Occlusion Culling in Batched Ray Traversal

Student: Mathijs Molenaar (Student ID: 5958970)
Supervisors: Jacco Bikker, Elmar Eisemann

December 20, 2018

Abstract

In this work we study ray traversal of scenes that do not fit in system memory. More precisely, we continue with the work of [PKGH97] on batched ray traversal. We propose to store a simplified representation of the scene's geometry to quickly cull rays that do not hit anything. This reduces the number of disk accesses and our results show that it can more than double rendering performance.

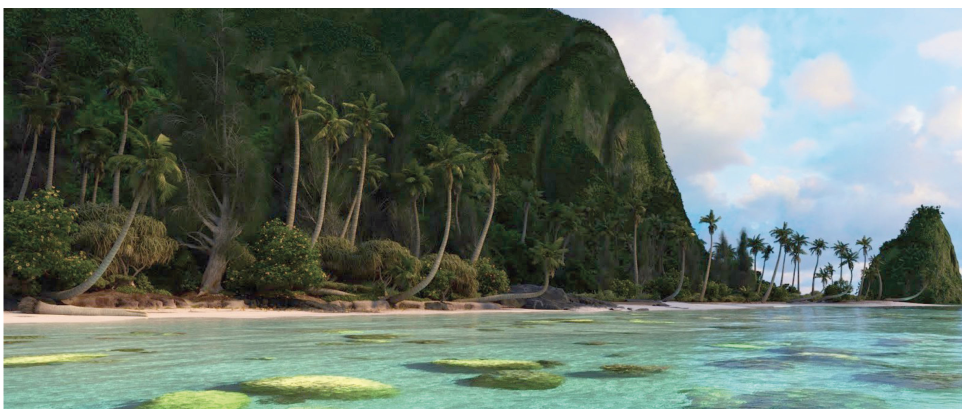


Figure 1: The Island scene by Walt Disney Animation Studios requires a lot of memory to render.

1. Introduction

Rendering large scenes is an ongoing challenge in the computer graphics industry. Real-time visualization of complex meshes and photo realistic rendering of large scenes on machines without sufficient system memory remains an open problem. The movie industry is among the most prominent industries in need of rendering large scenes with accurate global illumination.

This is caused by recent advances in compute power having set in motion a revolution in the movie industry, replacing rasterization (Reyes) based renderers [CCC87] with path traced renderers [Pha18]. While rasterization based production renderers have supported out-of-core rendering for a long time, this is not the case for the newly developed path traced renderers.

The recent availability of high performance ray tracing hardware on graphics cards (Nvidia RTX) amplifies the need for efficient out-of-core ray tracing. Compute based GPU ray tracing has been competitive with CPUs in terms of performance for some time now, but the movie industry has mostly shunned away from GPU rendering because of the limited device-local memory available on graphics cards [Pha18]. Graphics card memory is designed to be high bandwidth, requiring that memory chips are physically close to the GPU. This limitation means that graphics card memory capacity will always be limited compared to system memory.

The biggest challenge in out-of-core production rendering is that path tracing results in incoherent (random) memory-access patterns which result in high cache miss rates. This can have a big impact on performance because of the large bandwidth and latency disparity between system memory and (solid-state) disk storage.

In this paper we will discuss and improve batched ray traversal which was first introduced in [PKGH97] by reducing wasteful work through the introduction of "occlusion culling". Batched ray traversal is a technique that can be used in both in-core, out-of-core and distributed rendering to improve the memory coherence of ray tracing. In this paper we add occlusion culling to batched ray traversal and examine it's effects on out-of-core rendering.

2. Related Work

Large scenes may be visualized by replacing distant geometry by proxies as to reduce geometric complexity. Walt et al. [WDS05] present a distributed ray tracing system that renders proxy geometry on cache misses. A similar concept is used by Crassin et al. [CNLE09] to render volumetric data sets on the GPU. Proxy geometry can also be used in a static level-of-detail system to make the scene fit into memory, as is shown by Pantaleoni et al. [PFHA10]. Similarly, Yoon et al. [YLM06] replace geometry by oriented planes when this satisfies a screen-space error function.

When visualization of the full scene complexity is desired this requires either distributed and/or out-of-core rendering. A common approach is to abstract away where data resides in a paging memory layer. Cox and Ellsworth [CE97] show that application-controlled paging can improve performance over equivalent operating system functionality for out-of-core volume rendering. DeMarle et al. [DGP04] create a shared memory layer that distributes the scene over the available render nodes, hiding the complexity of data movement.

Compared to these generalized approaches, application controlled data movement may provide more room for domain specific optimizations. Christensen et al. [CLF+03] use application controlled caching of surface tessellation to aid out-of-core traversal performance. Wald et al. [WSB01] utilize a two-level acceleration structure hierarchy to manage data movement resulting in almost linear performance scaling in distributed rendering.

A common limitation with these works is that none of them tackle the issue of incoherent rays. Monte-Carlo solutions to global illumination, such as path tracing, lead to highly incoherent memory access patterns. While these previously mentioned systems work well for coherent ray distributions, disk bandwidth will form a bottleneck for random ray distributions.

To combat ray divergence, breadth-first traversal, ray stream traversal and ray reordering schemes have been proposed. In breadth-first traversal [WGBK07, GR08, Tsa09, RGD09] a collection of rays is traversed breadth-first through the acceleration (tree) structure. This ensures that each node in the hierarchy is loaded at most one time, at the cost of using the same traversal order for all rays. Ray stream traversal techniques [BAM14, FLPE15] allow for (approximate) front-to-back traversal by potentially visiting nodes twice. Ray reordering schemes [ENSB13, MBK+10] aim to sort the rays such that they are more coherent. Unlike specialized traversal algorithms they are decoupled from the acceleration structure.

2.1. Batched Ray Traversal

Batched ray traversal as proposed by Pharr et al. [PKGH97], which forms the basis of this paper, presents yet another way to extract coherence from an arbitrary distribution of rays. In order to improve coherence rays are “batched” (stored) at batching points inside the acceleration structure. When a batching point accumulates enough rays the underlying geometry and acceleration structure subtree are loaded and the rays are traversed. Unlike the previously mentioned techniques, batched ray traversal is able to extract coherence from rays passing through the same region of space even if their origins lie far apart or their directions are incoherent.

Budge et al. [BBS+09] found that this traversal architecture also lends itself well to distributed rendering. They present a path tracing renderer that uses batched ray traversal to render complex scenes on a heterogeneous set of compute devices. Son and Yoon [SY17] suggest that the scheduler may be improved by utilizing a device connectivity graph.

Navratil et al. [NFLM07] show that ray batching can also be applied to improve coherence one level up the memory hierarchy. They use batched ray traversal to reduce the number of CPU cache misses during traversal of the acceleration structure which improves performance. In the same vein, Bikker [Bik12] and Gasparian [Gas16] implement batched ray traversal for in-core path tracing using different acceleration structures.

2.2. Occlusion Culling

Occlusion culling is commonly used to improve performance in rasterized renderers. Techniques such as hierarchical z-buffers [GKM93], hierarchical occlusion maps [ZMHHI97] and incremental occlusion maps [Ail00] all utilize a conservatively estimate of the depth buffer to prevent fully occluded geometry from being rendered. Here, conservative means that the estimated depth values may never be smaller than their actual values. A common way to estimate the depth buffer is to render a strongly simplified version of the occluder geometry [e.g. Val11, SSLL14, Wih16]

Our novel idea is to apply a similar concept to ray tracing. In regular batched ray traversal, a ray is batched when traversal reaches (intersects the bounding volume of) a batching point. This entails that a ray might get batched even though it does not hit any geometry associated with that batching point. We can reduce the likelihood of this occurring by storing a simplified representation of the ge-

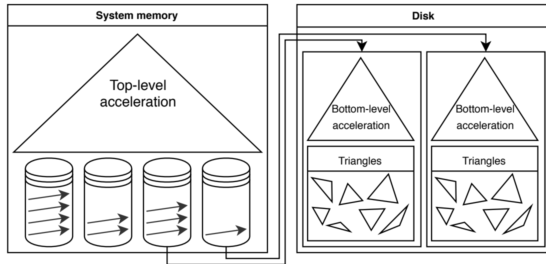


Figure 2: The two level acceleration structure hierarchy used by (out-of-core) batched ray traversal. In our system the ray batches are always stored in memory.

ometry at each batching point and intersecting rays against that representation before batching, akin to occlusion culling.

3. Overview

In this section we will give a more detailed overview of the techniques and data structures used in this research.

3.1. Batched Ray Traversal

Pharr et al. [PKGH97] use two levels of regular grids for batched ray traversal. The top grid is always in memory and its cells form batching points for the rays. For all geometry in a cell a second grid is created which is used to accelerate ray traversal. These "acceleration grids" are stored in a cache and may be loaded from disk when a cell (of the top level grid) needs to be traversed (Figure 2). Ray batches are also stored on disk (with a cache) to prevent them from occupying too much memory.

In later works the regular grids have been replaced by a combination of octrees [Bik12], kd-trees [NFLM07, BBS+09] and bounding volume hierarchies [Bik12, Gas16]. Different scheduling algorithms have also been introduced as to improve performance on distributed systems [BBS+09, SY17] or to limit the exponential growth in the number of rays [NFLM07] as is present in the original work by Pharr et al.

3.2. Occlusion culling

The occlusion culling methods for rasterization mentioned in the related work section require a conservative depth estimate to ensure correct operation. When using simplified geometry as occluders this entails that the simplification must be "inner conservative": it must be completely enclosed by the original geometry.

For our purposes this requirement is inverted. Any ray that hits the actual geometry must also hit the simplified representation. This means that the simplification must fully enclose the actual geometry. This can be attained by either creating a conservative geometric simplification or by storing a volume that fully contains the geometry.

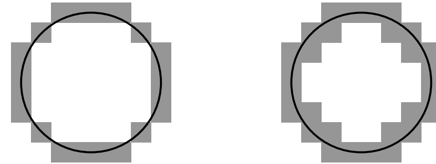


Figure 3: Thin- (left) and conservative (right) voxelization of a circle in 2D.

We have chosen for the latter by creating a voxelization of the geometry at each batching point as a means of simplification. Note that for correct operation the voxelization should be fully conservative (26-separating); that is: any cell touched by a triangle should be marked as part of the volume. This is a stricter requirement than thin voxelization (also referred to as 6-separating) which is often used in the space of computer graphics. Thin voxelization ensures that the resulting model is watertight but it does not make the guarantee that the resulting volume fully contains the geometry's surface. Figure 3 shows the difference between thin- and conservative voxelization in 2D although the same principles apply to 3D.

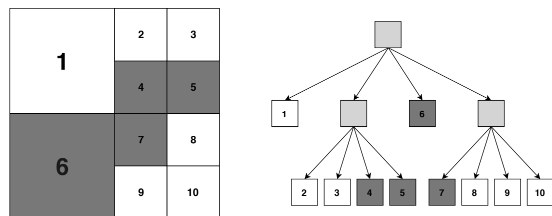


Figure 4: A sparse voxel octree saves memory by not storing homogeneous parts of the voxel grid. The illustration shows a sparse voxel quadtree for simplicity.

A voxel grid can be stored efficiently through a sparse voxel octree (SVO). Sparse voxel octrees are essentially just octrees where nodes storing uniform regions of space are not refined (Figure 4).

Although sparse voxel octrees are efficient they store redundant information, as was first discovered by Webber and Dillencourt [WD89]. Quadtrees and octrees often contain duplicate subtrees which is a waste of space. The authors suggest replacing duplicate subtrees by a single instance. In their specific case this reduced memory usage by an order of magnitude. Replacing duplicate subtrees by a single instance results in nodes with multiple parents (Figure 5). Since this breaks one of the fundamental properties of a tree (nodes always having a single parent) so we will refer to the results as Sparse Voxel Directed Acyclic Graphs (SVDAGs).

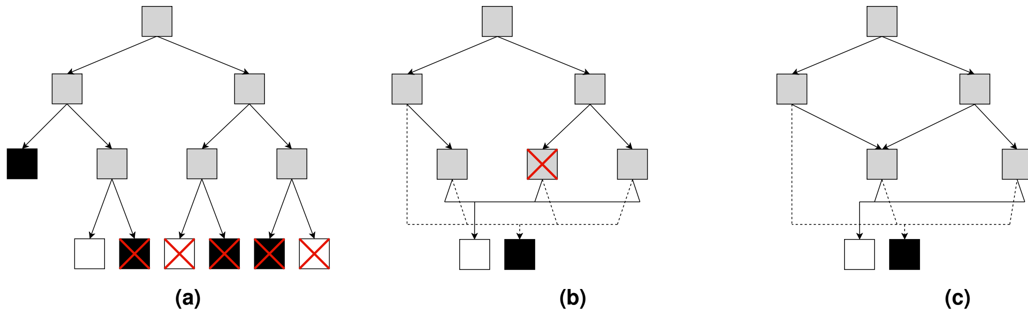


Figure 5: Compressing a sparse voxel octree into a sparse voxel directed acyclic graph, illustrated using a binary tree for clarity. Duplicate nodes are repeatedly replaced by a single instance starting from the bottom up.

An efficient way of reducing a quadtree or (sparse voxel) octree into a graph is by using bottom-up construction as was presented in [KSA13]. During a recursive traversal we can replace duplicate nodes from the bottom up by keeping a hash map of previously visited (unique) nodes. A similar process can also be applied directly to a voxel grid as is shown in [PU03]. By sharing the hash map between different SVOs we can achieve an additional memory savings over compressing each of them individually.

4. Implementation

The renderer that we use for this research was build from the ground up because, to our knowledge, there does not exist an open source batched ray traversal renderer. Although the renderer does support different material models we will only use matte Lambert shading (except for the Island scene where a transparent shader is used for the water such that the the seabed is visible). This was done to make shading as cheap as possible since we are only interested in ray traversal performance.

4.1. Batched Ray Traversal

As mentioned in the previous section, batched ray traversal accepts a plethora of different (combinations of) acceleration structures. The only requirement is that traversal of the top level structure needs to be able to store the traversal state efficiently (so that rays can be batched). For this work we have chosen to use a bounding volume hierarchy for both levels of the acceleration structure.

The top-level heirarch is a 4-wide BVH based on the work of Gasparian [Gas16]. By using a 4-wide BVH, ray/bounding box intersections can be performed in parallel using SSE (SIMD) instructions. Regular (depth-first) BVH traversal requires maintaining a stack of ancestors of the currently visited node. Storing the traversal stack along with each ray would require too much memory. So instead, the author suggests to store pointers to all ancestor (up to the root node) inside the BVH nodes themselves, trading traversal stack size for BVH node size. The traversal stack then only has to store 4 bits for each node to indicate which children have not been traversed yet.

We take a slightly different approach by only storing a single parent pointer in each node instead of the full list of ancestors. For a stack with a maximum depth of 8 this saves 32 bytes per BVH node (assuming 32 bit pointers and 16 byte alignment). This change means that backtracking requires multiple jumps through memory but in our testing this did not impact performance significantly. Although for out-of-core traversal this isn't as important (because of the relatively low cost of top-level traversal), we think it might be interesting to experiment with storing ancestor pointers with logarithmic steps: 1 up (parent), 2 up, 4 up, 8 up, 16 up. This might provide the best of both worlds in terms of memory usage (BVH nodes fit exactly in 2 cache lines) and the number of traversal steps required for back tracking ($O(log(N))$).

Using a BVH as the top-level acceleration structure also introduces a complication that previous works on batched ray traversal for out-of-core rendering did not have to deal with. In bounding volume hierarchies, leaf nodes may overlap which means that the first intersection found during front-to-back traversal might not be the closest intersection. Our solution is to pin geometry in memory when a ray intersects and only make the geometry evictable when all rays referring to it have been shaded or have found a closer intersection.

For the bottom-level acceleration structure we use an 8-wide BVH which is traversed using an AVX2 implementation of the wide vector single ray traversal algorithm presented in [FLP+17]. With this algorithm the stack not only stores all ancestors of the current node but also the entry distance to their axis-aligned bounding box. Each time a leaf is intersected the stack is compressed by removing nodes whose entry distance is further than the leaf intersection distance. This compression can be implemented with a single AVX512 instruction. How-

ever, our hardware only supports up to AVX2, so we use a look-up table to implement compression.

Traversal order is approximated by only sorting children based on the signs of a ray's direction vector components. The traversal orders for each of the 8 directions are precomputed and stored in the BVH nodes. During traversal this information, combined with a ray's direction vector signs, is used to sort the children into the desired order.

Supporting instancing in batched ray traversal is not something that any of the previous works mention. Of course, instancing could be supported by duplicating geometry but this would greatly increase the scene size. We first experimented with a similar idea by only storing instanced geometry once per batching point. However, this still resulted in prohibitively large data files (too large for our test system). The current implementation stores all unique geometry in a batching point (and the accompanying BVH) to a single file. Instanced geometry is stored in separate files (along with their BVHs), batching together many meshes as to not overload the file system with many small files. The cache system has been updated accordingly to store both batching point data and instanced geometry in a single LRU cache. To ensure parallelism the cache system loads data asynchronously using separate loading threads.

Creating batching points is also complicated by the inclusion of instanced geometry. We considered picking batching points such that either they have roughly equal amounts of instanced primitives or equal amounts of unique primitives. The former will result in roughly equal traversal times between batching points although batching points may vary heavily in terms of their size on disk. Alternatively, clustering based on the number of unique primitives results in batching points of roughly the same file size but with potentially large differences in traversal time when batching points contain many instanced objects. In our implementation we have chosen for the former because the latter would stress our fix to overlapping BVH leaf nodes (mentioned above).

To select the batching points we rely on the user to provide the scene as a collection of objects such that no out-of-core processing is necessary. Batching points are created by clustering objects such that they minimize the surface area heuristic. Clustering is implemented by constructing and flattening a binary BVH. To prevent large objects from causing unbalance they are split into smaller pieces using the same technique as we use for clustering (constructing and flattening a BVH over the primitives in the object).

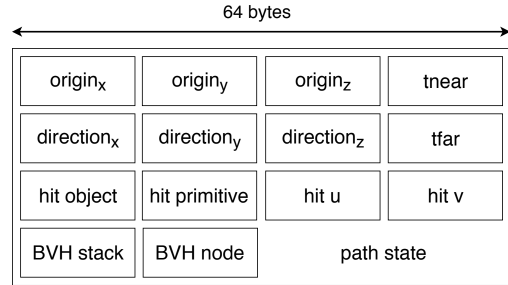


Figure 6: The data layout of a batched ray.

Ray batches are represented as a collection of fixed size block of rays (and other associated data). Batching points store a (intrusive) linked list of full blocks in addition to a single non-full block for each worker thread. Full blocks are kept separate because they are immutable which ensures that processing them is thread-safe. This design means that worker threads only have to communicate when their thread-local block gets full.

Picking a block size is a trade-off between performance and memory over-allocation. Batching a ray requires at least 56 bytes plus any additional integrator state (Figure 6). In our renderer we need 168 bytes to batch a ray so we use a relatively small block size of 8.

The scheduling algorithm we developed is inspired by the work of Bikker [Bik12] in which batching points are sorted based on the number of batched rays and then processed in descending order. The idea behind sorting is that processing starts at batching points with many rays and that missed rays will get forwarded to batching points with fewer rays. This approach has a lower overhead than a blocking scheduler such as used in [BBS+09]. Another advantage is that it provides a synchronization point which can be used to forward thread-local ray blocks to the list of full (immutable) blocks. Without this forwarding, rays could get stuck at thread-local blocks if those block would never fill up. This issue could be ignored which allows for scheduling without synchronization points but at a loss of determinism.

We found that processing all batching points each iteration does not give the system time to collect enough rays at rarely visited batching points. Therefor we only process a select number of batching points (the top 25% in terms of ray count) each iteration. A general issue with batched ray traversal is that rays can get stuck at hardly visited batching points until the end of the render when only few rays are left. Traversing these rays (and any new rays spawned by shading) will be very expensive because there are not enough

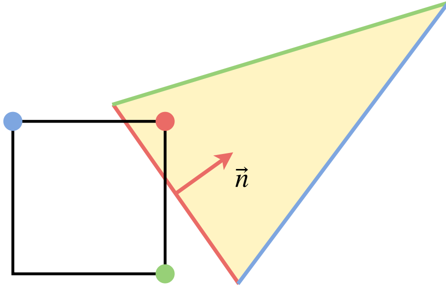


Figure 7: Voxelization: the 3 critical points with respect to the triangle's edges. In this image all critical points lie on the correct side of their respective edges.

rays in the system to hide disk latency. We try to address this by processing a small number of randomly picked batching points each iteration (10% of the batching points with at least one ray).

In our system ray batches are stored in system memory. So to limit memory usage we process at most 16 million paths at a time. New camera rays are generated by exhausting all samples of a pixel before moving to the next. The pixels are visited along a z-curve (Morton code order) to improve the coherence of primary rays.

4.2. Occlusion Culling

Like mentioned in section 3.2, our occlusion culling system is based on the traversal of SVDAGs. All geometry associated with a batching point is voxelized using the conservative voxelization algorithm presented in [SS10].

For each triangle we loop over all voxels in its bounding box and test whether they intersect the triangle. A triangle intersects when the plane on which it lies intersects with the voxel and the projections of the triangle and the voxel on the XY-, XZ- and YZ-planes overlap. The latter can be evaluated efficiently by testing for each (2D) edge function of the triangle whether the respective critical point of the (projected) voxel lies on the correct side of the edge. The critical point is the point in the projected voxel which has the highest value according to the edge-function (Figure 7). More intuitively, this is the point (corner) that lies furthest along the edge's normal vector (pointing inside the triangle).

The voxel grids are converted to sparse voxel octrees using the octree construction algorithm presented in the same paper. An octree is build from the bottom-up, creating parents for the previous level's nodes. We start by creating a list of "filled" voxels, ignoring empty ones. We take care to ensure that the voxels in the list appear in Morton order, which groups nodes belonging to the same parent. We then loop over the list, creating parent nodes which are also stored in a Morton ordered list. Nodes that span a completely filled region of space (all children are leafs) are inserted into the output list as new leafs, which ensures that fully filled regions are not refined. This process is repeated until the root of the octree is reached.

Octree nodes are stored using a "descriptor" bitmask which indicates the type of the children (empty, leaf or octree node), plus a list of child pointers (stored as 32 bit offsets into an array). This list is of variable length such that a node does not waste space on unused pointers.

Our SVDAG compression code is based on [PU03] and consists of a recursive traversal during which a separate SVDAG is created. The pseudo code for the SVDAG construction algorithm is shown in Algorithm 1. The *FIND_NODE* function uses a hash map to find matching nodes that have already been encountered. By sharing this hash map between SVOs we can eliminate duplicate subtrees across them.

ALGORITHM 1
Sparse Voxel Octree DAG compression

```
procedure COMPRESS_TREE(n)
    r ← Node()
    for 1 ≤ i ≤ 8 do
        if IS_INNER_NODE(i) then
            c ← n.children[i]
            r.children[i] ← COMPRESS_TREE(c)
        end if
    end for
    return FIND_NODE(r)
end procedure
```

SVDAGs can be traversed using any existing SVO traversal algorithm. We use the same depth-first algorithm as [KSA13], which is a simplified version of the algorithm presented in [LK11] (Listing 3 in the appendices). Like most tree traversal algorithms, a stack is used to keep track of the current voxel's ancestors.

The path taken to reach a voxel is encoded into its 3D position. Incoming rays are transformed such that the octree spans the space of $\begin{bmatrix}1 & 1 & 1\end{bmatrix}^T$ to $\begin{bmatrix}2 & 2 & 2\end{bmatrix}^T$. Keeping the dimensions between 1 and 2 allows the mantissa's of the (IEEE 754) floating point position vector components to be reinterpreted as integers which encode the child index of each ancestor with respect to its parent.

The next child to traverse is found by intersecting the ray with the axis-aligned planes passing through the current voxel's center. During traversal we keep track of the distance that the ray has traveled. By comparing this against the distances to the 3 planes we can determine which child node

Figure 8: The scenes that were used for testing rendered with our path tracer. From left to right: crown, landscape and Island.

to visit next. The ray/plane intersection tests as well as the required comparisons are implemented using SSE instructions, utilizing 3 out of the 4 available lanes.

5. Results & discussion

In this section we measure and discuss the performance impact of occlusion culling in our renderer.

5.1. Hardware & testing methodology

Our experiments were conducted on a system equipped with dual socket Intel Xeon 2667v4 CPUs, a SATA SSD and sufficient DDR4 memory. All timings were performed using the functionality provided by the chrono header file in the C++ standard template library. Direct-IO was used to prevent the operating system from caching files in system memory.

The scenes used for testing (Figure 8) are the Austrian imperial crown PBRTv3 model by Martin Lubich, the PBRTv3 landscape scene by Laubwerk and the Moana Island Scene by Walt Disney Animation Studios. Because the crown has a low triangle count compared to the other two scenes it was subdivided 3 times to artificially raise its triangle count by a factor of 27 (Table 1). Although the Landscape scene can be rendered on a mid- or high-end graphics card, the Island scene proves a challenge for even high-end systems. Note that our renderer only supports triangles and thus other geometry types (such as curves) were removed from the Island scene. The crown, landscape and Island scenes were rendered at resolutions of 1000x1400, 1024x576 and 1024x429 respectively using 128 samples per pixel for all tests.

	Crown	Landscape	Island
Unique	95,585,778	25,947,395	134,552,386
Instanced	95,585,778	4,330,133,089	31,443,289,446

Table 1: Triangle counts (discarding other primitives) of the tested scenes. In the first row triangles that are instanced multiple times are only counted once; the second row counts each individual instance.

Selecting the appropriate batching point size (in terms of the number of primitives) is not trivial. In previous works this might not have had a big impact on performance. In our work however it impacts both the computational overhead and the effectiveness of occlusion culling. We have arbitrarily chosen batching point sizes of at least 1000000, 5000000 and 10000000 primitives for the crown, landscape and Island scenes, which results in 74, 653 and 2355 batching points respectively.

5.2. Results

The goal of occlusion culling is to provide an early-out opportunity for ray batching, resulting in less bandwidth usage. Culling might also reduce the computational cost of rendering since it prevents rays from traversing bottom-level BVHs for which they do not intersect any primitives.

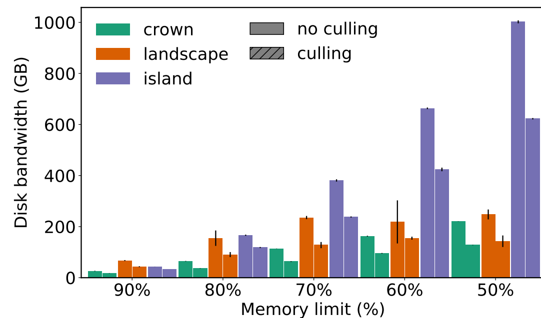


Figure 9: The effect of occlusion culling on the disk bandwidth used for rendering (including tail).

The resolution of the voxel grids from which the SVDAGs are generated impacts both the memory usage, computational overhead and effectiveness of the culling system. Figure 10a shows the total memory usage of the SVDAGs used for culling. As expected, memory usage of the SVDAGs scales cubically with the underlying voxel grid resolution. The compression ratio of the SVDAGs varies wildly for the landscape and Island scenes while for the crown scene the SVDAG compression seems to work better at higher resolutions. Note that the large discrepancies in SVDAG memory usage between the scenes can be attributed to the different number of SVDAGs per scene (which equals the number of batching points).

Increasing the resolution quickly has a diminishing effect on the number of rays that are culled

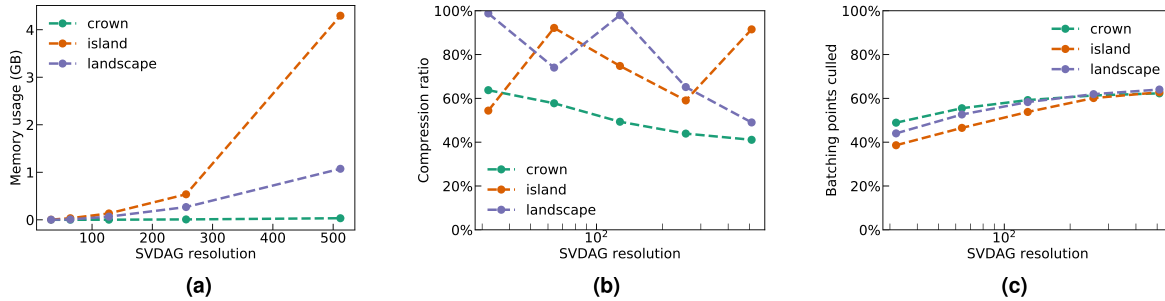


Figure 10: Memory usage (a), compression ratio vs SVO (b) and culling effectiveness (c) of the SVDAGs.

(Figure 10c). As expected, scenes with many primitives per batching points (like the Island scene) require a higher resolution for optimal culling effectiveness.

To test the effect of our occlusion culling system on out-of-core rendering, the scenes were rendered with different geometry memory limits with both occlusion culling enabled and disabled. In all tests a SVDAG resolution of 128^3 was used since it strikes a good balance between culling effectiveness and memory overhead. The reference memory limits were found by rendering the scenes without constraints and measuring the total amount of geometry data that was loaded. In the tests with culling enabled the memory used by the SVDAGs was subtracted from the geometry memory budget.

The results are shown in Figure 9 and 12a. The batched traversal scheme is effective at hiding disk latency when bandwidth is not a bottleneck, as is evident from the Island scene. Occlusion culling is able to reduce disk bandwidth by up to 45% and render time by up to 55%. Interestingly, occlusion culling improved performance even in situations where disk bandwidth was not the limiting factor. Even without a memory limit (just lazy loading), occlusion culling is able to improve performance by preventing unnecessary traversal of the large bottom-level BVHs.

5.3. Handling the tail

A potential problem with batched ray traversal is handling the tail of the computation. The tail is when almost all rays have been processed and the system is left with only few rays that are scattered over the batching points; Batched ray traversal is then not able to properly hide latency because disk bandwidth heavily outweighs BVH traversal cost.

Despite our efforts to reduce the problem by tweaking the scheduler, a lot of time is spent on processing this "tail" (Figure 11). The problem is especially bad in the Island scene where over 80% of the time is spent processing the final 16 million rays (out of a total of 56 million paths).

The easiest solution to this problem is to simply stop rendering when the number of rays in the system starts to drop. In our case this makes the system non-deterministic because the order in which rays are processed is arbitrary. Depending on run-time variables a ray might get processed, forwarded (if it missed) and processed again in the same iteration. If the system would not allow the forwarded rays to be processed in the same iteration then determinism could be attained.

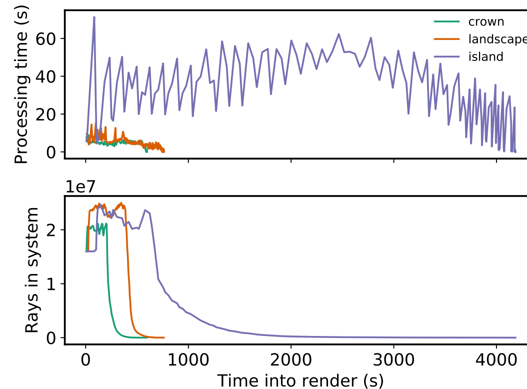


Figure 11: Most time is spent processing the final few rays when latency hiding is not possible.

To get an idea of what happens when we do not process the tail, we consider the render times as if the final 12 million rays were discarded (Figure 12b). Ignoring the tail has a big impact on the overall performance of the renderer. When there are enough rays to process the system is successful at hiding disk latency. Even when not processing the tail, occlusion culling is still able to improve performance in all scenarios. And when disk bandwidth does become a bottleneck, occlusion culling more than doubles performance.

5.4. Discussion

Our testing confirms that batched ray traversal can be an effective technique to hide disk latency in out-of-core rendering. Occlusion culling was able to improve performance of batched ray traversal in all tested scenarios. Our belief is that occlusion culling works so consistently well because it prevents expensive BVH traversal and helps rays progress faster. By allowing rays to skip batching

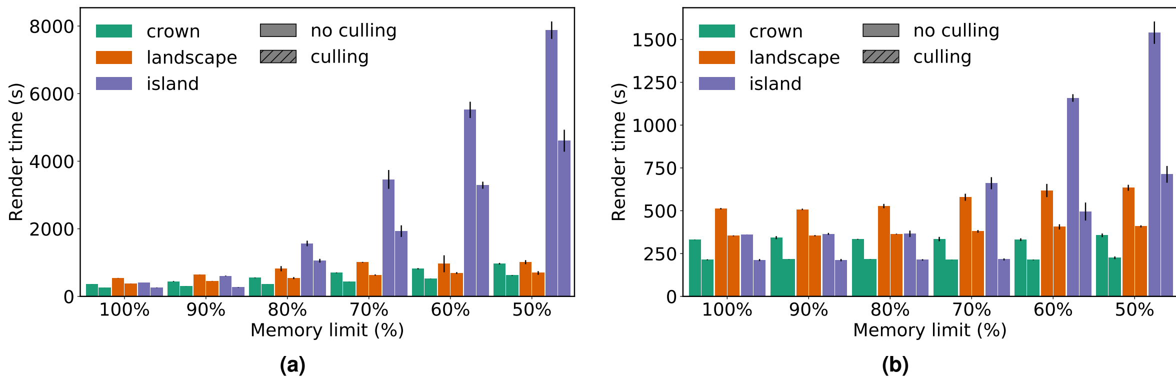


Figure 12: The impact of occlusion culling on the total render time at different memory limits. (a) shows the results when all rays are processed. (b) shows the results when the renderer would not process the tail (by discarding the last 12 million rays).

points less batching operations are required to find the closest intersections.

Instancing geometry is commonly used in the movie industry to such an extent that duplicating geometry is unfeasible. We have shown that it is possible to support instancing in out-of-core batched ray traversal and also that it performs well. Instancing does present a new challenge in picking the optimal batching points. We believe that this topic is something that could be researched.

Our implementation of batched ray traversal with occlusion culling has some shortcomings. Right now culling is only used to determine whether a ray potentially hits geometry. The distance (along the ray) to the closest hit voxel can be used to cull rays based on their search interval. With this information approximate hit points can be computed, which may be used to sort batched rays to improve coherence [MBK+10].

Our renderer also only supports very rudimentary shading. In production rendering, shading and out-of-core texture accesses are costly operations [ENSB13, LGXT17]. When shading and traversal run in parallel this may help the batched traversal scheme to hide disk latency with compute. However, if traversal has to share bandwidth with a texture cache then this might degrade traversal performance because a bandwidth bottleneck will be reached faster.

Finally, the scheduling algorithm that we used does not seem optimal. [PKGH97] for example relies on more detailed information like ray "weights" to select batching points to process. This however requires a lot of communication between worker threads, which is something that we tried to reduce.

We do believe however that letting the scheduler select multiple batching points at once and processing them in parallel is preferable over invoking the scheduler each time a thread runs out of work. Processing in iterations makes it easier to attain deterministic results, even when discarding rays to prevent a bottleneck at the tail. It also makes it easier to take snapshots of the current state, which could be used to make back-ups or to show the user an intermediate image.

6. Conclusion

We have discussed batched ray traversal as a means to improve memory coherency for ray traced global illumination. Our novel idea is to cull rays against conservative proxy geometry before batching them. To support our theory we have build an out-of-core renderer that implements batched ray traversal. Occlusion culling was added by voxelizing geometry and storing it in a Sparse Voxel Directed Acyclic Graph.

Our results confirm that batched ray traversal is effective at hiding disk latency and that render times do not change much as long as disk bandwidth is not a bottleneck. Enabling occlusion culling saw an improvement in render time across all tested scenarios. By reducing wasted work performance was improved even when no memory limit was set. Occlusion culling also reduces disk bandwidth which strongly improves render times in scenarios where the storage drive is a bottleneck.

From our testing we conclude that occlusion culling would have improved performance even if all geometry was loaded into memory ahead of time. Previous works have already shown that batched ray traversal can also be used to improve in-core traversal performance. Future work may investigate whether occlusion culling can also benefit in-core rendering. Although we tested culling in combination with batched ray traversal, the technique may also be applied to regular BVH traversal.

References

[Ail00] Timo Aila. Surrender umbra: A visibility determination framework for dynamic environments. *Master's thesis, Helsinki University of Technology*, 2000.

[BAM14] Rasmus Barringer and Tomas Akenine-Möller. Dy-

namic ray stream traversal. *ACM Trans. Graph.*, 33(4):151:1–151:9, July 2014.

[BBS+09] Brian Budge, Tony Bernardin, Jeff A Stuart, Shubhabrata Sengupta, Kenneth I Joy, and John D Owens. Out-of-core data management for path tracing on hybrid resources. In *Computer Graphics Forum*, volume 28, pages 385–396. Wiley Online Library, 2009.

[Bik12] Jacco Bikker. Improving data locality for efficient in-core path tracing. In *Computer Graphics Forum*, volume 31, pages 1936–1947. Wiley Online Library, 2012.

[CCC87] Robert L. Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. *SIGGRAPH Comput. Graph.*, 21(4):95–102, August 1987.

[CE97] Michael Cox and David Ellsworth. Application-controlled demand paging for out-of-core visualization. In *Visualization'97., Proceedings*, pages 235–244. IEEE, 1997.

[CLF+03] Per H. Christensen, David M. Laur, Julia Fong, Wayne L. Wooten, and Dana Batali. Ray Differentials and Multiresolution Geometry Caching for Distribution Ray Tracing in Complex Scenes. *Computer Graphics Forum*, 22(3):543–552, 2003.

[CNLE09] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 15–22, New York, NY, USA, 2009. ACM.

[DGP04] David E DeMarle, Christiaan P Gribble, and Steven G Parker. Memory-savvy distributed interactive ray tracing. In *Egpgv*, pages 93–100. Citeseer, 2004.

[ENSB13] Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. Sorted deferred shading for production path tracing. In *Computer Graphics Forum*, volume 32, pages 125–132. Wiley Online Library, 2013.

[FLP+17] Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, Bernd Hamann, and Achim Ebert. Accelerated single ray tracing for wide vector units. In *Proceedings of High Performance Graphics*, HPG '17, pages 6:1–6:9, New York, NY, USA, 2017. ACM.

[FLPE15] Valentin Fuetterling, Carsten Lojewski, Franz-Josef Pfreundt, and Achim Ebert. Efficient ray tracing kernels for modern cpu architectures. *Journal of Computer Graphics Techniques (JCGT)*, 4(4), 2015.

[Gas16] TG Gasparian. Fast divergent ray traversal by batching rays in a bvh. Master's thesis, Utrecht University, 2016.

[GKM93] Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238. ACM, 1993.

[GR08] Christiaan P Gribble and Karthik Ramani. Coherent ray tracing via stream filtering. In *Interactive Ray Tracing, 2008. RT 2008. IEEE Symposium on*, pages 59–66. IEEE, 2008.

[KSA13] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. High resolution sparse voxel dags. *ACM Trans. Graph.*, 32(4):101:1–101:13, July 2013.

[LGXT17] Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. Vectorized production path tracing. In *Proceedings of High Performance Graphics*, page 10. ACM, 2017.

[LK11] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2011.

[MBK+10] Bochang Moon, Yongyoung Byun, Tae-Joon Kim, Pio Claudio, Hye-Sun Kim, Yun-Ji Ban, Seung Woo Nam, and Sung-Eui Yoon. Cache-oblivious ray reordering. *ACM Trans. Graph.*, 29(3):28:1–28:10, July 2010.

[NFLM07] Paul Arthur Navratil, Donald S Fussell, Calvin Lin, and William R Mark. Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 95–104. IEEE, 2007.

[PFHA10] Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. Pantaray: Fast ray-traced occlusion caching of massive scenes. *ACM Trans. Graph.*, 29(4):37:1–37:10, July 2010.

[Pha18] Matt Pharr. Guest editor's introduction: Special issue on production rendering. *ACM Trans. Graph.*, 37(3):28:1–28:4, July 2018.

[PKGH97] Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 101–108, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[PU03] Eric Parker and Tushar Udeshi. Exploiting self-similarity in geometry for voxel based solid modeling. In *Proceedings of the Eighth ACM Symposium on Solid Modeling and Applications*, SM '03, pages 157–166, New York, NY, USA, 2003. ACM.

[RGD09] Karthik Ramani, Christiaan P Gribble, and Al Davis. Streamray: a stream filtering architecture for coherent ray tracing. In *ACM Sigplan Notices*, volume 44, pages 325–336. ACM, 2009.

[SS10] Michael Schwarz and Hans-Peter Seidel. Fast parallel surface and solid voxelization on gpus. *ACM Trans. Graph.*, 29(6):179:1–179:10, December 2010.

[SSLL14] Ari Silvennoinen, Hannu Saransaari, Samuli Laine, and Jaakko Lehtinen. Occluder simplification using planar sections. *Computer Graphics Forum*, 33(1):235–245, 2014.

[SY17] Myungbae Son and Sung-Eui Yoon. Timeline scheduling for out-of-core ray batching. In *Proceedings of High Performance Graphics*, HPG '17, pages 11:1–11:10, New York, NY, USA, 2017. ACM.

[Tsa09] John A. Tsakok. Faster incoherent rays: Multi-bvh ray stream tracing. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 151–158, New York, NY, USA, 2009. ACM.

[Val11] Will Vale. Practical occlusion culling on ps3. Game Developer Conference, 2011.

[WD89] Robert E Webber and Michael B Dillencourt. Compressing quadtrees via common subtree merging. *Pattern Recognition Letters*, 9(3):193 – 200, 1989.

[WDS05] Ingo Wald, Andreas Dietrich, and Philipp Slusallek. An interactive out-of-core rendering framework for visualizing massively complex models. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[WGBK07] Ingo Wald, Christiaan P Gribble, Solomon Boulos, and Andrew Kensler. Simd ray stream tracing-simd ray traversal with generalized ray packets and on-the-fly re-ordering. *Informe Técnico, SCI Institute*, 2007.

[Wih16] Graham Wihlidal. Optimizing the graphics pipeline with compute. Game Developer Conference, 2016.

[WSB01] Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive distributed ray tracing of highly complex models. In Steven J. Gortler and Karol Myszkowski, editors, *Rendering Techniques 2001*, pages 277–288, Vienna, 2001. Springer Vienna.

[YLM06] Sung-Eui Yoon, Christian Lauterbach, and Dinesh Manocha. R-lods: fast lod-based ray tracing of massive models. *The Visual Computer*, 22(9-11):772–784, 2006.

[ZMHHI97] Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E Hoff III. Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 77–88. ACM Press/Addison-Wesley Publishing Co., 1997.

7. Appendices

```cpp
int depth = intLog2(m_resolution);

struct NodeInfoN1 {
    uint32_t mortonCode; // Morton code (in level N-1).
    bool isLeaf;
    NodeOffset descriptorOffset;
};
std::vector<NodeInfoN1> previousLevelNodes;
std::vector<NodeInfoN1> currentLevelNodes;

// Creates and inserts leaf nodes.
uint32_t finalMortonCode = m_resolution * m_resolution * m_resolution;
for (uint32_t mortonCode = 0; mortonCode < finalMortonCode; mortonCode++) {
    if (grid.getMorton(mortonCode)) {
        currentLevelNodes.push_back({ mortonCode, true, 0 });
    }
}

auto createAndStoreDescriptor = [&](
        uint8_t validMask,
        uint8_t leafMask,
        const gsl::span<NodeOffset> childrenOffsets) {
    Descriptor d;
    d.validMask = validMask;
    d.leafMask = leafMask;

    NodeOffset descriptorOffset = static_cast<NodeOffset>(m_innerNodes.size());
    m_innerNodes.push_back(static_cast<NodeOffset>(d));

    // Store child offsets directly after the descriptor itself.
    m_innerNodes.insert(
        std::end(m_innerNodes),
        std::begin(childrenOffsets),
        std::end(childrenOffsets));

    return descriptorOffset;
};

// Work in a separate vector so m_innerNodes data doesnt change while
// inserting new descriptors.
NodeOffset rootNodeOffset = 0;
for (int N = 0; N < depth; N++) {
    std::swap(previousLevelNodes, currentLevelNodes);
    currentLevelNodes.clear();

    uint8_t validMask = 0x00;
    uint8_t leafMask = 0x00;
    eastl::fixed_vector<NodeOffset, 8> childrenOffsets;

    uint32_t prevMortonCode = previousLevelNodes[0].mortonCode >> 3;
    // Loop over all the cubes of the previous (more refined level)
    for (const auto& childNodeInfo : previousLevelNodes) {
        auto mortonCodeN1 = childNodeInfo.mortonCode;
        // Morton code of the node in the current level (stripping last 3 bits)
        auto mortonCodeN = mortonCodeN1 >> 3;
        if (prevMortonCode != mortonCodeN) {
            if ((validMask & leafMask) == 0xFF) {
                // Special case: all children are completely filled (1's):
                //  propagate this up the tree
                currentLevelNodes.push_back({ prevMortonCode, true, 0 });
            } else {
                // Different morton code (at the current level): we are finished
                // with the previous node => store it.
                auto descriptorOffset = createAndStoreDescriptor(
                    validMask, leafMask, childrenOffsets);
                currentLevelNodes.push_back(
                    { prevMortonCode, false, descriptorOffset });
            }
            validMask = 0;
            leafMask = 0;
            childrenOffsets.clear();
            prevMortonCode = mortonCodeN;
        }

        auto idx = mortonCodeN1 & ((1 << 3) - 1); // Right most 3 bits.
        validMask |= 1 << idx;
        if (childNodeInfo.isLeaf) {
            leafMask |= 1 << idx;
        } else {
            childrenOffsets.push_back(childNodeInfo.descriptorOffset);
        }
    }

    // Store final descriptor
    if ((validMask & leafMask) == 0xFF && N != depth - 1) {
        // Special case: all children are completely filled (1's): propagate
        // this up the tree (except if this node is the root node).
        currentLevelNodes.push_back({ prevMortonCode, true, 0 });
    } else {
        auto descriptorOffset = createAndStoreDescriptor(
            validMask, leafMask, childrenOffsets);
        auto lastNodeMortonCode = (previousLevelNodes.back().mortonCode >> 3);
        currentLevelNodes.push_back(
            { prevMortonCode, false, descriptorOffset });
        // Keep track of the offset to the root node.
        rootNodeOffset = descriptorOffset;
    }
}

// --> return rootNodeOffset;
```

Listing 1: C++ code to generate a SVO from a voxel grid

```cpp
int depth = intLog2(m_resolution);

struct NodeInfoN1 {
    uint32_t mortonCode; // Morton code (in level N-1).
    bool isLeaf;
    NodeOffset descriptorOffset;
};
std::vector<NodeInfoN1> previousLevelNodes;
std::vector<NodeInfoN1> currentLevelNodes;

// Creates and inserts leaf nodes.
uint32_t finalMortonCode = m_resolution * m_resolution * m_resolution;
for (uint32_t mortonCode = 0; mortonCode < finalMortonCode; mortonCode++) {
    if (grid.getMorton(mortonCode)) {
        currentLevelNodes.push_back({ mortonCode, true, 0 });
    }
}

auto createAndStoreDescriptor = [&](
        uint8_t validMask,
        uint8_t leafMask,
        const gsl::span<NodeOffset> childrenOffsets) {
    Descriptor d;
    d.validMask = validMask;
    d.leafMask = leafMask;

    NodeOffset descriptorOffset = static_cast<NodeOffset>(m_innerNodes.size());
    m_innerNodes.push_back(static_cast<NodeOffset>(d));

    // Store child offsets directly after the descriptor itself.
    m_innerNodes.insert(
        std::end(m_innerNodes),
        std::begin(childrenOffsets),
        std::end(childrenOffsets));

    return descriptorOffset;
};

// Work in a separate vector so m_innerNodes data doesnt change while
// inserting new descriptors.
NodeOffset rootNodeOffset = 0;
for (int N = 0; N < depth; N++) {
    std::swap(previousLevelNodes, currentLevelNodes);
    currentLevelNodes.clear();

    uint8_t validMask = 0x00;
    uint8_t leafMask = 0x00;
    eastl::fixed_vector<NodeOffset, 8> childrenOffsets;

    uint32_t prevMortonCode = previousLevelNodes[0].mortonCode >> 3;
    // Loop over all the cubes of the previous (more refined level)
    for (const auto& childNodeInfo : previousLevelNodes) {
        auto mortonCodeN1 = childNodeInfo.mortonCode;
        // Morton code of the node in the current level (stripping last 3 bits)
        auto mortonCodeN = mortonCodeN1 >> 3;
        if (prevMortonCode != mortonCodeN) {
            if ((validMask & leafMask) == 0xFF) {
                // Special case: all children are completely filled (1's):
                //  propagate this up the tree
                currentLevelNodes.push_back({ prevMortonCode, true, 0 });
            } else {
                // Different morton code (at the current level): we are finished
                // with the previous node => store it.
                auto descriptorOffset = createAndStoreDescriptor(
                    validMask, leafMask, childrenOffsets);
                currentLevelNodes.push_back(
                    { prevMortonCode, false, descriptorOffset });
            }
            validMask = 0;
            leafMask = 0;
            childrenOffsets.clear();
            prevMortonCode = mortonCodeN;
        }

        auto idx = mortonCodeN1 & ((1 << 3) - 1); // Right most 3 bits.
        validMask |= 1 << idx;
        if (childNodeInfo.isLeaf) {
            leafMask |= 1 << idx;
        } else {
            childrenOffsets.push_back(childNodeInfo.descriptorOffset);
        }
    }

    // Store final descriptor
    if ((validMask & leafMask) == 0xFF && N != depth - 1) {
        // Special case: all children are completely filled (1's): propagate
        // this up the tree (except if this node is the root node).
        currentLevelNodes.push_back({ prevMortonCode, true, 0 });
    } else {
        auto descriptorOffset = createAndStoreDescriptor(
            validMask, leafMask, childrenOffsets);
        auto lastNodeMortonCode = (previousLevelNodes.back().mortonCode >> 3);
        currentLevelNodes.push_back(
            { prevMortonCode, false, descriptorOffset });
        // Keep track of the offset to the root node.
        rootNodeOffset = descriptorOffset;
    }
}

// --> return rootNodeOffset;
```

Listing 2: SVDAG compression code (C++)

```cpp
constexpr int CAST_STACK_DEPTH = 23;// First bit of the exponent.

// Get rid of small ray direction components to avoid division by zero.
constexpr float epsilon = 1.1920928955078125e-07f;
if (abs(ray.direction.x) < epsilon)
    ray.direction.x = copysign(epsilon, ray.direction.x);
if (abs(ray.direction.y) < epsilon)
    ray.direction.y = copysign(epsilon, ray.direction.y);
```

```
    if (abs(ray.direction.z) < epsilon)
        ray.direction.z = copysign(epsilon, ray.direction.z);

    // Precompute the coefficients of tx(x), ty(y) and tz(z).
    // The octree is assumed to reside at coordinates [1, 2].
    vec3_f32 tCoef = vec3_f32(1.0f) / -abs(ray.direction);
    vec3_f32 tBias = tCoef * ray.origin;

    // Select octant mask to mirror the coordinate system so that ray direction is
    // negative along each axis.
    mask3 octantMask = ray.direction > 0.0f;
    int octantMaskBits = 7 ^ octantMask.bits(); // Mask out channel 4
    tBias = select(tBias, vec3_f32(3.0f) * tCoef - tBias, octantMask);

    // Initialize the current voxel to the first child of the root.
    const Descriptor* parent = rootNode;
    vec3_f32 pos = vec3_f32(1.0f);
    int scale = CAST_STACK_DEPTH - 1;
    float scaleExp2 = 0.5f;// exp2f(scale - CAST_STACK_DEPTH)

    // Initialize the active span of t-values
    const float tMin = max(0.0f, horizontalMax(vec3_f32(2.0f) * tCoef - tBias));
    const float tMax = horizontalMin(tCoef - tBias);

    if (tMin >= tMax)
        return {};

    // Store as vector to reduce scalar/vector conversions.
    vec3_f32 tMinVec(tMin);

    // Intersection of ray with the center planes of the root node.
    mask3 idxMask = (vec3_f32(1.5f) * tCoef - tBias) > tMin;
    int idx = idxMask.bitMask();
    pos = select(pos, vec3_f32(1.5f), idxMask);

    // Traverse voxels along the ray until we exit the octree.
    array<const Descriptor*, CAST_STACK_DEPTH + 1> stack;
    while (scale < CAST_STACK_DEPTH) {
        // === INTERSECT ===
        // Determine the maximum t-value of the cube by evaluating tx(), ty() and
        // tz() at its corner.
        vec3_f32 tCorner = pos * tCoef - tBias;

        // Process voxel if the corresponding bit in the parents valid mask is set.
        int childIndex = 7 - ((idx & 0b111) ^ octantMaskBits);
        if (parent->isValid(childIndex)) {
            float half = scaleExp2 * 0.5f;
            vec3_f32 tCenter = half * tCoef + tCorner;

            // === PUSH ===
            stack[scale] = parent;

            if (parent->isLeaf(childIndex)) {
                break;
            }

            // Find child descriptor corresponding to the current voxel.
            parent = getChild(parent, childIndex);

            // Select the child voxel that the ray enters first.
            scale--;
            scaleExp2 = half;

            idx = (tCenter > tMinVec).bitMask();
            pos = select(pos, pos + half, idxMask);
        } else {
            // === ADVANCE ===
            const float scaleExp2 = scaleExp2LUT[scale];
            vec3_f32 tcMax = horizontalMin(tCorner); // Slightly faster

            // Step along the ray.
            mask3 stepMask = tCorner <= tcMax;
            int stepMaskBits = stepMask.bitMask();
            pos = select(pos, pos - scaleExp2, stepMask);

            // Update active t-span and flip bits of the child slot index.
            tMinVec = tcMax;
            idx = stepMaskBits;

            // Proceed with pop if the bit flip disagree with the ray direction.
            if ((idx & stepMaskBits) != 0) {
                // === POP ===
                // Find the highest differing bit between the two positions.
                vec3_u32 differingBitsVec = select(
                    0u,
                    floatBitsToInt(pos) ^ floatBitsToInt(pos + scaleExp2),
                    stepMask);
                unsigned differingBits =
                    differingBitsVec[0] |
                    differingBitsVec[1] |
                    differingBitsVec[2];

                // When the ray exists the octree, at least of one the components
                // of pos will lie between 0.5 and 1.0. In the floating point
                // representation this means that the first bit of the exponent
                // changes. This results in the scale being set to 23, breaking the
                // traversal loop.
                scale = reverseBitScan(differingBitsVec); // Left-most set bit
                scaleExp2 = scaleExp2LUT[scale]; // exp2f(scale - s_max)

                // Restore parent voxel from the stack.
                parent = stack[scale];
            }

            // Round cube position and extract child slot index.
            vec3_u32 sh = floatBitsToInt(pos) >> scale;
            pos = intBitsToFloat(sh << scale);
            const vec3_u32 shMask1 = sh & 0x1;
            const vec3_u32 shMask2 = sh & 0x2;
            const vec3_u32 shMask1Shifted = shMask1 << vec3_u32(0, 1, 2);
            const vec3_u32 shMask2Shifted = shMask2 << vec3_u32(2, 3, 4);
            const vec3_u32 partialIdx = shMask1Shifted | shMask2Shifted;
            idx = partialIdx[0] | partialIdx[1] | partialIdx[2];
        } // Push / pop
    } // While

    // Indicate miss if we are outside the octree.
    if (scale >= CAST_STACK_DEPTH) {
        return {};
    } else {
        // Output result.
        return tMinVec[0];
    }
```

Listing 3: SVO / SVDAG traversal code (C++)

```
void intersect(const Ray& ray)
{
    SIMDRay simdRay;
    simdRay.originX = vec8_f32(ray.origin.x);
    simdRay.originY = vec8_f32(ray.origin.y);
    simdRay.originZ = vec8_f32(ray.origin.z);
    simdRay.invDirectionX = vec8_f32(1.0f / ray.direction.x);
    simdRay.invDirectionY = vec8_f32(1.0f / ray.direction.y);
    simdRay.invDirectionZ = vec8_f32(1.0f / ray.direction.z);
    simdRay.tnear = vec8_f32(ray.tnear);
    simdRay.tfar = vec8_f32(ray.tfar);
    // Store the offset into the child order LUT so we only have to compute it
    // once.
    auto shiftAmount = signShiftAmount(
        ray.direction.x > 0,
        ray.direction.y > 0,
        ray.direction.z > 0);
    simdRay.raySignShiftAmount = vec8_u32(shiftAmount);

    // Initialize the stack.
    alignas(32) std::array<uint32_t, 48> stackCompressedNodeHandles;
    alignas(32) std::array<float, 48> stackDistances;
    size_t stackPtr = 0;

    // Push root node onto the stack
    stackCompressedNodeHandles[stackPtr] = m_compressedRootHandle;
    stackDistances[stackPtr] = 0.0f;
    stackPtr++;

    // While the stack is not empty
    while (stackPtr > 0) {
        // Pop item from the stack
        stackPtr--;
        uint32_t compressedNodeHandle = stackCompressedNodeHandles[stackPtr];
        float distance = stackDistances[stackPtr];

        // The handle uses 1 bit to encode whether child is a leaf or an inner
        // node.
        uint32_t handle = decompressNodeHandle(compressedNodeHandle);
        if (isInnerNode(compressedNodeHandle)) {
            const auto* node = m_innerNodes.get(handle);

            // Intersect ray with the 8 children, return compressed result
            vec8_u32 childrenSIMD;
            vec8_f32 distancesSIMD;
            uint32_t numChildren = intersectInnerNode(
                node,
                simdRay,
                childrenSIMD,
                distancesSIMD);

            // Push children that intersect onto the stack
            if (numChildren > 0) {
                childrenSIMD.store(
                    stackCompressedNodeHandles.data() + stackPtr);
                distancesSIMD.store(stackDistances.data() + stackPtr);

                stackPtr += numChildren;
            }
        } else {
            const auto* leaf = m_leafs.get(handle);
            // Encode primitive count with 2 bits (leafs may store 1 to 5
            // primitives).
            uint32_t primitiveCount = leafNodePrimitiveCount(
                compressedNodeHandle);
            if (intersectLeaf(leaf, primitiveCount, ray)) {
                // Ray intersects at least one primitive
                simdRay.tfar.broadcast(ray.tfar);

                // Compress stack by removing items whose closest distance is
                // further than the new found distance to the closest primitive.
                size_t outStackPtr = 0;
                for (size_t i = 0; i < stackPtr; i += 8) {
                    vec8_u32 nodesSIMD;
                    vec8_f32 distancesSIMD;
                    distancesSIMD.loadAligned(stackDistances.data() + i);
                    nodesSIMD.loadAligned(
                        stackCompressedNodeHandles.data() + i);

                    // AVX2 compression using a look-up table to compute the
                    // permutation indices.
                    mask8 distMask = distancesSIMD < simdRay.tfar;
                    vec8_u32 compressPermuteIndices =
                        distMask.computeCompressPermutation();
                    distancesSIMD =
                        distancesSIMD.permute(compressPermuteIndices);
                    nodesSIMD = nodesSIMD.permute(compressPermuteIndices);

                    distancesSIMD.store(stackDistances.data() + outStackPtr);
                    nodesSIMD.store(
                        stackCompressedNodeHandles.data() + outStackPtr);

                    size_t numItems = std::min((size_t)8, stackPtr - i);
                    unsigned validMask = (1 << numItems) - 1;
                    outStackPtr += popCount(distMask.moveMask() & validMask);
                }
                stackPtr = outStackPtr;
            }
        }
```

```cpp
        }
    }



    uint32_t intersectInnerNode(
        const BVHNode* n,
        const SIMDRay& ray,
        vec8_u32& outChildren,
        vec8_f32& outDistances)
    {
        // Find the entry / exit distances of each child
        vec8_f32 tx1 = (n->minX - ray.originX) * ray.invDirectionX;
        vec8_f32 tx2 = (n->maxX - ray.originX) * ray.invDirectionX;
        vec8_f32 ty1 = (n->minY - ray.originY) * ray.invDirectionY;
        vec8_f32 ty2 = (n->maxY - ray.originY) * ray.invDirectionY;
        vec8_f32 tz1 = (n->minZ - ray.originZ) * ray.invDirectionZ;
        vec8_f32 tz2 = (n->maxZ - ray.originZ) * ray.invDirectionZ;
        vec8_f32 txMin = simd::min(tx1, tx2);
        vec8_f32 tyMin = simd::min(ty1, ty2);
        vec8_f32 tzMin = simd::min(tz1, tz2);
        vec8_f32 txMax = simd::max(tx1, tx2);
        vec8_f32 tyMax = simd::max(ty1, ty2);
        vec8_f32 tzMax = simd::max(tz1, tz2);
        vec8_f32 tmin = simd::max(ray.tnear,
            simd::max(txMin, simd::max(tyMin, tzMin)));
        vec8_f32 tmax = simd::min(ray.tfar,
            simd::min(txMax, simd::min(tyMax, tzMax)));

        // Get the (approximate) front-to-back ordering from the look-up table
        // stored inside the BVH nodes using the ray signs as indices. Note that
        // permutationOffsets is of type vec8_u32.
        const vec8_u32 indexMask = 0b111;
        const vec8_u32 simd24 = 24;
        vec8_u32 index =
            (n->permutationOffsets >> ray.raySignShiftAmount) & indexMask;

        // Permute tmin / tmax such that the children are ordered front-to-back.
        tmin = tmin.permute(index);
        tmax = tmax.permute(index);

        // Sort and compress the hit children and their entry distances.
        mask8 mask = tmin <= tmax;
        vec8_u32 compressPermuteIndices = mask.computeCompressPermutation();
        outChildren = n->children.permute(index).permute(compressPermuteIndices);
        outDistances = tmin.permute(compressPermuteIndices);
        return mask.count();
    }

    constexpr std::array<uint64_t, 256> genCompressLUT8()
    {
        std::array<uint64_t, 256> result = {};
        for (uint64_t mask = 0; mask < 256; mask++) {
            uint64_t indices = 0; // Permutation indices
            uint64_t k = 0;
            for (uint64_t bit = 0; bit < 8; bit++) {
                if ((mask & (1ull << bit)) != 0) { // If bit is set
                    indices |= bit << k; // Add to permutation indices
                    k += 8; // One byte
                }
            }
            result[mask] = indices;
        }
        return result;
    }
    constexpr std::array<uint64_t, 256> s_avxIndicesLUT = genCompressLUT8();

    // Implementation using a large look-up table. Alternatively the look-up table
    // could also be stored using only 3 bits per lane (3*8 = 24 bits per item), but
    // this requires more computations to unpack.
    __m256i computeCompressPermutationLUT()
    {
        uint64_t wantedIndices = s_avxIndicesLUT[m_bitMask];

        __m128i byteVec = _mm_cvtsi64_si128(wantedIndices);
        return _mm256_cvtepu8_epi32(byteVec);
    }

    // Alternative implementation that does not rely on a look-up table.
    __m256i computeCopmressPermutationAlternative()
    {
        // https://stackoverflow.com/questions/36932240/avx2-what-is-the-most-
        //  efficient-way-to-pack-left-based-on-a-mask/36951611

            // Unpack each bit to a byte
            uint64_t expandedMask = _pdep_u64(mask.m_bitMask, 0x0101010101010101);
            expandedMask *= 0xFF; // mask |= mask<<1 | mask<<2 | ... | mask<<7;
            // ABC... -> AAAAAAAABBBBBBBB...: replicate each bit to fill its byte

        // The identity shuffle for vpermps, packed to one index per byte
            const uint64_t identityIndices = 0x0706050403020100;
            uint64_t wantedIndices = _pext_u64(identityIndices, expandedMask);

        __m128i byteVec = _mm_cvtsi64_si128(wantedIndices);
        return _mm256_cvtepu8_epi32(byteVec);
    }
```

Listing 4: C++ AVX2 implementation of [FLP+17] for traversal of the bottom-level BVHs.

```cpp
struct InsertInfo
{
    uint32_t nodeHandle;
    uint64_t stack;
};

// Results:
//  - empty: traversal was paused (and ray batched)
//  - true:  closest hit was found
//  - false: no hit was found
std::optional<bool> intersect(Ray& ray, const InsertInfo& insertInfo)
{
    SIMDRay simdRay;
    simdRay.originX = vec4_f32(ray.origin.x);
    simdRay.originY = vec4_f32(ray.origin.y);
    simdRay.originZ = vec4_f32(ray.origin.z);
    simdRay.invDirectionX = vec4_f32(1.0f / ray.direction.x);
    simdRay.invDirectionY = vec4_f32(1.0f / ray.direction.y);
    simdRay.invDirectionZ = vec4_f32(1.0f / ray.direction.z);
    simdRay.tnear = vec4_f32(ray.tnear);
    simdRay.tfar = vec4_f32(ray.tfar);

    bool hit = false;
    auto [nodeHandle, stack] = insertInfo;
    const BVHNode* node = m_innerNodes.get(nodeHandle);
    while (true) {
        // Get the bit mask indicating which children we have not traversed yet.
        int bitPos = 4 * node->depth;
        uint64_t interestBitMask = (stack >> bitPos) & 0b1111;
        if (interestBitMask != 0) {
            // Convert to SSE (integer) mask.
            mask4 interestMask(
                interestBitMask & 0x1,
                interestBitMask & 0x2,
                interestBitMask & 0x4,
                interestBitMask & 0x8);

            // Compute the entry / exit distances of each child.
            vec4_f32 tx1 = (node->minX - simdRay.originX) * simdRay.invDirectionX;
            vec4_f32 tx2 = (node->maxX - simdRay.originX) * simdRay.invDirectionX;
            vec4_f32 ty1 = (node->minY - simdRay.originY) * simdRay.invDirectionY;
            vec4_f32 ty2 = (node->maxY - simdRay.originY) * simdRay.invDirectionY;
            vec4_f32 tz1 = (node->minZ - simdRay.originZ) * simdRay.invDirectionZ;
            vec4_f32 tz2 = (node->maxZ - simdRay.originZ) * simdRay.invDirectionZ;
            vec4_f32 txMin = min(tx1, tx2);
            vec4_f32 tyMin = min(ty1, ty2);
            vec4_f32 tzMin = min(tz1, tz2);
            vec4_f32 txMax = max(tx1, tx2);
            vec4_f32 tyMax = max(ty1, ty2);
            vec4_f32 tzMax = max(tz1, tz2);
            vec4_f32 tmin = max(simdRay.tnear, max(txMin, max(tyMin, tzMin)));
            vec4_f32 tmax = min(simdRay.tfar, min(txMax, min(tyMax, tzMax)));
            mask4 hitMask = tmin <= tmax;

            mask4 toVisitMask = hitMask && interestMask;
            if (toVisitMask.any()) {
                // Find closest unvisited child.
                vec4_f32 inf4(std::numeric_limits<float>::max());
                vec4_f32 maskedDistances = blend(inf4, tmin, toVisitMask);
                unsigned childIndex = horizontalMinIndex(maskedDistances);

                uint64_t toVisitBitMask = toVisitMask.bitMask();
                // Set the bit of the child we are visiting to 0.
                toVisitBitMask ^= (1llu << childIndex);
                // Set the bits in the stack corresponding to the current node to 0.
                stack = stack ^ (interestBitMask << bitPos);
                // And replace them by the new mask.
                stack = stack | (toVisitBitMask << bitPos);

                if (node->isInnerNode(childIndex)) {
                    nodeHandle = node->getInnerChildHandle(childIndex);
                    node = m_innerNodes.get(nodeHandle);
                } else {
                    auto handle = node->getLeafChildHandle(childIndex);
                    const auto& leaf = leafs.get(handle);
                    auto optResult = leaf.intersect(ray, hitInfo, userState, { nodeHandle, stack });
                    if (!optResult)
                        return {}; // Ray was paused.

                    if (*optResult) {
                        hit = true;
                        simdRay.tfar = vec4_f32(ray.tfar);
                    }
                }

                continue;
            }
        }

        // No children left to visit; find the first ancestor that has work left.

        // Set all bits after bitPos to 1.
        uint64_t oldStack = stack;
        stack = stack | (0xFFFFFFFFFFFFFFFF << bitPos);
        if (stack == 0xFFFFFFFFFFFFFFFF)
            break; // Stop traversal if stack is empty.

        int prevDepth = node->depth;
        nodeHandle = node->parentHandle;
        node = m_innerNodes.get(nodeHandle);
    }
}

unsigned horizontalMinIndex(__m128 vec)
{
    // min2: channels [0,1] = min(0,1), channels [2,3] = min(2,3)
    __m128 min1 = _mm_shuffle_ps(vec, vec, _MM_SHUFFLE(1, 0, 3, 2));
    __m128 min2 = _mm_min_ps(vec, min1);

    // min3: channels [0,1] = min(2,3), channels[2,3] = min(0,1)
    // min4: channels [0-3] = min(min(0,1), min(2,3))
    __m128 min3 = _mm_shuffle_ps(min2, min2, _MM_SHUFFLE(2, 3, 0, 1));
    __m128 min4 = _mm_min_ps(min2, min3);

    __m128 mask = _mm_cmpeq_ps(vec, min4);

    int bitMask = _mm_movemask_ps(mask);
    return bitScan32(static_cast<uint32_t>(bitMask));
}
```

Listing 5: C++ SSE implementation of [Gas16] for traversal of the top-level BVH

LITERATURE STUDY

OCCLUSION CULLING IN MEMORY-COHERENT RAY TRACING

December 20, 2018

Student: Mathijs Molenaar

Under the supervision of:

Jacco Bikker & Elmar Eisenmann

Contents

1 Introduction **3**

2 Motivation **4**

3 Previous Work **5**

3.1 Out-Of-Core Ray Tracing 5
3.2 Memory-Coherent Ray Tracing 7
3.3 Occlusion culling 10
3.3.1 Offline 11
3.3.2 Online 11
3.3.3 Occluder simplification 12
3.4 Sparse Voxel Directed Acyclic Graphs 14
3.5 Depth Map for Ray Tracing 16
3.5.1 Shadow map compression 16
3.5.2 Ray tracing using depth maps 18

1 INTRODUCTION

In 1987, still in the early days of computer graphics, Cook, Carpenter and Catmull presented the "Render Everything You Ever Saw" Reyes Image Rendering Architecture (Cook et al. [1987]). At that time it had already been used in production at Lucas Films and it would later also be used to render the first computer animated feature film "Toy Story". The Pixar Renderman renderer, which implements Reyes and was co-developed by the same authors, has become a staple in the movie industry for generating images of both live action- and computer animated movies.

In the years after the introduction of Reyes, many additions were made to increase the visual fidelity of Renderman. Most notably, rendering methods were developed that incorporated ray tracing into the Reyes architecture. In 2006 for example, ray tracing was added to provide high quality reflections for the movie “Cars“ (Christensen et al. [2006]).

In 1986, during the development of Reyes/Renderman, Kajiya presented the rendering equation Kajiya [1986]; which is an integral equation describing the amount of light emitted from a point on a surface in a give direction. The paper also describes a Monte Carlo solution to the equation which requires finding intersections between rays and geometry. This method of solving the rendering equation is referred to as path tracing. Although generating physically accurate images, path tracing was too computationally expensive for computers at the time.

It took over 10 years of computer chip development before the first short film rendering using path tracing, “Bunny“ from Blue Sky Studios, was released. This award winning short proved that path tracing was capable of rendering physically based effects such as global illumination and depth of field at a production quality level (Christensen et al. [2016]). “Bunny“ also inspired the development of the Arnold renderer at Solid Angle Angle. Arnold was used for a couple of shorts; before being used to render the first feature length path traced movie: “Monster House“.

Recently path-tracing has gained a lot of traction in the (animated) movie industry. Examples of this are Disney Animation developing Hyperion (Eisenacher et al. [2013]), DreamWorks Animation developing MoonRay (Lee et al. [2017]) and Pixar turning Renderman into a path tracer (Fascione et al. [2017]). This industry movement has been caused by the need for simpler control and has been fueled by the increased (parallel) processing power that has become available with the continuous improvements of computer hardware.

By design, path tracing is capable of supporting practically unlimited parallelism. However, like in Reyes, texture accesses from disk can become a bottleneck during shading. Both Hyperion and MoonRay sort shading request to improve the coherence of texture accesses

resulting in less time being spend on disk I/O. Hyperion creates very large batches of work and groups nearby hit points, thereby improving coherence. In MoonRay ray traversal and shading are performed in parallel by utilizing work queues. The work inside the queues is sorted in a similar manner to Hyperion.

Sorted deferred shading improves coherence, allowing for out-of-core texture accesses with minimal performance loss. Most production renderers (such as Hyperion, MoonRay and Renderman) do not support out-of-core geometry however. This means that scenes must fully fit in the computers memory (while leaving enough space for other parts the renderer). Industry specialists have indicated that these memory limitations are already a problem in production (Fascione et al. [2017]).

2 MOTIVATION

During my internship at Walt Disney Animation Studios I implemented out-of-core ray traversal, loosely based on the work of Pharr et al. [1997], in the Hyperion renderer. The implementation used Intel's Embree library since evaluating Embree was also part of my project. Because of limitations in Hyperion and Embree, and because of a lack of time; the resulting implementation did perform as desired.

With this master thesis project I hope to redeem this failed experiment by extending the algorithm and creating a more efficient implementation. Conceptually, this project will combine occlusion culling with memory-coherent ray tracing (Pharr et al. [1997]). Compression will ensure that the occlusion culling will not require an abundant amount of memory.

3 Previous Work

In this section I will go over some of the previous- and related work with respect to the topics covered in this project. First, related work on out-of-core ray tracing is discussed, followed by the research on memory-coherent ray tracing. Finally, I will give an overview of occlusion culling and the compression techniques that will be used in this project.

3.1 Out-Of-Core Ray Tracing

In 2001 Ingo Wald presented his work on a distributed ray tracer (**Wald et al. [2001a]**) that could render complex scenes at interactive frame rates. This paper was a continuation of his earlier work on packet traversal (Wald et al. [2001b]). By allowing packet traversal to run on a cluster of workstations he showed that it was possible to achieve performance that, at the time, was only possible on shared-memory supercomputers.

The paper's approach to distributed rendering is to treat each rendering node as a separate instance performing out-of-core computations. The display node acts as the master node: distributing rays over the available render nodes.

The scene is divided into voxels by building a coarse (with many primitives in the leaf nodes) BSP-tree (Binary Space Partitioning) over all geometry in the scene. For each voxel (leaf node) an additional BSP-tree is build, which is stored along with the geometry and shading data in a single file. These files are stored on a central server since duplicating the scene on each render node was deemed too expensive. When the traversal kernel encounters a missing voxel, the ray is suspended and an asynchronous loading thread is notified to load that voxel. A LRU (Least Recently Used) cache is used to determine which voxels should be evicted when memory runs low. During the asynchronous loading ray tracing continues with non-suspended rays.

DeMarle et al. [2004] takes a significantly different approach to distributed rendering. Load balancing is achieved using a work stealing scheme that removes the need for a central server. The scene is distributed over the system memory of all participating render nodes instead of storing it on a central server.

A page-based distributed shared memory layer makes memory distribution opaque to the rest of the system. Nodes are assigned ownership of different stripes of the shared memory space. Each node also has a cache which stores the most recently accessed pages of which they are not the owner. This shared memory layer removes the need for disk I/O; however, it limits the total scenes size to the sum of the available system memory on all the render

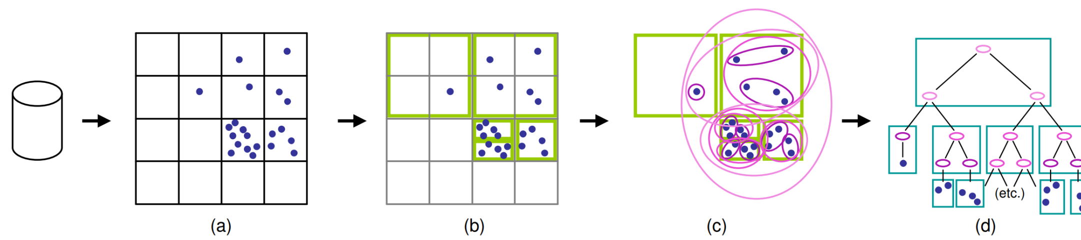


Figure 1: Pantaray's acceleration structure; (a). Buckets are coalesced and split into chunks (b) of up to 64KB. A BVH inside and among chunks (c) is broken into bricks (d) each brick is contiguous on disk. Source: Pantaleoni et al. [2010]

nodes.

Traversal of the acceleration structure is similar to that of Wald et al. [2001a]; When a page fault is triggered, the traversal thread is paused and the communication thread is requested to retrieve the page from the node that owns it.

Pantaleoni et al. [2010] describes the work on the PantaRay system that was used for rendering the movie Avatar. Unlike the other works mentioned here, PantaRay was only used for precomputing sparse directional occlusion caches; tessellation and final frame rendering were performed by Pixar Renderman with Reyes (Cook et al. [1987]).

Construction of the acceleration structure in PantaRay complex. The authors found that out-of-core construction of a BVH or k-d tree would easily be bottlenecked by I/O speed as it requires objects to be accessed multiple times in random order; so they introduced a new way of constructing an out-of-core acceleration structure.

In the first step of constructing the acceleration structure (figure 1), microgrids (small patches of geometry generated by Renderman) are divided over a regular 3D grid of buckets. A microgrid is assigned to all buckets in which it (spatially) resides. The resulting bucket-microgrid pairs are stored on disk. This bucketing pass creates manageable units of work that can fit into system memory.

The uniform grid is very coarse and often imbalanced. A k-d tree is build over all micro-grids (contained in the buckets) to create a more balanced structure. By coalescing nearby empty buckets as well as splitting large buckets this process creates roughly equally sized (64KB) leaf nodes called "chunks".

These chunks are used to build a two-level bounding volume hierarchy (BVH): a BVH over the microgrids within each chunk and a BVH over all chunks. Finally, the resulting BVH is split into "brick" treelets which are each stored contiguously on disk. For the purpose of level-of-detail, each brick stores the mean of the vertex data contained in the leaf nodes of

the tree below.

PantaRay is capable of leveraging the power of massively parallel GPU architectures and was designed with the help of Nvidia. In addition, it also contains a CPU back-end that was used for simple reflection occlusion and area light shaders only.

The CPU back-end loads bricks as they are needed. Recently accessed bricks are stored in a LRU cache. Similar to CPU caches, each thread has its own cache in conjunction to a shared cache between all the threads.

This caching system does not scale well to thousands of threads, as is required for an efficient GPU implementation. Instead, the authors solve the memory problem by means level-of-detail. Rays with similar origin positions (spherical sampling creates many rays with the same origin) are processed in batches. The system will only load as many bricks as GPU memory allows. Distant bricks that do not fit into GPU memory are represented as partially transparent bounding boxes (based on the vertex data propagated from the leaf nodes).

PantaRay proved to work well in production, although the level-of-detail solution may impact image quality. Furthermore, the LOD technique is not directly applicable to path tracing.

3.2 Memory-Coherent Ray Tracing

Pharr et al. [1997] presented a novel way of improving the performance of out-of-core ray tracing. Note that this paper was released a couple of years before any of the previously mentioned work. Hence, uniform grids were used as acceleration structure.

The first step in building the acceleration hierarchy is to divide the primitives in each mesh over a coarse uniform grid called the geometry grid. Voxels in the geometry grid may contain a couple thousand triangles which are stored contiguously on disk (with a cache to keep recently accessed voxels in system memory). The geometry grid thus helps group primitives in memory based on their spatial locality. For each non-empty voxel another uniform grid is constructed that acts as acceleration grid. Finally, another uniform grid is constructed over the whole scene, storing pointers to geometry grid voxels in the voxels. By pointing directly to geometry voxels this conceptually creates a two-level hierarchy.

Novel about this paper is how the top level grid is used as a scheduling grid. Each voxel of the scheduling grid has a queue of rays waiting to be intersected against the geometry inside that voxel. Newly spawned rays are added to the queue of the voxel in which their origin resides. A scheduler constantly selects a scheduling grid voxel, intersects all queued rays (using the acceleration grids), and performs any required shading calculations. Rays that

did not hit any geometry are pushed onto the queue of the next voxel along the ray. Newly spawned rays as a result of shading are re-added to the same voxels queue.

The system tries to minimize the number of times that a geometry voxel is loaded by guaranteeing that it will be used by many rays. To keep the memory usage down, ray queues may be stored on disk.

Scheduling of the voxels is based on a cost-benefit heuristic. The cost function estimates how expensive it will be to process the rays in a scheduling voxel (how much data should be loaded from disk). The benefit function estimates how much progress will be made towards the completion of the computation if a voxel were to be processed. In the paper, the benefit function is based on the amount of queued rays and the average weights of the rays (rays with high weights are more likely to spawn continuation rays).

In the paper the authors mention a failed experiment where they tried to cluster rays based on their origin. Rays in a cluster were sorted according to their direction, after which they were traced through the entire scene until an intersection was found. The authors dismiss the technique as it would be too reliable on rays having similar origins and because no coherence could be exploited for rays with significantly different origins passing through the same part of space. It is interesting that they mention this because it is very similar to the approach taken in Disney Animation's Hyperion renderer (Eisenacher et al. [2013]).

Navratil et al. [2007] takes inspiration from Pharr et al. [1997] and applies the queuing technique to an in-core ray tracer. The idea is to take the existing technique and apply it to a higher level of the memory hierarchy. The paper focuses on minimizing cache misses rather than disk accesses. The authors also present a scheduling scheme that bounds the maximum number of active rays at any time; which is something that Pharr et al. [1997] do not do.

A k-d tree is used as an acceleration structure instead of the uniform grid used in Pharr et al. [1997]. Compared to a grid, a k-d tree provides a better balance in the number of primitives per leaf. Pausing rays and resuming traversal in a k-d tree is slightly harder though. The paper does not mention how this is implemented; but there exist numerous stackless traversal algorithms that could be used.

Queue points in the tree are selected such that the sub-tree underneath fits inside the processors cache, while leaving some room for other data. The geometry and sub-trees are lazy-loaded into the cache during traversal; unlike previous work where they are loaded from disk ahead of time. This is possible because of the hardware differences between disk based storage and Random Access Memory (RAM). RAM memory can efficiently fetch memory pages in a random order while disk based storage would be bottlenecked by the movement of

the disk head.

Unlike Pharr et al. [1997], storing excess rays on disk is not an option (from a performance standpoint). Instead, the scheduling scheme is adjusted to minimize the number of active rays in the system at any time. Traversal now proceeds in batches of rays. First, all primary rays are inserted into the queue of the top level leaf node in which their origin resides. Then, the scheduler selects and processes queues by intersecting the rays (forwarding rays to the next node if they miss); this continues until all queues are empty. Shading is deferred until after traversal is finished. Newly spawned shadow- and continuation rays are traversed in separate batches to prevent exponential growth in the number of rays to be traversed.

The memory-coherent ray tracing architecture can also be applied to distributed (out-of-core) path tracing, as is shown by **Budge et al. [2009]**. The renderer proposed in the paper supports hybrid resources, meaning that it can run on a heterogeneous set of devices (e.g. both CPUs and GPUs).

The paper describes a hybrid data-management system which was designed as a layer of abstraction for generic distributed out-of-core applications. The abstraction layer consists of three key concepts: *kernels* that encapsulate the processing logic to complete a task, *static data* that provides access to constant application data and *transient data* that describes the actively manipulated workload. It is the abstraction layers responsibility to ensure that the static- and transient data are accessible to the kernel being executed. This means that it is responsible for copying data to device local memory, such as GPU memory .

Execution of the program is guided by a centralized work pool. Idle worker threads pop and execute the task from the pool with the highest priority. This priority is local to each worker thread and is based on the availability of the required data, processor preference, and the size of the workload.

The path tracer build on top of this system is similar to that of Navratil et al. [2007]; utilizing a two level k-d tree as acceleration structure. However like Pharr et al. [1997], it does not split traversal and shading into separate steps, allowing for unbounded grow in the number of rays. The paper does not mention this potential issue nor does it provide any data on the memory overhead of queuing rays.

Bikker [2012], like Navratil et al. [2007], uses the queuing traversal technique to improve the performance of in-core ray tracing by reducing cache misses. The difference is that an octree is used as the top level and a 4-wide BVH is used as the bottom level of the acceleration structure.

The octree supports pausing and resuming of ray traversal through a stackless traversal scheme utilizing neighbour links stored in the octree nodes (based on Havran et al. [1998]). The bottom level BVH is traversed using stream traversal Tsakok [2009] which utilizes the coherence of queued rays for extra performance. Like Pharr et al. [1997], new rays generated by shading are re-added to the traversal queues immediately.

Another novelty of this paper is how top level leaf nodes may have multiple ray queues associated with them. Ray queues have a maximum capacity; if a new ray would overflow the current ray queue then a new queue will be allocated. By bounding the size of the queues, run time memory allocations can be avoided and the number of partially filled queues being processed is reduced. By processing full ray queues first, partially filled queues have a chance of being filled by rays coming from neighboring octree leaf nodes. Processing full queues not only ensures that the main memory accesses can be amortized over many rays but it also improves the performance of the stream traversal.

Gasparian [2016] is a follow-up to the work of Bikker by one of his master students. In this work the top level octree is replaced by a 4-wide BVH. The implication of using a BVH as the top level acceleration structure is that implementing a stackless traversal scheme is much harder than with an octree or k-d tree.

The paper introduces a novel, ordered, stackless traversal scheme for 4-wide BVHs inspired by Áfra and Szirmay-Kalos [2014]. During traversal the ray stores a 64-bit bitstack containing, for all ancestors, the children that should be visited (4 bits per node). When a child is visited its corresponding bit is set to 0.

To resume traversal from a leaf node, the bitstack is used to find the lowest level ancestor with a child node that should be traversed. To efficiently move up the tree, each BVH node stores pointers to all of its ancestors. This makes moving up the tree a constant time operation at the cost of significantly increased memory requirements (in addition to the 8 bytes per ray for the bitstack). Traversal order is guaranteed by intersecting the ray with a nodes children (each time that node is visited) and selecting the closest child that needs to be traversed.

3.3 Occlusion culling

Occlusion culling is the act of determining which object are hidden behind other objects, and as such, do not have to be rendered. Traditionally, this has mostly been used in rasterized images. The reason for including it in this literature study is that is considered related work.

Occlusion culling techniques can be classified in to categories: offline and online.

3.3.1 Offline

Offline occlusion culling techniques compute the Potentially Visible Set (PVS) for regions of space (usually called cells) as a preprocess. In other words, these techniques try to find all objects that may be visible from any point inside a cell. This information is then used during run time to determine which objects might be visible (those in the PVS of the cell in which the camera resides). Usually this is combined with frustum culling to ensure that objects lying outside of the cameras field-of-view are not rendered.

There are a couple of downsides to this technique. First, storing the PVS of each cell may take an abundant amount of memory. This is especially true for open areas where a large part of the world may be visible. Second of all, computing the PVS of each cell in 3D (i.e. Coorg and Teller [1997]) is computationally expensive.

Therefore, a lot of research only considers occlusion culling of "2.5D" scenes, in which occlusion culling is performed in 2D by looking at the scene from the top down. Koltun et al. [2000] for example uses the top-down view of a scene to determine the PVS of a convex region of 2D space. It incrementally constructs a "virtual occluder" which represents the aggregate umbra of multiple occluders using a single line segment. The applicability of this type of algorithm is limited, as most scenes do not exhibit a "2.5D" structure.

3.3.2 Online

In contrast, online algorithms calculate the PVS during run time. Objects in the scene are stored in an acceleration structure such as a k-d tree or BVH. Potentially visible objects are selected by traversing the tree; child nodes are traversed if their bounding volume is determined to be potentially visible.

Determining the *potential* visibility of BVH nodes by culling them against the whole scene would be very inefficient. Instead, online algorithms usually select a subset of objects in the scene which are used as occluders. Occluder selection is heuristic guided and is usually based on factors such as: the view frustum, occluder size (as seen from the camera), visibility in last frame, and the occlusion factor in last frame.

Online testing of potential visibility is performed using fast image-space techniques. The hierarchical z-buffer (**Greene et al. [1993]**) was the first of such techniques and is still used in many modern games (Haar and Aaltonen [2015], Wihlidal [2016]). The hierarchical z-buffer is an image pyramid build over the original z-buffer where each sample contains the farthest z value of the corresponding four samples of the previous level.

Visibility of an axis-aligned bounding rectangle (representing an occludee) is tested recursively. If the z values of the covered samples are all larger than the maximum depth of the occludee then the occludee is considered visible; otherwise, we recursively continue to the next finer level of the hierarchy where the test is repeated.

Zhang et al. [1997] proposes to split occlusion testing from depth testing. A Hierarchical Occlusion Map (HOM) is build as an image pyramid, similar to the hierarchical z-buffer. But instead of storing depth, the hierarchy stores the occlusion factor. Pixels containing an occluder store a value of 1.0 and unoccluded pixels store 0.0.; occlusion values are combined using averaging.

Axis-aligned bounding rectangles are tested by selecting the level in the hierarchy where a sample and the bounding rectangle are roughly equally sized. If the overlapping samples are not completely opaque (value 1.0) then the recursion descends into the finer levels of the hierarchy. An object is *potentially* occluded if all overlapping pixels in the occlusion map are opaque (value 1.0). The algorithm also allows for culling of geometry that is hardly visible by slightly reducing this threshold.

Potentially occluded bounding rectangles are then tested against the depth estimation buffer, which is a low resolution z-buffer storing the *farthest* z value of the corresponding samples in the original z-buffer. A bounding rectangle is culled when its *nearest* z value is farther than the z values of all the samples it covers in the depth estimation buffer.

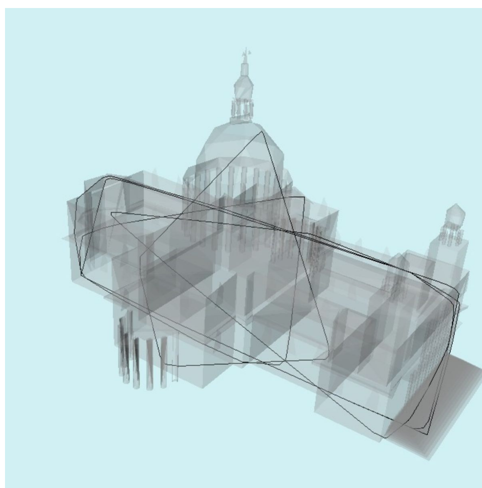
Aila [2000] introduces Incremental Occlusion Maps (IOM) which improves on Hierarchical Occlusion Maps (HOM). Unlike HOM, which utilizes graphics hardware; IOM implements a software rasterizer on the CPU. This allows IOM to only update the depth estimation buffer if the occluder changes the occlusion map, which saves a considerable amount of write operations. It also removes the need for copying the z-buffer to the CPU, which is a relatively slow operation. Another advantage of using a software rasterizer is that the pipeline can now provide valuable feedback with regards to the usefulness of occluders (which can guide occluder selection).

3.3.3 Occluder simplification

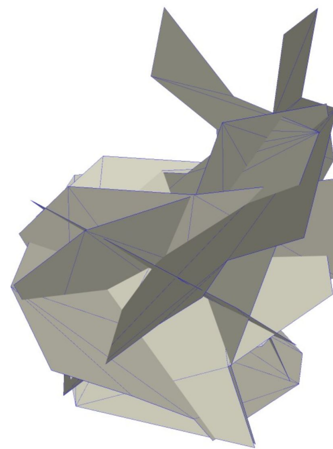
Occluder simplification, as to be used with any of the before mentioned algorithms, requires that the output mesh completely resides inside the original meshes volume. This is something that generic mesh simplification algorithms do not guarantee, so specialized simplification schemes have to be devised. Note that for ray tracing, this property should be reversed. The

original mesh must be contained inside the simplified version, such that any ray that hits the original mesh will also hit the simplified mesh. There is no research on this topic so we will only look at occluder generation for use in combination with the previously mentioned algorithms.

Law and Tan [1999] is one of the earliest works on occluder generation. It uses existing geometric simplification schemes to simplify the input mesh and then applies fixes to make the simplified mesh fit inside the original mesh. The edges of a simplified mesh are shifted towards the surface of the original mesh. This may result in invalid polygons as the connectivity might change, but this does not affect the visibility results.



(a) source: Brunet et al. [2001]



(b) source: Silvennoinen et al. [2014]

Figure 2: Examples of alternative ways of representing occluders. Left: hoops; Right: planar sections.

Germs and Jansen [2001] takes a 3D mesh of a building (without overhanging parts) and creates facades representing the sides of the building. Thus this work only considers a small set of “2.5D“ input meshes. A building is split at different z levels and for each level the top-down footprint is computed. These footprints are then simplified and combined back into a 3D mesh.

Cell based occlusion culling with convex occluders is simpler than with non-convex occluders. **Brunet et al. [2001]** points out that this requirement can be relaxed. Occluders do not need to be convex as long as their silhouette, as seen from any point in the view cell, is convex. Based on this knowledge the authors present a novel technique for representing occluders.

Occluders are represented by one or more non-planar, non-convex closed polylines called "hoops" (figure 2a). The algorithm computes hoops inside an occluder mesh such that their silhouette, as seen from the view cell, is convex. The algorithm starts by constructing a sparse voxel octree which stores the volume contained by the input mesh. It then builds the hoops by incrementally adding the corner vertices of the octree leaf nodes.

The idea of voxelizing a mesh and storing it in a sparse voxel octree for simplification is also used by **Darnell [2011]**. Oxel uses the sparse voxel octree to generate a 3D mesh consisting of a collection of axis aligned cubes.

The program first finds the voxel (inside of the mesh) that is furthest away from any external area. It then creates a box centered at that voxel and expands it until it reaches the outside of the mesh. This process repeated until enough volume is filled or the box count threshold is reached. The boxes are then filtered based on their contribution to the occlusion factor of the final occluder (as viewed from a discrete number of locations). Finally, intersecting boxes are merged into a single mesh to reduce the amount of geometry overdraw.

The last occluder simplification paper to be discussed is **Silvennoinen et al. [2014]**. This work generates multiple planar meshes, oriented in different directions, to represent an occluder (figure 2b).

To simplify the initial mesh, it is voxelized and then triangulated back into a mesh. Using rasterization, slices of the mesh are created from different angles and at different depths. To reduce the number of slices they are filtered based on their occlusion properties. At this point the slices are still stored as bitmaps. These bitmaps are converted to 2D polygon using edge loop extraction and simplification. For each slice a progressive simplification chain is build by incrementally removing the edge vertex of which the adjacent triangle has the lowest surface area. Finally, a greedy algorithm picks the optimal set of simplified slices that fits within the triangle budget.

3.4 Sparse Voxel Directed Acyclic Graphs

Sparse voxel octrees are a popular representation of volumetric data sets. This section will focus on reducing the storage requirements of sparse voxel octrees using directed acyclic graph (DAG) compression. Note that this is in no way an exhaustive list of all the literature in the field of octree compression.

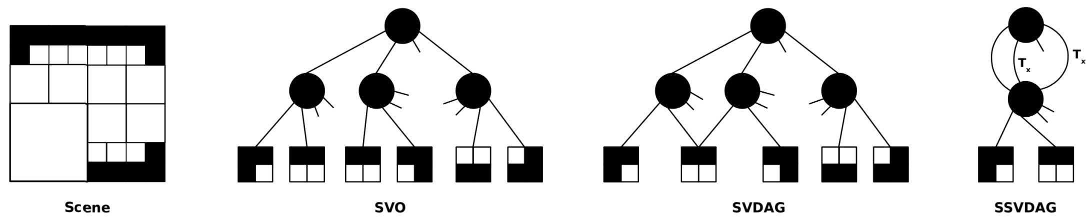


Figure 3: Illustration of how a sparse voxel octree (SVO) may represented using DAG and symmetry aware DAG compression. Source: Villanueva et al. [2016]

Webber and Dillencourt [1989] was the first to recognize that the repetition in a quad tree could be exploited to reduce its storage requirements. The paper introduces Common Subtree Merging (CSM). The goal is to find duplicate subtrees and replace them by a single instance. This creates a structure which is not a tree anymore because multiple nodes may share the same child. This is why the compression technique is also referred to as directed acyclic graph (DAG) compression.

The CSM technique can be trivially extended to an octree structure, as was first done in **Parker and Udeshi [2003]**. Unlike the original CSM paper, Parker and Udeshi [2003] explain the algorithm they used to convert a sparse voxel octree into a DAG. The algorithm traverses the octree depth first and construct a DAG at the same time. Visited nodes are inserted into the DAG if they unique; duplicate nodes are replaced by pointers to a single instance of those node. Two nodes are duplicates if all their child pointers are the same.

Kämpe et al. [2013] adds out-of-core construction of the DAG. A sparse voxel octree is constructed using only a few levels, and for each leaf node a subtree is constructed. For each subtree the DAG conversion is applied separately using an algorithm similar to that of Parker and Udeshi [2003]. The same algorithm is then applied to the top level of the tree. The resulting DAG is the same as if it were constructed in one pass. The authors only store pointers to non-empty children which saves a lot of space.

Villanueva et al. [2016] recognizes that a sparse voxel octree contains more common subtrees when considering reflective transformations. Subtrees are matched not only if they are duplicates, but also if they are the same if one undergoes a reflective transformation (mirroring along the x, y, and/or z axis).

This requires that nodes store the transformations (3 bits per child) that each child undergoes. The authors show that despite these extra bits the resulting graph uses less memory than a regular Sparse Voxel DAG (Kämpe et al. [2013]).

The authors found that 10% of the nodes that was referenced by nearly 90% of the pointers. Therefor they introduce a compact memory representation that replaces frequently referenced pointers by 16 bit offsets (and other pointers by 32 bit offsets). With these optimizations a 2x memory reduction was achieved compared to Kämpe et al. [2013].

3.5 Depth Map for Ray Tracing

Besides compressed sparse voxel octrees, compressed depth maps will also be evaluated as a way of finding approximate intersections. This section will go over the compression technique proposed in Scandolo et al. [2016] as well as some of its previous work.

3.5.1 Shadow map compression

In rasterized graphics, both video games and Reyes, shadow maps have been the most popular way of determining whether a point in space lies in shadow with respect to a light source. A shadow map is a depth buffer (z-buffer) that is constructed from the lights point of view. Shadow queries can be answered by transforming the position of a query into “light space“ and comparing it to the value stored in the shadow map.

Common problems with this technique are self-shadowing and self-unshadowing. Self-shadowing occurs when a surface is shadowed by itself because of numerical inaccuracies or sample position mismatches between the shadow map and the frame buffer. These same problems also cause self-unshadowing, which happens at the edge of a silhouette (as seen from the camera) when points are deemed visible from the light, while in fact they should be in shadow.

A popular approach to fixing these issues is to add a bias to the values read from the shadow map (which “moves“ the shadow map closer to the camera). Adding a bias was first introduced by **Reeves et al. [1987]**. For each query the shadow map is sampled multiple times using random jittering and with a bias that is stochastically chosen from an user defined range. Although this reduces the artifacts caused by adding a bias (“floating“ geometry), using a fixed bias has become more popular in the gaming industry.

Woo [1992] was the first to take advantage of the fact that shadow maps do not have to store the exact depth of the first surface. The only guarantee that needs to be made is that the depth value stored in the shadow map is farther than the closest surface (so that it is in light), but smaller than the second surface (so that it is in shadow). This is what we will refer to as

dual shadow mapping.

The paper uses the midpoint between the first and second surface as depth value in the final shadow map. This greatly reduces self shadowing although there are some edge cases where this problem may still occur.

Weiskopf and Ertl [2003] shows that the previous midpoint technique can be combined with a fixed bias for better results. The resulting bias is calculated as $z_{bias}(z_1, z_2) = min(\frac{z_2 - z_1}{2}, z_1 + z_{fixed_bias})$ where z_1 and z_2 are the depth of the first and second surface respectively. The authors also show that instead of using the second surface, which is back facing (and thus always in shadow), the third surface (which is front facing) could also be used as maximum depth.

Arvo and Hirvikorpi [2005] was the first to exploit dual shadow mapping for compression purposes. The idea behind the paper is to represent scan lines using a small number of line segments.

Construction of this structure requires finding the intersection point between each scanline and the edges of the closest and second closest triangles. This results in two polyline collections: the light- and the shadow frontier. The goal is then to find a polyline between those two frontiers.

The technique suggested in the paper creates a new polyline equidistant from both frontiers. It then uses a left-to-right sweep over the polyline, removing vertices if doing so would not cause the polyline to intersect with either frontier. The resulting polyline is stored in a binary tree for fast lookups.

Ritschel et al. [2007] uses a similar technique to compress a collection of equally sized shadow maps. Instead of compressing along scanlines, the authors suggest compressing along the same sample of different shadow maps. The shadow maps should be sorted by coherence for an optimal result; meaning that samples in adjacent shadow maps should store roughly the same depth interval.

Another difference from Arvo and Hirvikorpi [2005] is that depth intervals are stored instead of line segments. Conceptually intervals can be interpreted as axis aligned (fixed depth) line segments. These intervals are found using a simple $O(N)$ sweep algorithm.

Scandolo et al. [2016] was the first to utilize dual shadow mapping for compression along more than a single axis. Both image dimensions are utilized for compression by building

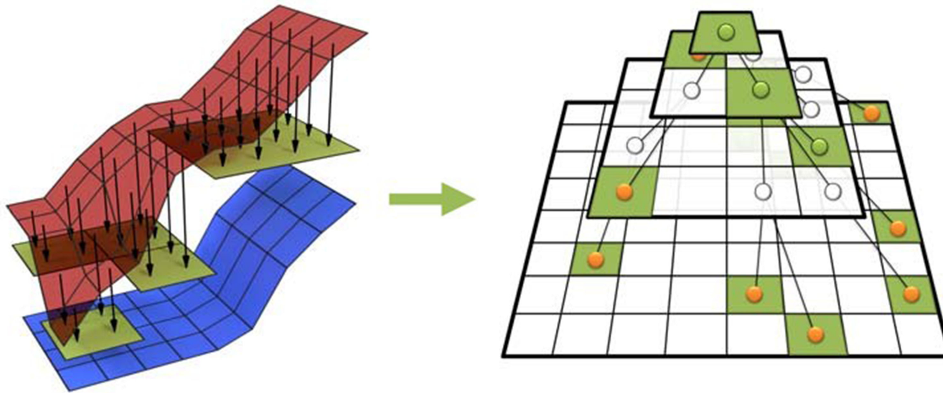


Figure 4: An illustration of how a dual shadow mapping can be compressed efficiently using a quadtree-like structure. Source: Scandolo et al. [2016]

a special kind of quadtree over the image domain (figure 4). This structure differs from a regular quadtree in that inner node may store values. These value transcend down the tree and indicate the value of missing children.

Both a top-down and a bottom-up construction technique are provided. Top-down construction may yield slightly better results while the bottom-up construction lends itself better to parallel execution. Both approaches also support compressing multiple shadow maps together by stacking them on top of each other, resulting in an octree instead of a quadtree.

3.5.2 Ray tracing using depth maps

Although depth maps are traditionally only used in rasterization (as shadow maps), they can also be used for ray tracing. **McGuire et al. [2017]** introduce light field probes that not only store radiance but also geometric information that can be used for world-space ray traversal.

Light probes are placed on a regular grid inside the scene's bounding box. Each probe maps directions ω to the radial distance to the closest point in that direction, as well as the normal vector at that point. These distances are stored using octahedral parameterization maps which are derived from high resolution cube maps. This representation uses less memory than regular cube maps and preserves the piecewise-linear projection required for efficient ray marching.

Ray tracing with respect to a light probe is implemented by transforming the 3D ray into 2D line segments (with endpoints where they intersect octahedron edges). These line segments are rasterized during ray marching. The depth map samples describe planar patches oriented towards the normal stored in those samples. Ray marching can thus have

3 different types of results: hit, miss or unresolvable. The latter occurs when a ray passes behind a voxel without hitting it. To resolve those ray queries the paper suggests querying one of the other nearby probes.

References

Attila T Áfra and László Szirmay-Kalos. Stackless multi-bvh traversal for cpu, mic and gpu ray tracing. In *Computer Graphics Forum*, volume 33, pages 129–140. Wiley Online Library, 2014.

Timo Aila. Surrender umbra: A visibility determination framework for dynamic environments. *Master's thesis, Helsinki University of Technology*, 2000.

Solid Angle. Arnold. `https://www.solidangle.com/about/`. Accessed May 23, 2018.

Jukka Arvo and Mika Hirvikorpi. Compressed shadow maps. *The Visual Computer*, 21(3): 125–138, 2005.

Jacco Bikker. Improving data locality for efficient in-core path tracing. In *Computer Graphics Forum*, volume 31, pages 1936–1947. Wiley Online Library, 2012.

Pere Brunet, Isabel Navazo, Jarek Rossignac, and Carlos Saona-Vázquez. Hoops: 3d curves as conservative occluders for cell-visibility. In *Computer Graphics Forum*, volume 20, pages 431–442. Wiley Online Library, 2001.

Brian Budge, Tony Bernardin, Jeff A Stuart, Shubhabrata Sengupta, Kenneth I Joy, and John D Owens. Out-of-core data management for path tracing on hybrid resources. In *Computer Graphics Forum*, volume 28, pages 385–396. Wiley Online Library, 2009.

Per H Christensen, Julian Fong, David M Laur, and Dana Batali. Ray tracing for the moviecars'. In *Interactive Ray Tracing 2006, IEEE Symposium on*, pages 1–6. IEEE, 2006.

Per H Christensen, Wojciech Jarosz, et al. The path to path-traced movies. *Foundations and Trends® in Computer Graphics and Vision*, 10(2):103–175, 2016.

Robert L Cook, Loren Carpenter, and Edwin Catmull. The reyes image rendering architecture. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 95–102. ACM, 1987.

Satyan Coorg and Seth Teller. Real-time occlusion culling for models with large occluders. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 83–ff. ACM, 1997.

Nick Darnell. Oxel. http://www.nickdarnell.com/hierarchical-z-buffer-occlusion-culling-generating-occlusion-volumes/, 2011. Accessed: 2018-05-29.

David E DeMarle, Christiaan P Gribble, and Steven G Parker. Memory-savvy distributed interactive ray tracing. In *Egpgv*, pages 93–100. Citeseer, 2004.

Christian Eisenacher, Gregory Nichols, Andrew Selle, and Brent Burley. Sorted deferred shading for production path tracing. In *Computer Graphics Forum*, volume 32, pages 125–132. Wiley Online Library, 2013.

Luca Fascione, Johannes Hanika, Marcos Fajardo, Per Christensen, Brent Burley, and Brian Green. Path tracing in production-part 1: production renderers. In *ACM SIGGRAPH 2017 Courses*, page 13. ACM, 2017.

TG Gasparian. Fast divergent ray traversal by batching rays in a bvh. Master's thesis, 2016.

Rick Germs and Frederik W Jansen. Geometric simplification for efficient occlusion culling in urban scenes. 2001.

Ned Greene, Michael Kass, and Gavin Miller. Hierarchical z-buffer visibility. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 231–238. ACM, 1993.

Ulrich Haar and Sebastian Aaltonen. Gpu-driven rendering pipelines. Siggraph, 2015. URL http://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pdf.

Vlastimil Havran, Jirí Bittner, and Jirí Zára. Ray tracing with rope trees. In *14th Spring Conference on Computer Graphics*, pages 130–140, 1998.

James T Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. High resolution sparse voxel dags. *ACM Transactions on Graphics (TOG)*, 32(4):101, 2013.

Vladlen Koltun, Yiorgos Chrysanthou, and Daniel Cohen-Or. Virtual occluders: An efficient intermediate pvs representation. In *Rendering Techniques 2000*, pages 59–70. Springer, 2000.

Fei-Ah Law and Tiow-Seng Tan. Preprocessing occlusion for real-time selective refinement. In *Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 47–53. ACM, 1999.

Mark Lee, Brian Green, Feng Xie, and Eric Tabellion. Vectorized production path tracing. In *Proceedings of High Performance Graphics*, page 10. ACM, 2017.

Morgan McGuire, Mike Mara, Derek Nowrouzezahrai, and David Luebke. Real-time global illumination using precomputed light field probes. In *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, page 2. ACM, 2017.

Paul Arthur Navratil, Donald S Fussell, Calvin Lin, and William R Mark. Dynamic ray scheduling to improve ray coherence and bandwidth utilization. In *Interactive Ray Tracing, 2007. RT'07. IEEE Symposium on*, pages 95–104. IEEE, 2007.

Jacopo Pantaleoni, Luca Fascione, Martin Hill, and Timo Aila. Pantaray: fast ray-traced occlusion caching of massive scenes. *ACM Transactions on Graphics (TOG)*, 29(4):37, 2010.

Eric Parker and Tushar Udeshi. Exploiting self-similarity in geometry for voxel based solid modeling. In *Proceedings of the eighth ACM symposium on Solid modeling and applications*, pages 157–166. ACM, 2003.

Matt Pharr, Craig Kolb, Reid Gershbein, and Pat Hanrahan. Rendering complex scenes with memory-coherent ray tracing. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 101–108. ACM Press/Addison-Wesley Publishing Co., 1997.

William T Reeves, David H Salesin, and Robert L Cook. Rendering antialiased shadows with depth maps. In *ACM Siggraph Computer Graphics*, volume 21, pages 283–291. ACM, 1987.

Tobias Ritschel, Thorsten Grosch, Jan Kautz, and Stefan Müeller. Interactive illumination with coherent shadow maps. In *Proceedings of the 18th Eurographics conference on Rendering Techniques*, pages 61–72. Eurographics Association, 2007.

Leonardo Scandolo, Pablo Bauszat, and Elmar Eisemann. Compressed multiresolution hierarchies for high-quality precomputed shadows. In *Computer Graphics Forum*, volume 35, pages 331–340. Wiley Online Library, 2016.

Ari Silvennoinen, Hannu Saransaari, Samuli Laine, and Jaakko Lehtinen. Occluder simplification using planar sections. In *Computer Graphics Forum*, volume 33, pages 235–245. Wiley Online Library, 2014.

John A Tsakok. Faster incoherent rays: Multi-bvh ray stream tracing. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 151–158. ACM, 2009.

Alberto Jaspe Villanueva, Fabio Marton, and Enrico Gobbetti. Ssvdags: symmetry-aware sparse voxel dags. In *Proceedings of the 20th ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 7–14. ACM, 2016.

Ingo Wald, Philipp Slusallek, and Carsten Benthin. Interactive distributed ray tracing of highly complex models. In *Rendering Techniques 2001*, pages 277–288. Springer, 2001a.

Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive rendering with coherent ray tracing. In *Computer graphics forum*, volume 20, pages 153–165. Wiley Online Library, 2001b.

Robert E Webber and Michael B Dillencourt. Compressing quadtrees via common subtree merging. *Pattern recognition letters*, 9(3):193–200, 1989.

Daniel Weiskopf and Thomas Ertl. Shadow mapping based on dual depth layers. In *Proceedings of Eurographics*, volume 3, pages 53–60, 2003.

Graham Wihlidal. Optimizing the graphics pipeline with compute. Game Developer Conference, 2016. URL `https://www.gdcvault.com/play/1023109/Optimizing-the-Graphics-Pipeline-With`.

Andrew Woo. The shadow depth map revisited. In *Graphics Gems III (IBM Version)*, pages 338–342. Elsevier, 1992.

Hansong Zhang, Dinesh Manocha, Tom Hudson, and Kenneth E Hoff III. Visibility culling using hierarchical occlusion maps. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 77–88. ACM Press/Addison-Wesley Publishing Co., 1997.