Construction of Efficient Kd-Trees for Static Scenes Using Voxel-visibility Heuristic

Byeongjun Choi, Byungjoon Chang, Insung Ihm*

Department of Computer Science and Engineering, Sogang University
35 Baekbeom-ro, Mapo-gu, Seoul 121-742, Korea

Abstract

In the ray-tracing community, the surface-area heuristic (SAH) is used as a de facto standard strategy for building high-quality kd-trees. Although widely accepted as the best kd-tree construction method, it is based only on the surface-area measure, which often fails to reflect effectively the rendering characteristics of a given scene. This paper presents new cost metrics that help produce improved kd-trees for static scenes by considering the visibility of geometric objects, which can affect significantly the actual distribution of rays during ray tracing. Instead of the SAH, we apply a different heuristic based on the new concept of voxel visibility, which allows more sophisticated estimation of the chance of a voxel being hit by rays. The first cost metric we present aims at constructing a single kd-tree that is used to trace both primary and secondary rays, whereas the second one is more relevant to secondary rays, involving reflection/refraction or shadowing, whose distribution properties differ from those for primary rays. Our experiments, using both CPU-based and GPU-based computation with several test scenes, demonstrate that the presented cost metrics can reduce markedly the cost of ray-traversal computation and increase significantly the overall frame rate for ray tracing.

Keywords: Ray Tracing, kd-tree construction, cost metric, surface area heuristic, voxel visibility

1. Introduction

1.1. Background

Among the various spatial data structures, the kd-tree is the most popular for static-scene ray tracing, thanks to its reliably higher acceleration performance. Although several attempts have been made to build a good kd-tree, the best technique currently known is the surface-area heuristic (SAH), which was introduced by MacDonald and Booth [1]. This is a typical greedy algorithm that constructs the kd-tree in a top-down manner by recursively subdividing a given bounding volume into two subvolumes. The key to the SAH is in its use of a cost-prediction function that, via minimization, directs where to position the splitting planes for effective space subdivision.

The traditional SAH is based on the theory of geometric probability [2], in which, simply assuming that rays are uniformly distributed in space, the probability that a random ray intersects with a given voxel V, i.e., an axis-aligned bounding box, is proportional to the voxel's surface area $SA(V)$. Given two parameters C_T and C_I that estimate the cost of a node-traversal step and a ray–triangle intersection step, respectively, the cost of partitioning V into two subvoxels V_L and V_R, using a splitting plane P, is modeled as

$$C_{sa}(V,P) = C_T + C_I\left(\frac{SA(V_L)}{SA(V)}|T_L| + \frac{SA(V_R)}{SA(V)}|T_R|\right), \quad (1)$$

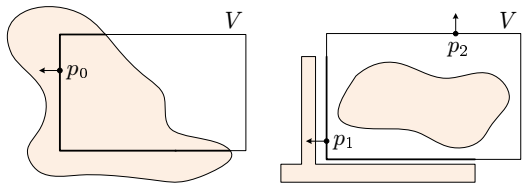
where T_L and T_R each represents the set of triangles that overlap V_L and V_R. The plane that minimizes this cost function is then regarded as the best split candidate (also refer to [3] for more details).

*Corresponding author (Tel: +82-2-705-8493, Fax: +82-2-704-8273)

Email addresses: oipini@sogang.ac.kr (Byeongjun Choi), jerrun@sogang.ac.kr (Byungjoon Chang), ihm@sogang.ac.kr (Insung Ihm)

Although used as a de facto standard method for constructing high-quality kd-trees, the SAH makes some assumptions that may not always hold for the ray-tracing computation. For example, the SAH assumes that rays are uniformly distributed in space, coming only from outside the given voxel, and are not blocked by objects during traversal. In practice, however, the ray distribution is often quite nonuniform in space, being influenced by the geometry and rendering parameters of a given scene.

One of the most influential factors that determine the actual ray distribution is the scene geometry. Figure 1(a) illustrates a situation where much of a voxel's boundary exists inside an opaque object. Here, the hidden surface area, marked by the thicker line, may not receive rays, leading to a probability much less than that implied by its surface area. Figure 1(b) shows an example of the incident ray being dependent on the neighboring geometry. Here, the surface area of the voxel, marked by the thicker line, is relatively less likely to be intersected by incoming rays than is the remaining open area. This is because rays will enter the voxel through the thicker-lined region mainly via reflection from the wall and floor, which are mostly occluded by the nearby object. If the reflector is diffusive, the probability would be much less in classic ray tracing in which diffusive reflection is often ignored.



(a) Hidden surface area (b) Occluded surface area

Figure 1: Two examples violating the assumptions made in the traditional SAH. (a) Unlike open boundary points, a hidden point p_0 may not attract an incoming ray. (b) Under the assumption that rays are uniformly distributed outside the voxel V, the probability of rays entering through p_2 will be higher than for p_1, because p_2 is more visible from outside.

In addition to geometric factors, the distribution of rays is affected by the properties of other rendering elements like lighting, material, and camera. For example, the surface reflection property determines how secondary rays are generated at a surface point when hit by a ray. However, such a ray distribution in a complex scene is often difficult to predict precisely; rather, it is necessary to rely on the simple assumptions made in the traditional SAH.

1.2. Our contribution

Aiming at accelerating ray-tracing computations via an improved kd-tree, we present two cost metrics that can help choose a better split plane during kd-tree construction. In building a kd-tree, our method attempts to exploit a measure of geometric visibility inherent in a given scene, which affects significantly the actual distribution of rays generated during ray tracing. For this, we introduce the new concept of *voxel visibility*, which enables better estimation of the chance of a voxel being hit by rays. The first cost metric aims at building a single kd-tree that is used to effectively trace both primary and secondary rays. The second cost metric is more relevant to secondary rays, involving reflection/refraction or shadowing, whose distribution properties differ from those for primary rays. By introducing an additional kd-tree dedicated to tracing the secondary rays, we find that ray-tracing performance can be improved further.

Because our method requires a nontrivial precomputation of visibility information in 3D space, it is currently limited to static scenes. However, while dynamic scenes are widely used nowadays, building efficient kd-trees for static scenes is still important especially when large ones are visualized. It may also be useful for dynamic scenes where only a small portion of triangles need to be updated. For instance, in a scene of a couple of persons moving around in a big hall, using two trees as suggested in previous work [4], say, an optimized kd-tree for the large static geometry, allowing an overall fast ray tracing and another hierarchy for the small dynamic geometry, permitting a fast update, could possibly be a more promising choice than rebuilding the entire large scene per every frame in spite of the recent successful GPU-based techniques for the complete hierarchy construction [5, 6, 7] (see Figure 2).

From experiments with several example scenes and by rendering via both CPU-based and GPU-based computation, we demonstrate that the presented kd-tree construction techniques produce a noticeable reduction in the cost of kd-tree traversal and ray–object intersection computations, leading to a significant increase in the overall frame rate for ray tracing.

Figure 2: Hybrid acceleration hierarchies. In this example scene, a man is chasing another man (78,028 triangles per man) in a cafe (1,735,079 triangles). When the entire geometry was built into a standard SAH-based kd-tree, it took 110.6 ms to ray-trace this frame at resolution 1024×1024 on an NVIDIA GeForce GTX 480 GPU. When we maintained two hierarchies, that is, a precomputed voxel-visibility based kd-tree for the static cafe and a dynamically updated bounding volume hierarchy (BVH) for the two men, and traced them respectively for rendering, it took 98.3 ms including the BVH update time. When 9 different camera views were tested while they were running, the two-tree scheme was faster by 7.1% on average. Considering that the above timing of the one-tree scheme does not contain its hierarchy rebuilding time for the entire geometry, the two-tree scheme may be a promising choice in such a situation.

2. Related work

As widely agreed, the kd-tree is known to be a reliably optimal acceleration structure for ray-tracing a static geometry, particularly when combined with such techniques as frustum traversal [8] and coherent packet tracing [9]. Although several strategies are possible, the SAH method is generally acknowledged to produce the best kd-trees [3, 10]. As introduced by MacDonald and Booth [1], this strategy chooses the splitting plane by minimizing a cost function based on the surface area. To improve its ray-tracing performance further, Havran investigated various factors that influence the performance of a hierarchical spatial-subdivision structure and suggested an enhanced cost metric [3]. Wald and Havran introduced an $O(n \log n)$ algorithm that builds robust kd-trees using the SAH [11]. Hunt et al. [12] and Popov et al. [13] presented a faster, scanning-based algorithm that approximates the SAH cost function but causes only a little performance degrade in terms of ray tracing time.

Alternatively, there have been other attempts to improve the performance of kd-trees. Adjusting the cost metric in favor of split planes that create some empty cells has been shown to be effective by many practitioners [3, 14]. Hunt showed that correcting the surface-area metric in terms of mailboxing improved the rendering performance by reducing the rate of intersection test [15]. As mentioned in the Introduction, the traditional SAH often suffers from (somewhat unrealistic) assumptions, which may lead to incorrect probabilities for a voxel being hit by rays. To ease this problem, Reinhard et al. [16] and Havran [3] introduced a blocking-factor concept that aims at measuring the degree to which rays are occluded by objects inside a given voxel. Fabianowski et al. modified the SAH cost metric to take into account rays originating inside voxels [17]. Interestingly, there was also an attempt to find a better cost metric that allows a more efficient kd-tree for organizing point datasets for photon mapping [18]. Recently, Ize and Hansen derived a cost metric that determines which child node a ray should traverse first for efficient occlusion-ray traversal [19].

Our approach is unique in that we exploit the notion of visibility for improving the quality of kd-trees. Computation of visibility is one of the most fundamental issues in 3D graphics and has often been an essential part in developing efficient graphics algorithms. In particular, there have been several attempts to mathematically define the visibility measure suitable for solving specific problems including, for instance, mesh simplification [20], path optimization [21], illumination computation [22], real-time walkthrough/shadow-ray acceleration [23], and so on. (Also see [24] for a general introduction.)

3. New cost metric for kd-tree construction

3.1. Outgoing and incident ray densities

To derive a new cost metric for kd-tree construction, we first define a density function $\delta(x, \omega) = \frac{d^2R}{d\omega dV}$, called the *outgoing ray density*, which specifies the amount of rays per unit solid angle per unit volume that originate from a point x in the direction ω. This function aims at describing the nonuniform distribution of rays, whether primary or secondary, generated in the scene domain during ray tracing.

Symbol	Quantity
$\delta(x,\omega)$	outgoing ray density
$\rho(x,n_x)$	incident ray density
$\Upsilon_{ext}(V)$	the amount of rays entering V from outside
$\Upsilon_{int}(V)$	the amount of rays originating from inside V
$\Upsilon(V)$	voxel visibility
$\rho^{sec}(x,n_x)$	surface visibility
$\Upsilon^{sec}_{ext}(V)$	
$\Upsilon^{sec}_{int}(V)$	surface-visibility based quantities
$\Upsilon^{sec}(V)$	
$C_{sa}(V,P)$	surface-area based cost function
$C_{vv}(V,P)$	voxel-visibility based cost function

Table 1: Symbols and their meaning.

Next, we define an *incident ray density*, $\rho(x,n_x) = \frac{dR}{dA}$, which expresses the differential number of rays that arrive at a differential area around a surface point x with normal n_x. It includes all incoming rays that originate from an arbitrary point in the visible region of the half space specified by x and n_x and arrive at x (see Figure 3).

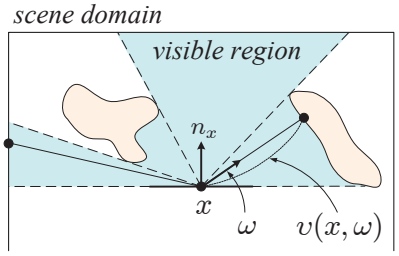


Figure 3: Space of possible ray origins with respect to a surface point x with normal n_x. Here, $\upsilon(x,\omega)$ denotes the distance to the first hit by the ray shot from x in the direction ω. This quantity, combined with the outgoing ray density, indicates the degree to which x is visible from possible ray origins along the sample direction. If there is no intersection, it is set to the distance to the boundary of the axis-aligned bounding volume of scene, which defines the space where rays may start.

By the definition of the incident ray density, it becomes that

$$\rho(x,n_x) \quad = \quad \int_\Omega R(x,\omega)\cos\theta\, d\omega,$$

where $R(x,w) = \frac{d^2R}{d\omega dA}$ denotes the amount of rays per unit solid angle per unit area arriving from the direction ω, and $\cos\theta$ is the angle between n_x and ω. Although easy to understand intuitively, precise computation of this measure is not easy to achieve, in general. In practice, we require a numerical approximation method for the density function.

Consider the hemisphere Ω specified by x and n_x. If m directions are sampled uniformly over the hemisphere (stratified sampling is used in our implementation), we obtain an approximation, which is expressed in terms of the ith sample direction ω_i:

$$\rho(x,n_x) \quad \approx \quad \sum_{i=0}^{m-1} R(x,\omega_i)\cos\theta_i \frac{2\pi}{m}. \tag{2}$$

Note that $R(x,\omega_i) = \int_0^{\upsilon(x,\omega_i)} \delta(x+t\cdot\omega_i, -\omega_i)\, dt$, where $\upsilon(x,\omega_i)$ is the visibility term described in the caption of Figure 3. If we make another approximation by subdividing the line segment $[0,\upsilon(x,\omega_i)]$ into intervals of length Δh, we get to the following formula

$$R(x,\omega_i) \quad \approx \quad \sum_{j=1}^{m_i} \delta(x_{ij},-\omega_i)\Delta h, \tag{3}$$

where $m_i = \lceil \upsilon(x,\omega_i)/\Delta h \rceil$ and $x_{ij} = x + (j\Delta h)\omega_i$. Then, by combining Eq. (2) and (3), we obtain the following approximation to the incident ray density:

$$\rho(x,n_x) \approx \frac{2\pi\Delta h}{m} \sum_{i=0}^{m-1} \left(\sum_{j=1}^{m_i} \delta(x_{ij},-\omega_i) \right) \cos\theta_i. \tag{4}$$

Interestingly, this general notion of incident ray density includes the notion of obscurance [25] as a special case. (Recall that the popularly used ambient occlusion is itself a special case of the obscurance.) That is, in the particular case in which the outgoing ray density is constant, say $\delta(x,\omega) = 1$, throughout the entire open space, Eq. (4) is simplified further as follows:

$$\rho(x,n_x) \approx \frac{2\pi}{m} \sum_{i=0}^{m-1} \upsilon(x,\omega_i)\cos\theta_i. \tag{5}$$

This formula, as a special case of the obscurance, is intuitively straightforward in that, only when the rays are uniformly generated in empty space, the number of rays hitting a differential area depends mainly on the openness of the space above the area.

3.2. External rays entering voxel V from outside

Having an approximation to the incident ray density at a point, the total number of rays arriving at a given (planar) surface area A with surface normal n_x can be estimated by calculating the integral

$\int_A \rho(x, n_x)\, dA$. Consider a voxel V in the scene domain. The number of *external rays* that enter V from outside is naturally expressed as

$$\Upsilon_{ext}(V) \;=\; \sum_{i=1}^{6} \int_{\partial V_i} \rho(x, n_x)\, dA, \qquad (6)$$

where ∂V_i, $i = 1, 2, \cdots, 6$, denote the respective rectangular faces of V.

Although this is a clear definition, it is impractical to evaluate it precisely, on the fly, for each voxel tested during kd-tree construction. To estimate the six component integrals more rapidly, we partition the scene domain into a rectangular grid, where the incident ray density is estimated at the center p_{ijk} of each grid cell c_{ijk} using the numerical formula of Eq. (4) or Eq. (5). In particular, each cell stores the six density values of $\rho(p_{ijk}, n)$ for $n \in \{x^+, x^-, y^+, y^-, z^+, z^-\}$, where x^+ and x^- represent the principal x-axis directions $(1, 0, 0)$ and $(-1, 0, 0)$, respectively, and the other vectors are defined similarly for the remaining two axes.

After the density map is prepared, $\Upsilon_{ext}(V)$ is approximated by summing the area-weighted incident ray density for all overlapped cells as follows:

$$\Upsilon_{ext}(V) \approx \sum_{c_{ijk} \cap\, \partial V \neq \emptyset} \rho(p_{ijk}, n_{ijk}) \cdot \Delta p_{ijk}.$$

Here, Δp_{ijk} denotes the area of the rectangle formed by the intersection of the voxel's boundary ∂V with cell c_{ijk}, and the direction n_{ijk} is inherited from the corresponding boundary face.

3.3. Internal rays originating within voxel V

In addition to the external rays, the rays that originate from inside the voxel V also cause to visit the kd-tree node corresponding to V. The amount of these *internal rays* can be expressed naturally in terms of a volume integral

$$\Upsilon_{int}(V) \;=\; \int_{V \cap \neg O} \int_{4\pi} \delta(x, \omega)\, d\omega dV, \qquad (7)$$

where O indicates the hidden space occupied by closed objects in the scene. If we assume a uniform density $\delta(x, \omega) = 1$ over the empty space, it simplifies to $\Upsilon_{int}(V) = 4\pi\mathcal{E}(V)$, where $\mathcal{E}(V)$ represents the volume of the empty region of V.

For efficient on-the-fly approximation to the integral during kd-tree construction, we perform additional preprocessing, whereby the scene's bounding volume is subdivided into a rectangular grid of small cells. To find cells in the open space, we apply a region growing technique that marks them by iteratively visiting neighboring cells in the empty region starting from a seed cell. Then, for each of the marked cells, the (estimated) total number of rays originating inside the cell is recorded. (For example, if we assume a uniform density, the stored value is the cell's volume.) Then, for voxel V, $\Upsilon_{int}(V)$ can be estimated rapidly by simple addition of these values for the cells inside V.

3.4. Cost metric based on voxel visibility

We can now obtain the total amount of rays that cause to traverse the corresponding node in the kd-tree by adding the two integrals

$$\Upsilon(V) \;=\; \Upsilon_{ext}(V) + \Upsilon_{int}(V). \qquad (8)$$

Informally, $\Upsilon(V)$ implies the visibility of voxel V, in that it indicates how visible the voxel is to rays possibly flowing in the scene. Therefore, in this work, we call the quantity $\Upsilon(V)$ the *voxel visibility*.

Based on this measure, we have a new cost-estimation function, which is similar to the surface-area metric in Eq. (1), but which considers the nonuniform ray distribution when building a more effective kd-tree:

$$C_{vv}(V, P) = C_T + C_I\left(\frac{\Upsilon(V_L)}{\Upsilon(V)}\,|T_L| + \frac{\Upsilon(V_R)}{\Upsilon(V)}\,|T_R|\right). \qquad (9)$$

As will be demonstrated shortly, the new metric tends to provide a more sophisticated estimate for the chance of node traversal than that of the SAH by considering the actual geometry of the scene. Note also that our visibility measure ameliorates the problem arising from the unrealistic assumption in the original SAH that no ray is hindered by geometric objects during its traversal. Unlike previous work, e.g., [3, 16], in which an explicit blocking-factor term is used in the cost function, our method implicitly incorporates the occlusion of rays by objects in the cost metric.

3.5. Adapting the visibility heuristic for secondary rays

Although the cost function $\Upsilon(V)$ described above is designed to handle all types of rays, whether primary or secondary (i.e., reflection/refraction and shadow rays), the idea of voxel visibility can be extended further for secondary rays. In contrast

to primary rays, secondary rays always originate from the surfaces of objects. Based on the heuristic that more secondary rays tend to be generated from more *visible* surface regions, we adjust the incident ray density function to take account of *surface visibility*.

Let y_i and n_{y_i} denote the position and normal direction, respectively, at the closest surface point visible along a ray originating from a given point x and pointing in the direction ω_i (see Figure 4). We take the incident ray density $\rho(y_i, n_{y_i})$ at y_i as a measure of how densely the secondary rays originate from y_i. Considering the angle ϕ_i between n_{y_i} and $-\omega_i$, the number of differential rays from the direction ω_i through the cone of solid angle $\frac{2\pi}{m}$ may be approximated as $\frac{2\pi}{m}\rho(y_i, n_{y_i})\cos\phi_i\cos\theta_i$. The total number of secondary rays can then be computed by summing these quantities over all directions as follows (compare with the approximation in Eq. (5)):

$$\rho^{sec}(x, n_x) \;=\; \frac{2\pi}{m}\sum_{i=0}^{m-1}\rho(y_i, n_{y_i})\cos\phi_i\cos\theta_i. \quad (10)$$

Using this modified density value, the external visibility of voxel V can be expressed as before:

$$\Upsilon^{sec}_{ext}(V) = \sum_{i=1}^{6}\int_{\partial V_i}\rho^{sec}(x, n_x)\,dA. \quad (11)$$

For a practical estimation of $\rho^{sec}(x, n_x)$, we subdivide triangles in the scene into smaller subtriangles and compute the incident ray densities at their centers in the preprocessing stage. Then, when the hemisphere around x is sampled, a nearest-neighbor filter is applied to obtain $\rho(y_i, n_{y_i})$, for which the density value of the subtriangle containing y_i is taken into account.

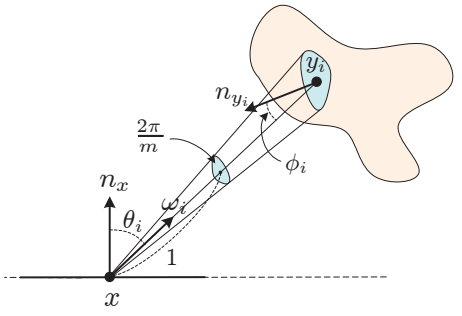


Figure 4: Incident ray density based on surface visibility.

Similarly, the internal visibility, which indicates the number of secondary rays originating inside V, can be obtained by a discrete sum of the incident ray density over surfaces contained in V:

$$\begin{aligned}\Upsilon^{sec}_{int}(V) &= \int_{\partial O\cap V}\rho(x, n_x)\,dA \\ &\approx \sum_{\forall\text{ subtri. } i \text{ in } V}\rho(x_i, n_{x_i})\cdot\Delta x_i,\end{aligned} \quad (12)$$

where ∂O is the surface of geometric objects, x_i is the center of the ith subtriangle, and Δx_i is its area. Then, using the surface-visibility-based cost metric

$$\Upsilon^{sec}(V) \;=\; \Upsilon^{sec}_{ext}(V) + \Upsilon^{sec}_{int}(V), \quad (13)$$

we can build another kd-tree, dedicated to tracing secondary rays.

3.6. Some implementation details

Building a hybrid kd-tree As explained before, our method estimates the incident ray density values at given grid cells, and then use them for the kd-tree construction. When the grid's resolution is high enough compared to the size of the given voxel, this approach usually provides a good estimate for the visibility metric. However, with a fixed grid resolution, it becomes less accurate as the relative size of a node's bounding box decreases. In our scheme, rather than relying on the possibly error-prone voxel-visibility metric computed during kd-tree construction, we apply the original SAH if the ratio of the grid-cell size to the voxel size is above a preset threshold value as denoted as C_L in Section 4. As a result, we actually obtain a hybrid kd-tree for which both voxel-visibility and surface-area heuristics are employed, the former for the upper levels and the latter for the lower levels of the kd-tree. This hybrid approach is a compromise between the computational cost and the accuracy of the estimation.

Placing an empty box at the top of the tree Another simple modification we have made for efficient kd-tree construction is to place an empty space at the top of the kd-tree, if it is appropriate. In [3], Havran noted that cutting off empty spaces is highly effective for the upper levels of the tree. In addition to the frequently used technique that tries to maximize empty spaces during the evaluation of cost metric, we attempt to find a large empty axis-aligned box inside a given scene in the preprocessing step and utilize it for building a better kd-tree.

Since finding the 'best' box is an intractable problem, we take a simple heuristic approach, based on the region growing algorithm described in Subsection 3.3. Starting from each empty cell found, an axis-aligned empty box is built by extending it into the positive x, y, and z directions as much as possible. Then the one with the largest volume is selected as the candidate. During the kd-tree construction, the six planes containing the faces of the empty box are first selected as splitting planes to set up the initial tree. Then our voxel-visibility heuristic is applied to each nonempty node to complete the tree-building computation.

As will be discussed later, this heuristic gives a meaningful performance increase in frame rate above that achieved without placing the empty box only if there exists a 'good' empty box. In addition, it must be noted that locating an empty box in the first stage of tree building may enable time and space savings in building the discretized incident ray density map because there is no need to calculate the density values for those cells residing wholly inside the empty box.

4. Experimental results

To show the effectiveness of our method, we have built kd-trees using the presented cost metrics, and compared their timing performances with those of corresponding SAH-based kd-trees. Table 2 shows statistics from the tested example scenes for various numbers of triangles, reflection/refraction bounces, and lights. As it is well understood, the ray-tracing performance depends noticeably on the parameter values for the applied rendering. To analyze the new cost metrics accurately, we started with a wide range of parameter values and then selected four or five representative parameter sets per scene (Figure 5 shows the selected camera views). In our experiments, we measured the execution times for both CPU- and GPU-based computation, using a 2.67 GHz Intel Core i7 CPU and an NVIDIA GeForce GTX 480 GPU.

Construction of the kd-trees All the tested kd-trees were built using a similar construction algorithm to that employed in the standard SAH method [11] except for the use of different cost metrics. Given $C_T = 1$, we used the relative intersection cost $C_I = 2$ (for both the CPU and the GPU) for the surface-area heuristic and $C_I = 0.7$ (CPU) and $C_I = 1$ (GPU) for the voxel-visibility heuristic. For our experiments, these were the values that achieved the best results. Note that the ratio of C_I to C_T influences the tree-construction process by affecting the termination of voxel subdivision. Because the metrics used in the SAH and in our voxel-visibility heuristic are different in nature, the ratios that give the best performance may be different for each heuristic.

Scene	# of triangles	Reflection bounces	# of lights
KITCHEN_A	36,002	2	1
KITCHEN_B	101,015	2	4
CONFERENCE	190,947	2	1
SPONZA	255,856	3	1
FAIRY	174,117	3	3

Table 2: Test scene statistics.

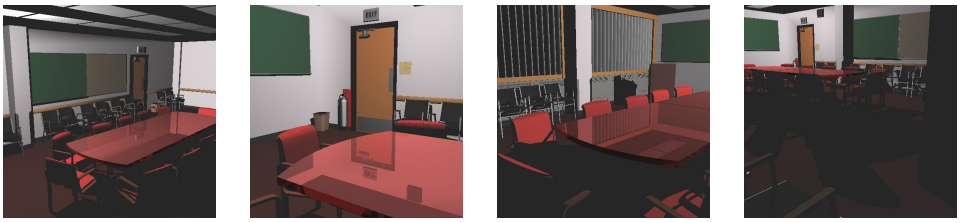
To provide the best ray-tracing speed for each scene, we chose, manually for each scene, an empty bonus multiplier that usually ranged between 0.5 and 0.9. In addition, the empty-box heuristic was applied during new tree construction unless said otherwise. Also, the switch was made from the voxel-visibility to surface-area heuristic when the ratio between the length of the smallest axis of the voxel and that of the cell became smaller than a given threshold C_L, whose value usually stayed around 15.

For each test scene, its domain was discretized using a rectangular grid of resolution $256 \times 256 \times 256$. To avoid the redundant computation, we first sampled 4,096 directions uniformly around each cell's center using stratified sampling, and then used this sampled data to compute the incident ray density for each of the six directions (this amounted to 2,048 direction samples over the hemisphere per density value). In addition, the surface visibility was evaluated at 1,000,000 sample points over the geometry.

Most of the preprocessing time was spent on building the incident ray density fields. The KITCHEN_B scene, for example, required 202.6 seconds to evaluate the density values (Eq. (5)) using the GPU, whereas only 7.21 seconds were necessary for the CPU to construct the hybrid kd-tree using the density field. When a second kd-tree was optionally built using the surface visibility, 19.8 seconds were required for the GPU to approximate the density values (Eq. (5)) at the

(a) KITCHEN_A 1 (b) KITCHEN_A 2 (c) KITCHEN_A 3 (d) KITCHEN_A 4 (e) KITCHEN_A 5

(f) KITCHEN_B 1 (g) KITCHEN_B 2 (h) KITCHEN_B 3 (i) KITCHEN_B 4



(j) CONFERENCE 1 (k) CONFERENCE 2 (l) CONFERENCE 3 (m) CONFERENCE 4

(n) SPONZA 1 (o) SPONZA 2 (p) SPONZA 3 (q) SPONZA 4 (r) SPONZA 5

(s) FAIRY 1 (t) FAIRY 2 (u) FAIRY 3 (v) FAIRY 4

Figure 5: Test scene statistics and camera views of example scenes. Because the visibility of the region seen by the camera affects the performance of our visibility-based kd-trees, we selected camera views of various visibilities, ranging from good to bad, to enable fair comparison with the SAH method.

1,000,000 surface sample points. The GPU then required 629.0 seconds to estimate the density values (Eq. (10)) at the grid centers (including the nearest surface-sample search using a $200 \times 200 \times 200$ grid). An additional 8.50 seconds were spent by the CPU constructing the actual kd-tree. On the other hand, it took 5.39 seconds for the CPU to build the corresponding SAH tree for the scene.

Comparison with the standard SAH In comparing the voxel-visibility heuristic with the standard SAH, two different tests were performed. In the first voxel-visibility heuristic test (VVH1), a single kd-tree was built via the visibility metric $\Upsilon(V)$ described in Section 3.4. This was used for tracing both primary and secondary rays. In the second voxel-visibility heuristic test (VVH2), we built a kd-tree for tracing only primary rays, and constructed another kd-tree, using the metric $\Upsilon^{sec}(V)$ described in Section 3.5, for tracing secondary rays. Table 7 shows the overall statistics for the frame rates achieved when a 1024×1024 image was generated by full ray tracing using four-threaded 4×4 SIMD ray packets for the CPU and CUDA blocks of 8×8 threads for the GPU.

As summarized in the graph of Figure 6, the new kd-tree construction heuristics achieve significant overall improvements in the rendering time, giving up to 53% and 42% speedup in the CPU-based and GPU-based computations, respectively. Although the speedup achieved differs slightly between the CPU-based and GPU-based versions, the measurements exhibit a quite consistent pattern of improvement. The test results indicate that one of the most significant factors affecting the rendering performance is the visibility of the region within the camera's viewing volume, as was the focus of our work. It is observed that our visibility-based cost metrics tend to generate leaf nodes in the higher-visibility regions at the upper (closer to root node) levels of trees than the SAH metric, enabling ray traversal within such regions to be more efficient. (Note that this does not mean that larger leaf nodes are created in the higher-visibility regions.) For example, when the camera views an open, mostly unoccluded space, as in Frame 5 of KITCHEN_A (Figure 5(f)), the visibility-based kd-trees give very good performance. However, they offer relatively little speedup in the rendering computation when the camera views a region of low visibility, as in Frame 2 of KITCHEN_A (Figure 5(c)). In general, however, the visibility-based kd-trees perform quite favorably when compared with the SAH, for both types of processors.

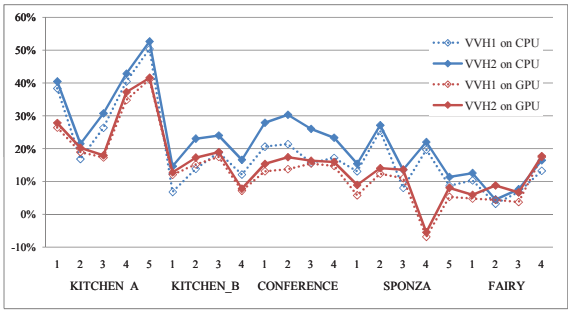
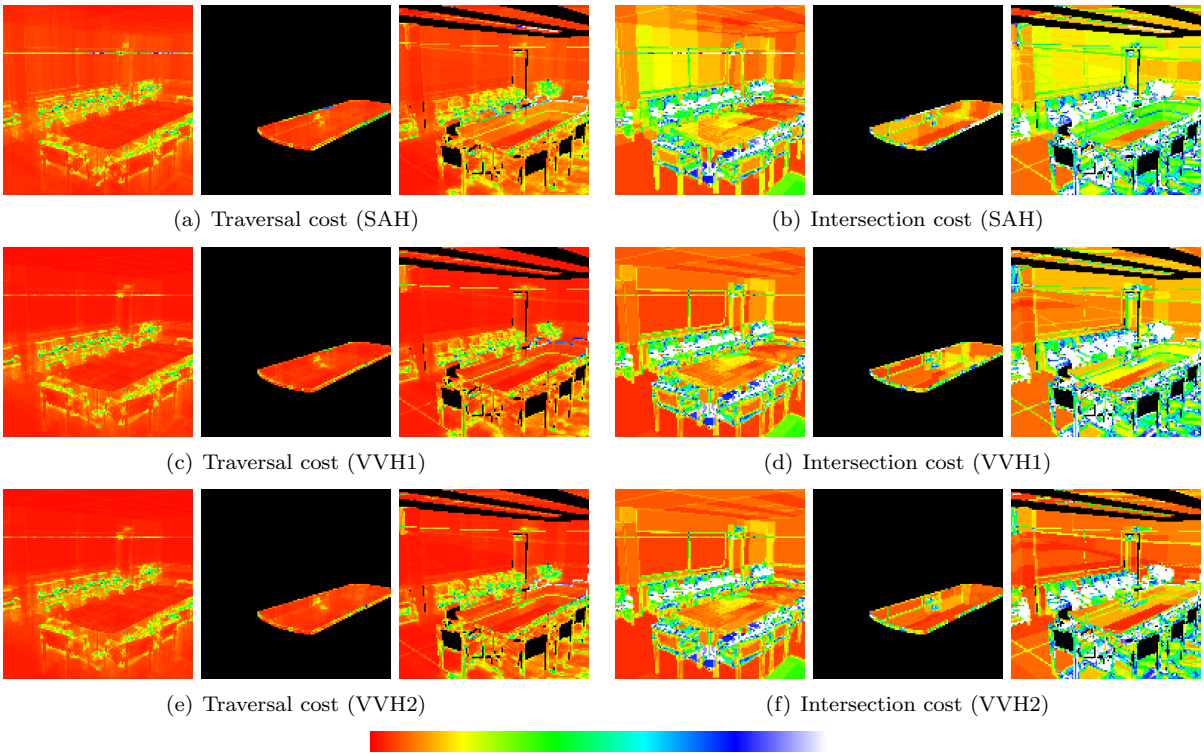


Figure 6: Speedup above the SAH method. Although the performance enhancement depends on several parameters of the test scenes, our experiments show that the cost functions based on voxel visibility perform very favorably when compared with the SAH.

As expected, the efficiency of the two-tree scheme (VVH2) over the one-tree scheme (VVH1) depends on how frequently secondary rays are generated in comparison with primary rays. For example, the same number (i.e., 1,048,576) of primary rays were shot to render the KITCHEN_B 2 and SPONZA 2 frames, but the numbers of traced secondary rays differed significantly, with 8,986,528 rays (1,163,616 reflection rays and 7,822,912 shadow rays) and 1,071,232 rays (120,640 reflection rays and 950,592 shadow rays) being shot, respectively. Using a second kd-tree dedicated to the secondary rays enabled an additional 9.2% speedup via the CPU for the KITCHEN_B frame, whereas only a 1.7% improvement was observed for the SPONZA frame. It would be natural for the two-tree scheme to become more effective when secondary rays outnumber primary rays, as in scenes containing many lights, soft shadows, and/or high numbers of reflection bounces.

It should be noted that the speedup values given in Table 7 are with respect to the entire ray-tracing time, including the setup and shading computations. The actual improvement attributable to the new kd-trees is better than those figures indicate, as is demonstrated in Table 8, which compares the costs of kd-tree traversal and ray–triangle intersection for the three types of rays. We observe that the cost metric $\Upsilon(V)$, used to build the kd-tree in VVH1, tends to reduce both costs equally. In contrast, the metric $\Upsilon^{sec}(V)$, applied to building the second kd-tree in VVH2, appears



(a) Traversal cost (SAH) (b) Intersection cost (SAH)

(c) Traversal cost (VVH1) (d) Intersection cost (VVH1)

(e) Traversal cost (VVH2) (f) Intersection cost (VVH2)

(g) Color map

Figure 7: Per-pixel counts of kd-tree traversal and ray–triangle intersection steps for the CONFERENCE scene. Each triplet of images shows a typical pattern for the respective costs to the CPU for handling primary, reflection/refraction, and shadow rays, in which the numbers of tree-traversal steps ('Traversal cost') and ray–triangle intersection operations ('Intersection cost') carried out per pixel are color coded, increasing from red through yellow, green, and blue to white. Here, white indicates 180 and 60 for the traversal and intersection computations, respectively.

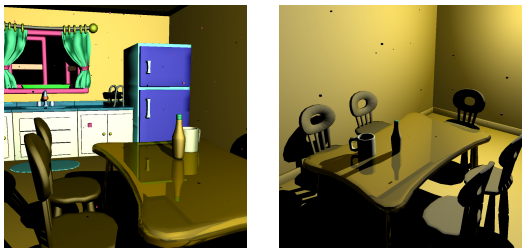
to offer more efficiency in reducing the intersection computation during secondary-ray tracing, which takes relatively longer than the traversal part on the CPU, despite the use of optimization techniques such as mailboxing. In some tests, the kd-tree traversal for the secondary rays was slower than for the SAH, as it tried to maximize the gain in the intersection part. In Figure 7, which illustrates the per-pixel counts of the tree-traversal steps and the ray–triangle intersection operations performed on the CPU, this tendency is clearly observed, with the major cost reduction being found more easily in the intersection part.

Effect of the empty-box heuristic As explained in Subsection 3.6, the heuristic of placing an empty box at the top of the kd-tree is often effective when a given scene contains large empty space inside it. Among the tested scenes, KITCHEN_B and KITCHEN_A are the two for which this heuristic worked best in most cases. As indicated by the statistics in Table 3, it offered good improvement in frame rate for both the SAH and VVH2 schemes, which is remarkable considering the simplicity of the technique. When the heuristic was applied to both construction methods for the KITCHEN_B scene, the voxel-visibility heuristic gave only 5% to 11% speedup over the SAH. This is because the selected empty box produced the top portion of the kd-tree so effectively that the advantage of exploiting the voxel visibility during the kd-tree construction was reduced. Nevertheless, for the KITCHEN_A scene, we can see that the voxel-visibility heuristic still achieved more than 20% of additional increase in frame rate.

Note that the empty-box heuristic is effective only when a given scene contains a 'good' empty box inside it. We tested the heuristic with another scene (KITCHEN_A*), which is the same as the KITCHEN_A scene except it additionally contains

Scene		1	2	3	4	5
KIT._B	SAH ×	1.07	1.14	1.19	0.74	
	SAH ○	1.17	1.25	1.33	0.82	
	VVH2 ×	1.09	1.22	1.25	0.77	
	VVH2 ○	1.23	1.39	1.46	0.86	
KIT._A	SAH ×	6.05	6.79	8.57	5.72	5.97
	SAH ○	6.82	6.89	9.59	6.76	7.43
	VVH2 ×	8.29	8.10	11.13	7.80	8.20
	VVH2 ○	8.48	8.33	11.37	8.08	9.08
KIT._A*	SAH ×	6.07	6.28	8.06	5.88	6.58
	SAH ○	6.09	6.30	8.08	5.90	6.65
	VVH2 ×	7.56	7.23	10.18	7.09	7.77
	VVH2 ○	7.32	7.42	10.05	6.88	7.85

Table 3: Effect of the empty-box heuristic. The SAH and VVH2 methods with (○) and without (×) the empty-box heuristic applied are compared. The scene KITCHEN_A* is identical to KITCHEN_A except that the model additionally contains 100 small cubes in the open space that hinder from locating a 'good' empty box. The frame rates achieved when a 1024×1024 image was generated on the CPU are reported.



(a) KITCHEN_A* 1 (b) KITCHEN_A* 3

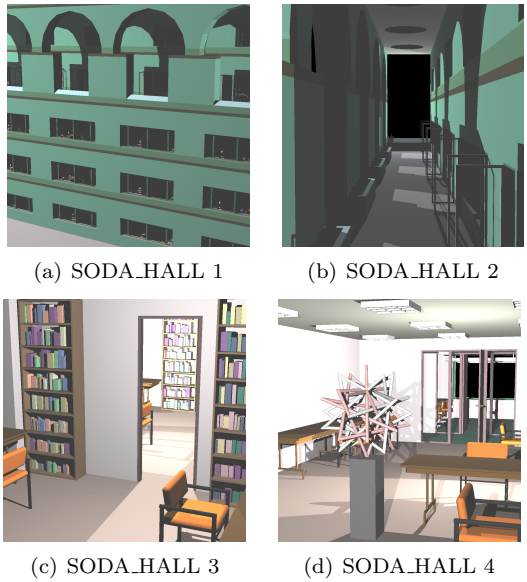
Figure 8: Two camera views of KITCHEN_A*.

100 randomly distributed small cubes in the open space, which hinder from finding large empty space (see Figure 8). As can be seen in Table 3, only insignificant improvement was accomplished by the heuristic in both kd-tree contraction methods. On the other hand, the voxel-visibility heuristic consistently achieved about 20% speedup on average whether the empty-box heuristic was applied or not. Lastly, it should be noted that using more than one empty box only slowed down the ray tracing computation because the increasing number of splitting planes for them diminishes the positive effect of the heuristic.

Test with some other models We also tested our method with some additional scenes to see how our method performs on larger models. Table 4 shows the statistics collected on the same CPU for the Soda Hall model (2.4M triangles), which has complex occlusion. To compute the voxel visibility, the entire scene domain was discretized into a $1024 \times 1024 \times 1024$ grid, where 2,048 directions were sampled using stratified sampling over the hemisphere around each cell's center. To compute the surface visibility, on the other hand, 1,000,000 points were uniformly distributed over the geometry. During new kd-tree construction, a switch from voxel-visibility to surface-area heuristic was made with the threshold value of $C_L = 10$, and the empty-box heuristic was not applied because it only made an insignificant improvement in rendering speed for this model.

Cam.	SAH	VVH1	VVH2
1	2.63	2.64 (0.5%)	2.58 (-1.6%)
2	2.91	3.06 (5.4%)	3.12 (7.4%)
3	2.24	2.47 (10.3%)	2.53 (13.1%)
4	0.79	1.50 (89.0%)	1.53 (92.9%)

Table 4: Rendering speed comparison with the standard SAH method. The frame rates achieved when a 1024×1024 image was generated by full ray tracing the Soda Hall model on the CPU are reported. For this model, the idea of placing an empty box at the top level of the kd-tree was not applied because it does not contain empty space inside the scene that is large enough to allow a significant performance increase.



(a) SODA_HALL 1 (b) SODA_HALL 2

(c) SODA_HALL 3 (d) SODA_HALL 4

Figure 9: Camera views of SODA_HALL.

Figure 9(a) to (d) show the four representative camera views that we selected for fair comparison with the SAH method. When the camera viewed the building from the outside (the SODA_HALL 1 frame), no improvement in frame rate was observed with our kd-trees. For this particular frame,

the SAH tree was sufficiently efficient because the SAH's assumption that rays come from outside the scene is rather effective due to the camera's position and orientation, and there was no room for the visibility idea to work. Using two trees (VVH2) only slowed down the rendering process.

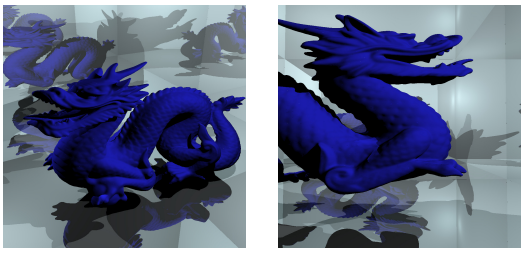
In the second frame (SODA_HALL 2), the camera was still outside the building. However, the nontrivial occlusion situation that its view created caused our kd-tree schemes to produce higher frame rates. When the camera was put in a room (the SODA_HALL 3 frame), the voxel-visibility based kd-trees started to provide significant improvement in frame rate over the standard SAH tree as intended. In the fourth test frame (SODA_HALL 4), we observed a somewhat unusual performance gain of 89.0% and 92.9% by our methods. We don't exactly know the reason for this, but conjecture that the SAH tree was built rather poorly around that particular region.

In addition, to understand how our methods perform in a simple situation like when a single scanned model is viewed, we tested with the DRAGON model (871K triangles) put in a rectangular room with reflective walls (see Figure 10 for some example views). As expected and clearly indicated in Table 5, it was hard to find any meaningful performance difference between the voxel-visibility and surface-area heuristics, particularly if a single-tree scheme (VVH1) was used. The reason for this poor performance is basically the same as that of the SODA_HALL 1 frame as we explained. With the two-tree scheme (VVH2), some increase in frame rate, although limited, was seen thanks to the efficient processing of reflection rays by the secondary, surface-visibility based kd-tree.

Cam.	SAH	VVH1	VVH2
1	1.35	1.34 (-0.2%)	1.39 (3.5%)
2	1.39	1.36 (-1.8%)	1.44 (3.5%)
3	1.31	1.33 (1.5%)	1.31 (0.2%)
4	1.52	1.53 (0.7%)	1.60 (4.8%)
5	1.92	1.92 (0.1%)	2.01 (5.0%)

Table 5: Rendering speed comparison with the standard SAH method. The frame rates achieved in the same CPU environment are reported for the DRAGON model.

Comparison with other methods We compared our cost metrics with another variant of the SAH, presented by Fabianowski et al. [17], which may be viewed as a simplified version of our method in that the cost function also reflects internal rays originating within voxels. In this test, we built the corresponding kd-trees simply by replacing our cost metric with the 'fast approximation' measure, and examined two pairs of constant parameters, namely $(C_T = 1, C_I = 3)$, which was used in their paper ('[FFD09] (a)'), and $(C_T = 1, C_I = 2)$, which showed the best performance improvement generally in our test environment ('[FFD09] (b)').



(a) DRAGON 3 (b) DRAGON 5

Figure 10: Two camera views of DRAGON.

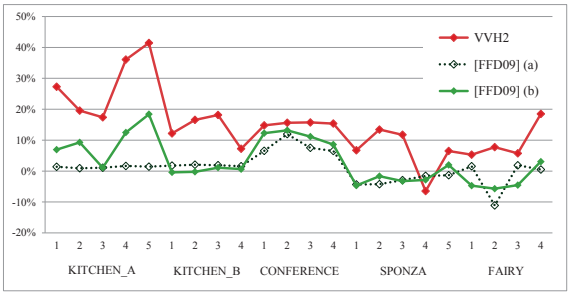


Figure 11: Comparison with the cost metric of [17].

As illustrated in Figure 11, the resulting speedups show that our method ('VVH2'), evaluated with the constant pair $(C_T = 1, C_I = 1)$ as mentioned above, consistently outperformed their approach, although the differences vary slightly depending on the chosen parameter set. Note that their idea of considering both internal and external rays in estimating the probability of a node being visited was valid. However, because they were more concerned with a rapid approximation to the probability, their simplified formulation was relatively limited in modeling effectively the ray distribution in space. Our visibility-based strategy requires more time for evaluating the cost metric, but as the scenes become more complex, the better our relative performance will become.

To confirm that these improvements were not specific to our implementation, we also experimented with a different GPU

ray tracer, publicly available at http://dcgi.felk.cvut.cz/members/havran/rtgpu2009/ [26]. To avoid modifying its core implementation, which employs a single kd-tree for acceleration, we examined only the single-tree scheme (VVH1), in which the tested kd-trees were simply converted to its file format. The experiment was performed on the NVIDIA GeForce GTX 280 GPU, on which (we assume) the ray tracer was implemented. Figure 12 displays the speedups achieved with respect to the standard SAH kd-tree scheme. Compared with the figures achieved with our GPU ray tracer ('VVH1 on GPU' in Figure 6), we again obtained good performance gains, although the graph exhibits a slightly different improvement pattern. Considering that this ray tracer was not tuned for our kd-tree construction algorithm, and that we still achieved on average a more than 15% increase in frame rate with the GPU, we believe that the results we have presented will not be limited to our particular ray tracers.

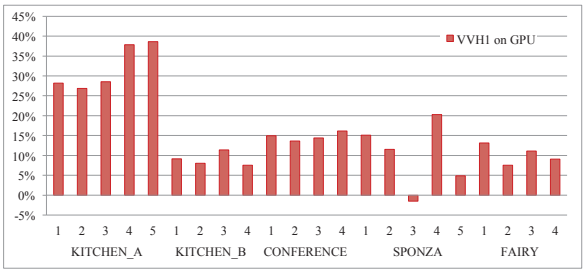


Figure 12: Speedup above the SAH method, using a publicly available GPU ray tracer [26].

Estimation of the expected costs Finally, we estimated and compared the expected costs of the complete kd-trees T that were built using the two different strategies, where we modified the often-used expected cost [3, 27], by replacing the surface-area term $SA(V)$ with the new term $RayHitCount(V)$:

$$C(T) = C_T \cdot \sum_{N \in Nodes} \frac{RayHitCount(V_N)}{RayHitCount(V_S)} + C_I \cdot \sum_{L \in Leaves} \frac{RayHitCount(V_L)}{RayHitCount(V_S)} n_L.$$

Here, V_S is the axis-aligned bounding box of the entire scene, n_L is the number of triangles in leaf node L, and $RayHitCount(V)$ of voxel V is the number of rays that either originate within V or start from outside but intersect with V, incurring traversal of the corresponding kd-tree node. The idea behind this modified cost is that, when random rays are generated repeatedly, $\frac{RayHitCount(V)}{RayHitCount(V_S)}$ is a better approximation to the real probability of the node being traversed, reflecting the specific geometric characteristics of a given scene.

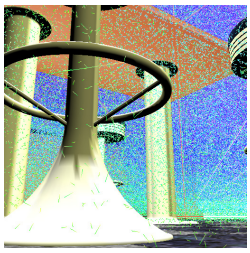
Scene	SAH	VVH1 (1)	VVH1 (2)
KITCHEN_A	30.71	26.48	23.42
KITCHEN_B	29.43	25.34	22.76
CONFERENCE	34.19	30.08	27.96
SPONZA	39.22	36.81	34.43
FAIRY	34.06	28.91	26.52

Table 6: Comparison of the expected costs of kd-trees using random rays selected stochastically according to the computed visibility. This experiment shows empirically that our voxel-visibility heuristic works as intended.

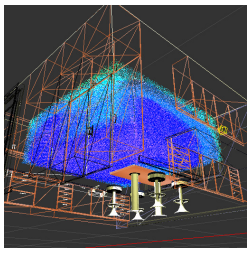
To simulate the rays that originate in empty space, we reused the grid explained in Section 3.2 and 3.3, and applied a rejection method to select cells based on the computed voxel visibility. We then generated rays from the centers of the cells by importance-sampling the six visibility values (refer to Figure 13(a) for an example). Table 6 shows the resulting costs, estimated by using two million of such random rays. As seen by the figures in the 'VVH1(1)' column that represent the costs for random rays that may exist anywhere in the entire empty space, the expected costs reduced markedly when the voxel-visibility techniques were applied, indicating that the new methods had built kd-trees that better match the actual geometry and possible ray distribution in the scene. When the viewer (camera) can stay only within some restricted empty space with good visibility, as shown in Figure 13(b), the expected costs decreased further, as expected (see the 'VVH1(2)' column in the table). In many applications, the viewer often tends to stay within a region of good visibility, such as during a walkthrough of a building model. In such cases, the visibility-based kd-trees will be able to provide an additional performance gain for the interactive ray tracing.

5. Concluding remarks

In this paper, we presented two heuristic methods for building efficient kd-trees for recursive ray tracing, and demonstrated their effectiveness through several examples. The implementation for both CPU-based and GPU-based computation showed



(a) Rays from the entire empty space



(b) Rays from a restricted empty space

Figure 13: Stochastically generated rays for estimating the expected costs for kd-trees (KITCHEN_B).

significant reductions in both ray-traversal and rendering times. The main idea was to exploit the 'visibility of voxels' so that those with higher visibility are processed more efficiently in the kd-trees. Although the efficiency was often degraded when, for example, the camera viewed a region of low visibility, the overall performance compared very favorably with that of the SAH.

Currently, our method is appropriate only for static scenes because the computation for constructing the incident ray density field usually takes several minutes for nontrivial scenes. Because our central concern was to enhance the run-time ray tracing performance for static scenes, we have implemented the precomputation code in a rather straightforward manner, in which the construction cost is basically linear to the numbers of ray samples and discretized cells in open space, and is logarithmic to the number of polygons.

It is interesting to see that, according to our preliminary experiment (for example, refer to the frame rates below measured on the CPU over several grids of different resolutions using 2,048 direction samples over the hemisphere), the achieved speedup was not so sensitive to the grid's resolution.

Scene	Grid	1	2	3	4
KITCHEN_B	32^3	1.22	1.36	1.43	0.84
	64^3	1.22	1.37	1.44	0.84
	128^3	1.23	1.38	1.45	0.85
	256^3	1.24	1.39	1.47	0.86
	512^3	1.26	1.39	1.47	0.86
	1024^3	1.26	1.41	1.48	0.87

Note that increasing the grid's resolution twice would provide an additional precision of only a few depths in the voxel-visibility based kd-tree. This observation may suggest that it is, in fact, cost-effective to construct an upper part of the kd-tree well using a grid of rather low resolution, and build the remaining part using the SAH, which is exactly what we described in Subsection 3.6. In any case, finding a way of obtaining an optimal level of discretization for the density field approximation that reduces the cost significantly without introducing serious discretization errors is left as a future research work.

Finally, we conjecture that the idea of visibility can lead to an effective tool for building efficient acceleration structures for the real-time ray tracing of dynamic scenes. For example, if we record a ray count per triangle that increments every time it is examined by a ray, whether primary or secondary, this additional form of 'visibility information' could be exploited to construct a high-quality kd-tree applicable to the next frame rendering. Combined with a fast SAH-based kd-tree construction method, e.g., [12], a visibility-based algorithm might be able to update kd-trees for dynamic scenes progressively and efficiently.

Acknowledgment

We are very grateful to the anonymous reviewers for their constructive comments and suggestions. This work was partially supported by the Industrial Strategic Technology Development Program funded by the Ministry of Knowledge Economy of Korea (No. 10035261-2011-01).

References

[1] J. MacDonald, K. Booth, Heuristics for ray tracing using space subdivision, The Visual Computer 6 (1990) 153–166.

[2] L. Santalo, Integral Geometry and Geometric Probability, Cambridge University Press, 2002.

[3] V. Havran, Heuristic ray shooting algorithms, Ph.D. thesis, Czech Technical University, Prague (2001).

[4] L. Szécsi, B. Benedek, L. Szirmay-Kalos, Accelerating animation through verification of shooting walks, in: Proc. of SCCG '03, 2003, pp. 231–238.

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

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

[7] Q. Hou, X. Sun, K. Zhou, C. Lauterbach, D. Manocha, Memory-scalable GPU spatial hierarchy construction, IEEE Transactions on Visualization and Computer Graphics 17 (4) (2011) 466–474.

[8] A. Reshetov, A. Soupikov, J. Hurley, Multi-level ray tracing algorithm, in: Proc. of ACM SIGGRAPH '05, 2005, pp. 1176–1185.

[9] I. Wald, P. Slusallek, C. Benthin, M. Wagner, Interactive rendering with coherent ray tracing, Computer Graphics Forum 20 (3) (2001) 153–164.
[10] I. Wald, Realtime ray tracing and interactive global illumination, Ph.D. thesis, Saarland University (2004).
[11] I. Wald, V. Havran, On building fast kd-trees for ray tracing, and on doing that in O(Nlog N), in: Proc. of the IEEE Symposium on Interactive Ray Tracing, 2006, pp. 61–69.
[12] W. Hunt, W. R. Mark, G. Stoll, Fast kd-tree construction with an adaptive error-bounded heuristic, in: Proc. of 2006 IEEE Symposium on Interactive Ray Tracing, 2006, pp. 81–88.
[13] S. Popov, J. Günther, H.-P. Seidel, P. Slusallek, Experiences with streaming construction of SAH KD-trees, in: Proc. of the IEEE Symposium on Interactive Ray Tracing, 2006, pp. 89–94.
[14] J. Hurley, A. Kapustin, A. Reshetov, A. Soupikov, Fast ray tracing for modern general purpose CPU, in: Proc. of Graphicon, 2002, p. 2002.
[15] W. Hunt, Corrections to the surface area metric with respect to mail-boxing, in: Proc. of IEEE Symposium on Interactive Ray Tracing, 2008, pp. 77–80.
[16] E. Reinhard, A. J. F. Kok, F. W. Jansen, Cost prediction in ray tracing, in: Proc. of the Eurographics Workshop on Rendering Techniques '96, 1996, pp. 41–50.
[17] B. Fabianowski, C. Fowler, J. Dingliana, A cost metric for scene-interior ray origins, in: Eurographics '09 Short Papers, 2009, pp. 49–52.
[18] I. Wald, J. Gunther, P. Slusallek, Balancing considered harmful - faster photon mapping using the voxel volume heuristic, Computer Graphics Forum 23 (3) (2004) 595–603.
[19] T. Ize, C. Hansen, RTSAH traversal order for occlusion rays, Computer Graphics Forum (Proc. of Eurographics '11) 30 (2) (2011) 297–305.
[20] E. Zhang, G. Turk, Visibility-guided simplification, in: Proc. of IEEE Visualization '02, 2002, pp. 267–274.
[21] M. Srikanth, P. Mathias, V. Natarajan, P. Naidu, T. Poston, Visibility volumes for interactive path optimization, The Visual Computer 24 (2008) 635–647.
[22] P. Dutré, P. Tole, D. Greenberg, Approximate visibility for illumination computations using point clouds, Tech. Rep. PCG–00–1, Program of Computer Graphics, Cornell University (2000).
[23] G. Schaufler, J. Dorsey, X. Decoret, F. Sillion, Conservative volumetric visibility with occluder fusion, in: Proc. of ACM SIGGRAPH '00, 2000, pp. 229–238.
[24] J. Bittner, P. Wonka, Visibility in computer graphics, Journal of Environment and Planning B: Planning and Design 30 (5) (2003) 729–756.
[25] S. Zhukov, A. Iones, G. Kronin, An ambient light illumination model, in: Rendering Technique '98 (Proc. of the Eurographics Workshop on Rendering), 1998, pp. 45–55.
[26] M. Zlatuska, V. Havran, Ray tracing on a GPU with CUDA – comparative study of three algorithms, in: Proc. of WSCG '10, communication papers, 2010, pp. 69–76.
[27] J. Bittner, V. Havran, RDH: Ray distribution heuristics for construction of spatial data structures, in: Proc. of SCCG '09, 2009, pp. 61–67.

Scene	Cam.	CPU			GPU		
		SAH	VVH1	VVH2	SAH	VVH1	VVH2
KITCHEN_A	1	6.04	8.36 (1.38x)	8.49 (1.40x)	38.63	48.83 (1.26x)	49.16 (1.28x)
	2	6.80	7.94 (1.17x)	8.26 (1.22x)	34.88	41.49 (1.19x)	41.70 (1.20x)
	3	8.65	10.94 (1.26x)	11.31 (1.31x)	47.53	55.76 (1.17x)	55.79 (1.18x)
	4	5.68	7.97 (1.40x)	8.11 (1.43x)	33.52	45.19 (1.35x)	45.61 (1.37x)
	5	5.99	9.01 (1.50x)	9.14 (1.53x)	34.29	48.40 (1.41x)	48.51 (1.42x)
KITCHEN_B	1	1.07	1.15 (1.07x)	1.23 (1.15x)	10.29	11.54 (1.12x)	11.55 (1.12x)
	2	1.13	1.29 (1.14x)	1.39 (1.23x)	11.70	13.48 (1.15x)	13.64 (1.17x)
	3	1.18	1.40 (1.19x)	1.47 (1.24x)	13.07	15.35 (1.17x)	15.44 (1.18x)
	4	0.74	0.82 (1.12x)	0.86 (1.17x)	6.88	7.38 (1.07x)	7.38 (1.07x)
CONFER-ENCE	1	7.56	9.12 (1.21x)	9.67 (1.28x)	26.85	30.37 (1.13x)	30.82 (1.15x)
	2	8.65	10.49 (1.21x)	11.27 (1.30x)	54.97	62.52 (1.14x)	63.56 (1.17x)
	3	7.07	8.19 (1.16x)	8.92 (1.26x)	27.59	31.86 (1.15x)	31.94 (1.16x)
	4	8.12	9.52 (1.17x)	10.02 (1.23x)	28.94	33.20 (1.15x)	33.38 (1.16x)
SPONZA	1	5.72	6.47 (1.13x)	6.60 (1.15x)	32.56	34.43 (1.06x)	34.75 (1.09x)
	2	5.55	6.96 (1.25x)	7.05 (1.27x)	33.82	37.99 (1.12x)	38.36 (1.14x)
	3	7.67	8.29 (1.08x)	8.72 (1.14x)	45.54	50.53 (1.11x)	50.89 (1.14x)
	4	6.47	7.74 (1.20x)	7.89 (1.22x)	33.85	31.51 (0.93x)	31.66 (0.94x)
	5	9.41	10.23 (1.09x)	10.48 (1.11x)	37.37	39.35 (1.05x)	39.80 (1.08x)
FAIRY	1	2.89	3.19 (1.10x)	3.25 (1.13x)	11.78	12.34 (1.05x)	12.40 (1.06x)
	2	3.97	4.10 (1.03x)	4.15 (1.05x)	18.54	19.35 (1.04x)	19.97 (1.09x)
	3	3.24	3.46 (1.07x)	3.49 (1.08x)	13.54	14.05 (1.04x)	14.32 (1.07x)
	4	3.37	3.81 (1.13x)	3.92 (1.16x)	22.07	25.93 (1.17x)	26.15 (1.18x)

Table 7: Rendering speed comparison with the standard SAH method (in frames per second). In this table, VVH1 refers to using the voxel-visibility heuristic to build a single kd-tree. VVH2 refers to using the extended heuristics described in Section 3.5 to build a second kd-tree dedicated to secondary rays. The numbers in parentheses represent the speedup achieved with respect to the SAH. All scenes were ray-traced with local shading, textures, reflection/refraction and shadows at a resolution of 1024×1024.

(Sec.)

Cam.		Primary			Reflection/Refraction			Shadow		
		SAH	VVH1	VVH2	SAH	VVH1	VVH2	SAH	VVH1	VVH2
1	Traversal	0.013	0.009 (1.44x)	0.009 (1.37x)	0.005	0.004 (1.31x)	0.005 (1.07x)	0.014	0.011 (1.32x)	0.013 (1.10x)
	Intersection	0.033	0.020 (1.59x)	0.023 (1.43x)	0.015	0.011 (1.36x)	0.010 (1.53x)	0.048	0.033 (1.44x)	0.031 (1.52x)
2	Traversal	0.013	0.011 (1.09x)	0.011 (1.13x)	0.006	0.005 (1.10x)	0.006 (0.98x)	0.013	0.010 (1.32x)	0.011 (1.17x)
	Intersection	0.029	0.027 (1.08x)	0.027 (1.06x)	0.017	0.015 (1.09x)	0.013 (1.25x)	0.043	0.034 (1.23x)	0.029 (1.48x)
3	Traversal	0.009	0.008 (1.20x)	0.008 (1.21x)	0.002	0.002 (1.07x)	0.002 (0.94x)	0.011	0.009 (1.17x)	0.010 (1.07x)
	Intersection	0.029	0.019 (1.49x)	0.020 (1.40x)	0.006	0.005 (1.14x)	0.005 (1.22x)	0.035	0.030 (1.15x)	0.027 (1.27x)
4	Traversal	0.011	0.009 (1.17x)	0.010 (1.11x)	0.007	0.006 (1.19x)	0.007 (1.02x)	0.017	0.012 (1.37x)	0.015 (1.14x)
	Intersection	0.028	0.019 (1.46x)	0.021 (1.32x)	0.018	0.014 (1.28x)	0.013 (1.47x)	0.052	0.035 (1.50x)	0.033 (1.56x)
5	Traversal	0.011	0.009 (1.20x)	0.009 (1.26x)	0.004	0.004 (1.09x)	0.005 (0.92x)	0.017	0.012 (1.41x)	0.014 (1.21x)
	Intersection	0.035	0.020 (1.72x)	0.020 (1.73x)	0.013	0.012 (1.09x)	0.009 (1.43x)	0.049	0.032 (1.56x)	0.026 (1.91x)
Ave.	Traversal		(1.21x)	(1.21x)		(1.16x)	(1.00x)		(1.33x)	(1.14x)
	Intersection		(1.44x)	(1.37x)		(1.19x)	(1.39x)		(1.38x)	(1.54x)

Table 8: Detailed analysis of ray-traversal costs. These measurements from the KITCHEN_A scene are typical of CPU-based computation times for the kd-tree traversal ('Traversal') and the ray–triangle intersection ('Intersection'). The numbers in parentheses represent the speedup achieved with respect to the SAH. As expected and shown in this table, the actual improvement by the new kd-trees in the computation of ray–traversal is better than those speedup values in Table 7, which are with respect to the entire ray tracing time. It is interesting to see that, on the CPU, the metric $\Upsilon^{sec}(V)$ for building the second kd-tree in VVH2, tends to offer more efficiency in reducing the intersection computation for the secondary rays, which takes relatively longer than the traversal part.