Two-Level Ray Tracing with Reordering for Highly Complex Scenes

Johannes Hanika* Alexander Keller† Hendrik P. A. Lensch‡
Ulm University mental images GmbH Ulm University

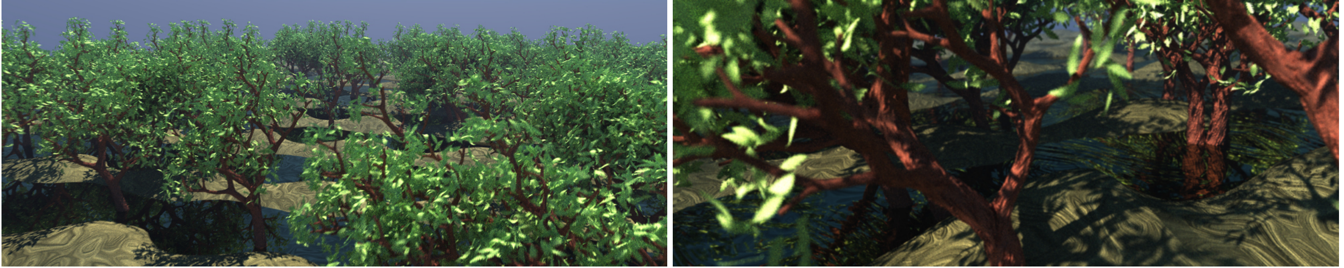


Figure 1: The forest with 100 trees has been modeled using displaced patches with motion blur and without instancing. At a resolution of 1920×768 with 32 samples per pixel, the left image has been rendered in 5:04 minutes on a 2.83 GHz quad core Q9550, where global illumination from the sun and sky was evaluated using path tracing. Fully tessellated, the model would consist of over $108 \cdot 10^9$ triangles. Due to on demand level of detail and ray sorting, only around 170 million triangles actually need to be created, without the need of caching.

Abstract

We introduce a ray tracing scheme, which is able to handle highly complex geometry modeled by the classic approach of surface patches tessellated to micro-polygons, where the number of micro-polygons can exceed the available memory. Two techniques allow us to carry out global illumination computations in such scenes and to trace the resulting incoherent sets of rays efficiently. For one, we rely on a bottom-up technique for building the bounding volume hierarchy (BVH) over tessellated patches in time linear in the number of micro-polygons. Second, we present a highly parallel two-stage ray tracing algorithm, which minimizes the number of tessellation steps by reordering rays. The technique can accelerate rendering scenes that would result in billions of micro-polygons and efficiently handles complex shading operations.

Index Terms: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing; I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms.

1 Introduction

In movie production, extreme geometric detail, complex shaders, and motion blur are needed to obtain visually compelling images. The Reyes architecture [CCC87] successfully deals with these challenges using a rasterization approach. The use of physically-based ray tracing is getting more and more common in the movie production, partly because the artists' experiences from real-world lighting design can be easily carried over. For this benefit, even the long render times are accepted [Gri09]. To retain the strengths of the Reyes architecture in a general ray tracing setting, we propose a two-level hierarchy approach, using reordering of computations instead of caching. After traversing a top-level hierarchy, rays are sorted to bundle those, which intersect the same bounding volume. Any necessary operation to be carried out in this volume, e.g. tessellation or loading a complex shader or bidirectional reflectance distribution function (BRDF) [NRH*77], is thus performed a minimum number of times. This results in significantly improved data locality, which allows us to fully ray trace computationally complex (procedural) displacements efficiently, i.e. corresponding to billions of micro-polygons without instancing (see Figure 1). Existing production pipelines can easily be extended to use our method, since the algorithm works on the same two-level data, such as displaced subdivision surfaces and sub-pixel-sized micro-polygons.

Rendering such scenes requires one to tessellate the free form patches or procedural displacements, which can be quite expensive with regard to computation and memory consumption. We accelerate ray tracing and global illumination by exploiting the two-level hierarchy of such scenes: The top-level hierarchy (Section 3.1) organizes the list of surface patches. After traversing the top-level, all rays are sorted according to patches they possibly intersect, increasing locality and minimizing the number of tessellation steps. The bottom-level consists of the micro-polygons which are tessellated on-the-fly on-demand. The micro-polygons of one patch are diced into a micro-polygon buffer, and a high-quality bounding volume hierarchy (BVH) is constructed in linear time in the number of micro-polygons (Section 3.2), exploiting the regular topology of a diced patch. Furthermore, the number of tessellation steps during rendering is reduced by adapting the level of detail (Section 3.3).

Altogether, the architecture collapses the inherent recursive nature of ray tracing to allow for better vectorization and combines the strengths of tracing ray packets [WBWS01], fast incoherent mono-ray traversal [DHK08], and rasterization: Our technique inherently handles displacements and procedural geometry, supports simple shader authoring and large depth complexity. It optimizes the utilization of memory bandwidth and coherence and furthermore is highly parallel.

2 Previous Work

A lot of work has been done to render complex geometry [SBB*06, LYM07, LYTM08] not specialized for the Reyes architecture. The fundamental assumptions and design principles of the Reyes image rendering architecture have allowed to model and render diverse and complex content, as postulated in the original publication [CCC87]. The concepts were so fundamental, that the many extensions (e.g. [HL90, LV00]) seamlessly complemented the basic architecture. As one of the design principles was to keep expensive

*johannes.hanika@uni-ulm.de
†alex@mental.com
‡hendrik.lensch@uni-ulm.de

ray tracing to a minimum, it is not surprising that the addition of minimal ray tracing turned out to be restrictive. The most recent ray tracing extension was profoundly described in [CFLB06]. With our technique we demonstrate how a ray tracing system can deal with the same complexity in geometry modeling and shading while adding the benefit of simple Monte Carlo-based global illumination computation using path tracing.

Key to our system is the reordering of rays to increase locality for ray tracing massive data which, in rather general settings, has been investigated before [LMW90, PKGH97, NFL07, BBS*09]. Our approach, however, directly benefits from the intrinsic data locality of the common two-level modeling approach: large surface patches in the top-level, and displacements or procedural details and complex shading at the bottom-level. This approach is particularly common in games, for example using parallax occlusion mapping [Tat05]. Ray tracing displaced primitives using tessellations [SB87], caches [PH96] or direct grid-like traversal [SSS00] has been investigated in depth, also on the GPU for geometry images [CHCH06]. The GPU can also be used to dice/tessellate Reyes patches [PO08]. In [BBLW07], an on-demand, recursive BVH traversal scheme for subdivision surfaces without displacements was introduced which is optimized for ray packets. Acceleration structures for ray tracing have been build in complexity below $\mathcal{O}(N \log N)$ before [HMF07]. In Section 3.2 we describe a simple method to explicitly construct a hardware-friendly acceleration structure in linear time.

In our two-level approach rays are reordered, grouping active rays which potentially intersect the same patch. Similar to the approach taken in the Kilauea render system [KS02] the resulting *(ray, patch)* lists can then be processed in parallel without caching strategies.

The Razor architecture [SMD*06] was designed to alleviate similar problems of ray tracing, to obtain better data access and computation patters. It uses current results of that time to accelerate ray tracing for the processors available by then. Today, cache lines (and fetches) get larger and data parallelism is getting wider, GPUs being the extreme example. Thus, linear memory access has become more important, and simple streaming of large blocks is often more efficient than highly recursive tree traversals and on-demand builds with a lot of branches. The Razor system does neither use displacements or pluggable artist-driven geometry shaders, so it is not optimized to avoid these computations, nor does it perform well for the excessive level of detail needed for production. To implement level of detail, the authors use a set of pairs of kd-trees for every two adjacent levels of detail, which are merged together in one acceleration structure. Additionally, at the lowest level, the vertices are stored in a 5x5 grid, which is created on-demand and traversed as in [SSS00]. Our method on the other hand comes along with two levels of hierarchy, has implicit levels of detail (in the upper levels of the bottom-level BVH), and can be diced, displaced and built with improved memory access and data parallelism.

An impressive, cache-based system was introduced by Pharr et al. [PH96, PKGH97]. It relies on a set of different caches for the rays, geometry and textures. Our ray reordering technique is simpler and transparently increases locality for rays, textures, BRDF data and geometry at the same time. Additionally, as soon as the required cache sizes get too large, caching did not perform well in our experiments.

Level of detail (LOD) has been added to both Reyes [CHPR07] and ray tracing architectures, e.g. using multi-resolution meshes [SMD*06] or simplification [YM06]. For ray tracing, the choice of LOD is commonly based on ray differentials [Ige99]. We show that in the case of our architecture, simpler mechanisms may be used.

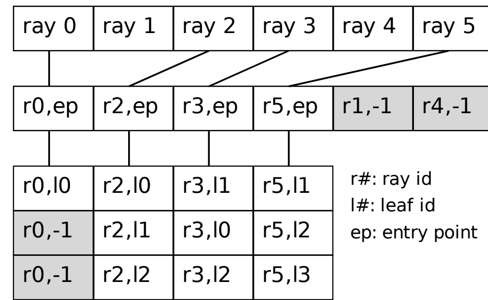


Figure 2: The buffers used to sort the rays. Top: main buffer holding the actual ray structs, containing information such as hit distance, normal, ray origin, reciprocal direction. This buffer is not sorted and can be used to derive pixel indices. After one iteration of QBVH intersection, the second buffer is filled in parallel with entry points and all inactive rays are removed. Finally, all patches with a possible intersection on the way are stored in the third buffer. In this example, if another ray terminates, enough memory will become available for each of the three remaining rays to store one more intersection in order to tackle a larger depth complexity, four in this case.

3 A Two-Level Ray Tracing Hierarchy

Our rendering system follows the two-level modeling approach commonly applied by artists who create coarse geometry using free form surfaces and then refine it by adding geometric detail and complex shaders.

In order to minimize the number of dicing operations we introduce an active ray buffer. Directly after traversing the top-level hierarchy, all potential intersections (maximum N per ray per iteration) are sorted by patch ID. Then the geometry and the shaders of each patch are prepared only once for this iteration. Lastly, all rays associated with the patch are now processed in one block of computation. As new rays might be generated due to recursive ray tracing the loop of top-level traversal, sorting, tessellation and bottom-level processing is iterated as needed. Finally, the result of the first hit point is added to the accumulation buffer.

Partitioning the computation this way greatly facilitates parallelization in each of the four processing steps. Even for global illumination computations where rays are typically incoherent after the first bounce, the explicit sorting step maximizes coherence for further processing.

3.1 Top-Level Hierarchy

The top-level hierarchy is represented by a Quad-BVH (QBVH or mBVH) [DHK08, EG08], whose leaves are the conservative bounding boxes of single patches. Ray tracing starts by generating rays and storing them in an active ray buffer. The traversal of the top-level hierarchy can then be executed in parallel by partitioning the ray buffer.

Rather than computing the first intersection directly, we gather N intersection candidates per ray. Each potential intersection with a leaf bounding box is recorded in pairs of the form (`rayid`, `leafid`). In an optimal case all possible intersections would fit into this buffer. Given a number R of rays and a memory size M, N is proportional to M/R. The intersection candidates are then sorted by `leafid` in order to group all rays intersecting the same leaf, thereby reducing multiple accesses to the same leaf node. This approach may resemble [NFL07], however, specific details are not disclosed in their work and only simulated memory traffic statistics are provided.

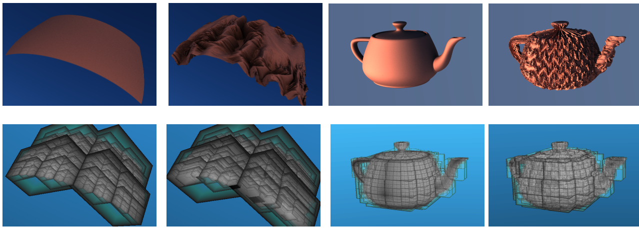


Figure 3: The top row shows a surface patch and the teapot without (left) and with displacement mapping (right). In the bottom row the bounding volume hierarchies implied by the micro-polygon array topology are visualized by rendering them transparently and darkening their contours. Differences between the hierarchies are difficult to spot, which indicates that reasonable displacement does not much affect the efficiency of the implied acceleration data structure.

For each `leafid` in the array, the leaf object is tessellated and the rays corresponding to the `leafid` are traced through the leaf object (see Section 3.2). In a parallel implementation each thread picks the next `leafid` as a task. Writing back intersection results to rays is either serialized by implementing a few locks for larger blocks of rays or, more efficiently, by writing the ray intersections to small buffers for each thread, which are synchronized at the end.

We conservatively determine for which rays the closest intersection has already been found by comparison against the following patch bounding box. These rays are terminated.

Once all rays are intersected, the top-level traversal is continued for all non-terminated rays using the last `leafid` along the ray direction as an entry point. These entry points have been stored explicitly per ray in an additional buffer (see Figure 2) because the original order of the (`rayid`, `leafid`) array is destroyed during reordering. Since traversal is ordered by ray direction, it is always clear which children to process next after the entry point, when stepping up in the hierarchy.

As the resulting number R' of non-terminated rays is typically significantly smaller than R, the next iteration can handle more potential intersections $N' \sim M/R'$, fully reusing the allocated buffers. This way, the process does not have to be repeated often, as the depth complexity of most scenes (the forest scene in Figure 1 has an overdraw of about 200) is reached quickly.

This scheme enables two more optimizations. First, in the presence of shaders, which require to access large memory blocks (such as measured BRDF data), many rays intersecting the respective surface will have an early out event at the same time and thus the memory does not have to be accessed several times. Second, to further reduce the need for repeated dicing over generations of rays, the early termination event can be used to shade a terminated block of rays, and spawn new ray directions, which can directly be intersected with the already diced originating patch and then be re-injected into the top-level traversal.

3.2 Bottom-Level Hierarchy

After the patches which might intersect a set of rays have been found by the top-level hierarchy, they have to be diced and displaced, evaluating geometry shaders. The resulting micro-polygons are stored in the micro-polygon buffer, which represents $2^m \times 2^m$ micro-polygons as a two-dimensional array of $(2^m+1) \times (2^m+1)$ vertices, where each four adjacent vertices define one micro-polygon.

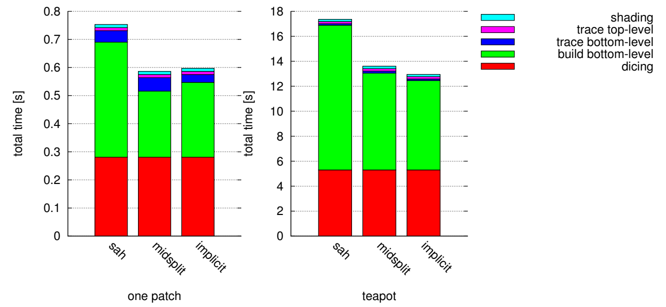


Figure 4: Timings comparing bottom-level construction strategies. On the left, a full SAH build of the micropolygon BVH is done, in the middle, a spatial median was used as split plane candidate, on the right is implicit construction. The one-patch scene and the teapot is as is Figure 3 (displaced version). These timings have been taken for primary rays only.

Surface patches must implement a tessellation method, that computes the micro-polygon vertices by either sampling or subdividing a surface patch, applies trimming and displacement, and stores interpolated (s,t) texture coordinates. Vertices are displaced along interpolated per-vertex displacement normals. To avoid holes between adjacent patches with different level of detail, conservative bounding boxes are needed for coarser tessellations, i.e. coarse displacements have to span all the possible range of the finer ones. This is done by min-max MIP maps on textures and interval arithmetic on procedural noise and patch geometry. Afterwards a loop over all micro-polygons evaluates whether or not the micro-polygon is clipped or trimmed. Unless the micro-polygon is discarded, its bounding box, color from texture, and normal by vertex differences are computed and stored.

Such a tessellation method must be aware of the resolution of the micro-polygon buffer. In case of insufficient resolution, surface patches must be split and the parts have to be processed separately.

3.2.1 Construction in Linear Time

The number of $4^m = 2^m \times 2^m$ micro-polygons and their topology suggest using a complete quad-tree of axis-aligned bounding boxes as acceleration hierarchy for ray tracing.

Constructing the bottom-level hierarchy starts by determining the conservative bounding boxes for each of the 4^m micro-polygons by calling the tessellation method. The bounding volumes of the inner nodes of the hierarchy are updated in a bottom-up manner using min-max MIP maps (similar to [CHCH06]). Trimming is implemented by marking bounding boxes as empty. They do not need to update their parent boxes and can also be handled transparently during ray traversal. Since the memory for the micro-polygon buffer data structure is allocated once for the whole rendering process, we always store the complete tree and do not compress empty bounding boxes.

Although, in general, complete trees for ray tracing cannot be recommended [Wäc08, Sec. 2.4.1], this concept is very appropriate for tessellated surface patches: Unless the patch is overly curved or extremely displaced, the array topology very well represents spatial proximity as illustrated in Figure 3 and results in fast ray tracing (see Figure 4).

While the construction time for a spatial acceleration structure typically is $\mathcal{O}(n \log n)$ in the number of triangles [WH06, Wal07], our bottom-up construction of the complete quad-tree is linear in the

number of nodes $\sum_{i=0}^{m} 4^i \in \mathcal{O}(4^m)$ and thus linear in the number of micro-polygons of one surface patch. In contrast to [HMF07], this can be done without an explicit input hierarchy and in a non-recursive manner, thus featuring a better memory access pattern.

Rays are then intersected with the acceleration structure using single ray traversal. Improved memory access by tracing ray packets did not pay off at this stage, as even these pre-sorted rays for one patch are very incoherent with regard to traversing the bottom-level hierarchy in the case of path tracing. We tessellate down to sub-pixel size and use the boxes of the leaf nodes directly as geometry, as this accuracy is sufficient [DK06].

In order to assess the quality of the implicit BVH construction for patches, we tested two very simple scenes, to avoid the effect of a complicated top-level hierarchy in the timing figures. In Figure 4, timings are plotted for a scene containing only one patch, and the displaced teapot scene (see Figure 3). The bottom-level ray tracing time can be slightly improved when using a general surface area heuristic (SAH) [GS87] when constructing the acceleration structure for the teapot (0.119 seconds SAH vs. 0.121 seconds implicit). For the simple one-patch scene, our implicit tree can even be ray traced faster (0.041 seconds SAH vs. 0.028 seconds implicit). This might also be due to the fact that our bottom-level QBVH traversal implementation exploits the special memory layout of the implicit BVH, using skip lists. It also explains the difference to the ray tracing time of the midsplit tree (0.048 seconds), which results in quite similar topology.

3.3 Level of Detail

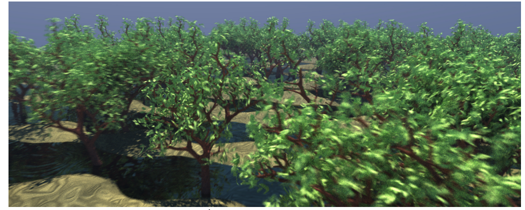
While it is common understanding that the availability of different levels of detail can vastly enhance rendering efficiency, care needs to be taken in order to avoid rendering artifacts due to the approximative nature of simplifications.

The level of detail is selected by choosing the resolution parameter m (see Figure 5) from Section 3.2 as the smallest m such that $4^m \geq R/\text{spp}$, where R is the number of rays that intersect the axis-aligned bounding box of the patch under consideration and spp is the number of samples per pixel. In order to ameliorate the self-intersection problem, secondary rays are offset by $\varepsilon = l/2^m$ along normal direction, where l is the length of the longest side of the bounding box of the actually intersected patch.

The intuition behind the selection heuristic is simple: Regions with high ray density (for example regions traversed by a bundle of specularly reflected rays) require a finer level of detail as compared to regions with less rays (as for example after diffuse reflection).

The selection heuristic does not guarantee that a ray intersects adjacent patches at the same level of detail. We therefore require all bounding boxes to be conservative. In our case this is guaranteed by the min-max MIP maps from Section 3.2.1, the use of interval arithmetic on the noise function, and the convex hull property of the control polygon of the Bézier patches. If now adjacent patches share an identical boundary, bounding boxes of adjacent patches at different levels of detail are guaranteed to touch at least and overlap most of the time. As a consequence, using intersections with the bounding boxes of the bottom level hierarchy instead of intersections with micropolygons guarantees hole free rendering.

While this method is simpler than other state of the art techniques like for example stitching together adjacent geometry [CFLB06, Sec. 6.6], it requires to select a sufficiently fine level of detail such that the resulting hole free approximation of the surfaces by boxes remains invisible [DK06].



	motion blur	no motion blur
dice [s]	211.06	104.10
bottom-level [s]	86.90	56.50
top-level [s]	27.70	24.76
shade [s]	25.20	25.36
sort [s]	34.84	28.82
#diced patches	1497633	1313136
total time [s]	179.0	133.0

Table 1: Timings for the forest scene with exaggerated motion blur on a Core 2 Quad. Motion blur results in a slowdown of a factor of two for the dicing stage, and micro-polygon intersection is slightly slower. As more patches have to be diced in the presence of motion blur, also the sorting time is increased. All times are total times, except dice and bottom-level times, which are accumulated over all four cores.

For directly visible geometry the selection heuristic results in marginally smaller than pixel-sized boxes, which reliably avoids level of detail popping artifacts during animation. However, if for example a patch only partially overlaps the viewport, then the visible part of the patch will have a much higher ray density as compared to what is determined by the selection heuristic. In a similar way, shading differences due to changing level of detail may become visible for secondary effects as for example self-shadowing.

While the selection heuristic rarely does not determine a sufficiently fine level of detail, ray differentials [Ige99] provide a widely used alternative and are easy to approximate in our system as all rays of a generation are traced at once (see Section 3.5). This allows for selecting the level of detail depending on the smallest distance between individual rays and complementing the coarse levels with directional opacity information as in [LBBS08], or using frequency domain filtering [HSRG07].

Because the bottom-level acceleration structure is a complete quadtree, the upper levels always represent the coarser levels of detail. Shading information such as color from texture, uv coordinates, and normals from vertex differences can be filtered on demand and rays can be terminated at individual levels of detail. Note that this will result in a slightly more complicated memory access pattern.

3.4 Motion Blur

Motion approximated by linear splines is standard in production (see e.g. [CFLB06, Sec.6.3]). Given the instants $t_0 < t_1 < \cdots < t_n$ defining the time intervals $[t_i, t_{i+1})$, tracing a ray at time $t \in [t_i, t_{i+1})$ is accomplished by instancing two micro-polygon buffers, one at time t_i and one at time t_{i+1}. The actual bounding boxes and micro-polygons used during ray traversal then are determined by linear interpolation. We use this method for the bottom-level hierarchy.

Concerning the top-level hierarchy, the same principles can be applied. However, due to the cost to construct the hierarchy, we chose to use only one hierarchy based on bounding boxes conservatively covering the whole time interval $[t_0, t_n)$. See Table 1 for a comparison of render times with and without motion blur.

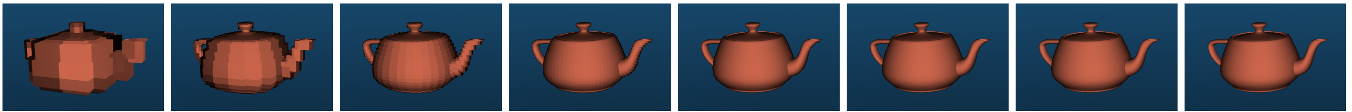


Figure 5: Illustration of the surface approximation by selecting the level of detail $m = 1,2,\ldots,8$ (from left to right).

dicing	1000 trees	100 trees
cache 10	1,897,385	1,161,468
cache 100	943,669	772,491
cache 1000	825,808	606,371
reordering	482,405	354,534

Table 2: This table shows how often patches have to be diced using a cache with 10, 100 and 1000 patches and our reordering method. Numbers are acquired using top-level traversal (i.e. independent of LOD) for the two forest scenes with motion blur at $1920 \times 768 \times 64$ rays. The reordering method only requires to store one diced patch per thread.

3.5 Tracing Rays in Groups and by Generation

Physically-based rendering requires a lot of rays to be traced. This number is typically too large to fit the required ray buffer into main memory. Also, at the beginning, not all rays are known. Some effects (such as soft shadows, ambient occlusion, reflections and so on) require several passes to be rendered, i.e. another generation, or wave, of rays to be shot.

There are several choices, which balance depth complexity, re-dicing, and memory requirements:

- Re-inject rays as needed after an early termination event. This is done by replacing the terminated ray by a newly spawned one, instead of removing it from the buffer. This will always utilize the ray buffer well and use the (`rayid`, `leafid`) buffer for new rays rather than to tackle depth complexity (see Section 3.1)
- Group rays by generation. This fixes the memory requirements for this wave of rays, but suffers from re-dicing for each pass.
- Tile the screen. This can exploit some locality for first generation lens connection rays, but as rays quickly become divergent, re-dicing is as bad as in the previous variant.

Our current implementation uses the second approach. In general it is most efficient to trace as many rays as possible (i.e. fit into main memory) at a time.

For a simple path tracer, it is sufficient to update a single (spectral or RGB) path contribution value in the ray at each bounce. In the presence of complex reflection shaders with splitting into S sub-paths, each new ray needs to be assigned the correct weight $1/S$, but great care has to be taken not to exceed the buffer limits. A similar approach could be taken to implement ambient occlusion.

Bidirectional path tracing can be done by first tracing a wave of S paths from the sensor and T paths from the lights at the same time (resulting in $S+T$ rays at a time). After that, $S \cdot T$ connection rays have to be spawned with the respective weights, for example calculated using multiple importance sampling [VG95]. To bring the number of connection rays down to $S+T$ as well, Russian roulette based on these weights can be used.

Figure 6: A tree rendered using our architecture. If it was dumped to a triangle mesh, it would consist of around 8.4 billion triangles (micro-polygons from 12k displaced Bézier patches). Due to the on-demand procedural geometry generation and the level of detail system, our system does not even create all these, and is able to render this scene with global illumination in a few minutes.

4 RESULTS

We implemented a Monte Carlo global illumination renderer on top of our ray tracing architecture. We chose a simple path tracer with next event estimation, i.e. paths are traced from the eye and with a depth of three, additionally sampling the direct light contribution at each interaction point. Russian roulette is used to decide whether to sample the hemisphere or the light sources. This way, a maximum number of *width* $\times$ *height* $\times$ *samples per pixel* $\times$ *4* rays is traced per frame. While uncommon in movie production, this is a good demonstration of the generality of our method.

To achieve equivalent detail in a regular mesh-based renderer, the micro-polygons would have to be dumped to a triangle soup which would exceed the capacities of these rendering systems. We therefore do not show comparisons with these. We tested the system on a variety of scenes ranging from trivial (Figure 3 left, equivalent to 260k triangles) over simple (Figure 3 right, equivalent to 16 million triangles), moderately complex (Figures 1,6 and 8) to massive scenes with detail equivalent to a mesh with over 1050 billion triangles (Figure 7), and scenes using reflection shaders accessing very large measured BRDF data sets (Figure 11). The trees are procedurally generated using L-systems with procedural displacement textures for the patches. The rest of the scenes is modeled in Bézier patches.

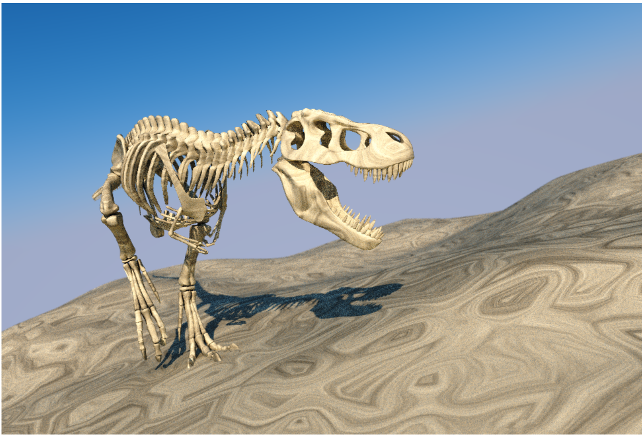


Figure 8: A dinosaur (taken out of the natural history museum from the lighting challenge site and converted to Bézier patches using vertex normals), with 56k patches. On the left, around 11 million micro-polygons out of 59 billion potential have been created and rendered using path tracing in 18 seconds on a Core 2 Quad. On the right, a noisy displacement has been applied, resulting in about 34 million created micro-polygons and an increased render time of 37 seconds. Both images are rendered at $960 \times 640 \times 16$ samples. Some of the time is spent in the (intentionally) expensive procedural displacement texture.

Figure 7: Stress test: this forest consists of two million patches, which would need a triangle mesh equivalent of more than 1050 billion triangles if fully tessellated. Fully path traced with motion blur, the image renders at $1920 \times 768 \times 16$ samples in 2:24 minutes on a four core machine.

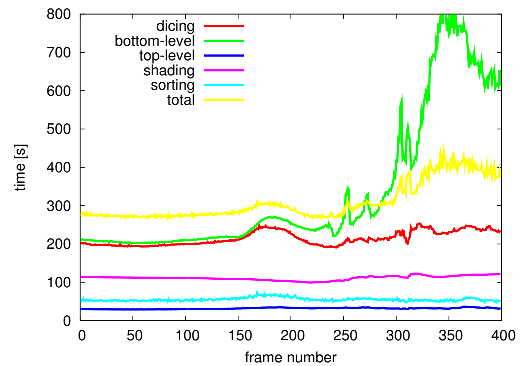


Figure 9: Statistics for the video where Figure 1 are still frames from. These numbers have been generated on a Core 2 Quad, using 32 samples per pixel in 1920×768 resolution, path traced up to path depth of three (plus evaluation of direct light at each bounce). Again all times are total times, except dice and bottom-level times, which are accumulated over all four cores. Near the end, the camera closes up to a branch, so one patch has to be tessellated very finely (the maximum $m = 10$ is reached).

As all available rays are intersected with the scene at once before shading is started, complex reflection shaders benefit from this deferred shading architecture. If a second sorting step is inserted after all ray intersections have been found, even one single data load operation per bounce can be guaranteed. This is necessary, if not all used BRDF data sets fit into main memory at the time. In the case of our test scene (Figure 11), this was not necessary, but the time spent to sort the ray buffer can almost be neglected compared to the time spent in shading (less than 10 seconds compared to 263 seconds for shading $960 \times 640 \times 64$ rays).

We compare our ray reordering to a caching based system, similar to [PH96]. Our results in Table 2 indicate that for highly complex scenes caches need to be very large to be efficient. With our reordering method, they are not necessary, and the implementation becomes thus simpler than [PKGH97].

The table indicates that significantly more than 1000 cache lines, each representing one tessellated patch, are necessary in order to reduce the number of dicing steps to those achieved with our reordering. Besides the top-level hierarchy, our scheme requires a fixed memory footprint independent of geometry complexity. For path tracing at 1920x768x64 rays, our implementation needs 6120 MB for representing the ray buffers (68 bytes per ray: $3\times$ float position, $3\times$ float direction, $3\times$ float normal, $1\times$ float hit distance, $3\times$ float color, $3\times$ float path tracing weight RGB, $1\times$ int16 shader index and $1\times$ int16 additional flags to mark e.g. shadow rays). To represent the full tree of a single diced patch at LOD $m = 10$ (1024×1024 micro-quads with bounding boxes, color, texture coordinates and normals at two time instances and every level of detail), 120 MB are required. Our approach requires a single buffer (corresponding to a single cache line), while the cache will grow linearly with the number of cache lines.

For an example how depth complexity is handled by the limited `(rayid, leafid)` buffer (see Section 3.1), see Table 3. It can be seen that even for scenes with very large overdraw, only few iterations are required. As some rays terminate, the buffer is quickly available for the remaining rays to store many more intersection candidates in the next iteration.

To get an impression of the impact of LOD, see Figure 10. The rendering times are dominated by dicing, so the graph shows only timings for top- and bottom-level traversal and shading. As expected, dicing and bottom-level tree construction seems to be linear

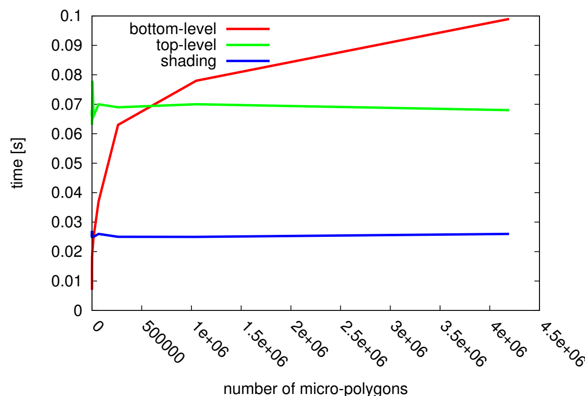


Figure 10: Rendering time is determined by the programmable parts of the system, i.e. shading and surface patch tessellation (tessellation time is linear and takes over 100 seconds for $4 \cdot 10^6$ micro-polygons and is therefore omitted in the plot). Timings are obtained for the displaced teapot example (Figure 3).

eye		bounce 1		bounce 2		bounce 3	
R	N	R	N	R	N	R	N
23592960	8	21589447	8	15585868	12	10052438	18
19009592	9	16419230	11	10886646	17	6707465	28
5840016	32	4549352	41	1529033	100	298166	100
1115814	100	406945	100	14093	100	82	100
1062	100	17	100	-	-	-	-
23592960	8	20674988	9	11985885	15	7311443	25
11479199	16	11145131	16	4617303	40	1775062	100
2110951	89	1241408	100	59364	100	204	100
7509	100	333	100	-	-	-	-

Table 3: Tackling depth complexity: when rays are terminated early due to sorted BVH traversal, the memory can be used to store more patch intersections N for the remaining rays R. This is an example for the forest in Figure 7 (top table) and Figure 1 (bottom table) for the four waves of path tracing with next event estimation. N is clipped to 100 to avoid excessive fragmentation of memory for simple cases.

and tracing bottom-level rays is about logarithmic. Top-level traversal does not change in this graph, as the top-level hierarchy is not affected by the LOD changes. Obviously a lot of time can be saved by reducing the number of micro-polygons per patch.

As illustrated in Table 4, our system is very efficient due to choosing the appropriate LOD and avoiding dicing for occluded patches altogether. The comparison here is carried out between actually created micro-polygons and the number of polygons in a triangle mesh with equivalent detail (maximum LOD $m = 10$). Note that this number is not overly large, this LOD is also chosen for some patches by the algorithm and becomes especially necessary for heavily displaced patches. The figures show that our system can robustly handle a vast amount of geometry, which surpasses the complexity demonstrated by the Razor system [SMD*06]. Also, we do not need to keep any diced micro-polygons which avoids the problem of flushing on-demand geometry.

5 Conclusion

We presented a ray tracing method, which is able to efficiently handle large amounts of data resulting from free form surface patches, details added by micro-polygon tessellation, and data intensive shaders. Expensive geometry and shaders (in terms of computation or memory access) are handled well due to reordering of computations which results in great data locality. Parallelization is simple as

scene	maximum	accessed
100 trees (Figure 1)	$108 \cdot 10^9$	$170 \cdot 10^6$
1000 trees (Figure 7)	$1,058 \cdot 10^9$	$317 \cdot 10^6$
dinosaur (Figure 8, left)	$59 \cdot 10^9$	$11 \cdot 10^6$
displaced dinosaur (Figure 8, right)	$59 \cdot 10^9$	$34 \cdot 10^6$

Table 4: Impact of LOD and early ray termination due to occlusion: ratio of micro-polygons actually created to a constant LOD of $m = 10$ while path tracing.

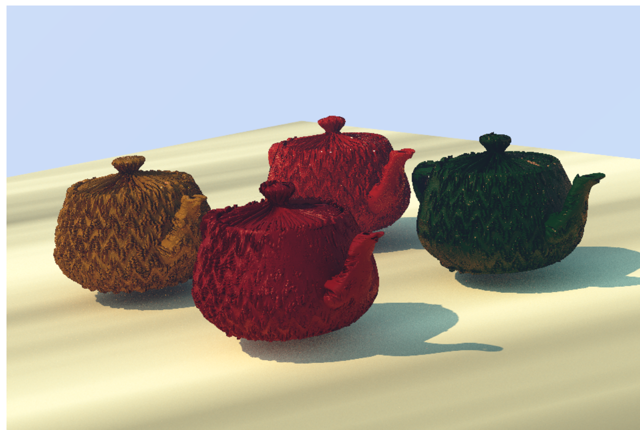


Figure 11: Four teapots rendered with measured BRDF data. One BRDF data set alone is over 300 MB large, and each teapot consists of over 16 million triangles. Thanks to reordering of shading computations, it can be guaranteed that each BRDF data set is loaded only once per bounce.

all rays traverse one phase before the next one is started. We introduced only one additional sorting step on the ray buffer, which has negligible impact on rendering time. The core of our contribution is the two-level ray tracing system with reordering, which can be easily augmented by other advanced rendering techniques, as we have demonstrated for simple LOD selection and path tracing. The same two-level approach with reordering can be combined with general global illumination algorithms or even accelerate out-of-core rendering.

There are no restrictions imposed on ray tracing. However, there are some limitations when using the method in a rendering system. First, the shading language needs a mechanism to dispatch a ray and correctly account for its contribution by the time it finishes (see Section 3.5). If rendering is based on BRDFs, this is straightforward.

Second, the presented LOD assumes good importance sampling. That is, it assumes that if rays are diverging, the contributing radiance is low frequency. It will thus result in blocky shadows if sampling a small direct light source over the hemisphere instead of the geometry. If this is recognized by the rendering algorithm, it can be used as a feature to speed up low-frequency indirect illumination.

In future work, the method can be complemented by specialized rendering algorithms which better exploit its strengths by reflecting the two-level nature, such as local high-frequency ambient occlusion inside a diced patch together with a far-field approximation for global illumination, similar to [KK04, AFO05].

Acknowledgements

The first author wishes to thank Holger Dammertz for countless productive discussions, Leonhard Grünschloß for proof reading, and Matthias Hullin for the measured BRDF data sets. The authors would like to thank the mental images GmbH for support and funding of this research.

References

[AFO05] ARIKAN O., FORSYTH D., O'BRIEN J.: Fast and detailed approximate global illumination by irradiance decomposition. *ACM Transactions on Graphics (Proc. SIGGRAPH 2005)* (2005), 1108–1114.

[BBLW07] BENTHIN C., BOULOS S., LACEWELL D., WALD I.: *Packet-based Ray Tracing of Catmull-Clark Subdivision Surfaces*. Tech. Rep. UUSCI-2007-011, SCI Institute, University of Utah, 2007.

[BBS*09] BUDGE B., BERNARDIN T., STUART J., SENGUPTA S., JOY K., OWENS J.: Out-of-core data management for path tracing on hybrid resources. In *Computer Graphics Forum (Proc. of Eurographics 2009)* (2009), pp. 385–396.

[CCC87] COOK R., CARPENTER L., CATMULL E.: The Reyes image rendering architecture. *Computer Graphics (Proc. SIGGRAPH '87)* (1987), 95–102.

[CFLB06] CHRISTENSEN P., FONG J., LAUR D., BATALI D.: Ray tracing for the movie 'Cars'. In *Proc. 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 73–78.

[CHCH06] CARR N., HOBEROCK J., CRANE K., HART J.: Fast GPU ray tracing of dynamic meshes using geometry images. In *GI '06: Proceedings of the 2006 conference on Graphics interface* (2006), pp. 203–209.

[CHPR07] COOK R., HALSTEAD J., PLANCK M., RYU D.: Stochastic simplification of aggregate detail. *ACM Transactions on Graphics 26*, 3 (2007), 79.

[DHK08] DAMMERTZ H., HANIKA J., KELLER A.: Shallow bounding volume hierarchies for fast SIMD ray tracing of incoherent rays. In *Computer Graphics Forum (Proc. 19th Eurographics Symposium on Rendering)* (2008), pp. 1225–1234.

[DK06] DAMMERTZ H., KELLER A.: Improving ray tracing precision by world space intersection computation. In *Proc. 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 25–32.

[EG08] ERNST M., GREINER G.: Multi bounding volume hierarchies. In *Proc. 2008 IEEE/EG Symposium on Interactive Ray Tracing* (2008), pp. 35–40.

[Gri09] GRITZ L.: Production perspectives on high performance graphics. Keynote Talk at the High Performance Graphics conference, 2009.

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

[HL90] HANRAHAN P., LAWSON J.: A language for shading and lighting calculations. *Computer Graphics (Proc. SIGGRAPH '90)* (1990), 289–298.

[HMF07] HUNT W., MARK W., FUSSELL D.: Fast and lazy build of acceleration structures from scene hierarchies. In *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 47–54.

[HSRG07] HAN C., SUN B., RAMAMOORTHI R., GRINSPUN E.: Frequency domain normal map filtering. *ACM Transactions on Graphics (Proc. SIGGRAPH 2007)* (2007), 28.

[Ige99] IGEHY H.: Tracing ray differentials. *Proc. of SIGGRAPH '99* (1999), 179–186.

[KK04] KOLLIG T., KELLER A.: Illumination in the presence of weak singularities. In *Proc. Monte Carlo and Quasi-Monte Carlo Methods 2002*. Springer, 2004, pp. 245–257.

[KS02] KATO T., SAITO J.: Kilauea parallel global illumination renderer. In *Fourth Eurographics Workshop on Parallel Graphics and Visualization* (2002), pp. 7–13.

[LBBS08] LACEWELL D., BURLEY B., BOULOS S., SHIRLEY P.: Raytracing prefiltered occlusion for aggregate geometry. In *IEEE Symposium on Interactive Raytracing 2008* (2008), pp. 19–26.

[LMW90] LAMPARTER B., MÜLLER H., WINCKLER J.: *The Ray-z-Buffer—An Approach for Ray Tracing Arbitrarily Large Scenes*. Tech. rep., Albert-Ludwigs University at Freiburg, 1990.

[LV00] LOKOVIC T., VEACH E.: Deep shadow maps. *Proc. of ACM SIGGRAPH 2000* (2000), 385–392.

[LYM07] LAUTERBACH C., YOON S.-E., MANOCHA D.: Ray-strips: A compact mesh representation for interactive ray tracing. In *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 19–26.

[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.

[NFL07] NAVRÁTIL P., FUSSELL D., LIN C.: Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 95–104.

[NRH*77] NICODEMUS F., RICHMOND J., HSIA J., GINSBERG I., LIMPERIS T.: Geometrical considerations and nomenclature for reflectance. *Final Report, National Bureau of Standards, Washington, DC. Inst. for Basic Standards* (1977).

[PH96] PHARR M., HANRAHAN P.: Geometry caching for ray-tracing displacement maps. In *Eurographics Rendering Workshop 1996* (1996), pp. 31–40.

[PKGH97] PHARR M., KOLB C., GERSHBEIN R., HANRAHAN P.: Rendering complex scenes with memory-coherent ray tracing. *Proc. of SIGGRAPH '97* (1997), 101–108.

[PO08] PATNEY A., OWENS J.: Real-time Reyes-style adaptive surface subdivision. *SIGGRAPH Asia '08: ACM SIGGRAPH Asia 2008 papers* (2008), 143:1–143:8.

[SB87] SNYDER J., BARR A.: Ray tracing complex models containing surface tessellations. *Computer Graphics (Proc. SIGGRAPH '87)* (1987), 119–128.

[SBB*06] STEPHENS A., BOULOS S., BIGLER J., WALD I., PARKER S.: An application of scalable massive model interaction using shared memory systems. In *Proceedings of the 2006 Eurographics Symposium on Parallel Graphics and Visualization* (2006), pp. 19–26.

[SMD*06] STOLL G., MARK W., DJEU P., WANG R., ELHASSAN I.: *Razor: An architecture for dynamic multiresolution ray tracing*. Technical report 06-21, Department of Computer Science, University of Texas at Austin, 2006.

[SSS00] SMITS B., SHIRLEY P., STARK M.: Direct ray tracing of displacement mapped triangles. In *Proc. Eurographics Workshop on Rendering Techniques 2000* (2000), pp. 307–318.

[Tat05] TATARCHUK N.: Practical dynamic parallax occlusion mapping. *ACM SIGGRAPH 2005 Sketches* (2005), 106.

[VG95] VEACH E., GUIBAS L.: Optimally combining sampling techniques for Monte Carlo rendering. *Proc. of SIGGRAPH '95* (1995), 419–428.

[Wäc08] WÄCHTER C.: *Quasi-Monte Carlo Light Transport Simulation by Efficient Ray Tracing*. PhD thesis, Universität Ulm, 2008.

[Wal07] WALD I.: On fast construction of SAH based bounding volume hierarchies. In *Proc. 2007 IEEE/EG Symposium on Interactive Ray Tracing* (2007), pp. 33–40.

[WBWS01] WALD I., BENTHIN C., WAGNER M., SLUSALLEK P.: Interactive rendering with coherent ray tracing. In *Computer Graphics Forum (Proc. Eurographics 2001)* (2001), pp. 153–164.

[WH06] WALD I., HAVRAN V.: On building fast *k*d-trees for ray tracing, and on doing that in $\mathcal{O}(n \log n)$. In *Proc. 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 18–20.

[YM06] YOON S.-E., MANOCHA D.: R-LODs: fast LOD-based ray tracing of massive models. *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches* (2006), 67.