization and Computer Graphics 20*, 3 (2014), 481–494. 2

[LSLS09] LAINE S., SILTANEN S., LOKKI T., SAVIOJA L.: Accelerated beam tracing algorithm. *Applied Acoustics 70*, 1 (2009), 172–181. 2

[LYTM08] LAUTERBACH C., YOON S.-E., TANG M., MANOCHA D.: Reducem: Interactive and memory efficient ray tracing of large models. *Computer Graphics Forum 27*, 4 (2008), 1313–1321. 2

[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. *Transactions on Graphics 29*, 3 (2010), 1–10. 2, 8

[MBW08] MATTAUSCH O., BITTNER J., WIMMER M.: Chc++: Coherent hierarchical culling revisited. *Computer Graphics Forum 27*, 3 (2008), 221–230. 3, 5

[Mit07] MITTRING M.: Finding next gen: Cryengine 2. In *SIGGRAPH Courses* (2007), ACM, pp. 97–121. 7

[Mor11] MORA B.: Naive ray-tracing: A divide-and-conquer approach. *Transactions on Graphics 30*, 5 (2011), 117. 2

[MWH*06] MÜLLER P., WONKA P., HAEGLER S., ULMER A., VAN GOOL L.: Procedural modeling of buildings. *Transactions on Graphics 25*, 3 (July 2006), 614–623. 9

[ND12] NOVÁK J., DACHSBACHER C.: Rasterized bounding volume hierarchies. *Computer Graphics Forum 31*, 2 (2012), 403–412. 2

[PBD*10] PARKER S. G., BIGLER J., DIETRICH A., FRIEDRICH H., HOBEROCK J., LUEBKE D., MCALLISTER D., MCGUIRE M., MORLEY K., ROBISON A., STICH M.: Optix: a general purpose ray tracing engine. *Transactions on Graphics 29* (2010), 66:1–66:13. 2

[PFHA10] PANTALEONI J., FASCIONE L., HILL M., AILA T.: Pantaray: Fast ray-traced occlusion caching of massive scenes. *Transactions on Graphics 29* (2010), 37:1–37:10. 3

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. In *Proc. SIGGRAPH* (1997), pp. 101–108. 2

[RAH07] ROGER D., ASSARSSON U., HOLZSCHUCH N.: Whitted ray-tracing for dynamic scenes using a ray-space hierarchy on the GPU. In *Proc. Eurographics Symposium on Rendering* (2007), pp. 99–110. 2

[RGD09] RAMANI K., GRIBBLE C. P., DAVIS A.: StreamRay: a stream filtering architecture for coherent ray tracing. *ACM SIGPLAN Notices 44*, 3 (2009), 325–336. 2

[SBM03] STANEKER D., BARTZ D., MEISSNER M.: Improving occlusion query efficiency with occupancy maps. In *Proc. Symposium on Parallel and Large-Data Visualization and Graphics* (2003), pp. 15–. 3

[ST90] SAITO T., TAKAHASHI T.: Comprehensible rendering of 3-D shapes. *Computer Graphics 24*, 4 (1990), 197–206. 3

[WMG*09] WALD I., MARK W. R., GÜNTHER J., BOULOS S., IZE T., HUNT W., PARKER S. G., SHIRLEY P.: State of the Art in Ray Tracing Animated Scenes. *Computer Graphics Forum 28*, 6 (2009), 1691–1722. 2

[ZRD14] ZIRR T., REHFELD H., DACHSBACHER C.: Object-order ray tracing for fully dynamic scenes. In *GPU Pro 5*. A K Peters/CRC Press, 2014. 2