PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes

Jacopo Pantaleoni* NVIDIA Research
Luca Fascione† Weta Digital
Martin Hill† Weta Digital
Timo Aila* NVIDIA Research

Figure 1: *The geometric complexity of scenes rendered in the movie Avatar often exceeds a billion polygons and varies widely: distant rocks and vegetation are tessellated to a level of meters and centimeters, while the faces of even distant characters are modeled to over 40,000 polygons from forehead to chin. The spatial resolution of occlusion caches precomputed by our system also spans several orders of magnitude.*

Abstract

We describe the architecture of a novel system for precomputing sparse directional occlusion caches. These caches are used for accelerating a fast cinematic lighting pipeline that works in the spherical harmonics domain. The system was used as a primary lighting technology in the movie Avatar, and is able to efficiently handle massive scenes of unprecedented complexity through the use of a flexible, stream-based geometry processing architecture, a novel out-of-core algorithm for creating efficient ray tracing acceleration structures, and a novel out-of-core GPU ray tracing algorithm for the computation of directional occlusion and spherical integrals at arbitrary points.

CR Categories: I.3.2 [Graphics Systems C.2.1, C.2.4, C.3)]: Stand-alone systems—; I.3.7 [Three-Dimensional Graphics and Realism]: Color,shading,shadowing, and texture—Raytracing;

Keywords: global illumination, precomputed radiance transfer, caching, out of core

1 Introduction

The movie Avatar featured unprecedented geometric complexity (Figure 1), with production shots containing anywhere from ten million to over one billion polygons.

To make the rendering of such complex scenes manageable while satisfying the need to provide fast lighting iterations for lighting artists and the director, modern relighting methods based on spherical harmonics (SH) [Ramamoorthi and Hanrahan 2001] and image-based lighting [Debevec 1998] were used. These methods can speed up the lighting iterations significantly, but unfortunately require an extremely compute and resource intensive precomputation of directional occlusion information. Directional occlusion encodes the visibility term used for lighting modulation as a function of direction, and is typically computed using ray tracing.

We describe PantaRay[1], a system designed to make this precomputation practical by leveraging the development of modern ray tracing algorithms for massively parallel GPU architectures [Aila and Laine 2009] and combining them with new out-of-core and level of detail rendering techniques.

The PantaRay engine is an out-of-core, massively parallel ray tracer designed to handle scenes that are roughly an order of magnitude bigger than available system memory, and that require baking spherical harmonics-encoded directional occlusion (SH occlusion) and indirect lighting information for billions of points with highly varying spatial density.

Our key contributions are the introduction of a flexible, stream-based geometry processing architecture, a novel out-of-core algorithm for constructing efficient ray tracing acceleration structures, and a novel out-of-core GPU ray tracing algorithm for the computation of directional occlusion and spherical integrals. These are

*e-mail:{jpantaleoni,taila}@nvidia.com
†e-mail:{lukes,martinh}@wetafx.co.nz

[1] A twist on the Greek aphorism *panta rei*, i.e. *everything flows*

ACM Reference Format
Pantaleoni, J., Fascione, L., Hill, M., Aila, T. 2010. PantaRay: Fast Ray-traced Occlusion Caching of Massive Scenes. *ACM Trans. Graph. 29*, 4, Article 37 (July 2010), 10 pages. DOI = 10.1145/1778765.1778774 http://doi.acm.org/10.1145/1778765.1778774.

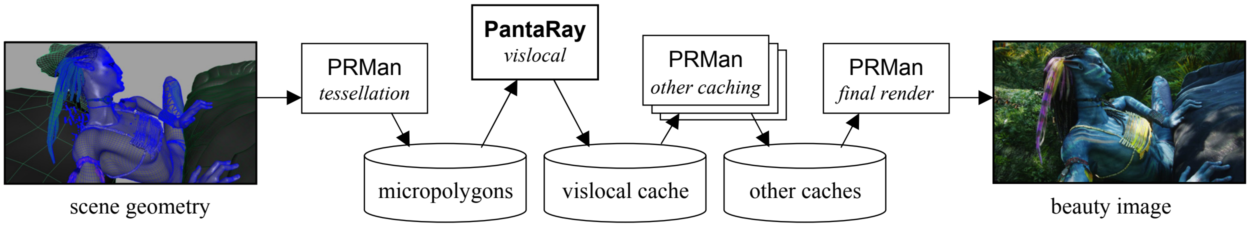


Figure 2: *A visual representation of the rendering pipeline used for the movie Avatar showing the various passes, the data flow among them, and the role played by our system.*

combined into a new precomputation system designed to efficiently handle very high levels of geometric complexity.

Our system has been integrated into the production pipeline of Weta Digital and is showcased in the movie Avatar, but the algorithmic contributions and design decisions discussed in this paper could be usefully applied in other domains, such as large-scale scientific visualization, which would benefit from rich lighting of extremely complex geometric datasets.

2 Related Work

Much research has addressed the topic of massive model rendering and visualization. Here we compare our system to some of the most relevant work.

There is a vast amount of literature on the topic of direct visualization of massive triangle meshes. Most such methods, including [Borgeat et al. 2005] and [Cignoni et al. 2004], subdivide the models into cells or patches and create multiple or progressive LOD representations of those elements through mesh simplification. As the goal of our system is not direct visualization but rather the computation of low-frequency directional occlusion information, these accurate simplification methods are not needed and we resort to much cruder representations. Moreover, as we target ray tracing, our out-of-core spatial index construction had the additional requirement of targeting high ray tracing efficiency, employing partitioning and subdivision methods based on the surface area heuristic (SAH) [Havran 2000].

Wald et al. [2005] and Yoon et al. [2006] introduced two systems based on *level of detail* (LOD) for ray tracing large triangle meshes. Unlike our approach, their systems relied on OS-level memory mapping functionality and targeted moderately parallel systems such as commodity multi-CPU systems, performing LOD selection in each thread independently. This strategy would not be portable to modern massively parallel GPU architectures. Moreover, no special effort was taken to speed up the out-of-core construction of the acceleration structure, which in the case of [Wald et al. 2005] took up to a day for a model containing 350M triangles.

Crassin et al. [2009] and Gobbetti et al. [2008] introduced two systems to render large volumetric datasets. These systems perform direct visualization of geometry represented as voxel grids, rather than computing complex visibility queries. Like our system, both approaches decompose computation into a CPU-based LOD selection phase and a GPU-based rendering phase. Their systems perform these steps to visualize the entire model from a single point of view at each frame, while we do it to compute directional occlusion from large batches of nearby points at the same time.

Christensen et al. [2003] presented a ray tracing system using ray differentials to perform LOD selection for high order surfaces. The described system is able to efficiently handle very large tessellations of the base meshes, but does not provide a level of detail scheme to handle base meshes which do not fit in main memory. This was essential for our approach, which needed to handle base meshes with hundreds of millions or billions of control polygons.

Budge et al. [2009] presented an out-of-core data management layer for path tracing on heteregeneous architectures. The system builds on a dataflow network of kernel queues and a rendering-agnostic task scheduler that prioritizes the execution of kernels based on data availability, queue size and other criteria. The path tracer exploits this generic framework by using a two-level acceleration structure, where each second level out-of-core hierarchy is bound to a distinct processing queue, extending the work of [Pharr et al. 1997]. The resulting algorithm shows good scalability and thus satisfies one of our main requirements. Unlike their work, we focus on developing highly efficient special-purpose algorithms for the computation of directional occlusion, minimizing I/O through careful LOD selection, and on the problem of efficient construction of high quality out-of-core acceleration structures.

Ragan-Kelley et al. [2007] introduced Lightspeed as an interactive lighting preview system that can greatly accelerate relighting with local light sources and shadow maps in the presence of programmable shaders. Unlike their work, we focus on the efficient computation of complex visibility for fast image based lighting in massive scenes.

3 System Overview

Lighting of the movie Avatar was performed with a spherical harmonics lighting pipeline based on the work of Ramamoorthi and Hanrahan [2001], in which light transport is decomposed into a multiple product integral:

$$L_o(x,\omega_o) = \int_{\Omega^+} L_i(x,\omega)\rho(x,\omega,\omega_o)V(x,\omega)\langle\omega,\hat{n}\rangle d\omega \qquad (1)$$

where L_o is the exitant radiance, x is the point of interest, ω_o is the outgoing direction, Ω^+ is the hemisphere above x, L_i is incident radiance, ω is the incident direction, ρ is the BRDF, V is the visibility function, $\hat{n}$ is the normalized surface normal and $\langle\cdot,\cdot\rangle$ indicates the scalar product operator.

In this framework, directional visibility is precomputed at sparse locations in the scene and stored in a spherical harmonics basis. Building on the work of Kautz et al. [2002], Ng et al. [2004], and Snyder [2006], this directional visibility can then be reused over many lighting cycles by performing a simple dot-product with the less expensive terms of the equation, which are computed at render time. Our system was built to efficiently perform this precomputation on massive scenes of unprecedented complexity.

The overall pipeline is divided into several computation *passes* as depicted in Figure 2. During preparation, the scene geometry is tessellated and divided into *micrograms* according to a camera-based metric, using a custom point cloud output driver in PhotoRealistic RenderMan (PRMan). We store these microgrids on disk in a stream representation which allows vertices to be associated

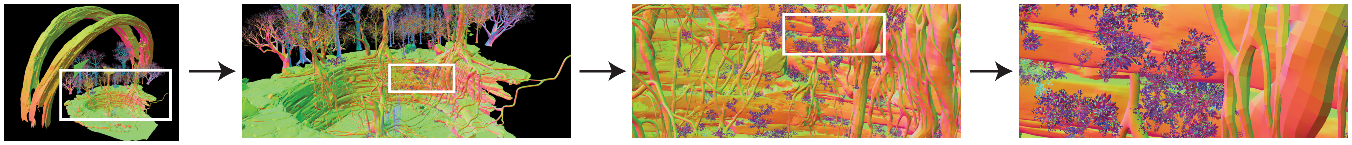


Figure 3: *Zooming into scene 6 shows the various levels of tessellation.*

with arbitrary user data, much like the *primitive variable* mechanism in PRMan [Upstill 1990] or the *vertex attribute* machinery in OpenGL [Segal and Akeley 1999]. In order to include occluding geometries not directly visible to the camera, assets outside of the viewing frustum are also tessellated, either using a relatively large overscan or according to a world-based metric. Figure 3 shows an example of the various tesselation densities encountered in a typical production scene.

The *vislocal pass* invokes our PantaRay engine to augment the microgrid stream with directional occlusion data encoded in the spherical harmonics basis and other precomputed quantities such as area light visibility, blurred reflections and occasionally one-bounce indirect lighting. All these properties are generated by programmable shaders using the ray tracing capabilities of our engine.

In the end the result of the PantaRay precomputation is used in PRMan to render the final images in what is called the *beauty pass*. In this pass, the lighting, BRDF and visibility fields are composed at render time at a very low cost, to the point where the lighting iterations can happen inside the beauty pipeline at final quality.

While the *vislocal* datasets can be reused for many lighting iterations, which greatly offsets their computation cost, computing *vislocal* remains an extremely resource-intensive process, and is a natural point to start looking for optimizations.

To illustrate the targeted complexity, the movie Avatar required baking scenes with tens of thousands of different plants modeled as subdivision surfaces at a resolution of 100K to 1M control polygons each, and hundreds of characters modeled at a resolution of 1-2M control polygons. Since occlusion is a global effect, out-of-camera objects must be kept during the computation. Similarly, translucence and subsurface scattering require processing geometry that is not directly visible from the camera. Rather than tracing full resolution models, lower resolution proxies could have been developed and used for far away assets. While our pipeline used stochastic simplification to reduce the complexity of vegetation before rasterization [Cook et al. 2007], we did not explore the possibility of performing any additional simplification to the ray tracing assets before they entered our system: we chose instead to construct a fully automated system capable of directly handling the raw model complexity rather than create a semi-automatic pipeline for proxy generation.

The highly variable spatial resolution of the PantaRay output presented another challenge: many shots in these scenes required a spatially varying baking resolution ranging from a few points per meter on distant geometry such as terrains, to several points per millimeter, for example to accurately represent the lighting on and under the characters' fingernails.

The speed and memory limitations of existing general purpose ray tracing technology, and the reduced flexibility and programmability in other special purpose baking tools, such as *ptfilter* [Christensen 2008], did not scale to these production needs. In practice, our goal was to raise the tractability limit of shots in the movie Avatar by roughly 2 orders of magnitude in terms of both speed and scene size while keeping a reasonable degree of programmability.

4 Architecture

Handling the necessary complexity inside a flexible ray tracing system requires efficient out-of-core and streaming techniques. To support the use of such methods throughout the entire software pipeline, we designed the system around the concept of *microgrid streams*, which are opaque sources of micrograds (that is *micropolygon grids* as in [Cook et al. 1987]). Microgrid streams can be read into main memory and eventually *rewound*, or restarted from the beginning. Such streams can represent either geometry stored on disk or procedural geometry. Each microgrid is essentially a small indexed mesh with up to 256 vertices forming micropolygons, where each micropolygon can have one, two, three or four vertices (to represent points, lines, triangles and quads). Vertices are represented by their position, a normal, a radius and any attached user data. We decided to disallow any form of random access for two reasons: first, geometry files are typically compressed to save disk space and potentially achieve higher I/O bandwidth; second, input streams could be procedurally generated, and the procedural generation function might not allow for individual primitive generation (as for example in some L-systems).

The input to PantaRay is an XML scene description, containing a list of *shaders*, a list of *geometries* and their associated binding relationships.

A *geometry* is a microgrid stream, which can specify both an *occluder* and a collection of *bake sets*. A *bake set* represents the central PantaRay unit of work, and specifies that the input stream should be cloned to a corresponding output stream and further decorated with a given list of shader output attributes. Geometries can further be instanced through a user-defined transformation, potentially specifying a procedural displacement shader.

Shaders are programmable units responsible for computing some required information at the vertices of each microgrid in a bake set.

The first task that PantaRay performs after parsing the scene file is building an out of core acceleration structure (AS) for the input occluder geometry. After the AS is built, PantaRay processes the bake sets and begins shader execution. The following sections describe these processes in detail.

4.1 Acceleration Structure Generation

The main bottleneck in building an out-of-core acceleration structure can easily be I/O speed, as typical bounding volume hierarchies (BVH) or *k*-d tree building strategies require touching all the objects multiple times. Even taking into account the performance of state-of-the-art storage technologies, the system had to assume that tens of thousand of concurrent processes would be using the same storage, requiring all non-local I/O to be modeled as a high latency, high bandwidth device.

Hence we developed a general purpose stream-based builder which tried to minimize the number of times the stream is rewound.

The first component of this builder is a streaming *bucketing* pass designed to handle hundreds of millions of microgrids. The bucketeer uses a simple binning approach: it constructs a regular 3d grid by first streaming the geometry once to count how many microgrids

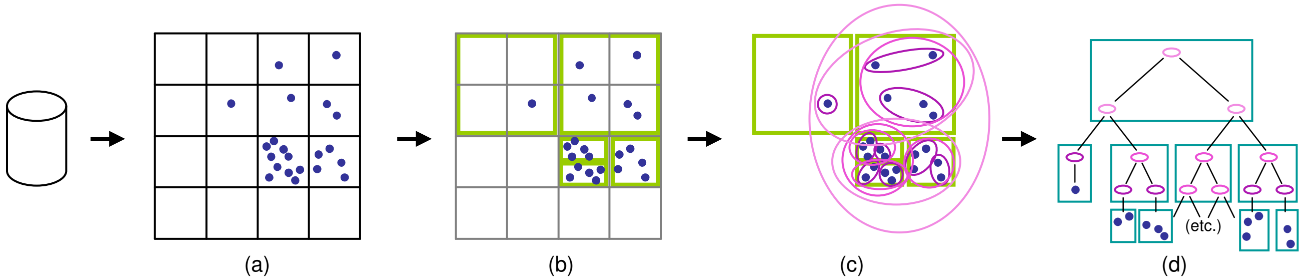


Figure 4: *Out-of-core spatial index construction. Micrograids stream from disk into a regular grid of buckets (a). Buckets are coalesced and split into chunks (b) of up to 64KB. A BVH inside and among chunks (c) is broken into bricks (d) of up to 256 nodes. Each brick is contiguous on disk.*

fall in each bucket, and then streaming it a second time to populate those buckets on local disk.

The first streaming pass reserves the correct amount of disk space for each bucket and creates an index, but also keeps statistics about the number of micrograids, micropolygons, vertices and byte size for each of them.

The second pass of the algorithm loops through each micrograid to find out all the buckets in which the micrograid falls, and records the micrograid-bucket pairs into an in-memory cache with a few million entries. Once the cache is full, the pairs are sorted by bucket index and written to disk in their corresponding slot, essentially making a single seek per bucket or less per cache flush.

The purpose of this bucketing pass is to create manageable units of work which could fit in memory. However, the resulting uniform grid is very coarse and often imbalanced, which makes it unsuitable for direct ray tracing. With extremely large scenes it frequently happens that a large portion of the buckets are empty or very sparsely populated, while a few remain too densely populated.

For these reasons, after the bucketing is done, we perform a *chunking* pass, whose purpose is to build a second disk-based spatial index with more uniform distribution of geometry, aggregating low-complexity buckets and splitting high-complexity ones untill all occupy roughly 64KB of memory. We consider an implicit k-d tree over the uniform grid of buckets. First, we perform a bottom-up propagation of statistics from the leaves to the parents, so that for each node it is possible to compute a rough estimate of the aggregate size (as some of the buckets in a subtree might contain duplicate references to the same micrograids, which will eventually be merged, it is excessively expensive to obtain exact values).

Next, we traverse the tree top-down and emit a new aggregate cluster of buckets (called a *chunk*) as soon as a node with suitable statistics is found. When we encounter a leaf which is too big, we split it further. This splitting operation is done either in memory using a classical BVH builder based on the SAH, or using the previously described bucketing algorithm if the leaf contains more data than a user-selected working-set size (typically corresponding to a few million micrograids). At this point geometry is split into chunks that fit in main memory, but the geometry within each chunk is unorganized. When we emit a new chunk, we build a SAH-based BVH on all its micropolygons. Finally, all the chunk hierarchies thus generated are organized together into a single *top-level* SAH-based BVH constructed in memory (it is sufficient to keep a bounding box and an index per chunk hierarchy).

As the trees are generated, they are further split into smaller treelets containing up to 256 nodes and 256 vertices each, called *bricks*, which are stored on disk as a contiguous segment of data. In such treelets, each node is either an internal node, containing a single pointer to the first of two consecutive children within the brick, or a leaf. Leaves can either point to the root of another brick, or reference a list of primitives. This process is illustrated in Figure 4.

As each brick contains less than 256 micropolygon references, all face and vertex indices are stored using an 8 bit representation, making topology information extremely compact. Vertex attributes from the leaf geometry are propagated up to each node in the treelets doing simple averaging, as they are later used for LOD. The only attribute which receives special treatment is *opacity*, which is multiplied at each level by a very crude estimate of the average occlusion caused by the entire subtree below the node to random rays hitting its bounding box:

$$\text{occlusion}(\mathit{subtree}) \approx \min\left(1, \frac{\sum_i \text{area}(\mathit{micropolygon}_i)}{\text{area}(\text{bbox}(\mathit{subtree}))}\right)$$

While in pathological cases the error of this estimate can be arbitrarily high, in practice it proved well suited for the kind of incoherent geometry found in the vegetation of Avatar's jungle. If needed, the precision of this approximation could be improved employing a technique similar to the ones described by Christensen [2008] or by Lacewell et al. [2008].

4.2 Shader Execution

Shaders are executed for each point defined in a bake set. As an optimization, a chain of filters can be applied to these *point streams* before the shaders actually process them. For example, we have implemented filters to perform the usual tasks of occlusion culling, backface culling and frustum culling, as well as camera-based and feature-based stream decimation.

After the filters are run, the point streams are broken up into a list of spatially organized batches (containing a number of points corresponding to up to 1M rays) built by a fast median-split k-d tree construction algorithm. The resulting batches generated by the k-d splits are reordered along a Hilbert curve passing through their centroids to maximize spatial coherence for the following processing passes.

The shading engine provides two basic building blocks to the shaders: a general purpose ray tracer (used for example by an area light shadows shader) and a custom, massively parallel ray tracer designed to perform spherical sampling queries (used by SH occlusion and one bounce indirect lighting shaders).

The decision to build a custom ray tracer for the latter was made to allow the application of smart, domain-specific LOD and streaming techniques to our most computationally intensive shaders. The next sections describe the two tracers in detail.

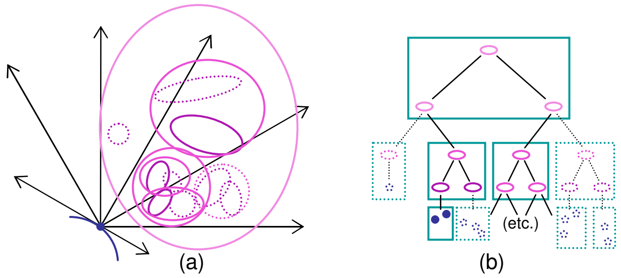


Figure 5: *LOD selection. (a) Sample rays emanate from a surface point. Bricks whose bounding volumes are smaller than the average distance between sample rays are unloaded from memory (dashed lines); the rest are loaded. (b) The result is a* cut *through the BVH which is detailed near the ray origins, but coarse farther away.*

```
load_bricks (batch_bbox, brick, $\sigma$)
load brick

foreach node in brick's treelet
  if node is a leaf pointing to another brick
    child_brick = node.brick()
    child_bbox = node.bbox()
    // check if child_brick's projection
    // subtends an angle larger than $\sigma$
    sq_diag = square_diagonal( child_bbox )
    sq_dist = square_distance( child_bbox, batch_bbox )
    if sq_diag > $\sigma^2$ * sq_dist
      load_bricks( batch_bbox, child_brick, $\sigma$ )
    else
      mark child_brick and its descendents as unloaded
```

Figure 6: *Pseudocode for the brick LOD selection procedure.*

4.3 Ray Tracing for Arbitrary Queries

The general-purpose ray tracer is currently used by simple reflection occlusion and area light shaders only. Both of these shaders need to cast just a few rays per point and typically require only a fraction of the runtime of the SH occlusion and one-bounce indirect lighting shaders.

Currently this ray tracer is implemented on the CPU using a standard multi-threaded stack-based BVH traversal kernel, where each thread processes separate rays. In order to amortize the I/O costs involved in traversing the disk-based acceleration structure, we keep the traversed bricks in a large shared LRU cache. Furthermore, to avoid excessive locking, each thread keeps a smaller local cache of the most recently used 32 bricks. We found that this approach scales quite well up to 8 CPU threads, giving a sustained CPU usage of about 95%.

4.4 Ray Tracing for Spherical Sampling

For each point, the SH occlusion and irradiance gathering shaders need to cast a set of rays (typically 512 or 1024) uniformly or cosine-distributed over a hemisphere or full sphere, and respectively project the resulting occlusion values into the spherical harmonics basis or integrate the incident radiance.

Running such shaders is extremely compute-intensive, as many scenes require baking up to billions of points; hence we decided to make use of the massive parallelism available in modern GPUs to accelerate the process.

```
trace()
while any active ray
  if any active ray is in leaf node
    perform wide_leaf_intersection() // Figure 8
    pop traversal stack

  for $i$ = 0 ... 8 // short while loop
    if node is a new brick
      if new brick is not loaded
        report intersection with new brick's bbox
        pop traversal stack
      else
        jump to new brick's root
    else if node is not leaf
      traverse to next node
    else
      break // node is a leaf
```

Figure 7: *Pseudocode for the SIMT tracing kernel. Each thread processes a separate ray.*

However, it was clear from the beginning that the ray tracing approach described in the previous section would not scale to thousands of threads due to the intrinsic locking and resource contention involved in keeping and updating a global LRU cache independently for each ray.

To solve the problem efficiently we designed a LOD-based algorithm which first determines and streams in all the required bricks for all rays generated by all points in a batch (Figure 5), and then proceeds to trace all rays through the resulting *cut* of the brick hierarchy with a single traversal kernel implemented in CUDA. Bricks which deemed unnecessary by the LOD selection scheme are approximated during traversal as partially transparent bounding boxes (Section 4.1). The rationale behind this approach is that while the whole scene doesn't fit in main memory and while it is often impossible to build a single low-resolution representation of the entire scene valid for all points at once, the amount of geometry needed to capture detailed occlusion in a confined region of space is usually small. Hence, we create a custom far-field representation for each batch.

The LOD selection scheme is based on a crude estimate of ray differentials coupled with an argument inspired by the Shannon sampling theorem: the initial assumption is that each ray represents, on average, a solid angle of $\sigma = 2\pi/n$ steradians (or $\sigma = 4\pi/n$ in the fully spherical case), where n is the number of samples taken per point. Furthermore, we assume that the sampling frequency is sufficient to sample the underlying signal implied by the scene geometry, so that for each point it suffices to represent the scene geometry up to a projected angular resolution of σ steradians. While these assumptions do not strictly hold in a mathematical sense, in practice this does not cause visually disturbing artifacts, as the result is roughly equivalent to smoothing the sampled signal by means of a box filter in the ray direction domain.

In order to determine all the bricks needed for all the points in a given batch, we run the algorithm in Figure 6. The bricks are loaded in the host memory in a contiguous memory arena managed by a single-threaded LRU cache.

To minimize memory fragmentation, we wrote a special purpose allocator that performs incremental *coalescing* by removing small unused memory fragments from in between successive bricks. An index array keeps track of which bricks are loaded and where. After loading is completed, the parts of the arena which changed in the last loading session are mirrored to the GPUs present in the sys-

```
wide_leaf_intersection()
tidx = threadIdx.x; // $[0, 32)$
// count the number of ray-primitive tasks.
// haveLeaf indicates if the current node is a leaf
[lo,hi) = scan(haveLeaf ? numPrims : 0);
numIsect = hi from thread 31 // how many left?

while (numIsect $> 0$)
{
  // select up to 32 ray-primitive tasks
  foreach primitive p, with lo+p $\in [0, 32)$
    write (tidx,p) pair to shared[lo+p]

  // get a ray-primitive task to execute
  (srcThread,p2) = shared[tidx];

  // copy needed variables from "srcThread" thread
  foreach 32bit variable var needed in intersection
    shared[tidx] = var; // write my own variable
    var2 = shared[srcThread]; // read from "srcThread"

  // intersection using vars copied from "srcThread"
  if (tidx $<$ numIsect)
    shared[tidx] = intersectRayPrim(p2);

  // collect intersections of my ray
  foreach prim p, with lo+p $\in [0, 32)$
    modify ray according to shared[lo+p]

  lo = lo-32; hi = hi-32; numIsect = numIsect-32;
}
```

Figure 8: *Pseudocode for warp-wide primitive intersection. This implementation assumes 32-wide SIMD/SIMT so that 32 threads execute all statements in lock-step. All threads in the executing warp must be active when entering the function.* shared *is a shared memory array, accessible to all threads.*

tem. We pay special care to minimize the number of host-to-GPU transfers by aggregating as many potentially non-contiguous dirty blocks of the memory arena as possible in each transfer, as long as the total number of bytes in the transfer exceeds the original size of the merged blocks by less than 20%.

Finally, we invoke a shading kernel on each GPU to generate and trace all the rays spawned by a subset of the points in the processed batch. We use the simplest possible load-balancing algorithm by letting each GPU process $1/m^{th}$ of the points, where m is the number of available GPUs. We build the subsets assigned to each GPU using another recursive median split pass, in order to maximize coherence.

The tracing algorithm (Figure 7, on the previous page) follows the lines of the *while-while* traversal kernel described by Aila and Laine [2009], adapted to handle the treelet organization of the underlying BVH. One notable difference is the way in which leaf nodes are processed. Due to the very high scene complexity and correspondingly high tree depth, we observed very low Single Instruction Multiple Thread (SIMT) utilization in primitive intersection, often close to 25%. Hence, we decided to implement a *warp-wide* intersection algorithm (Figure 8) which first counts the number of primitives in the leaf nodes currently being proccessed by the active rays in a warp and then distributes the ray-primitive pairs to all threads in the warp, finally performing a reduction to determine the closest intersection per ray. The resulting algorithm raised SIMT utilization to 50-60%.

5 Design Decisions

In this section we expand on some of the design decisions made and alternatives explored.

Acceleration structure construction. We chose to focus almost all efforts on I/O minimization, rather than on algorithm parallelization. In fact, while many parts of our algorithm lend themselves naturally to parallelization (as for example its deep bucketing passes), we believe that their heavy usage of intrinsically serial I/O resources would have dampened most of the potential gains for the kind of scenes we were interested in.

LOD-based ray tracing. We explored several approaches before settling on a single multi-resolution hierarchy of bricks representable at different levels of detail. We initially tried different approximation schemes in which we constructed separate scene representations for efficient in-memory tracing of the far-field, while performing accurate ray tracing queries against the original geometry in the near-field only. These schemes included creating both uniform and non-uniform voxelizations of the scene which could fit in main memory, represented using sparse octrees and *k*-d trees. The resolution of the surrounding voxelization was used at each baking point to size the near-field region so as to avoid any visible artifacts with respect to the local baking resolution.

The resulting algorithm proved very efficient for small and medium sized scenes, but failed completely to handle the most complex cases. The extremely uneven geometric density, and the even less uniform density of baking points, often spanning several orders of magnitude, made it impossible to build a single voxelization useful for the entire scene at once.

This led to our decision of adopting a multi-resolution scene representation coupled with a *local* strategy for far-field and near-field partitioning.

Numerical precision. Handling of complex scenes required special attention to numerical precision. Some of the scenes in Avatar were expressed in centimeters, with many submillimeter-sized polygons located several hundred meters from the origin.

To avoid self-intersection artifacts while keeping relatively good performance, we used a simple adaptive precision strategy: all computations, from acceleration structure traversal to primitive intersection, are initially performed using single-precision floating points. However, if an intersection is found which is within 1 millimeter from the ray origin, the intersection is recomputed at double precision. In practice, double precision primitive intersection is executed for less than 5% of the intersections. While this simple scheme still occasionally misses intersections, it substantially reduces self-intersections, the biggest cause of visual artifacts.

6 Performance

For all performance measurements, we tested our system on a server with dual-socket Intel Core i7-class quad-core CPUs with 16GB of main memory and 4 Tesla S1070 T10 GPUs, with 4GB of memory per GPU.

The ray tracing algorithm described in Section 4.4 exhibits extremely high cache efficiency under most difficult scenarios, reducing substantially the required amount of I/O. For example, on a scene containing more than one billion micropolygons, baked with a cache size of just 1GB, we could process each batch of 1024 points, corresponding to 1M ray queries, streaming on average slightly less than 2MB of geometry from disk. This is equivalent to less than 2 bytes of geometry per ray.

Shot	Occluder size	Bake size	BrickMap metrics	Avg I/O	Cache	Tracing speed	Trace	Turnaround
	/ grid	/ shaded	build time and size		efficiency		time	time
	μpolys / size	μpolys / rays	min / bricks / GB	byte / batch	hit / query	shaded rays/s		
1	1016 M / 4.00	1016 M / 508.0 G	94.95 / 1864 k / 46.76	1.967 M	99.28 %	8.659 M	16 h 41 m	18 h 16 m
2	575 M / 4.00	575 M / 287.0 G	53.81 / 1040 k / 26.24	2.008 M	98.52 %	13.51 M	6 h 3 m	6 h 57 m
3	157 M / 3.59	12 M / 5.9 G	17.46 / 318 k / 7.42	8.296 M	96.16 %	11.38 M	0 h 9 m	0 h 26 m
4	125 M / 4.02	125 M / 62.5 G	16.99 / 237 k / 5.737	2.124 M	99.21 %	15.42 M	1 h 9 m	1 h 26 m
5	92 M / 3.39	92 M / 46.1 G	16.15 / 198 k / 4.496	0.5071 M	98.86 %	20.62 M	0 h 38 m	0 h 54 m
6	88 M / 3.91	88 M / 44.0 G	11.41 / 174 k / 4.146	1.299 M	99.12 %	22.18 M	0 h 34 m	0 h 45 m

Table 1: *Measured SH occlusion baking statistics from six shots illustrated in Figures 9 and 10.* Occluder size *reports the number of occluding micropolygons and their average distribution in grids as output by the tessellation pass.* Bake size *shows the number of query micropolygons and the total number of shaded rays required during the bake.* BrickMap metrics *reports the time taken to build the acceleration structure, the number of bricks in it and its size on disk.* Average I/O *reports the average per-batch transfer size between host and device.* Cache efficiency *shows the percentage of queries that hit in cache.* Tracing speed *reports the number of shaded rays per second; this includes the total cost of following a ray including its continuations due to non-opaque hits.* Trace time *is the time spent in the ray tracing phase of the render while* Turnaround time *reports the sum of the BrickMap construction time and the trace time.*

On complex scenes, a CPU implementation of the LOD-based ray tracing kernel achieves roughly 700K rays per second per core on the CPU. On the same scenes, the CUDA implementation achieves 15M rays per second per GPU. We also implemented a CUDA kernel for performing the SH sample projection. The Tesla S1070 system can execute this kernel at roughly 30× the speed of a quad-core CPU.

Table 1 shows comprehensive statistics on a few representative shots of varying complexity. All measurements were taken with a cache size of just 1GB, to better show the effectiveness of our caching strategy. In practice, we could raise the cache size up to a maximum of 4GB, as this is the total amount of global memory visible to each GPU. The number of samples per second reported in this table refers to the number of *shaded* rays per second. In this case, shading includes both recursive ray tracing due to semi-transparent surfaces and SH sample projection.

Almost all of the scenes required only a few MBs of I/O per batch, allowing the process to be executed at a speed close to the peak speed of the tracing kernels. The only notable exception is scene #3, which required an average I/O of about 8.3 MB per batch. This is due to the fact that the density of the baking points in this scene is roughly 1.5 orders of magnitude lower than that of the occluders' geometry. As such, it represents a harder case for our system, which is forced to load all the bricks overlapped by the bounding box of the shading batch. Increasing the cache size to 4GB reduces the problem noticeably.

7 Conclusions

We have presented a novel system for fast precomputation of spherical harmonics-encoded directional occlusion, which allows rapid cinematic lighting cycles at final movie quality. The precomputation is performed by a new ray tracing engine capable of efficiently handling the scenes of unprecedented complexity encountered in the making of the movie Avatar.

Construction of the system involved many interesting design trade-offs and required novel algorithms for fast out-of-core acceleration structure construction and LOD-based ray tracing for directional occlusion and uniform spherical sampling queries. Both algorithms have been developed on top of an efficient geometry streaming architecture in order to maximize I/O efficiency. While this project was particularly concerned with maximizing I/O performance to match the raw ray tracing performance of modern architectures, the broader themes of optimizing bandwidth and latency will be crucial for the future of ray tracing in general. The current architectural trends to increase compute power by increasing the number of cores, and the filmmakers' desire to always increase scene complexity, will likely not be balanced by a corresponding increase in the memory subsystem performance. Our system was designed with these challenges in mind.

7.1 Future Work

While our acceleration structure construction is serial, for smaller scenes it would be possible to build a parallel in-memory version taking advantage of its deep bucketing passes, in a spirit similar to the work of Wald et al. [2007].

We would also like to extend our general purpose LOD-less ray tracer to take advantage of massively parallel GPU architectures by employing a multi-pass approach where tracing is interleaved with paging of missing bricks [Budge et al. 2009].

To further increase physical realism and overall flexibility by supporting higher frequency effects, we plan to investigate the integration of our system with relighting methods [Hašan et al. 2006].

Acknowledgements We thank David Luebke, Joe Letteri, Sebastian Sylwan and Paul Ryan for supporting our work within NVIDIA Corporation and Weta Digital Ltd. Special thanks to Nick McKenzie, Jon Allitt and Eric Enderton for their precious feedback throughout our research, and also to Paolo Selva, Jeremy Pickett and Sean Walker for their help and support with the assembly of the imagery for the paper. Finally, we are grateful to Twentieth Century Fox for allowing us the use of images from the movie Avatar. Images courtesy of Weta Digital Ltd © 2009 Twentieth Century Fox Film Corporation. All rights reserved.

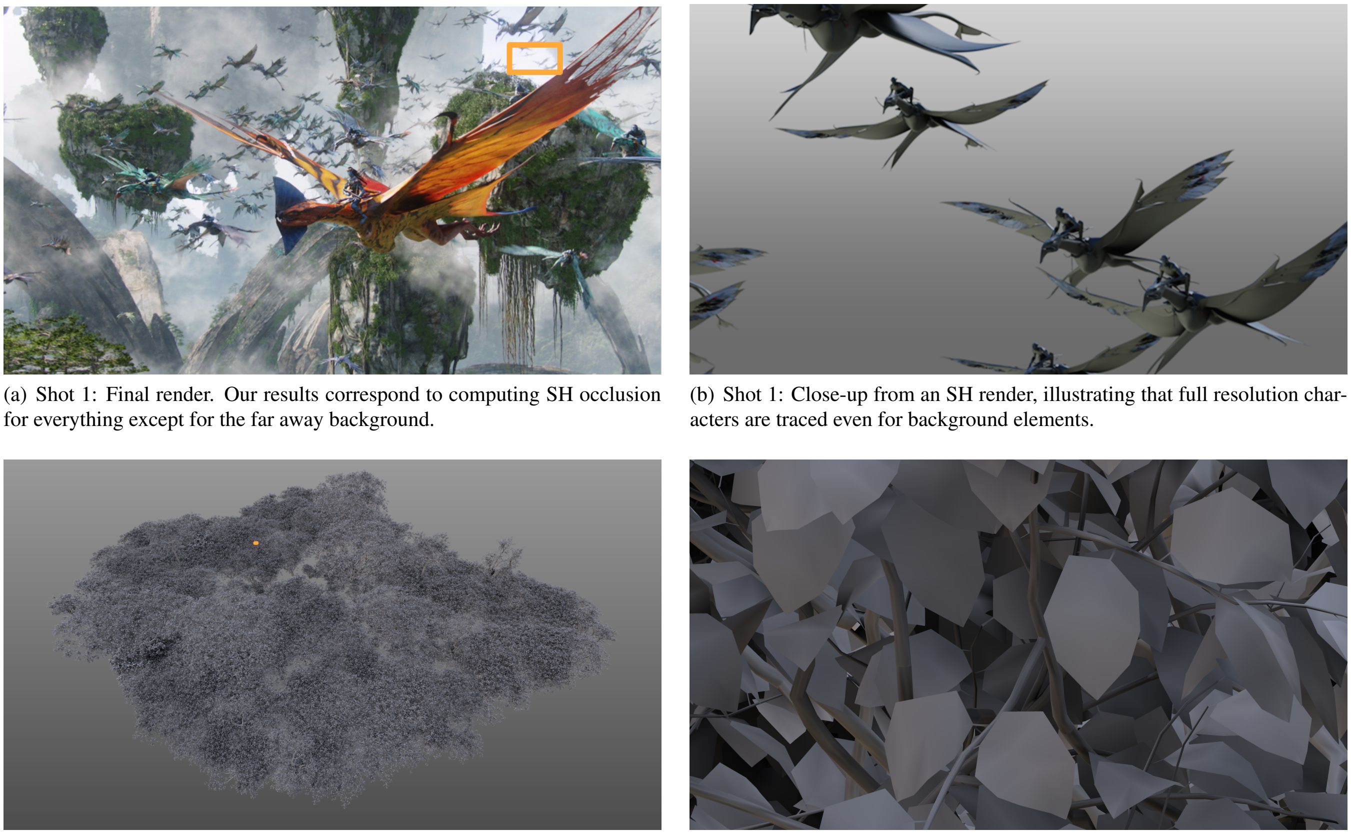
References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proc. High-Performance Graphics*, 145–149.

BORGEAT, L., GODIN, G., BLAIS, F., MASSICOTTE, P., AND LAHANIER, C. 2005. GoLD: interactive display of huge colored and textured models. *ACM Trans. Graph. 24*, 3, 869–877.

BUDGE, B. C., BERNARDIN, T., SENGUPTA, S., JOY, K., AND OWENS, J. D. 2009. Out-of-core data management for path tracing on hybrid resources. *Comp. Graph. Forum 28*, 2, 385–396.

CHRISTENSEN, P., LAUR, D., FONG, J., WOOTEN, W., AND BATALI, D. 2003. Ray differentials and multiresolution ge-



(a) Shot 1: Final render. Our results correspond to computing SH occlusion for everything except for the far away background.

(b) Shot 1: Close-up from an SH render, illustrating that full resolution characters are traced even for background elements.

(c) Shot 2: SH render. This forest was assembled for testing purposes. The orange marker indicates the close-up area.

(d) Shot 2: Close up of a small area, 7 × 4 pixels at film resolution.

Figure 9: *Renders for Shots 1 and 2.*

ometry caching for distribution ray tracing in complex scenes. *Comp. Graph. Forum 22*, 3, 543–552.

Christensen, P., 2008. Point-based approximate color bleeding. Pixar Technical Notes #08-01, July.

Cignoni, P., Ganovelli, F., Gobbetti, E., Marton, F., Ponchio, F., and Scopigno, R. 2004. Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models. *ACM Trans. Graph. 23*, 3, 796–803.

Cook, R. L., Carpenter, L., and Catmull, E. 1987. The REYES image rendering architecture. *Computer Graphics (Proc. SIGGRAPH 87) 21*, 4, 95–102.

Cook, R. L., Halstead, J., Planck, M., and Ryu, D. 2007. Stochastic simplification of aggregate detail. *ACM Trans. Graph. 26*, 3, 79.

Crassin, C., Neyret, F., Lefebvre, S., and Eisemann, E. 2009. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *Proc. ACM SIGGRAPH Symposium on Interactive 3D graphics and Games*, 15–22.

Debevec, P. 1998. Rendering synthetic objects into real scenes: bridging traditional and image-based graphics with global illumination and high dynamic range photography. In *Proc. ACM SIGGRAPH 98*, 189–198.

Gobbetti, E., Marton, F., and Iglesias Guitián, J. 2008. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer 24*, 7, 797–806.

Havran, V. 2000. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague.

Hašan, M., Pellacini, F., and Bala, K. 2006. Direct-to-indirect transfer for cinematic relighting. *ACM Trans. Graph. 25*, 3, 1089–1097.

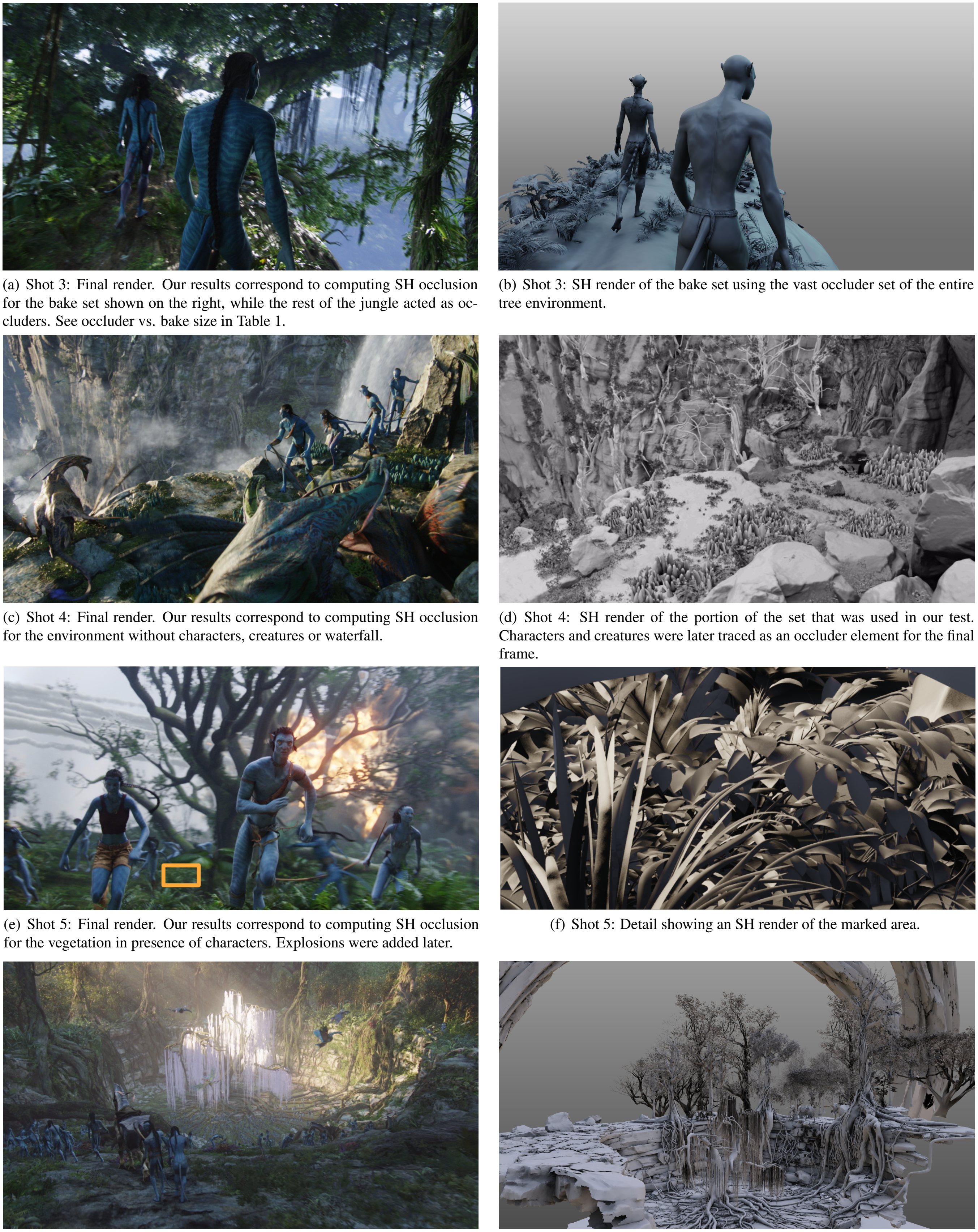
Kautz, J., Sloan, P.-P., and Snyder, J. 2002. Fast, arbitrary BRDF shading for low-frequency lighting using spherical harmonics. In *Proc. Eurographics Workshop on Rendering*, 291–296.

Lacewell, D., Burley, B., Boulos, S., and Shirley, P. 2008. Raytracing prefiltered occlusion for aggregate geometry. In *Proc. IEEE Symposium on Interactive Raytracing*, 19–26.

Ng, R., Ramamoorthi, R., and Hanrahan, P. 2004. Triple product wavelet integrals for all-frequency relighting. *ACM Trans. Graph. 23*, 3, 477–487.

Pharr, M., Kolb, C., Gershbein, R., and Hanrahan, P. 1997. Rendering complex scenes with memory-coherent ray tracing. In *Proc. ACM SIGGRAPH 97*, 101–108.

Ragan-Kelley, J., Kilpatrick, C., Smith, B. W., Epps, D., Green, P., Hery, C., and Durand, F. 2007. The lightspeed automatic interactive lighting preview system. *ACM Trans. Graph. 26*, 3, Article 25.



(a) Shot 3: Final render. Our results correspond to computing SH occlusion for the bake set shown on the right, while the rest of the jungle acted as occluders. See occluder vs. bake size in Table 1.

(b) Shot 3: SH render of the bake set using the vast occluder set of the entire tree environment.

(c) Shot 4: Final render. Our results correspond to computing SH occlusion for the environment without characters, creatures or waterfall.

(d) Shot 4: SH render of the portion of the set that was used in our test. Characters and creatures were later traced as an occluder element for the final frame.

(e) Shot 5: Final render. Our results correspond to computing SH occlusion for the vegetation in presence of characters. Explosions were added later.

(f) Shot 5: Detail showing an SH render of the marked area.

(g) Shot 6: Final render.

(h) Shot 6: SH render of the portion of the set that was used in our test.

Figure 10: *Renders for Shots 3–6 with final renders on the left and Spherical Harmonics (SH) renders on the right.*

RAMAMOORTHI, R., AND HANRAHAN, P. 2001. An efficient representation for irradiance environment maps. In *Proc. ACM SIGGRAPH 2001*, 497–500.

SEGAL, M., AND AKELEY, K. 1999. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Khronos group.

SNYDER, J., 2006. Code generation and factoring for fast evaluation of low-order spherical harmonic products and squares. Microsoft TechReport MSR-TR-2006-53, May.

UPSTILL, S. 1990. *The RenderMan Companion*. Addison-Wesley.

WALD, I., DIETRICH, A., AND SLUSALLEK, P. 2005. An interactive out-of-core rendering framework for visualizing massively complex models. In *ACM SIGGRAPH 2005 Courses*, 17.

WALD, I. 2007. On fast construction of sah-based bounding volume hierarchies. In *Proc. IEEE Symposium on Interactive Ray Tracing*, 33–40.

YOON, S.-E., LAUTERBACH, C., AND MANOCHA, D. 2006. R-LODs: fast LOD-based ray tracing of massive models. *The Visual Computer 22*, 9, 772–784.