DOI: 10.1111/cgf.12341

COMPUTER GRAPHICS *forum*
Volume 00 (2014), number 00 pp. 1–11

SATO: Surface Area Traversal Order for Shadow Ray Tracing

Jae-Ho Nah and Dinesh Manocha

University of North Carolina at Chapel Hill, NC, USA
jhnah@msl.yonsei.ac.kr, dm@cs.unc.edu

Abstract

We present the surface area traversal order (SATO) metric to accelerate shadow ray traversal. Our formulation uses the surface area of each child node to compute the TO. In this metric, we give a traversal priority to the child node with the larger surface area to quickly find occluders. Our algorithm reduces the pre-processing overhead significantly, and is much faster than other metrics. Overall, the SATO is useful for ray tracing large and complex dynamic scenes (e.g. a few million triangles) with shadows.

Keywords: ray tracing, shadows, traversal order

ACM CCS: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism–Ray Tracing.

1. Introduction

A key problem in interactive rendering is generating images with high-quality shadows. In general, ray tracing provides a simple solution that generates accurate hard and soft shadows with good scalability to large models. This involves tracing shadow and non-shadow rays and accelerating intersection tests using hierarchies.

When we trace a shadow ray, we do not need to exactly find the closest hit point; we are only interested in whether a shadow ray is occluded. This 'any hit' property can be exploited to use more efficient traversal orders (TOs) or heuristics for shadow rays; if an occluded shadow ray is first intersected with an occluder in an upper-level node, the traversal of the shadow ray can be quickly terminated. Many prior approaches use the 'any hit' property: the ray termination surface area heuristic (RTSAH) TO [IH11], the shadow ray distribution heuristic (SRDH) [FLF12] and the use of volumetric occluders [DKH09]. These techniques work well in static scenes, but can be slow for dynamic scenes when the hierarchy is updated at each frame.

Many applications use large and complex dynamic scenes with millions of triangles. In these applications, there is a trade-off between the time spent in updating the hierarchy and the time spent in traversing the hierarchy for ray intersections. A good, tight-fitting hierarchy can reduce the total time spent in tree traversal. Moreover, it is relatively easy to parallelize tree traversal using single instruction, multiple data or multi-core capabilities (SIMD). As a result, tree updating time can become a major bottleneck in large dynamic scenes. This has implications in terms of choice of techniques to accelerate traversal of shadow rays. If the additional overhead of the data-structure pre-processing is more than the reduced traversal time, use of such pre-processing may increase the overall render time.

1.1. Main results

We present the new surface area TO (SATO) metric for shadow ray tracing. Our formulation gives higher traversal priority to the child node with the larger surface area. This TO can quickly find large primitives in the upper-level nodes. Due to its simplicity, SATO offers the following advantages. First, it does not have high pre-processing overhead. Second, it can be easily integrated with existing SAH-based tree construction or update methods, so parallelization of SATO computation is simple. Third, when SATO is combined with the tree rotation method [KIS*12], the resulting algorithm can selectively update the TO only if a rotation occurs in a bounding volume hierarchy (BVH).

We combine the SATO metric with kd-trees and BVHs and compare its performance with RTSAH using the Manta interactive ray tracer [BSP06]. The SATO's traversal performance is competitive with that of RTSAH in our benchmarks. However, the computation cost of SATO is significantly lower than that of RTSAH. We highlight the performance benefits of SATO in large dynamic scenes.

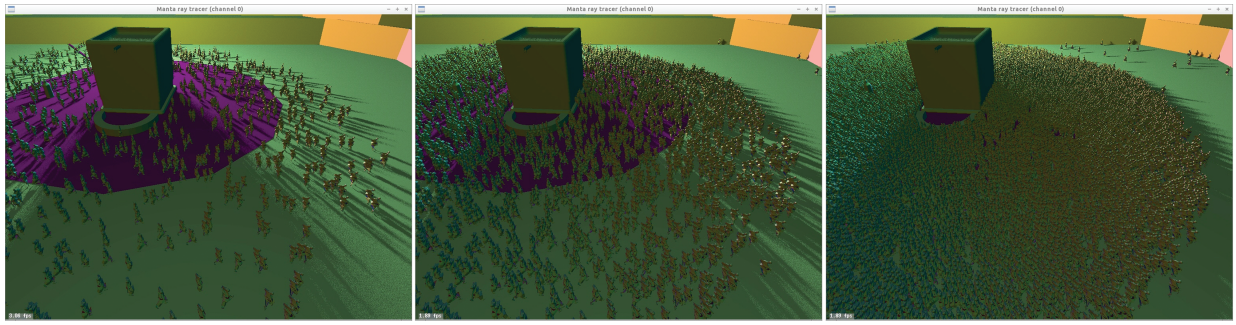

Figure 1: *Ray-traced images in the dense crowd simulation scenario with different numbers of agents: (left) 1 000 agents, (middle) 4 000 agents and (right) 16 000 agents. Each agent corresponds to a dynamic object in the scene, and at each frame, we update the hierarchy. Our surface area TO (SATO) metric only requires 1.6 ms with a 4-core Core i7 CPU to calculate the TO for 16 000 agents with 10.9M triangles, which makes it attractive for large dynamic scenes. Moreover, the SATO metric decreases the ray-tracing time by 15% in this complex dynamic scene. In contrast, the approximate RTSAH metric requires 208 ms to calculate the TO.*

The rest of the paper is organized in the following manner. We give a brief overview of related work in Section 2. We present the SATO in Section 3 and highlight its performance in Section 4.

2. Related Work

In this section, we first summarize the SAH. Next, we describe acceleration techniques for shadow ray tracing which are closely related to our work. Finally, we give an overview of hierarchy update methods that can be used with SATO.

2.1. Surface area heuristic

The SAH [GS87, MB90] has been widely used for high-quality tree construction. Greedy SAH construction determines the split position using the expected traversal cost using the following equation:

$$C_V(p) \approx K_T + K_I \left(\frac{SA(V_L)}{SA(V)} T_L + \frac{SA(V_R)}{SA(V)} T_R \right), \qquad (1)$$

where $C_V(p)$ is the expected cost of the voxel V with the split position p, K_T is a traversal cost, K_I is an intersection cost and V_{L} and V_{R} are voxels of the left and right child node, respectively. $SA(x)$ is the surface area of the voxel x, and T_{L} and T_{R} are the number of primitives in the left and right child node, respectively. This metric assumes that the probability of a node being pierced by rays is proportional to the node's surface area with finite random rays.

2.2. Acceleration of shadow ray tracing

Djeu *et al*. [DKH09] proposed the use of volumetric occluders in a kd-tree and the quick descent breadth-first search order. This method accelerates shadow ray tracing on opaque watertight meshes.

For more general shadow computation, Ize and Hansen [IH11] proposed the RTSAH TO for both BVHs and kd-trees. The RTSAH metric calculates the expected cost of terminating traversal in both the left and right child nodes. To specify which child node should be traversed first, a flag in each parent node is then updated using the expected cost; this flag is used for shadow ray traversal. The approximate RTSAH metric for BVHs [IH11] reduces the pre-processing cost by ignoring empty spaces in a BVH.

Feltman *et al*. [FLF12] presented the SRDH metric using the representative set of shadow rays. Like the ray distribution heuristic (RDH) [BH09], the SRDH exploits the traversal results of a small representative set of rays for tree construction. By exploiting the actual traversal results, the SRDH constructs a shadow-ray-specialized BVH and determines effective TO of the BVH. This method increases the overall pre-processing time by computing an additional BVH for shadow ray traversal, but this method may take fewer traversal steps than RTSAH.

Vinkler *et al*. [VHS12] presented a visibility-driven BVH construction algorithm. This algorithm employs an improved SAH that uses the visibility of primitives to construct higher quality BVHs. This method can accelerate the tracing of both shadow and non-shadow rays in highly occluded scenes.

Latuerbach *et al*. [LMM09] presented a selective ray-tracing algorithm for shadows. This algorithm first performs shadow mapping, after that, selectively performs ray tracing on potentially inaccurate pixels to compute correct shadows. Because this algorithm only shoots rays corresponding to a small subset of the entire pixels, shadows can be quickly computed with this algorithm.

2.3. Fast hierarchy updates

Because SAH construction has O($n$log$n$) time complexity [WH06], full SAH construction during each frame of a dynamic scene can be costly, especially for large scenes. Many techniques for fast hierarchy updates have been proposed to address this problem.

Fast tree reconstruction methods utilize cheaper split criteria or parallelization. The methods can be classified into SAH construction on CPUs [WBS07, CKL*10] and GPUs [LGS*09, WZL11, KA13], spatial median/SAH hybrids on CPUs [Wal07] and GPUs [ZHWG08, LGS*09, GPM11], object median/SAH hybrids on CPUs [SSK07] and approximate agglomerative clustering (AAC) on CPUs [GHFB13].

BV refitting [LYTM06, WBS07] updates a BVH without topological changes. This method can quickly update a BVH, but it can degrade tree quality. In order to prevent the tree quality degradation caused by BV refitting, Kopta *et al*. [KIS*12] presented a tree rotation algorithm that can be integrated into a refit procedure.

Other techniques are based on partial or lazy updates. In partial updates, only dynamic parts of the scene are updated. These methods either use a separate dynamic tree [SBSK03] or multi-level hierarchies [KNPY13]. The lazy tree construction method [DHW*11] computes the subtree associated with a node only when a ray enters that node.

Pre-processing overhead for specific cache-efficient tree layouts can become a bottleneck for ray-tracing dynamic scenes. In order to address this issue, Nah *et al*. [NPK*10, NPP*11] proposed an ordered depth-first layout for kd-trees. In this method, the child node with the larger surface area is stored next to its parent node using the SAH assumption. This results in better cache efficiency than depth-first layouts and can be easily combined with the SATO metric.

3. Surface-Area Traversal Order

In this section, we present the SATO metric that is used for fast shadow ray traversal. In terms of design, we have two major goals: to minimize the TO calculation time for dynamic scenes and to quickly find a large occluder for shadow ray tracing. To achieve these goals, we simply use the surface area of each node when we determine the TO. Note that we assume that the acceleration structure is a binary tree.

In order to evaluate the benefits of SATO, we introduce a sub-metric of SATO called PrimSATO. In PrimSATO, we calculate the average or maximum surface area of each primitive in each node and give the traversal priority to the child node with the higher value of the primitives' surface area. This method helps to quickly find larger occluders in ray traversal. We also introduce a primitive number TO (PrimNumTO); this metric gives the traversal priority to the child node with the less number of primitives. We then present three assumptions that are used to analyse the relationship among PrimSATO, PrimNumTO and SATO using the surface area of each node. To prevent confusion, we will refer to the latter metric as NodeSATO in the rest of the paper. After that, we describe PrimSATO, NodeSATO and PrimNumTO. Finally, we describe the ray traversal algorithm using SATO. Figure 2 illustrates the use of SATO.

3.1. Assumptions

Our metric is based on the following assumptions:

Assumption 1. *Large primitives are usually located in the upper-level nodes in a tree, so intersecting with a large primitive first can result in early termination of a shadow ray.*

Assumption 2. *In SAH-constructed kd-trees, the child node with the larger surface area between two child nodes has a higher probability of enclosing larger primitives.*

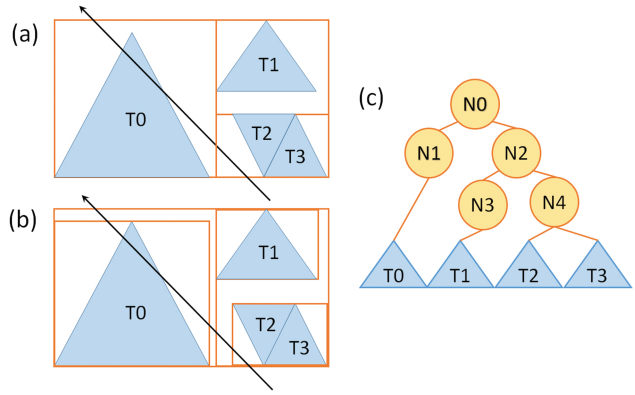

Figure 2: *An example of the SATO with a kd-tree (a) and a BVH (b). (c) shows the hierarchy of both the kd-tree in (a) and the BVH in (b). In this example, PrimSATO, PrimNumTO and NodeSATO compute the same TO. To quickly find a larger occluder, a shadow ray first visits the child node with the higher traversal priority as determined by SATO. In the above figure, when the SATO is used, TO is N0, N1 and T0. In contrast, when the front-to-back TO is used, the TO is N0, N2, N4, T2, T3, N1 and T0.*

Assumption 3. *In BVHs, the child node with the larger surface area between two child nodes also has a higher probability of enclosing larger primitives.*

Assumption 1 is related to PrimSATO. This assumption is based on the character of SAH-constructed trees; the SAH tends to keep larger primitives near the root [Smi98]. Therefore, if we visit a node enclosing a large primitive first, the number of traversal steps can be reduced.

Assumptions 2 and 3 are related to NodeSATO. These assumptions are based on Assumption 1, so they imply a close relationship between NodeSATO and PrimSATO.

We can explain the reasoning behind Assumption 2 based on the following criteria. In SAH kd-trees, the child node with the larger surface area would have a higher probability of containing a smaller number of primitives. This is possible when the left node's expected cost $(SA(V_L) \cdot T_L$ in Equation (1) and the right node's expected cost $(SA(V_R) \cdot T_R)$ are comparable. In many cases, the SAH costs would be comparable in the upper-level nodes, because the split plane is inside the median interval (between the spatial and object median) if there are no objects intersecting with the split plane [MB90, Hav00]. In this case, it is often one of these two cases: It either has a more sparse primitive distribution, or a higher average surface area of the primitives in that child node. Empty spaces between primitives in the first case can be reduced based on empty space maximizing techniques [Hav00, HKRS02]. Therefore, the second case will have a high probability, and the child node with the larger surface area will have a higher probability of enclosing large primitives. Additionally, if SAH values are comparable, a higher surface area implies less number of primitives. Thus, NodeSATO and PrimNumTO will compute similar TOs in a SAH kd-tree.

BVHs have different characteristics than kd-trees, and the analysis given for kd-tress above may not be applicable to BVHs. Because BVH nodes have a tighter fit to the primitives than kd-tree nodes [LYTM06], there may be empty space that is not enclosed by both child nodes. Thus, the SAH costs between the child nodes may not be comparable if the split plane is determined in such a way that it maximizes empty spaces. It means that the correlation between the node's surface area and the number of primitives in BVHs can be lower than that in kd-trees. However, in contrast to kd-trees, the BV of a parent BVH node includes the BVs of its primitives. Therefore, the child node with the larger surface area among two child BVH nodes will also have a higher probability of enclosing larger primitives, as is the case with kd-trees (Assumption 3).

3.2. PrimSATO

The goal of PrimSATO is to quickly find a large occluder when tracing a shadow ray, and is same as that of NodeSATO. In order to determine which child node has larger primitives, the PrimSATO calculation is performed during the tree construction. Before the tree construction, we first calculate the surface area of each primitive. After that, when an inner node is created, we average the surface area values of the primitives in the node or find the maximum value of the primitives. We call the method using the average value $\text{PrimSATO}_{\text{AVG}}$ and the method using the maximum value $\text{PrimSATO}_{\text{MAX}}$. By comparing the calculated average or maximum value of the two child nodes, we can give the traversal priority to the child node with the higher surface area value of the primitives.

The PrimSATO procedure is straightforward and simple, but the pre-processing cost of PrimSATO may not be negligible in large dynamic scenes due to the cost of computing the surface areas. Moreover, PrimSATO cannot be tightly coupled with BVH update methods [LYTM06, WBS07, KIS*12]. The reason is that we cannot get the primitive information enclosed by an inner node after the tree is constructed; a BVH node usually stores its bounding box and child node pointers.

3.3. NodeSATO and PrimNumTO

NodeSATO addresses the weaknesses of PrimSATO; NodeSATO compares only the surface areas of each child node, giving traversal priority to the child node with the larger surface area. If there is a high traversal similarity between NodeSATO and PrimSATO, NodeSATO will be a more attractive metric in terms of TO calculation time.

NodeSATO's simplicity means that this metric can be easily integrated into all SAH-based tree construction/update methods (see Appendix for details). If SAH is used for kd-tree construction, NodeSATO metric requires only one additional computation (the node comparison). Since we calculate the child nodes' surface areas during SAH kd-tree construction, the calculated surface areas can be reused. Parallelization of NodeSATO is also trivial if parallel tree construction/update methods [Wal07, CKL*10, KIS*12] are used. Additionally, NodeSATO can be combined with lazy tree construction [DHW*11], because NodeSATO requires only local information about the node (child nodes' surface areas) instead of a traversal of the entire subtree.

When NodeSATO is combined with a tree rotation method [KIS*12] in dynamic scenes, we can selectively update the TO of only the rotated nodes. According to [KIS*12], a rotation is performed only if the rotation can decrease the SAH cost of a node. Therefore, if a rotation is not performed, it indicates that the node's bounding box is still near optimal and we can simply reuse the TO calculated during the previous frame. If a rotation is performed, NodeSATO calculations are performed with up to three inner nodes (the current node and the inner child nodes of the current node) because a rotation affects a subtree with a depth of two.

PrimNumTO gives traversal priority to the child node with fewer primitives. Thus, PrimNumTO computation only requires a simple comparison between the child nodes. However, this method cannot be combined with BVH update methods [LYTM06, WBS07, KIS*12] due to the same reasons that were described for PrimSATO in Section 3.2

3.4. Ray traversal with the SATO metric

NodeSATO, PrimSATO and PrimNumTO have a common traversal procedure, which is similar to RTSAH traversal [IH11]. SATO requires a 1-bit flag per node (e.g. the isLeftCheaper flag in Manta), which can be embedded in the node structure without additional memory overhead. When tracing shadow rays, the flag bit is used in determining the TO. Instead of using the flag, we can rearrange the node's child order using SATO. However, if the acceleration structure is not a BVH but a kd-tree, the 1-bit flag is still needed for rearrangement, because kd-tree node rearrangement using surface area is the same as the ordered depth-first layout [NPK*10, NPP*11]; the 1-bit flag is used to indicate whether the node has been rearranged or not for front-to-back traversal.

4. Performance and Comparison

In this section, we first compare NodeSATO, PrimSATO and other TO metrics using the benchmarks used in [IH11] and [FLF12]. Next, we present experimental results for several dynamic scenes. Finally, we analyse and describe the limitations of our metric.

4.1. SATO versus other TO metrics

We have tested our method in the Manta interactive ray tracer [BSP06] (revision 2542) on a PC with 3.4 GHz Intel Core i7 with 8 GB RAM. Ray tracing was performed using eight threads with hyperthreading; tree construction and pre-processing for each algorithm was performed using a single thread.

We use the static test scenes used in [IH11] (Figure 3) for evaluating the performance. The Mad Science scene with 80K triangles was rendered using five samples per pixel, 14 area lights with 25 samples per shading point and 36 ambient occlusion rays per shading point. The Carnival scene with 446K triangles was rendered using five samples per pixel and 100 ambient occlusion rays. The Bedroom scene with 361K triangles was rendered using 11 point lights. The Shadow Overlap scene with 2056K triangles was rendered using

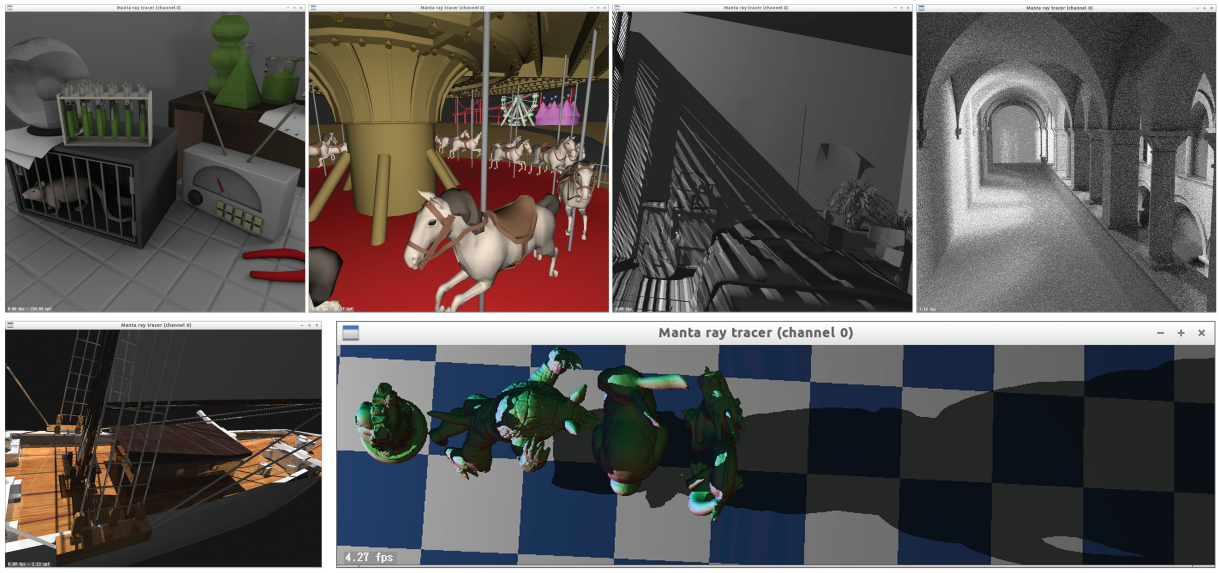

Figure 3: *Static benchmark scenes: (top row) Mad Science, Carnival, Bedroom, Sponza, (bottom row) Ship and Shadow Overlap. The Shadow Overlap scene was rendered at 1024×256 pixels, the Ship scene was rendered at 1024×768 pixels and the others were rendered at 1024×1024 pixels.*

Table 1: *TO calculation time for a BVH and a KD-tree (unit: millisecond, lower is better). A single thread was used.*

	BVH						kd-tree				
	RTSAH	Approx RTSAH	PrimSATO AVG	PrimSATO MAX	NodeSATO	PrimNum	RTSAH	PrimSATO AVG	PrimSATO MAX	NodeSATO	PrimNum
Mad Science	28.8	0.9	2.3	2.1	<0.1	<0.1	10.3	5.3	5.1	0.2	0.3
Carnival	185.3	5.4	19.6	18.5	0.6	0.2	39.9	25.8	23.1	1.4	1.6
Bedroom	135.7	3.7	12.0	10.9	0.4	0.1	53.2	33.7	32.1	0.8	1.2
Shadow Overlap	752.7	29.2	154.2	145.0	3.1	1.0	173.8	106.3	103.8	2.8	3.4
Sponza	18.4	0.7	1.8	1.6	<0.1	<0.1	9.0	2.9	2.2	0.3	0.3
Ship	1.3	<0.1	<0.1	<0.1	<0.1	<0.1	0.7	0.2	0.2	<0.1	<0.1

one sphere area light with 10 samples per sample point. This scene consists of four objects with different complexity: Happy Buddha with 1M triangles, Armadillo with 30K triangles, Bunny with 69K triangles and Dragon with 871K triangles.

The two following benchmarks are difficult scenes in terms of computing the optimal TO, in contrast to the scenes described above. The Sponza scene with the arcade view is described in [FLF12] consists of 66K triangles and two sphere area light sources. This scene was rendered with two-level path tracing, and the number of samples per shading point was one. This scene has low amount of occlusion and an equal number of shadow and non-shadow rays, so TOs do not greatly affect the overall traversal time. The Ship scene with 4K triangles was rendered using 1 area light with 16 samples per sample point. In this scene, an axis-aligned occluder (a cabin) is located near the shaded deck and non-axis-aligned occluders (thin ropes and an anchor) are located near the light source. A BVH node that includes non-axis-aligned objects has large empty spaces, assigning TO to non-axis-aligned occluders may not be a good choice. Thus, the front-to-back order is near optimal.

To measure the performance of NodeSATO and PrimSATO in different traversal methods, we use three traversal methods: packetized ranged BVH traversal [WBS07] with 8×8 ray packets, single-ray BVH traversal and single-ray kd-tree traversal. With this experimental setup, we compare a front-to-back TO, a random TO [FLF12], two RTSAH implementations (RTSAH and approximate RTSAH) [IH11] and four SATO implementations (NodeSATO, $\text{PrimSATO}_{\text{AVG}}$, $\text{PrimSATO}_{\text{MAX}}$ and PrimNumTO). As described in [IH11], approximate RTSAH is only applicable to BVHs because approximate RTSAH for BVHs and RTSAH for kd-trees are essentially same. For the random TO, we have used a fast random-number generation function in Manta [BSP06], because the standard rand() function is not suitable for parallel processing. In all TOs, early termination is commonly performed when the first intersection is found.

Table 2: *Render time (unit: seconds, lower is better). Eight threads with hyperthreading were used. ARP is an abbreviation of average relative performance when a front-to-back TO is the reference method (higher is better). Approximate RTSAH is only applicable to BVHs, as described in [IH11]. Bold values indicate the highest performance.*

	Front to back	Random	RTSAH	Approx RTSAH	PrimSATO$_{AVG}$	PrimSATO$_{MAX}$	NodeSATO	PrimNum
Packetized BVH traversal								
Mad Science	392.71	266.62	247.38	240.05	**226.31**	228.87	231.39	257.71
Carnival	151.19	114.00	81.46	97.34	83.22	98.54	82.55	**77.35**
Bedroom	0.48	0.28	0.27	0.26	**0.24**	0.26	0.25	0.26
Shadow Overlap	0.18	0.18	0.14	0.14	**0.13**	0.14	0.14	0.13
Sponza	0.76	0.75	0.75	0.73	0.74	0.74	0.76	**0.73**
Ship	1.31	1.40	1.46	1.42	1.32	**1.31**	1.41	1.49
ARP	1.00×	1.24×	1.40×	1.38×	**1.48×**	1.39×	1.44×	1.44×
Single-ray BVH traversal								
Mad Science	577.71	452.49	320.57	**316.22**	326.85	325.32	334.72	380.81
Carnival	163.88	91.06	63.35	77.36	63.77	79.10	65.96	**59.24**
Bedroom	2.36	1.79	1.50	1.49	**1.47**	1.52	1.47	1.54
Shadow Overlap	0.24	0.23	0.22	0.22	**0.21**	0.21	0.21	0.21
Sponza	0.93	0.95	0.94	0.91	0.93	**0.91**	0.94	0.91
Ship	2.57	2.62	2.66	2.62	2.49	**2.37**	2.62	2.72
ARP	1.00×	1.23×	1.50×	1.44×	**1.52×**	1.44×	1.48×	1.48×
Single-ray kd-tree traversal								
Mad Science	402.86	295.11	**238.51**	–	276.23	240.06	249.61	293.20
Carnival	102.53	59.06	67.22	–	61.48	65.24	59.01	**58.29**
Bedroom	1.08	0.86	0.82	–	0.89	**0.81**	0.89	0.90
Shadow Overlap	0.19	0.21	0.20	–	0.18	**0.18**	0.18	0.18
Sponza	**0.46**	0.47	0.48	–	0.48	0.47	0.46	0.48
Ship	1.14	1.16	**1.08**	–	1.12	1.09	1.14	1.14
ARP	1.00×	1.21×	1.25×	–	1.23×	**1.28×**	1.26×	1.22×

Table 3: *The number of traversal steps per ray (left) and the number of intersection tests per ray (right). Lower is better.*

	Front to back	Random	RTSAH	Approx RTSAH	PrimSATO$_{AVG}$	PrimSATO$_{MAX}$	NodeSATO	PrimNum
Packetized BVH traversal								
Mad Science	120.5 / 21.0	90.8 / 12.0	83.7 / 11.0	82.7 / 11.0	82.6 / 9.7	82.6 / 9.7	84.5 / 9.8	92.3 / 11.9
Carnival	378.8 / 24.6	297.2 / 17.5	204.6 / 14.3	257.1 / 13.5	205.5 / 15.9	236.5 / 18.5	207.4 / 15.9	206.7 / 9.4
Bedroom	100.7 / 12.6	65.1 / 8.5	64.9 / 7.3	64.7 / 6.8	64.0 / 6.4	65.7 / 7.1	64.5 / 6.6	64.3 / 7.0
Shadow Overlap	111.4 / 2.6	111.9 / 2.6	86.2 / 2.5	84.2 / 2.5	78.6 / 2.6	86.5 / 2.6	84.1 / 2.5	80.6 / 2.6
Sponza	276.3 / 11.2	273.8 / 10.9	273.7 / 11.1	267.7 / 11.1	270.4 / 11.1	267.7 / 11.2	278.3 / 11.3	265.3 / 10.8
Ship	50.3 / 27.1	51.0 / 28.5	51.2 / 30.6	51.8 / 29.0	47.6 / 27.6	50.0 / 24.9	48.4 / 31.8	49.2 / 32.2
Single-ray BVH traversal								
Mad Science	57.1 / 8.5	45.4 / 6.0	36.2 / 4.0	35.2 / 4.3	40.4 / 3.7	40.5 / 3.7	41.1 / 3.7	47.3 / 4.5
Carnival	84.4 / 13.3	45.9 / 6.5	33.9 / 5.0	42.2 / 5.2	34.3 / 5.3	42.6 / 6.9	34.3 / 5.6	32.3 / 4.0
Bedroom	74.9 / 9.4	57.2 / 7.5	50.2 / 5.8	51.1 / 5.4	50.6 / 5.2	51.6 / 5.6	50.7 / 5.3	52.5 / 5.7
Shadow Overlap	28.6 / 1.7	27.3 / 1.5	27.7 / 1.6	27.5 / 1.6	27.4 / 1.6	27.7 / 1.7	27.0 / 1.6	27.3 / 1.5
Sponza	61.9 / 3.2	63.0 / 3.1	64.1 / 3.2	62.1 / 3.1	63.7 / 3.2	62.3 / 3.2	64.2 / 3.2	61.8 / 3.1
Ship	44.3 / 25.3	43.5 / 27.0	44.7 / 28.4	45.6 / 26.9	41.1 / 26.0	43.5 / 23.2	41.6 / 30.5	43.2 / 30.0
Single-ray kd-tree traversal								
Mad Science	44.3 / 8.9	37.9 / 5.4	30.4 / 4.2	–	33.7 / 5.3	33.5 / 3.8	33.0 / 4.5	34.6 / 5.5
Carnival	52.6 / 9.8	37.3 / 5.3	37.1 / 5.3	–	34.6 / 4.8	36.1 / 5.3	32.7 / 4.4	32.6 / 4.0
Bedroom	41.1 / 8.3	32.5 / 6.1	31.2 / 6.5	–	36.0 / 6.3	30.2 / 6.0	35.8 / 6.7	35.8 / 6.8
Shadow Overlap	33.4 / 2.3	32.9 / 1.9	32.7 / 2.3	–	31.7 / 2.0	31.5 / 2.0	32.4 / 2.0	32.4 / 2.0
Sponza	37.5 / 3.0	36.9 / 2.8	38.4 / 3.0	–	38.2 / 3.0	37.6 / 3.0	38.1 / 2.9	38.2 / 3.0
Ship	33.2 / 6.8	32.7 / 7.2	31.0 / 6.4	–	31.7 / 6.8	31.5 / 6.5	32.6 / 7.4	34.0 / 6.9

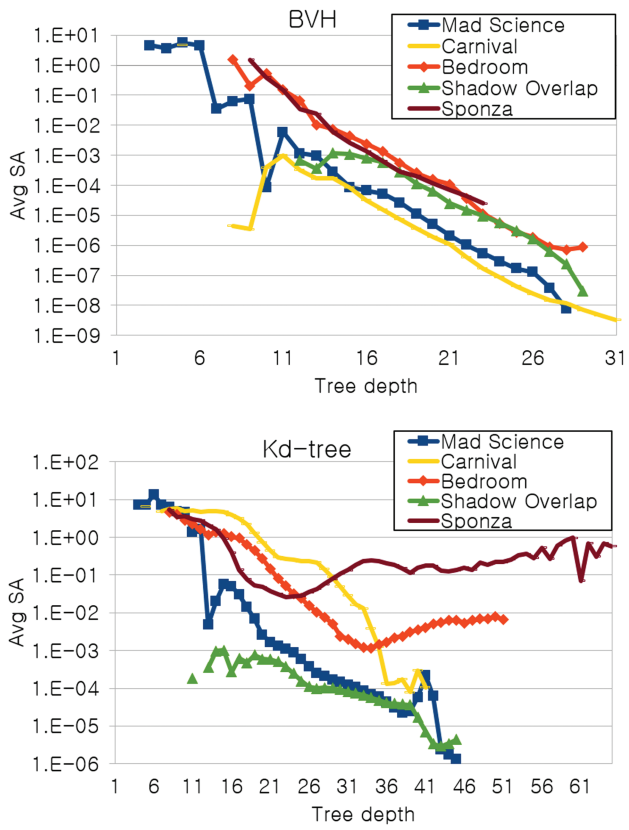

Figure 4: *The average surface area of primitives in one leaf node per tree depth. The above figure indicates the result of the BVHs, and the below figure indicates the result of the kd-trees. These results support Assumption 1 in Section 3.*

We first measure the TO calculation time of each method (Table 1). This task was performed as a pre-process in static scenes, and a single thread was used. For the RTSAH and approximate RTSAH calculation, we have used a separate function implemented in Manta [BSP06]. For NodeSATO, PrimSATO and PrimNumTO calculations, we have implemented the TO calculation code in the existing tree construction function. According to the results in Table 1, PrimSATO is faster than RTSAH but slower than approximate RTSAH. NodeSATO and PrimNumTO are the fastest method in kd-trees and BVHs, respectively. The reason is that NodeSATO requires an additional surface area calculation per node in BVH construction, but does not require that computation in kd-tree construction.

In contrast to the embedded NodeSATO/PrimSATO calculation codes in the tree build function, the RTSAH calculation code has been implemented as a separate function. Thus, in this RTSAH implementation, node data should be fetched again for the separate function. However, even if we consider the implementation difference, the experimental result shows that the NodeSATO and PrimNumTO calculations are significantly faster than fast approximate RTSAH calculation.

Second, we measure ray-tracing performance using render time (Table 2). We also measure additional statistics for better analysis

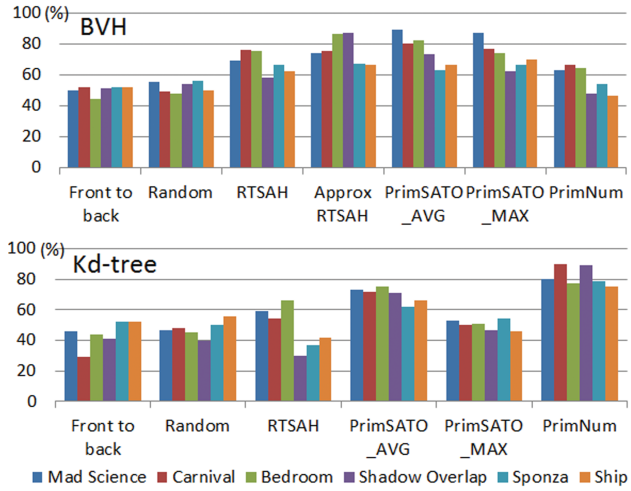

Figure 5: *Comparison of TO similarity between NodeSATO and other metrics. This result supports Assumptions 2 and 3 in Section 3.*

(Table 3). Among the metrics, $\text{PrimSATO}_{\text{AVG}}$ and $\text{PrimSATO}_{\text{MAX}}$ demonstrate the highest speedups in BVHs and kd-trees, respectively. These different speedups can be explained by the difference between object hierarchies and spatial hierarchies. Each primitive is located in a single BVH node if spatial splits are not used for the BVH construction, so $\text{PrimSATO}_{\text{AVG}}$ is a more reasonable choice in this case. In contrast, a primitive can be located in multiple kd-tree nodes. That is, a ray can find the hit point on the outside of the current node if a primitive intersects the current node and the other nodes. Thus, for kd-trees, $\text{PrimSATO}_{\text{MAX}}$ can be more efficient. Additionally, NodeSATO and PrimNumTO do not result in the fastest performance, but exhibit performance that is comparable to RTSAH and PrimSATO. The random TO exhibits lower speedups than RTSAH and SATO.

The Sponza and Ship scenes demonstrate the limitations of each method. Occluded shadow rays account for a small part of the total rays in the Sponza scene, so all TO metrics do not result in a significant difference in the overall performance. In the Ship scene with BVHs, only PrimSATO results in similar or faster performance than front-to-back ordering. The reason is that PrimSATO based on actual primitives' surface areas is robust for non-axis-aligned objects.

To evaluate the performance of SATO, we perform two additional experiments. First, to verify Assumption 1 in Section 3 we measure the average surface area of primitives in a leaf node per tree depth (Figure 4). The results in Figure 4 explain the advantage of PrimSATO. According to the results, the average surface area of primitives in a leaf node is inversely proportional to the tree depth. In other words, higher-level nodes have a higher probability of enclosing larger primitives. Even though the kd-trees for the Sponza and Bedroom scenes show gradual increases from specific depths, these results can be explained by large primitives overlapped with multiple nodes.

Second, we measure the traversal similarity between NodeSATO and other metrics (Figure 5). To calculate the similarity, we compare NodeSATO's TO flag and the others' TO flags at each traversal step.

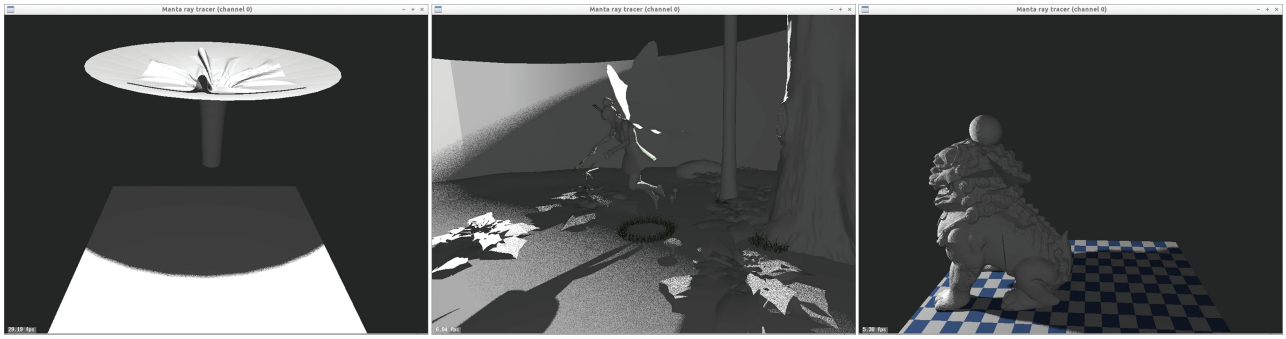

Figure 6: *Dynamic benchmark scenes: Funnel, Fairy Forest and Lion. The captured images were rendered with soft shadows.*

The data in Figure 5 were collected by single-ray BVH and kd-tree traversal. In the kd-tree traversal, NodeSATO exhibits higher similarities with $\text{PrimSATO}_{\text{AVG}}$ and PrimNumTO. This result supports Assumption 2. In contrast, $\text{PrimSATO}_{\text{MAX}}$ does not exhibit a high similarity, because the surface area of a node can be smaller than that of the primitive in the node. This situation occurs when a large primitive overlaps multiple nodes. In the BVH traversal, NodeSATO has a higher similarity with $\text{PrimSATO}_{\text{AVG}}$ and $\text{PrimSATO}_{\text{MAX}}$. This result supports Assumption 3; there is a high traversal similarity between PrimSATO and NodeSATO.

4.2. Experimental results with dynamic scenes

We use the same rendering system and hardware configuration described in Section 4.1 to evaluate our approach on dynamic scenes. To deal with dynamic objects, we use the tree rotation algorithm developed by Kopta *et al*. [KIS*12]. We have integrated NodeSATO into the tree rotation function. We also evaluate the selective NodeSATO update scheme (described in Section 3.3 and Appendix) and measure its performance by enabling and disabling the scheme. PrimSATO and PrimNumTO are excluded in this experiment, because they are difficult to integrate into the BV refit/rotation procedure, as described in Section 3. For ray traversal, we use the ranged BVH traversal algorithm [WBS07] with the 8×8 packet size. Both tree updates (including the NodeSATO calculation) and ray tracing were performed using eight threads with hyperthreading; the RTSAH and approximate RTSAH calculations were performed using a single thread.

We test our method in four dynamic scenes (Figures 1(right) and 6). The Funnel scene only consists of 18K triangles, so in this scene, render time is much more important than tree update and TO calculation time. The Fairy is a mid-sized game-like scene with 174K triangles, and the Lion scene is a large scene with 1.6M triangles. The Crowd Simulation scene [CGZM11] (Figure 1), with 10.9M triangles, is the largest scene in our test set; the tree update and TO calculation times are significant in this scene.

We apply two different shadow settings to the test scenes: hard shadows and soft shadows. For hard shadows, one point light source is used. For soft shadows, one spherical area light source and four samples per light are used. The soft-shadow setting increases the proportion of shadow rays, so the performance improvements by using an efficient TO also increase in this case.

Table 4 summarizes the experimental results in the dynamic scenes. According to the results, NodeSATO achieves 4–18% performance improvements in the total frame time (tree update+TO calculation+rendering). Additionally, the selective NodeSATO

Table 4: *Experimental results in dynamic scenes with soft and hard shadows (unit: millisecond, lower is better). Note that the BVH update time includes keyframe animation time, BV refitting time and rotation time. Note that the RTSAH/approximate RTSAH TO was calculated using a single thread.*

	BVH update time	TO calculation time	Render time (hard/soft)	Total time (hard/soft)
Funnel (18K triangles)				
Front to back	0.6	0.0	13.7/33.5	14.3/34.1
Random	0.6	0.0	13.8/34.7	14.4/35.3
RTSAH	0.6	11.8	12.8/27.3	25.2/39.7
Approx RTSAH	0.6	0.9	13.5/31.1	15.0/32.6
NodeSATO	0.6	<0.1	13.0/31.4	13.6/32.0
NodeSATO (selective)	0.6	<0.1	12.6/28.5	13.2/29.1
Fairy forest (174K triangles)				
Front to back	6.1	0.0	57.7/167.4	63.8/173.5
Random	6.1	0.0	49.2/152.2	55.3/158.3
RTSAH	6.1	95.6	51.1/155.5	152.8/257.2
Approx RTSAH	6.1	3.3	49.5/149.2	58.9/158.6
NodeSATO	6.1	<0.1	48.3/149.1	54.4/155.2
NodeSATO (selective)	6.1	<0.1	47.7/149.6	53.8/155.7
Lion (1.6M triangles)				
Front to back	52.2	0.0	58.2/158.5	110.4/210.7
Random	52.2	0.0	58.4/163.6	110.6/215.8
RTSAH	52.2	950.9	54.8/122.5	1057.9/1125.6
Approx RTSAH	52.2	37.5	54.1/119.6	143.8/209.3
NodeSATO	52.2	1.2	54.3/125.5	107.7/178.9
NodeSATO (selective)	52.2	0.1	53.4/129.3	105.7/181.6
Crowd simulation (10.9M triangles)				
Front to back	369.1	0.0	201.7/442.5	570.8/811.6
Random	369.1	0.0	203.2/442.4	572.3/811.5
RTSAH	369.1	5389.5	167.7/385.9	5926.3/6144.5
Approx RTSAH	369.1	208.5	166.9/384.7	740.5/958.3
NodeSATO	369.1	8.2	172.1/379.7	549.4/757.0
NodeSATO (selective)	369.1	1.6	169.8/375.3	540.5/746.0

update scheme is effective in reducing the TO calculation time. This is possible because the rotation percentage in the Funnel, Fairy Forest, Lion and Crowd Simulation scenes is only 4.8%, 8.4%, 7.4% and 6.4%, respectively. The render time with and without the selective NodeSATO update scheme is similar even though the selective update scheme partially reuses the TO calculated during the previous frame. Due to the parallelization and the selective update schemes, NodeSATO could achieve very low TO calculation times in a large scene (1.6 ms in the Crowd Simulation scene).

In contrast, RTSAH increases the total frame time in all scenes due to the additional TO calculation time; fast approximate RTSAH also increases the total frame time in the Lion scene with hard shadows and the Crowd Simulation scene. This overhead can be lowered by parallelization of RTSAH calculation and code in-lining, as NodeSATO does. However, NodeSATO is still faster than the parallelized RTSAH and approximate RTSAH methods in terms of TO calculation time. Additionally, random TO does not yield noticeable performance improvements except for the Fairy Forest scene.

Because of tree update overheads and simple shadow settings in the dynamic scenes, the relative performance improvements of NodeSATO in dynamic scenes are lower than those in the static scenes. However, in cases where we apply both faster tree update methods and complex shadows, we expect that NodeSATO will exhibit similar performance improvements on static and dynamic scenes, because NodeSATO has negligible TO calculation overhead. Also, we expect that performance gap between NodeSATO and approximate RTSAH will increase in more complex dynamic scenes because of the TO calculation overhead.

4.3. Analysis and limitations

4.3.1. *Comparison to SRDH*

NodeSATO and PrimSATO would not be faster than SRDH [FLF12] in static scenes. This is because our metric only determines the TO, but SRDH affects both tree construction and TO. However, the SRDH increases tree construction time by up to 20 s for pre-rendering using representative ray sets. In contrast, NodeSATO, with its selective update scheme, only requires less than 2 ms to determine the TO. Therefore, we believe that NodeSATO is better suited for dynamic scenes than SRDH.

4.3.2. *Limitations*

Our SATO metric does not guarantee performance improvements in all situations. First, NodeSATO, PrimSATO and PrimNumTO only accelerate occluded shadow rays, just as RTSAH [IH11] does. Thus, if most shadow rays are not occluded or the fraction of shadow rays is relatively low, performance improvements from using our metric would decrease.

Second, NodeSATO and PrimNumTO assume that the tree is constructed or updated by the SAH. If the tree is constructed by either spatial median or object median, NodeSATO and PrimNumTO might not work well. Additionally, if BV refitting is used for BVH updates, NodeSATO's efficiency can be degraded due to large BVs. Thus, additional tree update approaches based on the SAH, such as the tree rotation algorithm [KIS*12] or treelet restructuring [KA13], should be used if BV refitting significantly degrades the tree quality.

Third, the SATO metric's efficiency is influenced by the scene characteristics. For example, if there are a large object that consists of many highly tessellated triangles and a small object that consists of a few large triangles, NodeSATO will pick the former object for giving traversal priority and cannot accelerate the shadow ray traversal. In this case, PrimSATO and PrimNumTO will be more effective. However, if variability in scene primitive size is very small, all our metrics (NodeSATO, PrimSATO and PrimNumTO) may not find an optimal traversal route, because they are designed to quickly find large occluders in the upper-level nodes.

5. Conclusions and Future Work

We have presented the SATO metric to accelerate shadow ray tracing. To determine the TO between two child nodes, the SATO compares each node's surface area (NodeSATO), the surface areas of the primitives in the node (PrimSATO) or the number of primitives in the node (PrimNumTO); NodeSATO and PrimSATO assign higher traversal priority to a node with larger surface area, and PrimNumTO gives traversal priority to a node with fewer primitives. Each metric has its own strength. NodeSATO can be easily implemented in existing SAH-based tree construction or update methods. Additionally, NodeSATO has negligible calculation overheads, so it is very useful in dynamic scenes. PrimSATO is competitive with other metrics when compared on the basis of traversal performance, particularly with scenes including non-axis-aligned triangles. PrimNumTO has the fastest TO calculation time with BVH construction.

In terms of future work, we would like to combine the SATO with other methods and measure the effects of the combination. First, if static and dynamic scenes are handled separately [SBSK03, KNPY13], the SRDH can be used for high-quality static tree/subtree construction, as well as TO determination, while the SATO can be used for dynamically updating trees or subtrees. Additionally, we are interested in combining the SATO with lazy tree construction [DHW*11] and AAC BVH construction [GHFB13]. Because the AAC algorithm lifts large occluders to higher levels of the BVH [GHFB13], this could be a synergistic combination. Also, the combination of SATO and an ordered depth-first layout [NPK*10, NPP*11] could provide both lower traversal steps and cache-miss rates in kd-tree traversal. Finally, an extension of the tree quality metric [AKL13] to the special case of shadow rays can be a future research topic, as described in [AKL13].

Acknowledgements

We would like to thank Junho Park for the fruitful discussions, the reviewers for their valuable comments and the following people for helping us with our experimental setup: Thiago Ize (the static scenes used in the RTSAH paper); Priyadarshi Sharma, Sujeong Kim and Sean Curtis (the dynamic scenes).

This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded

by the Ministry of Education (NRF-2012R1A6A3A03040332) and in part by ARO Contracts W911NF-10-1-0506, W911NF-12-1-0430, W911NF-13-C-0037 and NSF awards 0917040 and 1320644.

The Mad Science scene is by Dan Konieczka and Giorgio Luciano; the Carnival by Dan Konieczka, the Bedroom was modelled by David Vacek and designed by David Tousek; all three scenes are available from the 3dRender.com Lighting Challenges. The Happy Buddha, Bunny, Dragon and Armadillo are courtesy of the Stanford Computer Graphics Laboratory. The Sponza, Ship and Fairy Forest scenes are courtesy of Marko Dabrovic, Arauna realtime ray tracing and the Utah 3D Animation Repository, respectively. The Funnel and Lion scenes are courtesy of the UNC Dynamic Scene Benchmarks.

Appendix: Pseudo-Code of the NodeSATO Metric

1) NodeSATO calculation in tree construction

find the best split plane

if (the split can minimize the SAH cost)

  make an inner node

  compare each child node's SA

  set isLeftCheaper of the node

else

  make a leaf node

2) Selective NodeSATO calculation with a tree rotation

if (a rotation can decrease the SAH cost)

  do the rotation

  compare the SAs of the rotated inner nodes

  set isLeftCheaper of the nodes

else

  skip the TO calculation and keep the isLeftCheaper flag

3) BVH traversal using the SATO

  (same as the RTSAH traversal)

if (a shadow ray intersects with an inner node)

  front_son = node.isLeftCheaper? 0 : 1;

  do an intersection test with node.child+front_son

  do an intersection test with node.child+1-front_son

References

[AKL13] Aila T., Karras T., Laine S.: On quality metrics of bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2013* (2013), pp. 101–107.

[BH09] Bittner J., Havran V.: Rdh: Ray distribution heuristics for construction of spatial data structures. In *SCCG '09: Proceedings of the 25th Spring Conference on Computer Graphics* (2009), pp. 51–58.

[BSP06] Bigler J., Stephens A., Parker S. G.: Design for parallel interactive ray tracing systems. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006* (2006), pp. 187–196.

[CGZM11] Curtis S., Guy S. J., Zafar B., Manocha D.: Virtual tawaf: A case study in simulating the behavior of dense, heterogeneous crowds. In *Proceedings of the 1st IEEE Workshop on Modeling, Simulation and Visual Analysis of Large Crowds* (2011).

[CKL*10] Choi B., Komuravelli R., Lu V., Sung H., Bocchino R. L., Adve S. V., Hart J. C.: Parallel SAH k-D tree construction. In *Proceedings of the Confernce on High Performance Graphics 2010* (2010).

[DHW*11] Djeu P., Hunt W., Wang R., Elhassan I., Stoll G., Mark W. R.: Razor: An architecture for dynamic multiresolution ray tracing. *ACM Transactions on Graphics 30*, 5 (2011), 115:1–115:26.

[DKH09] Djeu P., Keely S., Hunt W.: Accelerating shadow rays using volumetric occluders and modified kd-tree traversal. In *Proceedings of the Conference on High Performance Graphics 2009* (2009), pp. 69–76.

[FLF12] Feltman N., Lee M., Fatahalian K.: SRDH: Specializing BVH construction and traversal order using representative shadow ray sets. In *Proceedings of Conference on High Performance Graphics 2012* (2012), pp. 49–55.

[GHFB13] Gu Y., He Y., Fatahalian K., Blelloch G.: Efficient BVH construction via approximate agglomerative clustering. In *Proceedings of the Conference on High Performance Graphics 2013* (2013), pp. 81–88.

[GPM11] Garanzha K., Pantaleoni J., McAllister D.: Simpler and faster HLBVH with work queues. In *Proceedings of the Conference on High Performance Graphics 2011* (2011), pp. 59–64.

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

[Hav00] Havran, V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University in Prague, Nov. 2000.

[HKRS02] Hurley J., Kapustin, A., Reshetov A. Soupikov A.: Fast ray tracing for modern general purpose CPU. In *Proceedings of Graphicon 2002* (2002).

[IH11] Ize, T., Hansen C.: RTSAH traversal order for occlusion rays. *Computer Graphics Forum* (*Proceedings of EUROGRAPHICS 2011*) *30*, 2 (2011), 297–305.

[KA13] Karras T., Aila T.: Fast parallel construction of high-quality bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2013* (2013), pp. 89–99.

[KIS*12] Kopta D., Ize T., Spjut J., Brunvand E., Davis A., Kensler A.: Fast, effective BVH updates for animated scenes. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games* (2012), pp. 197–204.

[KNPY13]Kang Y.-S., Nah J.-H., Park W.-C., Yang S.-B.: gkDtree: A group-based parallel update kd-tree for interactive ray tracing. *Journal of Systems Architecture 59*, 3 (2013), 166–175.

[LGS*09] Lauterbach C., Garland, M., Sengupta S., Luebke D., Manocha D.: Fast BVH construction on GPUs. *Computer Graphics Forum 28*, 2 (2009), 375–384.

[LMM09] Lauterbach C., Mo Q., Manocha D., : *Selective Ray Tracing for Interactive High-Quality Shadows*. Tech. Rep. TR09-004, University of North Carolina at Chapel Hill, 2009.

[LYTM06] Lauterbach C., Yoon S.-E., Tuft D., Manocha D.: RT-DEFORM: Interactive ray tracing of dynamic scenes using BVHs. In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006* (2006), pp. 39–45.

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

[NPK*10] Nah J.-H., Park J.-S., Kim J.-W., Park C., Han T.-D.: Ordered depth-first layouts for ray tracing. In *Proceedings of ACM SIGGRAPH ASIA 2010 Sketches* (2010), pp. 55:1–55:2.

[NPP*11]Nah J.-H., Park J.-S., Park C., Kim J.-W., Jung Y.-H., Park W.-C., Han T.-D.: T&I engine: Traversal and intersection engine for hardware accelerated ray tracing. *ACM Transactions on Graphics 30*, 6 (2011), 160:1–160:10.

[SBSK03]Szécsi L., Benedek B., Szirmay-Kalos L.: Accelerating animation through verification of shooting walks. In *SCCG '03: Proceedings of the 19th Spring Conference on Computer Graphics* (2003), ACM, pp. 231–238.

[Smi98]Smits B.: Efficiency issues for ray tracing. *Journal of Graphics Tools 3*, 2 (Feb. 1998), 1–14.

[SSK07]Shevtsov M., Soupikov A., Kapustin A.: Highly parallel fast KD-tree construction for interactive ray tracing of dynamic scenes. *Computer Graphics Forum 26*, 3 (2007), 395–404.

[VHS12]Vinkler M., Havran V., Sochor J.: Visibility driven BVH build up algorithm for ray tracing. *Computers & Graphics 36*, 4 (2012), 283–296.

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

[WBS07] Wald, I., Boulos S. Shirley P.: Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics 26*, 1 (2007), 6:1–6:18.

[WH06] Wald I., Havran V.: On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006* (2006), pp. 61–69.

[WZL11] Wu Z., Zhao F., Liu X.: SAH KD-tree construction on GPU. In *HPG'11: Proceedings of the Conference on High Performance Graphics 2011* (2011), pp. 71–78.

[ZHWG08] Zhou K., Hou Q., Wang R., Guo B.: Real-time KD-tree construction on graphics hardware. *ACM Transactions on Graphics 27*, 5 (2008), 1–11.