Next Event Backtracking

Jendersie, Johannes
TU Clausthal, Germany

September 2, 2019

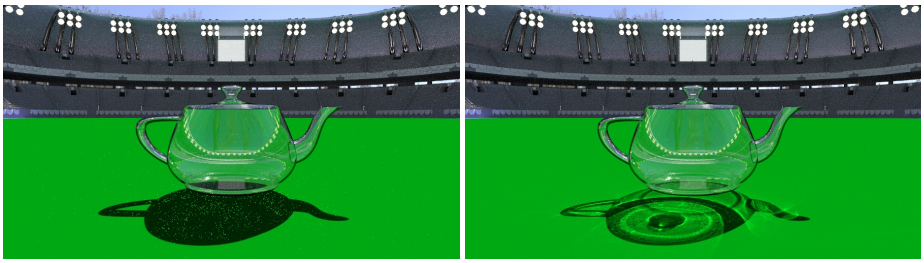


Figure 1: Bidirectional Path Tracing (left, 212 spp) compared to Path Tracing with Next Event Backtracking (341 spp, equal time 15 min). Bidirectional Path Tracing and many more methods fail to produce the caustic in this scenario, because the light paths rarely hit the small teapot. Our method implicitly guides the light paths to the important regions and scales well for large scenes and many lights.

Abstract

In light transport simulation, challenging situations are caused by the variety of materials and the relative length of path segments. Path Tracing can handle many situations and scales well to parallel hardware. However, it is not able to produce paths which have a smooth surface in connection with a small light source. Here, photon transports perform superior, which can be ineffective if the smooth object is small compared to the scene size.

We propose to use the last segment of a Path Tracer path as the first segment of a photon path. As a result, the strengths of next event estimation are inherited by the photon transport and photons are guided toward the regions where they are most useful. To that end, we developed a lock-free sparse octree, which we use for fast and robust density estimates. Summarizing, the new method can outperform state of the art algorithms like Vertex Connection and Merging in certain scenarios.

1 Introduction

In stochastic light transport simulation there are three basic operators to create light paths: *random walks*, *connections* and *merges*. Random walks start at the observer or a light source and use ray tracing to find the next intersection point (vertex) in a randomly sampled direction. If they hit a surface with the adjoint quantity, a full light transport path is found. Random walks have a high chance for finding contributions if there are large light sources and/or the path to the light source is close to deterministic (for example, if the emitter is directly visible or reflected by mirrors).

Connections can be created from each vertex of a random walk path. By choosing a random point on an adjoint surface, a contribution can be computed over arbitrary large distances. This is called *Next Event Estimation* (NEE), because the next event in the random walk could be the very same location. The strength of NEE is that it can find light sources with a very small solid angle, which have a much smaller probability to be found via random walk alone. Additionally, it is possible to use binary trees to select connection points proportional to their expected contribution to reduce the variance.

Finally, merges combine the vertices of two random walks by assuming they hit the same point, although they might be separated by a small distance. That means, if the end points of two paths have a distance smaller than some threshold, they are combined to a full contribution path. In general, merges have one more random sampling event than random walks and connections, leading to a higher variance in the first place. However, their strength comes from the fact that neighbor points can be searched in the large set of vertices over all paths (often millions). Merges are the only really successful operator to find the difficult specular-diffuse-specular (SDS) paths.

We combine connections and merges by using the connection segment as the first path segment of a new light walk. Thus, merges inherit the strengths of next event estimation. Therefore, the vertices, from which the connections were initiated, are transformed into photon-emitting virtual light sources. This requires an estimate of the density of such NEE vertices. For that purpose, we introduce a new octree-based data structure which is faster than kd-tree-based neighborhood searches. Independent of the used data structure, every density estimate over a finite area will be biased and noisy, leading to a small bias in our new method.

2 Related Work

The foundation of our method is the *Path Tracing* (PT) algorithm which goes back to Kayjia [Kaj86]. Today, it is widely used in production renderers [FHF*17] due to its simplicity and extensibility. It combines the two operators random walk and NEE by weighting the two independent estimates with respect to their effectiveness for each individual path. This combination of two or more samplers for the same quantity is called *Multiple Importance Sampling* (MIS) in Monte

Carlo-based simulations. In graphics, the balance or power heuristic are used to compute the weights. They were introduced by Veach and Guibas [VG95b] and extended to a large set of operators since then.

PT can handle many situations, but fails for caustics and SDS paths. *Bi-directional Path Tracing* (BPT) [VG95a, LW93] is a stronger method. It uses a random walk from the observer and another one from the light source, and computes all connections between the vertices of the two paths. It is able to find caustics, but still fails for SDS paths.

The only successful methods to efficiently find SDS paths are based on photon transports. *Photon mapping* [Jen96] is able to produce caustic and SDS paths, but has problems with the scene scale. Large scenes or small specular objects as well as many light sources can lead to highly noisy results. Jensen already used a dedicated *Caustic Map* where photons are explicitly send into directions of specular objects. This will still fail if there are many large specular objects and requires a generalization for glossy objects. Contrarily, NEB shoots photons into important regions independent of the material.

Similarly, the first importance based method to distribute photons was introduced by Peter and Pietrek [PP98]. It uses piecewise constant functions, created from the importance information of a view path tracing pass, to distribute photons. Due to this piecewise approximation the method has problems with highly glossy surfaces. The major difference to NEB is that the guidance in NEB will happen implicitly.

A different approach to improve the photon map quality is to stir the deposition of photons. Suykens and Willems [SW00] fixed the photon mapping density to a constant, which reduces the number of photons in bright regions. Likewise, Keller and Wald [KW00] used an importance map to control the density of stored photons. However, both methods are still sampling a large number of photons paths and only reduce the number of stored photons, wasting the others.

Georgiev et al. [GKDS12] and Hachisuka et al. [HPJ12] simultaneously introduced the MIS-weight computation to successfully combine BPT with merges. The *Vertex Connection and Merging* (VCM) algorithm is one of the most robust methods so far. Still, it may fail for selected situations, where the MIS-weight underestimates the variance of certain merge events. This issue was overcome by Jendersie et al. [JG18, Jen19] with a small change to the heuristic. While we only demonstrate our novel operator – the next event backtracking (NEB) – in Path Tracing, it is possible to integrate it into BPT or VCM, too. However, NEB is already capable of handling many complex light situations without doing so.

Another important variant of the random walk operator is the *Markov Cain Monte Carlo* (MCMC) sampling. Instead of generating independent random sampling events, MCMC samplers apply small mutations to the paths. Then, the new mutations are randomly discarded or accepted with respect to a target function. This allows MCMC samplers to generate distributions of an unknown function to reduce noise opposed to the naïve walk. For an overview of MCMC methods we refer to the survey of Šik et al. [ŠK18]. It is thinkable to use NEB

in connection with MCMC random walks, which we leave as future work.

Manifold Next Event Estimation (MNEE) from Hanika et al. [HDF15] is the most similar method to ours. There, random connections, which are blocked by refractive surfaces, are iteratively moved on the surface until they form a valid path. This iteration requires multiple expensive evaluations and shadow tests and is biased in case there is an ambiguity of multiple possible lights paths. Finally, MNEE is not able to find caustics from mirrors and prisms, where the reflection surface is not blocking the direct connection between the caustic and the light emitter. NEB shares the weakness in that scenario, but at least handles it to a better degree than MNEE. We believe that NEB is more robust than MNEE in general.

Other methods which target the problem of small caustic throwing objects are based on guidance. A general guidance method like the method from Müller et al. [MGN17] reduces the overall variance in all sampling events by steering the samplers into the direction of the adjoint quantity. A method which explicitly targets the exploration of visible caustics was invented by Hachisuka and Jensen [HJ11]. It uses MCMC sampling for the light paths with the visibility of caustics as target function. Recently, Grittman et al. [GPGSK18] improved the caustic handling in large scenes by restricting the use of photons adaptively and learning a guidance information at the light emitter. Other than these methods our NEB does not learn those connections over time. Instead, it samples these cases explicitly with a much higher density.

3 Next Event Backtracking

The basic idea is simple: whenever a path vertex is connected to a light source, we use this very same connection as the first segment of a photon path by creating a virtual light source at the path vertex. This, however, has several complex implications and a lot of potential for possible modifications. The outline of the algorithm (Figure 2) is as follows:

1. Trace paths as in *Path Tracing*
   (a) Store the hit-points (called *NEE vertex*)
2. NEE and photon tracing
   (a) Estimate virtual light source density
   (b) Compute NEEs
   (c) Trace photons and apply contribution directly
   (d) [Optional] Trace photons from the light source as usual
3. Compute self emittance contributions

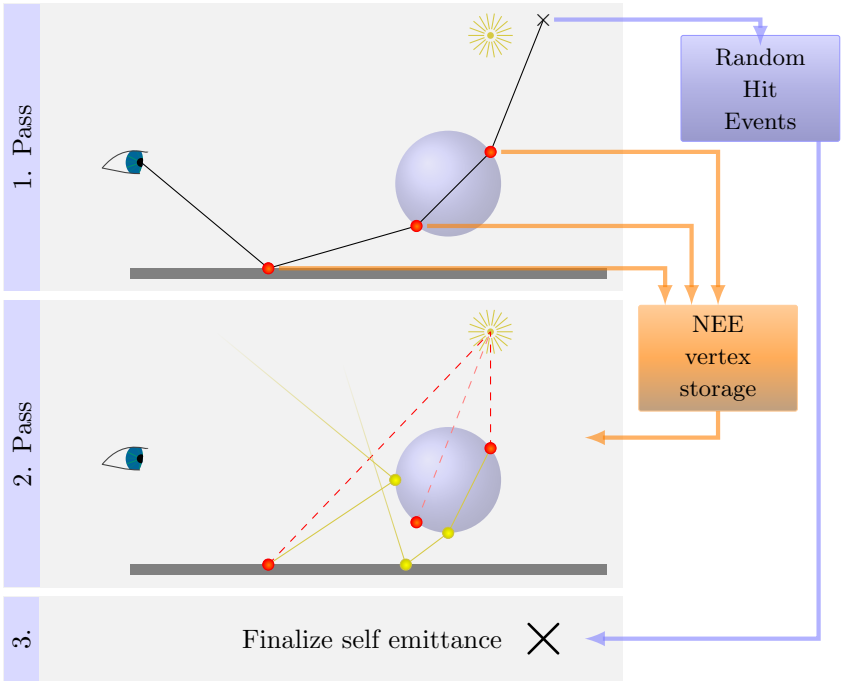


Figure 2: Algorithm outline: In the first pass a PT is executed and the intermediate results are stored. In the second pass photons are traced from the non-zero-estimate vertices from pass one. Finally the contributions from random hits, photons and NEE are weighted to compute the final contribution.

3.1 Trace View Paths

For tracing paths we use a conventional Path Tracer. However, it is not possible to compute the results for random hits and NEE immediately. To calculate the MIS weights, it is necessary to know the photon events which itself depend on the density of NEE vertices (which are being created in this pass). For this reason the intermediate results for the events must be stored.

3.2 NEE and Photon Tracing

Our overall goal is to transform the stored vertices into virtual light sources which emit photons. To transform the incident differential irradiance (unit $\mathrm{W\,m^{-2}}$), which is available through NEE, into a flux Φ (unit W) we need the density of source vertices. Effectively, we need to know the area A of the virtual light source to integrate the incoming irradiance:

$$\Phi = \mathrm{d}E \cdot \mathrm{d}A = \frac{\mathrm{d}E}{\rho}. \tag{1}$$

where ρ is the density of NEE vertices at the current virtual light source. We can insert $\mathrm{d}A = 1/\rho$ because the area, which is associated with each vertex, depends on this density: If there are more vertices, each one represents a smaller part of the surface area.

Note that it is not important how the vertices were generated. It only matters how many vertices are stored in a local area to turn each of them into an unbiased[1] emitter. Especially, the past (sampling events, Russian Roulette, ...) of the view path vertices is not important.

So, in step (a) the density of vertices must be computed at each vertex. This can be achieved with k-nearest neighbor searches or and additional data structure. We use a sparse octree to integrate density over regions and time, as will be detailed in Section 5. The advantage of the octree is a higher performance and less noise in the estimates.

Now (step (b)), we can compute a next event estimation for each of the stored vertices and compute its contribution

$$L_E = w_{E,k} \cdot \tau(\nu_k) \cdot f(\nu_k, \mathbf{d}) \cdot \mathrm{d}E \tag{2}$$

with

$$\begin{aligned}
\mathbf{d} &= \text{Direction of NEE} \\
\mathrm{d}E &= \text{Differential irradiance of NEE} \\
\nu_k &= \text{Vertex with index } k \text{ (camera has index 0)} \\
\tau(\nu) &= \text{Path throughput from MC sampling } \prod_{i=0}^{k} \frac{f_i}{p_i} \\
f(\nu, \mathbf{d}) &= \text{Bidirectional Scattering Distribution Function (BSDF)} \\
w_{E,k} &= \text{MIS weight (see Section 4).}
\end{aligned}$$

[1] Unbiased, if we know the true density, which is not the case

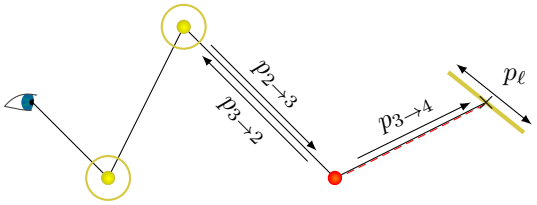


Figure 3: MIS-weight computation between several events along the same path. The shown example path has length $\ell = 4$.

It must be added to the pixel, in which the path originated. The details to compute the MIS weight $w_{E,k}$ are given in the next section.

Photons can be traced (step (c)) by starting a new random walk at the current vertex, after applying Equation (1). The first sampling event is based on the NEE connection direction $\mathbf{d}$ as incident direction and the vertex's BSDF. It is not necessary to store photons, because we can invert the search for merge events. Instead of searching for photons around the view path vertices, we can as well search for view path vertices around photons, which produces identical results. Therefore, the NEE vertex storage must be some kind of spatial data structure to support neighborhood searches.

Then the contribution of a photon Φ_i at ν_i from direction $\mathbf{d}_i$ for a found vertex ν_k is

$$L_P = w_{P,k,i} \cdot \tau(\nu_k) \cdot f(\nu_k, \mathbf{d}_i) \cdot \Phi_i \cdot K(\|\nu_k - \nu_i\|) \tag{3}$$

where details on the weight will follow in the next section, again. The kernel K can be the uniform kernel $1/\pi r^2$ for the search radius r or any other kernel used in photon mapping.

Optionally, it is possible to trace photons from the light source (step (d)) and add their contributions the same way as the photons from NEE vertices. Only the MIS weights must be adapted accordingly. We will show in section 6 that adding those photons complements NEB where it is weakest.

3.3 Compute Self Emittance Contributions

Since it was not possible before step (2.a) to compute the MIS weights properly, we needed to store random hits of light source in pass 1. Now, we can iterate over the stored events and compute the contributions

$$L_e = w_e \cdot \tau(\nu_k) \cdot L_e(\nu_k) \tag{4}$$

4 MIS Weights

In our basic algorithm there are three types of events which must be weighted against each other. When conventional photons are traced too, there is one

additional event type. Each path of length ℓ has a single random hit event, one NEE event if $\ell \geq 2$ and $\max(0, \ell - 2)$ photon merge events as depicted in Figure 3. Optionally, there are $\max(0, \ell - 1)$ merges with usual photons.

In general, MIS-weights can be computed with the balance heuristic [VG95b] which reads

$$w_j = \frac{n_j p_j^*}{\sum_i n_i p_i^*} \tag{5}$$

where p^* are the path sampling densities of the different events, with respect to the area measure, and n are the number of samples drawn with the respective sampler.

For random hits we have $n_e = 1$ and

$$p_e^* = \prod_{i=0}^{\ell-1} p_{i \to i+1} \tag{6}$$

for the path density up to the vertex k. The *Probability Density Functions* (PDFs) $p_{i \to i+1} = p_i \cdot \cos\theta / d^2$ (unit m^{-2}) describe the probability to reach vertex ν_{i+1} via random sampling of the BSDF at vertex ν_i with the PDF p_i.

The path density for NEEs is defined as

$$p_E^* = p_\ell \prod_{i=0}^{\ell-2} p_{i \to i+1} \tag{7}$$

where p_ℓ is the sampling density per area to sample the specific point on the light source when attempting a random connection. The number of NEE samples is usually but not necessarily $n_E = 1$.

Next, we need the PDF for the new photon samplers. It is closely related to the NEE PDF in Equation (7) because each photon path starts with a next event estimation and therefore has to use p_ℓ. From there, a random walk in backward direction is performed up to the merge vertex ν_k. Further, it consists of another random walk beginning at the observer. Finally, we need the chance for a successful merge $\rho_{\ell-1} \cdot \pi r^2 / n_T$ with n_T being the total number of photon path starting points, i.e. the number of stored NEE vertices. Together this gives

$$p_{P,k}^* = p_\ell \cdot \frac{\rho_{\ell-1} \cdot \pi r^2}{n_T} \cdot \prod_{i=k}^{\ell-2} p_{i+1 \to i} \cdot \prod_{i=0}^{k-1} p_{i \to i+1} \tag{8}$$

and $n_P = n_T$ for the sampler count because we reuse photons from all paths. The above merge chance is derived as following: First, it must be proportional to the size of our search region πr^2. The larger the search region, the larger the chance to find something. Second, we need the probability density per area to start the respective photon random walk. The density $\rho_{\ell-1}$ is the one we computed in step (2.a). It is the total number of events per area at this start vertex. Thus, $\rho_{\ell-1}/n_T$ is the PDF per area of a single sample. Both together give the chance to find the chosen photon sub-path.

Finally, we have

$$p^*_{LP,k} = p_\ell \cdot \pi r^2 \cdot \prod_{i=k}^{\ell-1} p_{i+1\to i} \cdot \prod_{i=0}^{k-1} p_{i\to i+1} \tag{9}$$

for the conventional photons [GKDS12, HPJ12]. The number of samples is the number of additional photon paths n_{LP} due to global reuse of photons or $n_{LP} = 0$ if disabled.

Plugging Equations (6) to (9) and the sampler counts into the balance heuristic (5) gives us the searched weights:

$$w_e = \frac{p^*_e}{p_{\text{sum}}} \qquad w_E = \frac{n_E \cdot p^*_E}{p_{\text{sum}}} \qquad w_{P,k} = \frac{n_P \cdot p^*_{P,k}}{p_{\text{sum}}} \qquad w_{LP,k} = \frac{n_{LP} \cdot p^*_{LP,k}}{p_{\text{sum}}}$$

$$\text{with } p_{\text{sum}} = p^*_e + n_E \cdot p^*_E + n_P \sum_{k=1}^{\ell-2} p^*_{P,k} + n_{LP} \sum_{k=1}^{\ell-1} p^*_{LP,k} \tag{10}$$

5 Density Estimation

An important point for the correctness and performance of the NEB operator is the estimate of the density ρ (required in Equation (1)). Density estimation is a well explored research area for which we refer to the book of Silverman [Sil86] and the survey of Sheather [She04]. Unfortunately, the density estimation becomes the bottleneck of the algorithm fast and the choice of the data structure is very important.

Our first approach was to use a hash-grid to query the number of photons with a predefined search radius. This is referred to as naïve estimator in Silverman's book and has a fundamental drawback: Its bias gets very large if the distribution of NEE vertices is irregular. In low density regions (less than one vertex per search region) the estimate will always find the query point, but likely no others. Therefore, it may overestimate the density by an unbounded factor. On the other hand, in high density regions it will blur the density function more than necessary.

Our second approach was to use a kd-tree [Ben75] to estimate the density with the *k-nearest neighbor* approach. The nearest neighbor approach scales much better with irregular densities. We found that the bias was acceptable with a $k \geq 4$ neighbors. However, the kd-tree maintenance and query time dominated our rendering time vastly. We did not try to use faster builder implementations like the ParKD method from Choi et al. [CKL*10], because they would not help to improve the query time performance.

5.1 The Density Octree

To improve performance, we implemented a dedicated data structure for the density estimate. Other than the hash-grid or the kd-tree, this new data structure is not able to find the actual vertices because we store particle counts

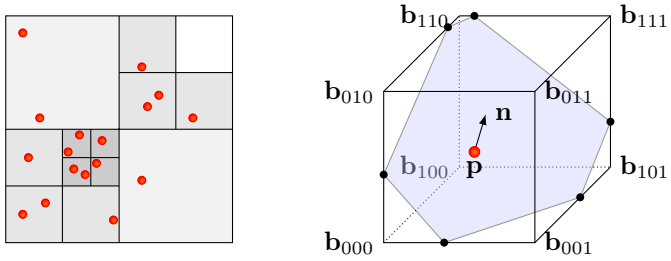


Figure 4: Concepts of the density octree. Left: a sparse quadtree with a uniform count of particles per cell. Right: one possible case for the intersection area between plane and cube.

explicitly. Therefore, it achieves a speedup of 3000× opposed to the kd-tree while having a comparable bias.

The idea is to use a sparse octree which stores an atomic counter per cell. Whenever a vertex is created, it increases the counter of the cell in which it lies by one. If the number is greater than some predefined threshold, the cell is split into eight new cells. The resulting tree has large cells in low density regions and small ones in high density regions, like the kd-tree. An example quadtree (2D octree) is shown in Figure 4.

5.2 Splitting

If splitting a cell, we need to initialize the eight new cells. Unfortunately, the distribution of points which filled the cell is not known anymore and we need to make a guess. Without further knowledge we can only distribute the counter uniformly to all child cells.

However, dividing the counter by eight, the number of children, systematically produces an underestimated initialization. The reason is that a 2D surface only intersects expectedly four of the eight children. Therefore, using the parent counter divided by four as initialization value turned out to be less biased in practice. The too large values in cells without an surface intersection do not matter, since they are never queried.

5.3 Queries

When a query is made, the count c in the respective cell must be converted to a density. Assuming locally flat surfaces and that the current surface is the only one inside the cell, the intersection area between the cell's bounding box $\mathcal{B}$ and the plane $\mathcal{P}$ can be calculated. Thereby, the plane is defined by the surface position $\mathbf{p}$ and normal $\mathbf{n}$. Thus, our density estimate is

$$\rho = \frac{c}{|\mathcal{B} \cap \mathcal{P}|} = \frac{c}{A}, \tag{11}$$

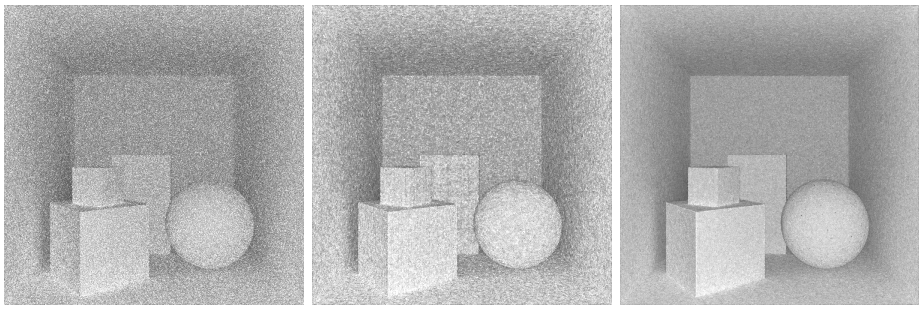


Figure 5: Queried density from a 4-nearest neighbor search (left) compared to the octree results after the first and the fourth iteration.

where the intersection area is computed with

$$A = \begin{cases} \mathbf{s}_x \cdot \mathbf{s}_y, & \mathbf{n}_x = \mathbf{n}_y = 0 \\ \mathbf{s}_x \left| \frac{1}{\mathbf{n}_y \mathbf{n}_z} \sum_{i=0}^{3} \epsilon_i \max\left(0, \langle \mathbf{n}, \mathbf{p} \rangle - \langle \mathbf{n}, \mathbf{b}_i \rangle\right) \right|, & \mathbf{n}_x = 0 \\ \left| \frac{1}{2\mathbf{n}_x \mathbf{n}_y \mathbf{n}_z} \sum_{i=0}^{7} \epsilon_i \max\left(0, \langle \mathbf{n}, \mathbf{p} \rangle - \langle \mathbf{n}, \mathbf{b}_i \rangle\right)^2 \right| \end{cases} \tag{12}$$

$$\mathbf{s} = \mathbf{b}_{111} - \mathbf{b}_{000}$$

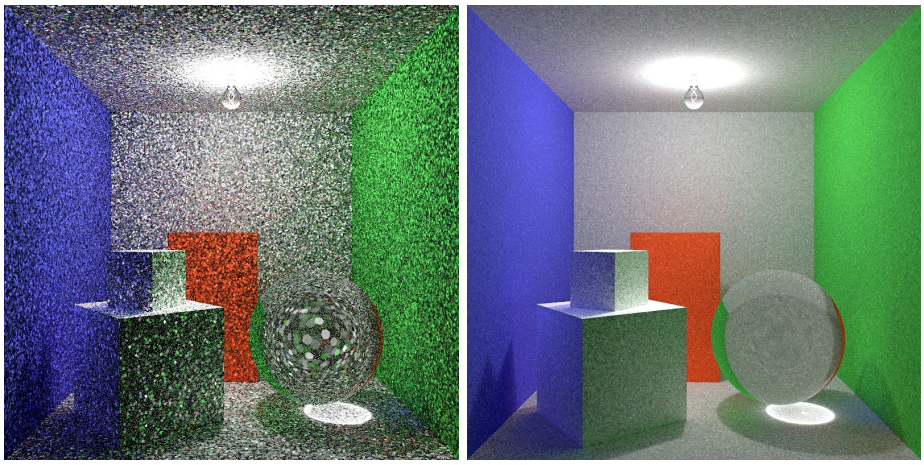
$$\epsilon_i = \begin{cases} 1, & i \in \{000, 011, 101, 110\} \\ -1, & i \in \{100, 010, 001, 111\} \end{cases}.$$

Here, x, y and z are the indices of the dimensions, $\mathbf{s}$ is the size of the box and ϵ_i is the parity of the eight vertices. In the first case, two of the dimensions are zero. W.l.o.g. only $\mathbf{n}_z \neq 0$ is shown. The terms for the other two dimensions are defined analogously. Similarly, case two shows the situation where one dimension of $\mathbf{n}$ is zero. Again, there are three analogous terms for all dimensions. The third case applies if no dimension is zero. A derivation can be found in Appendix A.

An advantage of the dedicated structure is that the density can be integrated over time. Therefore, the split threshold must be increased proportionally to the iteration count and Equation (11) must be divided by the number of iterations. This reduces noise and increases the independence between the current sample set and the density estimate. Figure 5 visualizes the query results and shows a fast convergence of the estimate. After four iterations the octree already has significantly less noise than the kd-tree-based search. Unfortunately, there are small dark points (see image on the right) which are caused by floating-point precision issues. These can cause too bright photons in the image which are often hidden by the MIS.

5.4 Memory Layout Details

We store the eight counters of the sibling cells in a consecutive sequence. This allows us to use a single pointer in a parent to address all its children. Moreover,



NEB, 68spp NEB + LP, 14spp

Figure 6: Equal time comparison (5 min) without and with additional photons (Option 2.c., Section 6.1) in a light bulb scenario.

it is possible to encode the counter and the child index in the same integer to save memory. If the stored value is negative, its absolute value is the index of the first child node. Otherwise it is a counter.

The aforementioned split threshold is set to four in our implementation. Using larger numbers introduces too much bias, because the blurring area increases and the planar surface assumption is less valid. Using smaller numbers increases the memory consumption and leads to more noise.

The necessary memory depends on two factors only: the expected number of NEE vertices divided by the split threshold. Particularly, it is independent of the scene. For typical setups the density octree requires less than 50 MB (often less than 10 MB suffice).

For more details on the lock-free implementation, we provide the code in Appendix B.

6 Modifications

Having a robust density estimate, all tools for NEB are given. In this section, we evaluate simple modifications to improve the performance or robustness of the basic algorithm

6.1 Conventional Light Photons

While the basic algorithm is very strong in scenarios like the teapot in a stadium (Figure 1), it fails when the caustic throwing object is much closer to the light source than to the receiver. Consider the example of a light bulb – an emitter

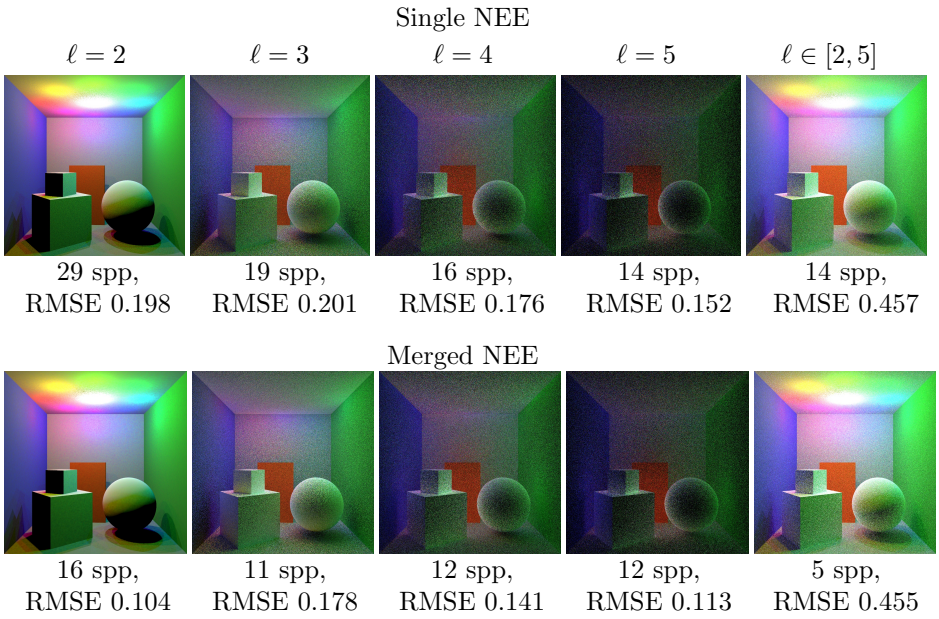


Figure 7: Equal time comparison (1 min) of Path Tracing with and without NEE recycling.

inside a glass ball. Only NEEs on the surface of the bulb produce contributing photons, but only very few paths randomly hit the comparable small bulb. All other vertices in the scenes have no NEE contribution, because the glass ball is blocking the connection to the light source. Therefore, the number of useful NEEs and photons are both very small, leading to high variance results.

We observed that the failure cases of NEB occur in situations where the conventional photon tracing, which starts at light sources, is strong. Hence, combining NEE photons and light photons by the means of MIS promises a more robust algorithm. In Figure 6 we demonstrate the effectiveness of this combination. The renderer becomes much more effective with respect to time although its iteration count decreases. The reason for both is that many more photons are found and merged at each position. Besides the additional tracing of photons, this requires more evaluations of the BSDF and the MIS weights.

6.2 NEE Recycling

Since we already store the NEE vertices in a search data structure, it seems reasonable to share the results of NEE events. Therefore, it is necessary to store the NEE information along with the vertices and to query those with a neighborhood search. Then, all available NEEs at one vertex must be averaged and the effective count of NEEs n_E in Equation (10) increases to the number of found events.

In Figure 7 we show an experiment with and without NEE reuse to judge the effectiveness of the proposed modification. Enabling the merges is clearly slower due to the range query and the additional evaluations of BSDFs. For the range queries we used a hash grid with a fixed query radius. Despite the lower number of samples, the noise level (*Root Mean Squared Error*) is slightly better when reusing the NEEs. However, the effectiveness decreases with path length and gets worse than usual PT for practical path lengths. Repeated experiments with different merge radii had the same outcome. The reason for the low effectiveness is that the noise in indirect lighting is dominated by the random walk and not the NEE.

Concluding, the idea of reusing the NEE events sounds promising, but does not pay off in this form. Therefore, we used only the one primary NEE without this modification all other experiments.

7 Comparison to Other Methods

The next event backtracking operator has clear strengths and weaknesses. It is strong whenever NEE is more likely than other events on the path. This is the case for small or distant light sources like in Figure 1. Additionally, it scales well with many lights, if contribution-sensitive NEEs are used. It is weak, whenever the vertex density is much smaller than the light contribution. In this situation, combining NEB with light photons (+LP) can alleviate this problem.

In Figure 8 we show an equal time comparison of the NEB method to other rendering algorithms in selected light situations. For the first two scenarios our method is superior due to the uniform sampling density of NEE vertices on the glass surfaces. The Mirrors scenario shows a small degeneration in quality where the distance to the caustic receiving surface increases (mirrors at the top). The Reflector scenario represents the worst case. Here, the tiny, bright surfaces close to the point light source are seldom found by a Path Tracer.

In Figure 9 more realistic scenes are compared. For situations like in the Watch scene, NEB is superior to the state of the art VCM. In other situations (Bathroom or Christmas scene) it shows more noise than VCM without modifications. Enabling the additional tracing of light photons makes NEB equally effective as VCM. In all our experiments we found that NEB+LP is very strong for caustics and SDS paths regardless of the scene scale. For diffuse indirect lighting it performs on a comparable level to most other methods.

Comparing NEB as a method to produce high quality photons maps against the older methods [PP98, SW00, KW00], it produces a medium quality map. In most scenes, it performs similarly to the other methods without wasting samples for the estimation of importance distributions. However, if there are small reflectors close to the light source (light bulb or Figure 8) it will be less effective, because it does not respect the radiance distribution. Adding standard photons compensates this weakness, but their distribution does not follow importance again. However, since none of the photons is ever stored, the memory consumption of additional photons is of no interest.

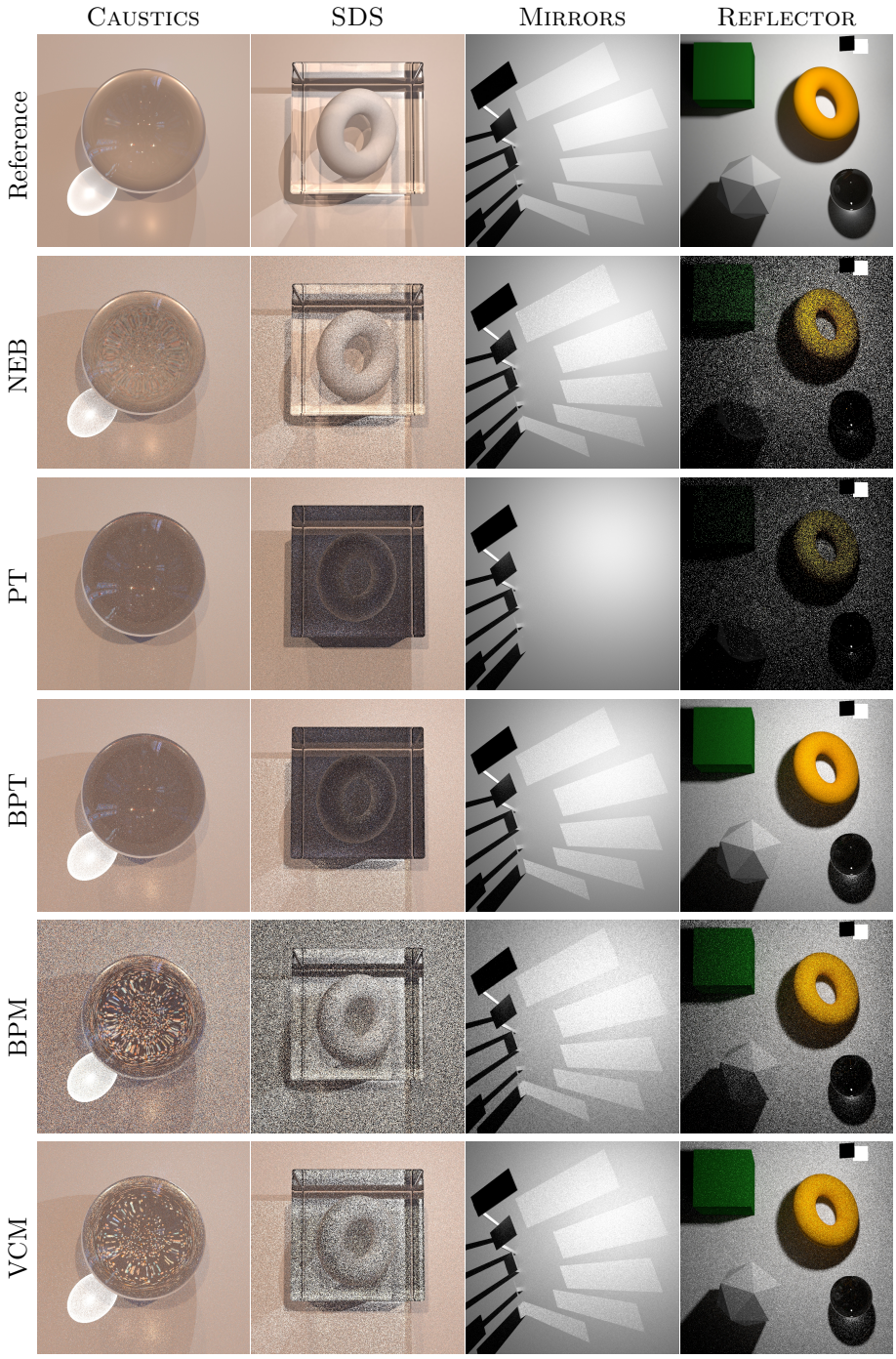


Figure 8: Equal time comparison (1 min) of different rendering algorithms in difficult light scenarios.

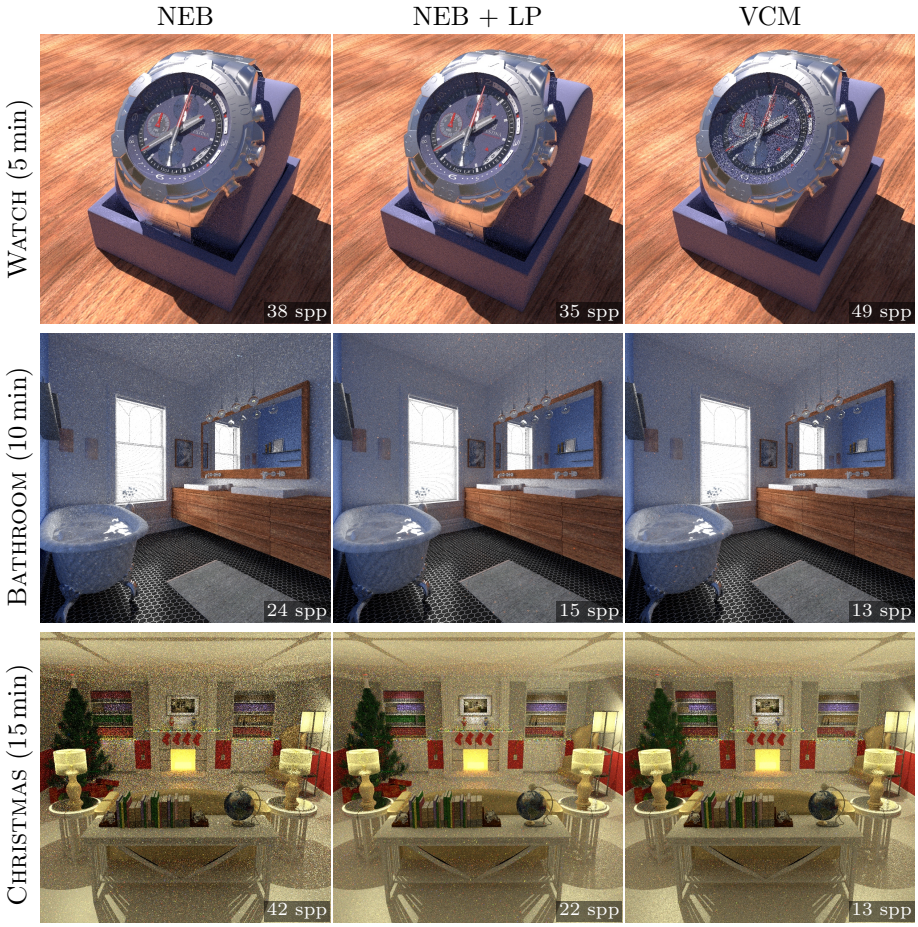


Figure 9: Comparison of NEB to VCM in more realistic scenes.

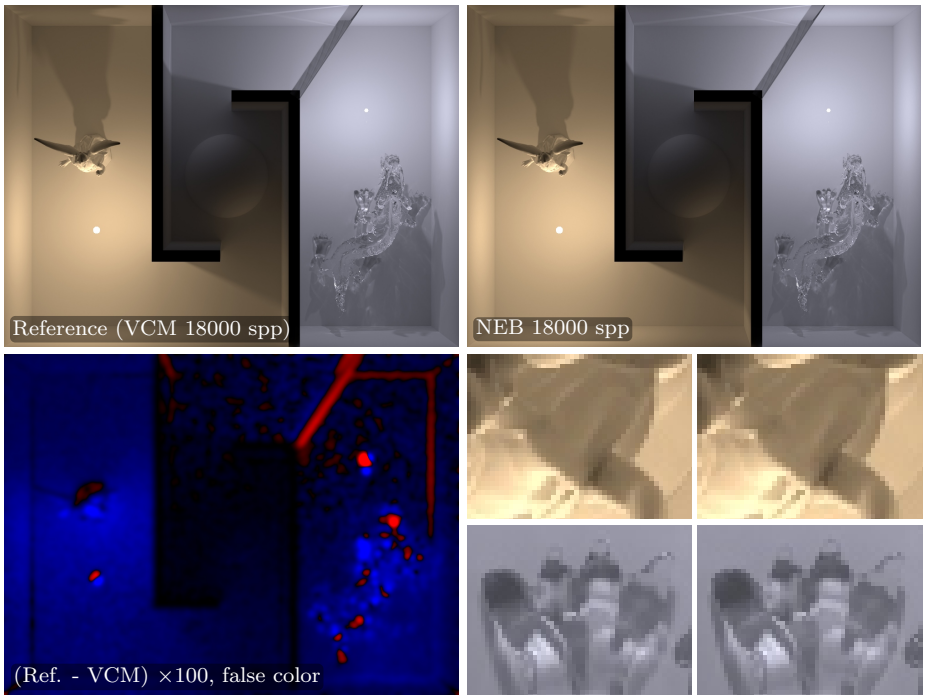


Figure 10: Bias visualization. The scene shows multiple indirect lighting, caustics and SDS paths. The bottom left image is the blurred difference image: reference minus NEB times 100 in false colors. Regions with a wobbly pattern are unbiased, whereas most parts are systematically larger in the NEB rendering (blue regions). Only in a few regions the bias is visible directly (closeups).

7.1 Bias

In Figure 10 we show a scene with many separated lighting situation to demonstrate the relatively small bias which is introduced by the density estimates. While there is a measurable bias, it is barely visible. Even in the most sever cases, a good display device is necessary to see that NEB is slightly brighter in some regions (closeups). The worst error which we observed happens if the planarity assumption does not hold. In Figure 8, MIRRORS one such artifact can be found on the edge of the top most caustic. A possible solution would be to additionally split the octree dependent on a local curvature criterion, to ensure that the planarity assumption does not fail catastrophically.

8 Conclusions and Future Work

We proposed a new light transport operator – the next event backtracking – which is strong in situations where the usual photon transport fails. It can

successfully render caustics of small objects in large scenes due to an implicit guidance of light paths toward the important regions and is often very strong for SDS paths.

The most difficult problem is a robust and fast density estimate to convert irradiance into flux at the selected positions. We introduced an octree-based data structure which is fast, but may cause small artifacts from its grid structure leading to overly bright photons.

Our NEB algorithm used the operator in the setup of a conventional Path Tracer. In this configuration, certain light situations can still cause high variance. The modification to add conventional photon tracing, beginning at the light source, solves this problem. Effectively, NEB is weak if standard photon tracing is strong and vice versa. Therefore, both together result in a very robust algorithm. Only scenes with bad visibility between observer and light source are still a problem.

One of the practical weaknesses is the memory consumption, which is slightly higher than in other photon mapping-based approaches. The view path vertex storage exchanges the one for photons and has a comparable size. Additionally, there is the octree to store the density values with up to 50 MB and the storage for emissive events with another 50 MB per one million paths.

Also, it would be interesting to implement the algorithm on a GPU. In contrast to BPT or VCM, the number of connections is only linear in path lengths. This reduces divergence and per-path-time such that our method should scale well on parallel hardware.

Another future avenue would be to use MCMC samplers for the view paths or to use the NEB paths as seed paths for MCMC samplers. This would increase the method's robustness with respect to bad visibility of the light source due to high occlusion.

Finally, our approach allows the use of arbitrarily sampled positions as photon emitters. It would be possible (but likely ineffective) to distribute emitters equally on the surfaces – independent of camera and light sources. More interesting would be to use Markov chains to sample the emitters on the surfaces themselves. Similar to Lightweight Photonmapping [GPGSK18], it would then be possible to remove emitters on surfaces where other samplers are more successful and to increase the density otherwise.

Acknowledgments

I want to thank those people who make 3D scenes publicly available. The villa and the bathroom scene are taken form the PBRT repository. The stadium was modeled by Ericchip1983 and is provided on blendswap.com/blends/74526. The wrist watch was created by HeraSK (blendswap.com/blends/70232).

References

[Ben75] Bentley J. L.: Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM 18*, 9 (Sept. 1975), 509–517. doi:10.1145/361002.361007.

[CKL*10] Choi B., Komuravelli R., Lu V., Sung H., Bocchino R. L., Adve S. V., Hart J. C.: Parallel SAH k-D Tree Construction. In *Proc. of High Performance Graphics (HPG)* (2010), Eurographics Association, pp. 77–86. https://github.com/bchoi/ParKD. URL: http://dl.acm.org/citation.cfm?id=1921479.1921492.

[FHF*17] Fascione L., Hanika J., Fajardo M., Christensen P., Burley B., Green B.: Path Tracing in Production - Part 1: Production Renderers. In *ACM SIGGRAPH Courses* (2017), pp. 13:1–13:39. doi:10.1145/3084873.3084904.

[GKDS12] Georgiev I., Křivánek J., Davidovič T., Slusallek P.: Light Transport Simulation with Vertex Connection and Merging. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia) 31*, 6 (2012), 192:1–192:10. URL: https://cgg.mff.cuni.cz/~jaroslav/papers/2012-vcm/, doi:10.1145/2366145.2366211.

[GPGSK18] Grittmann P., Pérard-Gayot A., Slusallek P., Křivánek J.: Efficient Caustic Rendering with Lightweight Photon Mapping. *Computer Graphics Forum 37*, 4 (2018), 133–142. doi:10.1111/cgf.13481.

[HDF15] Hanika J., Droske M., Fascione L.: Manifold Next Event Estimation. *Computer Graphics Forum (Proc. EGSR) 34*, 4 (July 2015), 87–97. doi:10.1111/cgf.12681.

[HJ11] Hachisuka T., Jensen H. W.: Robust Adaptive Photon Tracing using Photon Path Visibility. *ACM Transactions on Graphics (TOG) 30*, 5 (2011), 114.

[HPJ12] Hachisuka T., Pantaleoni J., Jensen H. W.: A Path Space Extension for Robust Light Transport Simulation. *ACM Transactions on Graphics (Proc. of SIGGRAPH Asia) 31*, 6 (Nov. 2012), 191:1–191:10. doi:10.1145/2366145.2366210.

[Jen96] Jensen H. W.: Global Illumination using Photon Maps. In *Proc. of Eurographics Workshop on Rendering (EGWR)* (1996), Springer, pp. 21–30. URL: http://dl.acm.org/citation.cfm?id=275458.275461.

[Jen19] Jendersie J.: Variance Reduction via Footprint Estimation in the Presence of Path Reuse. In *Ray Tracing Gems*, Haines E., Akenine-Möller T., (Eds.), 1 ed. Apress, Feb. 2019, pp. 557–569. URL: http://raytracinggems.com.

[JG18] Jendersie J., Grosch T.: An Improved Multiple Importance Sampling Heuristic for Density Estimates in Light Transport Simulations. In *Proc. of Eurographics Symposium on Rendering EI&I Track (EGSR)* (July 2018), Eurographics Association, pp. 65–72. doi:10.2312/sre.20181173.

[Kaj86] Kajiya J. T.: The Rendering Equation. *Computer Graphics (Proc. ACM SIGGRAPH) 20*, 4 (Aug. 1986), 143–150. doi:10.1145/15886.15902.

[KW00] Keller A., Wald I.: Efficient Importance Sampling Techniques for the Photon Map. In *Proc. of Vision, Modeling, and Visualization (VMV)* (2000), pp. 271–278.

[LW93] Lafortune E. P., Willems Y. D.: Bi-Directional Path Tracing. In *Proc. of Conference on Computational Graphics and Visualization Techniques* (1993), pp. 145–153. URL: https://lirias.kuleuven.be/handle/123456789/132773.

[MGN17] Müller T., Gross M., Novák J.: Practical Path Guiding for Efficient Light-Transport Simulation. In *Proc. of Eurographics Symposium on Rendering (EGSR)* (June 2017). doi:10.1111/cgf.13227.

[PP98] Peter I., Pietrek G.: Importance Driven Construction of Photon Maps. In *Proc. of Eurographics Workshop on Rendering (EGWR)* (1998), Springer-Verlag, pp. 269–280. doi:10.1007/978-3-7091-6453-2_25.

[She04] Sheather S. J.: Density Estimation. *Statistical Science 19*, 4 (2004), 588–597. URL: http://www.jstor.org/stable/4144429.

[Sil86] Silverman B. W.: *Density Estimation for Statistics and Data Analysis*. Chapman and Hall/CRC, Apr. 1986.

[ŠK18] Šik M., Křivánek J.: Survey of Markov Chain Monte Carlo Methods in Light Transport Simulation. *IEEE Transactions on Visualization and Computer Graphics (TVCG)* (Nov. 2018). doi:10.1109/TVCG.2018.2880455.

[SW00] Suykens F., Willems Y. D.: Density Control for Photon Maps. In *Proc. of Eurographics Workshop on Rendering (EGWR)* (2000), Springer, pp. 23–34. URL: http://graphics.cs.kuleuven.be/publications/PHOTONDC/index.html, doi:10.1007/978-3-7091-6303-0_3.

[VG95a] Veach E., Guibas L. J.: Bidirectional Estimators for Light Transport. In *Photorealistic Rendering Techniques*. Springer Berlin Heidelberg, 1995, pp. 145–167. URL: http://dx.doi.org/10.1007/978-3-642-87825-1_11.

[VG95b] Veach E., Guibas L. J.: Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proceedings of SIGGRAPH* (1995), ACM, pp. 419–428. doi:10.1145/218380.218498.

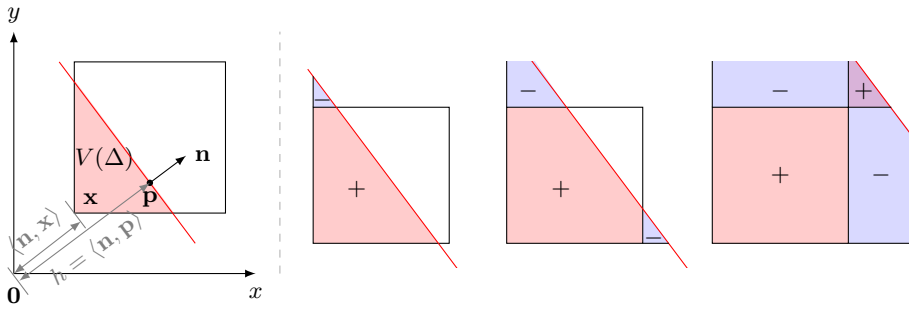


Figure 11: Intersection area from simplex volume. In 2D the simplex-volume is the area of a triangle. Left: a single vertex of the box lies below the plane $(\mathbf{p}, \mathbf{n})$. Right: sequence of events when moving the plane along $\mathbf{n}$.

A Intersection Area between Plane and Box

We are sure that this formula is not new, but we were not able to find a cite-able reference for Equation (12). We found the derivation on Mathoverflow (https://math.stackexchange.com/a/885662/661978) and repeat it here for completeness.

The basic idea is to compute the volume V inside the box and below the plane with respect to the plane offset $h = \langle \mathbf{n}, \mathbf{p} \rangle$. Then the derivation of V yields the searched area. As primary conditions we have $\|\mathbf{n}\| = 1$ and that the box is axis aligned. Let

$$\Delta(h, \mathbf{x}) = \{\mathbf{y} \in \mathbb{R}^d : \forall \mathbf{y}_i \geq \mathbf{x}_i \wedge \langle \mathbf{n}, \mathbf{y} \rangle \leq h\} \tag{13}$$

be the d-dimensional simplex, which starts at $\mathbf{x}$ as depicted in Figure 11. The volume of this simplex can be computed from a product of its (axis aligned) edge lengths with

$$V(\Delta(h, \mathbf{x})) = V(\Delta(h - \langle \mathbf{n}, \mathbf{x} \rangle, \mathbf{0})) = \frac{\max(0, h - \langle \mathbf{n}, \mathbf{x} \rangle)^d}{d! \prod_i \mathbf{n}_i}. \tag{14}$$

The clamping to zero is necessary if the entire box is on the upper side of the plane.

Now, moving the plane along $\mathbf{n}$, it will pass all vertices of the box in a sorted order. This sequence is shown in Figure 11, too. After passing the second vertex, the simplex from the first one will overestimate the volume. Here, the second simplex starting at the second vertex must be subtracted. The same applies to the third vertex. After the forth vertex, the subtracted areas from the second and third vertex will overlap and the volume is underestimated. Thus, the volume of the forth simplex must be added again. This alternating sign is described by the parity ϵ as given in Equation (12).

Hence, the volume of the box $\mathcal{B}$ with respect to the plane is

$$V(\mathcal{B}, h) = \frac{1}{d! \prod_i \mathbf{n}_i} \sum_{i=0}^{2^d - 1} \epsilon_i \max(0, h - \langle \mathbf{n}, \mathbf{b}_i \rangle)^d. \tag{15}$$

Finally, the change of volume over h is dominated by the area of the infinitesimal slab which gives the area

$$\begin{aligned} A(\mathcal{B}, h) &= \frac{\partial V(\mathcal{B}, h)}{\partial h} \\ &= \frac{1}{(d-1)! \prod_i \mathbf{n}_i} \sum_{i=0}^{2^d-1} \epsilon_i \max(0, h - \langle \mathbf{n}, \mathbf{b}_i \rangle)^{(d-1)}. \end{aligned} \tag{16}$$

Equation (12) is then obtained by applying the above equation for one, two and three dimensions. Each component of $\mathbf{n}$ which is zero means that the normal is perpendicular to the respective edge of the box. Then, the size $\mathbf{s}$ of the box must be multiplied with the area of the simplex from the reduced dimension.

B Octree Implementation

To use the following C++ source codes you must provide some implementation of a 3D vector with following operations:

- arithmetic (+, -, *, /)
- array access [0] to [2] for the three dimensions
- functions like dot(), abs(), len()

Helper function which implements Equation (12):

```
inline float sq(float x) { return x * x; }

// Compute the area of the plane-box intersection
// https://math.stackexchange.com/questions/885546
// https://math.stackexchange.com/a/885662
// or appendix A
inline float intersection_area(const vec3& bmin, const vec3& bmax,
                               const vec3& pos, const vec3& normal) {
  vec3 cellSize = bmax - bmin;
  vec3 absN = abs(normal);
  // 1D cases
  if(abs(absN[0] - 1.0f) < 1e-3f) return cellSize[1] * cellSize[2];
  if(abs(absN[1] - 1.0f) < 1e-3f) return cellSize[0] * cellSize[2];
  if(abs(absN[2] - 1.0f) < 1e-3f) return cellSize[0] * cellSize[1];
  // 2D cases
  for(int d = 0; d < 3; ++d) if(absN[d] < 1e-4f) {
    int x = (d + 1) % 3;
    int y = (d + 2) % 3;
    // Use the formula from stackexchange: phi(t) = max(0,t)^2 / 2 m_1 m_2
    // -> l(t) = sum^4 s max(0,t-dot(m,v)) / m_1 m_2
    // -> A(t) = l(t) * h_3
    float t = normal[x] * pos[x] + normal[y] * pos[y];
    float sum = 0.0f;
    sum += max(0.0f, t - (normal[x] * bmin[x] + normal[y] * bmin[y]));
    sum -= max(0.0f, t - (normal[x] * bmin[x] + normal[y] * bmax[y]));
    sum -= max(0.0f, t - (normal[x] * bmax[x] + normal[y] * bmin[y]));
    sum += max(0.0f, t - (normal[x] * bmax[x] + normal[y] * bmax[y]));
    return cellSize[d] * abs(sum / (normal[x] * normal[y]));
  }
  // 3D cases
  float t = dot(normal, pos);
  float sum = 0.0f;
  sum += sq(max(0.0f, t - dot(normal, bmin)));
  sum -= sq(max(0.0f, t - dot(normal, vec3{bmin[0], bmin[1], bmax[2]})));
  sum += sq(max(0.0f, t - dot(normal, vec3{bmin[0], bmax[1], bmax[2]})));
  sum -= sq(max(0.0f, t - dot(normal, vec3{bmin[0], bmax[1], bmin[2]})));
  sum += sq(max(0.0f, t - dot(normal, vec3{bmax[0], bmax[1], bmin[2]})));
  sum -= sq(max(0.0f, t - dot(normal, vec3{bmax[0], bmin[1], bmin[2]})));
  sum += sq(max(0.0f, t - dot(normal, vec3{bmax[0], bmin[1], bmax[2]})));
  sum -= sq(max(0.0f, t - dot(normal, bmax)));
  return abs(sum / (2.0f * normal[0] * normal[1] * normal[2]));
}
```

Helper to track the depth of the tree:

```
template<typename T>
inline void atomic_max(std::atomic<T>& a, T b) {
  T oldV = a.load();
  while(oldV < b && !a.compare_exchange_weak(oldV, b)) ;
}
```

```
// A sparse octree with atomic insertion to measure the density of elements
// in 3D space.
class DensityOctree {
  static constexpr int SPLIT_FACTOR = 4;
  // At some time the counting should stop -- otherwise the counter will
  // overflow inevitable.
  static constexpr int FILL_ITERATIONS = 1000;
public:
  void set_iteration(int iter) {
    int iterClamp = min(FILL_ITERATIONS, iter);
    m_stopFilling = iter > FILL_ITERATIONS;
    m_densityScale = 1.0f / iterClamp;
    m_splitCountDensity = SPLIT_FACTOR * iterClamp;
    // Set the counter of all unused cells to the number of expected samples
    // divided by 4. A planar surface will never extend to all eight cells.
    // It might intersect 7 of them, but still the distribution is on a
    // surface. Therefore, the SPLIT_FACTOR many particles are distribute
    // among 4 cells. This gives a much better value than dividing the
    // factor by 8.
    if(!m_stopFilling)
      for(int i = m_allocationCounter.load(); i < m_capacity; ++i)
        m_nodes[i].store(ceil(SPLIT_FACTOR / 4.0f * iter));
  }

  void initialize(const vec3& sceneMin, const vec3& sceneMax, int capacity) {
    // Slightly enlarge the volume to avoid numerical issues on the boundary
    vec3 sceneSize = (sceneMax - sceneMin) * 1.002f;
    m_sceneSizeInv = 1.0f / sceneSize;
    m_sceneScale = len(sceneSize);
    m_minBound = sceneMin - sceneSize * (0.001f / 1.002f);
    // Round up to 8n+1 - otherwise we cannot use the last [1,7] entries.
    m_capacity = 1 + ((capacity + 7) & (~7));
    m_nodes = std::make_unique<std::atomic_int32_t[]>(m_capacity);;
    // Allocate the root node with a count of 0
    m_allocationCounter.store(1);
    m_nodes[0].store(0);
    m_depth.store(0);
  }

  // Overwrite all counters with 0, but keep allocation and child pointers.
  void clear_counters() {
    int n = m_allocationCounter.load();
    for(int i = 0; i < n; ++i)
      if(m_nodes[i].load() > 0)
        m_nodes[i].store(0);
  }

  void increment(const vec3& pos) {
    if(m_stopFilling) return;
    vec3 normPos = (pos - m_minBound) * m_sceneSizeInv;
    int countOrChild = increment_if_positive(0);
    countOrChild = split_node_if_necessary(0, countOrChild, 0);
    int edgeL = 1;
    int currentDepth = 0;
    while(countOrChild < 0) {
      edgeL *= 2;
      ++currentDepth;
      // Get the relative index of the child [0,7]
      ivec3 intPos = (ivec3{ normPos * edgeL }) & 1;
      int idx = intPos[0] + 2 * (intPos[1] + 2 * intPos[2]);
      idx -= countOrChild;  // 'Add' global offset (which is stored negative)
      countOrChild = increment_if_positive(idx);
      countOrChild = split_node_if_necessary(idx, countOrChild, currentDepth);
    }
  }
```

```
float get_density(const vec3& pos, const vec3& normal, float* size = 0) {
  vec3 offPos = pos - m_minBound;
  vec3 normPos = offPos * m_sceneSizeInv;
  // Get the integer position on the finest level.
  int gridRes = 1 << m_depth.load();
  ivec3 iPos { normPos * gridRes };
  // Get root value. This will most certainly be a child pointer...
  int countOrChild = m_nodes[0].load();
  // The most significant bit in iPos distinguishes the children of the
  // root node. For each level, the next bit will be the relevant one.
  int currentLvlMask = gridRes;
  while(countOrChild < 0) {
    currentLvlMask >>= 1;
    // Get the relative index of the child [0,7]
    int idx = ((iPos[0] & currentLvlMask) ? 1 : 0)
        + ((iPos[1] & currentLvlMask) ? 2 : 0)
        + ((iPos[2] & currentLvlMask) ? 4 : 0);
    // 'Add' global offset (which is stored negative)
    idx -= countOrChild;
    countOrChild = m_nodes[idx].load();
  }
  if(countOrChild > 0) {
    // Get the world space cell boundaries
    int currentGridRes = gridRes / currentLvlMask;
    ivec3 cellPos = iPos / currentLvlMask;
    vec3 cellSize = 1.0f / (currentGridRes * m_sceneSizeInv);
    vec3 cellMin = cellPos * cellSize;
    vec3 cellMax = cellMin + cellSize;
    float area = intersection_area(cellMin, cellMax, offPos, normal);
    // Sometimes the above method returns zero. Therefore, we restrict the
    // area to something larger than 1/100 of an approximate cell area.
    float cellDiag = m_sceneScale / currentGridRes;
    float minArea = cellDiag * cellDiag;
    if(size) { *size = cellDiag; minArea *= 0.1f; }
    else minArea *= 0.01f;
    return m_densityScale * countOrChild / max(minArea, area);
  }
  return 0.0f;
}

// The robust version additional samples four neighbors in the tangent
// plane and returns the median of those results. This removes noise and
// therefore outliers in the rendering.
float get_density_robust(const vec3& pos, const scene::TangentSpace& ts) {
  float d[5];
  float cellDiag = 1e-3f;
  int count = 0;
  d[0] = get_density(pos, ts.geoN, &cellDiag);
  cellDiag *= 1.1f;
  if(d[0] > 0.0f) ++count;
  d[count] = get_density(pos + ts.shadingTX * cellDiag, ts.geoN);
  if(d[count] > 0.0f) ++count;
  d[count] = get_density(pos - ts.shadingTX * cellDiag, ts.geoN);
  if(d[count] > 0.0f) ++count;
  d[count] = get_density(pos + ts.shadingTY * cellDiag, ts.geoN);
  if(d[count] > 0.0f) ++count;
  d[count] = get_density(pos - ts.shadingTY * cellDiag, ts.geoN);
  if(d[count] > 0.0f) ++count;
  // Find the median via selection sort up to the element m.
  // Prefer the greater element, because overestimations do not
  // cause such visible artifacts.
  int m = count / 2;
  for(int i = 0; i <= m; ++i) for(int j = i+1; j < count; ++j)
    if(d[j] < d[i])
      std::swap(d[i], d[j]);
  return d[m];
}

int capacity() const { return m_capacity; }
int size() const { return min(m_capacity, m_allocationCounter.load()); }
```

```
private:
  // Nodes consist of 8 atomic counters OR child indices. Each number is
  // either a counter (positive) or a negated child index.
  std::unique_ptr<std::atomic_int32_t[]> m_nodes;
  std::atomic_int32_t m_allocationCounter;
  std::atomic_int32_t m_depth;
  vec3 m_minBound;
  vec3 m_sceneSizeInv;
  float m_sceneScale;
  float m_densityScale;     // 1/#iterations to normalize the counters
  int m_capacity;
  int m_splitCountDensity;  // The number when a node is split must be a
                            // multiple of 8 and must grow proportional
                            // to #iterations
  bool m_stopFilling;

  // Returns the new value
  int increment_if_positive(int idx) {
    int oldV = m_nodes[idx].load();
    int newV;
    do {
      if(oldV < 0) return oldV; // Do nothing, the value is a child pointer
      newV = oldV + 1;          // Increment
      // Write if nobody changed the value in between
    } while(!m_nodes[idx].compare_exchange_weak(oldV, newV));
    return newV;
  }

  // Returns the next child pointer or 0
  int split_node_if_necessary(int idx, int count, int currentDepth) {
    // The node must be split if its density gets too high
    if(count >= m_splitCountDensity) {
      // Only one thread is responsible to do the allocation
      if(count == m_splitCountDensity) {
        int child = m_allocationCounter.fetch_add(8);
        if(child >= m_capacity) { // Allocation overflow
          // Avoid overflow of the counter (but keep a large number)
          m_allocationCounter.store(int(m_capacity + 1));
          return 0;
        }
        // We do not know anything about the distribution of photons
        // -> equally distribute. Therefore, all eight children are
        // initilized with SPLIT_FACTOR on set_iteration().
        m_nodes[idx].store(-child);
        // Update depth
        atomic_max(m_depth, currentDepth+1);
        // The current photon is already counted before the split
        // -> return stop
        return 0;
      } else {
        // Spin-lock until the responsible thread has set the child pointer
        int child = m_nodes[idx].load();
        while(child > 0) {
          // Check for allocation overflow
          if(m_allocationCounter.load() > m_capacity)
            return 0;
          child = m_nodes[idx].load();
        }
        return child;
      }
    }
    return count; // count is already a child
  }
};
```