EUROGRAPHICS 2004 / M.-P. Cani and M. Slater
(Guest Editors)

Volume 23 (2004), Number 3

Balancing Considered Harmful – Faster Photon Mapping using the Voxel Volume Heuristic –

Ingo Wald[†‡], Johannes Günther[†], and Philipp Slusallek[†]

[†]Computer Graphics Group, Saarland University
Saarbrücken, Germany
{guj,slusallek}@graphics.cs.uni-sb.de

[‡]now at Max-Planck-Institute for Computer Science
Saarbrücken, Germany
wald@mpi-sb.mpg.de

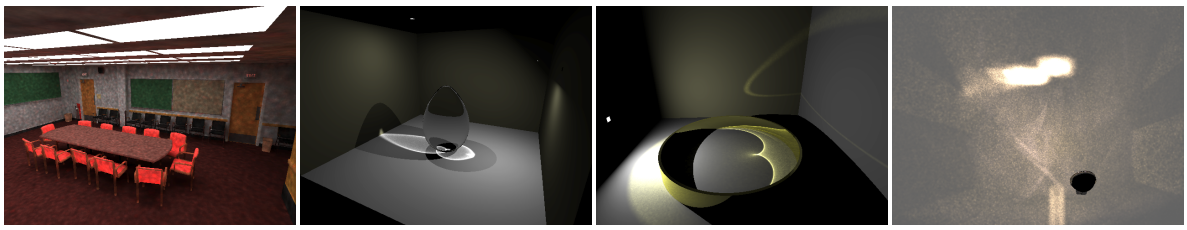

Figure 1: *Our example scenes, from purely diffuse illumination to highly detailed caustics: a) Direct visualization of a diffuse photon map in the "Conference" scene, using 2.1 million photons, b) the "GlassEgg" with a sharp, compact caustic, c) the "MetalRing", with highly detailed caustics distributed over the entire room, and d) a visualization of the illumination patterns generated by the highly complex "HeadLight" model, using a total of 2.3 million photons. In these four scenes, our presented technique improves the performance of photon map queries by a factor of 1.3 to 5.8, resulting in overall speedups (including ray tracing) of 35, 65, 55, and 83 percent, respectively.*

Abstract

Photon mapping is one of the most important algorithms for computing global illumination. Especially for efficiently producing convincing caustics, there are no real alternatives to photon mapping. On the other hand, photon mapping is also quite costly: Each radiance lookup requires to find the k nearest neighbors in a kd-tree, which can be more costly than shooting several rays. Therefore, the nearest-neighbor queries often dominate the rendering time of a photon map based renderer.
In this paper, we present a method that reorganizes – i.e. un*balances – the kd-tree for storing the photons in a way that allows for finding the k-nearest neighbors much more efficiently, thereby accelerating the radiance estimates by a factor of 1.2–3.4. Most importantly, our method still finds exactly the same k-nearest-neighbors as the original method,* without *introducing any approximations or loss of accuracy. The impact of our method is demonstrated with several practical examples.*

Keywords: Global Illumination, photon mapping, simulation, caustics, kd-tree, nearest neighbor query

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Global Illumination I.3.7 [Computer Graphics]: Raytracing

1. Introduction

Fast and high-quality global illumination is a long-standing goal in computer graphics. Over the last twenty years, a large variety of algorithms have been developed, such as (bidirectional) path tracing [Kaj86, LW93, VG94], Metropolis Light Transport [VG97], many different types of radiosity algorithms [HSA91, SAG94, Kel97], and photon mapping [Jen96, Jen97, Jen01].

At least for restricted lighting models, the recent improvements in ray tracing performance [WSBW01, WPSBS03, Wal04] now allow for computing global illumination at interactive frame rates [WKBKS02, BWS03]. However, these algorithms

support only a subset of all possible light transport paths, which is particularly problematic for accurately rendering caustics.

Today, the most efficient technique for producing convincing caustics is photon mapping. In fact, photon mapping is a relatively simple technique: In a preprocessing step, particles are shot from the light sources into the scene, and their hit points (including position, direction, and photon power) are recorded. These photons are then organized in a kd-tree for efficient lookups. This kd-tree is usually stored in a left-balanced form. After the preprocessing step, the (ir-)radiance at any given surface point can be estimated via density estimation among the k nearest neighbor (kNN) photons, which can be determined by traversing the kd-tree[Jen01, Ben75].

Though photon mapping is both simple and able to produce smooth, convincing caustics, it is unfortunately also quite costly: even when ignoring all preprocessing cost for generating the photons and for building the kd-tree, each radiance lookup requires a traversal of the index structure in order to find the k nearest neighbors. This can easily be more costly than shooting several rays (especially when using a fast ray tracer), and can easily become the bottleneck. For example, on a 2.2GHz Pentium IV the conference scene in Figure 1a can be ray traced at 3.6 frames per second (using ray casting and simple shading). However, when directly visualizing a 2M photon map – i.e. performing exactly one radiance estimate per primary ray – the frame rate drops to 0.1 fps.

In this paper, we are going to present a technique that explicitly *un*balances the kd-tree by recalling some of the lessons learned in building high-quality BSP trees for ray tracing. Applying some of these lessons to the kd-trees used in photon mapping, we can significantly reduce the time spent in nearest neighbor queries, and thus increase the performance of the photon map's radiance estimate by roughly a factor of 1.3–3.5.

2. Background and Previous Work

For global illumination, there has been vast amount of previous work, e.g. [Kaj86, LW93, VG94, VG97, HSA91]-[Kel97, SAG94, Jen96, Jen97], which we cannot cover in more detail. Instead, we concentrate only on work that is directly related to our proposed technique: Photon mapping on one hand, and the construction of optimized kd-trees in ray tracing on the other hand.

2.1. Photon Mapping

Since its original publication by Jensen et al. [Jen96, Jen97], photon mapping has become one of the most widely used techniques for global illumination, mostly because of its ability to efficiently generate smooth, high-quality caustics. This is achieved by shooting photons into the scene, storing their hit points, and using these photons to estimate the radiance at any given point. This estimate is performed by locating the k nearest neighboring photons to this surface point, and using those for density estimation [Jen01].

Using the k nearest photons for reconstructing the irradiance, the size of the reconstruction filter automatically adapts to the local photon density. This enables a highly detailed reconstruction of caustics (where the photon density is high) while still giving a smooth estimate of the diffuse illumination (for which the filter automatically gets larger).

2.2. Efficient k-Nearest Neighbor Queries

In order to efficiently find these k nearest neighbors, the photons are stored in a special kd-tree: Each node in the kd-tree represents both a photon and an axis-aligned splitting plane passing through that photon. In order to keep the memory consumption low, this kd-tree is represented in a very compact form [Jen01]: First, the direction and power of each photon is stored in a discretized form, using 2 bytes for the discretized direction, and a 4-byte RGBE [War92] representation for the energy. The splitting plane is stored as an additional 16-bit `short` for alignment reasons.

Second, the kd-tree is stored in a left-balanced form, for which the children of a node can be addressed implicitly without pointers. Taken together, this allows for a very memory-efficient representation using only 20 bytes per photon [Jen01]. Virtually all photon map implementations today follow this scheme.

As the kNN query is quite expensive, several researchers have recently proposed other, faster ways of performing the radiance estimate, usually in the context of interactive visualizations of the photon map (e.g. [WKBKS02, PDCJH03, MM02, WS03]). However, these techniques usually rely on approximations, e.g. by only approximating the k-nearest neighbors, or by using a fixed query radius that does not automatically adapt to the photon density. In contrast to these approaches, we are taking care not to introduce any approximations, but rather find exactly the same set of nearest neighbors as the original technique.

2.3. Cost Functions for Building BSP Trees

Kd-trees are not only used in photon mapping, but also quite extensively in ray tracing. In order to stress the difference to the kd-trees used in photon mapping, we will in the following denote these with "BSP trees". Though there are some minor differences (see below), in principle both kd-trees and BSP trees refer to the same kind of three-dimensional, binary search trees with axis-aligned splitting planes.

Though both kd-trees and BSPs are used for similar applications, the exact way that they are stored and built varies significantly: In photon mapping, virtually all photon map

implementations use the left-balanced tree without pointers as described above. In ray tracing, however, it is a well-known fact that splitting at the median – i.e. balancing the tree – actually performs *worse* than any other technique [Hav01]. Though balancing the tree appeals to intuition (as it guarantees a minimum average node depth), it is a common misconception that this structure would also yield minimal traversal times. Balancing is optimal *only* for binary searching, and if all nodes have equal access probabilities. Neither of these two prerequisites is fulfilled for range queries (such as ray traversal and kNN queries), nor for unevenly distributed primitives such as photons or triangles.

Thus, in ray tracing researchers have thoroughly investigated how to build BSP trees in other – more optimal – ways (e.g. [MB90, Sub90, Hav01, HKRS02]). In fact, using an optimized BSP tree can result in significant performance improvements. For example, using better BSP trees recently allowed for improving the performance of the (already quite fast) RTRT/OpenRT realtime ray tracing system by almost a factor of two [WPSBS03, Wal04].

2.3.1. Surface Area Heuristic (SAH)

Today, the best traversal performance in ray tracing can be achieved using the "Surface Area Heuristic" (SAH). This heuristic *estimates* the cost of all potential splitting planes, thereby allowing for placing the planes where they are most effective. To estimate the cost of splitting a voxel V into two halves V_L and V_R, two factors have to be considered: First, the cost $C(V_L)$ and $C(V_R)$ of traversing each of the two children, and second, the probabilities $P(V_L)$ and $P(V_R)$ of traversing the respective children assuming that the parent node has been hit. Then, the cost $C(V)$ of splitting V by plane S can be estimated as

$$C(S:V\rightarrow\{V_L,V_R\}) = C_{trav} + P(V_L)C(V_L) + P(V_R)C(V_R),$$

where C_{trav} is the (measured) cost for performing a single traversal step. As this is a recursive definition of C, one usually approximates the cost for the leaves as

$$C(V_L) \approx C_{isec}N_L \quad \text{and} \quad C(V_R) \approx C_{isec}N_R,$$

where C_{isec} is the cost for a ray-primitive intersection, and N_L, N_R are the number of primitives overlapping V_L and V_R, respectively.

In order to determine the probabilities $P(V_L)$ and $P(V_R)$ of traversing the left and right children, respectively, one assumes that the rays are equally distributed and do not terminate inside the voxel. Then, it can be shown [MB90, San02] that the probability of traversing V_L and V_R is

$$P(V_L) = \frac{SA(V_L)}{SA(V)} \quad \text{and} \quad P(V_R) = \frac{SA(V_R)}{SA(V)},$$

where $SA(V)$ is the surface area (hence the name) of the voxel V (see [San02]: The "conditional probability of intersecting two convex bodies one inside the other knowing that we intersect the biggest is the ratio of surfaces"). Note that for the BSP trees used in ray tracing, in general neither $P(V_L)+P(V_R)=1$ (since a ray can intersect both halves), nor $N_L+N_R=N_V$ (since triangles may overlap both sides) !

Using this heuristic to estimate the cost of all potential splitting planes S, it is possible to place the splitting plane where $C(S)$ reaches a minimum.

Additionally, this cost estimate can be used to automatically determine when to stop the subdivision: As soon as $\min C(S) > C_{isec}N_V$, further subdivision will not pay off. Using such optimized BSP trees, the average number of nodes, leaves, and primitives visited during traversal of a ray is usually much smaller than for BSPs built with other techniques (see [Hav01, Wal04]).

3. The Voxel Volume Heuristic (VVH)

With the positive results that optimized BSPs achieve in ray tracing, it seems promising to investigate whether – and how – similar heuristics can be used to speed up the notoriously slow kNN queries used in photon mapping. Obviously, once we are no longer using balanced kd-trees, we can no longer address the children implicitly, and thus have to store explicit children pointers. Though these could be stored in a compressed form (using one pointer for addressing both children, plus two bits for encoding which children are actually present), we currently spend two full pointers (i.e. 8 bytes) for addressing the children. While this obviously increases our memory consumption, the total memory impact with 8 bytes per node is quite small. As the number of photons in practice is usually in the range of at most a few million, eight bytes per photon today is quite affordable. If the kd-tree is not "too" unbalanced, one could also avoid the pointers by using a balanced kd-tree with "holes" inside. For practical kd-trees however these holes are likely to generate a much larger memory overhead than adding pointers.

3.1. Differences between kd-Trees and BSP trees

As mentioned above, there are a few differences between ray traversal (RT) in a BSP, and k-nearest-neighbor query (kNN) in a kd-tree. Most importantly, kNN queries do not perform their traversal along an infinite line as RT, but rather perform a query inside a spherical volume. This obviously influences the probabilities of traversing the children.

Additionally, kd-trees (as used in photon mapping) are structurally different from BSPs in that they store *point data*, whereas BSP trees store primitives with a spatial extent. These primitives can therefore be contained in multiple voxels, and can even overlap other primitives. In contrast to this, storing only point data allows kd-trees to be structurally much simpler than BSP trees: Kd-trees store photons also in inner nodes, always store exactly one photon per node, and do not differentiate between inner nodes and leaf nodes.

3.2. Implications on our Cost Function

These properties have several implications on our to-be-designed cost function: First, there is no difference between inner nodes and leaf nodes, so there is also no difference between a traversal step and a primitive intersection. This entirely removes the need for the C_{isec} and C_{trav} factors. Second, as each splitting plane in a kd-tree passes through a photon, the number of potential splitting planes is limited. This greatly simplifies the search for the best split plane position, which for BSP trees requires to determine all intersections between primitives and voxel sides [HKRS02, Hav01]. Instead having exactly one photon per node predetermines the number of splits to be performed, thereby completely removing the need for a termination criterion.

In summary, this boils down to the simple formula

$$C(V \rightarrow \{V_L, V_R\}) = 1 + P(V_L)N_L + P(V_R)N_R,$$

which has to be evaluated for each dimension X, Y, and Z, and for each photon position inside the current voxel. The "1" is the (constant) cost for traversing the node itself. As this constant is added to each potential split plane, it doesn't affect the minimum anyway, and can be safely ignored.

3.3. Determining $P(V_L)$ and $P(V_R)$

As mentioned above, the different traversal used in kNN and RT respectively influences the probabilities of traversing the respective children. Thus, we cannot simply use the same probabilities as used in the SAH, but have to adapt our cost function. Similary to the SAH – which assumes infinitely long rays and equal ray distribution – we will in the following have to make some assumptions on the query distribution and query radius.

The query radius obviously depends on the query position, and the average query radius is not known in advance (though it could be estimated [Jen01]). In practice however most photon map implementations also specify a (tight) maximum query radius R_{max} for efficiency reasons [Jen01]. In the following, we will simply use this maximum search radius for building our kd-tree. Similarly, one could estimate the average query radius based on the volume of the voxel and the number of photons, which would even yield a different estimate for different regions of the scene. So far however, the maximum query radius has been sufficient.

For the query distribution, we simply assume an equal distribution of query locations. Then, the voxel V will be traversed by all queries that happen inside a range of R around V. We will denote all these positions – i.e. the voxel V extended by R in all directions – as $V \pm R$.

As queries can only occur on the surface of geometric primitives (at least as long as we do not consider volumetric effects), the probabilities for traversing the left and right nodes should be proportional to the surface area of all *primitives* enclosed in $V_L \pm R$ respectively $V_R \pm R$ (i.e. *not* the surface area of the *voxel*), yielding

$$P(V_L) = \frac{PSA(V_L \pm R)}{PSA(V \pm R)} \text{ and } P(V_R) = \frac{PSA(V_R \pm R)}{PSA(V \pm R)},$$

where $PSA(V \pm R)$ denotes the surface area of all geometric scene primitives overlapping $V \pm R$.

As the primitive surface area can be quite costly to compute, we ignore the variation in primitive density and instead approximate this value as

$$P(V_L) \approx \frac{Vol(V_L \pm R)}{Vol(V \pm R)} \text{ and } P(V_R) = \frac{Vol(V_R \pm R)}{Vol(V \pm R)}.$$

Note that this is quite a severe approximation. Alternatively, $PSA(V \pm R)$ could also be approximated by sampling the geometry. So far however we have only used the coarse approximation $PSA(V \pm R) \approx Vol(V \pm R)$, which in practice seems towork quite well. For efficiency reasons, we further approximate $Vol(V \pm R)$ as

$$Vol(V \pm R) \approx \Pi_{i=x,y,z}(V_{i,max} - V_{i,min} + 2R).$$

Extending the volume by this range also has some interesting, automatic side effects: First, it correctly handles very flat cells, which would get a zero cost if this term would not be used. Second, as the extension-radius is constant, its influence is high for small voxels, and small for large voxels. Thus, for large voxels (i.e. voxels much larger than the query radius) the influence of R almost vanishes, leading to an almost entirely volume-based heuristic. For small voxels however, the R term starts to dominate, in fact leading to an automatic balancing for small voxels. This in fact is fortunate, as for small voxels the probability is high that most of the photons have to be touched anyway (for which balancing is good, see above), whereas the influence of the volume for larger voxels leads to few traversal steps until the query radius is being reached.

This explanation also suggests for which cases our heuristic can be expected to be most efficient: If the query radius is very large, all the photons in this large radius have to be touched anyway, so few improvements can be expected. With a similar argument we can deduce that our VVH will be most effective for scenes with a highly uneven photon distribution, and if the (average) query radius is relatively small. Though this may not be the case for diffuse illumination, it fortunately is the case for our targeted applications, i.e. the visualization of highly detailed, sharp caustic patterns. Note that our method should also work well for the local pass acceleration technique proposed by Christensen et al. [Chr00].

3.4. Efficient Construction

Once the cost function is defined, the optimized kd-tree can be built in multiple ways. However, care has to be taken to implement this operation efficiently. For example, the naive way of recomputing N_L and N_R from scratch for each potential split plane yields a quadratic complexity in each splitting

step, which leads to intolerable construction times except for a few hundred photons.

A more efficient way of building the hierarchy is to iterate over all three potential dimensions in turn, sort all photons in the respective dimension (e.g. using quicksort), and then incrementally update N_L and N_R while iterating over the sorted photon list. Iterating over the photons costs only $O(N)$, so the complexity of each splitting step is only $O(NlogN)$ for the quicksort. Though this combines to a complexity of $O(Nlog^2N)$ for the full construction algorithm, this approach is rather simple to implement, and already much more efficient than the naive method.

3.4.1. Construction in $O(NlogN)$

Finally, there is an even more efficient – though somewhat more complex to implement – way of building the hierarchy (in spritit of [Vai89]): As each splitting plane passes through a photon, each photon p_i implies three "potential splitting planes" $X = p_{i,x}$, $Y = p_{i,y}$, and $Z = p_{i,z}$. In the beginning, we build one large set of potential splitting planes $(i,d,p_{i,d})|i = 1..N, d = X,Y,Z$, i.e. with all potential splits of all three dimensions in the same list. We then that into a list $(i_j,d_j,s_j), j = 1..3N$ such that $s_j \leq s_{j+1}$, This sorting costs $O(3Nlog3N) = O(NlogN)$, and has to be performed only once.

In order to find the best splitting plane, we then iterate through this sorted list, and – for each dimension X, Y, and Z – incrementally update the number of photon to the left of, on, or to the right of the current. As the input list is sorted by increasing split value, visiting the potential split plane $K = p_{i,k}$ simply means that in dimension k one photon goes from the right to the left side, which allows for updating N_L and N_R incrementally. For each visited potential split position, we evaluate the cost function, and track the split with minimal cost.

After having iterated through the list, the best split has been determined. We then sweep through this list a second time, and split it – without changing the order of the elements – into the list of photons to left of, and those to the right of the split plane, respectively. As we do not change the order of the elements, the two output lists will automatically remain sorted, and can each be used for the next splitting step in the left and right subtrees, respectively. Iterating twice through the sorted input list costs $O(2N)$, which for all splitting steps amounts to a total of $O(NlogN)$. As the initial sorting has to be performed only once at the beginning, we end up with an overall complexity for the entire construction algorithm of $O(NlogN)$, which is the same complexity as for balancing the kd-tree [Jen01].

4. Results and Discussion

After having described both background and implementation of our optimized kd-trees, we can now measure their impact on photon mapping performance.

4.1. Test Scenes

This impact is likely to depend on both scene and actual photon distribution: For example, rendering only a sharp, focussed caustic is likely to result in a much stronger variation in photon distribution than rendering a scene with smooth, diffuse illumination.

Therefore, in order to cover as wide a range of photon distributions as possible, we have chosen four test scenes (see Figure 1) with different photon distributions: First, the "Conference" scene contains only diffuse illumination with a relatively equal photon distribution. Second, the "GlassEgg" contains a single, concentrated caustic on the floor. Third, the "MetalRing" contains several sharp caustics scattered all over the scene. Finally, as a practical example we have chosen to also include the "HeadLight" scene [BWDS02], in which the illumination from a (real) car headlight is being simulated. The curved reflector and front glass in this scene result in very detailed, high-quality caustic patterns that have to be simulated with 2.3 million photons in order to achieve reasonable quality. Even more photons would be desirable, but could not reasonably be tested on the machine used for the experiments. Using the combination of fast photon mapping and interactive ray tracing, the goal was to interactively visualize these illumination patterns, at least for a static scene.

Note that thought these test scenes also cover a wide range of geometric complexity (from 1036 triangles in the MetalRing scene, to 280k triangles in the Conference scene), the geometric complexity of the scene is less of an issue: The query time itself is totally independent of the kind and number of geometric primitives [Jen01], and the ray tracing time also depends but weakly on the scene complexity [WPSBS03].

4.2. Construction Time

Even though we are mainly interested in photon-mapped walkthroughs with a precomputed photon map (for which the construction time is less of an issue), the applicability of our method also depends on the generation cost.

As can be seen from Table 1, building our VVH kd-tree often is considerably slower than building the left-balanced tree. This is true even for the $O(NlogN)$-algorithm presented in Section 3.4. This higher cost – at the same computational complexity – is due to the need for evaluating the cost function for each potential split plane in each splitting step, which is not the case for the balanced code.

Note however that the measurements for left-balancing the kd-tree have been performed using a highly optimized implementation [Wal99] that is roughly 2–4 times as fast as the original code published in [Jen01]. Even compared to this highly optimized implementation, our overhead for realistic numbers of photons is less than a minute, which is

Scene	N	balanced	VVH	overhead
Conference	152k	0.17	2.98	17.5
	2.1M	4.14	53.5	12.9
GlassEgg	96.8k	0.41	1.81	4.41
	323k	3.81	6.73	1.76
	1.9M	95.2	47.6	0.49
MetalRing	156k	0.20	3.08	19
	2.1M	9.91	56.7	5.7
HeadLight	2.3M	74	66	0.88

Table 1: *Construction time (in seconds) for the different kd-trees in our example scenes. As can be seen, the balanced version is usually faster in construction, but the total construction time for the VVH is still tolerable when compared to the time for shooting the photons, or for rendering an entire frame. When using many photons, the VVH sometimes is even faster than left-balancing the kd-tree.*

quite tolerable for offline-rendering or walk-through applications. For certain models with large numbers of photons, our code surprisingly is even faster than balancing the tree.

Additionally, in contrast to this highly optimized balancing implementation, our attention on the VVH so far has focussed on the design of the cost function, and not on an efficient implementation. We believe there is plenty of room left for reducing the construction time of our method. Finally, the time for building our optimized kd-tree is usually in the same range as the time required for shooting the photons.

4.3. Reduction of the Number of Traversal Steps

As mentioned before, the actual goal of using a cost function for unbalancing the kd-tree was to reduce the number of traversal steps, i.e. the number of photons touched during the nearest-neighbor query. As can be seen in Table 2, our optimized kd-trees can significantly reduce the number of these traversal steps, especially for scenes with a varying photon density, and for queries with a small average query radius. However, even in the conference scene – with a relatively uniform photon distribution and with a very large radius for getting a smooth approximation of the diffuse illumination – we still achieve some gains of roughly 30%. Also as expected, the VVH outperforms the balanced version especially for sharp caustics, where it even leads to improvements by a factor of 2–6.

Note that some of these gains would vanish once larger k and R would be used during rendering. This however would hardly be used in practice, as the caustic then would be significantly blurred out.

4.4. Impact on Radiance Estimate Performance

As the cost for the radiance estimate is nearly linear in the number of nodes visited during traversal, the savings outlined in the previous section translate to similar savings in the radiance estimate: As can be seen in Table 3, the number of queries per second can be increased by up to 3.4, depending on the scene and rendering parameters.

Scene	N/k	photons	bal.	VVH	factor
Conference	152k/150	diffuse	366	296	1.23
	2.1M/80	diffuse	371	259	1.43
GlassEgg	96.8k/40	caustic	59.4	10.7	5.55
	323k/100	caustic	119	20.5	5.80
	1.9M/100	caustic	97.5	16.8	5.80
MetalRing	156k/100	caustic	110	54.6	2.01
	2.1M/100	caustic	125	57.0	2.19
HeadLight	2.3M/50	caustic	275	124	2.22

Table 2: *Average number of traversal steps for finding the k nearest photons. Using our optimized kd-tree reduces the number of visited photons by 1.2–5.8, depending on the scene and settings. In particular for the caustic scenes, we achieve improvements of 2x–6x. Note that the average is smaller than k, as the maximum query radius will avoid most traversal steps in regions where no caustic is present.*

Note however that the radiance estimate also includes several cost factors that can not be affected by our method. For example, once all photons are found, the radiance estimate requires to filter the resulting k photons. While this cost is already included in the measurements in Table 3, we have used only a simple box filter in our experiments. More costly filters (see e.g. [Jen01]) may reduce the performance impact.

Scene	N/k	balanced	VVH	speedup
Conference	152k/150	46k	57k	1.24
	2.1M/80	43k	59k	1.37
GlassEgg	96.8k/40	756k	2586k	3.40
	323k/100	299k	1031k	3.44
	1,9M/100	318k	1074k	3.37
MetalRing	156k/100	212k	357k	1.68
	2.1M/100	170k	302k	1.70
HeadLight	2.3M/50	75k	139k	1.85

Table 3: *Number of kNN queries on a 2.2GHz Pentium-IV CPU, using both our optimized BSPs (VVH) and the left-balanced version with different parameters. As can be seen, we achieve significant speedups of 1.7–3.4, especially for highly varying photon distributions and highly localized queries (i.e. $k \ll N$).*

As mentioned in the previous section, savings will deteriorate for larger k, and improve for smaller k. Though an excessively large k can in some scenes result in very small

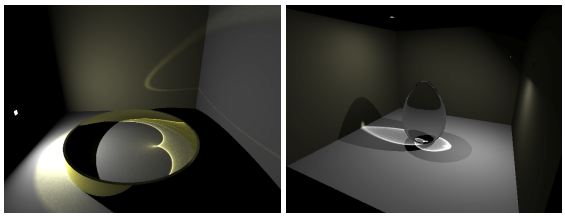

Figure 2: *The MetalRing and GlassEgg scenes, rendered with both direct illumination (one sample per light source) and highly detailed caustics (both using $N = 2M$, $k = 100$, $R = 0.05$). Using our voxel volume heuristic, we increase the number of radiance estimates by roughly a factor of 1.7 and 3.4, respectively. Even including all other cost for computing shadows, reflections, and refraction (which can not be accelerated with out method), this amounts to a performance increase of 50% and 70%, respectively.*

Figure 3: *Direct visualization of diffuse photons in the conference scene, with (left:) very wide queries using $N = 152k$, $k = 150$ (and correspondingly significant blurring), and (right:) very localized queries using $N = 2M$, $k = 80$ (correspondingly quite noisy). Whereas the speedup for the large query radius is only 19%, the small query radius yields an improvement of 35%. This is particularly interesting since we did not expect any gains at all for as equally distributed diffuse photons.*

savings, so far we have never found a setting in which our method performed worse than the balanced version.

4.5. Impact on Final Rendering Performance

So far, we have only measured the performance impact on query times and radiance estimate. In practice however systems using the photon map also spend parts of their time on other tasks, such as on ray tracing, on calculating direct illumination [Jen01, SWZ96], on sampling and computing BRDFs, and on final gathering or irradiance caching [WH92, DBB03]

Obviously, the less time the renderer spends in kNN queries, the smaller the (relative) performance advantage of our method will be. This is especially true for renderers that use photon mapping mainly for diffuse illumination, and which make extensive use of final gathering and irradiance caching [WH92, DBB03]. These applications will not only spend only a small fraction of their total rendering time in kNN queries, they will also use relatively "wide" queries (with small N and large k) in the final gather, thereby limiting the impact of our method. This can for example be seen in the conference scene (see Figure 3 and Table 4), in which the final performance impact of our method is only in the range of 19–35%, even without shooting any shadow rays.

For our target application of visualizing high-quality caustics however our method is well suited: As we can compute both direct and indirect diffuse illumination in more efficient ways [WPSBS03, BWS03], we are mainly interested in a fast and high-quality visualization of the caustic photon map. This also implies a highly varying photon distribution and very small query radii in order to reproduce fine caustic details.

For this scenario, our method can lead to significant performance gains, as shown in Table 4: For example, in the

Scene	N/k	balanced	VVH	speedup
Conference	152k/150	0.118	0.141	1.19
	2.1M/80	0.108	0.146	1.35
GlassEgg	96.8k/40	0.88	1.23	1.40
	323k/100	0.61	1.01	1.66
	1.9M/100	0.62	1.02	1.65
MetalRing	156k/100	0.48	0.71	1.48
	2.1M/100	0.40	0.62	1.55
Headlight	2.3M/50	0.24	0.44	1.83

Table 4: *Frame rate (in frames per second) when visualizing a precomputed photon map in several test scenes (see Figure 1), using a single 2.2GHz Pentium IV CPU at 640x480 pixels. As can be seen, the kd-trees built using our voxel volume heuristic (VVH) achieve significant performance increases in the range of 20–80%, even including all other cost, e.g. for shooting primary, secondary and shadow rays. For the HeadLight scene for example, this translates to 4–5fps using 5 dual-CPU PCs.*

GlassEgg and Metal Ring scenes (see Figure 2), we achieve speedups of 40–66 percent, even including the cost for computing shadows, reflections and refraction, which cannot be accelerated by faster kNN queries. Furthermore we currently do not use the fast SSE code [WPSBS03] for shooting these rays, so the ray shooting cost already starts to dominate the rendering time, especially after the kNN queries have been accelerated by factors of 1.7 and 3.4, respectively. Once we reactivate the fast SSE code for tracing the rays, the (relative) speedup of our faster queries will be even higher.

In the HeadLight scene (see Figure 4), some 2.3 million photons were required to sufficiently simulate the detailed caustic pattern of the lamp. As this caustic is relatively spread out, some noise remains, and even more photons would have been beneficial. In this scene, using the optimized kd-tree allowed for improving the frame rate (using a single 2.2GHz Pentium-IV CPU) from 0.24 to 0.44 frames

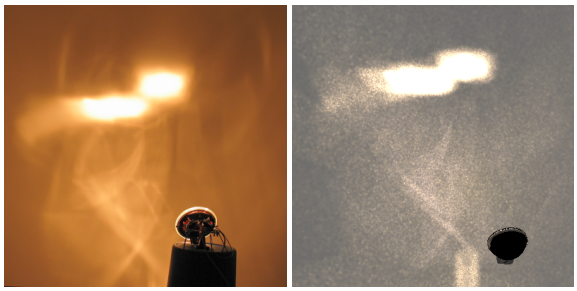

Figure 4: *Simulation of a caustic from a (real) car headlight, which can be inspected interactively. Left: In order to reproduce the detailed patterns of the spread-out caustic, 2.2 million photons have been used. Using our optimized kd-trees, the frame rate could be improved from 0.24 to 0.44 frames per second (on a single CPU). Right: a photo of the real lamp, for comparison.*

per second. Using 5 dual-CPU PCs, this corresponds to 4–5 frames per second at video resolution even for this highly detailed caustic.

5. Summary and Conclusions

In this paper, we have presented a method that takes some of the ideas from building fast BSPs for ray tracing, and applied them to build good kd-trees for photon mapping.

We have shown that our optimized kd-trees can outperform the currently used balanced ones by a factor of 1.3–3.5, depending on the actual photon map parameters (e.g. average query radius and number of photons) and on the actual photon distribution. Even if a considerable amount of time is spent in other tasks – such as tracing shadow rays or computing reflections and refractions – we can still achieve significant speedups of 40 to 66 percent. In the HeadLight example, where most of the time is spent in kNN queries, we achieve a speedup of 83%.

Moreover, our method is totally orthogonal to all other techniques typically used for efficient photon mapping and fast global illumination (at least if these don't require modifications to the BSP tree themselves), and can be combined with *any* photon map based renderer without major changes and without imposing any restrictions or approximations.

5.1. Future Work

As discussed above, the only operation where our method leads to an actual cost increase is the construction time, where our method is often considerably more costly than building a left-balanced tree. As we did not consider this as crucial, so far this issue has been neglected, and should be addressed as future work.

More importantly, it seems interesting to investigate whether changes to the cost functions can help in further improving the performance of the kNN queries. For example, as already discussed in Section 3.3, photon queries can only happen on the surface of geometric primitives. Thus, the query density is by no means uniform (as assumed in our cost function), but is closely related to the density of geometric primitives. This should be somehow integrated into the cost function.

Since the memory layout of the kd-tree is no longer fixed as in the pointer-less version, changing the memory layout of the kd-tree to a more cache-friendly way [Hav99] may also yield further performance gains.

Finally, it seems interesting to investigate the use of SIMD instructions [Int02] for accelerating the query itself. Except for a data-parallel approach of performing four independent queries in parallel (as in [WSBW01]), an alternative approach is to reorganize the kd-tree such that it always stores four neighboring photons in each node. This might eventually include to entirely switch from a kd-tree to a ray-tracing style BSP tree that stores photons (in multiples of four) in the leaves, and to quickly traverse this BSP using SSE.

Acknowledgements

We would like to thank Hella Corp for their headlight model, and Vlastimil Havran for helpful discussions. Finally, this work has been supported by Intel and *in*Trace Corp.

References

[Ben75] BENTLEY J. L.: Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM 18*, 9 (1975), 509–517. 2

[BWDS02] BENTHIN C., WALD I., DAHMEN T., SLUSALLEK P.: Interactive Headlight Simulation – A Case Study of Distributed Interactive Ray Tracing. In *Proceedings of the 4th Eurographics Workshop on Parallel Graphics and Visualization (PGV)* (2002), pp. 81–88. 5

[BWS03] BENTHIN C., WALD I., SLUSALLEK P.: A Scalable Approach to Interactive Global Illumination. *Computer Graphics Forum 22*, 3 (2003), 621–630. (Proceedings of Eurographics). 1, 7

[Chr00] CHRISTENSEN P. H.: Faster Photon Map Global Illumination. *Journal of Graphics Tools 4*, 3 (Apr. 2000), 1–10. 4

[DBB03] DUTRE P., BEKAERT P., BALA K.: *Advanced Global Illumination*, 1st ed. A K Peters, July 2003. ISBN: 1568811772. 7

[Hav99] HAVRAN V.: Analysis of Cache Sensitive Representation for Binary Space Partitioning Trees. *Informatica 23*, 3 (May 1999), 203–210. ISSN: 0350-5596. 8

[Hav01] HAVRAN V.: *Heuristic Ray Shooting Algorithms*. PhD thesis, Faculty of Electrical Engineering, Czech Technical University in Prague, 2001. 3, 4

[HKRS02] HURLEY J. T., KAPUSTIN A., RESHETOV A., SOUPIKOV A.: Fast Ray Tracing for Modern General Purpose CPU. In *Proceedings of Graphicon* (2002). Available from http://www.graphicon.ru/-2002/papers.html. 3, 4

[HSA91] HANRAHAN P., SALZMAN D., AUPPERLE L.: A Rapid Hierarchical Radiosity Algorithm. In *Computer Graphics (Proc. of ACM SIGGRAPH)* (1991), pp. 197–206. 1, 2

[Int02] INTEL CORP.: Intel Pentium III Streaming SIMD Extensions. http://developer.intel.com/vtune/cbts/-simd.htm, 2002. 8

[Jen96] JENSEN H. W.: Global Illumination using Photon Maps. *Rendering Techniques* (1996), 21–30. (Proceedings of the 7th Eurographics Workshop on Rendering). 1, 2

[Jen97] JENSEN H. W.: Rendering Caustics on Non-Lambertian Surfaces. *Computer Graphics Forum 16*, 1 (1997), 57–64. 1, 2

[Jen01] JENSEN H. W.: *Realistic Image Synthesis Using Photon Mapping*. A K Peters, 2001. ISBN 1-56881-147-0. 1, 2, 4, 5, 6, 7

[Kaj86] KAJIYA J. T.: The Rendering Equation. In *Computer Graphics (Proceedings of ACM SIGGRAPH)* (1986), Evans D. C., Athay R. J., (Eds.), vol. 20, pp. 143–150. 1, 2

[Kel97] KELLER A.: Instant Radiosity. *Computer Graphics (Proceedings of ACM SIGGRAPH)* (1997), 49–56. 1, 2

[LW93] LAFORTUNE E., WILLEMS Y.: Bidirectional Path Tracing. In *Proc. 3rd International Conference on Computational Graphics and Visualization Techniques (Compugraphics)* (1993), pp. 145–153. 1, 2

[MB90] MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *Visual Computer 6*, 6 (1990), 153–65. 3

[MM02] MA V. C. H., MCCOOL M. D.: Low Latency Photon Mapping Using Block Hashing. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware* (2002), pp. 89–99. 2

[PDCJH03] PURCELL T. J., DONNER C., CAMMARANO M., JENSEN H. W., HANRAHAN P.: Photon Mapping on Programmable Graphics Hardware. In *Proceedings of the ACM SIGGRAPH/Eurographics Conference on Graphics Hardware* (2003), pp. 41–50. 2

[SAG94] SMITS B., ARVO J., GREENBERG D.: A Clustering Algorithm for Radiosity in Complex Environments. In *Proceedings of ACM SIGGRAPH* (1994), pp. 435–442. 1, 2

[San02] SANTALO L.: *Integral Geometry and Geometric Probability*. Cambridge University Press, 2002. ISBN: 0521523443. 3

[Sub90] SUBRAMANIAN K. R.: *A Search Structure based on K-d Trees for Efficient Ray Tracing*. PhD thesis, The University of Texas at Austin, Dec. 1990. 3

[SWZ96] SHIRLEY P., WANG C., ZIMMERMAN K.: Monte Carlo Techniques for Direct Lighting Calculations. *ACM Transactions on Graphics 15*, 1 (1996), 1–36. 7

[Vai89] VAIDYA P. M.: An O(N log N) Algorithm for the All-Nearest-Neighbors Problem. *Distrete and Computational Geometry*, 4 (1989), 101–115. 5

[VG94] VEACH E., GUIBAS L.: Bidirectional Estimators for Light Transport. In *Proceedings of the 5th Eurographics Workshop on Rendering* (1994), pp. 147 – 161. 1, 2

[VG97] VEACH E., GUIBAS L.: Metropolis Light Transport. In *SIGGRAPH 97 Conference Proceedings* (Aug. 1997), Whitted T., (Ed.), Annual Conference Series, ACM SIGGRAPH, Addison Wesley, pp. 65–76. 1, 2

[Wal99] WALD I.: *Photorealistic Rendering using the Photon Map*. Master's thesis, Numerical Algorithms Group, University of Kaiserslautern, Germany, Sept. 1999. 5

[Wal04] WALD I.: *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University, 2004. Available at http://www.mpi-sb.mpg.de/~wald/PhD/. 1, 3

[War92] WARD G.: Real Pixels. In *Graphics Gems II*, Arvo J., (Ed.). Academic Press, 1992. 2

[WH92] WARD G. J., HECKBERT P.: Irradiance Gradients. In *Third Eurographics Workshop on Rendering* (1992), pp. 85–98. 7

[WKBKS02] WALD I., KOLLIG T., BENTHIN C., KELLER A., SLUSALLEK P.: Interactive Global Illumination using Fast Ray Tracing. *Rendering Techniques* (2002), 15–24. (Proceedings of the 13th Eurographics Workshop on Rendering). 1, 2

[WPSBS03] WALD I., PURCELL T. J., SCHMITTLER J., BENTHIN C., SLUSALLEK P.: Realtime Ray Tracing and its use for Interactive Global Illumination. In *Eurographics State of the Art Reports* (2003). 1, 3, 5, 7

[WS03] WAND M., STRASSER W.: Real-Time Caustics. *Computer Graphics Forum 22*, 3 (2003), 611. (Proceedings of Eurographics). 2

[WSBW01] WALD I., SLUSALLEK P., BENTHIN C., WAGNER M.: Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum 20*, 3 (2001), 153–164. (Proceedings of Eurographics). 1, 8