

Real-Time Stochastic Lightcuts

DAQI LIN, University of Utah
CEM YUKSEL, University of Utah



Fig. 1. Our real-time stochastic lightcuts method renders a scene with unbiased sampling of direct lighting from 24,782 emissive triangles with a sampling time of 11.5 ms (including tracing shadow ray and lighting computation) and a total frame time of 23 ms on a NVIDIA RTX 2080 card, using 4 light samples per pixel. The screen resolution is 1920 × 1080. SVGF [Schied et al. 2017] and TAA are applied to filter the sampling result.

We present real-time stochastic lightcuts, a real-time rendering method for scenes with many dynamic lights. Our method is the GPU extension of stochastic lightcuts [Yuksel 2019], a state-of-art hierarchical light sampling algorithm for offline rendering. To support arbitrary dynamic scenes, we introduce an extremely fast light tree builder. To maximize the performance of light sampling on the GPU, we introduce cut sharing, a way to reuse adaptive sampling information in light trees in neighboring pixels.

CCS Concepts: • Computing methodologies → Rendering; Ray tracing.

Additional Key Words and Phrases: Many lights, ray tracing, importance sampling.

ACM Reference Format:

Daqi Lin and Cem Yuksel. 2020. Real-Time Stochastic Lightcuts. *Proc. ACM Comput. Graph. Interact. Tech.* 3, 1 (May 2020), 18 pages. <https://doi.org/10.1145/3384543>

Authors' addresses: Daqi Lin, University of Utah, daqi@cs.utah.edu; Cem Yuksel, University of Utah, cem@cemyuksel.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

2577-6193/2020/5-ART \$15.00

<https://doi.org/10.1145/3384543>

1 INTRODUCTION

Scenes with many lights are commonplace in real-time rendering applications like video games. Yet, handling many lights has been a challenge in real-time rendering, due to the complexity of accumulating illumination from all lights for all pixels. Most game engines handle scenes with many lights using a mixture of baking and tile-based deferred rendering [Olsson and Assarsson 2011; Olsson et al. 2012] to determine which pixels should be illuminated by which lights. However, baked lighting is difficult to use with animations and tile-based deferred rendering only works if the lights have relatively small influence ranges (though stochastic ranges can be used for lights with unbounded influence ranges [Tokuyoshi and Harada 2016] to approximate the lighting). Recent Monte Carlo sampling methods for real-time rendering [Moreau and Clarberg 2019; Moreau et al. 2019] lift these limitations, but their computation cost limits the number of light samples that can be used at real-time frame rates, resulting in noisy lighting estimations.

We present an extension of stochastic lightcuts [Yuksel 2019] and describe how it can be used to achieve high-performance rendering with many lights on the GPU (Figure 1). Our goal is to minimize the overhead of sampling from a *light tree*, so that we can afford more light samples within the same render time, achieving higher quality (i.e. lower noise). Thus, we accept sacrificing the quality of the light tree to save computation time, which can be used towards more light samples, resulting in a net gain in quality. To achieve this, we use a perfect (i.e. balanced, full, and complete) binary light tree. A perfect binary tree allows extremely fast construction and also boosts the light sampling performance. Moreover, we introduce a novel weight computation scheme for hierarchical importance sampling that provides improvements in sampling quality in our test scenes. Furthermore, we introduce *cut sharing*, which allows a block of pixels to share the same *cut* through the light tree for minimizing the overhead of cut selection. We show how cut sharing enables the use of *interleaved sampling* [Segovia et al. 2006] to further improve the performance or light sampling quality. We compare our real-time stochastic lightcuts method with prior work on sampling many lights and show that our method improves the speed of light sampling, thereby providing higher quality with more light samples within the same render time budget.

2 BACKGROUND

In this section we briefly overview the related work for handling many lights and summarize stochastic lightcuts [Yuksel 2019].

2.1 Related Work

Early work on rendering with many lights includes sorting lights by contributions [Ward 1994], importance sampling [Shirley et al. 1996], building octrees for light clustering [Paquette et al. 1998], and constructing a local illumination environment [Fernandez et al. 2002]. The introduction of virtual point lights (VPLs) for approximating global illumination [Keller 1997] has drawn more research interests into the many-lights problem. Walter et al. [2005] introduced the lightcuts method as an efficient solution to the many-lights problem. The method is based on a light tree which resembles the hierarchical approach introduced by Paquette et al. [1998]. For each point in the scene, where lighting is evaluated, lightcuts picks a *cut* through the light tree and only computes the representative lights of the internal light tree nodes above this cut. It is also possible to avoid computing a cut for each pixel using reconstruction cut [Walter et al. 2005] or lightcut interpolation [Rehfeld and Dachsbacher 2016].

Another efficient solution to the VPL-related many-lights problem is matrix row-column sampling (MRCS) [Hašan et al. 2007] which approximates the lighting matrix of the scene. Extensions of lightcuts [Davidović et al. 2012; Walter et al. 2006, 2012] and techniques inspired by MRCS

[Davidovic et al. 2010; Hašan et al. 2008; Huo et al. 2015; Ou and Pellacini 2011] improve the efficiency or handle more general scenarios Dachsbacher et al. [2014].

The Lighting Grid Hierarchy method [Yuksel and Yuksel 2017] introduces a temporally coherent approximation of a large number of VPLs for rendering self illuminating volumes by using multiple representations of the illumination at different resolutions. The method was recently extended for real-time approximation of global illumination [Lin and Yuksel 2019].

Vévoda and Křivánek [2016] used the clusters formed by lightcuts for adaptive importance sampling of direct illumination instead of using them directly as the illumination approximation. A following work [Vévoda et al. 2018] uses Bayesian online regression to learn the light selection probability distributions for the light clusters. Keller et al. [2017] added additional node information into the light tree, where each node stores directionally varying light intensities, instead of a single flux. Adaptive tree splitting [Estevez and Kulla 2018] introduced a light bounding volume hierarchy (BVH) with a technique that splits light tree traversal based on the cluster variance. Recently, Yuksel [2019] introduced stochastic lightcuts which provides low noise results using much fewer samples than other methods.

With the advancement of high-quality denoisers and ray tracing capable GPUs, it is now possible to implement Monte Carlo sampling algorithms on the GPU using limited samples. Recently, Moreau and Clarberg [2019] presented a version of adaptive tree splitting for real-time rendering, and Moreau et al. [2019] introduced a two-level light BVH builder for dynamic scenes.

2.2 Stochastic Lightcuts

Stochastic lightcuts [Yuksel 2019] extends the lightcuts method [Walter et al. 2005]. Lightcuts builds a binary light tree prior to rendering. During rendering the light tree is evaluated from top to bottom for selecting a *cut* through the light tree. The cut is initially placed at the root node. For each node of the light tree along the cut, if the maximum possible illumination from the subtree under the node is above a threshold, the cut is moved one level below.

Stochastic lightcuts [Yuksel 2019] uses a *hierarchical importance sampling* technique (similar to [Estevez and Kulla 2018]) for randomly selecting a light sample within each subtree under the chosen cut, converting lightcuts to an unbiased light sampling method. For each light tree node above the cut, hierarchical importance sampling traverses the subtree below the node from top to bottom, randomly picking one of the child nodes under each node, down to a leaf node that contains a single light source, which is chosen as the light sample.

3 REAL-TIME STOCHASTIC LIGHTCUTS

Our real-time stochastic lightcuts method contains two key components that make it GPU friendly. First, we use a perfect binary light tree, which is extremely fast to build (Section 3.1) and efficient to traverse (Section 3.2). Second, we share cuts within $k \times k$ pixel blocks (Section 3.3), instead of computing a cut for each pixel during light sampling. Our cut sharing technique naturally enables interleaved sampling (Section 3.4), which can be used for further accelerating light sampling or improving its quality.

Our goal is to minimize the light tree construction and light sample selection times. The choices we make for achieving this goal, however, may adversely affect the quality of the tree and the light sample distribution. Yet, this reduction in sampling quality can be offset by using more light samples. Thus, our ultimate goal is using the time we save during the tree construction and light sample selection towards more light samples for a better illumination estimation.

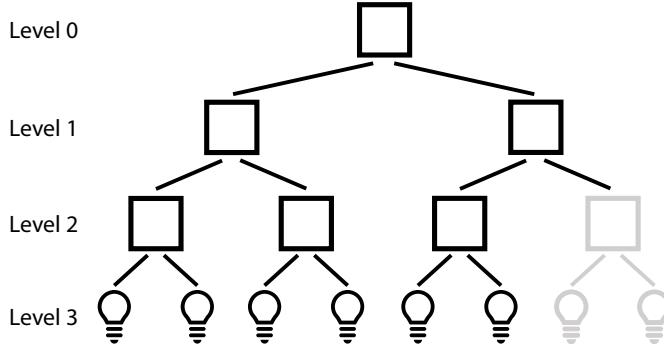


Fig. 2. An example of a perfectly balanced light tree with four levels. Bogus lights with zero intensities are appended to the end of the leaf level (level 3) to round up the number of lights to the nearest greater power of two. Bogus lights and bogus nodes are marked with gray color.

3.1 Perfectly Balanced Light Tree

As mentioned above, neither the existing agglomerative or divisive light clustering methods are fast enough to fully rebuild the light tree in real-time. To solve this problem, we use a perfect (i.e. balanced, complete, and full) binary light tree. In a perfect tree, all leaf nodes that contain the individual light sources appear at the bottom level of the tree. Since the number of leaf nodes in a perfect tree must be a power of two, we add *bogus lights* as needed (Figure 2).

Each node in the light tree stores the bounding box of the underlying lights and the total light intensity. Since we use a perfect tree, there is no need to store child node pointers, as the child node indices can be computed directly from the parent node index. Additionally, each leaf node stores the corresponding light ID. Note that light IDs cannot be directly computed from leaf node indices in dynamic scenes, since the leaf index of a light can vary each frame.

For minimizing the construction time, our builder sorts the light sources based on the Morton code of their positions (conceptually similar to a Morton code BVH builder [Lauterbach et al. 2009]). The leaf nodes are generated by directly copying the light information in the sorted order into the leaf nodes of the tree. Afterwards, computing the internal node data is a straightforward gathering process in a bottom-up order. For each internal node, the bounding boxes and intensities of the child nodes are simply added together. Bogus lights are assigned zero intensities and excluded from the bounding box computation. Since this gathering process is fully deterministic and commutative, we can generate level ℓ from any level ℓ_s below it, such that $\ell_s > \ell$, not necessarily the level immediately below it (where $\ell_s = \ell + 1$). This means that any combination of levels can be generated in parallel, and the construction process can be parallelized in a way that fully exploits the capability of the GPU.

While our perfectly balanced light tree is fast enough to be rebuilt for every frame, we also provide an option to use a two-level light tree, where a light tree is split into one *top-level* and multiple *bottom-levels*. This is to improve the quality of light sampling for scenes with sparsely distributed light meshes of heterogeneous sizes, where Morton code sorting on all lights might not reproduce the spatial proximity very well. In a two-level light tree, each bottom-level is built separately as a perfect binary light tree. After all bottom-levels are built, they are used as leaf nodes of the top-level tree, which is also built as a perfect binary tree. Notice that in this case the combined light tree as a whole is not necessarily a perfect tree. Furthermore, two level tree introduces complexity in light tree traversal, but it can improve the sampling quality. An additional benefit of a two-level light tree is that it allows instancing.

3.2 Light Tree Traversal

During rendering, we first select a cut through the light tree, similar to the original lightcuts method [Walter et al. 2005]. Yet, unlike lightcuts, we cannot afford to pick a cut with hundreds of nodes, which would result in too many light samples to achieve real-time performance. Instead, we pick a cut with a relatively small, user-defined number of nodes. Therefore, cut selection is not performed until convergence with a given error threshold. In fact, given a small number of nodes for the cut, it is almost guaranteed that we can never satisfy a reasonable error threshold. Thus, we do not use an error threshold parameter and our cut selection terminates until a user-defined number of nodes (i.e. subtrees) are selected.

Given any subtree root, it is easy to select a light sample by traversing our perfect light tree down to a leaf node. We use the hierarchical importance sampling approach of stochastic lightcuts [Yuksel 2019]. At each internal node, one of its child nodes is randomly selected based on their weights computed at the shaded point. To improve the sampling quality in our test scenes, we introduce a novel weight computation scheme for hierarchical importance sampling.

Ideally, the weights should be proportional to the expected illuminations of the child nodes. However, when the shaded point is inside the bounding box of a node, its expected illumination goes to infinity. Yuksel [2019] solves this problem by effectively ignoring the distance term when the shaded point is too close to either one of the child node bounding boxes. We use a different strategy. Since the expected illumination can go to infinity, we approximate the expected weights by computing two weights at two different distances from the shaded point: the closest distance d_j^{\min} and the farthest distance d_j^{\max} within the bounding box of the child node j , such that

$$w_j^{\min} = \frac{F_j(\mathbf{x}, \omega) \|I_j\|}{(d_j^{\min}(\mathbf{x}))^2} \quad w_j^{\max} = \frac{F_j(\mathbf{x}, \omega) \|I_j\|}{(d_j^{\max}(\mathbf{x}))^2}, \quad (1)$$

where \mathbf{x} is the shaded point, $F_j(\mathbf{x}, \omega)$ is the reflectance bound, and I_j is the total light intensity within the node. Note that when the child node bounding boxes are relatively small and far from the shaded point, w_j^{\min} and w_j^{\max} approach to the same values. Given two child nodes j and k , we compute two probabilities for picking j as

$$p_j^{\min} = \frac{w_j^{\min}}{w_j^{\min} + w_k^{\min}} \quad p_j^{\max} = \frac{w_j^{\max}}{w_j^{\max} + w_k^{\max}}. \quad (2)$$

We use the average of these two $p_j = (p_j^{\min} + p_j^{\max})/2$ as the probability of picking child node j . The only singularity with this approach is when both d_j^{\min} and d_k^{\min} are zero (i.e. \mathbf{x} is within the bounding box of both child nodes), in which case we ignore the distance terms for computing w_j^{\min} and w_k^{\min} . In our tests, we have found that this new weight computation scheme can improve the quality of light sampling.

Note that a bogus node (or a bogus light) has zero probability to be selected due to its zero intensity. When a *dead branch* is detected, we simply return a light with zero intensity. The intensity of the selected light is divided by the selection probability, and shadows are computed via ray tracing on the GPU.

3.3 Light Sampling with Cut Sharing

Cut selection of lightcuts performed independently for each pixel can be expensive. Yet, neighboring pixels often share the same cuts. This is particularly the case when the number of light samples (i.e. the number of nodes above the cut) is small. Based on this observation, we can accelerate cut selection by performing it for a group of pixels, rather than independently for each pixel. Thus,

a group of nearby pixels share the same cut through the light tree. We call this *cut sharing*. Note that cut sharing does not cause sampling correlation, since pixels within a group are still free to pick different light samples using the same cut.

Cut sharing can be broken down into two passes. First, a cut computation pass is executed once for each $k \times k$ pixel block where one of the pixels is randomly selected as the representative pixel whose geometric and material properties are used to select the cut. In the second pass, we perform light sampling, such that each pixel in the $k \times k$ block uses the same cut.

Cut sharing accelerates light sampling with a smaller additional memory footprint for storing the cut. Notice that our cut sharing method bears some resemblance to Adaptive Direct Illumination Sampling [Vévoda and Krivánek 2016], where a cut is shared by each scene cell in a 3D grid. In comparison, our screen-space cut sharing enables efficient GPU implementation and screen-space subsampling as explained below.

3.4 Interleaved Sampling

Since the cut sharing technique partitions all scene lights into disjoint light clusters (i.e. light subtrees under the cut) for every $k \times k$ pixel block, interleaved sampling of many lights [Segovia et al. 2006; Wald et al. 2002] can be naturally used here, such that each pixel in an $m \times m$ sub-block only samples from a subset of all light clusters, where $m \leq k$ and k is a multiple of m . Note that, though we use square blocks and sub-blocks, they can be rectangular as well. The lighting contribution from the lights within a sub-block is later shared with the neighboring pixels in a reconstruction process.

Interleaved sampling [Segovia et al. 2006; Wald et al. 2002] partitions all light sources into m^2 subsets. Each subset is assigned to one pixel within a block of $m \times m$ pixels. The lighting for each pixel is computed using its subset, reducing the number of light sources used per pixel by $1/m^2$. After computing lighting, a blurring kernel filters the irradiance buffer to remove the structured noise artifacts, using a discontinuity buffer to avoid blurring across geometric edges.

In our method, we instead partition the light subtrees under the chosen cut. Each pixel within a sub-block of $m \times m$ pixels receives a subset of the light subtrees and samples only those subtrees. Note that both interleaved sampling (using a subset of the light subtrees) and light sampling (randomly picking lights within each subtree) lead to noise in lighting estimation. Therefore, a simple blurring kernel (like a Gaussian filter used in prior work) may not be sufficient to clear the noise. Instead a geometry-aware denoiser, like SVGF [Schied et al. 2017], can be applied using more aggressive parameters (such as more blending passes) than typical settings used without interleaved sampling.

The benefit of interleaved sampling is that it can effectively increase the number of light clusters (i.e. the number of nodes along the cut) for a local pixel neighborhood without increasing the number of light samples per pixel. This way, we can process a deeper cut, which often leads to higher quality. Furthermore, deeper cuts would also mean fewer steps needed for hierarchical importance sampling to reach a leaf node, which improves the sampling performance.

4 IMPLEMENTATION DETAILS

For generating our perfect light tree (or a bottom-level tree when using a two-level light tree), we first sort all light primitives (points or triangles) based on their centroid positions, using a 30-bit Morton code with 10 bits for each of the x, y, and z coordinates. The centroid positions are quantized using the bounding box of all light sources in the tree (i.e. the global light bounds). We use parallel bitonic sort [Batcher 1968] on 64-bit key-value pairs to produce a sorted index list. This sorted index list is used to populate the leaf level. In our implementation we limit the maximum number of light primitives in each leaf node to 1.

After generating the leaf level, we generate the internal levels in groups. All levels in a group are built in parallel. Given source level ℓ_s , and d destination levels $\ell_s + 1$ to $\ell_s + d$, each destination

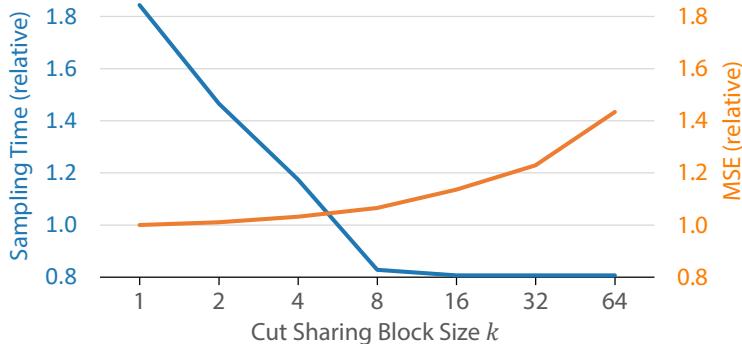


Fig. 3. The relationship between cut sharing block size k and sampling time (blue line) and MSE (orange line), generated using the camera view and sample count in Figure 9. The values are relative to the sampling time and MSE of the image rendered without cut sharing (lower is better).

level is built directly from the source level ℓ_s . Thus, each destination node at level ℓ is formed by merging $2^{\ell-\ell_s}$ source nodes.

In our implementation, the nodes of perfect binary trees do not store light orientation bounding cones. While this overestimates the geometry term used for computing cluster error bounds and sampling weights, we observed only subtle loss of sampling quality in our tests. On the other hand, skipping the cone storage allows us to pack the entire data per light tree node into 32 bytes and helps to align the light tree levels to 64 byte cache lines.

For cut sharing, we use $k = 8$ as the block size for a balance between speed and quality. In our test, we have observed that using a block size smaller than 8 introduced a significant overhead, resulting in slower rendering than not using cut sharing. On the other hand, using a larger block size than 8 rapidly increased the MSE with only minor improvement in sampling time (Figure 3). With a block size of 8, the interleaved sampling sub-block size m can be chosen as 2, 4, or 8.

Using n light samples per pixel, we pick a cut with n subtrees shared by an entire block. Our interleaved sampling implementation distributes the n subtrees to the pixels within a sub-block as evenly as possible. Each pixel with index p within an $m \times m$ sub-block (i.e. $p \in \{0, 1, \dots, m^2 - 1\}$) uses the subtrees t_p through $t_{p+1} - 1$, where $t_p \in \{0, 1, \dots, n - 1\}$, such that

$$t_p = \left\lfloor \frac{pn}{m^2} \right\rfloor. \quad (3)$$

In our implementation, we also rotate the light cluster assignment of pixels within a sub-block, so that the same pixel does not get the same clusters every frame. This allows achieving better quality with spatiotemporal filtering. At each frame f , each pixel i within a sub-block is assigned an index of $p = i + f \pmod{m^2}$.

5 RESULTS

We present test results in different scenes with many lights. Our results do not include interleaved sampling (Section 3.4), unless otherwise stated. We compute the flux of textured emitters using the method introduced by Moreau and Clarberg [2019]. All scenes but Lumberyard Bistro use a one-level light tree. The *Amazon Lumberyard Bistro* scene uses a two-level light tree mentioned in Section 3.1 to improve the sampling quality since the mesh lights have more irregular shapes than other scenes. All scenes only evaluate direct lighting from the light sources or virtual lights. Direct lighting samples are only generated by sampling the lights, without multiple importance

Table 1. Breakdown of the light tree construction time.

	<i>Crytek Sponza</i>	<i>Cornell Box</i>
Geometry update	N/A (0%)	0.01 ms (6%)
Compute bounds	0.03 ms (7%)	0.02 ms (10%)
Morton code generation	0.01 ms (3%)	0.01 ms (4%)
Sorting	0.28 ms (63%)	0.07 ms (45%)
Building the leaf level	0.03 ms (8%)	0.01 ms (5%)
Building internal levels	0.08 ms (19%)	0.05 ms (29%)
Total Time	0.43 ms (100%)	0.15 ms (100%)

sampling. All random numbers used for sampling are generated using the GPU Tiny Encryption Algorithm [Zafar et al. 2010]. The results are rendered at 1920×1080 resolution using an NVIDIA RTX 2080 graphics card on a computer with an Intel Core i7-8700K CPU and 16GB RAM. We implemented our algorithm using the Direct3D 12 graphics API with DirectX Raytracing (DXR) capability. All timings are averaged over 512 frames.

5.1 Light Tree Construction

We present the breakdown of light tree construction time in Table 1 using two scenes. In the *Crytek Sponza* scene, we generate approximately 100,000 virtual point lights from a single sun (point) light by periodically varying the sun angle via ray tracing in every frame. The *Cornell Box* scene includes 12 animated mesh lights with 12,146 emissive triangles. The bounding box of all lights is first computed using parallel reduction. Notice that sorting is the bottleneck of our light tree construction.

Our light tree construction on the GPU is more than two orders of magnitude faster than the agglomerative clustering on the CPU that produces an unbalanced tree. Agglomerative clustering takes 251 ms for Crytek Sponza and 22 ms for Cornell Box on CPU, while our tree construction takes only 0.43 ms and 0.15 ms on the GPU, respectively.

Note that it is possible to construct an unbalanced light tree using divisive clustering [Walter et al. 2008], which splits nodes by spatial median in a top-down order. Divisive clustering is faster than agglomerative clustering and more suitable for a GPU implementation. Nonetheless, divisive clustering is not expected to produce a light tree with higher quality that would lead to lower noise in light sampling, as compared to agglomerative clustering. Also, since it would produce an unbalanced tree, it would not have the sampling efficiency of our perfect trees, even if it could be optimized to achieve construction speeds closer to our method.

5.2 Comparison to Prior Work

We compare our method to a real-time implementation of adaptive tree splitting (ATS) [Moreau et al. 2019]. The light BVH for ATS is built and updated on the CPU using surface area orientation heuristic. Our method uses a single-level perfect light tree, which is rebuilt every frame on the GPU.

One advantage of our method is that by improving the efficiency of light sampling, we can use more light samples. This is demonstrated in the example in Figure 4. Using (approximately) the same light sampling time (excluding the light tree construction time) in this scene, our method can process 11 light samples during the time it takes for ATS to process 5 light samples per pixel. As a result, the noise (prior to filtering) is substantially less with our method.

Another example comparing our method to ATS for equal sampling time (excluding light tree construction time) is shown in Figure 5. Again, our method is able to process more light samples

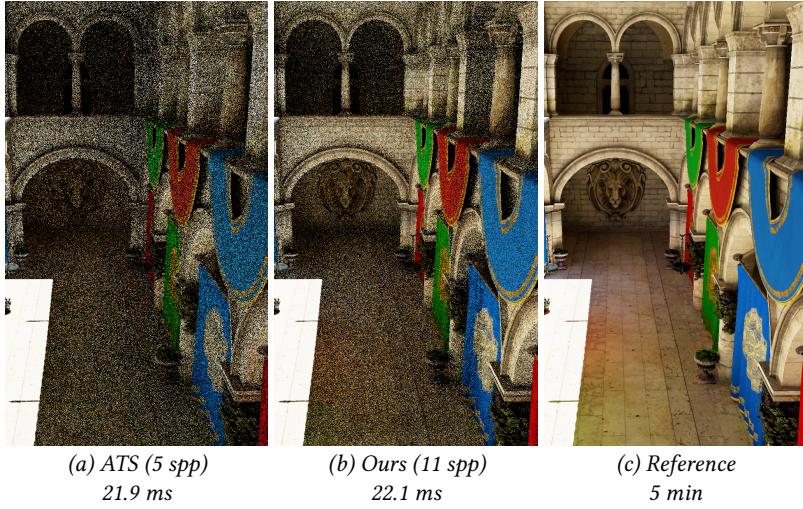


Fig. 4. Equal sampling time (excluding construction time) comparison between (a) a real-time implementation of Adaptive Tree Splitting (ATS) [Moreau et al. 2019] and (b) our method, using a dynamic lighting condition with 100,000 VPLs in the Crytek Sponza scene.

within the same sampling time, resulting in lower noise. Figure 6 shows a close-up of the same scene before and after filtering. Notice that the noise with ATS is not entirely eliminated after filtering and presents itself as lower-frequency noise.

The comparison results are summarized in Table 2, along with ATS results using equal sample count. Notice that for the *Crytek Sponza* scene the light tree update time for ATS on the CPU is orders of magnitude slower than our light tree construction on the GPU. For the *Cornell Box* scene, on the other hand, we use a two-level light tree with ATS and only the small top-level tree is updated every frame and the bottom-level trees (one for each mesh light) remain constant, significantly reducing the light tree update time of ATS. In comparison, our light tree construction still works multiple times faster, even though it rebuilds an entire (single-level) light tree for every frame from scratch on the GPU. The table also includes mean square error (MSE) and structural similarity index (SSIM) values. Note that the tree quality of ATS allows it to achieve lower noise (i.e. lower MSE and SSIM). This advantage of ATS is due to the fact that we use a perfect tree that delivers lower sampling quality. On the other hand, ATS requires substantially longer sampling time to process as many samples as our method.

Table 2. Comparison to real-time Adaptive Tree Splitting (ATS).

		Light Tree Update	Sampling Time	Sample Count	Average MSE	Average SSIM
	ATS	CPU: 176.6 ms	21.9 ms	5	0.090	0.223
	ATS	CPU: 176.6 ms	48.1 ms	11	0.058	0.277
	Ours	GPU: 0.4 ms	22.1 ms	11	0.064	0.254
	ATS	CPU: 0.6 ms	18.4 ms	4	0.084	0.168
	ATS	CPU: 0.6 ms	58.9 ms	13	0.033	0.222
	Ours	GPU: 0.2 ms	18.1 ms	13	0.040	0.200

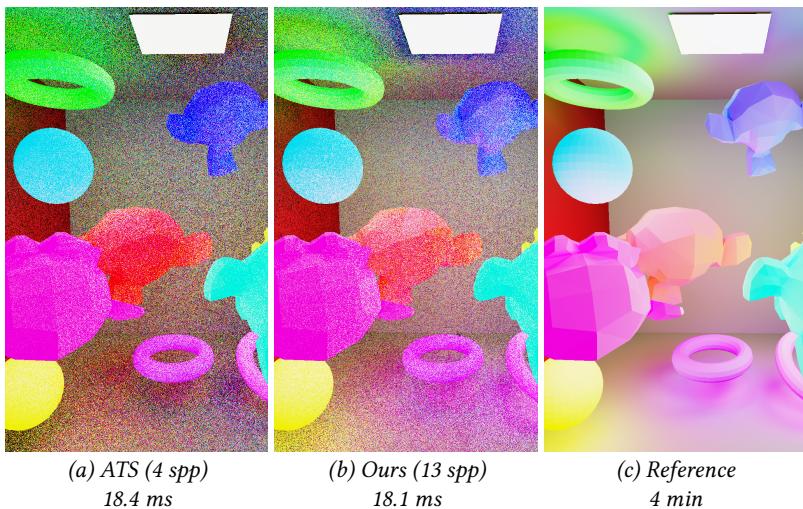


Fig. 5. Equal sampling time (excluding light tree construction time) comparison between (a) a real-time implementation of Adaptive Tree Splitting (ATS) [Moreau et al. 2019] and (b) our method, with dynamic illumination in the Cornell Box scene.

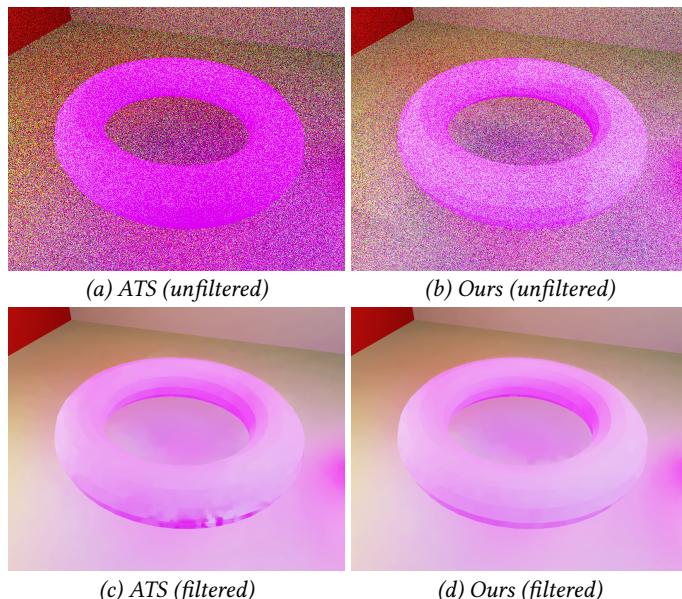


Fig. 6. Equal sampling time (≈ 18 ms) comparison between (a) a real-time implementation of Adaptive Tree Splitting (ATS) [Moreau et al. 2019] and (b) our method in the Cornell Box scene. (c-d) the bottom row shows the results after filtering with SVGF ($\alpha = 0.2$) [Schied et al. 2017]. Noise on the lower part of the torus in the filtered ATS result is apparent even after temporal accumulation of samples.

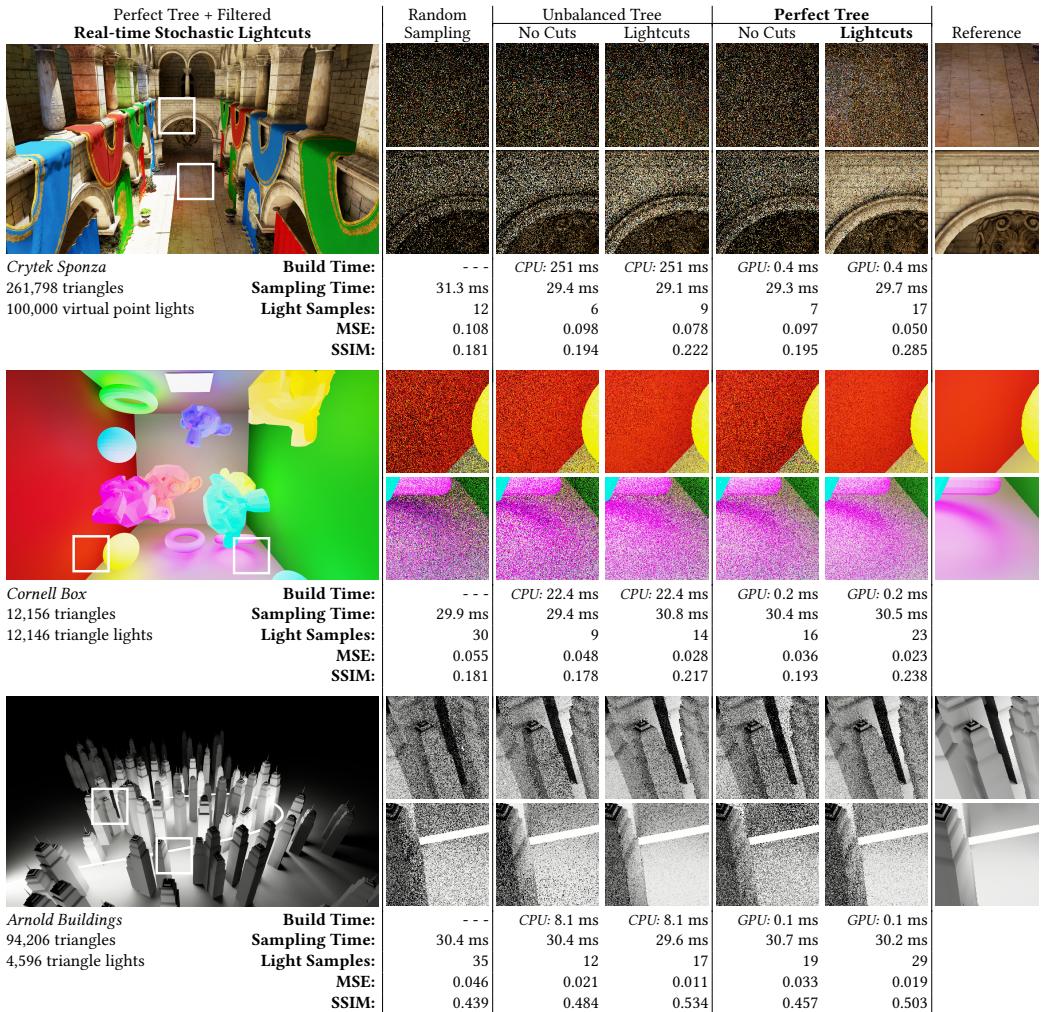


Fig. 7. Visual and quantitative comparison of light sampling methods with (approximately) equal sampling time. The unbalanced trees are built on the CPU using agglomerative clustering and perfect trees are build using our method on the GPU.

5.3 Evaluation of Light Sampling Strategies

In Figures 7 & 8 we present (approximately) equal sampling time comparisons of our real-time stochastic lightcuts method to two alternatives: uniform *random sampling* without a light tree and sampling directly from a light tree with *no cuts* (i.e. without selecting a cut). Sampling with no cuts uses hierarchical importance sampling, starting from the root of the tree for each light sample, using our weight computation scheme. Note that directly sampling a light tree is the approach used in some recent work [Moreau and Clarberg 2019; Moreau et al. 2019]. In comparison, our method uses stochastic *lightcuts* with cut sharing; thus, hierarchical importance sampling traverses the selected subtrees. In these Figures we also use two alternatives for light trees: *unbalanced tree* generated using agglomerative clustering on the CPU and *perfect tree* generated using our method on the GPU. The nodes of the unbalanced trees contain orientation bounding cones and child

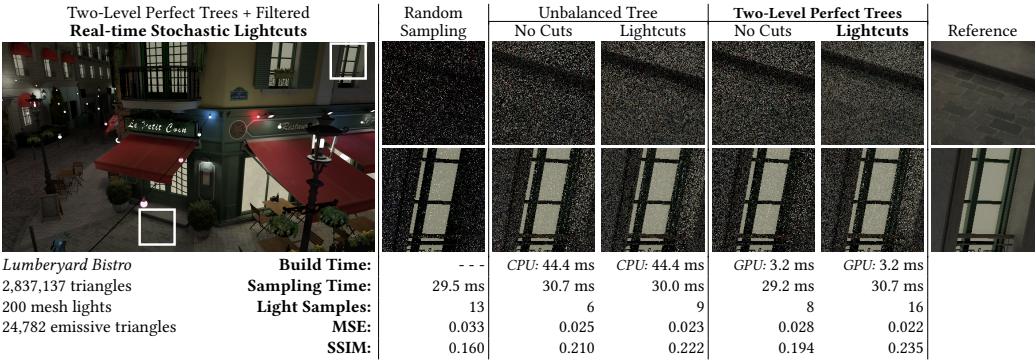


Fig. 8. Visual and quantitative comparison of light sampling methods with (approximately) equal sampling time. The unbalanced trees are built on the CPU using agglomerative clustering and perfect trees are build using our method on the GPU.

index offsets, which provide better sampling quality but some reduction in sampling speed (in addition to the extended build time). The examples in Figure 7 use single-level perfect trees and the example in Figure 8 uses a two-level perfect tree (the bottom-level perfect trees are built on the CPU in our implementation). The reference images are generated using stochastic lightcuts results with 65,536 total samples per pixel.

Notice that in all four scenes the highest light sample count is achieved by either our stochastic lightcuts method with perfect tree or random sampling. While uniform random sampling cannot effectively make use of the relatively high sample count, our method leads to a relatively low noise solution. The only exception is the *Arnold Buildings* scene, where the quality improvement of using an unbalanced tree makes a significant-enough improvement, so that our stochastic lightcuts method with an unbalanced tree provides the best quality, even by using fewer light samples than our method with a perfect light tree. This shows the performance gain of using a perfect light tree may not always offset the potential reduction in the tree quality. However, this comparison does not include the substantially longer build time of the unbalanced tree. Notice that our method benefits from the fast sampling speed which allows it to use 1.4× - 2.8× more samples than other methods. On the other hand, methods based on unbalanced light trees have prohibitively expensive built time (8.1 ms - 251.0 ms) which make them impractical to use for fully dynamic scenes.

We provide numerical comparisons using equal sample count (8 light samples per pixel) in Table 3, showing total frame render time, sampling time (including cut selection), mean square error (MSE), and structural similarity index (SSIM). Notice that our method with a perfect tree is faster in both total frame time and sampling time than all alternatives, except for random sampling. In fact, our method delivers a higher sampling speed than random sampling in the *Crytek Sponza* scene, which is likely due to more coherent shadow rays. However, the quality improvement over random sampling (as can be seen by the MSE and SSIM numbers) is substantial, as expected.

Also notice that, using 8 light samples per pixel, there is little numerical difference in quality between using lightcuts and directly sampling the light tree with no cuts. Indeed, in the *Arnold Buildings* scene, using no cuts actually provides slightly better quality. This is mainly because, when using relatively few light samples, the selected cut may not always provide a good clustering of lights and some light trees may have substantially stronger illumination than others. Therefore, using one light sample per subtree, as we use in our stochastic lightcuts approach, does not always provide

Table 3. Comparisons using 8 light samples per pixel.

	Random Sampling	Unbalanced Tree		Perfect Tree	
Frame Time		No Cuts	Lightcuts	No Cuts	Lightcuts
<i>Crytek Sponza</i>	29.9 ms	300.1 ms	287.9 ms	43.7 ms	27.4 ms
<i>Cornell Box</i>	13.8 ms	53.1 ms	44.7 ms	22.3 ms	19.1 ms
<i>Arnold Buildings</i>	12.7 ms	26.5 ms	21.5 ms	18.9 ms	15.9 ms
<i>Lumberyard Bistro</i>	28.1 ms	52.0 ms	36.9 ms	39.9 ms	28.7 ms
Sampling Time					
<i>Crytek Sponza</i>	20.8 ms	39.3 ms	26.1 ms	33.4 ms	17.4 ms
<i>Cornell Box</i>	7.7 ms	25.7 ms	18.6 ms	14.8 ms	11.8 ms
<i>Arnold Buildings</i>	6.4 ms	19.9 ms	15.1 ms	12.4 ms	9.8 ms
<i>Lumberyard Bistro</i>	18.1 ms	41.0 ms	26.9 ms	29.2 ms	18.7 ms
MSE					
<i>Crytek Sponza</i>	0.119	0.088	0.093	0.092	0.085
<i>Cornell Box</i>	0.123	0.053	0.048	0.070	0.065
<i>Arnold Buildings</i>	0.107	0.030	0.039	0.063	0.065
<i>Lumberyard Bistro</i>	0.035	0.022	0.025	0.028	0.028
SSIM					
<i>Crytek Sponza</i>	0.170	0.208	0.205	0.201	0.214
<i>Cornell Box</i>	0.155	0.173	0.179	0.162	0.165
<i>Arnold Buildings</i>	0.369	0.451	0.436	0.403	0.401
<i>Lumberyard Bistro</i>	0.149	0.232	0.208	0.195	0.188

the best sample distribution. Nonetheless, the performance improvement due to cut sharing allows using more light samples per pixel, thereby improving the final result within the same render time.

Moreover, using an unbalanced tree often improves the sampling quality with the same number of samples. On the other hand, the cost of building an unbalanced tree and sampling it leads to using fewer light samples within the same render time.

5.4 Light Sampling Improvements

Our cut sharing method (Section 3.3) can provide substantial reduction in sampling time. An example of this is shown in Figure 9, comparing sampling quality and performance with and without cut sharing. Notice that images with cut sharing can include visual artifacts prior to filtering, particularly near depth discontinuities (Figure 9a). Yet, after applying spatiotemporal filtering, it produces indistinguishable filtered results (Figure 9c). In terms of performance, cut sharing provides about 20% speedup in sampling time in this example, including the overhead of the cut selection computation, which takes less than 2% of the light sampling time.

As we explain in Section 3.1 our method permits using two-level light trees. Using a two-level light tree introduces some additional cost, both in light tree construction and light sampling. On the other hand, depending on the scene, the improvement over using a single perfect tree can be substantial. We demonstrate this in Figure 10 using the *Amazon Lumberyard Bistro* scene. In this case, using a two-level light tree slightly increases the light sampling time from 17.4 ms to 18.7 ms, but it also reduces the noise. A two-level tree is expected to provide some improvement

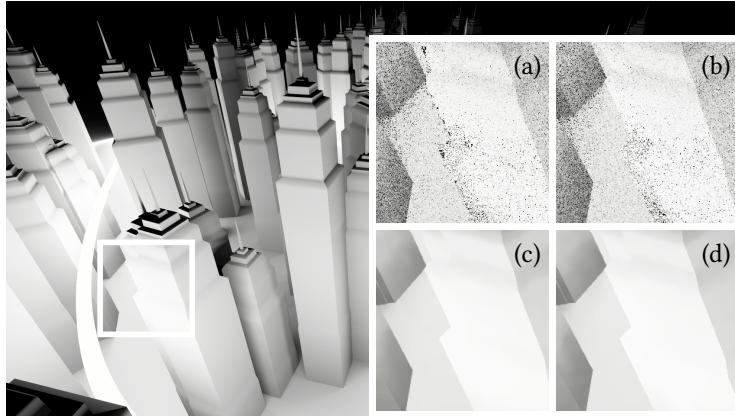


Fig. 9. Images rendered with and without cut sharing using 32 light samples per pixel: (a) cut sharing with $k = 8$ (sampling time: 31.3 ms) before filtering, (b) no cut sharing (sampling time: 37.8 ms) before filtering, (c) cut sharing after filtering, and (d) no cut sharing after filtering. The filtered results are generated using an SVGF filter [Schied et al. 2017] with $\alpha = 0.2$ and accumulation of 5 frames.

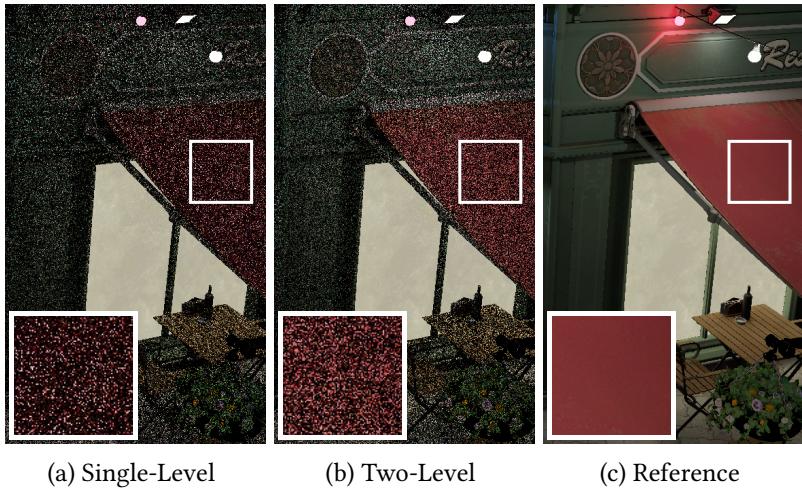


Fig. 10. The quality improvement of using a two-level tree: images for both (a) single-level and (b) two-level trees are rendered using stochastic lightcuts with 8 light samples per pixel.

in quality and some reduction in performance, but whether a two-level tree would be beneficial over a single-level perfect tree depends on the scene.

As we explain in Section 3.4, our cut sharing approach allows using interleaved sampling for further reducing the sampling cost or increasing the effective sample count. The example in Figure 11 shows the *Amazon Lumberyard Bistro* scene rendered with and without interleaved sampling, using 8 samples per pixel. In this case, interleaved sampling uses a sub-block size of 2×2 . Therefore, each cut for each sub-block contains 32 light samples. As a result, interleaved sampling can approximate the quality of using 32 light samples by computing 8 light samples per pixel. Note that interleaved sampling is particularly effective when combined with spatiotemporal filtering.

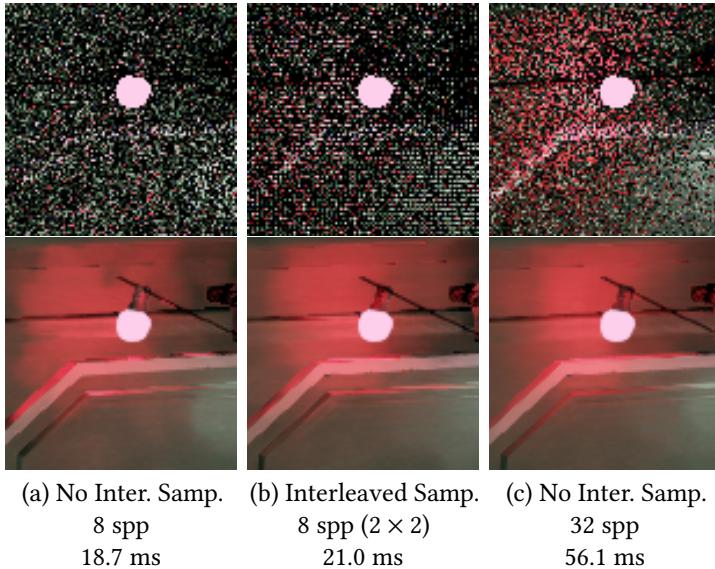


Fig. 11. *Interleaved sampling:* (a) no interleaved sampling with 8 samples per pixel, (b) interleaved sampling within a 2×2 sub-block with 8 samples per pixel, approximating 32 samples per pixel, and (c) no interleaved sampling with 32 samples per pixel. The images on the bottom row show results filtered results with SVGF [Schied et al. 2017] without temporal accumulation (to amplify the difference of quality). The images are rendered using the screen and camera configuration of Lumberyard Bistro in Figure 8.

Figure 12 compares the performance of hierarchical importance sampling using the new weight computation method we introduced in Section 3.2 to the weight computation scheme in the stochastic lightcuts paper [Yuksel 2019]. Notice that in all our test scenes our new weights computation provides a minor but visible improvement in sampling quality.

6 LIMITATIONS

While our perfect binary trees can be built extremely fast on the GPU using Morton codes, the resulting light trees lead to more noise as compared to unbalanced light trees generated with agglomerative clustering. This is because using a balanced tree and only considering spatial proximity when partitioning the nodes limits the quality of the light tree. Although the improvement in sampling speed with perfect trees can compensate for the loss in tree quality in our tests, they might be less effective in some scenes.

While our cut sharing technique has been effective in our tests, it might lead to visual artifacts in some scenes. In particular, specular highlights on highly glossy surfaces near cut sharing block boundaries may form discontinuities that would be difficult to detect with a geometry-aware filter. Furthermore, since our cut sharing method uses screen-space blocks, the chosen cut for a block may not be ideal for all pixels of the block, particularly near depth discontinuities, and it is not suitable for more general applications, like path tracing or ray traced reflections.

7 CONCLUSION

We have presented a real-time light sampling technique for scene with many lights. Our method extends stochastic lightcuts by using a perfect binary light tree with a novel weight computation

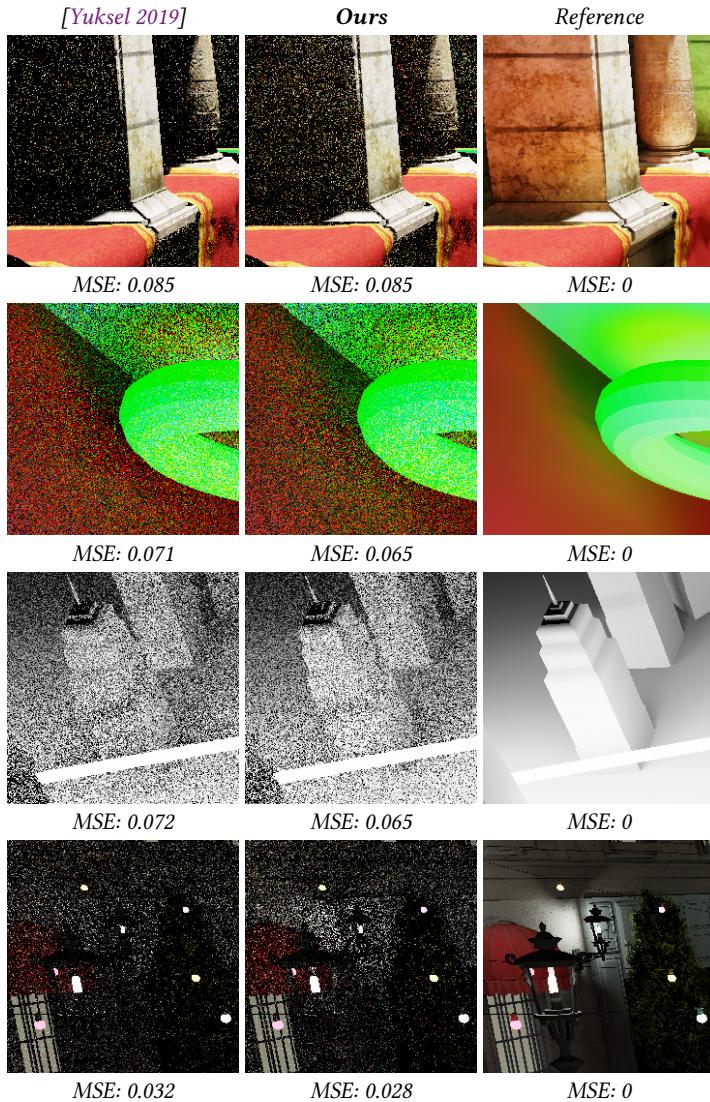


Fig. 12. Hierarchical importance sampling using the weight computation scheme of Yuksel [2019] and our new weight computation method (Section 3.2).

scheme and cut sharing. By minimizing the cost of light sampling, our method allows using more light samples within the same render time for achieving higher sampling quality. Since our method does not restrict the types of lights that can be sampled and the light tree can be constructed efficiently every frame, we can accommodate fully dynamic scenes with a variety of light types.

ACKNOWLEDGMENTS

We thank Morgan McGuire [2017] for providing the remastered version of Amazon Lumberyard Bistro we used in our tests and the anonymous reviewers for their helpful feedback.

REFERENCES

- Kenneth E Batcher. 1968. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 307–314.
- Carsten Dachsbacher, Jaroslav Křivánek, Miloš Hašan, Adam Arbree, Bruce Walter, and Jan Novák. 2014. Scalable Realistic Rendering with Many-Light Methods. *Computer Graphics Forum* 33, 1 (2014), 88–104.
- Tomáš Davidovič, Iliyan Georgiev, and Philipp Slusallek. 2012. Progressive lightcuts for GPU. In *ACM SIGGRAPH 2012 Talks*. ACM, 1.
- T Davidovic, J Krivnek, M Hasan, P Slusallek, and K Bala. 2010. Combining global and local lights for high-rank illumination effects. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)* 29, 5 (2010).
- Alejandro Conty Estevez and Christopher Kulla. 2018. Importance sampling of many lights with adaptive tree splitting. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1, 2 (2018), 25.
- Sebastian Fernandez, Kavita Bala, and Donald P Greenberg. 2002. Local Illumination Environments for Direct Lighting Acceleration. *Rendering Techniques* 2002 (2002), 13th.
- Miloš Hašan, Fabio Pellacini, and Kavita Bala. 2007. Matrix row-column sampling for the many-light problem. *ACM Transactions on Graphics (TOG)* 26, 3 (2007), 26.
- Miloš Hašan, Edgar Velázquez-Armendariz, Fabio Pellacini, and Kavita Bala. 2008. Tensor Clustering for Rendering Many-light Animations. In *Proceedings of Eurographics Workshop on Rendering*. 1105–1114.
- Yuchi Huo, Rui Wang, Shihao Jin, Xinguo Liu, and Hujun Bao. 2015. A matrix sampling-and-recovery approach for many-lights rendering. *ACM Transactions on Graphics (TOG)* 34, 6 (2015), 210.
- Alexander Keller. 1997. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co., 49–56.
- Alexander Keller, Carsten Wächter, Matthias Raab, Daniel Seibert, Dietger van Antwerpen, Johann Korndörfer, and Lutz Kettner. 2017. The iray light transport simulation and rendering system. In *ACM SIGGRAPH 2017 Talks*. ACM, 34.
- Christian Lauterbach, Michael Garland, Shubhabrata Sengupta, David Luebke, and Dinesh Manocha. 2009. Fast BVH construction on GPUs. In *Computer Graphics Forum*, Vol. 28. Wiley Online Library, 375–384.
- Daqi Lin and Cem Yuksel. 2019. Real-Time Rendering with Lighting Grid Hierarchy. *Proc. ACM Comput. Graph. Interact. Tech. (Proceedings of I3D 2019)* 2, 1, Article 8 (2019), 17 pages.
- Morgan McGuire. 2017. Computer Graphics Archive. <https://casual-effects.com/data>
- Pierre Moreau and Petrik Clarberg. 2019. Importance Sampling of Many Lights on the GPU. In *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*, Eric Haines and Tomas Akenine-Möller (Eds.). Apress, Berkeley, CA, 255–283.
- Pierre Moreau, Matt Pharr, and Petrik Clarberg. 2019. Dynamic Many-Light Sampling for Real-Time Ray Tracing. In *High-Performance Graphics - Short Papers*, Markus Steinberger and Tim Foley (Eds.). The Eurographics Association.
- Ola Olsson and Ulf Assarsson. 2011. Tiled shading. *Journal of Graphics, GPU, and Game Tools* 15, 4 (2011), 235–251.
- Ola Olsson, Markus Billeter, and Ulf Assarsson. 2012. Clustered deferred and forward shading. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. Eurographics Association, 87–96.
- Jiawei Ou and Fabio Pellacini. 2011. LightSlice: matrix slice sampling for the many-lights problem. *ACM Trans. Graph.* 30, 6 (2011), 179–1.
- Eric Paquette, Pierre Poulin, and George Drettakis. 1998. A Light Hierarchy for Fast Rendering of Scenes with Many Lights. *Computer Graphics Forum* 17, 3 (1998), 63–74.
- Hauke Rehfeld and Carsten Dachsbaucher. 2016. Lightcut interpolation. In *Proceedings of High Performance Graphics*. 99–108.
- Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbaucher, Aaron Lefohn, and Marco Salvi. 2017. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*. ACM, 2.
- Benjamin Segovia, Jean Claude Iehl, Richard Mitanchey, and Bernard Péroche. 2006. Non-interleaved deferred shading of interleaved sample patterns. In *Graphics Hardware*. 53–60.
- Peter Shirley, Changyaw Wang, and Kurt Zimmerman. 1996. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics (TOG)* 15, 1 (1996), 1–36.
- Yusuke Tokuyoshi and Takahiro Harada. 2016. Stochastic light culling. *Journal of Computer Graphics Techniques* Vol 5, 1 (2016).
- Petr Vévoda, Ivo Kondapaneni, and Jaroslav Křivánek. 2018. Bayesian online regression for adaptive direct illumination sampling. *ACM Transactions on Graphics (TOG)* 37, 4 (2018), 125.
- Petr Vévoda and Jaroslav Křivánek. 2016. Adaptive direct illumination sampling. In *SIGGRAPH ASIA 2016 Posters*. ACM, 43.
- Ingo Wald, Thomas Kollig, Carsten Benthin, Alexander Keller, and Philipp Slusallek. 2002. Interactive global illumination. (2002).
- Bruce Walter, Adam Arbree, Kavita Bala, and Donald P. Greenberg. 2006. Multidimensional Lightcuts. *ACM Transactions on Graphics (Proceedings of SIGGRAPH '06)* 25, 3 (2006), 1081–1088.

- Bruce Walter, Kavita Bala, Milind Kulkarni, and Keshav Pingali. 2008. Fast agglomerative clustering for rendering. In *2008 IEEE Symposium on Interactive Ray Tracing*. IEEE, 81–86.
- Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P Greenberg. 2005. Lightcuts: a scalable approach to illumination. *ACM Transactions on graphics (TOG)* 24, 3 (2005), 1098–1107.
- Bruce Walter, Pramook Khungurn, and Kavita Bala. 2012. Bidirectional Lightcuts. *ACM Transactions on Graphics* 31, 4, Article 59 (2012), 11 pages.
- Gregory J Ward. 1994. Adaptive shadow testing for ray tracing. In *Photorealistic Rendering in Computer Graphics*. Springer, 11–20.
- Cem Yuksel. 2019. Stochastic Lightcuts. In *High-Performance Graphics (HPG 2019)*. The Eurographics Association.
- Can Yuksel and Cem Yuksel. 2017. Lighting grid hierarchy for self-illuminating explosions. *ACM TOG (Proc. SIGGRAPH)* 36, 4 (2017), 110.
- Fahad Zafar, Marc Olano, and Aaron Curtis. 2010. GPU random numbers via the tiny encryption algorithm. In *Proceedings of the Conference on High Performance Graphics*. Eurographics Association, 133–141.