# Occlusion Culling in Batched Ray Traversal

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

January 21, 2019

**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-

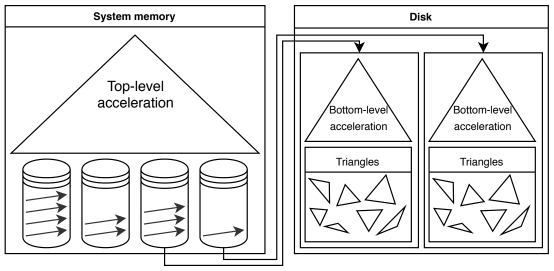

**Figure 2:** The two level acceleration structure hierarchy used by (out-of-core) batched ray traversal. In our system the ray batches are always stored in memory.

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.

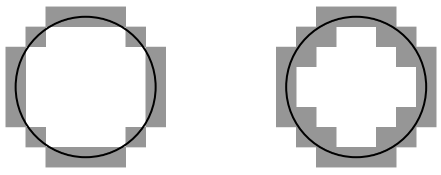

**Figure 3:** Thin- (left) and conservative (right) voxelization of a circle in 2D.

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.

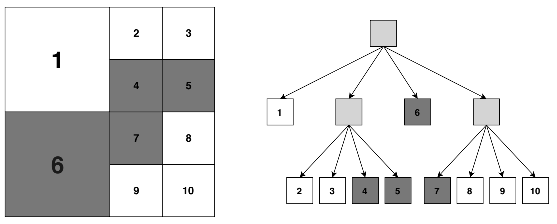

**Figure 4:** A sparse voxel octree saves memory by not storing homogeneous parts of the voxel grid. The illustration shows a sparse voxel quadtree for simplicity.

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).

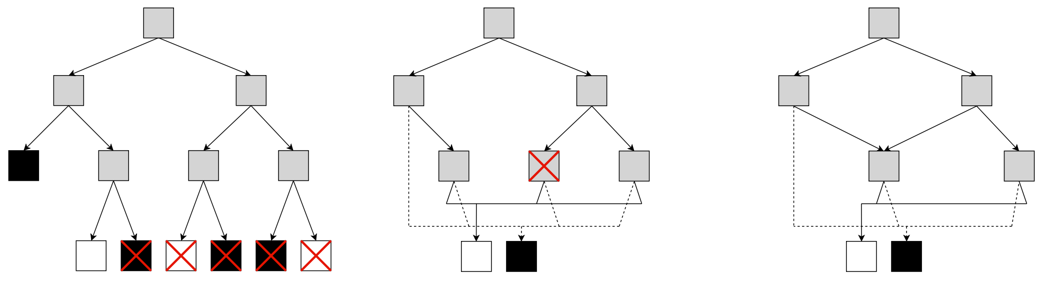

**Figure 5:** Compressing a sparse voxel octree into a sparse voxel directed acyclic graph, illustrated using a binary tree for clarity. Duplicate nodes are repeatedly replaced by a single instance starting from the bottom up.

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. However, 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).

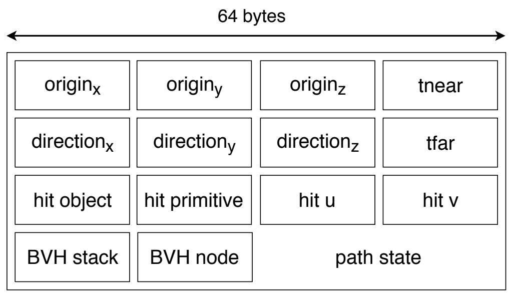

**Figure 6:** The data layout of a batched ray.

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

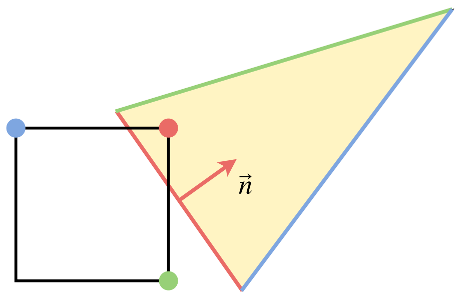

**Figure 7:** Voxelization: the 3 critical points with respect to the triangle's edges. In this image all critical points lie on the correct side of their respective edges.

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($n$)
    $r \leftarrow Node()$
    for $1 \leq i \leq 8$ do
        if IS_INNER_NODE($i$) then
            $c \leftarrow n.children[i]$
            $r.children[i] \leftarrow$ 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 $\begin{bmatrix}1 & 1 & 1\end{bmatrix}^\mathsf{T}$ to $\begin{bmatrix}2 & 2 & 2\end{bmatrix}^\mathsf{T}$. 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

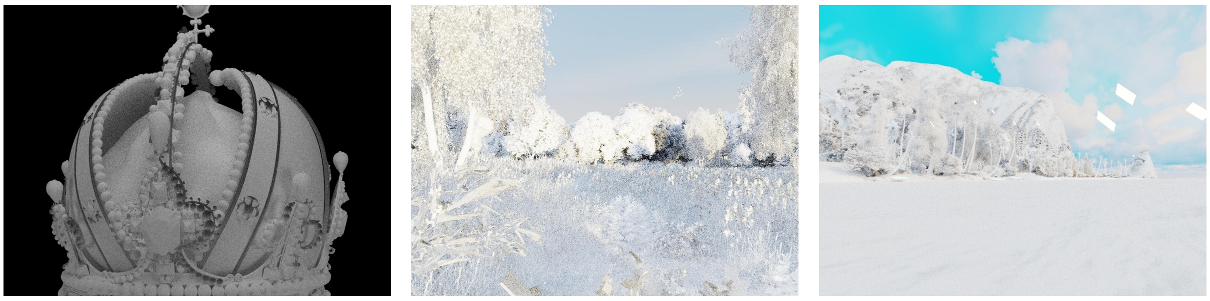
**Figure 8:** The scenes that were used for testing rendered with our path tracer. From left to right: crown, landscape and Island.

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.

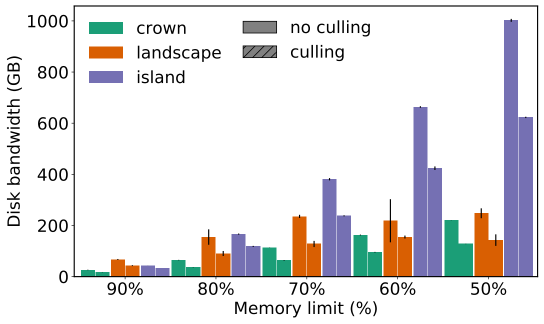

**Figure 9:** The effect of occlusion culling on the disk bandwidth used for rendering (including tail).

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

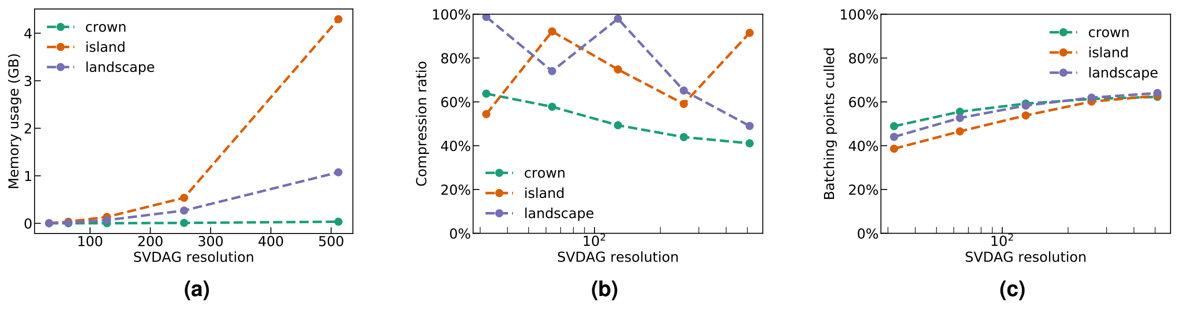

**Figure 10:** Memory usage (a), compression ratio vs SVO (b) and culling effectiveness (c) of the SVDAGs.

(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.

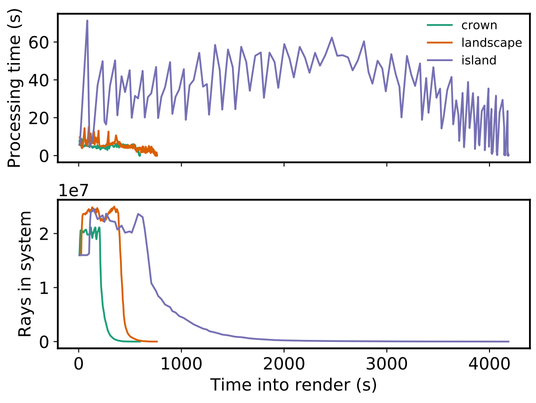

**Figure 11:** Most time is spent processing the final few rays when latency hiding is not possible.

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

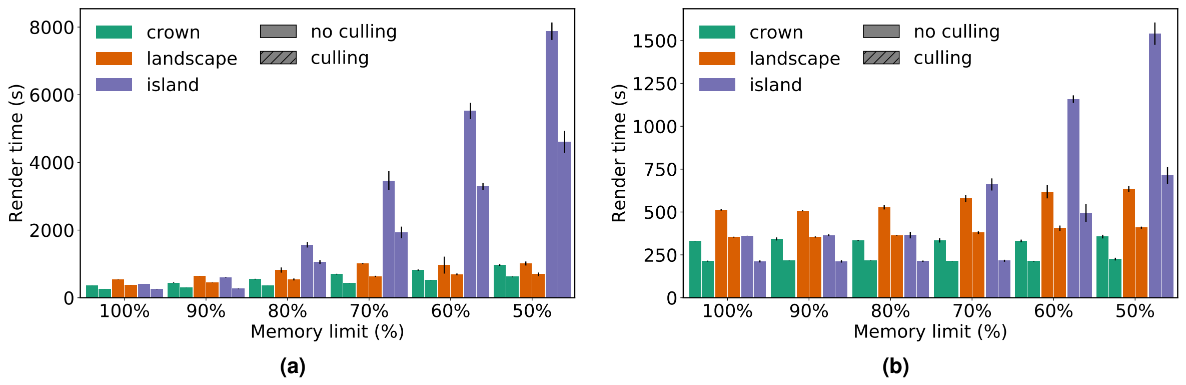

**Figure 12:** The impact of occlusion culling on the total render time at different memory limits. (a) shows the results when all rays are processed. (b) shows the results when the renderer would not process the tail (by discarding the last 12 million rays).

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. Dynamic 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++)

```cpp
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;
            }
        }
    }
}



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