Lightweight Multidimensional Adaptive Sampling for GPU Ray Tracing

Daniel Meister
The University of Tokyo

Toshiya Hachisuka
University of Waterloo

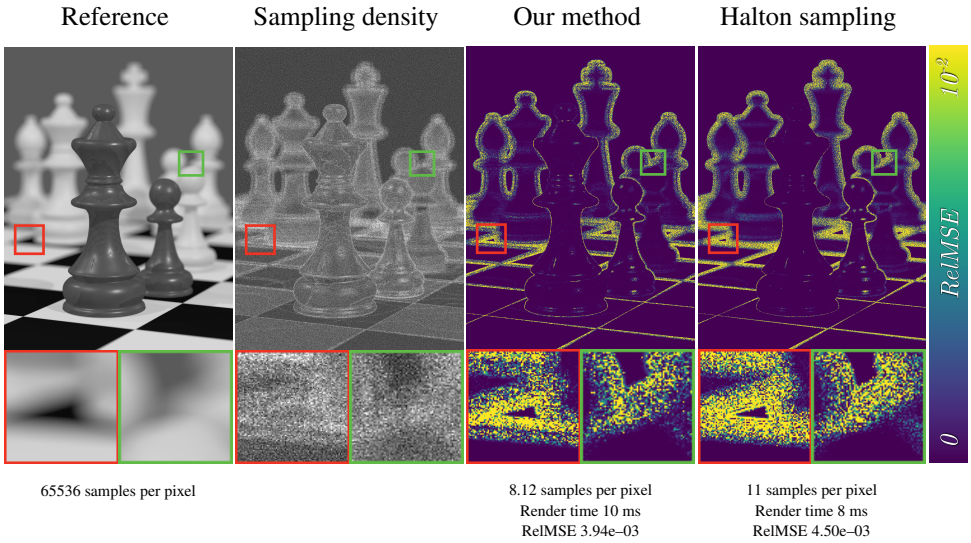

Figure 1. Comparison of our method and Halton sampling for the Chess scene with depth of field. Our method adaptively places samples into the regions of rapid changes (i.e., high-frequency discontinuities or significant depth-of-field effect). Our method provides visually smoother results with slightly lower error in equal time. Sampling density shows projected sample distribution on the image plane.

Abstract

Rendering typically deals with integrating multidimensional functions that are usually solved through numerical integration, such as Monte Carlo or quasi-Monte Carlo. Multidimensional adaptive sampling [Hachisuka et al. 2008] is a technique that can significantly reduce the error by placing samples into locations of rapid changes. However, no efficient parallel version is available, limiting its practical utility in interactive and real-time applications. We reformulate the algorithm by exploiting the fact that different locations can be sampled in parallel to be suitable for modern GPU architectures. We start by placing a fixed number of initial samples

in each cell of a uniform grid, where these initial samples are subsequently used for error estimation to adaptively refine the cells with higher error. We implemented our algorithm in CUDA and evaluated it in the context of hardware-accelerated ray tracing via OptiX within various scenarios, including distribution ray tracing effects such as motion blur, depth of field, direct lighting with an area light source, and indirect illumination. The results show that our method achieves error reduction by up to 83% within a given time budget that comprises only fractions of a second.

1. Introduction

Realistic image synthesis heavily relies on Monte Carlo integration to simulate various light transport effects [Cook et al. 1984; Kajiya 1986]. A resulting image via Monte Carlo integration is prone to high-frequency noise, which disappears with an increasing number of samples. Hence, to synthesize visually plausible images, it is necessary to use a large number of samples. In real-time or interactive ray tracing, we have a very limited time budget that allows us to trace only a few samples per pixel, which is typically not sufficient. How to simulate light transport correctly and robustly under these limited conditions remains an open question.

Adaptive sampling [Zwicker et al. 2015] is a class of algorithms that concentrates samples at regions of rapid changes, allowing to reconstruct images of the same quality in less time with significantly fewer samples in comparison with standard sampling methods. Adaptive sampling was successfully adapted for offline rendering; however, to our knowledge, the potential of adaptive sampling and reconstruction in the context of real-time or interactive ray tracing is largely unexplored to date.

We speculate that the main reason why adaptive sampling is not often employed for interactive applications is because it is difficult to design an efficient adaptive sampling technique on GPUs. Using adaptive sampling techniques only pays off if the performance gain outweighs the additional overhead due to adaptive sampling and reconstruction. In real-time or interactive ray tracing, the per-sample cost is very low, and thus it is typically challenging to justify the overhead due to adaptive sampling. Adaptive sampling techniques usually keep track of an additional data structure constructed over all samples to efficiently decide where to generate new samples. Due to its adaptive nature, this data structure is incrementally updated as new samples are generated, which is generally problematic to implement on GPUs.

In this paper, we take multidimensional adaptive sampling [Hachisuka et al. 2008] as one of the adaptive sampling techniques successfully applied in offline rendering. The method adaptively samples a higher-dimensional space of Monte Carlo samples using a KD-tree as an underlying data structure for storing samples. The algorithm works iteratively by sampling a single leaf node of the KD-tree at a time, and then employing a priority queue to efficiently find an appropriate leaf node. It significantly

reduces error compared to nonadaptive sampling; however, parallelization of the algorithm is not trivial due to its sequential and adaptive nature.

We revisit and reformulate the multidimensional adaptive sampling method to be highly parallel and thus suitable for GPUs while causing only a small overhead. Our key technical contribution is adaptive sampling of multiple regions in parallel without a priority queue. We borrow some ideas from the efficient construction of acceleration data structures for ray tracing. We apply these ideas in a different context as we build a data structure in a higher-dimensional space of random numbers instead of geometric primitives in 3D. We evaluated the proposed method in the context of hardware-accelerated ray tracing with RTX using OptiX, showing that we can reconstruct images with lower error than the quasi-Monte Carlo integration with Halton sampling given the same time.

2. Related Work

2.1. Adaptive Sampling

The topic of adaptive sampling and reconstruction encompasses many different ideas in rendering, so we mention only the most relevant work. We refer to a survey by Zwicker et al. [2015] for a more comprehensive overview. Mitchell [1987] in his seminal work proposed one of the first adaptive sampling algorithms operating in the image space. This work has led to several follow-up works that perform adaptive sampling in the image space as well [Zwicker et al. 2015].

Different lines of work have explored the idea of performing adaptive sampling in spaces other than the image space. The irradiance or radiance caching techniques [Ward et al. 1988; Křivánek et al. 2005] accelerate rendering of indirect illumination by adaptively sampling points in the object space to cache irradiance or radiance coming onto these points, which are typically stored in a tree-like data structure (e.g., KD-tree) to accelerate spatial queries of the cache. A GPU-friendly modification to irradiance/radiance caching was proposed by several researchers [Gautron et al. 2005; Wang et al. 2009]. They all pointed out the difficulty of maintaining and refining such adaptive data structures on the GPU.

Hachisuka et al. [2008] proposed to adaptively sample the space of input random numbers of the ray tracing process. Unlike irradiance/radiance caching, their method is agnostic to the type of light transport effects. The authors demonstrated applications to motion blur, depth of field, and area light sources. Similarly to irradiance/radiance caching, this method also involves a rather complex process of refinement and construction of a hierarchical data structure. We propose a GPU-friendly modification to this work. Just like the original method by Hachisuka et al. [2008], we demonstrate that our method can support many different types of light transport effects.

More recently, Munkberg et al. [2016] proposed caching and reconstruction in the texture space. Similarly to the method of Hachisuka et al. [2008], this method can accelerate the rendering of many different types of light transport effects. The method assumes the presence of reasonable parameterization of surfaces (i.e., texture space). Our method does not assume the presence of such parameterization.

The reconstruction step in adaptive sampling is closely related to image denoising in the sense that denoising reconstructs a smooth image from a noisy image. Deep learning–based denoisers have became popular in real-time ray tracing as they cope well with noisy estimates at very low sampling rates (e.g., one sample per pixel). Chaitanya et al. [2017] proposed a deep learning–based denoised algorithm based on a recurrent autoencoder for real-time rendering. Kuznetsov et al. [2018] proposed a deep learning–based adaptive sampling and reconstruction, using the same architecture as Chaitanya et al. [2017]. Hasselgren et al. [2020] build on the work of Kuznetsov et al. [2018], replacing initial sampling by warped reprojection to improve temporal stability. Unlike this prior work, we do not employ deep learning.

2.2. Acceleration Data Structures

Our adaptive sampling utilizes recent advances in acceleration data structures for ray tracing. Just like adaptive sampling, acceleration data structures have been extensively studied in the past, so we mention only the most relevant work here. We refer to a recent survey by Meister et al [2021].

In the context of adaptive sampling, we need to build a KD-tree from initial samples, and a KD-tree is often used as an acceleration data structure. Karras [2012] proposed a very fast construction algorithm of bounding volume hierarchies (BVHs) and KD-trees on the GPU based on the sorting of geometric primitives along the Morton curve [Morton 1966]. Our observation is that we do not need to construct and store a tree structure as it is necessary neither for adaptive sampling nor for reconstruction, as we show later. Thus, we generate initial samples on a uniform grid, where the resolution of the grid is given by the number of Morton bits. Vinkler et al. [2017] extended Morton codes by encoding sizes of geometric primitives and using a varying number of bits for different dimensions. In a similar fashion, we use more Morton bits for image dimensions as they need to be sampled more densely to capture important high-frequency details.

Garanzha et al. [2011] proposed a top-down BVH construction algorithm that builds the hierarchy iteratively, level by level, with work queues on the GPU. We refine the tree in the same manner as we adaptively add new samples. Meister and Bittner [2018] proposed a bottom-up BVH construction algorithm on the GPU. The algorithm merges multiple cluster pairs in each iteration. They avoid using the priority queue to determine which cluster pairs to be merged, which is conceptually similar to how we determine which regions to be adaptively sampled.

The reconstruction step, when we reconstruct the resulting image from samples, can be implemented either as gathering or splatting. Gathering is conceptually similar to ray tracing; for each pixel, we traverse the pixel extent to collect all samples contributing to the pixel. Splatting simply projects samples directly onto the corresponding pixels. Though splatting is simpler as it does not require traversing the KD-tree, we need to synchronize accesses to the pixels as multiple threads so that process samples may access the same pixel. Aila and Laine [2009] proposed an efficient stack-based traversal algorithm of bounding volume hierarchies on the GPU with persistent warps and dynamic fetch. Gathering can be efficiently implemented in a similar manner.

3. Multidimensional Adaptive Sampling

We first recapitulate the original multidimensional adaptive sampling [Hachisuka et al. 2008]. Light transport problems can be formulated as definite integrals in the following form:

$$F(x,y) = \int_{\vec{u}\in\mathcal{U}} f(x,y,\vec{u})d\vec{u}, \tag{1}$$

where (x,y) is a location on image $\mathcal{I}$, $\vec{u}$ is a vector of random numbers of integration domain $\mathcal{U}$, $F(x,y)$ is the pixel value at location (x,y), and $f(x,y,\vec{u})$ is the function value of the integrand at location (x,y) using vector $\vec{u}$.

For instance, motion blur can be formulated as a one-dimensional integral over time t where $\vec{u}$ is scalar value u parameterizing time t such that $f(x,y,u)$ returns the result of Monte Carlo ray tracing through the pixel at location (x,y) at time t. One can formulate many interesting light transport effects under this formulation such as direct lighting with an area light source (2D integral over the light source), depth of field (2D integral over lens), and indirect illumination ($2m$D integral over directions for m bounces). Note that we typically integrate across the pixel extent in practice to account for antialiasing that requires two additional dimensions.

The algorithm operates on the idea of *sampling and reconstructing* integrand $f(x,y,\vec{u})$ over the product space $\mathcal{I}\times\mathcal{U}$ of image $\mathcal{I}$ and integral $\mathcal{U}$ spaces. We summarize high-level ideas of the algorithm that works in the following three steps.

3.1. Initial Sampling

The algorithm starts by generating a number of uniform random samples in $\mathcal{I}\times\mathcal{U}$. The algorithm then builds a KD-tree over those initial samples such that each leaf node contains less than a predefined number of samples.

Each leaf node in the KD-tree corresponds to an axis-aligned hyperrectangle in $\mathcal{I}\times\mathcal{U}$ where its first two dimensions define its image extent. We assign a constant

value to each leaf node:

$$f_{N_k}(x, y, \vec{u}) = \begin{cases} \bar{f}_{N_k} & \text{if } (x, y, \vec{u}) \in N_k, \\ 0 & \text{otherwise,} \end{cases} \tag{2}$$

where $\bar{f}_{N_k}$ is the average value of samples in leaf node N_k. Given this partitioning, the KD-tree essentially approximates the integrand $f(x, y, \vec{u})$ as a piecewise constant function:

$$f(x, y, \vec{u}) \approx \sum_{N_k} f_{N_k}(x, y, \vec{u}). \tag{3}$$

3.2. Adaptive Sampling

In the second step, the algorithm estimates the error for each leaf node based on the samples within the leaf node, which roughly measures how the leaf node needs to be refined. In particular, we compute error E_{N_k} of leaf node N_k via the contrast metric [Mitchell 1987; Hachisuka et al. 2008]:

$$E_{N_k} = V_{N_k} \left[\varepsilon + \frac{1}{n_{N_k}} \sum_{(x_i, y_i, \vec{u_i}) \in N_k} \frac{|f(x_i, y_i, \vec{u_i}) - \bar{f}_{N_k}|}{\bar{f}_{N_k}} \right], \tag{4}$$

where V_{N_k} is the hypervolume of leaf node N_k, n_{N_k} is the number of samples in leaf node N_k, $(x_i, y_i, \vec{u_i})$ are coordinates of the ith sample in leaf node N_k, and ε is a small positive constant. Based on these errors, the algorithm then selects a leaf node with the highest error, generates new sample within a bounding sphere centered around the leaf node, and splits the node if the maximum capacity is exceeded. A priority queue is employed to efficiently find the leaf node with the highest error. The addition of ε ensures that the error is nonzero even when all the samples in a leaf happen to have the same value, leaving a possibility of being split when its volume V_{N_k} is large enough.

3.3. Reconstruction

In the last step, the algorithm analytically integrates the piecewise approximation given by Equation (3) to estimate $F(x, y)$ using a Riemann sum over the per-leaf estimates that projects onto image location (x, y):

$$F(x, y) \approx \sum_{\substack{N_k \\ (x,y) \in P_k}} \frac{V_{N_k}}{A_{P_k}} \bar{f}_{N_k}, \tag{5}$$

where P_k is a projection of the hyperrectangle of node N_k onto image $\mathcal{I}$ and A_{P_k} is the area of this projection. To estimate integrals for all the pixels simultaneously, we can either gather values $\bar{f}_{N_k}$ for each pixel or splat values $\bar{f}_{N_k}$ onto the corresponding pixels.

3.4. Antialiasing

The original algorithm evaluates value $F(x, y)$ only at a single point on the image plane using Equation (5). In practice, we want to account for variation in the image plane across the pixel extent:

$$I_{i,j} = \int_{y_j}^{y_{j+1}} \int_{x_i}^{x_{i+1}} F(x, y) dx dy. \tag{6}$$

To estimate $I_{i,j}$, we use the approximation from Equation (5) multiplied by the area of the intersection of projection of the node P_k and pixel extent $P_{i,j}$:

$$I_{i,j} \approx \sum_{N_k} \frac{V_{N_k}}{A_{P_k}} A_{P_k \cap P_{i,j}} \bar{f}_{N_k}, \tag{7}$$

where $P_{i,j} = [x_i, x_{i+1}] \times [y_j, y_{j+1}]$ is the pixel extent and $I_{i,j}$ is a value of the corresponding pixel.

4. Parallel Multidimensional Adaptive Sampling

We explain our parallel version of multidimensional adaptive sampling suitable for GPU ray tracing. The key observation is that the information from the KD-tree is used only as partitioning of the domain at the reconstruction step. We thus do not need to store an entire KD-tree, but can store only the last level consisting of leaf nodes that form the partitioning of the domain. Similar to the original algorithm [Hachisuka et al. 2008], our algorithm consists of three steps: initial sampling, adaptive sampling, and reconstruction.

4.1. Initial Sampling

We start by subdividing the space into a uniform grid, where the resolution of the grid is determined by the number of dimensions d of the space and the number of Morton bits b for each dimension. To sample the image space axes more densely, we use extra bits e for each image axis [Vinkler et al. 2017]. These extra bits correspond to higher-order bits in the Morton code followed by bits of all the dimensions (including image dimensions). We sample each cell uniformly using l samples where l is less than or equal to the maximum number of samples per leaf. Hence, the total number of initial samples is $2^{(db+2e)}l$. Figure 2(a) illustrates this step.

The grid cells are processed in parallel, each by one thread such that its index corresponds to the Morton code. First, we determine the cell coordinates by extracting corresponding bits from the Morton code. Using the cell coordinates and the resolution of the grid, we determine the bounding box of the cell and generate l samples uniformly distributed in this bounding box. The grid cells correspond to the initial

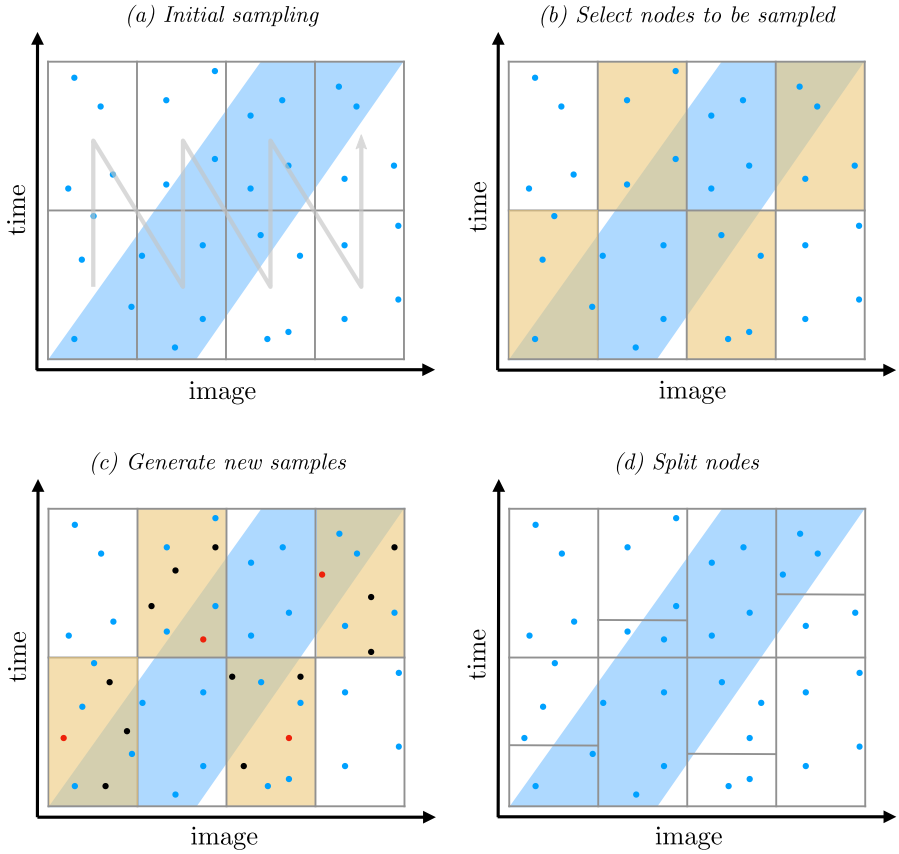

Figure 2. An example of adaptive sampling in 2D with the 1D time axis and 1D image axis (only for illustrative purposes). (a) Initial sampling uses one bit per dimension with one extra bit for the image axis, resulting in a regular grid of 4×2 cells, while each contains four uniformly distributed samples. The Morton curve shows how the cells are processed. (b) The algorithm selects nodes to be sampled prioritizing those with higher error (brown). (c) Several candidates are generated (black), and the best candidate (red) is selected and insert into the node. (d) Nodes are split if the capacity is exceeded. We repeat this process (steps (b) to (d)) until a desirable number of samples is reached.

partitioning of the domain (i.e., leaf nodes) that is further refined in the adaptive sampling step. Note that, as we are interested only in the final partitioning, we do not need to store the tree structure built over the grid cells.

4.2. Adaptive Sampling

Adaptive sampling is done iteratively where multiple nodes are processed in parallel. Each iteration consists of two passes. In the first pass, we compute the error in the same manner as in the original algorithm (see Equation (4)). We also track maximum error E_{max} that we will use in the next pass to determine whether the node will be

sampled or not. The maximum error can be easily computed using parallel reduction.

In the second pass, we perform adaptive sampling itself. We use stochastic sampling to select nodes to be sampled. We allow to sample node N_k only if $(E_{N_k}/E_{\max})^\alpha \geq \xi$, where ξ is a uniformly distributed random number in $[0, 1)$ and α is a nonnegative number, one of the parameters of the algorithm that controls the number of nodes to be sampled. If α is close to zero, almost all nodes will be sampled, which is similar to uniform sampling. With increasing α, the number of sampled nodes decreases, focusing on those with higher error values. Notice that the node with the maximum error always passes this test regardless of α to make sure that at least one node is sampled per iteration. As the error is always positive (see Equation (4)), each node has a nonzero probability to be sampled.

If the node passes this test, we use the best candidate method to determine a new sample; the best candidate method maximizes the Euclidean distance to the closest sample point considering only samples within the bounding box. After adding a sample to the node, if the maximum capacity is exceeded, the node is split along its maximum extent using the object median split (i.e., samples are split into two halves of roughly the same size) [Meister et al. 2021]. This step performs two parallel prefix scans: one to determine storage locations of new samples and one to determine storage locations of new nodes. The left child then overwrites the current node, and the corresponding prefix scan value determines the storage location of the right child. This process is repeated until a predefined number of samples is reached (see Figure 2).

4.3. Reconstruction

The reconstruction algorithm uses splatting to process all the nodes in parallel. A projected bounding box of each node onto the image plane determines the pixels for splatting. For each pixel, the pixel extent is projected to the domain, resulting in a bounding box that spans across all other dimensions. The intersection of the pixel bounding box and the node bounding box results in another bounding box. The algorithm multiplies the average value of the samples in the node by the hypervolume of this bounding box, which corresponds to one term of the sum in Equation (7). This value is atomically added to the corresponding pixel, using three atomic additions (one for each color channel). Note that we also tried gathering, but it was slower than splatting, requiring more memory as we need to store the tree structure to collect the corresponding samples for each pixel.

4.4. GPU Implementation

Our algorithm can be easily implemented via general-purpose GPU computing APIs such as CUDA or OpenCL. The whole algorithm may consist of four kernels: initial sampling, error computation, adaptive sampling, and reconstruction. Each kernel

processes nodes and samples. Two buffers store sample coordinates (d floats) and values (3 floats), and four buffers stores sample indices (l ints), bounding boxes ($2d$ floats), seeds (1 int), and error values (1 float). The error computation kernel retains the errors from the previous iterations and recomputes them only for new nodes. We have found that this optimization saves a significant portion of the computation time. A warp-wide prefix scan with one atomic addition per warp determines the storage locations of new nodes and samples, which is more efficient than a full prefix scan. We compute the maximum error across the nodes in a similar manner using the atomic maximum operation.

5. Results

We implemented our algorithm in CUDA 11.5 and OptiX 7.2. All experiments were executed on a PC equipped with the RTX 3080 Ti GPU. We evaluated the proposed method in the context of hardware-accelerated ray tracing with RTX simulating various stochastic effects, including motion blur, depth of field, direct lighting with an area light, and indirect illumination. Our implementation builds on the OptiX SDK samples, where each effect corresponds to one sample project. Our modification to each sample is limited to simply supporting the proposed method without any low-level optimization specific to each. The reference images are rendered by quasi-Monte Carlo integration with a Halton sampler with random toroidal shift. The error metric is relative mean squared error (RelMSE), where each squared difference is further divided by the sum of the squared reference value and a small value (10^{-2}) to better emphasize visual differences. No subsampling of light sources is done.

5.1. Parameter Settings

As the original method, we can configure the number of candidates in the best candidate sampling and the maximum number of samples per node. We can also set the scaling factor of the image dimensions, which causes the image dimensions to be split more frequently as we use the maximum extent heuristic for splitting the node, and thus the image extent will be sampled more densely. The scaling factor is a fraction of the image resolution. For example, using scaling factor 1/16 and image resolution 1024×768, the image axes are scaled by $1024/16 = 64$ and $768/16 = 48$.

Unlike the original algorithm, the number of initial samples is given by the number of Morton bits, extra bits for image dimensions, and the number of initial samples per cell (Section 4.1). The number of extra bits emulates the scaling factor for initial samples (i.e., more bits result in a grid with a finer resolution for image dimensions). Note that the extra bits and scaling factors are used exclusively to each other; extra bits are used only for the initial sampling, while the scaling factor influences only the adaptive sampling.

One can set an arbitrary value for the average number of samples per pixel in our

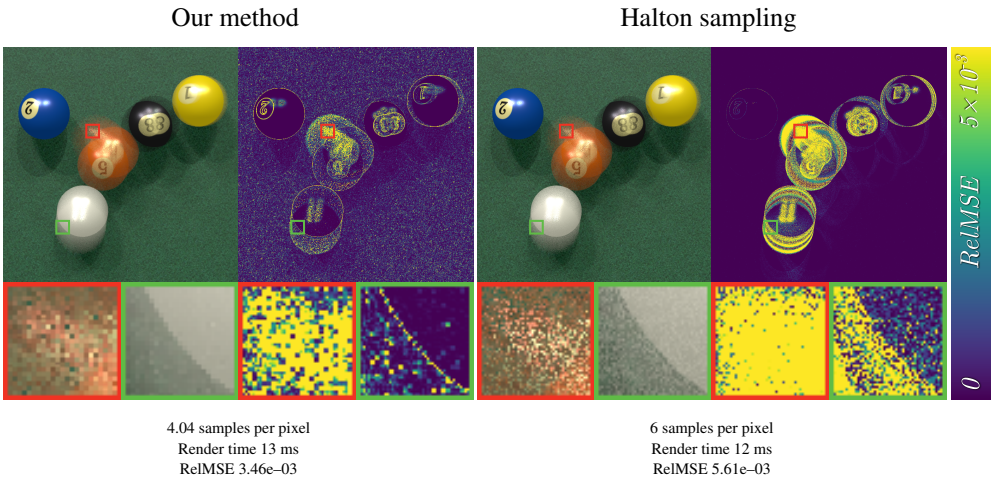

Figure 3. Comparison of our method and Halton sampling for the Pool scene with motion blur. Our method produces results with less noise, especially in regions with a significant motion blur effect.

algorithm. The product of this average number and the number of pixels is roughly equal to the total number of samples. In practice, the actual total number of samples can be slightly larger as each iteration generates multiple samples in parallel. The parameter α controls the number of samples generated in each iteration (Section 4.2).

In all the test cases, we used four candidates in the best candidate sampling and the maximum of four samples per node and four initial samples per cell. We tuned the other parameters per individual effects and test scenes by exhaustively searching over many different configurations. Table 1 summarizes the parameters.

5.2. Motion Blur

Motion blur uses one extra dimension for the time beside two dimensions for the image domain. Figure 3 shows an equal time comparison of our method and Halton sampling for the Pool scene rendered in image resolution 1024×1024 with nine directional lights. Since OptiX does not support nonuniform time intervals for motion blur, we resampled the input transformation sequence into 11 uniform samples to emulate nonuniform time intervals. Our method concentrates samples into regions with strong motion, providing results with less high-frequency noise while using fewer samples than Halton sampling (39% lower RelMSE).

5.3. Depth of Field

Depth of field uses two extra dimensions to sample an offset for ray origin in order to achieve a defocus blur. Figure 1 and Figure 4 show equal time comparisons for the Chess and Bistro scenes, respectively. The Chess scene was rendered in image

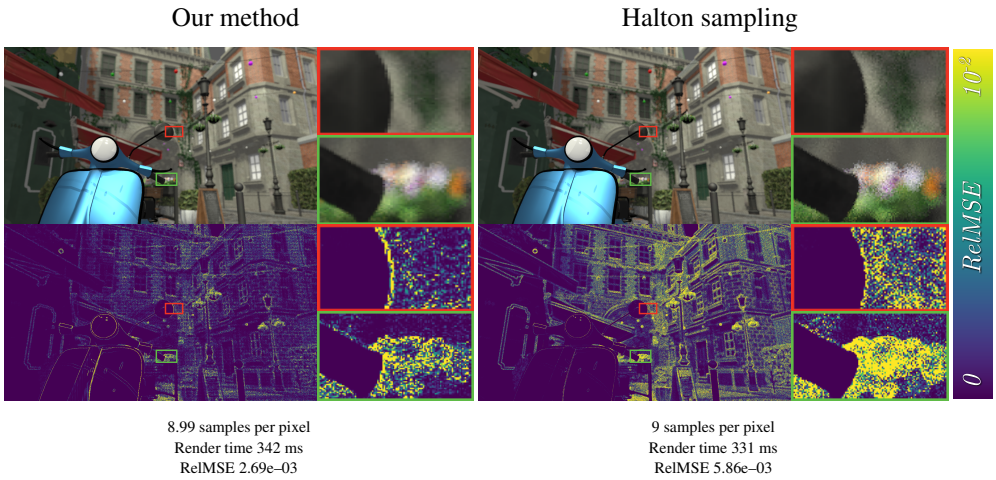

Figure 4. Comparison of our method and Halton sampling for the Bistro scene with depth of field. Our method provides images with lower error and less noise, particularly in the background that exhibits a strong depth-of-field effect. Notice that our method uses even more samples than Halton sampling thanks to our efficient initial sampling phase.

resolution 768×1024 with four point lights. The Bistro scene was rendered in image resolution 1920×1080 with 31 directional lights. The Bistro scene is a very complex test case (e.g., resolution, geometry, occlusion) requiring significantly more time to be rendered for both methods. The ray tracing and shading parts dominate the computation time, while the adaptive sampling part takes only a marginal part of the total time (see Table 1). Our method indeed can use slightly more samples than Halton sampling thanks to our more efficient initial sampling. In both cases, we achieved smoother results as our method places samples into the out-of-focus regions (12% and 54% lower RelMSE, respectively).

5.4. Direct Lighting

Direct lighting with an area light source uses two extra dimensions for sampling a point on the light source. Figure 5 shows the equal time comparison for the Dragon and Cobblestone scenes rendered in image resolution 1024×768 and 1920×1080, respectively, using one area light in both cases. In the case of the Dragon scene, we were not able to achieve lower error than Halton sampling (52% higher RelMSE). The reason is that Halton sampling converges very quickly and the sample cost is relatively low; we evaluate only one light sample on the area light source per sample. In this case, the overhead of adaptive sampling outweighs its improvement. The Cobblestone scene contains finely-tessellated geometry with a high-frequency texture. In this case, out method achieves slightly better results than Halton sampling (3% lower RelMSE).

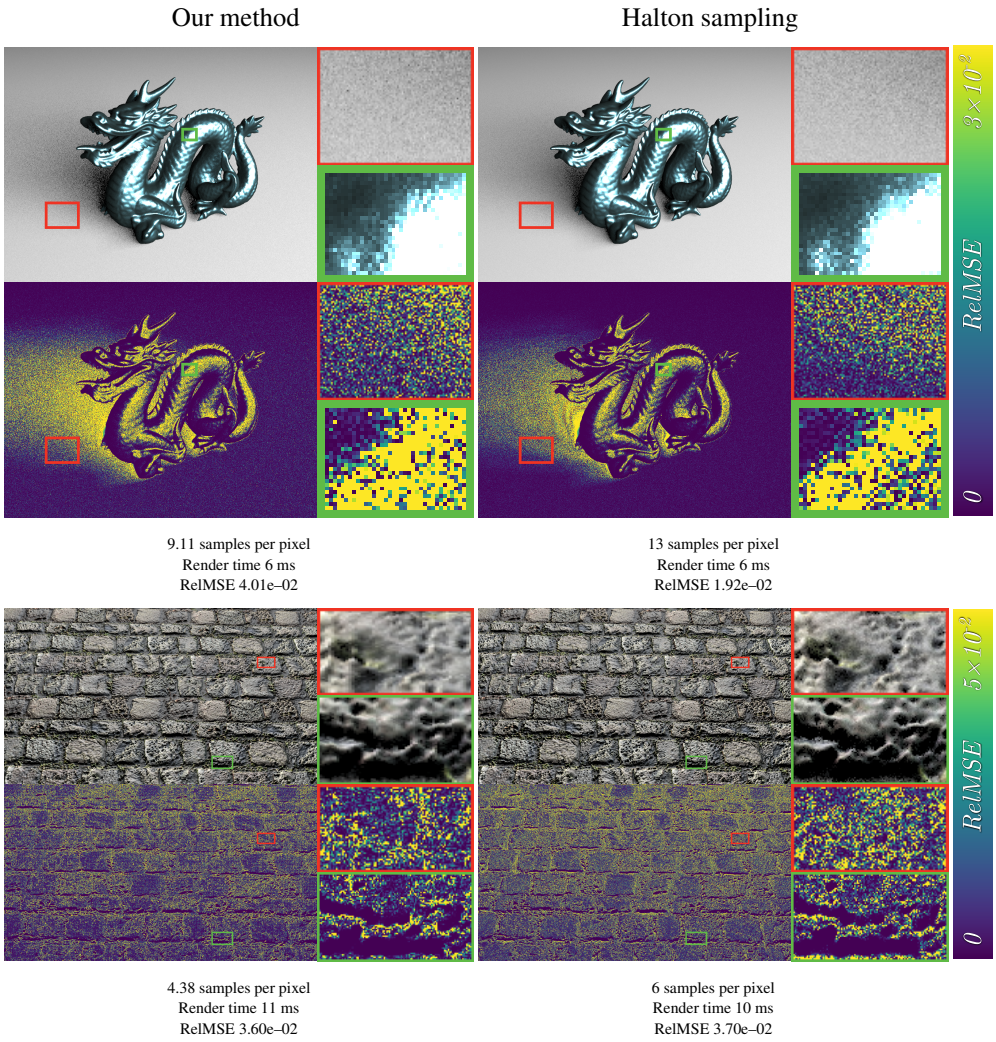

Figure 5. Comparison of our method and Halton sampling for the Dragon and Cobblestone scenes with direct lighting by an area light. The Dragon scene is rather a failure case. The reason is that overhead caused by our method outweighs the performance gain of adaptive sampling.

5.5. Path Tracing

We used path tracing with the next event estimation with one point light source. Because we use a point light source, we need only two extra dimensions per bounce to sample outgoing directions. Figure 6 shows the equal time comparison for the Cornell Box and Breakfast scenes. The Cornell Box scene was rendered in image resolution 1024×1024 with two indirect bounces. The Breakfast scene was rendered in resolution 1024×768 with one indirect bounce. In both cases, our method is able to use even slightly more samples than Halton sampling, similarly to the Bistro scene. As

	Pool	Chess	Bistro	Dragon	Cobble.	C. Box	Breakfast
Effect	MB	DOF	DOF	DL	DL	PT	PT
Dimensions	3D	4D	4D	4D	4D	6D	4D
Scaling factor	1/16	1	1/16	1/4	1/16	1	1/2
Morton bits per dimension	1	1	0	0	0	1	1
Extra image bits per dimension	8	8	10	10	10	7	8
Parameter α	1/4	1/4	1/32	1/64	1/64	1/16	1/32
Triangles	57k	50k	3858k	871k	10323k	36	808k
Lights	9	4	31	1A	1A	1	1
Iterations	6	6	6	2	3	3	2
Initial samples	2M	4.1M	4.1M	4.1M	4.1M	4.1M	4.1M
Total samples	4.2M	6.4M	18.6M	7.1M	9.1M	8.5M	7.2M
Initial sampling time (ms)	0.22	0.64	0.56	0.64	0.61	1.46	0.64
Error computation time (ms)	0.68	0.94	2.10	0.58	0.65	0.70	0.59
Adaptive sampling time (ms)	2.22	2.92	8.97	1.29	2.52	4.21	1.27
Reconstruction time (ms)	0.84	1.25	4.73	0.40	1.53	1.80	0.59
Total sampling time (ms)	3.96	5.76	16.35	2.91	5.30	8.18	3.09
Trace time (ms)	9.30	4.41	325.24	3.08	5.22	13.02	20.11
Total time (ms)	13.26	10.17	341.59	5.99	10.52	21.19	23.21

Table 1. Configuration of our test cases (top) and breakdown of times for different phases of our method (bottom). The effects are motion blur (MB), depth of field (DOF), direct lighting with an area light source (DL), and path tracing with indirect illumination (PT).

the effect of indirect illumination is relatively smooth, our method can easily handle it. As the indirect illumination practically influences the whole image, we achieved significant error reduction (83% and 48% lower RelMSE, respectively). Indirect lighting is a good example where we can observe the influence of the scaling factor and the number of extra bits used for image dimensions (see Figure 7). Though a lower scaling factor and fewer extra bits lead to lower overall error, some high-frequency details might be blurred. On the other hand, a higher scaling factor and more extra bits allow sampling of image dimensions more densely, preserving the high-frequency details at the cost of higher overall error. A rule of thumb is to set the scaling factor and the extra bits as small as possible while preserving high-frequency details.

5.6. Computation Times

Table 1 shows the breakdown of times needed for different phases of our algorithm. The initial sampling phase takes only a small part of the total time. Its computation time depends on the number of initial samples (see Section 4.1); in all cases, it takes about one millisecond. One important consequence of the initial sampling phase is that we can evaluate more samples than the number of pixels, which is not the case for the standard wavefront rendering algorithms. Processing more samples in parallel generally leads to better utilization of computational resources. Furthermore, the initial samples are generally well distributed as they are nonuniformly distributed across dimensions and can be easily stratified. This leads to lower times and lower error, allowing us to trace even more samples than the standard wavefront algorithm (see

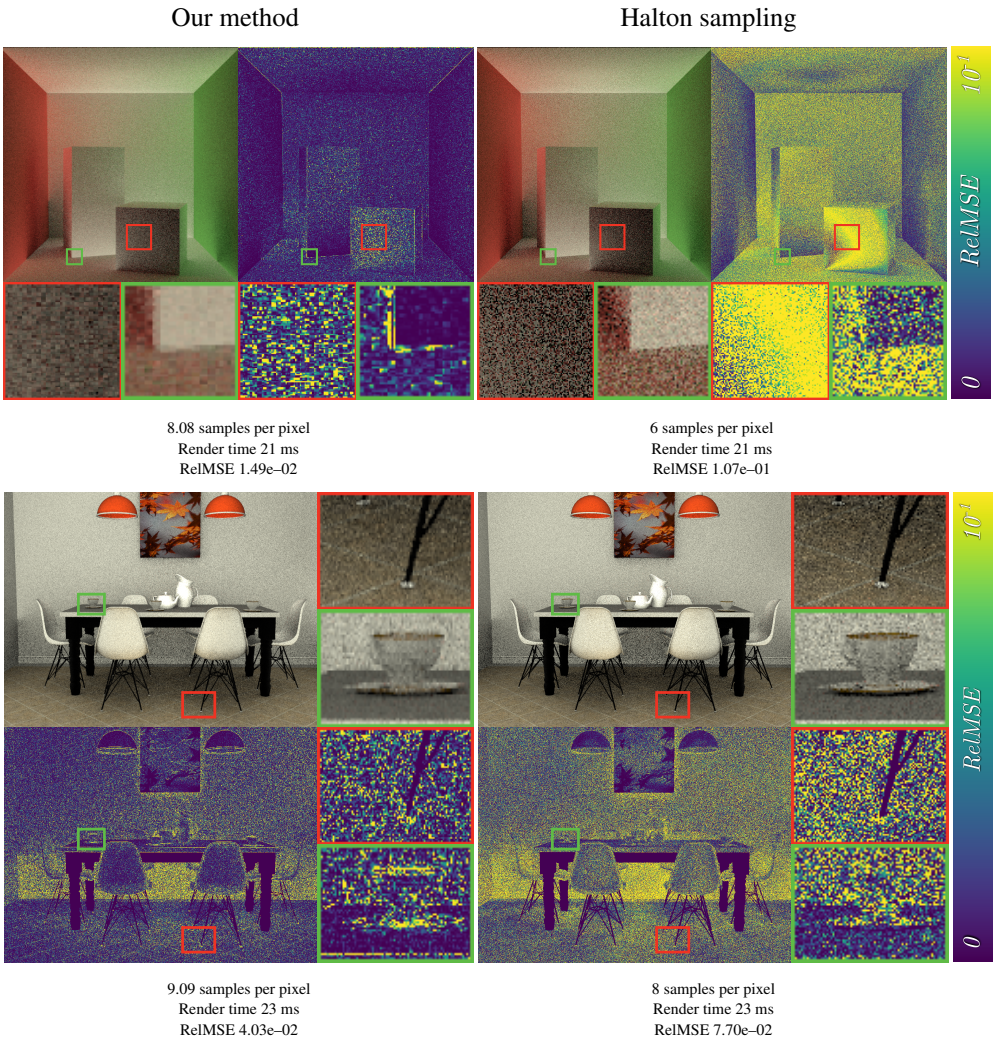

Figure 6. Comparison of our method and Halton sampling for the Cornell Box and Breakfast scenes with indirect illumination. Because the indirect illumination influences the whole image, our method significantly reduces error practically in all regions. In both cases, our method achieves a higher sample count than Halton sampling thanks to better utilization of parallel resources.

Figures 4 and 6). The most time-consuming part is adaptive sampling. Though each iteration is relatively fast, as adaptive sampling is done in multiple iterations, it takes a significant portion of the total time. In typical scenarios, we can afford only a few iterations, which is usually sufficient to achieve good results. The number of iterations needed does not depend only on the parameter α but also on other parameters such as the number of initial samples and the number of desired samples per pixel. The reconstruction phase does not have any parameters, and its time complexity is given by the number of pixels and the total number of samples. The total time needed for

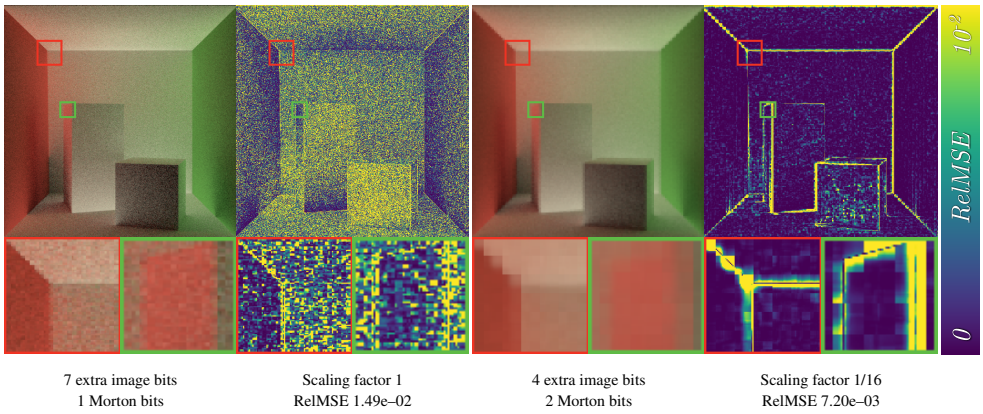

Figure 7. Influence of the number of Morton bits, extra bits for image dimensions, and scale factor on the trade-off between high-frequency details with higher RelMSE (left) versus low-frequency blur with lower RelMSE (right).

adaptive sampling is typically lower than the ray tracing and shading times.

6. Discussion and Limitations

We discuss the limitations of the proposed method and compare it to the original algorithm [Hachisuka et al. 2008].

6.1. Limitations

Our parallel algorithm shares limitations with the original sequential algorithm. Theoretically, multidimensional adaptive sampling can be used for an arbitrary number of dimensions. In practice, it is only applicable for low dimensions (e.g., 3D to 6D) partially due to the efficiency of the KD-tree. Another problem is that different dimensions may require different number of samples (e.g., image space versus time). We account for this fact by scaling the image axes relative to the others. This simple technique is less ideal. For example, some dimensions may be part of a lower-dimensional manifold embedded in the higher-dimensional space (due to hitting a background or using techniques such as Russian roulette). In this case, uniformly scaling each axis is likely suboptimal.

Compared to simple Monte Carlo rendering, our method needs to store additional data in memory. This storage requirement can severely limit the maximum number of samples on GPUs. The storage cost for one sample in our implementation is 72–108 bytes (depending on the number of dimensions), including KD-tree nodes, bounding boxes, sample indices, sample coordinates, sample values, errors, and seeds. Similarly to the original algorithm, we could circumvent this issue by splitting the image

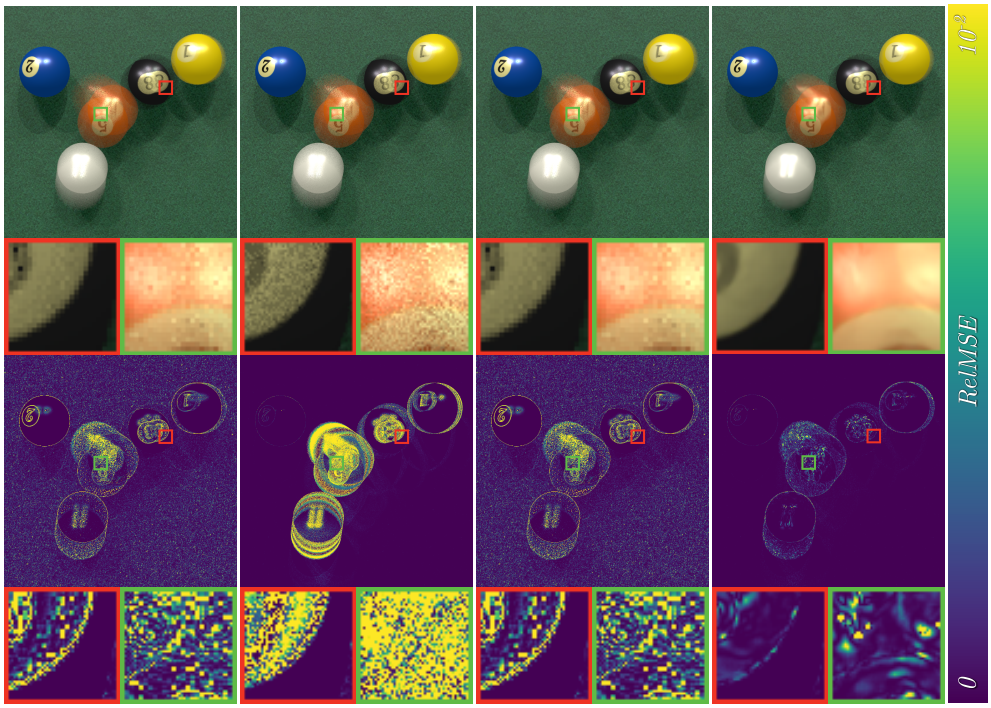

Figure 8. Comparison of results without denoising (left) and with denoising (right). Traditional denoisers excel in improving uncorrelated estimates provided by Halton sampling; however, they have no influence on the results provided by our method.

into multiple tiles and process them sequentially; however, this might decrease the performance significantly not only by the sequential processing but also due to the overhead of launching multiple GPU processes.

Our method is not always better than Halton sampling. The cost of taking one raw sample must be high enough to justify the additional overhead in our method. An example of such a failure case is the Dragon scene with direct lighting (Figure 5). Besides direct lighting with an area light source, we also tested ambient occlusion in this scene. Both effects are fairly inexpensive to compute than other effects and typically converge after a small number of samples, making our method less effective.

Denoising works very well with uncorrelated estimates provided by Monte Carlo or quasi-Monte Carlo integration. We conducted an experiment with the OptiX denoiser [Chaitanya et al. 2017] to see how it copes with the output of our method (see Figure 8). We can see that the denoiser has practically no effect on correlated estimates in our method as the denoiser is trained for the uncorrelated estimates. A different method [Back et al. 2020] that supports correlated estimates would be necessary for our method.

6.2. Differences from the Algorithm by Hachisuka et al.

The original algorithm [Hachisuka et al. 2008] was designed in the context of offline rendering, where rendering times range from hundreds to thousands of seconds. In such a case, the sample cost is relatively high, and thus it pays off to use fewer high-quality adaptive samples instead of more uniform samples. In our case, we compete with hardware-accelerated ray tracing, where the per-sample cost is very low, taking only a few milliseconds. In other words, we cannot afford to spend too much time on adaptive sampling; otherwise, its cost outweighs the benefits. We were thus forced to implement some simplifications and changes in the design of our algorithm to increase efficiency at the cost of slight degradation in quality.

In the initial sampling phase, the original algorithm constructs a KD-tree from the initial samples. In our case, our method generates samples directly on the uniform grid such that the grid cells correspond to the leaf nodes. We do not store the interior nodes and keep only the leaf nodes, and thus there is no need to construct this initial tree. The results with initial samples in our algorithm are already better than the results with uniform and independent samples in the original algorithm, because of stratification and extra bits used for the image axes in our initial samples. Another positive side effect is that the rays induced by our initial samples are more coherent [Meister et al. 2020]. Moreover, the number of initial rays is not limited to a multiple of the image resolution, which helps to utilize computational resources better.

In the adaptive sampling phase, the original algorithm employs a priority queue to determine the leaf node with the maximum error. Using the priority queue is not suitable for parallel processing due to its sequential nature. Our algorithm does not use the priority queue and computes the maximum error using parallel reduction in each iteration that is subsequently used to determine nodes to be sampled

The original algorithm samples a slightly larger extent than just a bounding box of the leaf node. Hachisuka et al. proposed to sample as a bounding sphere to spill samples into neighboring nodes, which increases the chance to capture important features. This approach is not suitable for parallel processing as multiple samples might be inserted into the same leaf node in parallel. Though it is possible to utilize atomic locks to prevent such conflicts, we have found that it causes significant overhead.

A possible solution would be to "spill" the error into the neighboring nodes to avoid using atomic locks. This process can be done efficiently as a gather operation similarly to filtering. This approach, however, leads to less accurate sampling of important features as samples are also placed into regions that would not be sampled otherwise. A practical implementation of this approach would need to perform a range search to find the neighboring nodes, which requires the tree structure and causes additional overhead.

Instead, we decided to use the simplest solution, which is to sample only the extents of the bounding boxes. For the best candidate method used in the original

algorithm, we consider only samples in the leaf node to further speed up the sampling process.

In the reconstruction phase, our algorithm does not account for the anisotropic behavior of the function as in the original algorithm. Nonetheless, this part of the original algorithm requires expensive estimation of gradients, and thus it is not suitable for real-time or interactive rendering. We also simplified the evaluation of the function value in a leaf node using the average value of samples in the leaf node instead of performing an additional nearest-neighbor search as in the original algorithm. Our method integrates over the pixel area compared to the original algorithm that evaluates the function value in the center of a pixel (see Section 3).

The original algorithm may miss some important features due to insufficient initial samples. For instance, if all samples in a leaf node have the same value, then the error is close to zero (cannot be zero due to ε), even the function within the cell may not be constant. These features will also be missed in later stages as the algorithm focuses only on regions with high error. Our method uses a stochastic approach such that every node has a nonzero probability of being sampled further. Thus, the error progressively disappears with an increasing number of adaptive samples, unlike the original algorithm.

7. Conclusion

We successfully adapted multidimensional adaptive sampling in the context of interactive and real-time ray tracing. We reformulated the original algorithm [Hachisuka et al. 2008] to be suitable for modern GPU architectures. We exploit the fact that we do not need to store the whole KD-tree, which allows us to simplify the algorithm significantly. In particular, we generate initial samples within the cells of the uniform grid that are subsequently used as an initial partitioning of the domain. This approach is simple yet efficient, and also the initial samples are well distributed. We proposed a simplified reconstruction algorithm based on splatting samples onto the image plane, relying on efficient implementation of atomic operations in modern graphics hardware. We applied the proposed method to various stochastic effects, including motion blur, depth of field, ambient occlusion, and indirect illumination. We evaluated our method in the context of hardware-accelerated ray tracing via OptiX, achieving significant error reduction (up to 83%) in comparison with Halton sampling within a limited time budget comprising only fractions of a second.

Acknowledgements

We would like to thank Jakub Bokšanský, Yuji Moroto, Aaryaman Vasishta, and Rex West for the valuable feedback. We would like to also thank Morgan McGuire for providing the test scenes [McGuire 2017], the authors of the original multidimensional

adaptive sampling algorithm for providing the Chess and Pool scenes [Hachisuka et al. 2008], and Blend Swap (blendswap.com/blend/18005) for providing the Cobblestone scene. This work was supported by the Japan Society for the Promotion of Science (Postdoctoral Fellowship for Research in Japan, ID P19759).

Index of Supplemental Materials

The source codes of the algorithm can be downloaded from Github: `https://github.com/meistdan/pmdas`.

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the Conference on High-Performance Graphics 2009*, Association for Computing Machinery, New York, 145–149. URL: https://doi.org/10.1145/1572769.1572792. 47

BACK, J., HUA, B.-S., HACHISUKA, T., AND MOON, B. 2020. Deep combiner for independent and correlated pixel estimates. *ACM Transactions on Graphics 39*, 6, 242:1–242:12. 59

CHAITANYA, C. R. A., KAPLANYAN, A. S., SCHIED, C., SALVI, M., LEFOHN, A., NOWROUZEZAHRAI, D., AND AILA, T. 2017. Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics 36*, 4, 98:1–98:12. URL: https://doi.org/10.1145/3072959.3073601. 46, 59

COOK, R. L., PORTER, T., AND CARPENTER, L. 1984. Distributed ray tracing. *ACM SIGGRAPH Computer Graphics 18*, 3, 137–145. URL: https://doi.org/10.1145/964965.808590. 44

GARANZHA, K., PANTALEONI, J., AND MCALLISTER, D. 2011. Simpler and faster HLBVH with work queues. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, Association for Computing Machinery, New York, 59–64. URL: https://doi.org/10.1145/2018323.2018333. 46

GAUTRON, P., KŘIVÁNEK, J., BOUATOUCH, K., AND PATTANAIK, S. 2005. Radiance cache splatting: A GPU-friendly global illumination algorithm. In *ACM SIGGRAPH 2005 Classes*. Association for Computing Machinery, New York, 78:1–78:10. URL: https://doi.org/10.1145/1401132.1401231. 45

HACHISUKA, T., JAROSZ, W., WEISTROFFER, R. P., DALE, K., HUMPHREYS, G., ZWICKER, M., AND JENSEN, H. W. 2008. Multidimensional adaptive sampling and reconstruction for ray tracing. *ACM Transactions on Graphics 27*, 3, 1–10. URL: https://doi.org/10.1145/1360612.1360632. 43, 44, 45, 46, 47, 48, 49, 58, 60, 61, 62

HASSELGREN, J., MUNKBERG, J., SALVI, M., PATNEY, A., AND LEFOHN, A. 2020. Neural temporal adaptive sampling and denoising. *Computer Graphics Forum 39*, 2, 147–155. URL: https://diglib.eg.org/handle/10.1111/cgf13919. 46

KAJIYA, J. T. 1986. The rendering equation. *ACM SIGGRAPH Computer Graphics 20*, 4, 143–150. URL: https://doi.org/10.1145/15886.15902. 44

KARRAS, T. 2012. Maximizing parallelism in the construction of BVHs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, Eurographics Association, Goslar, 33–37. URL: https://research.nvidia.com/publication/maximizing-parallelism-construction-bvhs-octrees-and-k-d-trees. 46

KUZNETSOV, A., KALANTARI, N. K., AND RAMAMOORTHI, R. 2018. Deep adaptive sampling for low sample count rendering. *Computer Graphics Forum 37*, 4, 35–44. URL: https://diglib.eg.org/handle/10.1111/cgf13473. 46

KŘIVÁNEK, J., GAUTRON, P., PATTANAIK, S., AND BOUATOUCH, K. 2005. Radiance caching for efficient global illumination computation. *IEEE Transactions on Visualization and Computer Graphics 11*, 5, 550–561. URL: https://doi.org/10.1109/TVCG.2005.83. 45

MCGUIRE, M., 2017. Computer Graphics Archive, July. URL: https://casual-effects.com/data. 61

MEISTER, D., AND BITTNER, J. 2018. Parallel locally-ordered clustering for bounding volume hierarchy construction. *IEEE Transactions on Visualization and Computer Graphics 24*, 3, 1345–1353. URL: https://doi.org/10.1109/TVCG.2017.2669983. 46

MEISTER, D., BOKSANSKY, J., GUTHE, M., AND BITTNER, J. 2020. On ray reordering techniques for faster GPU ray tracing. In *Proceedings of Symposium on Interactive 3D Graphics and Games*, Association for Computing Machinery, New York, 13:1–13:9. URL: https://doi.org/10.1145/3384382.3384534. 60

MEISTER, D., OGAKI, S., BENTHIN, C., DOYLE, M. J., GUTHE, M., AND BITTNER, J. 2021. A survey on bounding volume hierarchies for ray tracing. *Computer Graphics Forum (Proceedings of Eurographics) 40*, 2, 683–712. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.142662. 46, 51

MITCHELL, D. P. 1987. Generating antialiased images at low sampling densities. *ACM SIGGRAPH Computer Graphics 21*, 4, 65–72. URL: https://doi.org/10.1145/37401.37410. 45, 48

MORTON, G. 1966. A computer oriented geodetic database and a new technique in file sequencing. Tech. rep., IBM Germany Scientific Symposium Series. URL: https://dominoweb.draco.res.ibm.com/0dabf9473b9c86d48525779800566a39.html. 46

MUNKBERG, J., HASSELGREN, J., CLARBERG, P., ANDERSSON, M., AND AKENINE-MÖLLER, T. 2016. Texture space caching and reconstruction for ray tracing. *ACM Transactions on Graphics 35*, 6, 249:1–249:13. URL: https://doi.org/10.1145/2980179.2982407. 46

VINKLER, M., BITTNER, J., AND HAVRAN, V. 2017. Extended morton codes for high performance bounding volume hierarchy construction. In *Proceedings of High-Performance Graphics*, Association for Computing Machinery, New York, 9:1–9:8. URL: https://doi.org/10.1145/3105762.3105782. 46, 49

WANG, R., WANG, R., ZHOU, K., PAN, M., AND BAO, H. 2009. An efficient GPU-based approach for interactive global illumination. *ACM Transactions on Graphics 28*, 3, 91:1–91:8. URL: https://doi.org/10.1145/1576246.1531397. 45

WARD, G. J., RUBINSTEIN, F. M., AND CLEAR, R. D. 1988. A ray tracing solution for diffuse interreflection. *ACM SIGGRAPH Computer Graphics 22*, 4, 85–92. URL: https://doi.org/10.1145/378456.378490. 45

ZWICKER, M., JAROSZ, W., LEHTINEN, J., MOON, B., RAMAMOORTHI, R., ROUSSELLE, F., SEN, P., SOLER, C., AND YOON, S.-E. 2015. Recent advances in adaptive sampling and reconstruction for Monte Carlo rendering. *Computer Graphics Forum (Proceedings of Eurographics) 34*, 2, 667–681. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12592. 44, 45

Author Contact Information

Daniel Meister
The University of Tokyo
Yayoi, Bunkyo, Tokyo
Japan
meistdan@gmail.com
https://meistdan.github.io

Toshiya Hachisuka
University of Waterloo
Waterloo, Ontario
Canada
toshiya.hachisuka@uwaterloo.ca
https://cs.uwaterloo.ca/~thachisu

Daniel Meister and Toshiya Hachisuka, Lightweight Multidimensional Adaptive Sampling for GPU Ray Tracing, *Journal of Computer Graphics Techniques (JCGT)*, vol. 11, no. 3, 43–64, 2022
http://jcgt.org/published/0011/03/03/

Received:	2021-06-29		
Recommended:	2022-02-21	Corresponding Editor:	Marc Stamminger
Published:	2022-08-15	Editor-in-Chief:	Marc Olano