IMPROVING RAY TRAVERSAL BY USING SEVERAL SPECIALIZED KD-TREES

Roberto Torres, Pedro J. Martín, Antonio Gavilanes and Luis F. Ayuso
Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Madrid, Spain

Keywords: Ray Tracing, Surface Area Heuristics, KD-tree, GPU, CUDA.

Abstract: In this paper, we present several variants of the Surface Area Heuristics (SAH) to build kd-trees for specific sets of rays' directions. In order to cover the whole space of directions, several sets of directions are considered and each of them leads to a different specialized kd-tree. We call *Multi-kd-tree* to the set of these kd-trees. During rendering, each ray will traverse the kd-tree associated with the set containing its direction. In order to evaluate the efficiency of our proposal, we have implemented a *Path Tracing* and an *Ambient Occlusion* renderer on GPU with CUDA. A SAH-based kd-tree has been compared to a Multi-kd-tree and we show that all the new heuristics exhibit a better performance than SAH over usual scenes.

1 INTRODUCTION

Ray tracing algorithms cover a family of algorithms devoted to the generation of 2D images from a 3D representation of the scene. In these algorithms, rendering is carried out by shooting rays throughout the scene. The final results usually exceed in realism those obtained with the graphics pipeline algorithm. This is the reason why ray tracing is the favourite choice in the generation of photo-realistic images (Pharr and Humphreys, 2010).

A common task of every ray tracer, which is usually the most time-consuming step, is to find the nearest intersection per ray (*traversal* step). In order to accelerate this task, several data structures have been developed to organize the scene. Their advantage is that their traversal algorithms can quickly reject whole regions, avoiding many intersection tests. Examples of these structures are *uniform grids*, *kd-trees*, *octrees* and *bounding volume hierarchies (BVHs)*.

The most efficient hierarchical structures for ray tracing are built with SAH (Goldsmith and Salmon, 1987) using the greedy top-down algorithm by (MacDonald and Booth, 1990), originally presented for kd-trees, and later adapted to BVHs by (Wald, 2007). However, SAH involves assumptions about rays than can be replaced by more realistic ones to build structures with better performance during rendering (Havran and Bittner, 1999; Hunt and Mark, 2008; Fabianowski et al., 2009; Bittner and Havran, 2009).

On the other hand, GPUs are massively-parallel devices that have been used to implement ray tracers, typically binding each thread to a ray during traversal. However, a thread can stall others in the underlying SIMT architecture, mainly due to global memory readings and runtime divergences. This fact has led the design of effective GPU-based ray traversal. The first proposals (Günther et al., 2007; Popov et al., 2007), which were based on *ray packets* as traversal units, were discarded by (Aila and Laine, 2009) because many rays were forced to traverse regions of the scene they did not intersect. Nevertheless, an appropriate arrangement of rays in the device can exploit coalesced readings and cache hits of modern hardware. Therefore, recent trends use data-parallel primitives to rearrange rays in the device either at the beginning (Garanzha and Loop, 2010) or repeatedly during the traversal (Torres et al., 2011). The aim is to get a trade-off between the overload due to the rearrangement of rays and the increase of performance.

The main contribution of this paper is the development of new heuristics from a mathematical formulation of the original SAH. These heuristics specialize SAH for different sets of ray directions by restricting their domain or by assuming non-uniform probabilities. In order to cover the whole space of directions, several sets are used and a kd-tree is built for each of them. The set of these kd-trees is called a *Multi-kd-tree*. We have tested our heuristics using two ray tracing algorithms implemented with CUDA: *Path Tracing* and *Ambient Occlusion*. Before traversing, secondary rays are classified and arranged on the

Torres R., J. Martín P., Gavilanes A. and F. Ayuso L..
IMPROVING RAY TRAVERSAL BY USING SEVERAL SPECIALIZED KD-TREES.
DOI: 10.5220/0003844702150226
In *Proceedings of the International Conference on Computer Graphics Theory and Applications* (GRAPP-2012), pages 215-226
ISBN: 978-989-8565-02-0

device according to the Multi-kd-tree components. In both renderers, Multi-kd-trees exhibit better behaviour than a single SAH-based kd-tree over usual scenes, concerning traversal steps and runtime performance.

2 RELATED WORK

There is an extensive literature about acceleration structures for ray tracing. (Havran, 2000) proved that SAH-based kd-trees were very efficient concerning static scenes on CPU. Thus, subsequent work tried to move these structures from CPU to GPU. (Foley and Sugerman, 2005) presented two techniques to traverse kd-trees without a stack: *kd-tree restart* and *kd-tree backtrack*. Nevertheless, the amount of traversed nodes was greater than the one involved in the classic traversal, due to the fact that many nodes were visited several times. (Horn et al., 2007) improved kd-tree restart by using a small fixed-size stack taking advantage of the new GPU characteristics. In addition, (Popov et al., 2007) implemented a kd-tree traversal without stack on GPU by using ropes and ray packets.

As far as BVHs are concerned, (Thrane et al., 2005) was the first proposal in implementing a BVH on GPU. Afterwards, (Günther et al., 2007) designed a packet-based BVH traversal on CUDA by means of a stack that was implemented on shared memory. (Torres et al., 2009) implemented a stackless traversal on a roped BVH using packets. After that, (Aila and Laine, 2009) proved that a single-ray traversal on BVH is faster than a packet-based one due to the high memory bandwidth of GPUs. (Garanzha and Loop, 2010) developed a faster traversal by sorting the rays and breath-first traversing the BVH.

Regarding SAH, several papers have focused on improving it for specific sets of rays. (Havran and Bittner, 1999) presented several heuristics where probabilities are approximated as ratios of areas by using either orthogonal, perspective or spherical projection. Recently, (Hunt and Mark, 2008) developed a new heuristics adapted to rays in perspective space to build kd-trees by using oblique projections. (Fabianowski et al., 2009) designed a variant of SAH supposing that rays' origins are inside the scene, which is suitable for secondary and shadow rays. (Bittner and Havran, 2009) used a representative ray set to approximate the probability as the ratio of the number of intersected rays.

3 KD-TREE BASED ON SAH

A kd-tree is a binary tree responsible for organizing the objects in the scene. The volume associated with the root is the AABB (*Axis-Aligned Bounding Box*) of the whole scene and each inner node contains a plane aligned with the axes that subdivides this volume into two voxels. Thus, the volume associated with each node is the AABB that results from reducing the root's voxel with its ancestor planes. In addition, each leaf contains a list of triangles overlapping its AABB.

In order to build good kd-trees, it is essential to measure their quality. This is usually formalized by the following recursive cost function (MacDonald and Booth, 1990):

$$\begin{aligned} Cost(l) &= Cost_{tri} \cdot N_{tri}(l) \\ Cost(i) &= Cost_{plane} + P(L|i) \cdot Cost(L) \\ &\quad + P(R|i) \cdot Cost(R) \end{aligned}$$

where l is a leaf node, i is an inner node, L and R respectively denote the left and right children of i, $Cost_{tri}$ is the cost of intersecting a ray with a triangle, $N_{tri}(l)$ is the number of triangles of l, and $Cost_{plane}$ is the cost of intersecting a ray with a plane. $P(A|B)$ is the probability for any ray to intersect the AABB of node A, provided that it already intersects the AABB of node B.

The aim of the construction is to find a kd-tree with minimum cost. However, there are two values in the previous equations that have to be estimated: the probability $P(\cdot|\cdot)$ and the costs related to the children L and R.

With respect to the children's costs, trying to build all possible trees and choosing the one minimizing the cost is unfeasible in general. Therefore, children are assumed to be leaves and so, their costs are quickly computed according to the cost function. In consequence, the construction behaves as a greedy top-down algorithm that looks for the best division of an inner node into two new leaves with the lowest local cost. We follow the $O(NlogN)$ algorithm by (Wald and Havran, 2006) for the kd-tree construction.

The probability $P(A)$ can be evaluated by using geometric probability as a ratio of measures

$$P(A) = \frac{\mu(A)}{\mu(Scene)}$$

where A is the AABB of a node and *Scene* is the AABB of the whole scene —for the sake of clarity, we will identify a node with its AABB along this paper. Notice that, if A and B are AABBs inside *Scene* and $A \subseteq B$, then

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A)}{P(B)} = \frac{\mu(A)}{\mu(B)}$$

In order to specify μ, three facts are usually assumed about rays' directions (Wald and Havran, 2006):

1. All directions are equally likely, i.e. they have constant probability.
2. The origin of each ray is out of the scene.
3. The rays do not get blocked during the traversal, i.e. they finish out of the scene.

Notice that these assumptions consider directions as lines, i.e. directions ω and $-\omega$ result in the same line. So, one half of the vectors on the unit sphere are enough to cover all rays.

A particular measure μ_1 leads to the original SAH formulation as follows. Consider the function $hit_r(A)$ that returns 1 when a ray r hits the AABB A, and 0 otherwise. Under the previous assumptions, $hit_r(A)$ can be estimated as the projected area of A on any plane whose normal is the direction ω of the ray r. Thus, if we consider a set of rays, the measure is the integral over the domain of directions. As explained above, a hemisphere $\mathcal{H}$ on the unit sphere is enough to cover all directions. Mathematically, the measure μ_1 is then expressed as

$$\mu_1(A) = \int_{\omega \in \mathcal{H}} \text{proj\_orth}(A, \omega)\, d\sigma(\omega)$$

where ω is a unit ray direction, $d\sigma$ is the differential solid angle and $\text{proj\_orth}(A, \omega)$ is the area of the orthogonal projection of A on any plane whose normal is ω.

Since we work with AABBs, the latter measure can be evaluated as follows, using the hemisphere with $\omega_Z \geq 0$:

$$\mu_1(A) = \int_{\omega \in \mathcal{H}} \sum_{i \in \{X,Y,Z\}} |N_i \cdot \omega| A_i \, d\sigma(\omega)$$

$$= \int_0^{\frac{\pi}{2}} \int_0^{2\pi} (|\omega_X| \cdot A_X + |\omega_Y| \cdot A_Y + |\omega_Z| \cdot A_Z) \sin\theta \, d\phi d\theta$$

$$= 2\pi(A_X + A_Y + A_Z)$$

where A_X, A_Y and A_Z are the areas of one face of each pair of parallel faces, and $N_X = (1,0,0)$, $N_Y = (0,1,0)$ and $N_Z = (0,0,1)$ are their normals. If $SA(A)$ denotes the surface area of the AABB A, the probability $P(A|B)$ can be computed as

$$P(A|B) = \frac{\mu_1(A)}{\mu_1(B)} = \frac{2\pi(A_X + A_Y + A_Z)}{2\pi(B_X + B_Y + B_Z)} = \frac{SA(A)}{SA(B)}$$

which corresponds to the SAH formulation.

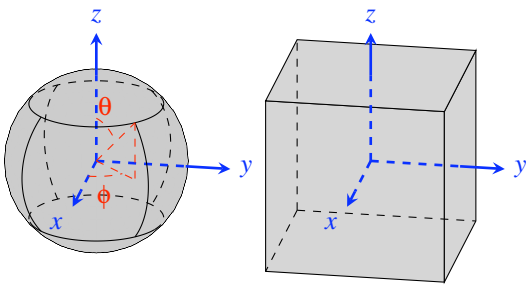


Figure 1: Distribution of the spherical patches (left) and cubic patches (right). For the sake of clarity, the six patches are shown in both figures, however, only three are considered.

4 SPECIALIZED HEURISTICS

The original SAH assumes three facts about rays (Section 3). We will define variants of SAH by changing the original assumptions about rays' directions:

1. Considering different sets of directions rather than the whole hemisphere. This leads to specialized kd-trees that result in better performance for rays whose directions belong to these sets.
2. Considering a non-uniform distribution for rays. Given a direction N, we will suppose that rays are more probable as their directions are closer to N. This results in a kd-tree specialized in the surroundings of N.

In addition, we generalize the way $hit_r(A)$ is estimated using oblique projections. Actually, we will consider orthogonal and oblique projections under the two new assumptions.

4.1 Spherical Heuristics

We relax the assumption that every ray is possible by restricting the directions to a fixed set. Nevertheless, we keep on assuming that the probability of all rays is uniform. Specifically, we split half of the direction space into three pairwise disjoint spherical patches as Figure 1 on the left shows. In that sense, the three spherical patches can be expressed as

$$SP_i = \{(\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta) \mid \theta \in \Theta_i, \phi \in \Phi_i\}$$

where $i \in \{X,Y,Z\}$, and Θ_i and Φ_i are the intervals in Table 1. The value $\theta_0 = acos(\frac{2}{3})$ has been chosen for the patches to have the same area and, therefore, the sets of directions have the same size.

As mentioned, we have two possibilities for choosing the projection. Thereby, SPHERE-ORTH and SPHERE-OBLI will respectively denote the heuristics for the orthogonal and oblique projection.

Table 1: Bounds and normalized weights for spherical and cubic heuristics. The values w_X, w_Y and w_Z are the normalized weights in percentage for the face areas A_X, A_Y and A_Z, respectively.

Patch	**Bounds (spherical coord.)**		**SPHERE-ORTH**			**SPHERE-OBLI**		
	Θ	Φ	w_X	w_Y	w_Z	w_X	w_Y	w_Z
SP_X	$[\theta_0, \pi-\theta_0]$	$[\frac{-\pi}{4}, \frac{\pi}{4}]$	55.04	22.80	22.15	53.47	23.59	22.92
SP_Y	$[\theta_0, \pi-\theta_0]$	$[\frac{\pi}{4}, \frac{3\pi}{4}]$	22.80	55.04	22.15	23.59	53.47	22.92
SP_Z	$[0, \theta_0]$	$[0, 2\pi]$	22.04	22.04	55.90	22.66	22.66	54.67
Patch	**Bounds (cartesian coord.)**			**CUBE-ORTH**			**CUBE-OBLI**	
	x	y	z	w_X	w_Y	w_Z	w_X	w_Y	w_Z
CP_X	$\{1\}$	$[-1,1]$	$[-1,1]$	51.29	24.35	24.35	50.00	25.00	25.00
CP_Y	$[-1,1]$	$\{1\}$	$[-1,1]$	24.35	51.29	24.35	25.00	50.00	25.00
CP_Z	$[-1,1]$	$[-1,1]$	$\{1\}$	24.35	24.35	51.29	25.00	25.00	50.00

In that way, each spherical patch represents a set of directions and leads to one different measure per projection type. The three measures for SPHERE-ORTH are

$$\mu_2^{(i)}(A) = \int_{\omega \in SP_i} \text{proj\_orth}(A, \omega)\, d\sigma(\omega)$$

for the patches SP_i, $i \in \{X, Y, Z\}$. For example, the probability $P(A|B)$ for patch SP_X in SPHERE-ORTH is

$$P(A|B) = \frac{\mu_2^{(X)}(A)}{\mu_2^{(X)}(B)} = \frac{w_X \cdot A_X + w_Y \cdot A_Y + w_Z \cdot A_Z}{w_X \cdot B_X + w_Y \cdot B_Y + w_Z \cdot B_Z}$$
$$= \frac{0.5504 \cdot A_X + 0.2280 \cdot A_Y + 0.2215 \cdot A_Z}{0.5504 \cdot B_X + 0.2280 \cdot B_Y + 0.2215 \cdot B_Z}$$

In general, when the integrals are solved, we obtain a weighted addition of the areas A_X, A_Y and A_Z. After that, we normalize these values by extracting their sum as a common factor. We call these normalized weights w_X, w_Y and w_Z, whose values have been included for the three spherical patches in Table 1. Notice how the area A_X has a bigger weight when considering rays with directions on the spherical patch SP_X. The use of SPHERE-ORTH leads to three different kd-trees, one for each spherical patch, i.e. the measure $\mu_2^{(i)}$ is used during the construction of the kd-tree related to SP_i.

In SPHERE-OBLI, the planes for the oblique projection must be chosen. We have tested the planes YZ for SP_X, XZ for SP_Y and XY for SP_Z. E.g., the measure for SP_Z is

$$\mu_3^{(Z)}(A) = \int_{\omega \in SP_Z} \text{proj\_obli}_{XY}(A, \omega)\, d\sigma(\omega)$$
$$= \int_{\omega \in SP_Z} \left|\frac{\omega_X}{\omega_Z}\right| A_X + \left|\frac{\omega_Y}{\omega_Z}\right| A_Y + A_Z \, d\sigma(\omega)$$

By solving the integrals and normalizing the weights, we obtain

$$P(A|B) = \frac{0.2266 \cdot A_X + 0.2266 \cdot A_Y + 0.5467 \cdot A_Z}{0.2266 \cdot B_X + 0.2266 \cdot B_Y + 0.5467 \cdot B_Z}$$

for SP_Z. See Table 1 for the normalized weights related to SP_X and SP_Y.

4.2 Cubic Heuristics

Other sets of directions can be obtained if they are taken on the surface of a cube. Similar to (Hunt and Mark, 2008), we have chosen the cube $[-1,1]^3$ as Figure 1 shows on the right. As before, directions are considered as lines, so we use three faces on the cube. They are pairwise disjoint and called cubic patches CP_X, CP_Y and CP_Z. We call CUBE-ORTH to the heuristics when the orthogonal projection is used, and CUBE-OBLI if the oblique projection is applied.

The new three measures in CUBE-ORTH are

$$\mu_4^{(i)}(A) = \int_{\omega \in CP_i} \text{proj\_orth}\left(A, \frac{\omega}{|\omega|}\right) dA(\omega)$$

for $i \in \{X, Y, Z\}$. Notice the normalization of the vector ω unlike the spherical heuristics. For example, the measure for CP_Z is

$$\mu_4^{(Z)}(A) = \int_{-1}^{1}\int_{-1}^{1} \text{proj\_orth}\left(A, \frac{(x, y, 1)}{\sqrt{x^2+y^2+1}}\right) dxdy$$

By solving and normalizing, the probability for CP_Z is

$$P(A|B) = \frac{0.2435 \cdot A_X + 0.2435 \cdot A_Y + 0.5129 \cdot A_Z}{0.2435 \cdot B_X + 0.2435 \cdot B_Y + 0.5129 \cdot B_Z}$$

In CUBE-OBLI, the oblique projection is taken into account. Using the same projection planes used for SPHERE-OBLI, we obtain the measure for CP_Z as follows

$$\mu_5^{(Z)}(A) = \int_{\omega \in CP_Z} \text{proj\_obli}_{XY}\left(A, \frac{\omega}{|\omega|}\right) dA(\omega)$$
$$= \int_{-1}^{1}\int_{-1}^{1} |x| \cdot A_X + |y| \cdot A_Y + A_Z \, dxdy$$
$$= 4\int_{0}^{1}\int_{0}^{1} x \cdot A_X + y \cdot A_Y + A_Z \, dxdy$$

Table 2: Normalized weights in percentage for cosine heuristics, taking different values of β. We only present the case for N_X. The other cases can be obtained by suitably swapping columns.

	COS-ORTH			COS-OBLI		
β	w_X	w_Y	w_Z	w_X	w_Y	w_Z
1	43.99	28.00	28.00	33.33	33.33	33.33
2	50.00	25.00	25.00	43.99	28.00	28.00
3	54.08	22.95	22.95	50.00	25.00	25.00
4	57.14	21.42	21.42	54.08	22.95	22.95
5	59.55	20.22	20.22	57.14	21.42	21.42
10	67.01	16.49	16.49	65.90	17.04	17.04

Then

$$P(A|B) = \frac{0.25 \cdot A_X + 0.25 \cdot A_Y + 0.5 \cdot A_Z}{0.25 \cdot B_X + 0.25 \cdot B_Y + 0.5 \cdot B_Z}$$

for CP_Z. Similar expressions can be obtained for CP_X and CP_Y. Table 1 displays the values of the normalized weights for these heuristics.

4.3 Cosine Heuristics

In this heuristics, we assume that all directions are possible but all of them are not equally probable. We will suppose that directions near a given unit direction N are more likely than others. We accomplish it by multiplying the projected area related to a unit direction ω by the factor $(\omega \cdot N)^\beta$, where β is a positive real number. Again, two types of projections can be considered, resulting in two heuristics, COS-ORTH for orthogonal projections and COS-OBLI for oblique projections.

We have tested three values for the direction N, $N_X = (1,0,0)$, $N_Y = (0,1,0)$ and $N_Z = (0,0,1)$. For each of them we have integrated over the hemisphere surrounding N, that is, we have used the hemispheres with $\omega_X \geq 0$, $\omega_Y \geq 0$ and $\omega_Z \geq 0$, denoted as $\mathcal{H}_X$, $\mathcal{H}_Y$ and $\mathcal{H}_Z$, respectively. Each hemisphere leads to a different measure and it produces a specific kd-tree. Notice that domains are not pairwise disjoint for the cosine heuristics.

The measures for COS-ORTH and COS-OBLI are respectively

$$\mu_6^{(i)}(A) = \int_{\omega \in \mathcal{H}_i} (\omega \cdot N_i)^\beta \cdot \text{proj\_orth}(A, \omega)\, d\sigma(\omega)$$

$$\mu_7^{(i)}(A) = \int_{\omega \in \mathcal{H}_i} (\omega \cdot N_i)^\beta \cdot \text{proj\_obli}(A, \omega)\, d\sigma(\omega)$$

for $i = \{X, Y, Z\}$. In Table 2, we present the normalized weights for N_X, taking different values of β. The weights for N_Y and N_Z result from permuting the weights for N_X, since one rotation of $\pi/2$ radians is enough to transform $\mathcal{H}_X$ into $\mathcal{H}_Y$ or $\mathcal{H}_Z$.

5 KD-TREE SELECTION

We apply the $O(NlogN)$ top-down algorithm by (Wald and Havran, 2006) for the kd-tree construction. However, instead of using the surface area to calculate the conditional probability, we apply any of the measures above described. We call kd-tree$_n^{(i)}$ to the kd-tree built with $\mu_n^{(i)}$ (the n-th measure and the set of directions SP_i or CP_i, or the normal N_i). Since, the use of a single kd-tree for the whole scene would benefit some rays but would penalize others, we build three kd-trees (kd-tree$_n^{(X)}$, kd-tree$_n^{(Y)}$ and kd-tree$_n^{(Z)}$) in order to cover the whole direction space. We call *Multi-kd-tree* to the set of these kd-trees.

The process of traversing a Multi-kd-tree by a ray in the spherical and cubic heuristics can be summarized as follows. First of all, each ray selects the kd-tree to traverse. In the case of cubic patches, it is identical to the selection of a face in the *cube mapping* technique. In the case of spherical patches, if $|\omega_Z| \geqslant \cos(\theta_0)$ then the ray chooses kd-tree$_n^{(Z)}$, and otherwise $max(|\omega_X|, |\omega_Y|)$ is used to choose kd-tree$_n^{(X)}$ or kd-tree$_n^{(Y)}$. Once a kd-tree of the Multi-kd-tree is selected by the ray, it is subsequently traversed as usual.

For the cosine heuristics, we use the kd-trees related to normals N_X, N_Y and N_Z. Each ray chooses the kd-tree to traverse by using the selection procedure of the spherical heuristics.

6 IMPLEMENTATION DETAILS

We have implemented a Path Tracing (PT) and an Ambient Occlusion (AO) on CUDA to test the performance of a Multi-kd-tree according to the new heuristics. The scenes used in our tests are BUNNY, FAIRYFOREST, CONFROOM, SPONZA and SIBENIK (Tables 6 and 7). A roof has been added to FAIRYFOREST and a bounding box enclosing BUNNY to prevent the rays from getting away from the scene. The images generated have a resolution of 1024×1024 and every surface is diffuse.

The construction of all kd-trees is made on CPU before rendering. The time spent in the construction of each kd-tree with the new heuristics is almost the same as with SAH.

Before rendering, all the kd-trees needed are allocated together on device memory. In the *node array*, all the nodes of these kd-trees are allocated, and the nodes corresponding to the same kd-tree are contiguous. In the *reference array*, the references to triangles of every leaf are stored. The indices to the root of

Table 3: Number of triangles and memory footprint used by a SAH-based kd-tree and a Multi-kd-tree built with SPHERE-ORTH. *Num.Nodes* is the number of nodes (either inner or leaf) of the kd-trees. *Num.Ref.* is the total number of references to triangles inside the leaves. Each node requires 16 bytes and each reference 4 bytes.

		SAH			SPHERE-ORTH		
Scene	Triangles	Num.Nodes	Num.Ref.	Memory	Num.Nodes	Num.Ref.	Memory
BUNNY	69,475	536,639	343,082	9.49 MB	1,738,331	1,092,768	30.69 MB
F.FOREST	174,119	1,257,457	922,883	22.70 MB	3,983,961	2,901,640	71.85 MB
CONFROOM	282,761	1,570,225	1,433,336	29.42 MB	5,253,325	4,723,711	98.17 MB
SPONZA	67,464	436,899	367,534	8.06 MB	1,339,641	1,141,669	24.79 MB
SIBENIK	80,143	358,779	311,503	6.66 MB	1,100,537	965,394	20.47 MB

each kd-tree are stored on another array, the *header array*. Table 3 shows the number of nodes (either inners or leaves) and the memory footprint used by a SAH-based kd-tree and a Multi-kd-tree built with SPHERE-ORTH. As it can be seen, the used memory of the Multi-kd-tree is about three times the space required by a SAH-based kd-tree. The remaining heuristics exhibit similar memory requirements.

Path Tracing. This renderer considers two levels of recursion: primary rays and secondary rays. It is composed of three kernels: *RayGeneration* (*RG*), *TraversalIntersection* (*TI*) and *Shading* (*SH*). The flowchart of the CUDA kernels can be seen in Figure 2 on the left. Notice that this algorithm is an *implicit* path tracer, i.e. no shadow ray is traced from the intersection points to lights. In order to complete the final image, several *iterations* of the kernels are used, being its number externally controlled.

Kernel *RG* is devoted to generating primary rays from the camera (a pinhole camera). In each iteration, four different random samples per pixel are generated, so the total amount of rays traced in parallel is $4MRays = 4 \times 1024^2$ rays per iteration. In this kernel, each ray chooses the kd-tree to traverse as already described (Section 5).

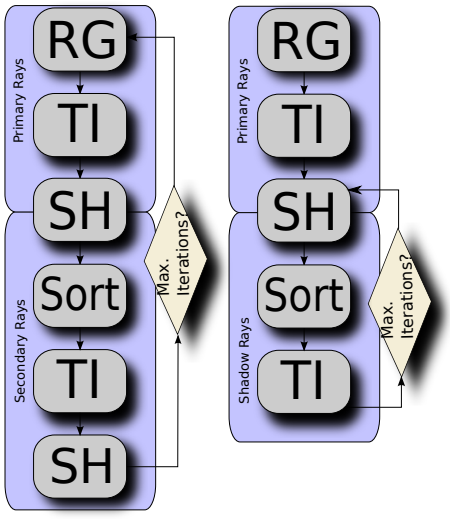


Figure 2: Flowchart of the kernels of Path Tracing (on the left) and Ambient Occlusion (on the right).

Kernel *TI* finds the nearest intersection point for each ray. This kernel is actually the algorithm *persistent while-while* by (Aila and Laine, 2009) adapted to kd-trees. At the beginning of this kernel, the header array is queried by each ray and the root of the kd-tree is retrieved to start traversing.

Kernel *SH* accumulates the color of the rays in the image buffer. If the rays are primary, then this kernel also generates the new secondary ray from each primary ray. These rays are generated on the hemisphere surface according to the cosine probability. In this kernel, and similar to *RG*, the new secondary rays choose the kd-tree to be traversed on the subsequent *TI* launching.

Ambient Occlusion. This renderer also considers two levels of recursion: primary rays and shadow rays. It is also composed of three kernels (Figure 2 on the right), which are very similar to the kernels of PT: *RayGeneration* (*RG*), *TraversalIntersection* (*TI*) and *Shading* (*SH*). In order to complete the final image, multiple iterations of the shadow rays are executed, so primary rays are only traced once at the beginning of the render. In addition, *RG* only generates one sample per pixel, so $1024^2 = 1MRays$ primary rays will be traversed in parallel. In this kernel, identically to PT, each ray selects the kd-tree to traverse.

TI has two configurations. In the first one, the kernel finds the nearest intersection point for each ray, which is suitable for primary rays. In the other, the traversal is finished as soon as an intersection point is found, which is suitable for shadow rays.

SH generates six shadow rays from each intersection point found by kernel *TI*. So $6 \times 1024^2 = 6MRays$ shadow rays will be traversed in parallel in each iteration. Each shadow ray chooses the kd-tree to be traversed, similarly to primary rays.

Ray Arrangement. Primary rays are stored on an array following the Morton code of the image pixels. In this way, contiguous rays are very likely to choose the same kd-tree to traverse. However, secondary rays

Table 4: Traversal Steps on average for Path Tracing and Ambient Occlusion. The number in parenthesis is the gain in percentage w.r.t. SAH. Bold numbers are the maximum of each row.

Path Tracing					
Primary Rays					
Scene	SAH	SPHERE-ORTH	SPHERE-OBLI	CUBE-ORTH	CUBE-OBLI
BUNNY	34.04	**30.27(11.08)**	32.30(5.11)	32.38(4.90)	32.33(5.02)
F.FOREST	48.82	**45.38(7.04)**	45.42(6.96)	45.70(6.38)	45.71(6.38)
CONFROOM	38.46	35.07(8.80)	**34.78(9.56)**	34.89(9.29)	35.01(8.96)
SPONZA	37.66	**34.52(8.33)**	34.74(7.75)	34.67(7.95)	34.97(7.13)
SIBENIK	45.03	39.39(12.51)	39.01(13.35)	**38.43(14.63)**	38.67(14.11)
Secondary Rays					
Scene	SAH	SPHERE-ORTH	SPHERE-OBLI	CUBE-ORTH	CUBE-OBLI
BUNNY	32.09	**29.85(6.99)**	30.88(3.78)	30.76(4.15)	30.75(4.18)
F.FOREST	51.51	**48.71(5.43)**	48.76(5.32)	49.11(4.65)	49.11(4.64)
CONFROOM	39.83	38.79(2.60)	38.83(2.50)	38.81(2.55)	**38.77(2.66)**
SPONZA	41.17	39.60(3.81)	39.53(3.98)	39.51(4.02)	**39.50(4.06)**
SIBENIK	48.01	46.01(4.16)	45.97(4.23)	46.02(4.13)	**45.78(4.63)**
Ambient Occlusion					
Primary Rays					
Scene	SAH	SPHERE-ORTH	SPHERE-OBLI	CUBE-ORTH	CUBE-OBLI
BUNNY	34.23	**30.46(10.99)**	32.50(5.04)	32.57(4.83)	32.53(4.96)
F.FOREST	49.03	**45.60(6.99)**	45.64(6.90)	45.92(6.33)	45.92(6.33)
CONFROOM	38.55	35.17(8.76)	**34.88(9.52)**	34.98(9.26)	35.11(8.93)
SPONZA	37.73	**34.59(8.31)**	34.81(7.73)	34.74(7.93)	35.04(7.12)
SIBENIK	47.31	41.52(12.24)	41.13(13.06)	**40.56(14.26)**	40.80(13.75)
Shadow Rays					
Scene	SAH	SPHERE-ORTH	SPHERE-OBLI	CUBE-ORTH	CUBE-OBLI
BUNNY	28.91	**26.44(8.55)**	28.07(2.90)	28.19(2.50)	28.19(2.49)
F.FOREST	42.58	**40.24(5.49)**	40.29(5.36)	40.62(4.59)	40.65(4.52)
CONFROOM	31.28	30.84(1.41)	30.82(1.46)	30.77(1.63)	**30.73(1.74)**
SPONZA	34.35	33.02(3.86)	**32.96(4.03)**	32.96(4.03)	32.90(4.20)
SIBENIK	39.55	37.93(4.10)	37.89(4.20)	37.94(4.06)	**37.82(4.36)**

are randomly generated over a hemisphere, so contiguous rays are likely to choose different kd-trees. This fact results in texture caches misses even from the beginning of *TI* since the roots of the kd-trees are very far each other. This is experimentally checked as the fact that there is fewer traversal steps w.r.t. SAH but the performance is not higher. In order to solve it, a new kernel *Sort* is added before *TI* for secondary and shadow rays and these rays are rearranged on the array. Specifically, they are sorted w.r.t. the index (to the header array) of its kd-tree. This is done on GPU using the radix sort primitive included in CUDPP 1.1.1 (Harris et al., 2010). Since at most three values are required (either one for the SAH-based kd-tree or three for the Multi-kd-tree), the sorting is carried out on the two least significant bits.

7 RESULTS

Our implementations have been tested on a NVidia GeForce 285 GTX with 1GB of DRAM on the scenes previously mentioned. The constants of the kd-tree construction are $Cost_{plane}$=1 and $Cost_{tri}$=1.

In Tables 4 and 5 we compare a single SAH-based kd-tree to a Multi-kd-tree built with our spherical and cubic heuristics. Only the kernels *TI* are measured, which are the most time-consuming according to our experiments. Specifically, traversal takes around 75%-83% of the whole rendering time. The comparison is given in traversal steps per ray on average (Table 4) and runtime performance (Table 5). A traversal step is either a plane-ray intersection or a triangle-ray intersection. The runtime performance is measured in MRays/s=1024^2 rays per second. Each scene is evaluated by positioning several cameras looking at different locations and executing

Table 5: MRays/s for Path Tracing and Ambient Occlusion when the sorting is included (*inc.*) and not included (*n.inc.*). The number in parenthesis is the gain in percentage w.r.t. SAH. Bold numbers are the maximum of each row. For secondary and shadow rays, only columns with the sorting included (*inc.*) are considered.

Path Tracing									
Primary Rays									
Scene	SAH	SPHERE-ORTH		SPHERE-OBLI		CUBE-ORTH		CUBE-OBLI	
BUNNY	141.12	**147.45(4.29)**		144.10(2.06)		144.44(2.29)		143.68(1.77)	
F.FOREST	101.70	105.92(3.98)		105.78(3.85)		**106.04(4.09)**		105.54(3.64)	
CONFROOM	149.19	156.16(4.46)		**157.52(5.29)**		155.91(4.31)		156.78(4.84)	
SPONZA	171.75	**178.78(3.93)**		177.90(3.45)		178.41(3.73)		177.83(3.42)	
SIBENIK	143.31	155.06(7.57)		156.16(8.22)		156.66(8.52)		**156.83(8.61)**	
Secondary Rays									
Scene	SAH	SPHERE-ORTH		SPHERE-OBLI		CUBE-ORTH		CUBE-OBLI	
		n.inc.	inc.	n.inc.	inc.	n.inc.	inc.	n.inc.	inc.
BUNNY	36.29	37.14	36.12	**38.42**	**37.33**	37.72	36.67	37.33	36.30
		(2.27)	(-0.47)	**(5.53)**	**(2.78)**	(3.77)	(1.02)	(2.76)	(0.01)
F.FOREST	19.36	20.41	20.09	20.62	20.29	20.54	20.22	**20.66**	**20.33**
		(5.11)	(3.61)	(6.08)	(4.58)	(5.73)	(4.23)	**(6.25)**	**(4.75)**
CONFROOM	26.21	27.96	27.37	28.11	27.51	**28.16**	**27.57**	27.79	27.21
		(6.26)	(4.25)	(6.76)	(4.74)	**(6.94)**	**(4.93)**	(5.69)	(3.67)
SPONZA	26.14	28.47	27.83	28.52	27.91	28.45	27.84	**28.73**	**28.11**
		(8.16)	(6.08)	(8.35)	(6.34)	(8.11)	(6.10)	**(9.01)**	**(7.00)**
SIBENIK	19.66	21.72	21.37	21.76	21.40	21.71	21.35	**21.76**	**21.41**
		(9.46)	(7.95)	(9.63)	(8.12)	(9.40)	(7.90)	**(9.64)**	**(8.14)**
Ambient Occlusion									
Primary Rays									
Scene	SAH	SPHERE-ORTH		SPHERE-OBLI		CUBE-ORTH		CUBE-OBLI	
BUNNY	78.72	**81.29(3.16)**		80.36(2.03)		79.59(1.09)		79.71(1.23)	
F.FOREST	63.44	**65.09(2.53)**		65.09(2.53)		64.92(2.28)		64.85(2.18)	
CONFROOM	87.87	94.28(6.79)		**94.45(6.96)**		93.15(5.66)		94.29(6.80)	
SPONZA	112.70	**117.33(3.94)**		116.82(3.52)		117.11(3.76)		116.32(3.11)	
SIBENIK	79.41	83.08(4.40)		84.02(5.48)		83.93(5.38)		**84.55(6.07)**	
Shadow Rays									
Scene	SAH	SPHERE-ORTH		SPHERE-OBLI		CUBE-ORTH		CUBE-OBLI	
		n.inc.	inc.	n.inc.	inc.	n.inc.	inc.	n.inc.	inc.
BUNNY	**46.89**	48.00	46.33	48.28	46.59	47.04	45.44	47.19	45.58
		(2.31)	(-1.19)	(2.87)	(-0.63)	(0.32)	(-3.18)	(0.63)	(-2.87)
F.FOREST	30.80	32.02	31.21	32.26	31.45	**32.27**	**31.46**	32.23	31.43
		(3.79)	(1.30)	(4.51)	(2.07)	**(4.54)**	**(2.10)**	(4.43)	(1.99)
CONFROOM	52.80	54.88	52.57	54.72	52.43	**55.32**	**52.98**	55.11	52.78
		(3.77)	(-0.44)	(3.49)	(-0.72)	**(4.54)**	**(0.32)**	(4.18)	(-0.05)
SPONZA	47.02	50.49	48.65	**50.90**	**49.03**	50.65	48.79	50.87	49.00
		(6.87)	(3.34)	**(7.62)**	**(4.09)**	(7.15)	(3.62)	(7.55)	(4.02)
SIBENIK	37.07	39.73	38.58	39.80	38.64	**39.96**	**38.79**	39.84	38.68
		(6.68)	(3.88)	(6.83)	(4.04)	**(7.21)**	**(4.42)**	(6.95)	(4.15)

several iterations per camera position.

Kernel *Sort* is always launched before *TI* for secondary rays in PT and shadow rays in AO. In Table 5, the left columns of each heuristics (tagged with *n.inc.*) show the performance of kernel *TI*, not including the overload of *Sort*. On the right columns (tagged with *inc.*), the runtime of kernel *Sort* is taken into account and added to the runtime of kernel *TI*. Observe that this sorting does not affect the results in Table 4.

As it can be seen in Table 4, on average, the rays that traverse the Multi-kd-trees take less traversal steps to reach their nearest intersection points. We obtain a gain of up to 14.63% for primary rays and 6.99% for secondary ones in PT, and up to 14.26%

Table 6: Analysis of the COS-ORTH heuristics for Path Tracing w.r.t. traversal steps (middle column) and runtime performance (right column).

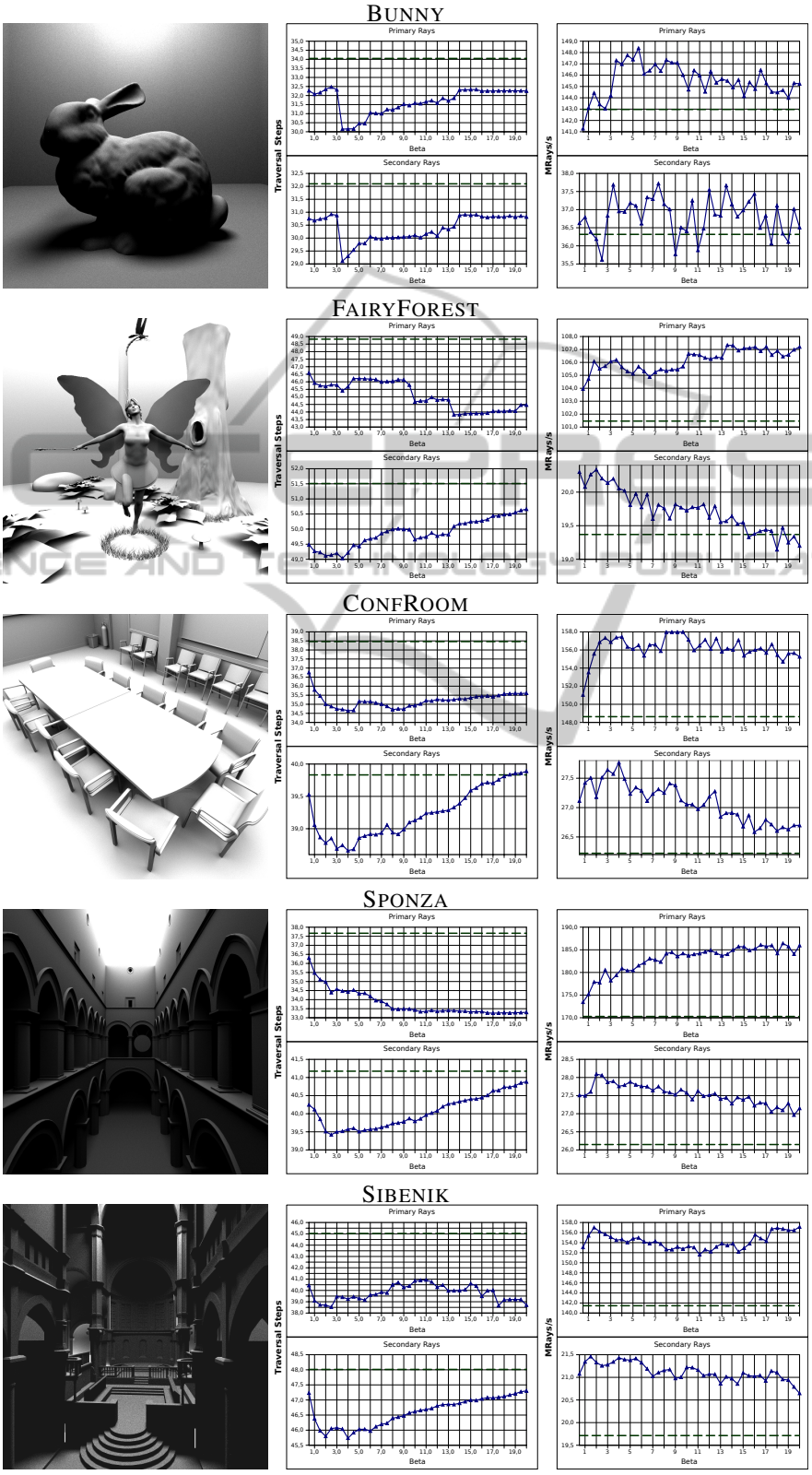
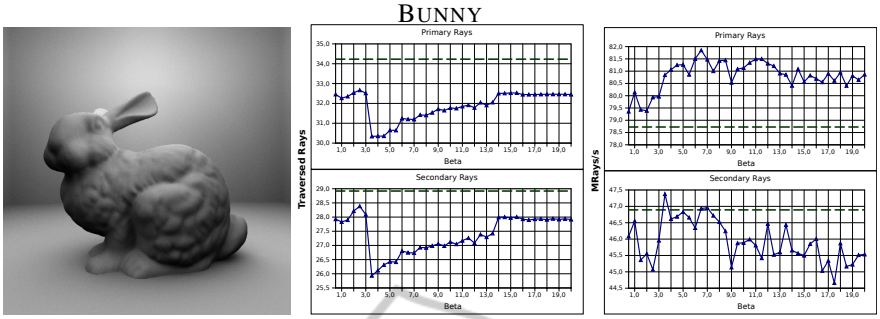
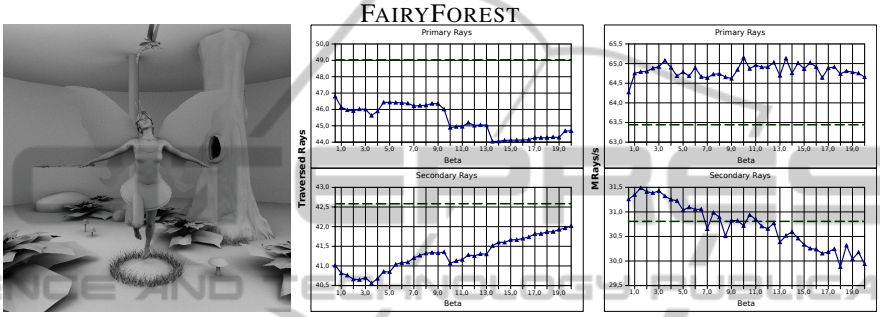


Table 7: Analysis of the COS-ORTH heuristics for Ambient Occlusion w.r.t. traversal steps (middle column) and runtime performance (right column).

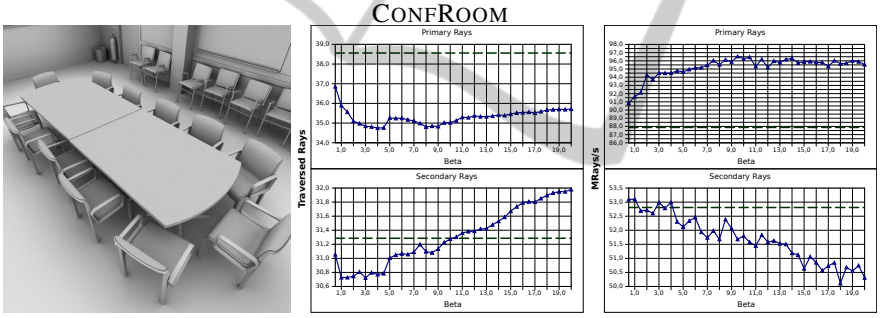
BUNNY



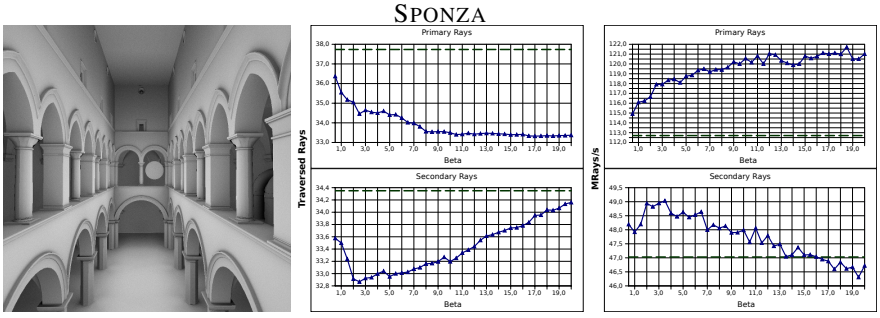
FAIRYFOREST



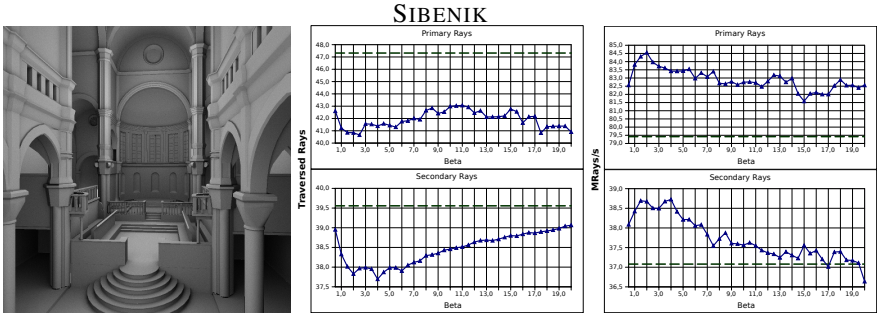
CONFROOM



SPONZA



SIBENIK



for primary and 8.55% for shadow rays in AO. Primary rays in PT and AO are almost identical, so their results are very similar. Shadow rays in AO take fewer traversal steps than secondary rays in PT. This makes sense because the average length of shadow rays in AO is shorter than that of secondary rays in PT.

Concerning the execution model of GPUs, the traversal of different rays is not totally independent from each other. Therefore, texture cache misses and divergences can make the runtime execution different than expected. Even primary rays suffer from these stalls since the decrease in traversal steps do not agree with the improve in performance. For instance, SPHERE-ORTH takes 11.08% less traversal steps than SAH for BUNNY in PT (Table 4), but it only reaches an improvement of 4.29% in performance (Table 5). On the contrary, this clear difference does not hold for secondary rays. It is true that the sorting can entail an increase of their coherence, but secondary rays are randomly spawned and sorting only considers the kd-tree selection. Thus, reports highly depend on how these rays are concretely built during rendering.

Regarding sorting, the performance of our heuristics exceeds that of SAH when the overload due to sorting is not considered (columns *n.inc.* in Table 5). When this overload is included (columns *inc.*) our heuristics keep overcoming in most cases. The overload is more relevant in AO since shadow rays traverse fewer steps on average. Notice that scenes BUNNY and CONFROOM have the lowest average traversal steps and their results show that the overload make their runtime performance mostly slower w.r.t. SAH.

We have also compared SAH with the cosine heuristics. The settings are the same than previous heuristics. We have measured the traversal steps and the runtime performance (including *Sort*) by ranging β from 0.5 to 20 in steps of 0.5. Tables 6 and 7 show the results for COS-ORTH (blue curves) for PT and AO, respectively. The results for COS-OBLI are not depicted because they have a similar behaviour. A dashed horizontal line is added to the charts to compare this heuristics with SAH.

With respect to traversal steps in PT, it can be seen a decrease of them as β increases until it reaches a value between 2 and 3.5, for primary and secondary rays. These values of β lead to similar weights regarding the spherical and cubic heuristics. After that, the behaviour of rays becomes scene-dependant. The charts of runtime performance have an inverse behaviour, since the fewer traversal steps the rays traverse, the higher the runtime performance is.

The charts of traversal steps for primary and shadow rays have a similar shape in AO. Again, the steps traversed by shadow rays are fewer than those for secondary rays in PT due to their shorter length.

Comparing the performance charts between PT and AO, AO exhibits a better performance than PT, but the difference between COS-ORTH and SAH is larger for PT (Table 6) than for AO (Table 7). The explanation of this is the same as previous heuristics, i.e. the constant overload of sorting is more relevant for those rays with fewer traversal steps.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we have presented six new heuristics developed from a mathematical description of the original SAH. These heuristics specialize SAH for different sets of ray directions by restricting their domain or assuming different probabilities. In order to cover the whole space of directions, several sets have been proposed and a kd-tree has been built for each of them (*Multi-kd-tree*). The traversal of a Multi-kd-tree reports fewer traversal steps and better runtime performance than a single SAH-based kd-tree over usual scenes.

However, runtime performance does not agree with the number of traversal steps, due to the execution on SIMT hardware. This fact is even more relevant for secondary or shadow rays due to their random spawning. It is necessary further research about this issue to fill the gap between traversal steps and runtime performance on parallel hardware.

A tighter division of the direction space could be realized. However, two considerations must be taken into account. First, all the information needed for the traversal has to be stored in device memory. So, a bigger amount of divisions entails more memory requirements. Second, the selection of the kd-tree to traverse has to be quick. In this work, only few comparisons are needed, which makes the selection negligible with respect to the whole traversal.

Finally, the cosine heuristics have been developed independently to the spherical and cubic heuristics. It would be interesting to analyze the behaviour of spherical or cubic patches in which rays are distributed according to the cosine heuristics.

ACKNOWLEDGEMENTS

This paper has been supported by the Spanish projects CCG10-UCM/TIC-5476 and GR35/10-A-921547.

Thanks to The Stanford 3D Scanning Repository for the BUNNY model, The Utah 3D Animation Repository for the FAIRYFOREST scene and Marko Dabrovic for the SIBENIK and SPONZA scenes.

REFERENCES

Aila, T. and Laine, S. (2009). Understanding the Efficiency of Ray Traversal on GPUs. In *High-Performance Graphics 2009*, pages 145–149.

Bittner, J. and Havran, V. (2009). RDH: Ray Distribution Heuristics for Construction of Spatial Data Structures. In *SCCG 2009*, pages 61–67, Budmerice, Slovakia.

Fabianowski, B., Flower, C., and Dingliana, J. (2009). A Cost Metric for Scene-Interior Ray Origins. In *Eurographics 2009 Short Papers*, pages 49–52.

Foley, T. and Sugerman, J. (2005). KD-Tree Acceleration Structures for a GPU Raytracer. In *Graphics Hardware 2005*, pages 15–22.

Garanzha, K. and Loop, C. (2010). Fast Ray Sorting and Breadth-First Packet Traversal for GPU Ray Tracing. In *Eurographics 2010*.

Goldsmith, J. and Salmon, J. (1987). Automatic Creation of Object Hierarchies for Ray Tracing. *IEEE Computer Graphics and Application*, 7(5):14–20.

Günther, J., Popov, S., Seidel, H.-P., and Slusallek, P. (2007). Realtime Ray Tracing on GPU with BVH-based Packet Traversal. In *Eurographics Symposium on Interactive Ray Tracing 2007*, pages 113–118.

Harris, M., Owens, J. D., Sengupta, S., Tseng, S., Zhang, Y., Davidson, A., and Satish, N. (2010). CUDA Data Parallel Primitives Library (CUDPP 1.1.1). http://code.google.com/p/cudpp/.

Havran, V. (2000). *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Faculty of Electrical Engineering, Czech Technical University in Prague.

Havran, V. and Bittner, J. (1999). Rectilinear Trees for Preferred Ray Sets. In *SCCG 1999*, pages 171–178, Budmerice, Slovakia.

Horn, D. R., Sugerman, J., Mike, H., and Hanrahan, P. (2007). Interactive KD-Tree GPU Raytracing. In *I3D 2007*, pages 167–174.

Hunt, W. and Mark, W. R. (2008). Adaptive Acceleration Structures in Perspective Space. In *IEEE Symposium on Interactive Ray Tracing*, pages 11–17.

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

Pharr, M. and Humphreys, G. (2010). *Physically Based Rendering: From Theory to Implementation (second edition)*. Morgan Kaufmann.

Popov, S., Günther, J., Seidel, H.-P., and Slusallek, P. (2007). Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum (Proceedings of Eurographics)*, 26(3):415–424.

Thrane, N., Simonsen, L. O., and Orbaek, A. P. (2005). A Comparison of Acceleration Structures for GPU Assisted Ray Tracing. Technical report, University of Aarhus.

Torres, R., Martin, P. J., and Gavilanes, A. (2011). Traversing a BVH Cut to Exploit Ray Coherence. In *GRAPP 2011*, pages 140–150.

Torres, R., Martín, P. J., and Gavilanes, A. (2009). Ray casting using a roped BVH with CUDA. In *Proc. Spring Conference on Computer Graphics*, pages 107 – 114.

Wald, I. (2007). On Fast Construction of SAH-Based Bounding Volume Hierarchies. In *Symposium on Interactive Ray Tracing 2007*, pages 33–40.

Wald, I. and Havran, V. (2006). On Building Fast KD-Trees for Ray Tracing, and on Doing That in O(NlogN). In *Symposium on Interactive Ray Tracing*, pages 61–69.