Performance Evaluation of Acceleration Structures for Cone-tracing Traversal

Roman Wiche David Kuri

Virtual Engineering Lab, Volkswagen Group

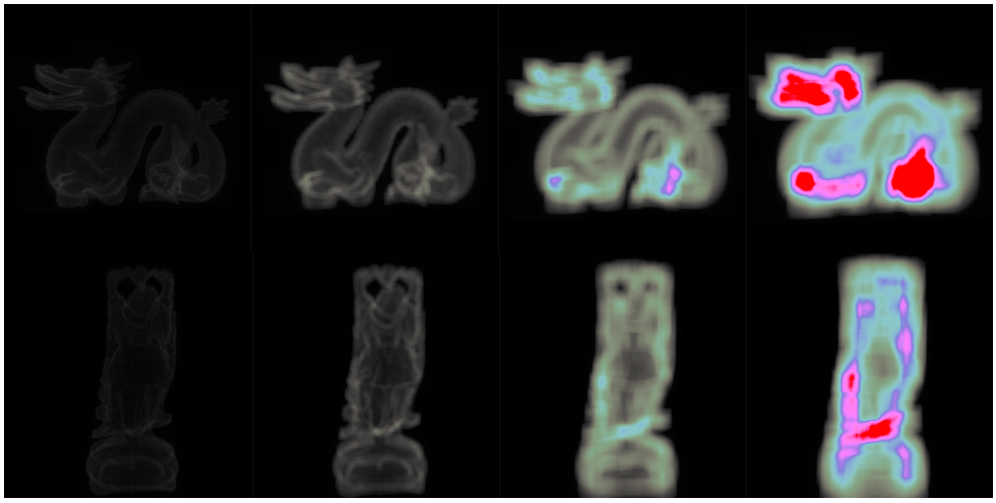


Figure 1. STANFORD DRAGON and HAPPY BUDDHA heatmaps rendered by tracing primary cones through a bounding volume hierarchy with varying cone aperture α. From left to right: 0.04°, 0.5°, 1.5°, 2.5°. Heat color corresponds to number of traversal steps and encountered triangles.

Abstract

This paper focuses on the technical question of how to apply acceleration structures used for polygonal scenes from ray tracing to cone tracing. We examine cone-traversal performance for k-d trees and bounding volume hierarchies. Our results demonstrate which accelerator to prefer for cone tracing given corresponding apertures and provide an estimation when cones of varying sizes could replace a specified number of ray samples with the same traversal performance but without subsampling.

 ISSN 2331-7418

1. Introduction

With the advent of NVIDIA RTX and DirectX Raytracing, ray tracing achieves practical relevance for demanding real-time applications. Its success lies in synthesizing physically-based, photorealistic renderings. Overall, ray tracing and its underlying operations like ray-scene intersections, shading points, and tracing secondary rays are well-understood principles.

One major challenge is noise in path tracing for interactive scenes. The noise originates from undersampling when integrating over areas or higher-dimensional spaces using a Monte-Carlo approach with infinitesimal rays. Solutions for this range from tracing more rays (e.g., through acceleration structures [Stich et al. 2009]) to denoising machine-learning algorithms [Chaitanya et al. 2017].

Another noteworthy approach uses higher-order primitives, also called generalized rays, for tracing. One example is cone tracing introduced by Amanatides [1984]. Although offering advantages like implicitly antialiased images, inherent soft shadows, and glossy reflections, cone-tracing research is sparse and poses many challenges. Transforming a polygonal ray-tracing pipeline to cone tracing involves a complete paradigm shift with many yet unanswered questions of how a polygonal, purely cone-traced rendering would work. In contrast to rays, there is no consensus about how to shade multiple triangles inside a cone, how to trace secondary cones, or what the result of a cone-scene intersection in the scope of rendering should look like.

Given all these open questions, we focus on how to return a cone-scene intersection acquired by tracing cones through acceleration structures. We examine k-d trees and bounding volume hierarchies (BVHs), as these are the most common accelerators for rays. This may also prove valuable for applications outside of rendering (e.g., in the fields of physics or AI).

Our contributions are:

- a common methodology for cone acceleration-structure traversal;
- a discussion on k-d tree traversal methods for cones;
- a cone-frustum-axis-aligned bounding box (AABB) intersection routine for BVH traversal;
- an evaluation of ray vs. cone, and cone k-d tree vs. cone BVH traversal.

2. Related Work

Amanatides [1984] created the first cone-tracing renderings including antialiasing, fuzzy shadows, and glossy reflections, without any acceleration structure. Crassin et al. [2011] presented voxel cone tracing for real-time global illumination. They

avoid complex polygonal meshes by using a sparse voxel octree. Hence, they do not intersect cones with triangle meshes, but look up prefiltered values while stepping along cone axes. Qin et al. [2014] published the most recent work on cone tracing for fur rendering. They employ a BVH but omit how they traverse it or compute cone-AABB intersections in detail.

Besides cone tracing, there are a few other generalized ray approaches like hypercubes [Arvo and Kirk 1987], pencils [Shinya et al. 1987], beams [Overbeck et al. 2007], ray-bounds [Ohta and Maekawa 1990], or frusta [Teller and Alex 1998].

For recent and significant acceleration-structure developments, refer to Stich et al. [2009], Karras et al. [2013], and Vinkler et al. [2016].

3. Acceleration Structures for Cone Tracing

An acceleration structure facilitates the tracing of a ray through a polygonal scene by providing only triangles that can potentially be hit and need to be tested. Furthermore, an accelerator allows the process to terminate when no more triangles closer than the current closest hit are present.

In contrast, the result of a cone traversal through an acceleration structure is more ambiguous. We define it as

$$I = ((i_1, t_1), (i_2, t_2), ..., (i_n, t_n)), \tag{1}$$

where I is a depth-sorted list of user-defined length n with no duplicates and n pairs containing a triangle ID i and parametric distance t from a cone origin to a representative position on the triangle projected onto the cone-direction axis.

We obtain I usng the following routine: As illustrated in Figure 2, we input a cone with origin S, direction $\hat{v}$, and aperture angle α (e.g., by casting a primary cone

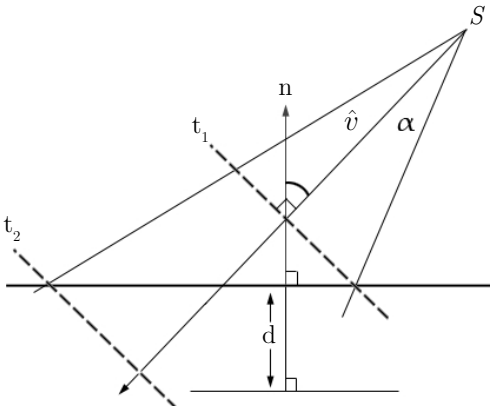


Figure 2. Side view of a cone-split-plane intersection proposed by Tsakok [2008].

covering one pixel). Then, this cone traverses through an accelerator until n pairs with the nearest t of all triangles in the accelerator are collected. We insert pairs into I when accessing a leaf by insertion sorting, whereas t can be obtained after a positive cone-triangle intersection (e.g., [Eberly 2008; Qin et al. 2014]).

The routine may be provided with an optional $t_{\text{near}-\text{clipping}}$ and $t_{\text{far}-\text{clipping}}$ for more efficient traversals. In addition, this allows the routine to be called consecutively for the same cone if n triangles from a preceding traversal are not sufficient for shading by setting $t_{\text{near}-\text{clipping}} = t_n$. When I is filled with n pairs, further nodes can be culled if t_n is smaller.

Standard Stack-based k-d Tree Traversal

Tsakok [2008] described a standard stack-based k-d tree traversal with conic packets without applying it to cone tracing. He derived the two intersection distances along a cone direction where the outer cone edges intersect a split plane as

$$t_{1,2} = \frac{d - \hat{n} \cdot S}{\hat{n} \cdot \hat{v} \pm \tan\alpha\sqrt{1 - (\hat{n} \cdot \hat{v})^2}}, \tag{2}$$

where d is the split-plane position, $\hat{n}$ the normal of the split plane, S the cone origin, $\hat{v}$ the cone direction, and α the aperture angle. Figure 2 illustrates this intersection. Then, t_1 and t_2 are integrated in a standard stack-based k-d tree traversal for rays [Pharr et al. 2016]. The main difference is how to select which child node to traverse next. Figure 3 illustrates how this works for cones, and Listing 1 contains corresponding pseudocode.

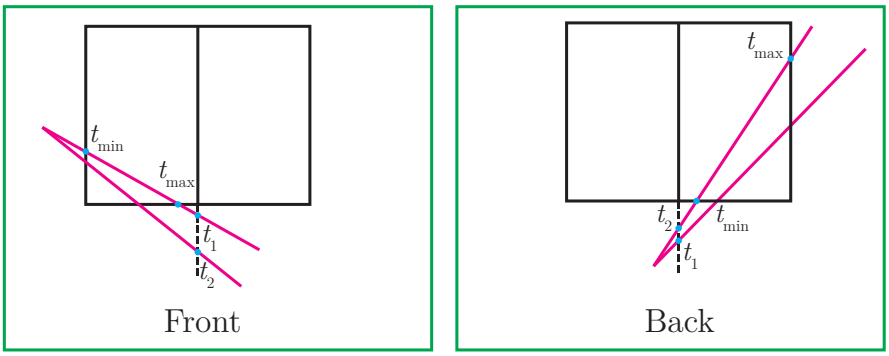


Figure 3. Node selection for the standard stack-based k-d tree traversal with a cone. If $t_1 > t_{\text{max}}$, only the front node is visited. If $t_2 < t_{\text{min}}$, only the back node is visited. Otherwise, the front node is visited next and the back node is pushed on a stack.

```
Prerequisite: We hit k-d tree box with cone in range [t_min, t_max]

While got node to process:
  If InteriorNode:
    Compute t_min-plane, t_max-plane according to Figure 2 and Equation (2)
    t_min-plane = +∞ If < 0
    t_max-plane = +∞ If < 0
    t_1 = min(t_min-plane, t_max-plane), t_2 = max(t_min-plane, t_max-plane)

    If t_2 < t_min:
      nextNode = backNode
    Elif t_1 > t_max:
      nextNode = frontNode
    Else:
      If t_max >= t_near-clipping and max(t_1, t_min) <= t_n:
        push (backNode, max(t_1, t_min), t_max) to stack
      nextNode = frontNode
      t_max = min(t_max, t_2)
  Else:
    Intersect cone with triangles in Leaf
    Pop nodes from stack for processing until one is found which
    satisfies t_max-pop >= t_near-clipping and t_min-pop <= t_n. Then,
    t_min = t_min-pop, t_max = t_max-pop
```

Listing 1. Standard stack-based k-d tree traversal for cones. For a HLSL implementation, see [Wiche 2019].

Discussion of Advanced k-d Tree Traversals

The following considerations for advanced cone-traversal techniques beyond the standard lean on Santos et al. [2012].

Ropes++

Ropes++ searches for a current leaf by subsequently going down a tree with a ray-k-d tree-box intersection position. Then, it tests a ray with all triangles in this leaf. Finally, it computes which face a ray is leaving and follows a rope to a next node.

The first step could be executed with a cone by computing the min and max intersection positions. By hitting multiple nodes, a list of entering leaves would need to be maintained. However, the most inherent problem arises when deciding which rope to follow (see Figure 4). A cone might exit a leaf on five faces in the worst case. When pursuing each rope, this creates an unmanageable, exponentially growing and fanned-out node tree. Culling nodes by saving t_{min} generally fails due to a cone fanning-out and being at different distances across the cone.

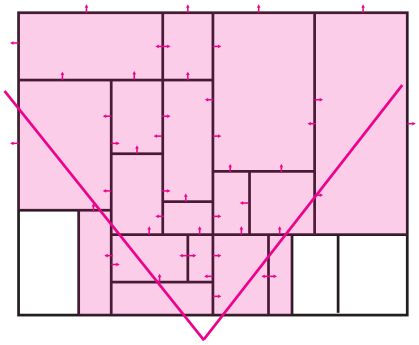


Figure 4. Complexity of fanned-out cone and choice of rope for ropes++ traversal.

8-byte-standard traversal

The 8-byte-standard traversal reduces the size of each stack element from 12 to eight bytes by omitting $t_{\min}$ as it equals $t_{\max}$ of the previous node.

This approach cannot be used for cones because $t_{\max}$ of the previous node differs from $t_{\min}$ of the next node, since a cone has two t-values for one plane, as demonstrated in Figure 3.

Short-stack pushdown hybrid traversal

The short-stack pushdown hybrid traversal saves a smaller stack and falls back to a pushdown traversal when exceeding the stack. A pushdown fallback performs better than a complete k-d-restart by conservatively pushing down restart nodes. This is always executed when just the front or back node is hit. A ray can skip reoccurring nodes by setting $t_{\min}$ further, to the end of a last visited node.

This skipping logic fails similarly to the 8-byte traversal. If $t_{\min}$ is set to the further value, next relevant nodes between both t are falsely skipped. If $t_{\min}$ is assigned conservatively with the closer value, an infinite loop occurs, as the algorithm assumes that the previously considered node was not processed yet.

Sequential traversal

The sequential traversal is similar to ropes++ but cannot use ropes referencing a next node. Instead, it searches for each following node from the top. Being a predecessor of ropes++, comparable problems arise when applied to cones.

Pushdown, k-d-restart, k-d-backtrack

Other algorithms like complete pushdown, complete k-d-restart, or k-d-backtrack are likely to perform worse for cones because they have already been shown to run slower for rays than the above-considered algorithms. The lack of performance of these algorithms lies in their logic to consider next-possible nodes, independent of the concept of a ray or cone.

BVH Traversal

Since BVHs are commonly built from AABBs, we must provide a cone-AABB intersection routine. Implementations may demand a range from a cone origin to AABBs to consider closer nodes first or to cull. For our cone-scene intersection routine in Section 3, a returned range is essential for culling nodes.

There is no well-known algorithm satisfying these requirements. Although there is a cone versus oriented bounding-box intersection algorithm [Eberly 2015], it is computationally expensive and requires a lookup table. Reshetov et al. [2005] described a k-d tree traversal with frusta including an intersection test with leaf AABBs. However, applying this to a frustum around cones would require to compute or save more information for a cone, as planes and corner rays of a frustum are used in their algorithm. Both algorithms return a boolean instead of a range.

Novel Cone-frustum-AABB intersection

We reduce cone-AABB to cone-frustum-AABB intersections to optimize the traversal. A cone-frustum (i.e., square pyramid) implicitly built from α serves as a tight bounding volume around a cone. Our proposed algorithm achieves an exact result of a cone-frustum hitting an AABB and the resulting $t_{\min}$ and $t_{\max}$ provide a good range estimation.

We extend the ray-slabs-AABB test [Shirley 2018] with t_1 and t_2 from Tsakok's cone-frustum-plane test [Tsakok 2008] (see Figure 2). For each axis, the new algorithm computes four t-values, t_1 and t_2 for the min and max AABB plane, respectively. However, the four t-values cannot be easily used to find $t_{\min-\text{axis}}$ and $t_{\max-\text{axis}}$ by taking the overall min and max. Instead, we categorize cases where Equation (2) assigns t negative values. Tsakok indicates a cone-edge-plane miss by setting negative t to $+\infty$, but this is invalid for chaining cone-frustum-plane tests in our AABB test because $+\infty$ or negative values falsify $t_{\min-\text{axis}}$ and $t_{\max-\text{axis}}$ and thus $t_{\min}$ and $t_{\max}$. For correct $t_{\min-\text{axis}}$ and $t_{\max-\text{axis}}$, the algorithm distinguishes between cases depicted in Figure 5. Applying the new $t_{\min-\text{axis}}$ and $t_{\max-\text{axis}}$ to a ray-AABB test scope results in a cone-frustum-AABB test. Listing 2 provides pseudocode.

Traversal Variants

In contrast to a ray, a cone can hit multiple nodes at one particular distance. Consequently, cone traversals raise many sophisticated challenges absent for rays. Neither a depth-first-search with a stack nor a breadth-first-search with a queue always performs an ideal traversal.

We consider two forms of traversal for cone tracing. See Figure 6. In the end, they return the same depth-sorted triangle list described in Section 3 but differ in the traversal order of accelerator nodes.

```
t_min = 0, t_max = +∞
for each axis:
  Compute float4 t4 with two cone-plane-intersections with
  aabbMin[axis], aabbMax[axis] according to Figure 2 and
      Equation (2)
  Compute t_min-axis, t_max-axis from cases in Figure 5:
    If S between planes:
      If Case a.1: t_min-axis = 0, t_max-axis = +∞
      Else: t_min-axis = 0, t_max-axis = max(t4)
    Else:
      If Case b.1: discard
      Elif Case b.2: t_min-axis = min(t4), t_max-axis = max(t4)
      Else: t_min-axis = min(positivesOf(t4)), t_max-axis = +∞

  t_min = max(t_min, t_min-axis)
  t_max = min(t_max, t_max-axis)
  if t_min > t_max: discard

return true with range [t_min, t_max]
```

Listing 2. Cone-frustum-AABB intersection routine. For a basic and improved implementation in HLSL refer to Wiche [2019].

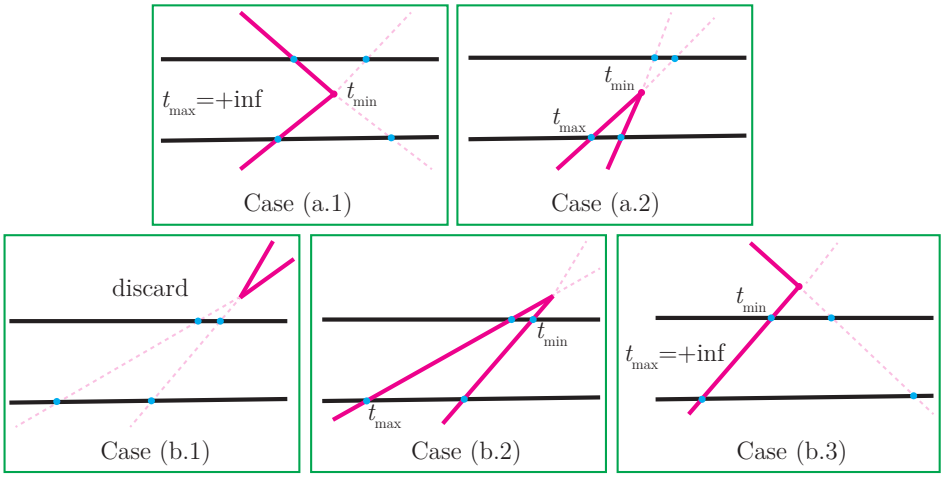


Figure 5. Cone-frustum-plane test cases for each axis. Case a: Cone origin is in between AABB planes. Case (a.1): One cone edge hits one plane, the other edge hits the other plane. Neither $t_{\text{min}-\text{axis}}$ nor $t_{\text{max}-\text{axis}}$ are updated. Case (a.2): Both cone edges hit only one plane. Only $t_{\text{max}-\text{axis}}$ is updated. Case b: Cone origin is outside of AABB planes. Case (b.1): Cone does not hit any plane. The cone-frustum never hits the box. Case (b.2): Both cone edges hit both planes. All t have to be considered for $t_{\text{min}-\text{axis}}$ and $t_{\text{max}-\text{axis}}$. Case (b.3): One cone edge hits both planes, the other edge hits none. Only the closer positive t is considered for $t_{\text{min}-\text{axis}}$.

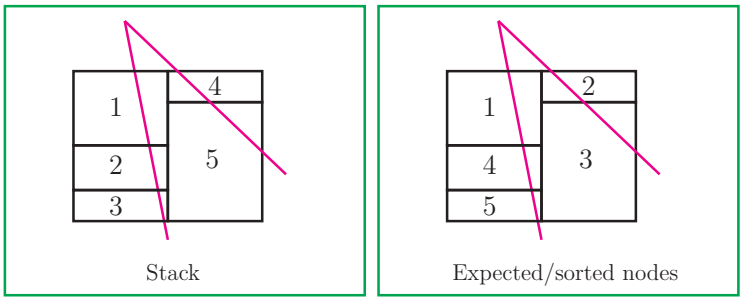


Figure 6. Cone traversal types. Stack: Depth-first, goes deep into closer node first. Sorted nodes: Receiving nodes as expected.

Node-sorted traversal

A node-sorted list can be created by insertion sorting nodes while traversing. This may provide better early outs because it is guaranteed to traverse the nodes in the best order. For example, if t_n is closer than a next node in a sorted list, the algorithm can return because there is no nearer node and thus no nearer triangle due to a sorted traversal. However, maintaining this list creates overhead, as a correct order needs to be ensured in each traversal step.

Stack traversal

This variant only traverses roughly in correct order and is not guaranteed to be precise regarding the node sequence. Early outs are harder to execute and more nodes on a stack are considered. Nevertheless, a stack is easier to maintain than a sorted list.

4. Results

Experiment Setup

We built a modular ray- and cone-tracing framework where accelerators are created offline on the CPU and traversal algorithms are executed on the GPU via DirectX Compute Shaders.

Our goal is to compare the traversal performance only, without focusing on the performance of ray- or cone-triangle intersections. Therefore, we set $n = \infty$ (Section 3) to traverse a complete accelerator and sum the number of encountered triangles in leaves without computing their intersections. The traversal result is visualized as a heatmap that roughly depicts the total computational complexity of node traversal and primitive intersection per pixel (see Figure 1).

We compare a standard stack-based k-d tree traversal of rays with a stack-based k-d tree cone traversal. For the BVH traversal, we compare the ray-AABB algorithm

from Kensler [2018] with our novel cone-frustum-AABB algorithm. We perform the stack-based cone-traversal variant.

A similar k-d tree as described by Pharr et al. [2016] and the hybrid SBVH by Stich et al. [2009] were implemented. The number of bin planes was set to 32, and the properties $\alpha = 10^{-5}$ and reference unsplitting were chosen [Stich et al. 2009]. The termination criterion to stop dividing the nodes was selected according to the number of children ≤ 8 triangles. A surface-area heuristic stopped the dividing process automatically when the cost for a split was higher than for an existing node. The resulting acceleration structures are of similar complexity.

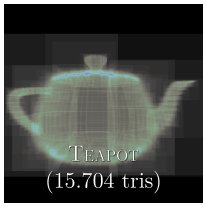
The experiments were executed on an Intel Core i7-6700K 4x4.0GHz with NVIDIA GeForce GTX 1080. The thread group size was set to $16 \times 8 \times 1$, and the resolution was assigned to 1024^2 pixels. We launch one thread on the GPU for each pixel. Single meshes of varying complexities and more holistic scenes with larger spaces were tested. Three viewpoints encompassing different aspects of the test scenes were selected and averaged.

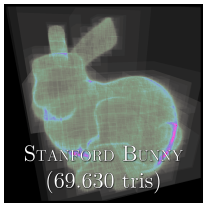
Results Discussion

The results of our experiments are depicted in Table 1, Table 2, and Figure 7. We compare eight test meshes with a BVH and k-d tree for a different number of ray samples and cone apertures α. The measured time refers to the total rendering time whereas the number of nodes and triangle encounters is averaged per pixel.

The selection of an accelerator for cone tracing seems to be highly dependent on the cone aperture α. For primary cones with $0.04°$ (fully covering one pixel at 1024^2 pixel resolution), the k-d tree traversal time outperformed the BVH for each test scene, up to a factor of 5. However, already for an aperture of $0.5°$, there is no clear winner. Depending on the test scene, the BVH starts to outperform the k-d tree from apertures $1.5°$ to $5°$. For the k-d tree, there seems to be a threshold depending on the test scene, where the time increases drastically from one measurement to the next one whereas the BVH time increases more smoothly. The average number of nodes traversed per millisecond for the k-d tree drops by 67% whereas the same number for the BVH just decreases by 34% (from $0.04°$ to $1.5°$). We observe that the summed number of visited triangles in the k-d tree is higher than in the BVH. This would have a significant impact when also computing ray- and cone-triangle intersections.

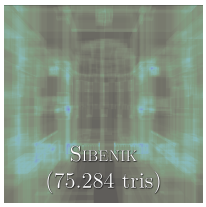
Regarding a comparison of rays and cones, neither of them are clearly superior. For some meshes, a primary cone for the k-d tree requires the same computation time as approximately eight rays. However, for other test scenes, a primary cone is as computationally expensive as 16 to 64 rays. Moderately sized cones for the BVH could potentially outperform rays, although it is hard to estimate the full potential as it is unknown how many ray samples would be required to sample a volume of a cone with a given aperture without subsampling in these cases. Still, the tables allow us to

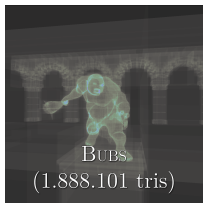




			TEAPOT			BUNNY			DRAGON			HAPPY BUDDHA		
			Avg. Millis.	Avg. Nodes	Avg. Inter.	Avg. Millis.	Avg. Nodes	Avg. Inter.	Avg. Millis.	Avg. Nodes	Avg. Inter.	Avg. Millis.	Avg. Nodes	Avg. Inter.
BVH	Rays	4	4.7	62.9	29.1	5.2	79.8	22.3	6.3	76.8	34.4	6.2	70	32
		16	14.3	251.9	116.8	17.2	319.9	90.1	21	308	138	19	282	133
		64	48.7	1007.8	467.3	60.3	1279	361	74	1235	553	69	1129	532
	Cone α	0.04°	18	18	10	34	24.6	10	94	39.6	53	116	50.6	82
		0.5°	35	30	27	93	57.3	52	421	197	449	537	262	627
		1.5°	93	71	88	337	198	256	2918	1010	2652	3513	1341	3570
		2.5°	178	132	184	734	432	613	5496	2473	6699	6926	3266	8915
		5°	497	370	573	3338	1426	2176	-	-	-	-	-	-
k-d tree	Rays	4	4.7	132	48	9.5	183	48	34.6	184	90	41.4	163	36.6
		16	14	532	196	21.4	736	195	79	739	249	98.6	654	148.6
		64	48	2129	785	67.6	2946	783	164.5	2960	998	210	2621	596
	Cone α	0.04°	7	35	14	14	50	16	32	60.6	28	35	58.6	28
		0.5°	47	54	64	61	136	126	492	516	737	760	751	1028
		1.5°	275	170	259	365	534	727	4180	3387	5633	6615	5196	7964
		2.5°	725	369	596	1611	1222	1825	-	-	-	-	-	-
		5°	1567	1369	2411	4685	4212	6744	-	-	-	-	-	-

Table 1. TEAPOT, BUNNY, DRAGON, and HAPPY BUDDHA comparison.





			SIBENIK			CRYTEK SPONZA			CONFERENCE			BUBS		
			Avg. Millis.	Avg. Nodes	Avg. Inter.	Avg. Millis.	Avg. Nodes	Avg. Inter.	Avg. Millis.	Avg. Nodes	Avg. Inter.	Avg. Millis.	Avg. Nodes	Avg. Inter.
BVH	Rays	4	8.3	184	104.6	14.2	360	244	5.7	115	80	7.6	121	52
		16	29.2	740	419.6	57	1443	980.3	19.3	464	322	25.6	485	209
		64	107.5	2963	1679	211	4411	3923	70	1857	1289.6	93.5	1943	836
	Cone α	0.04°	20.6	46.6	28	68.3	93	67	43	30.3	22	93	41	37
		0.5°	32	60	54	135	133.3	163	250	52.1	76	2142	264	639
		1.5°	66	102	152	394	301	612	952	170.7	395	5368	1681	4658
		2.5°	114	161	301	1154	578	1392	1130	387	989	-	-	-
		5°	315	388	900	3669	1745	4779	3039	1357	2314	-	-	-
k-d tree	Rays	4	6.6	304	176	15.4	484	299.6	4.8	191	122	30.9	305.3	135
		16	22	1219	707	41	1939	1201	15.3	765	490	75.6	1222	540
		64	78	3515	2833	137.6	6395	3440.6	53	3062	1960	177	3522	2163
	Cone α	0.04°	7.6	77	46	25	128.6	85	8	47	32	45.6	95	59
		0.5°	18	111	103	96	282	346	51	84.7	147	2596	1007	1718
		1.5°	56	238	344	1023	1061	1831	156	259	870	9121	6808	12834
		2.5°	121	435	745	2863	2450	3682	322	566	2241	-	-	-
		5°	446	1234	2436	-	-	-	2298	1894	4373	-	-	-

Table 2. SIBENIK, CRYTEK SPONZA, CONFERENCE, and BUBS comparison.

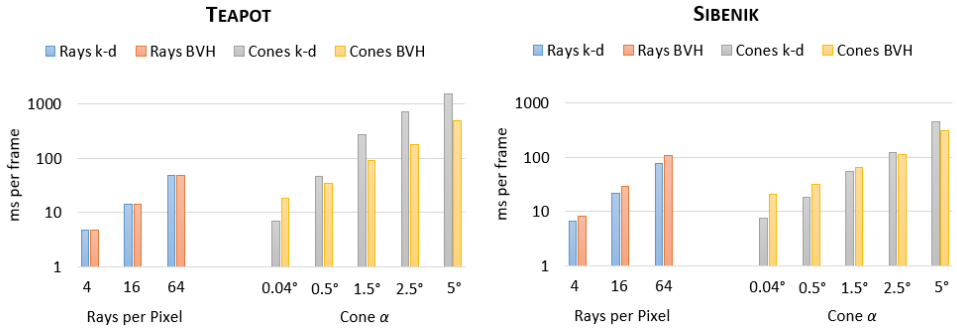


Figure 7. TEAPOT and SIBENIK data from Table 1 and Table 2.

estimate when to replace a specified number of ray samples with a cone of a certain size with a comparable traversal speed.

One factor which will significantly influence the obtained results in a real-world rendering scenario are ray- and cone-triangle intersection tests. In many cases, the rays and cones would not have traversed the acceleration structures as deeply as in the conducted tests. With ray- and cone-triangle intersections, they would have detected positive intersections earlier and aborted the traversal because they would have found a closest hit triangle or filled a depth-sorted list already. Further research is required to assess a more holistic comparison.

A complete traversal is especially a problem for cones since cones fan out and start to hit more nodes. This can lead to the accelerator-in-a-cone problem illustrated in Figure 8. In a ray tracing context, this result is counter-intuitive at first, because the chance of a ray hitting many nodes decreases with its distance. However, in the worst-case, a complete accelerator might lie in one cone, transforming into a decelerator. There needs to be a fallback solution if a cone contains many nodes. One solution might be to integrate a heuristic automatically selecting few representative triangles in nodes. Furthermore, a similar approach to voxel cone tracing [Crassin et al. 2011] could be employed when a cone encompasses too many triangles or nodes.

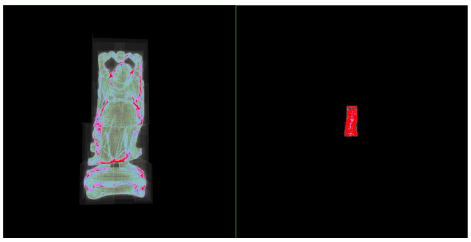


Figure 8. Accelerator-in-a-cone problem: Left and right image took the same rendering time. Although only few cones hit the accelerator in the right image, they are forced to traverse significantly more nodes than compared to the left side.

5. Conclusion

In this article, we focused on the cone traversal for k-d trees and BVHs. While the k-d tree performed better with small cones, the BVH usually outperformed the k-d tree quickly for growing cone apertures. The results estimate when a cone could replace multiple rays with comparable traversal performance but without subsampling. Besides rendering, these findings might be of interest for cone casting in physics, AI, or other computer graphics applications.

However, there are still many more open research questions to solve until cone tracing for polygonal scenes can evolve into a practical rendering technique. Operations like cone-triangle intersections and shading multiple triangles inside a cone will have a significant impact on the comparison between both tracing techniques.

With this article, we hope to spark interest in cone-tracing algorithms and inspire more contributions in this field.

Acknowledgments

We would like to thank the Stanford Computer Graphics Laboratory for the HAPPY BUDDHA, the STANFORD DRAGON, and the STANFORD BUNNY; Martin Newell for the NEWELL- or UTAH TEAPOT; Frank Meinl for the CRYTEK SPONZA; Anat Grynberg and Greg Ward for the CONFERENCE ROOM; Marko Dabrovic for the original SPONZA and the SIBENIK CATHEDRAL; Ryan Vance for the BUBS; Morgan McGuire for providing most of these test meshes [McGuire 2011]; Marc Olano and Patrick Cozzi for the support during the publication process; and Eric Haines, Tomas Akenine-Möller, and the anonymous reviewers for their supportive comments and reviews.

References

AMANATIDES, J. 1984. Ray Tracing with Cones. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, SIGGRAPH '84, 129–135. URL: http://doi.acm.org/10.1145/800031.808589. 2

ARVO, J., AND KIRK, D. 1987. Fast Ray Tracing by Ray Classification. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, SIGGRAPH '87, 55–64. URL: http://doi.acm.org/10.1145/37401.37409. 3

CHAITANYA, C. R. A., KAPLANYAN, A. S., SCHIED, C., SALVI, M., LEFOHN, A., NOWROUZEZAHRAI, D., AND AILA, T. 2017. Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder. *ACM Trans. Graph. 36*, 4 (July), 98:1–98:12. URL: http://doi.acm.org/10.1145/3072959.3073601. 2

CRASSIN, C., NEYRET, F., SAINZ, M., GREEN, S., AND EISEMANN, E. 2011. Interactive Indirect Illumination Using Voxel Cone Tracing: A Preview. In *Symposium on Interactive 3D Graphics and Games*, ACM, New York, NY, USA, I3D '11, 207–207. URL: http://doi.acm.org/10.1145/1944745.1944787. 2, 12

EBERLY, D. 2008. Intersection of a Triangle and a Cone. Tech. rep., Geometric Tools, LLC. URL: https://www.geometrictools.com/Documentation/IntersectionTriangleCone.pdf. 4

EBERLY, D., 2015. Intersection of an Oriented Box and a Cone. https://www.geometrictools.com/Documentation/IntersectionBoxCone.pdf. [Accessed 2017-03-23]. 7

KARRAS, T., AND AILA, T. 2013. Fast Parallel Construction of High-quality Bounding Volume Hierarchies. In *Proceedings of the 5th High-Performance Graphics Conference*, ACM, HPG '13, 89–99. URL: http://doi.acm.org/10.1145/2492045.2492055. 3

MCGUIRE, M., 2011. Computer Graphics Archive, August. URL: http://graphics.cs.williams.edu/data. 13

OHTA, M., AND MAEKAWA, M. 1990. Ray-Bound Tracing for Perfect and Efficient Anti-Aliasing. *The Visual Computer 6*, 3, 125–133. URL: http://dx.doi.org/10.1007/BF01911004. 3

OVERBECK, R., RAMAMOORTHI, R., AND MARK, W. R. 2007. A Real-time Beam Tracer with Application to Exact Soft Shadows. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, Eurographics Association, EGSR'07, 85–98. URL: http://dx.doi.org/10.2312/EGWR/EGSR07/085-098. 3

PHARR, M., WENZEL, J., AND HUMPHREYS, G. 2016. *Physically Based Rendering, Third Edition: From Theory To Implementation*, 3rd ed. Morgan Kaufmann Publishers Inc. URL: http://pbr-book.org/. 4, 10

QIN, H., CHAI, M., HOU, Q., REN, Z., AND ZHOU, K. 2014. Cone Tracing for Furry Object Rendering. *IEEE Transactions on Visualization and Computer Graphics 20*, 8 (Aug.), 1178–1188. URL: http://dx.doi.org/10.1109/TVCG.2013.270. 3, 4

RESHETOV, A., SOUPIKOV, A., AND HURLEY, J. 2005. Multi-level Ray Tracing Algorithm. In *ACM SIGGRAPH 2005 Papers*, ACM, SIGGRAPH '05, 1176–1185. URL: http://doi.acm.org/10.1145/1186822.1073329. 7

SANTOS, A., TEIXEIRA, J. M., FARIAS, T., TEICHRIEB, V., AND KELNER, J. 2012. Understanding the Efficiency of kD-tree Ray-Traversal Techniques over a GPGPU Architecture. *International Journal of Parallel Programming 40*, 3, 331–352. URL: http://dx.doi.org/10.1007/s10766-011-0186-1. 5

SHINYA, M., TAKAHASHI, T., AND NAITO, S. 1987. Principles and Applications of Pencil Tracing. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, ACM, SIGGRAPH '87, 45–54. URL: http://doi.acm.org/10.1145/37401.37408. 3

SHIRLEY, P., 2018. New Simple Ray-Box Test from Andrew Kensler. http://psgraphics.blogspot.com/2016/02/new-simple-ray-box-test-from-andrew.html. [Accessed 2018-10-13]. 7, 10

STICH, M., FRIEDRICH, H., AND DIETRICH, A. 2009. Spatial Splits in Bounding Volume Hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, ACM, HPG '09, 7–13. URL: http://doi.acm.org/10.1145/1572769.1572771. 2, 3, 10

TELLER, S., AND ALEX, J. 1998. Frustum Casting for Progressive, Interactive Rendering. Tech. rep., Massachusetts Institute of Technology. URL: https://dl.acm.org/citation.cfm?id=888662. 3

TSAKOK, J. A. 2008. *Fast Ray Tracing Techniques*. Master's thesis, University of Waterloo. URL: https://uwspace.uwaterloo.ca/bitstream/handle/10012/3947/thesis.pdf. 3, 4, 7

VINKLER, M., HAVRAN, V., AND BITTNER, J. 2016. Performance Comparison of Bounding Volume Hierarchies and Kd-Trees for GPU Ray Tracing. *Computer Graphics Forum 35*, 8, 68–79. URL: http://dx.doi.org/10.1111/cgf.12776. 3

WICHE, R., 2019. Supplemental Materials for Performance Evaluation of Acceleration Structures for Cone Tracing Traversal. Implementations in HLSL. https://github.com/bromanz/Perf-Eval-Accel-Cone-Tracing. [Accessed 2019-08-27]. 5, 8, 15

Index of Supplemental Materials

HLSL reference code for a standard stack-based k-d tree traversal, and a basic and improved cone-frustum-AABB intersection implementation is provided by Wiche [2019] and http://jcgt.org/published/0009/01/01/Perf-Eval-Accel-Cone-Tracing.zip.
.

Author Contact Information

Roman Wiche
Virtual Engineering Lab, Volkswagen Group
Berliner Ring 2
38440 Wolfsburg, Germany
hello@roman-wiche.com

David Kuri
Virtual Engineering Lab, Volkswagen Group
Berliner Ring 2
38440 Wolfsburg, Germany
info@davidkuri.de

R. Wiche, D. Kuri, Performance Evaluation of Acceleration Structures for Cone-tracing Traversal, *Journal of Computer Graphics Techniques (JCGT)*, vol. 9, no. 1, 1–17, 2020
http://jcgt.org/published/0009/01/01/

Received:	2019-01-17		
Recommended:	2019-08-19	Corresponding Editor:	Patrick Cozzi
Published:	2020-01-13	Editor-in-Chief:	Marc Olano

© 2020 R. Wiche, D. Kuri (the Authors).
The Authors provide this document (the Work) under the Creative Commons CC BY-ND

3.0 license available online at http://creativecommons.org/licenses/by-nd/3.0/. The Authors further grant permission for reuse of images and text from the first page of the Work, provided that the reuse is for the purpose of promoting and/or summarizing the Work in scholarly venues and that any reuse is accompanied by a scientific citation to the Work.