Volume xx (200y), Number z, pp. 1–8

Accelerating ray tracing with origin offsets

T. Zoet (ICA-4275438)

University of Utrecht, Netherlands

Abstract

In ray tracing, a large fraction of rendering time consists of traversal of the acceleration structure. Most methods that aim to accelerate this process do so by skipping nodes high in the tree. We propose offsetting ray origins just before starting traversal. This allows rays to be moved out of nodes deep in the tree, avoiding the scattered, cache-unfriendly memory access associated with these rarely visited nodes. We do this using a precomputed set of hemispheres placed on the geometry surface which are guaranteed not to contain any geometry, thus allowing rays starting inside a hemisphere to be moved to the hemispheres boundary.

CCS Concepts

• ***Computing methodologies*** $\rightarrow$ ***Ray tracing;***

1. Introduction

Ray tracing has wide applications. It is used in collision detection algorithms, simulations of physical phenomena and graphics. Recent advancements in ray tracing hardware have made real-time ray tracing accessible to the masses. In the past year, the first video games that employ ray tracing in part of their rendering algorithm were released.

To perform efficient intersection tests between rays and a collection of objects, acceleration structures are used. The two most common acceleration structures in modern high-performance renderers, the kD-tree and bounding volume hierarchy (BVH), aim to bring the average case time complexity of tracing a single ray down to near logarithmic complexity.

In this paper, we improve ray tracing performance by offsetting extension rays just before starting traversal of the BVH. Many rays visit the leaf node they originate from without finding an intersection. These visits are expensive. Often the node and primitive data need to be fetched from main memory. Access patterns are scattered and there are many leaf nodes, making them unlikely to be in the cache. Ray-triangle intersection tests also have a high cost.

In a preprocessing phase, a set of hemispheres is placed on the surface of the scene geometry. Each hemisphere is given a radius such that it is guaranteed not to contain any intersecting geometry. Given a ray, we fetch the nearest hemisphere(s) and perform low cost ray-sphere intersection tests to determine a distance along which it is safe to move the ray origin. If the origin is moved far enough, the ray is moved out of the leaf node it originated from and potentially one or more nodes higher in the BVH. When the overhead of offset calculation is outweighed by the reduced traversal cost, the rendering process will become faster.

2. Related work

The average time complexity of tracing a single ray is close to logarithmic time. This is achieved through various data structures and algorithms. These methods can be put into roughly 3 categories:

1. Acceleration structures. The most commonly used acceleration structures, the kD-tree and BVH, aim to bring the cost of tracing a single ray from linear to near logarithmic time. They do this by recursively partitioning space and objects, respectively.
2. Amortization of traversal cost. By grouping rays together in packets, the cost of fetching the acceleration structure data can be amortized over many rays.
3. More efficient traversal of the acceleration structure. These methods often make use of high-level knowledge about rays, such as the difference between nearest-hit and any-hit (occlusion) rays, or connectivity between rays, to guide traversal towards parts of the tree that are more likely to intersect the rays.

The above categorization will not cover all methods. Some can be put into more than one category, whereas others will defy all categorization. The following sections summarize the main works of each category, providing a context for the contributions in this paper.

2.1. Acceleration structures

Fujimoto et al. [FTI86] use uniform spatial subdivision to partition the scene into regular cells, putting objects in the cells they overlap. This grid is traversed with a ray. The ray is intersected with all

the objects in each cell it passes. In contrast, Glassner [Gla84] uses an adaptive scheme, placing objects in the recursively partitioned octree. Cleary and Wyvill [CW88] provide an extensive analysis of the uniform grid, showing that it, at that time, outperformed hierarchical methods by an order of magnitude.

Earlier, the kD-tree (or binary tree) was introduced by Bentley [Ben75], although not in the context of ray tracing. The kD-tree has found wide application, especially after the introduction of $O(n\ log\ n)$ construction algorithms that result in high-quality trees. Goldsmith and Salmon [GS87] introduce a simple heuristic to guide the construction of spatial subdivision trees. They estimate the cost of adding an object to the tree using the number of children and surface area of newly created nodes. By evaluating this cost for all splitting options (e.g. all 3 axes for spatial median kD-trees) it is possible to select the optimal partitioning, under the assumption of a uniform ray distribution. MacDonald and Booth [MB90] extend this surface area heuristic (SAH) to include the number of objects contained in nodes, resulting in more accurate cost estimates. While the SAH is a greedy heuristic, only considering the 'current' split without looking at potential splits in deeper levels, MacDonald and Booth show that it performs much better than using a fixed split order (e.g. always along the longest axis or by cycling through x, y and z).

Clark [Cla76] introduced the BVH. In contrast to the kD-tree, the BVH partitions objects, not space. Construction of the BVH is similar to the kD-tree, and allows the use of e.g. the surface area heuristic. The main difference is how the bounds of the children of a node are defined. The kD-tree places a split plane along one of the axes of a node, assigning each side to one of its children. Objects are then assigned to the child whose bounds they overlap. In the BVH, objects are partitioned based on their centroid relative to the split plane. The bounds of each child node are then recalculated as the union of the bounds of all objects assigned to them. As such, the children of a node can have bounds that either overlap or are completely disjoint.

Havran [Hav00] provides an in-depth analysis of a wide range of acceleration structures, including the kD-tree and BVH, and concludes that the kD-tree gives the best ray tracing performance. Wald [Wal07] greatly improves the speed of BVH construction and Stich [SFD09] solves the problem of scenes with non-uniformly sized triangles. Together, this closes the performance gap between the kD-tree and BVH. Because of its flexibility, the BVH has become the most widely used acceleration structure in modern renderers.

2.2. Amortization

Wald et al. [WSBW01] group several rays together in a single packet. The size of this packet is generally chosen to be the width of the vector registers on the CPU. The acceleration structure is then traversed with the packet. If one of the rays in the packet intersects a subtree, the entire packet will visit those nodes. When rays are coherent (i.e. share a similar origin and direction) they would have followed a similar path through the tree had they been traced individually. By tracing them together the cost of fetching the nodes from memory is shared between the rays. Additionally, intersection tests between rays and bounding boxes and triangles can be performed using vector instructions. Overall, packet traversal (4-wide) offers a 2-3x performance improvement when compared to single ray traversal.

Reshetov et al. [RSH05] describe the Multi-Level Ray Tracing Algorithm (MLRTA). They represent a potentially large group of rays with a beam. This beam is used for the initial traversal of the acceleration structure. This traversal looks for an entry point into the tree: the first node for which both subtrees contain a leaf node that intersect the beam. Once the entry point has been found, the individual rays start traversal at this point. Depending on the coherency of the rays in the beam, the entry point can be quite deep, allowing the rays to skip a significant portion of initial traversal. Reshetov et al. report performance an order of magnitude above then state-of-the renderers, allowing interactive rendering on commodity hardware. The process of finding entry points was further optimized by Fowler et al. [FCM09]. They find deeper entry points, and find these with fewer traversal steps.

Overbeck et al. [ORM08] use packets of up to 1024 rays and propose a traversal algorithm specifically aimed at reflection and refraction rays. These rays suffer from quickly degrading coherence as the number of bounces for each path increases.

2.3. Improved traversal

There is an important distinction to make between nearest-hit rays and occlusion rays. Nearest-hit rays (e.g. primary rays, reflection rays, diffuse rays) require finding the exact closest point of intersection. This means that when a choice needs to be made between visiting two nodes, the nearest node should always be traversed first, leaving no room for more optimal traversal orders. This is not the case for occlusion rays (e.g. shadow rays, ambient occlusion rays), which only require determining whether there is any intersection along the ray. The exact order in which nodes are visited is not relevant. Boulos and Haines [BH10] show that occlusion rays can often dominate the total rendering time. It is therefore worthwhile to devise different traversal strategies for occlusion rays.

MacDonald and Booth [MB90] were the first to propose an alternative traversal algorithm. They add neighbor links, or ropes, between the leaves of a kD-tree. Once a ray has been traced, any extension ray originating at the intersection point can then be traced by following the links between nodes, starting at the leaf in which traversal previously ended. This traversal method prevents visiting many internal nodes, reducing the overall traversal time at the cost of additional memory usage. Note that this method does not work for the BVH, where nodes can overlap or have gaps between them. Havran et al. [HBZ98] show that ropes result in a 10-20% reduction of total rendering time, compared to standard kD-tree traversal.

Havran and Bittner [HB07] improve traversal times by augmenting the internal nodes of a kD-tree. Their work mainly aims to reduce the memory overhead introduced by ropes. This is achieved by storing bounding boxes in the nodes at every n^{th} level, where n is a parameter of the construction algorithm. When tracing an extension ray, traversal starts at the last visited augmented node. If traversing the subtree does not result in an intersection, the exit point of the ray relative to the last visited augmented node is calculated. Then, the first augmented node containing the exit point at

a higher (shallower) level is used to restart traversal. If the extension ray would have followed a similar path to the previous ray, the number of visited nodes is reduced. The authors show considerable speedups of up to 35%, however in shallow scenes the benefits are outweighed by the overhead.

Hendrich et al. [HPMB19] use convex frustum shafts to cull parts of a BVH. They subdivide the scene into regular voxels. For each voxel a number of shafts in many directions is created, all fully containing the voxel. These shafts are then intersected with the BVH to construct the candidate list, the deepest nodes that intersect the shaft. When tracing a ray, the frustum shaft containing this ray is retrieved. If such a shaft exists, the entire candidate list is placed on the traversal stack, instead of the root node of the BVH, effectively skipping all nodes above the candidates. Hendrich et al. show that this saves on average 42% in traversal steps.

Djeu et al. [Djeu09] reduce the traversal time of occlusion rays. kD-tree leaf nodes contained entirely by the geometry are marked as volumetric occluders. Traversal is terminated when these nodes are encountered. Tracing occluded rays is sped up by several factors in some scenes. However, un-occluded rays incur a penalty. Additionally, the scene geometry must be manifold to be able to mark nodes as occluders.

Ize and Hansen [IH11] describe a more robust method for accelerating occlusion rays. They introduce the Ray Termination Surface Area Heuristic (RTSAH). The RTSAH drops the assumption of no ray termination that is used by standard SAH [GS87]. Using the new cost function, they determine a traversal order for each pair of sibling nodes in an existing BVH, giving priority to the one with the lowest cost. This traversal decision can be stored in the parent node using a single bit, generally requiring no memory overhead. When tracing an occlusion ray, and this ray needs to visit both children of the current node, the bit is used to determine the optimal traversal path. Overall, the RTSAH results in up to 50% reduction of visited nodes.

The work presented in this paper also falls under the third category. We build a set of hemispheres on top of the acceleration structure that are guaranteed not to contain any geometry. We then use the hemispheres to calculate an offset for each extension ray, quickly skipping deep parts of the acceleration structure hierarchy.

3. Offsetting rays

Testing a ray against a single leaf node is more expensive than a single internal node. This has two main reasons. First, leaf nodes are visited less often than the shallower internal nodes. As a result, they are less likely to be in the cache. The same holds for the primitive data in the leaf node. Second, intersecting a ray with the primitives is much more computationally expensive than a test against an axis-aligned bounding box. Most methods that aim to accelerate traversal focus on skipping internal nodes or amortizing traversal costs over multiple rays and quickly reaching the leaf nodes. It could be beneficial to focus on skipping leaf nodes, avoiding a lot of scattered memory access and some expensive intersection tests.

Consider rendering an opaque sphere, represented using many triangles. After the primary rays originating from the camera have been traced, any extension ray starting on the sphere's surface will not hit any additional geometry: a sphere is a convex object and all rays will be pointed away from the surface. Regardless, traversal will first traverse down into the leaf node where the previous ray ended, since the new ray's origin is contained in the leaf's bounds. Offsetting the ray origin along a short distance could be enough to move it out of the leaf node. A larger offset would then cull additional nodes above and adjacent to the leaf node. While a sphere is the best-case scenario due to its convexity, the idea of offsetting ray origins extends to more complex scenes. Figure 1 shows a 2D example with a circle.

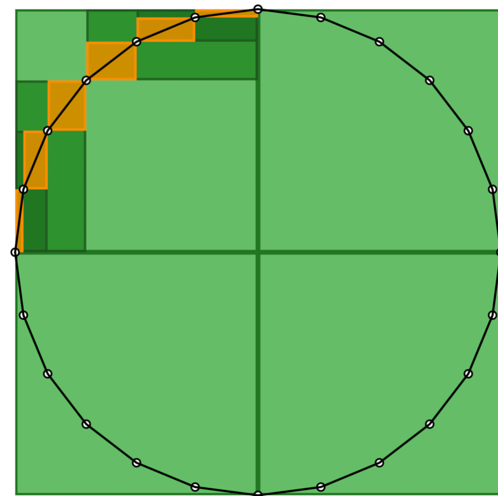

Figure 1: *Part of a BVH built on a circle. In shades of green internal nodes. In orange leaf nodes. Note that the distance required to move a ray starting on the surface out of a leaf node is a fraction of the scene size.*

By offsetting rays, we are essentially performing part of the work that is done by acceleration structure traversal: cutting of part of the search space that does not contain geometry. If this is done more efficiently with an alternative method, rendering will become faster.

To calculate an offset, we will need some auxiliary data structure that, when queried with a ray, moves the ray's origin. This data structure should use little additional memory to prevent degradation of performance due to increased memory fetches, which is explicitly what we are trying to prevent. Additionally, the query should have a low computational cost. Assuming single-sided triangles and no transparent materials, all rays will start on one side of the geometry. Given a ray starting on a triangle, we wish to calculate an offset on one side of the plane (as defined by the triangle's location and normal) in any given direction. In other words, we are looking for something very much like an empty (non-occluded) hemisphere around the ray origin. This naturally leads to using actual hemispheres stored on the geometry to offset the rays.

The following happens during a preprocessing phase. We place one or more hemispheres on each triangle. At each hemisphere location, we query the scene for the nearest intersection on the positive side of the plane. This can be done using the existing accel-

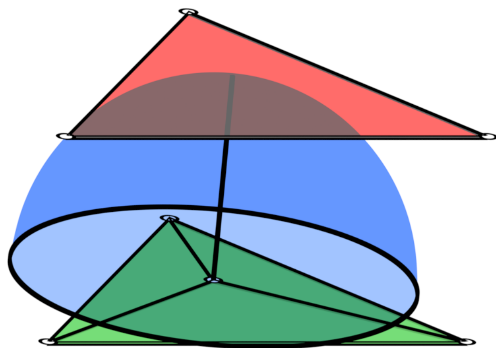

Figure 2: *A hemisphere (blue) placed in the center of a triangle (green). The radius is limited by the triangle (red) above it.*

eration structure. The distance to the intersection is then the radius of the hemisphere. The volume of the hemisphere is guaranteed not to contain any occluding geometry. During rendering, querying the set of hemispheres is straightforward. Given a ray, we retrieve the hemispheres attached to the triangle it originates from. The offset is then calculated using basic ray-sphere intersection tests. After offsetting the ray origin, traversal can be started.

The next two sections will describe various potential hemisphere placement strategies. The first section covers how the hemisphere locations can be chosen such that they best achieve our goal: moving the rays out of as many leaf nodes as possible. While the most optimal solution is of course placing an infinite number of hemispheres, when it comes to the actual effects on performance as measured in execution time a balance between saving traversals and limiting overhead needs to be found. The second section describes low-level optimizations to reduce overhead, such as a carefully chosen memory layout, caching behaviour and vectorization.

3.1. Hemisphere sets

If we assume rays to start at a uniform random location on the triangle, the probability of a ray receiving any offset is directly proportional to the relative area of the triangle that is covered by the hemisphere(s). The average offset is then guided by the radii of the hemispheres. We wish to move as many rays as possible out of the leaf node they start in, while at the same time limiting the number of hemispheres. What follows are five placement strategies that balance these two things.

Center Sphere Set. The first and simplest method is to store a single hemisphere located at the center of each triangle. At this location the hemisphere will in general have the most potential for moving a ray out of the triangle's bounding box. Figure 2 shows an example of a centered hemisphere.

Vertex Sphere Set. We can also place a hemisphere on each triangle vertex. Compared to the previous solution this method is more resistant to occluding geometry at the center. If only one area of the triangle is occluded there is still potential for a large enough hemisphere being placed on one or more of the vertices to offset nearby rays far enough to leave the triangle bounds. There is an important downside: if a vertex is part of a concave corner of the surface geometry the radius of the hemisphere will be 0.

Median Sphere Set. To solve the problem of concave geometry we can move the hemispheres closer to the triangle center. One option is to place the hemispheres between each vertex and the triangle center. They would then essentially be placed along the medians, the line segments from the vertices to the midpoint of the opposite edges. Together, the hemispheres can cover a large area of the triangle. Optionally, the center hemisphere could be stored as well for larger coverage.

Polygon Sphere Set. In some situations the center sphere set could perform better with only a slight modification. Consider a scene with many groups of adjacent triangles lying in the same plane, essentially forming (non-convex) polygons. A triangle in such a group could make use of the hemisphere of one of its neighbours. This neighbouring hemisphere might be much larger because the triangle itself lies closer to an occluding part of the scene than the neighbour. If the neighbouring hemisphere covers a large part of the triangle, it can offset many rays for much larger distances than the triangle's own hemisphere. Selecting the optimal hemisphere from all neighbours is a hard optimization problem. A potentially good approximation would be to select the largest of all hemispheres and assigning this to each triangle of the polygon.

Random Sphere Set. Instead of placing hemispheres at fixed locations, it is also possible to distribute them randomly. With a uniform distribution, the hemispheres can give good coverage of the triangle area. Most importantly, this has some consequences for storage that are covered in the next section.

3.2. Optimization

By placing a fixed number of hemispheres per triangle, retrieval has constant time complexity. It also allows keeping the querying code simple, minimizing the strain on the instruction cache. Furthermore, it opens up options for aligning the data to cache lines and efficient vectorized intersection tests.

Center Sphere Set. The center sphere set is the method with the lowest storage requirements. Assuming we store the hemispheres using single precision scalars (floats), this would require a mere 16 bytes: 4 for the radius, 4 for each of the 3 axes. This is only a quarter of the most common cache line size (64), so retrieving a single hemisphere will require at most one memory fetch. It is even possible to only store the radius and recalculate the center using the vertices. This would require only 4 bytes per hemisphere and has the potential to reduce the number of fetches even further. Whether or not this is actually the case will depend on the implementation details of the renderer. If each extension ray is calculated directly after tracing the previous ray and the triangle data is still in the cache, recalculation might be faster.

Vertex Sphere Set. This method uses 3 times more memory than the center sphere set. Depending on whether the vertices are reused, the overhead will be 12 or 48 bytes per triangle. As a result, the data for a single triangle is potentially stored across cache line boundaries. This can be solved by over-aligning the data to 16 or 64 bytes (increasing memory overhead by a third), but whether this will be

beneficial will depend on the exact scene that is being rendered as well as the hardware that is used. Another option is to store the radii directly with the vertex data. Obviously, this change will impact all other parts of the renderer. It could degrade performance for the acceleration structure or might simply be impossible to implement (an example of this would be when using indexed triangles, where vertices can be shared by multiple triangles).

Median Sphere Set. The median sphere set can be used with or without the center hemisphere. Also storing the center hemisphere will not add much overhead. If the medians are recalculated from the vertices, the first step is calculating the center, and storing the 4th radius allows aligning the data to 16 bytes. If the hemisphere centers are stored, adding the extra hemisphere will increase the per triangle data to 64 bytes, exactly a cache line.

Polygon Sphere Set. The polygon sphere set can be stored and queried in two distinct ways. First, using the method for the center sphere set. In that case the same optimizations can be applied. Second, using an indexed lookup. Since hemispheres are used by multiple triangles, they need not be stored multiple times. This can reduce memory overhead, if we assume that we store the hemisphere centers as well as the radii. Total memory usage would be one index per triangle to fetch the hemisphere, and one hemisphere per polygon. The number of polygons will depend on the nature of the scene and how many triangles are in the same plane as their neighbours.

Random Sphere Set. To generate random locations on the surface of a triangle, we can use the triangle's index as the seed of a random number generator. The hemisphere centers are then calculated as a random interpolation of the vertices. During construction, only the radii need to be stored. When querying the sphere set, the triangle index and vertices are used to regenerate the hemisphere centers. The total memory overhead is a single scalar per hemisphere, the radius. The number of fetches for a query starts with the vertices. Each hemisphere adds a single scalar.

Further optimization can be done using SIMD instructions. For example, using SSE will allow querying the center sphere set with 4 rays at once. This has two advantages. Firstly, 4 operations are performed at once. Secondly, the latency of the fetching of the hemispheres is shared between the 4 rays. Using wider vector instructions (AVX and AVX-512) should bring even more speedups, although at diminishing returns.

4. Results

Test scenes are taken from the Computer Graphics Archive [McG17], see also Table 1 and Figure 3. All models are scaled such that their longest axis is set to exactly 20 units. For each of the scenes, several representative viewpoints are selected. Results are averaged over these viewpoints. Three ray classes are tested in isolation to account for different behaviours: ambient occlusion rays, diffuse rays and shadow rays.

Ambient occlusion rays are rendered using a range of radii set to a fixed percentage of the scene size. Each primary ray that finds an intersection is used to generate exactly one occlusion ray from a cosine-weighted distribution.

scene	triangles	memory overhead (MiB)
buddha	1.1M	16.6
bunny	144K	2.2
cathedral	75K	1.1
dragon	871K	13.3
hairball	2.9M	43.9
livingroom	581K	8.9
rungholt	5.8M	88.7
sponzacrytek	262K	4
sponzadragon	1.1M	17.3

Table 1: *Number of triangles per scene and memory overhead for the 16-byte center sphere set.*

The diffuse renders are made using a simple backward path tracing algorithm. In each scene, several triangles with an emissive material are added such that the scene is properly lit. The normal geometry has a simple diffuse material. Primary rays are casted and, when they hit a diffuse material, extended. Each path is extended until an emissive polygon is found, the ray leaves the scene, or a maximum depth is reached.

For the shadow rays a single point light is added to a central position in each scene. Each primary ray that hits an object is then extended to the point light.

Two types of statistics are collected: the number of visited nodes per ray (split by internal nodes and leaves) and the runtimes. Note that for the diffuse renders the number of nodes is also calculated per ray, not per path. Generating and tracing primary rays, writing to the image buffer or any other steps are not considered when measuring the runtimes. Only the following steps of the entire program are tracked:

1. Generating extension rays.
2. Calculating offsets.
3. Tracing the extension rays.

All experiments were run single-threaded on a 2.60GHz Intel i7-6700HQ with 8GB DDR4 RAM under Windows 10. For BVH construction and intersection tests the Intel Embree ray tracing library (version 3.5.2) was used [WWB*14]. All images were rendered at 4096x4096 pixels.

4.1. Traversals

Table 2 shows the number of visited nodes and leaves for ambient occlusion rays. The rays were given a length, or radius, of a percentage of the scene size. The radii range from 1% to 50%. The **baseline** column shows the absolute number of visited internal nodes and leaves. These numbers increase as the radius, and thus the search space, becomes bigger. When using any of the hemisphere sets to offset the rays, the average number of visited nodes and leaves decreases, as shown by each column corresponding to the respective hemisphere set.

The median hemisphere set consistently outperforms all other sets. This is only by a small margin, though, indicating that increasing the number of hemispheres per triangle has little added benefit. When comparing the results of the vertex sphere set with the center

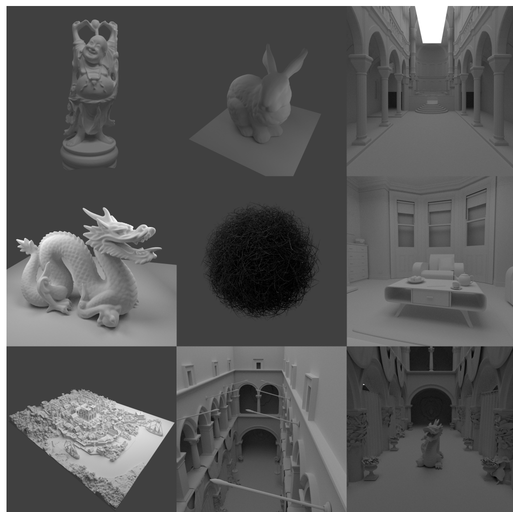

Figure 3: *All test scenes, as seen from one of the viewpoints. From left to right, top to bottom: Buddha, Bunny, Cathedral, Dragon, Hairball, Living Room, Rungholt, Sponza Crytek, Sponza and Dragon.*

sphere set, we can see that it performs better in some scenes, but worse in others. This seems related to the nature of the geometry: the architectural scenes that contain more concave geometry, such as the Cathedral, make the vertex sphere set less useful. The polygon sphere set shows no improvement over the center sphere set, suggesting that selecting the largest hemisphere is a poor optimization strategy. The random sphere set uses 4 hemispheres per triangle, the same as the median sphere set, but does not perform better. This indicates that placing the hemispheres on fixed locations gives good coverage of the entire triangle area.

Scenes such as Buddha, Bunny and Dragon show a reduction of visited leaf nodes by up to 85% for short rays. The number of visited internal nodes is more than halved. For longer rays reductions are not as large, although still at least 30%. The strong reduction for short rays is due to many rays receiving an offset that is larger than the radius, causing traversal to terminate in the root node. Figure 4 shows this for the Dragon scene: the fraction of rays for which the calculated offset (using the center sphere set) is larger than the radius is high for short radii and then drops quickly. For scenes that do not contain both large open spaces and convex geometry, this fraction drops much quicker.

The Rungholt scene contains almost exclusively axis-aligned geometry. Because of this, the majority of extension rays do not start inside the bounding box of the leaf node the primary ray ended in, and there is little improvement with offsets. Also shown in Table 2 are results for the Rungholt scene rotated by 15°. The baseline shows that the number of visited internal nodes increases by about 20%, while the number of visited leaves doubles. When using offsets, this increase is partially undone.

The Sponza Dragon scene is the Sponza Crytek scene with the dragon model added in its center. While the Sponza Crytek scene shows negligible reduction for long rays, with the dragon relative

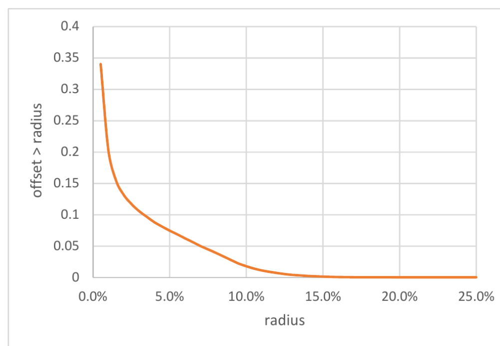

Figure 4: *Fraction of AO rays for which the offset is larger than the radius (in the Dragon scene with the center sphere set).*

improvements become larger, showing that the sphere sets adapt to varying geometry. The Sponza Dragon scene could be seen as a common use case for e.g. games, where detailed character models move through buildings.

Table 3 and Table 4 show the results for diffuse and shadow rays. The overall behaviour for all scenes and sphere sets is similar to that of the ambient occlusion rays, with some slight differences. The scenes in which the AO rays saw the largest reductions do not fare as well. Conversely, some of the worst performers for long AO rays (such as the Living Room and Sponza Crytek scenes) do show more significant reductions.

4.2. Runtimes

The traversal results have shown that the median sphere set consistently outperforms all other methods. However, this is only by a small margin. The center sphere set shows improvements within a range of several percent. Because the center sphere set uses only a quarter of the memory that is required by the median sphere set, the results in this section only cover the center sphere set. Alternatives have not been thoroughly optimized, and would probably not perform better, as calculating the offsets is mostly memory bound. The center hemispheres were stored as a simple array, retrievable using the triangle index. Each hemisphere occupies 16 bytes: 3 floats for the center, 1 float for the radius.

Figure 5 shows the ambient occlusion ray trace times. For short rays (1% of the scene size) the Dragon scene is traced using only 73% (827ms vs 1132ms) of the original time. As the ray length increases this difference becomes smaller, with 85% (1339ms vs 1576ms) at 50% of the scene size. Rungholt is traced slower by up to 3% at all ray lengths. For all but the shortest rays, the same happens for the Cathedral and Sponza Crytek scenes. In all cases, performance improvements are in accordance with the traversal improvements reported in the previous section: more saved traversal steps means shorter trace times.

The diffuse trace times are displayed in Figure 6. The results follow the same pattern as for the AO rays: a loss of 3% for Rungholt and gains up to 20% for the Dragon. The results for shadow rays, as shown in Figure 7, again follow the same pattern.

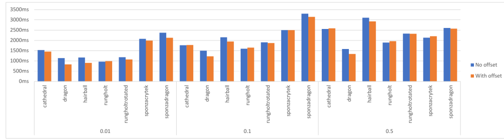

Figure 5: *Ambient occlusion ray trace times.*

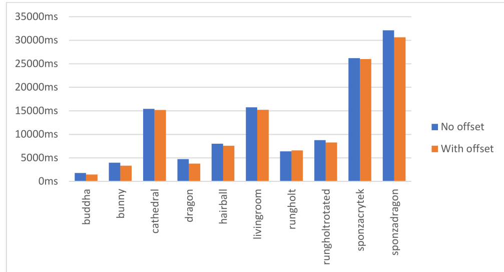

Figure 6: *Diffuse ray trace times.*

When combining the observations about visited nodes and trace times, we can draw an important conclusion about the computational overhead of offset calculation: this overhead is small. Consider the Rungholt scene, where the reduction of visited leaf nodes is 1% or less for all ray types, and trace times increase by only 3%.

5. Conclusion

We presented a simple auxiliary data structure to accelerate ray tracing. We precompute a set of hemispheres located on the surface of the scene's geometry that are guaranteed not to contain any objects. When given an extension ray, we retrieve the hemispheres that start on the same geometric primitive and offset the ray origin. With a large enough offset, the ray will be moved out of the leaf node it started in, thus saving expensive intersection tests at the bottom of the tree during traversal. Additional internal nodes may also be culled if the ray is moved out of their bounding box.

Improvements vary between scenes. In the most optimal case, the number of visited leaf nodes can be reduced by several factors. Additionally, there is a large reduction of the number of visited internal nodes. However, this is only for short occlusion rays in scenes containing large open spaces and convex geometry. On the other end of the spectrum there are scenes for which there is next to no reduction of visited nodes. Changes in actual trace time range from speedups of up to 25% to a loss of 3%. This last number indicates that the overhead of offset calculation is low.

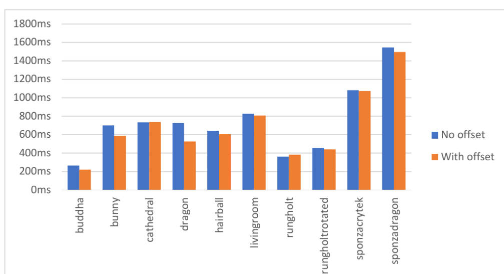

Figure 7: *Shadow ray trace times.*

The memory overhead added by the hemisphere set is linear in the scene size, requiring a fixed number of bytes per triangle. The optimized hemisphere set used in this paper uses 16 bytes per triangle. The hemisphere set is also completely independent of the acceleration structure, making it compatible with any form of traversal of the BVH or even alternative acceleration structures.

6. Future work

Only the center sphere set implementation was thoroughly optimized. While the alternatives offer little extra reduction of visited nodes at the cost of several times more memory overhead, it might be worthwhile to investigate their performance when properly optimized.

The polygon sphere set could perform better than the center sphere set with a better heuristic for selecting the optimal hemisphere. Taking into account additional properties, such as the area of overlap between a hemisphere and the triangles, might result in a higher quality hemisphere set.

Concave geometry has been shown to result in little reduction of visited nodes. This may be improved by using other methods to offset rays, such as other geometric primitives that better represent empty spaces in these areas.

A deciding factor for the viability of hemisphere sets is how they perform in a GPU renderer. Offset calculation will have a different overhead and the traversal algorithms used on the GPU tend to be different.

Efficient construction of the hemisphere sets was not a subject of this work. This will become especially relevant for dynamic scenes, where quick reconstruction of (parts of) the hemisphere set will be needed.

7. Acknowledgements

I would like to thank dr. ing. Jacco Bikker for providing the original idea of offsetting rays and his guidance during this research project. Special thanks to dr. dr. Egon van den Broek for reviewing this paper and giving useful suggestions.

References

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Communications of the ACM 18*, 9 (1975), 509–517. 2

[BH10] BOULOS S., HAINES E.: Sorted bvhs. *Ray Tracing News 23*, 2 (2010), 6. 2

[Cla76] CLARK J. H.: Hierarchical geometric models for visible surface algorithms. *Communications of the ACM 19*, 10 (1976), 547–554. 2

[CW88] CLEARY J. G., WYVILL G.: Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer 4*, 2 (1988), 65–83. 2

[FCM09] FOWLER C., COLLINS S., MANZKE M.: Accelerated entry point search algorithm for real-time ray-tracing. In *Proceedings of the 25th Spring Conference on Computer Graphics* (2009), ACM, pp. 59–66. 2

[FLPE15] FUETTERLING V., LOJEWSKI C., PFREUNDT F.-J., EBERT A.: Efficient ray tracing kernels for modern cpu architectures. *Journal of Computer Graphics Techniques (JCGT) 4*, 4 (2015).

[FTI86] FUJIMOTO A., TANAKA T., IWATA K.: Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications 6*, 4 (1986), 16–26. 1

[Gla84] GLASSNER A. S.: Space subdivision for fast ray tracing. *IEEE Computer Graphics and applications 4*, 10 (1984), 15–24. 2

[GS87] GOLDSMITH J., SALMON J.: Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications 7*, 5 (1987), 14–20. 2, 3

[Hav00] HAVRAN V.: *Heuristic ray shooting algorithms*. PhD thesis, Ph. d. thesis, Department of Computer Science and Engineering, Faculty of . . . , 2000. 2

[HB07] HAVRAN V., BITTNER J.: Ray tracing with sparse boxes. In *Proceedings of the 23rd Spring Conference on Computer Graphics* (2007), ACM, pp. 49–54. 2

[HBZ98] HAVRAN V., BITTNER J., ZÁRA J.: Ray tracing with rope trees. In *14th Spring Conference on Computer Graphics* (1998), pp. 130–140. 2

[HPMB19] HENDRICH J., POSPÍŠIL A., MEISTER D., BITTNER J.: Ray classification for accelerated bvh traversal. In *Computer Graphics Forum* (2019), vol. 38, Wiley Online Library, pp. 49–56. 3

[IH11] IZE T., HANSEN C.: Rtsah traversal order for occlusion rays. In *Computer Graphics Forum* (2011), vol. 30, Wiley Online Library, pp. 297–305. 3

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 3 (1990), 153–166. 2

[McG17] MCGUIRE M.: Computer graphics archive, July 2017. https://casual-effects.com/data. URL: https://casual-effects.com/data. 5

[ORM08] OVERBECK R., RAMAMOORTHI R., MARK W. R.: Large ray packets for real-time whitted ray tracing. In *2008 IEEE Symposium on Interactive Ray Tracing* (2008), IEEE, pp. 41–48. 2

[RSH05] RESHETOV A., SOUPIKOV A., HURLEY J.: Multi-level ray tracing algorithm. In *ACM Transactions on Graphics (TOG)* (2005), vol. 24, ACM, pp. 1176–1185. 2

[SFD09] STICH M., FRIEDRICH H., DIETRICH A.: Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), ACM, pp. 7–13. 2

[Wal07] WALD I.: On fast construction of sah-based bounding volume hierarchies. In *2007 IEEE Symposium on Interactive Ray Tracing* (2007), IEEE, pp. 33–40. 2

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive rendering with coherent ray tracing. In *Computer graphics forum* (2001), vol. 20, Wiley Online Library, pp. 153–165. 2

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: a kernel framework for efficient cpu ray tracing. *ACM Transactions on Graphics (TOG) 33*, 4 (2014), 143. 5

scene	radius	baseline	center	vertex	median	polygon	random
buddha	1%	7.71 / 0.80	46% / 19%	43% / 17%	44% / 15%	45% / 19%	44% / 16%
buddha	5%	8.45 / 0.94	61% / 34%	59% / 32%	59% / 31%	60% / 34%	59% / 31%
buddha	10%	8.73 / 1.01	63% / 39%	61% / 38%	62% / 37%	63% / 40%	62% / 37%
buddha	25%	8.90 / 1.06	64% / 43%	63% / 42%	63% / 41%	64% / 44%	63% / 41%
buddha	50%	8.90 / 1.06	65% / 44%	63% / 42%	63% / 41%	64% / 44%	63% / 42%
bunny	1%	6.50 / 0.26	47% / 40%	38% / 32%	41% / 33%	45% / 39%	42% / 34%
bunny	5%	7.10 / 0.33	63% / 54%	58% / 47%	59% / 48%	62% / 53%	60% / 49%
bunny	10%	7.40 / 0.39	66% / 61%	62% / 55%	63% / 56%	66% / 60%	64% / 57%
bunny	25%	7.55 / 0.52	71% / 72%	68% / 68%	69% / 68%	70% / 71%	69% / 69%
bunny	50%	7.36 / 0.55	73% / 75%	70% / 71%	71% / 71%	72% / 74%	71% / 72%
cathedral	1%	5.78 / 0.12	68% / 84%	79% / 93%	57% / 73%	80% / 86%	60% / 77%
cathedral	5%	6.16 / 0.18	85% / 90%	89% / 95%	81% / 83%	91% / 91%	81% / 86%
cathedral	10%	6.60 / 0.26	89% / 94%	93% / 97%	86% / 89%	92% / 94%	87% / 90%
cathedral	25%	7.51 / 0.62	93% / 98%	95% / 99%	90% / 96%	95% / 98%	91% / 97%
cathedral	50%	7.03 / 0.91	95% / 99%	97% / 99%	93% / 98%	96% / 99%	93% / 98%
dragon	1%	6.32 / 0.43	55% / 17%	49% / 15%	50% / 15%	54% / 17%	50% / 15%
dragon	5%	7.08 / 0.57	66% / 40%	62% / 39%	62% / 39%	66% / 40%	63% / 39%
dragon	10%	7.43 / 0.65	69% / 50%	66% / 49%	66% / 49%	69% / 50%	66% / 49%
dragon	25%	7.41 / 0.66	71% / 53%	67% / 52%	68% / 52%	70% / 53%	69% / 53%
dragon	50%	6.96 / 0.62	71% / 54%	69% / 53%	70% / 53%	71% / 54%	70% / 53%
hairball	1%	9.18 / 1.45	76% / 58%	71% / 64%	66% / 43%	73% / 58%	67% / 47%
hairball	5%	12.12 / 2.74	93% / 81%	93% / 83%	91% / 74%	92% / 81%	91% / 76%
hairball	10%	14.16 / 3.76	94% / 87%	94% / 89%	93% / 83%	94% / 87%	93% / 84%
hairball	25%	16.08 / 4.66	96% / 90%	95% / 92%	94% / 87%	95% / 90%	95% / 88%
hairball	50%	16.47 / 4.81	96% / 91%	96% / 92%	95% / 89%	96% / 91%	95% / 89%
livingroom	1%	5.17 / 0.10	60% / 70%	64% / 71%	49% / 64%	77% / 77%	53% / 66%
livingroom	5%	5.62 / 0.18	84% / 85%	84% / 85%	77% / 82%	88% / 88%	79% / 83%
livingroom	10%	5.92 / 0.24	90% / 89%	89% / 89%	86% / 87%	93% / 92%	87% / 88%
livingroom	25%	6.39 / 0.35	92% / 93%	92% / 93%	90% / 92%	95% / 95%	91% / 93%
livingroom	50%	5.41 / 0.49	96% / 97%	96% / 97%	94% / 96%	97% / 98%	95% / 96%
rungholt	1%	6.55 / 0.22	83% / 99%	80% / 98%	81% / 98%	82% / 99%	82% / 98%
rungholt	5%	7.59 / 0.44	93% / 100%	91% / 99%	92% / 99%	93% / 100%	92% / 99%
rungholt	10%	8.07 / 0.54	94% / 100%	92% / 99%	93% / 99%	94% / 100%	93% / 99%
rungholt	25%	8.18 / 0.59	95% / 100%	93% / 99%	94% / 99%	95% / 100%	94% / 99%
rungholt	50%	8.17 / 0.59	95% / 100%	93% / 99%	94% / 99%	95% / 100%	94% / 99%
rungholtrotated	1%	7.98 / 0.45	77% / 70%	73% / 65%	74% / 64%	78% / 82%	75% / 66%
rungholtrotated	5%	9.15 / 0.74	88% / 84%	86% / 82%	86% / 81%	90% / 91%	87% / 82%
rungholtrotated	10%	9.49 / 0.89	90% / 88%	88% / 86%	89% / 86%	92% / 93%	89% / 86%
rungholtrotated	25%	9.66 / 0.95	91% / 90%	89% / 88%	90% / 88%	92% / 94%	90% / 88%
rungholtrotated	50%	9.66 / 0.95	92% / 90%	90% / 88%	90% / 88%	93% / 94%	91% / 89%
sponzacrytek	1%	10.33 / 0.18	86% / 80%	90% / 79%	79% / 74%	90% / 82%	81% / 75%
sponzacrytek	5%	11.05 / 0.28	95% / 89%	95% / 88%	92% / 86%	96% / 90%	93% / 87%
sponzacrytek	10%	11.62 / 0.39	96% / 93%	96% / 92%	94% / 91%	96% / 93%	94% / 91%
sponzacrytek	25%	9.16 / 0.60	98% / 98%	98% / 97%	98% / 97%	99% / 98%	98% / 97%
sponzacrytek	50%	5.82 / 0.46	99% / 99%	99% / 99%	98% / 98%	99% / 99%	98% / 98%
sponzadragon	1%	10.96 / 0.26	86% / 71%	87% / 70%	82% / 67%	87% / 73%	81% / 67%
sponzadragon	5%	12.50 / 0.43	93% / 85%	93% / 84%	92% / 83%	94% / 86%	92% / 83%
sponzadragon	10%	13.66 / 0.60	95% / 91%	95% / 90%	94% / 89%	95% / 91%	94% / 90%
sponzadragon	25%	11.60 / 0.74	97% / 95%	97% / 95%	96% / 95%	97% / 96%	96% / 95%
sponzadragon	50%	7.52 / 0.62	98% / 97%	98% / 97%	97% / 96%	98% / 97%	97% / 97%

Table 2: *Ambient occlusion traversal.* ***radius****: ray length relative to scene size.* ***baseline****: #visited internal nodes / #visited leaf nodes per ray without offset.* ***sphere sets****: visited nodes / leaves relative to baseline.*

scene	baseline	center	vertex	median	polygon	random
buddha	9.93 / 2.38	68% / 32%	67% / 31%	67% / 29%	68% / 32%	67% / 30%
bunny	8.12 / 2.24	74% / 45%	70% / 42%	71% / 40%	73% / 45%	71% / 41%
cathedral	10.09 / 2.29	94% / 91%	96% / 97%	91% / 86%	96% / 93%	92% / 88%
dragon	9.49 / 2.42	74% / 48%	71% / 42%	71% / 40%	74% / 49%	71% / 40%
hairball	19.62 / 8.39	95% / 83%	95% / 87%	94% / 78%	95% / 84%	94% / 79%
livingroom	9.21 / 2.18	92% / 87%	92% / 88%	90% / 82%	95% / 92%	91% / 84%
rungholt	9.35 / 1.02	95% / 99%	93% / 97%	94% / 97%	95% / 99%	94% / 98%
rungholtrotated	11.99 / 2.16	89% / 70%	87% / 66%	88% / 65%	92% / 85%	88% / 66%
sponzacrytek	16.20 / 4.20	96% / 93%	96% / 94%	94% / 91%	97% / 94%	95% / 91%
sponzadragon	18.77 / 4.87	95% / 91%	95% / 91%	94% / 89%	95% / 92%	94% / 89%

Table 3: *Diffuse traversal.* ***baseline****: #visited internal nodes / #visited leaf nodes per ray without offset.* ***sphere sets****: visited nodes / leaves relative to baseline.*

scene	baseline	center	vertex	median	polygon	random
buddha	10.18 / 0.40	72% / 47%	70% / 44%	71% / 44%	72% / 47%	71% / 44%
bunny	10.42 / 0.15	81% / 64%	78% / 57%	79% / 58%	80% / 64%	79% / 60%
cathedral	7.27 / 0.05	93% / 89%	94% / 94%	90% / 80%	96% / 90%	91% / 83%
dragon	8.15 / 0.18	71% / 34%	69% / 32%	69% / 32%	70% / 34%	70% / 32%
hairball	16.66 / 1.43	97% / 90%	97% / 91%	96% / 85%	96% / 89%	96% / 87%
livingroom	6.38 / 0.05	91% / 77%	90% / 76%	88% / 70%	93% / 83%	89% / 72%
rungholt	6.96 / 0.08	93% / 99%	91% / 97%	92% / 98%	92% / 99%	92% / 98%
rungholtrotated	8.78 / 0.20	88% / 79%	86% / 72%	87% / 73%	87% / 82%	87% / 75%
sponzacrytek	15.61 / 0.09	97% / 86%	96% / 83%	95% / 81%	97% / 87%	96% / 82%
sponzadragon	17.71 / 0.14	96% / 78%	95% / 76%	95% / 74%	96% / 79%	95% / 75%

Table 4: *Shadow traversal.* ***baseline****: #visited internal nodes / #visited leaf nodes per ray without offset.* ***sphere sets****: visited nodes / leaves relative to baseline.*