Implementation of Random Parameter Filtering

PRADEEP SEN and SOHEIL DARABI
UNM Advanced Graphics Lab

Monte Carlo (MC) rendering systems can produce spectacular images but are plagued with noise at low sampling rates. In a recent paper [Sen and Darabi 2011b], we observed that this noise occurs in regions of the image where the sample values are a direct function of the random parameters used in the Monte Carlo system. Therefore, we proposed a way to identify MC noise by estimating this functional relationship from a small number of input samples. To do this, we treat the rendering system as a black box and calculate the statistical dependency between the outputs and inputs of the system. We then use this information to reduce the importance of the sample values affected by MC noise when applying an image-space, cross-bilateral filter, which removes only the noise caused by the random parameters but preserves important scene detail. The process of using the functional relationships between sample values and the random parameter inputs to filter MC noise is called *random parameter filtering* (RPF), and we demonstrate that it can produce images in a few minutes that are comparable to those rendered with a thousand times more samples. Furthermore, our algorithm is general because we do not assign any physical meaning to the random parameters, so it works for a wide range of Monte Carlo effects, including depth of field, area light sources, motion blur, and path-tracing. In this technical report, we present the complete set of implementation details necessary to reproduce the results of our paper, and we show some additional results produced by our technique.

Categories and Subject Descriptors: I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism—*Raytracing*

General Terms: Rendering

Additional Key Words and Phrases: Monte Carlo rendering, global illumination

1. INTRODUCTION

Monte Carlo (MC) rendering systems can produce beautiful, photo-realistic images by simulating light transport through a series of multidimensional integrals at every pixel of the image: integration of the radiance over the aperture of the camera, over the area light sources of the scene, over the time the shutter is open, and even over the pixel for antialiasing. For a pixel in the image $I(i, j)$, this process can be written as:

$$I(i,j) = \int_{i-\frac{1}{2}}^{i+\frac{1}{2}} \int_{j-\frac{1}{2}}^{j+\frac{1}{2}} \cdots \int_{-1}^{1} \int_{-1}^{1} \int_{t_0}^{t_1} f(x, y, \cdots, u, v, t)\, dt\, dv\, du\, \cdots\, dy\, dx.$$

MC renderers estimate these integrals by taking many point samples of the scene function $f()$, a black-box, functional representation of the ray-tracing system given a specific scene. This sampling process involves tracing rays with sets of random parameters that correspond to the dimensions of integration, e.g., t (the moment in time of the ray for motion blur), u and v (the position of the ray on the aperture of the camera for depth of field) and so on. In path-tracing [Kajiya 1986], the Monte Carlo system integrates over randomly selected paths from the camera's image plane through the scene in order to compute full global illumination effects.

If we evaluate the scene function at enough of these multi-dimensional samples, the MC rendering system will converge to the actual value of the integral, resulting in a physically correct image. Unfortunately, the variance of the estimate of the integral decreases as $O(1/N)$ with the number of samples, so a large number of samples are needed to get a noise-free result. This means that although we can get a very noisy approximation of the final image in a few minutes (as shown in the top image of Fig. 1), we usually need a long time (as much as a day per image) to get a result that is acceptable for high-end rendering applications, shown in the bottom image of Fig. 1. This limits the use of Monte Carlo rendering systems in modern digital film production.

An obvious way to try to address this problem is to apply a noise-reduction filter to the noisy image. Indeed, this approach has been explored by researchers in the past (e.g., [Lee and Redner 1990; Rushmeier and Ward 1994; Jensen and Christensen 1995; McCool 1999; Xu and Pattanaik 2005; Dammertz et al. 2010]), but with limited success. The fundamental problem is that filters cannot easily determine what is unwanted noise (introduced by the MC integration process) and what is valid scene content, since scene content can often have a noisy appearance.

In a recent paper [Sen and Darabi 2011b], we introduced a new approach called random parameter filtering (RPF), a simple, post-process technique based on a bilateral filter [Tomasi and Manduchi 1998] that works in image space after the samples have been computed and that is easy to integrate with a conventional MC rendering system. Our algorithm is able to identify Monte Carlo noise and separate it from scene-dependent noise using a simple observation: the undesired MC noise occurs whenever the sample values are a function of the random parameters used in the Monte Carlo system.

For example, in a scene with an area light source that is point-sampled by the MC system to compute soft shadows, we see that in fully dark areas (the umbra), the final shaded color of the samples is not a function of the random position of the sample on the light source because no matter where the sample is located the shadow ray is always blocked. These regions are not noisy because the random parameters do not affect the output. A similar thing happens in fully lit regions, where the shadow ray is able to reach the light source regardless of its position on the light source. In both of these regions, the scene function $f()$ is constant with respect to the random point on the light source and so its position does not affect the output of the function.

In the penumbra regions, however, some of the shadow rays will reach the light source while others will be blocked by occluders, depending on where we position the sample on the light source. This means that the color of the sample in these parts of the image will be a function of the position of the sample on the light source, which is why these regions contain undesired Monte Carlo noise. This same observation holds true for other Monte Carlo effects, such as depth of field, motion blur, path-tracing, and Russian roulette, as we show in our paper [Sen and Darabi 2011b].

Our key insight (and the basis of the random parameter filtering algorithm) is that if we estimate these functional relationships between the inputs and the outputs of the rendering system, we can reduce the importance of sample features that depend on the random parameters when applying a cross-bilateral filter [Eisemann and Durand 2004; Petschnigg et al. 2004] to reduce MC noise but

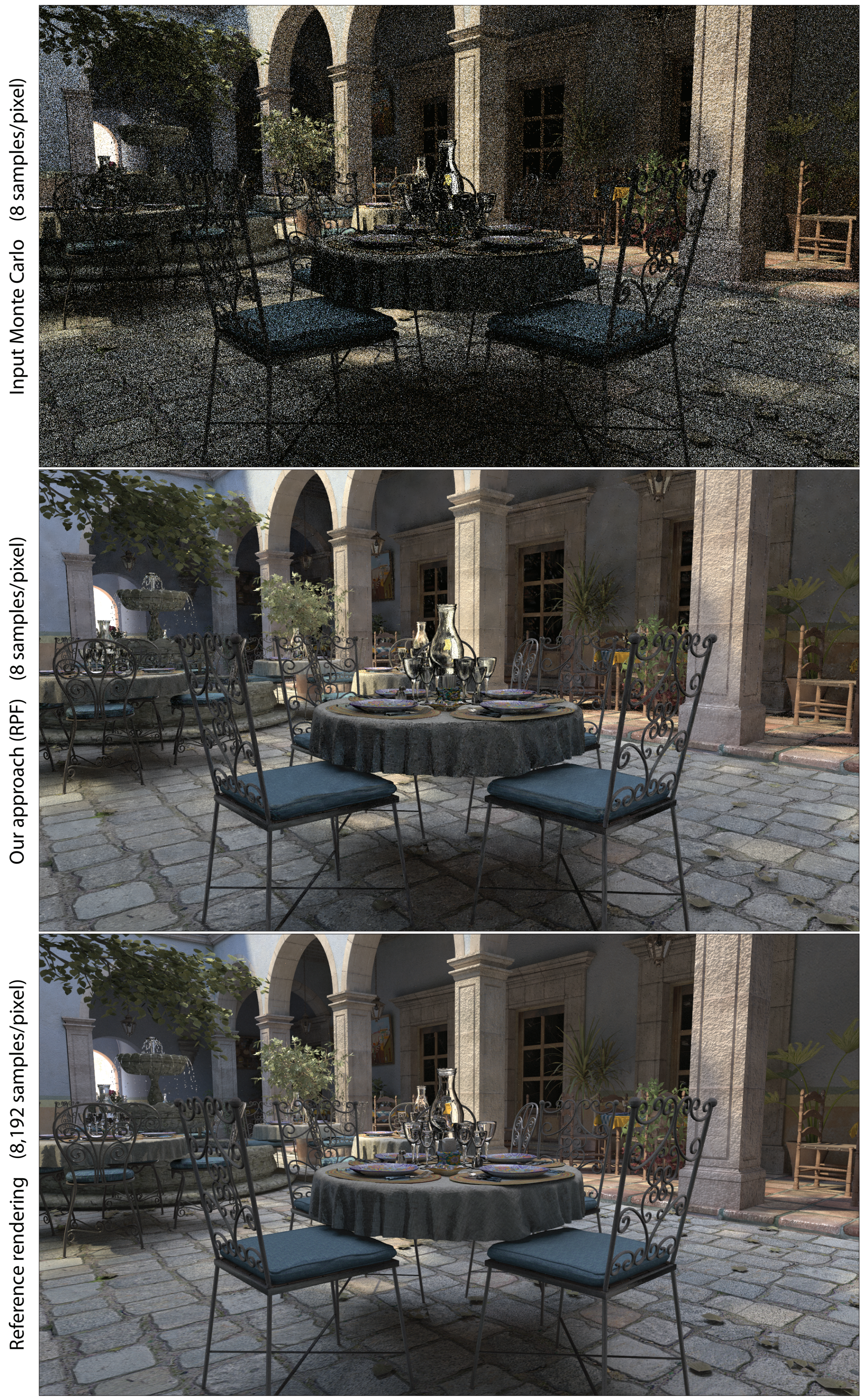

Input Monte Carlo (8 samples/pixel)
Our approach (RPF) (8 samples/pixel)
Reference rendering (8,192 samples/pixel)

Fig. 1. Our algorithm takes as input a small set of MC samples, which are fast to compute but very noisy. We then estimate the functional dependency between the sample values and the random parameters used in the rendering system, which enables us to filter out the MC noise without blurring scene detail. The result is comparable to a rendering with a large number of samples but more than 100× faster. This path-traced image shows the input rendering at 8 samples/pixel (top), the result of our method (middle), and the reference at 8,192 samples/pixel for comparison (bottom). The reference frame took more than 24 hours to compute, while ours took 14 minutes. This is the full-frame version of Fig. 1 in the paper [Sen and Darabi 2011b].

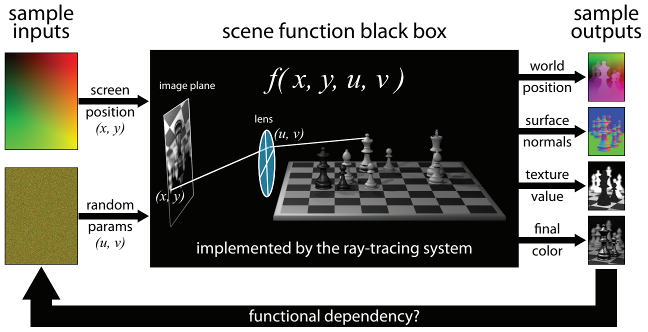


Fig. 2. We observe that determining *where* in the image we have Monte Carlo noise boils down to identifying the regions in which the sample values are functions of the random parameters. To do this, our algorithm treats the rendering system as a black box with scene function $f()$ that is evaluated deterministically by the ray tracing system for a specific scene, in this case the CHESS scene of Fig. 3 with depth-of-field. This function takes as its only inputs the x, y position on the image as well as the random parameters for Monte Carlo integration (here the u, v position on the lens). Since these are the only inputs to the deterministic system, the outputs of the black box must all be functions of these inputs (or constant with respect to them). These outputs are a set of features for each sample, such as world position, surface normal, texture value (the output of the texture lookup), and of course, sample color (the final shaded value). Our algorithm estimates the functional relationship by taking a set of samples in a neighborhood $\mathcal{N}$ and treating the input and output values of this neighborhood as statistical random variables. We then look for the *statistical dependence* of the outputs on the inputs using mutual information. This allows us to determine which scene features are highly dependent on our random parameters so that we can lower their weight during bilateral filtering.

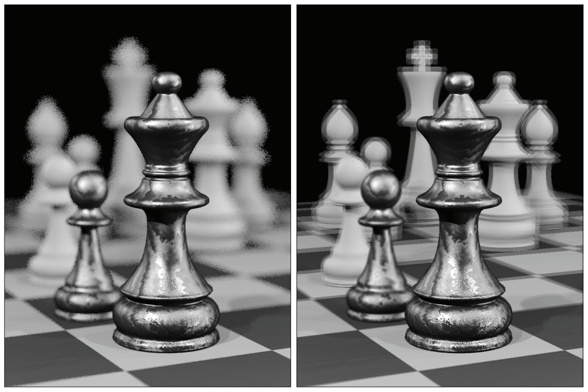


Fig. 3. The MC noise that we seek to eliminate is directly caused by the random parameters whenever the sample values are a function of them. We demonstrate this by changing the input parameters but keeping the scene function $f()$ the same, using the CHESS scene with depth of field as an example. If we set the u, v parameters to random values for each sample, we get the noisy image shown on the left. If, instead, we use uniform parameters for u, v, the noise is replaced with banding artifacts, shown on the right. Both images were rendered with 4 samples/pixel so here four sets of bands are clearly seen. Note that only the regions where the sample color is a function of the random parameters are affected by changing the parameters u, v, so the regions that are in focus in this DoF scene are mostly unchanged.

preserve scene detail. Unfortunately, finding the functional relationship between sample features and the random parameters in closed, mathematical form is impossible for complex scenes. Furthermore, finding where $f()$ is constant with respect to the random parameters is not easy with a small number of samples. To do this, we would ideally fix the sample position on the image x, y and vary the random parameters to see if they affect the output. However, because we can only compute a small number of samples, we typically vary x, y as well as the random parameters, which makes it difficult to determine whether differences in sample values are caused by changes in the random parameters or by changes in the image position x, y.

So to estimate these functional dependencies, we propose instead to treat the rendering system as a black box with scene function $f()$ as shown in Fig. 2, which outputs other scene features in addition to the sample color. We then model the inputs and outputs of $f()$ as random variables, and estimate the functional relationships by looking for statistical dependencies between them. To do this, we use the concept of *mutual information* from the field of information theory, which tells us how much information the inputs give us about a specific output.

To understand our model, we can conceptually replace the black box with a complex electronic circuit that implements function $f()$. For parts of the image where the outputs are functions of the random parameters (i.e., $f()$ varies with respect to the random parameters for these x, y image positions) there exists a connection somewhere in the complex circuitry between the random inputs and the outputs. On the other hand, if the outputs are not functions of the random parameters (i.e., $f()$ is constant with respect to them for a fixed x, y), the connection between the output and the random parameter inputs is effectively severed for these x, y regions in the image. This means that in these regions the "signal" from the random inputs never reaches the output, so they do not appear noisy in the final image.

With this analogy in mind, what we are doing in Monte Carlo rendering is "wiggling" the input parameters with a random number generator and observing the "wiggle" in the output in the form of noise, which only occurs when the output is a function of these noisy inputs (i.e., there is some path in the complex circuitry that connects them). The statistical dependence between the wiggle at the output and that at the inputs is what our mutual information metric uses to estimate the functional dependency between them.

Of course, there might be other ways to determine the functional relationship between the outputs and inputs, such as somehow fixing the samples positions in x, y while only changing the random parameters and looking at the variance of $f()$ to see if the outputs are affected. Although methods like this might be able to establish a functional dependency, one nice feature of the mutual information metric is that it scales up and down as the *amount* of dependency changes. For example, the sample color in a depth-of-field scene can become a function of the random position on the lens the instant that scene intersection point moves away from the focus plane, resulting in a variance metric that has almost a step-function response. The mutual information metric, on the other hand, scales gradually, so we can use it to size our cross-bilateral filter like we do in this work.

However, mutual information does not always perfectly estimate the functional relationship, as we discuss in the paper. It can fail, for example, if the wiggle in the output is masked by a complex random function inside the circuit model, such as when the output is the function of the input through a look-up table (e.g., a texture) that contains a set of random numbers as elements. This makes the input and output appear to be statistically independent and therefore hides their connection from the mutual information metric. However, compared to the other metrics we tested (see Sec. 6.3 of the paper [Sen and Darabi 2011b]), mutual information worked reasonably for a variety of complex scenes. Nevertheless, the study of other metrics for the purposes of random parameter filtering is an interesting topic for future research.

Table I. Notation used in this paper

n	number of random parameters used to render the scene
m	number of scene features available in the feature vector
s	number of samples per pixel
$\mathbf{x}_i$	sample vector containing the features of the ith sample
$\mathbf{p}_i$	the floating-point (x, y) position of the ith sample on the screen
$\mathbf{r}_i$	$n \times 1$ vector of random parameters used to compute the ith sample
$\mathbf{f}_i$	$m \times 1$ vector of scene features associated with the ith sample
$\mathbf{c}_i$	original color vector of the ith sample
$\bar{\mathbf{v}}$	vector with mean removed and normalized by standard deviation
$D_{\mathbf{q}}^{\mathbf{v}}$	dependency of $\mathbf{q}$ on $\mathbf{v}$
$W_{\mathbf{c},k}^{\mathbf{r}}$	fractional contribution of *all* random parameters on the kth color channel
$W_{\mathbf{f},k}^{\mathbf{r}}$	fractional contribution of *all* random parameters on the kth scene feature
$W_{\mathbf{c}}^{\mathbf{f},k}$	fractional contribution of the kth scene feature on *all* color channels
w_{ij}	weight that sample j contributes to sample i ($w_{ij} \neq w_{ji}$).
$\mathbf{c}'_i$	final filtered color vector of the ith sample
$\mathcal{P}$	a pixel in the image representing a set of s samples
$\mathcal{N}$	set that defines the neighborhood of samples used for filtering the samples
$\mathbf{m}_{\mathcal{Q}}^{\mathbf{v}}$	vector with per-element mean of vector $\mathbf{v}$ over set $\mathcal{Q}$
$\boldsymbol{\sigma}_{\mathcal{Q}}^{\mathbf{v}}$	vector with per-element standard deviation of vector $\mathbf{v}$ over set $\mathcal{Q}$
t	number of the iteration in our multi-pass filtering approach
M	maximum number of samples in neighborhood $\mathcal{N}$

Algorithm 1 Random Parameter Filtering (RPF) Algorithm

```
Input: scene to render and $s$ the number of samples/pixel
Output: final image
 1: Render scene with $s$ samples/pixel and output sample vector $\mathbf{x}$
    for every sample [Sec. 3.1]
 2: for all samples $i$ in image $\mathcal{I}$ do
 3:     $\mathbf{c}'_i \leftarrow \mathbf{c}_i$
 4: end for
 5: $box = \{55, 35, 17, 7\}$ [Sec. 3.2]
 6: for iteration step $t = 0, 1, 2, 3$ do
 7:     box size $b = box[t]$
 8:     max number of samples $M \leftarrow b^2 \times s/2$
 9:     for all pixels $\mathcal{P}$ in image $\mathcal{I}$ do
10:         $\mathcal{N} \leftarrow$ Pre-process Samples($\mathcal{P}, b, M$) [Sec. 4]
11:         $\{\alpha, \beta\} \leftarrow$ Compute Feature Weights($t, \mathcal{N}$) [Sec. 5]
12:         $\mathbf{c}'' \leftarrow$ Filter Color Samples($\mathcal{P}, \mathcal{N}, \alpha, \beta, \mathbf{c}'$) [Sec. 6]
13:     end for
14:     for all samples $i$ in image $\mathcal{I}$ do
15:         $\mathbf{c}'_i \leftarrow \mathbf{c}''_i$
16:     end for
17: end for
    /* all samples in image $\mathcal{I}$ have been filtered... box filter to com-
    pute final pixel values */
18: for all pixels $\mathcal{P}$ in image $\mathcal{I}$ do
19:     Box filter samples in pixel $\mathcal{P}$ to compute final pixel value
20: end for
21: return final image
```

Note that although the presence of MC noise requires that $f()$ vary with respect to the random parameters in parts of the image (i.e., there should be a connection between the random inputs and the outputs in these regions), we attribute the noise we see in the final image to the random parameters themselves, not to the variance in the integrand $f()$. We can see this clearly if we replace the input parameters with uniformly sampled values, as shown in Fig. 3. Although the scene function $f()$ is unchanged, the noise has been replaced by banding artifacts in which the differences between sample values (which are used by our bilateral filter) have different properties than when generated with random numbers. Of course, only regions where the sample value is a function of the random parameters are affected by this change in input, while the other regions remain the same. Our algorithm uses mutual information to detect the presence of noise from the inputs in the functional outputs, which is why we say that it is able to filter out the noise generated by the random parameters in Monte Carlo rendering.

One advantage of our proposed algorithm is that it is general and can handle any of the effects that can be computed using Monte Carlo integration. Unlike some of the other algorithms that have been proposed (e.g., [Soler et al. 2009; Egan et al. 2009]), we do not need to attribute physical meaning to the random parameters. Instead, our algorithm embodies the spirit of traditional Monte Carlo integration, where one only needs to provide random values for each parameter that the system should integrate over. In our case, we must simply specify the random parameters that are to be filtered so that the algorithm can remove the noise from each of these effects in the final image. This allows us to handle a wide range of Monte Carlo effects, including depth of field, area light sources, motion blur, path-tracing, etc. We can also handle effects that require discrete integration (summation) such as integrating over multiple light sources using a discrete random number to select between them, or using Russian roulette to randomly either transmit or reflect a ray off a semitransparent surface.

We present most of the key theoretical ideas behind the random parameter filtering algorithm in our main paper [Sen and Darabi 2011b]. In this technical report, we supplement that discussion with pseudocode and more implementation detail that will be helpful to readers who are interested in reproducing our algorithm. Since some of the discussion here is by necessity duplicated from our original paper, we have taken the liberty of reusing text from that paper when appropriate but without citation in order not to clutter the discussion. We also present a set of results using test scenes as well as scenes from the paper. Note section and figure references refer to this document unless we specify that we are referring to something in the original paper.

2. PSEUDOCODE

We begin by presenting the pseudocode for our entire algorithm, which is divided into four parts. Algorithm 1 shows the overall algorithm and explains how each of the individual pieces fit together. Algorithm 2 explains how we preprocess the samples by clustering them and removing their mean and standard deviation (Sec. 4). Algorithm 3 shows how we statistically compute the feature weights for our cross-bilateral filter using the statistical dependency of features on the random variables (Sec. 5). Finally, Algorithm 4 shows how to filter the samples (Sec. 6). The sections that follow go into sufficient detail to make it possible for interested readers to implement the RPF algorithm. The notation we shall use in this technical report is similar to that of the main paper and is listed in Table I.

3. RENDERING THE SAMPLES AND SETTING UP OUR POST-PROCESS FILTER

Our algorithm is a post-process filter, so the first step is to render the samples at the given sampling rate of s samples/pixel and then we apply our filtering process which consecutively performs the filter in several iterations.

3.1 Rendering samples and creating feature vectors

We first render the samples at a fixed sampling density and store vector $\mathbf{x}$ for each sample. The data structure is simply a floating point array with enough space for the number of samples (computed as the image resolution times the number of samples/pixel

s) and enough space for each (we used 27 floats per sample to store all the information). For the scene features in $\mathbf{f}$, our algorithm stores for each sample the normal, world-space position, and texture values (the set of floats from texture lookups used by the surface shader to produce the surface color) for the first intersection point of the ray, and the world position and normal for the second intersection in a path tracer. The same features are stored for every scene, even if an object does not have the specific feature (a zero is substituted) or if the shader does not use the feature when computing the final color (features that do not affect the final color are ignored). Since all these features are available to the rendering system at some point during the tracing of the ray, outputting the feature vector for the sample is simply a matter of caching the information after it is calculated. This is standard practice in rendering systems when creating a G-buffer [Saito and Takahashi 1990] for deferred shading [Deering et al. 1988].

In addition to these scene-dependent features, our algorithm also stores the random parameters used by the Monte Carlo system so that it can identify the functional relationships between the inputs and the outputs. Wherever possible, we use these random parameters in the form that most closely reflects their use in the renderer. For example, the random u, v position on the lens for depth of field can be computed in several ways: we can use two uniformly distributed random numbers from 0 to 1 that are then scaled and translated into a -1 to 1 range on a square lens, or we can use them to pick θ and r values that uniformly sample a circular lens without rejection, etc. Rather than deal with the raw random parameters that have no physical meaning, we use the final random values as they are used by the rendering system. In the case of the position on the lens, we would use the final u, v values ranging from -1 to 1 as our random parameters because these relate to the physical process simulated by the rendering system. In most cases the random parameters are floating point values, but they could also be integers, such as when we use a discrete random number to select an individual light source for lighting with multiple light sources.

Note that in industrial rendering systems these random parameters are often determined with pre-computed sequences of low-discrepancy numbers provided to the renderer. In this case, we would not need to store the random parameters in the sample vector since the post-process filter could use this same sequence to recompute the random parameters on the fly. In our implementation, however, we did the brute-force approach and simply saved out all of the random parameters our scenes were using. We used the PBRT2 [Pharr and Humphreys 2010] and LuxRender [LuxRender 2011] Monte Carlo rendering systems to compute the samples.

3.2 Applying multiple filter iterations

To estimate the functional dependencies of sample values on the inputs to the MC rendering system using mutual information, we must select a set of samples to process. We cannot use every sample in the image because the functional dependencies change from region to region (e.g., an image may have some regions in focus and others out of focus, and these have different dependencies on the random parameters). Therefore, as we loop over every pixel in the image, we select a local neighborhood of samples $\mathcal{N}$ around that pixel to measure the local statistics for mutual information. However, we need to decide how big to make the block size that defines the extent of neighborhood $\mathcal{N}$.

If we use a large block size, there will be more samples to calculate statistics (improving the accuracy of our dependency estimates) and provide us with more samples to filter out noise. Unfortunately, larger block sizes have less locality and might cause problems when the block overlaps regions with different functional dependencies, such as regions where the amount of defocus blur changes. To resolve these two competing considerations, we found it best to use a multi-pass approach, where our algorithm loops over the image several times using different block sizes. We start at a larger block size and then shrink it down in a series of iterations. We found four iterations to be sufficient, starting at a block size of 55 pixels wide and then going down to 35, 17 and finally 7. At each step, we filter the samples' colors with the weighted bilateral filter of Eq. 17 using the samples in $\mathcal{N}$, and then use that new filtered color in the next pass of the algorithm (except to compute statistical dependencies, since they are always computed with the original sample color).

By going from larger to smaller, we first address the low-frequency noise that a smaller filter kernel would leave behind and then, as we reduce the block size, we eliminate the localized noise and clean up the detail. The multi-pass approach also reduces the maximum block size needed for filtering, since we can emulate a larger filter by progressively applying a smaller kernel. This allows us to get good performance and quality at the same time.

Algorithm 2 Pre-process Samples [Sec. 4]

```
Input: set of samples in pixel $\mathcal{P}$, box size $b$, maximum number of
    samples $M$
Output: set of samples in neighborhood $\mathcal{N}$
 1: $\sigma_{\mathbf{p}} \leftarrow b/4, \mathcal{N} \leftarrow \mathcal{P}$
 2: Compute mean $(\mathbf{m}^{\mathbf{f}}_{\mathcal{P}})$ and standard deviation $(\sigma^{\mathbf{f}}_{\mathcal{P}})$ of the fea-
    tures of samples in pixel $\mathcal{P}$ for clustering
    /* add samples to neighborhood $\mathcal{N}$ */
 3: for $q = 1$ to $M - s$ do
 4:   Select a random sample $j$ from samples inside the box but
      outside $\mathcal{P}$ with distribution based on $\sigma_{\mathbf{p}}$ [Sec. 4.1]
 5:   $flag \leftarrow 1$
      /* perform clustering [Sec. 4.2] */
 6:   for scene feature $k = 1$ to $m$ do
 7:     if $|\mathbf{f}_{j,k} - \mathbf{m}^{\mathbf{f}}_{\mathcal{P},k}| > \{3|30\}\sigma^{\mathbf{f}}_{\mathcal{P},k}$ and
          $|\mathbf{f}_{j,k} - \mathbf{m}^{\mathbf{f}}_{\mathcal{P},k}| > 0.1$ or $\sigma^{\mathbf{f}}_{\mathcal{P},k} > 0.1$ then
 8:       $flag \leftarrow 0$
 9:       break
10:     end if
11:   end for
12:   if $flag$ equals to 1 then
13:     $\mathcal{N} \leftarrow$ sample $j$
14:   end if
15: end for
    /* neighborhood $\mathcal{N}$ now ready for statistical analysis */

    /* compute normalized vector for each sample by removing
    mean and dividing by standard deviation [Sec. 4.3] */
16: Compute mean $(\mathbf{m}^{\mathbf{x}}_{\mathcal{N}})$ and standard deviation $(\sigma^{\mathbf{x}}_{\mathcal{N}})$ of samples
    in neighborhood $\mathcal{N}$
17: for all samples $i$ in $\mathcal{N}$ do
18:   $\bar{\mathbf{x}}_i \leftarrow (\mathbf{x}_i - \mathbf{m}^{\mathbf{x}}_{\mathcal{N}})/\sigma^{\mathbf{x}}_{\mathcal{N}}$
19: end for

20: return set of samples in neighborhood $\mathcal{N}$
```

4. PRE-PROCESSING THE SAMPLES

After we have rendered our samples, we are ready to apply the random parameter filtering algorithm, which is performed independently for every pixel of the image. As we loop over each pixel, the first thing we need to do is to pre-process the samples in the block around the pixel to create a neighborhood $\mathcal{N}$ of samples to use to compute statistics and perform filtering. This section follows Sec. 4.3 of the original paper and the pseudocode listed in Alg. 2.

4.1 A statistical approach to Gaussian filtering

To run our algorithm, we need to apply our bilateral filter to the samples in the block of pixels. However, for large block sizes, the process of calculating the contribution of all samples to any given sample is time consuming because the number of samples increases as $O(N^2)$ with block size. To accelerate this process, we select a random subset of samples within the block and only use these samples for statistical analysis and to filter the samples within the pixel. This is a form of Monte Carlo estimation as well, and it significantly accelerates our calculations although it introduces a slight error. However, we found this small error to be reasonable considering the improvement in running time for our algorithm. This is a small acceleration modification that we mention briefly in Sec. 4.5.1 of the original paper [Sen and Darabi 2011b].

Our bilateral filter in Eq. 17 weights samples based on screen position distance with a Gaussian of variance $\sigma_{\mathbf{p}}^2$ that depends on the block size ($\sigma_{\mathbf{p}} = b/4$). Since we are picking a set of random samples, we can draw them with a Gaussian distribution with variance $\sigma_{\mathbf{p}}^2$ around the pixel in question in order to essentially perform importance sampling [Robert and Casella 2005]. This allows us to remove the first term from our bilateral filter calculation as discussed in Sec. 6.

4.2 Clustering samples to avoid mixing statistics

As mentioned in Sec. 4.3.1 of the original paper [Sen and Darabi 2011b], we need to perform some kind of clustering when placing samples in neighborhood $\mathcal{N}$ to avoid mixing statistics. To begin, we always include in neighborhood $\mathcal{N}$ the set of samples $\mathcal{P}$ at the current pixel: $\mathcal{N} \Leftarrow \mathcal{P}$. We then pick a random set of samples with a Gaussian distribution from the block of pixels (as described in Sec. 4.1) and add them to the neighborhood $\mathcal{N}$ only if all of their scene features $\mathbf{f}$ are within 3 standard deviations of the mean for the pixel. So given sample j, that is within the block of pixels:

$$\mathcal{N} \Leftarrow \mathcal{N} \cup j \quad \text{if} \quad |\mathbf{f}_{j,k} - \mathbf{m}^{\mathbf{f}}_{\mathcal{P},k}| < 3\boldsymbol{\sigma}^{\mathbf{f}}_{\mathcal{P},k} \quad \text{for all } k.$$

In practice, we use a value of 30 instead of 3 when testing the world position since it varies much more than the other features. Also, we only do this test when $\boldsymbol{\sigma}^{\mathbf{f}}_{\mathcal{P},k} > 0.1$ because we do not want to throw all the samples away in cases where the variance is very small, such as constant-valued regions.

4.3 Normalization of scene features

Before we compute the statistical dependencies for a set of samples in a neighborhood, we must first remove the mean and divide by the standard deviation for each of the elements in the sample vector. The reason for this is that the features in $\mathbf{f}$ reside in very different coordinate systems (world positions could be in the range of 0 to 1000, while the normal vector could have components in the range of 0 to 1, for example). If we do not correct for this discrepancy, we would inadvertently give larger weight to certain features when calculating dependency that may not necessarily be more important. This procedure is quite common in the machine learning community as well, since they are confronted with a similar problem [Hastie et al. 2001]. This is also related to Mahalanobis distance [Mahalanobis 1936], but in this case it is as if we assume that the covariance between scene features is zero, resulting in a diagonal covariance matrix. We represent vectors that have been normalized in this manner with a bar (e.g., $\mathbf{f}$ becomes $\bar{\mathbf{f}}$).

Algorithm 3 Compute Feature Weights [Sec. 5]

```
Input: iteration step $t$, set of samples in neighborhood $\mathcal{N}$
Output: color weights $\alpha$ and feature weights $\beta$ for bilateral filter
    /* Compute the dependencies for the colors using the samples in $\mathcal{N}$ */
 1: for color channel $k = 1$ to 3 do
 2:     for random parameter $l = 1$ to $n$ do
 3:         $D^{\mathbf{r},l}_{\mathbf{c},k} \leftarrow \mu(\bar{\mathbf{c}}_{\mathcal{N},k}; \bar{\mathbf{r}}_{\mathcal{N},l})$ with Eq. 1
 4:     end for
 5:     Calculate $D^{\mathbf{r}}_{\mathbf{c},k}$ with Eq. 4 using the $D^{\mathbf{r},l}_{\mathbf{c},k}$ terms

 6:     for position parameter $l = 1$ to 2 do
 7:         $D^{\mathbf{p},l}_{\mathbf{c},k} \leftarrow \mu(\bar{\mathbf{c}}_{\mathcal{N},k}; \bar{\mathbf{p}}_{\mathcal{N},l})$ with Eq. 1
 8:     end for
 9:     Calculate $D^{\mathbf{p}}_{\mathbf{c},k}$ with Eq. 5 using the $D^{\mathbf{p},l}_{\mathbf{c},k}$ terms

10:     for scene feature $l = 1$ to $m$ do
11:         $D^{\mathbf{f},l}_{\mathbf{c},k} \leftarrow \mu(\bar{\mathbf{c}}_{\mathcal{N},k}; \bar{\mathbf{f}}_{\mathcal{N},l})$ with Eq. 1
12:     end for
13:     Calculate $D^{\mathbf{f}}_{\mathbf{c},k}$ with Eq. 6 using the $D^{\mathbf{f},l}_{\mathbf{c},k}$ terms

14:     Calculate $W^{\mathbf{r}}_{\mathbf{c},k}$ with Eq. 10 using $D^{\mathbf{r}}_{\mathbf{c},k}$ and $D^{\mathbf{p}}_{\mathbf{c},k}$
        /* I now have everything I need to compute $\alpha_k$ */
15:     $\alpha_k \leftarrow \max(1 - 2(1 + 0.1t)W^{\mathbf{r}}_{\mathbf{c},k}, 0)$
16: end for
17: Calculate $D^{\mathbf{r}}_{\mathbf{c}}$ and $D^{\mathbf{p}}_{\mathbf{c}}$ and $D^{\mathbf{f}}_{\mathbf{c}}$ by adding up the $D^{\mathbf{r}}_{\mathbf{c},k}$, $D^{\mathbf{p}}_{\mathbf{c},k}$, and $D^{\mathbf{f}}_{\mathbf{c},k}$ terms over the three color channels with Eq. 8
    /* Compute the dependencies for the scene features using samples in $\mathcal{N}$ */
18: for scene feature $k = 1$ to $m$ do
19:     for random parameter $l = 1$ to $n$ do
20:         $D^{\mathbf{r},l}_{\mathbf{f},k} \leftarrow \mu(\bar{\mathbf{f}}_{\mathcal{N},k}; \bar{\mathbf{r}}_{\mathcal{N},l})$ with Eq. 1
21:     end for
22:     Calculate $D^{\mathbf{r}}_{\mathbf{f},k}$ with Eq. 2 using the $D^{\mathbf{r},l}_{\mathbf{f},k}$ terms

23:     for position parameter $l = 1$ to 2 do
24:         $D^{\mathbf{p},l}_{\mathbf{f},k} \leftarrow \mu(\bar{\mathbf{f}}_{\mathcal{N},k}; \bar{\mathbf{p}}_{\mathcal{N},l})$ with Eq. 1
25:     end for
26:     Calculate $D^{\mathbf{p}}_{\mathbf{f},k}$ with Eq. 3 using the $D^{\mathbf{p},l}_{\mathbf{f},k}$ terms

27:     Calculate $D^{\mathbf{f},k}_{\mathbf{c}}$ with Eq. 7 using the $D^{\mathbf{f},k}_{\mathbf{c},l}$ terms from line 11

28:     Calculate $W^{\mathbf{r}}_{\mathbf{f},k}$ with Eq. 9 using $D^{\mathbf{r}}_{\mathbf{f},k}$ and $D^{\mathbf{p}}_{\mathbf{f},k}$

29:     Calculate $W^{\mathbf{f},k}_{\mathbf{c}}$ with Eq. 12 using $D^{\mathbf{f},k}_{\mathbf{c}}$, $D^{\mathbf{r}}_{\mathbf{c}}$, $D^{\mathbf{p}}_{\mathbf{c}}$, and $D^{\mathbf{f}}_{\mathbf{c}}$

30:     $\beta_k \leftarrow W^{\mathbf{f},k}_{\mathbf{c}} \cdot \max(1 - (1 + 0.1t)W^{\mathbf{r}}_{\mathbf{f},k}, 0)$
31: end for

32: return $\alpha$ and $\beta$
```

5. COMPUTING THE FEATURE WEIGHTS

This section describes the core of our algorithm that computes the color/feature weights α and β. We begin by explaining how we use mutual information to compute the statistical dependencies between a sample feature and the inputs to the Monte Carlo system, and finish by describing how we calculate the weights for our bilateral filter.

5.1 Calculating mutual information

Since it is difficult to derive an exact functional relationship between scene features and the inputs of the rendering system $\mathbf{p}_i$ and $\mathbf{r}_i$ for complex scenes, we propose instead to see if there is a statistical dependency (i.e., does knowing the inputs tell us something about the scene features). This is the basic meaning of mutual information, a concept from the field of information theory [Cover and Thomas 2006], which is the exact measure of dependence between two random variables and indicates how much information one tells us about another. The mutual information between two random variables X and Y can be calculated as:

$$\mu(X;Y) = \sum_{x\in X}\sum_{y\in Y} p(x,y)\log\frac{p(x,y)}{p(x)p(y)}, \tag{1}$$

where these probabilities are computed over the neighborhood of samples $\mathcal{N}$ around a given pixel.

To calculate the mutual information between two vectors $\mathbf{x}$ and $\mathbf{y}$ (or in our case, e.g., $\bar{\mathbf{f}}_{\mathcal{N},k}$ and $\bar{\mathbf{r}}_{\mathcal{N},l}$), we first calculate the histogram of each of them (for computing $p(x)$ and $p(y)$) as well as their joint histogram (for $p(x,y)$) and plug their probabilities into Eq. 1 to get $\mu(\mathbf{x},\mathbf{y})$. To compute the histograms, we first make all the values positive by subtracting the minimum element in the vector and quantize the elements into integer bins by rounding their values. We count how many times the values of $\mathbf{x}$ fall inside each bin and find the probabilities by dividing by the length of $\mathbf{x}$. The joint histogram is calculated in a similar way, except with pairs of values $(\mathbf{x},\mathbf{y})$. To implement this, we examined the mutual information code from the Matlab Central website [Peng 2007] and rewrote our own version in C.

5.2 Estimating statistical dependencies on inputs

We begin by showing how we calculate the dependency of the k^{th} scene feature on all random parameters (given by $D^{\mathbf{r}}_{\mathbf{f},k}$) using mutual information. Our heuristic approximates this by measuring the dependency on individual random parameters and adding them up. Therefore, we first calculate the statistical dependency between the k^{th} scene feature and the l^{th} random parameter by $D^{\mathbf{r},l}_{\mathbf{f},k} = \mu(\bar{\mathbf{f}}_{\mathcal{N},k};\bar{\mathbf{r}}_{\mathcal{N},l})$, and then approximate the dependency of the k^{th} scene feature on all n random parameters as:

$$D^{\mathbf{r}}_{\mathbf{f},k} = \sum_{1\le l\le n} D^{\mathbf{r},l}_{\mathbf{f},k} = \sum_{1\le l\le n}\mu(\bar{\mathbf{f}}_{\mathcal{N},k};\bar{\mathbf{r}}_{\mathcal{N},l}). \tag{2}$$

The dependency of the k^{th} scene feature on screen position ($D^{\mathbf{p}}_{\mathbf{f},k}$) and color dependencies $D^{\mathbf{r}}_{\mathbf{c},k}$ and $D^{\mathbf{p}}_{\mathbf{c},k}$ are similarly computed:

$$D^{\mathbf{p}}_{\mathbf{f},k} = \sum_{1\le l\le 2} D^{\mathbf{p},l}_{\mathbf{f},k} = \sum_{1\le l\le 2}\mu(\bar{\mathbf{f}}_{\mathcal{N},k};\bar{\mathbf{p}}_{\mathcal{N},l}), \tag{3}$$

$$D^{\mathbf{r}}_{\mathbf{c},k} = \sum_{1\le l\le n} D^{\mathbf{r},l}_{\mathbf{c},k} = \sum_{1\le l\le n}\mu(\bar{\mathbf{c}}_{\mathcal{N},k};\bar{\mathbf{r}}_{\mathcal{N},l}), \tag{4}$$

$$D^{\mathbf{p}}_{\mathbf{c},k} = \sum_{1\le l\le 2} D^{\mathbf{p},l}_{\mathbf{c},k} = \sum_{1\le l\le 2}\mu(\bar{\mathbf{c}}_{\mathcal{N},k};\bar{\mathbf{p}}_{\mathcal{N},l}). \tag{5}$$

We also compute the dependency of the k^{th} color channel on all the scene features, so that later we can reduce the weight for features that do not contribute to the final color:

$$D^{\mathbf{f}}_{\mathbf{c},k} = \sum_{1\le l\le m} D^{\mathbf{f},l}_{\mathbf{c},k} = \sum_{1\le l\le 2}\mu(\bar{\mathbf{c}}_{\mathcal{N},k};\bar{\mathbf{f}}_{\mathcal{N},l}). \tag{6}$$

We also compute a related term, how all color channels are dependent on the k^{th} scene feature:

$$D^{\mathbf{f},k}_{\mathbf{c}} = \sum_{1\le l\le 3} D^{\mathbf{f},k}_{\mathbf{c},l} = \sum_{1\le l\le 3}\mu(\bar{\mathbf{c}}_{\mathcal{N},l};\bar{\mathbf{f}}_{\mathcal{N},k}). \tag{7}$$

Finally, the $D^{\mathbf{r}}_{\mathbf{c}}$, $D^{\mathbf{p}}_{\mathbf{c}}$, and $D^{\mathbf{f}}_{\mathbf{c}}$ terms are calculated by summing over the color channels:

$$D^{\mathbf{r}}_{\mathbf{c}} = \sum_{1\le k\le 3} D^{\mathbf{r}}_{\mathbf{c},k}, \quad D^{\mathbf{p}}_{\mathbf{c}} = \sum_{1\le k\le 3} D^{\mathbf{p}}_{\mathbf{c},k}, \quad D^{\mathbf{f}}_{\mathbf{c}} = \sum_{1\le k\le 3} D^{\mathbf{f}}_{\mathbf{c},k}. \tag{8}$$

5.3 The error of our approximation

Ideally, we would calculate the statistical dependency of the k^{th} scene feature on all random parameters using the joint mutual information $\mu(\mathbf{r}_{\mathcal{N},1},\mathbf{r}_{\mathcal{N},2},\ldots,\mathbf{r}_{\mathcal{N},n};\mathbf{f}_{\mathcal{N},k})$. Unfortunately, this joint mutual information can be difficult and expensive to compute as the number n gets larger, because the histogram grows to the power of n while the number of samples we have to do statistics grows linearly. This means that our ability to compute the histogram accurately degenerates quickly and it becomes much slower to do so (the curse of dimensionality strikes again). For this reason, we approximate this instead by measuring the dependency on individual random parameters and adding them up as described in the last section. Here, we examine what effect this has in our overall calculation of statistical dependency.

To keep things simple, let us assume that we have two statistically independent random variables R_1 and R_2 that are the inputs to the system and produce output feature Y. We would like to measure $\mu(R_1,R_2;Y)$, but instead we approximate it as $\mu(R_1;Y)+\mu(R_2;Y)$. What difference does this make? The derivation below (where $H()$ is entropy) shows that we are underestimating the statistical dependence:

$$\begin{aligned}\mu(R_1,R_2;Y) &= H(R_1,R_2) - H(R_1,R_2|Y)\\ &= H(R_1)+H(R_2|R_1)-H(R_1|Y)-H(R_2|R_1,Y)\\ &= \mu(R_1;Y) + H(R_2|R_1) - H(R_2|R_1,Y)\end{aligned}$$

If R_1 and R_2 are independent, then $H(R_2|R_1) = H(R_2)$. So:

$$\begin{aligned}\mu(R_1,R_2;Y) &= \mu(R_1;Y) + H(R_2) - H(R_2|R_1,Y)\\ &= \mu(R_1;Y) + \mu(R_2;R_1,Y)\\ &= \mu(R_1;Y) + \mu(Y,R_1;R_2)\\ &= \mu(R_1;Y) + \mu(R_2;Y) + \mu(R_1;R_2|Y)\end{aligned}$$

So our approximation that $\mu(R_1,R_2;Y) \approx \mu(R_1;Y)+\mu(R_2;Y)$ effectively assumes that $\mu(R_1;R_2|Y) = 0$. This means that we are essentially ignoring the information that the output feature tells us about relationship between the inputs, which might not be zero even though the inputs are statistically independent. To understand why, we can set the function $f()$ to act as an XOR gate of two inputs. If we know one of the inputs and the output, we can automatically determine the other input, even though the two inputs may be statistically independent.

Since $\mu(R_1;R_2|Y) \ge 0$, our approximation is an underestimate of the true joint mutual information between the random parameters and the scene feature. However, in practice we found that our approximation works quite reasonably, even for intricate scenes with complex relationships between the random parameters and the scene features.

5.4 Computing the fractional contributions

Since the sample features are only functions of the random parameters $\mathbf{r}_i$ and the screen position $\mathbf{p}_i$, our heuristic computes the fractional contribution of the random parameters to the k^{th} scene feature with the following formula:

$$W^{\mathbf{r}}_{\mathbf{f},k} = \frac{D^{\mathbf{r}}_{\mathbf{f},k}}{D^{\mathbf{r}}_{\mathbf{f},k} + D^{\mathbf{p}}_{\mathbf{f},k} + \varepsilon}, \tag{9}$$

Note the addition of the ε term to Eq. 3 of the original paper [Sen and Darabi 2011b]. This addition prevents degeneration when the dependencies $D^{\mathbf{r}}_{\mathbf{f},k}$ and $D^{\mathbf{p}}_{\mathbf{f},k}$ are both small. This expression tells us how much the k^{th} feature was affected by the random parameters as a fraction of the contributions from both sets of inputs, with the reasonable assumption that the position and random parameters are statistically independent. When the sample is only a function of the random parameters, this value will be close to 1, and when it is dependent only on the screen position it will be 0. In the common case where we have some contribution from both inputs (e.g., a partially out-of-focus object is dependent on both screen position and (u, v)), Eq. 9 simply interpolates between the two.

We also have a similar formula using the dependencies of the k^{th} sample color channel on the random parameters ($D^{\mathbf{r}}_{\mathbf{c},k}$) and on the screen position ($D^{\mathbf{p}}_{\mathbf{c},k}$) to compute the fractional contribution of the random parameters on the k^{th} color channel :

$$W^{\mathbf{r}}_{\mathbf{c},k} = \frac{D^{\mathbf{r}}_{\mathbf{c},k}}{D^{\mathbf{r}}_{\mathbf{c},k} + D^{\mathbf{p}}_{\mathbf{c},k} + \varepsilon}. \tag{10}$$

We will also need the overall contribution of the random parameters on the color $W^{\mathbf{r}}_{\mathbf{c}}$ for use in sizing our filter in Eq. 19, which we get by averaging the $W^{\mathbf{r}}_{\mathbf{c},1}$, $W^{\mathbf{r}}_{\mathbf{c},2}$, $W^{\mathbf{r}}_{\mathbf{c},3}$ terms:

$$W^{\mathbf{r}}_{\mathbf{c}} = \frac{1}{3}(W^{\mathbf{r}}_{\mathbf{c},1} + W^{\mathbf{r}}_{\mathbf{c},2} + W^{\mathbf{r}}_{\mathbf{c},3}). \tag{11}$$

Finally, we compute the $W^{\mathbf{f},k}_{\mathbf{c}}$ term that tells us how much the color depends on a specific feature:

$$W^{\mathbf{f},k}_{\mathbf{c}} = \frac{D^{\mathbf{f},k}_{\mathbf{c}}}{D^{\mathbf{r}}_{\mathbf{c}} + D^{\mathbf{p}}_{\mathbf{c}} + D^{\mathbf{f}}_{\mathbf{c}}}. \tag{12}$$

5.5 Computing α and β

Once the statistical dependencies have been calculated, we compute the normalized dependencies we use to determine our $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ parameters. In our original paper [Sen and Darabi 2011b], we use the formulas:

$$\alpha_k = 1 - W^{\mathbf{r}}_{\mathbf{c},k}, \tag{13}$$

$$\beta_k = W^{\mathbf{f},k}_{\mathbf{c}}(1 - W^{\mathbf{r}}_{\mathbf{f},k}), \tag{14}$$

which are Eqs. 6 and 13 from the original paper. However, we found that the results are slightly improved if we adjust $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ in each iteration as the block size decreases by giving more weight to the dependency on the random parameters. The idea behind this is that when the block sizes are large, there will be an increase in dependency on the spatial screen position because of the natural spatial variations in the image, but at the same time our statistics are more corrupted because of the mixing of statistics that happens with large block sizes as discussed in Sec. 4.3.1 of the paper. Therefore, we give more weight to the normalized dependency on the random parameters as the block size goes down with each iteration. To do this adjustment, we modify Eqs. 13 and 14 to be:

$$\alpha_k = \max(1 - 2(1 + 0.1t)W^{\mathbf{r}}_{\mathbf{c},k}, 0), \tag{15}$$

$$\beta_k = W^{\mathbf{f},k}_{\mathbf{c}} \cdot \max(1 - (1 + 0.1t)W^{\mathbf{r}}_{\mathbf{f},k}, 0), \tag{16}$$

where the t term is the iteration of our multi-pass approach, with the first pass $t = 0$. The incorporation of the t term increases the weight of $W^{\mathbf{r}}_{\mathbf{c},k}$ and $W^{\mathbf{r}}_{\mathbf{f},k}$ upon each successive iteration, and the $\max()$ term is added to ensure that the value stays positive.

6. FILTERING THE SAMPLES

Our approach filters the color of samples $\mathbf{x}_i$ using a weighted bilateral filter in which the importance of the color and scene features is adjusted to reflect their dependence on the random parameters:

Algorithm 4 Filter Color Samples [Sec. 6]

Input: set of samples in pixel $\mathcal{P}$, set of samples in neighborhood $\mathcal{N}$, color α and feature weights β, and previous sample colors $\mathbf{c}'$

Output: filtered color of the samples $\mathbf{c}''$

```
1:  Calculate $W^{\mathbf{r}}_{\mathbf{c}}$ with Eq. 11 using the $W^{\mathbf{r}}_{\mathbf{c},k}$ terms from line 14 in Alg. 3
2:  $\sigma^2 \leftarrow 8\sigma_8^2/s$
3:  $\sigma^2_{\mathbf{c}} = \sigma^2_{\mathbf{f}} \leftarrow \frac{\sigma^2}{(1-W^{\mathbf{r}}_{\mathbf{c}})^2}$
    /* filter the colors of samples in pixel $\mathcal{P}$ using bilateral filter */
4:  for all samples $i$ in $\mathcal{P}$ do
5:      $\mathbf{c}''_i \leftarrow 0, w \leftarrow 0$
6:      for all samples $j$ in $\mathcal{N}$ do
7:          Calculate $w_{ij}$ with Eq. 18 using $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$
8:          $\mathbf{c}''_i \leftarrow \mathbf{c}''_i + w_{ij}\mathbf{c}'_j$
9:          $w \leftarrow w + w_{ij}$
10:     end for
11:     $\mathbf{c}''_i \leftarrow \mathbf{c}''_i/w$
12: end for
    /* address issues with HDR [Sec. 6.1] */
13: Compute mean $\mathbf{m}^{\mathbf{c}''}_{\mathcal{P}}$ and std. dev. $\sigma^{\mathbf{c}''}_{\mathcal{P}}$ of filtered colors
14: for all samples $i$ in $\mathcal{P}$ do
15:     for color channel $k = 1$ to 3 do
16:         if $(\mathbf{c}''_{i,k} - \mathbf{m}^{\mathbf{c}''}_{\mathcal{P},k}) > \sigma^{\mathbf{c}''}_{\mathcal{P},k}$ then
17:             $\mathbf{c}''_{i,k} \leftarrow \mathbf{m}^{\mathbf{c}''}_{\mathcal{P},k}$
18:         end if
19:     end for
20: end for
21: return filtered color of the samples $\mathbf{c}''$
```

$$\begin{aligned} w_{ij} = &\exp[-\frac{1}{2\sigma^2_{\mathbf{p}}}\sum_{1\le k\le 2}(\bar{\mathbf{p}}_{i,k} - \bar{\mathbf{p}}_{j,k})^2]\times \\ &\exp[-\frac{1}{2\sigma^2_{\mathbf{c}}}\sum_{1\le k\le 3}\alpha_k(\bar{\mathbf{c}}_{i,k} - \bar{\mathbf{c}}_{j,k})^2]\times \\ &\exp[-\frac{1}{2\sigma^2_{\mathbf{f}}}\sum_{1\le k\le m}\beta_k(\bar{\mathbf{f}}_{i,k} - \bar{\mathbf{f}}_{j,k})^2], \end{aligned} \tag{17}$$

where w_{ij} is the contribution (or weight) of j^{th} sample to the i^{th} sample during filtering. Because of the way we select the samples in neighborhood $\mathcal{N}$ randomly using a Gaussian distribution with standard deviation $\sigma_{\mathbf{p}}$ (where $\sigma_{\mathbf{p}} = b/4$), the first term of this expression is dropped and becomes:

$$\begin{aligned} w_{ij} = &\exp[-\frac{1}{2\sigma^2_{\mathbf{c}}}\sum_{1\le k\le 3}\alpha_k(\bar{\mathbf{c}}_{i,k} - \bar{\mathbf{c}}_{j,k})^2]\times \\ &\exp[-\frac{1}{2\sigma^2_{\mathbf{f}}}\sum_{1\le k\le m}\beta_k(\bar{\mathbf{f}}_{i,k} - \bar{\mathbf{f}}_{j,k})^2]. \end{aligned} \tag{18}$$

As mentioned in Sec. 4.5.1 of the original paper [Sen and Darabi 2011b], the variances of the Gaussians for both the color and the feature are set to the same value:

$$\sigma^2_{\mathbf{c}} = \sigma^2_{\mathbf{f}} = \frac{\sigma^2}{(1 - W^{\mathbf{r}}_{\mathbf{c}})^2}, \tag{19}$$

where the $W^{\mathbf{r}}_{\mathbf{c}}$ term is calculated by Eq. 11. We divide these variances by $(1 - W^{\mathbf{r}}_{\mathbf{c}})^2$ because, in the end, we only care about the sample color and want a large filter wherever the color depends a lot on the random parameters (i.e., is very noisy). This term adjusts

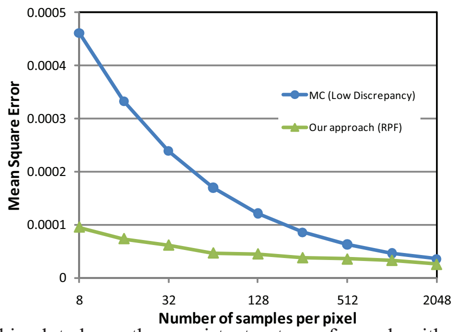


Fig. 4. This plot shows the consistent nature of our algorithm by showing that it has lower MSE than traditional Monte Carlo for a subset of the ROBOTS scene all the way to 2,048 samples/pixel.

the size of the Gaussian based on the overall noise level, making it large when needed. We could have rolled the $\sigma_{\mathbf{c}}^2$ and $\sigma_{\mathbf{f}}^2$ terms into the α_k and β_k coefficients in Eq. 17, but because the $\sigma_{\mathbf{c}}^2$ and $\sigma_{\mathbf{f}}^2$ terms depend on all three color channels (because of the $W_{\mathbf{c}}^{\mathbf{r}}$ term) as opposed to α_k (whose $W_{\mathbf{c},k}^{\mathbf{r}}$ term varies per color channel), it was easier to separate them. This way, the $\sigma_{\mathbf{c}}^2$ and $\sigma_{\mathbf{f}}^2$ terms modulate the overall size of the Gaussian while α_k and β_k adjust it further based on dependencies with the random parameters. The σ^2 parameter was selected by experimenting with scenes at 8 samples/pixel, but is scaled inversely by the number of samples per pixel s (so as s grows, σ^2 gets smaller): $\sigma^2 = 8\sigma_8^2/s$. For noisy scenes (e.g., indoor path-tracing scenes) we used $\sigma_8^2 = 0.02$, while for all others we set $\sigma_8^2 = 0.002$.

Because we divide the constant σ_8^2 by the number of samples when computing the filter's variance σ^2, our approach is a biased but consistent estimator, meaning that the estimator converges to the value of the integral as the number of samples per pixel s goes to infinity. As $s \to \infty$, the expression above results in $\sigma_{\mathbf{c}}^2 = \sigma_{\mathbf{f}}^2 = 0$, which produces a weight $w_{ij} = 1$ only when $i = j$ and zero elsewhere. Therefore, the color of the samples are not filtered at all, so our approach converges to standard Monte Carlo, which is a consistent estimator. This is shown in Fig. 4, which shows that our approach has a lower mean-squared error (MSE) than conventional MC all the way to 2,048 samples/pixel.

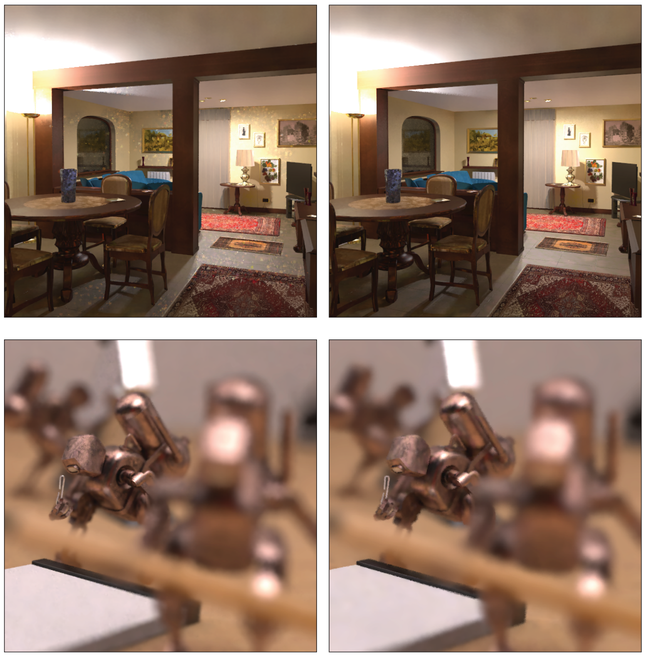
Once we have obtained our filter weights w_{ij} using the weighted bilateral filter of Eq. 18, we use these weights to blend in the color contributions from these samples:

$$\mathbf{c}''_{i,k} = \frac{\sum_{j\in\mathcal{N}} w_{ij}\mathbf{c}'_{j,k}}{\sum_{j\in\mathcal{N}} w_{ij}}, \tag{20}$$

where the denominator is never zero because at least $w_{ii} = 1$ (a sample fully contributes to itself). Note that this process filters the colors of individual samples (not pixels), and we perform this separately for every pixel in the image, since statistics change from pixel to pixel. After all samples in the image have been filtered, we repeat the process with a new iteration as shown in Alg. 1.

6.1 Handling sample spikes in high dynamic range

In our original paper we talk about how we remove HDR spikes from our renderings using a simple classification based on the standard deviation of the pixel average value. In Fig. 5 we show what happens when we do not perform this step. This mostly affects indoor path-traced scenes such as the one shown in the top row, because in these kinds of scenes the light sources are small but can still be accidentally hit in an early bounce in the path-tracing process, resulting in very bright samples. However, not all scenes required this step, such as the bottom ROBOTS scene which was unaffected by the application of this technique.



without handling statistical outliers   with the algorithm of Sec. 6.1

Fig. 5. This figure shows how statistical outliers can still cause problems even if we filter them because their colors are orders of magnitude greater than those around them. The top row shows a path-traced scene where this problem is visible, and our solution on the right. However, this only affected a small subset of the scenes we tested (specifically the path-traced scenes with small, bright light sources). On the bottom row, we show another path-traced scene that does not have this problem.

7. ADDITIONAL RESULTS

In this section, we present results from our algorithm to supplement those in the original paper [Sen and Darabi 2011b]. We begin by doing a breakdown in Fig. 6 to help explain how our algorithm achieves the result in Fig. 1. First, we show a rendering of the same scene using only direct lighting in Fig. 6a to highlight the illumination contribution from path tracing. When compared to the result in Fig. 1, we can see that many regions of the image are completely dark when using only direct lighting because these regions are totally occluded from the sky light source. This means that the illumination in these regions that is visible in Fig. 1 is due exclusively to path-tracing. Our algorithm is able to denoise these regions by examining the relationship between the sample values and the random parameters used to compute the bounces of each path.

Path-tracing is notoriously noisy, and when we examine the input Monte Carlo samples to our algorithm (shown Fig. 1 top image), we see that much of the detail in the textures in the scene is completely gone. This is more evident if we multiply the color channel by 1000, shown in Fig. 6b. Many of the pixels remain black, which indicates that these pixels have no useful color information. So how does our algorithm use an image-space filter to recover this detail and produce the result in Fig. 6h? The key is our cross-bilateral filter that examines other sample features, such as world position (Fig. 6c), surface normal (Fig. 6d), and texture value (Fig. 6e), each weighted depending on their amount of functional dependency on the random parameters. In this case, the samples' colors are extremely noisy because the path tracing produces a lot of noise while computing the global illumination. Our algorithm detects the connection between the sample color and the random parameters of the path tracer, shown in Fig. 6f, and essentially ignores the color when bilateral filtering. The texture value, on the other hand, is found to have little dependence on the random parameters (Fig. 6g) so it is weighted heavily by the cross-bilateral filter.

Therefore, to filter a sample we ignore its color but pay close attention to its texture value. When blending in values from around the filter kernel, we only blend together samples with similar texture values. So if our sample hits a dark part of the texture, we blend in samples from other parts of the texture that are also dark. Essentially, our filter combines many noisy samples of dark texture together to approximate a noise-free dark texture. Of course, some blurring of the texture detail occurs, caused by the fact that we use a large filter kernel to help denoise this very noisy image (ideally, we would prefer to use a small filter to help preserve detail). We call these competing constraints the "dueling filter" problem (see Sec. 6.5 of the main paper). The study of ways to address the dueling filtering problem is an interesting subject of future work.

We follow this analysis of the SAN MIGUEL result with additional results that complement those in our original paper [Sen and Darabi 2011b]. We begin with simple test cases (many of which we used to debug and test our algorithm) shown in Figs. 7 - 12, followed by more complex and interesting scenes in Figs. 13 - 18. These figures all used the same settings for the RPF algorithm, highlighting the fact that the proposed method is reasonably robust for use in production environments.

8. CONCLUSION

In this technical report, we have described the details necessary to implement our random parameter filtering algorithm, which removes the noise in Monte Carlo rendering by estimating the functional dependency between sample values and the random parameters used to compute them. This implementation is only one exploration of the proposed technique, and we are hopeful that others will build on this work and improve the quality of the results.

ACKNOWLEDGMENTS

Lei Xiao conducted the experiments to produce several images for this technical report and implemented the RPF algorithm using the pseudocode presented to verify its accuracy. He, along with Nima Khademi Kalantari, were part of valuable discussions that improved the description of the algorithm presented here. Frances Strong helped proofread drafts of the final manuscript. We also thank Toshiya Hachisuka, Ryan Overbeck, and Kevin Egan for help with the implementations of their respective algorithms. Finally, we gratefully acknowledge the sources of the scenes shown in this technical report (in order of appearance): SAN MIGUEL – Guillermo M. Leal Llaguno (PBRT2 book), CHESS – Wojciech Jarosz, PERSIAN ROOM – Luca Cugia, ROBOTS – Jesper Lloyd, BUDDHA – Stanford 3D scanning repository, GLOSSY BALLS – PBRT2 book, DRAGONS – Stanford 3D scanning repository (PBRT2 book), POOLBALL – Toshiya Hachisuka, TOASTERS – Andrew Kensler, CAR – Turbosquid user graphicdoom (car model), Wikipedia user Jongleur100 (wall texture), Kevin Egan (scene). This work was funded by National Science Foundation CAREER award IIS-0845396.

REFERENCES

COVER, T. AND THOMAS, J. 2006. *Elements of Information Theory*, Second ed. John Wiley & Sons, Hoboken, New Jersey.

DAMMERTZ, H., SEWTZ, D., HANIKA, J., AND LENSCH, H. P. 2010. Edge-avoiding À-trous wavelet transform for fast global illumination filtering. In *Proceedings of High Performance Graphics 2010*. 67–75.

DEERING, M., WINNER, S., SCHEDIWY, B., DUFFY, C., AND HUNT, N. 1988. The triangle processor and normal vector shader: a VLSI system for high performance graphics. In *ACM SIGGRAPH '88*. ACM, New York, NY, USA, 21–30.

EGAN, K., TSENG, Y.-T., HOLZSCHUCH, N., DURAND, F., AND RAMAMOORTHI, R. 2009. Frequency analysis and sheared reconstruction for rendering motion blur. *ACM Trans. Graph. 28,* 3, 1–13.

EISEMANN, E. AND DURAND, F. 2004. Flash photography enhancement via intrinsic relighting. *ACM Trans. Graph. 23*, 673–678.

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 Trans. Graph. 27,* 3, 1–10.

HASTIE, T., TIBSHIRANI, R., AND FRIEDMAN, J. H. 2001. *The elements of statistical learning: data mining, inference, and prediction*. New York: Springer-Verlag.

JENSEN, H. W. AND CHRISTENSEN, N. J. 1995. Optimizing path tracing using noise reduction filters. In *Winter School of Computer Graphics (WSCG) 1995*. 134–142.

KAJIYA, J. T. 1986. The rendering equation. *SIGGRAPH Comput. Graph. 20,* 4, 143–150.

LEE, M. AND REDNER, R. 1990. A note on the use of nonlinear filtering in computer graphics. *IEEE Computer Graphics and Applications 10,* 3 (May), 23–29.

LUXRENDER. 2011. http://www.luxrender.net/.

MAHALANOBIS, P. C. 1936. On the generalized distance in statistics. *Proceedings of the National Institute of Sciences of India 2,* 1, 49–55.

MCCOOL, M. D. 1999. Anisotropic diffusion for Monte Carlo noise reduction. *ACM Trans. Graph. 18,* 2, 171–194.

OVERBECK, R. S., DONNER, C., AND RAMAMOORTHI, R. 2009. Adaptive wavelet rendering. *ACM Trans. Graph. 28,* 5, 1–12.

PENG, H. 2007. Matlab package for mutual information computation. http://www.mathworks.com/matlabcentral/fileexchange/14888.

PETSCHNIGG, G., SZELISKI, R., AGRAWALA, M., COHEN, M., HOPPE, H., AND TOYAMA, K. 2004. Digital photography with flash and no-flash image pairs. *ACM Trans. Graph. 23*, 664–672.

PHARR, M. AND HUMPHREYS, G. 2010. *Physically Based Rendering: From Theory to Implementation*, Second ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

ROBERT, C. P. AND CASELLA, G. 2005. *Monte Carlo Statistical Methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

RUSHMEIER, H. E. AND WARD, G. J. 1994. Energy preserving non-linear filters. In *ACM SIGGRAPH '94*. New York, NY, USA, 131–138.

SAITO, T. AND TAKAHASHI, T. 1990. Comprehensible rendering of 3-d shapes. In *ACM SIGGRAPH '90*. ACM, New York, NY, USA, 197–206.

SEN, P. AND DARABI, S. 2010. Compressive estimation for signal integration in rendering. *Computer Graphics Forum 29,* 4, 1355 1363.

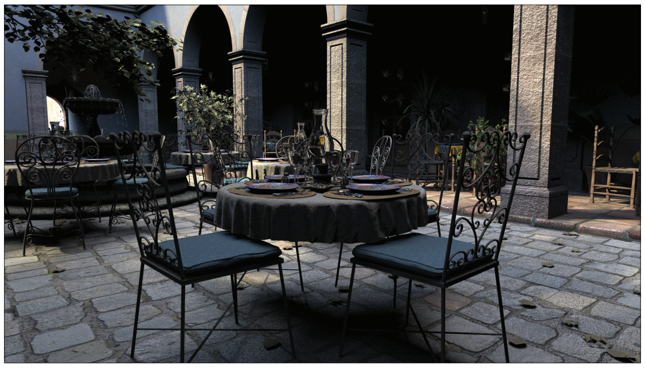
SEN, P. AND DARABI, S. 2011a. Compressive rendering: A rendering application of compressed sensing. *IEEE Transactions on Visualization and Computer Graphics 17*, 487–499.

SEN, P. AND DARABI, S. 2011b. On filtering the noise from the random parameters in Monte Carlo rendering (submitted). *ACM Trans. Graph. XX,* Y.

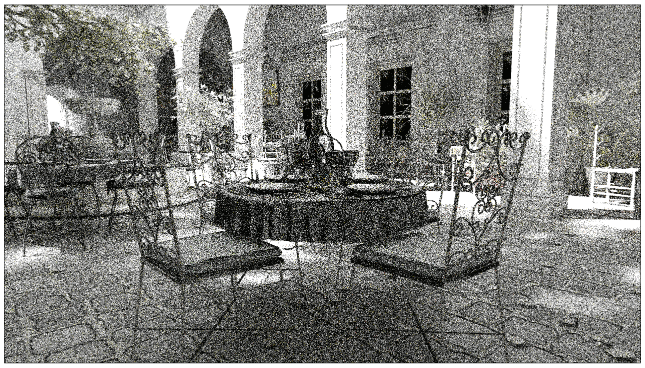
SOLER, C., SUBR, K., DURAND, F., HOLZSCHUCH, N., AND SILLION, F. 2009. Fourier depth of field. *ACM Trans. Graph. 28,* 2, 1–12.

TOMASI, C. AND MANDUCHI, R. 1998. Bilateral filtering for gray and color images. In *ICCV 98*. IEEE, 839.

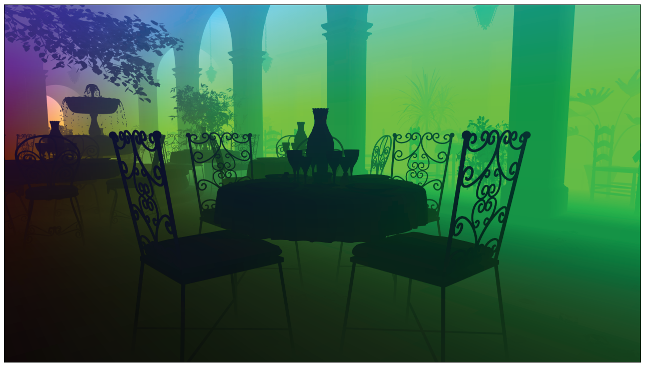
XU, R. AND PATTANAIK, S. N. 2005. A novel Monte Carlo noise reduction operator. *IEEE Computer Graphics and Applications 25*, 31–35.



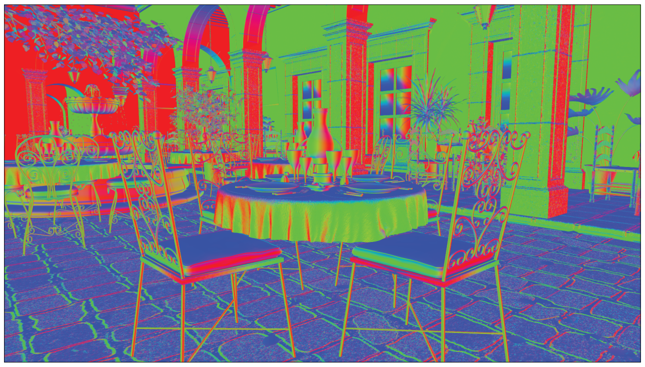
(a) Direct illumination only, 64 samples/pixel (for comparison)



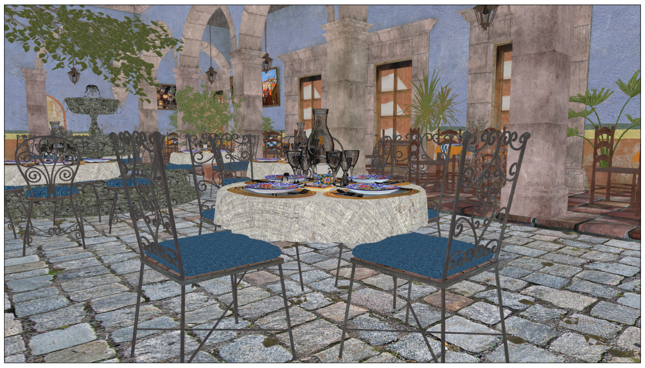
(b) Colors of MC input samples $\times 1000$



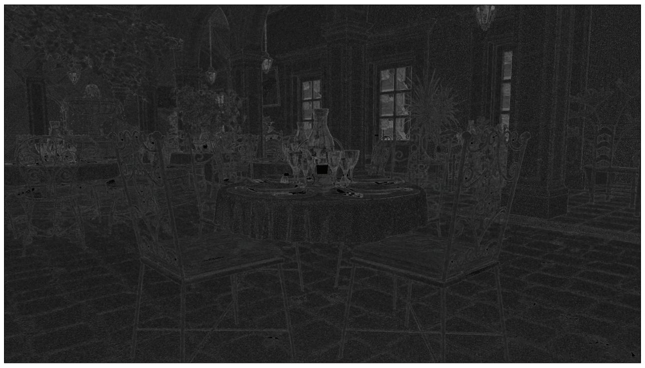
(c) World position



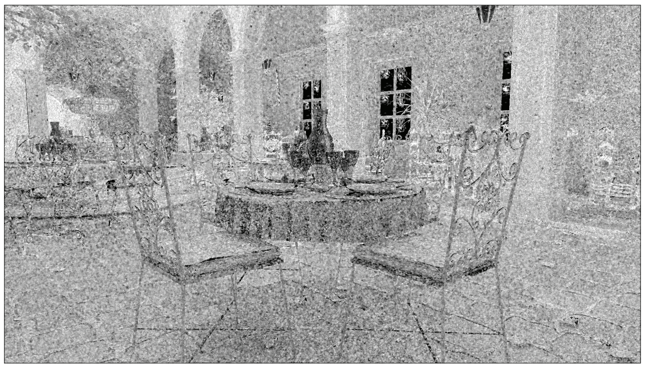
(d) Surface normals



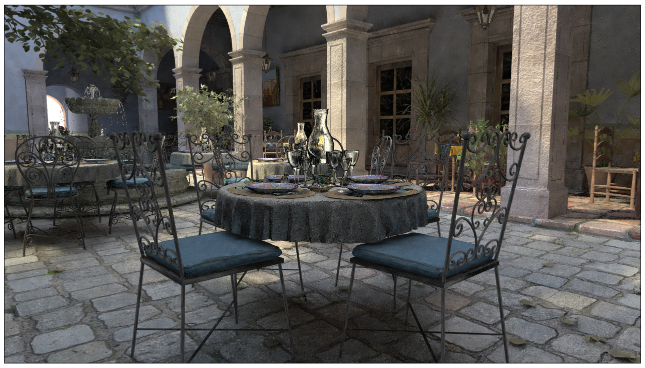
(e) Texture value



(f) Dependency on texture value

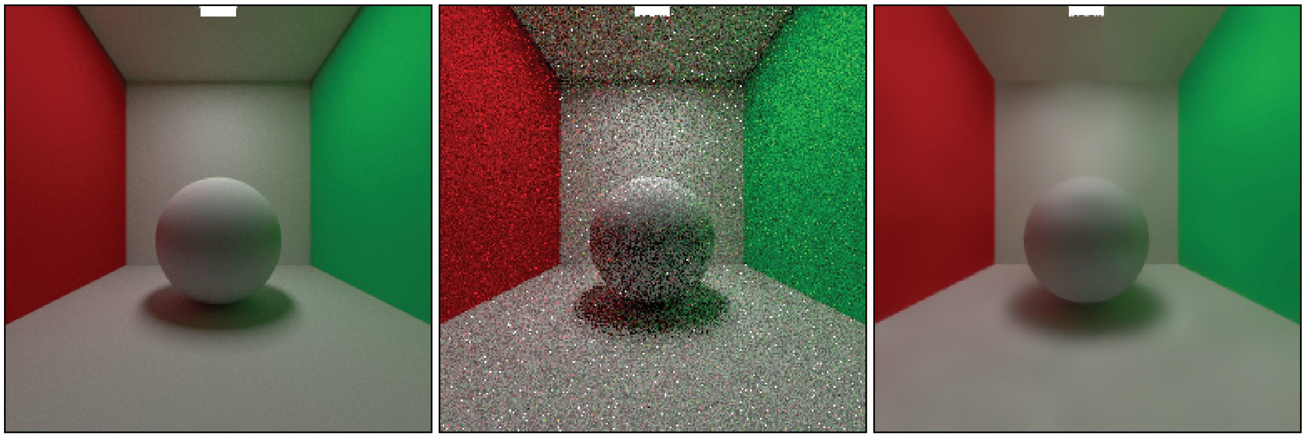


(g) Dependency on color



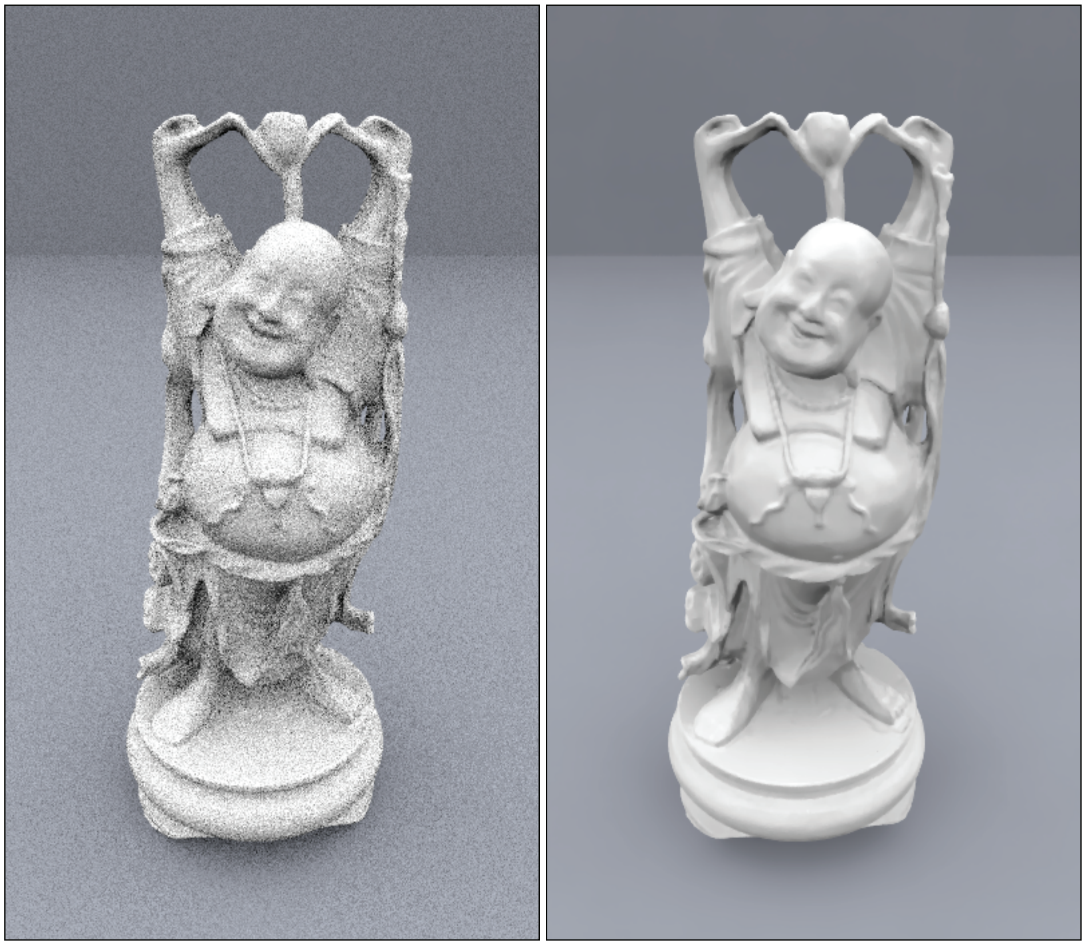
(h) Output of our approach

Fig. 6. This figure breaks down some of the intermediate values of our algorithm to give some insight into how we obtain the result shown in the middle image of Fig. 1. The values in sub-figures (b-e) are computed by averaging the appropriate sample value over each pixel.



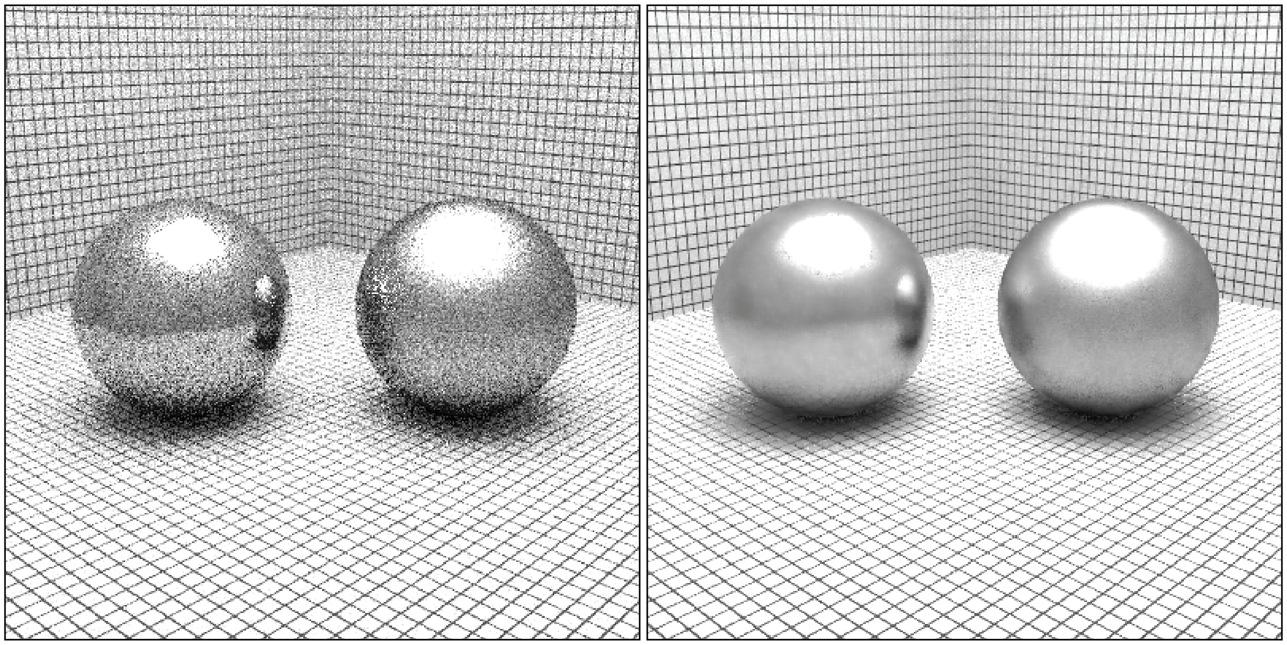
(a) Reference Monte Carlo (8,192 spp) **(b)** Input Monte Carlo (8 spp) **(c)** Our approach (RPF)

Fig. 7. This figure shows the CORNELL BOX scene with a diffuse ball and demonstrates our ability to filter global illumination effects, such as the color bleeding from the colored walls that is visible in the ball's shadow.



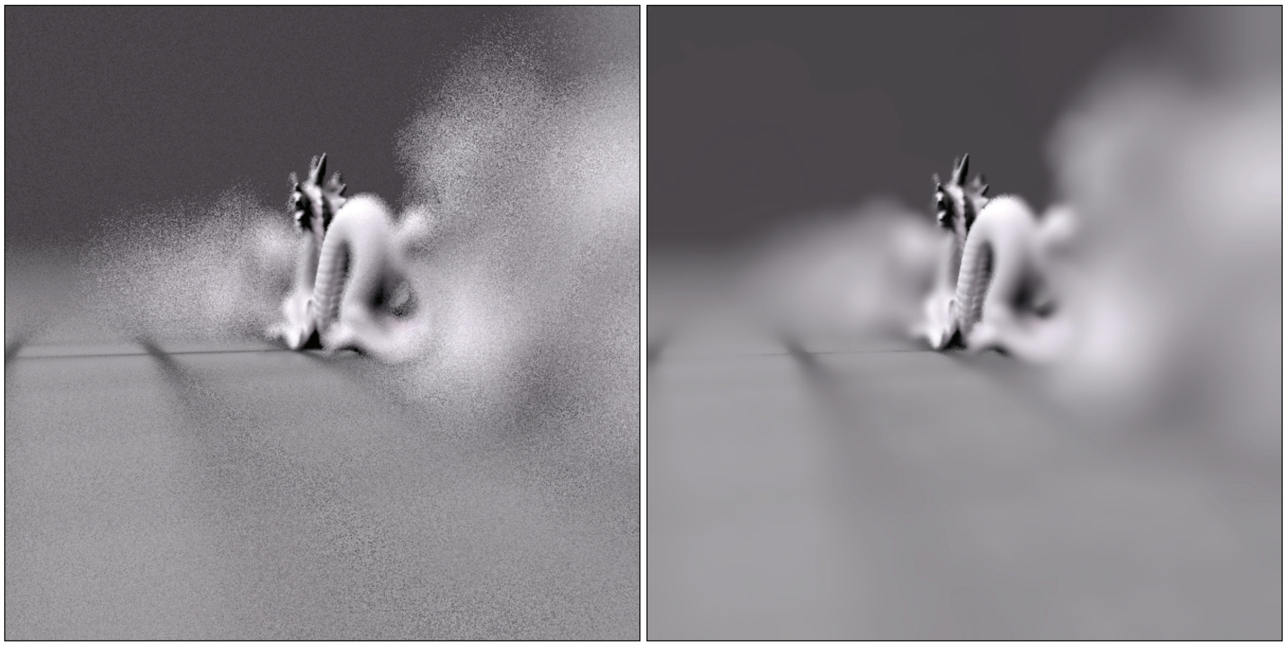
(a) Input Monte Carlo (8 spp) **(b)** Our approach (RPF)

Fig. 8. Here, the simple BUDDHA model is illuminated by a disk area light source. Our algorithm is able to remove the random noise from the area light source but keeps the geometrical detail in the Buddha statue.



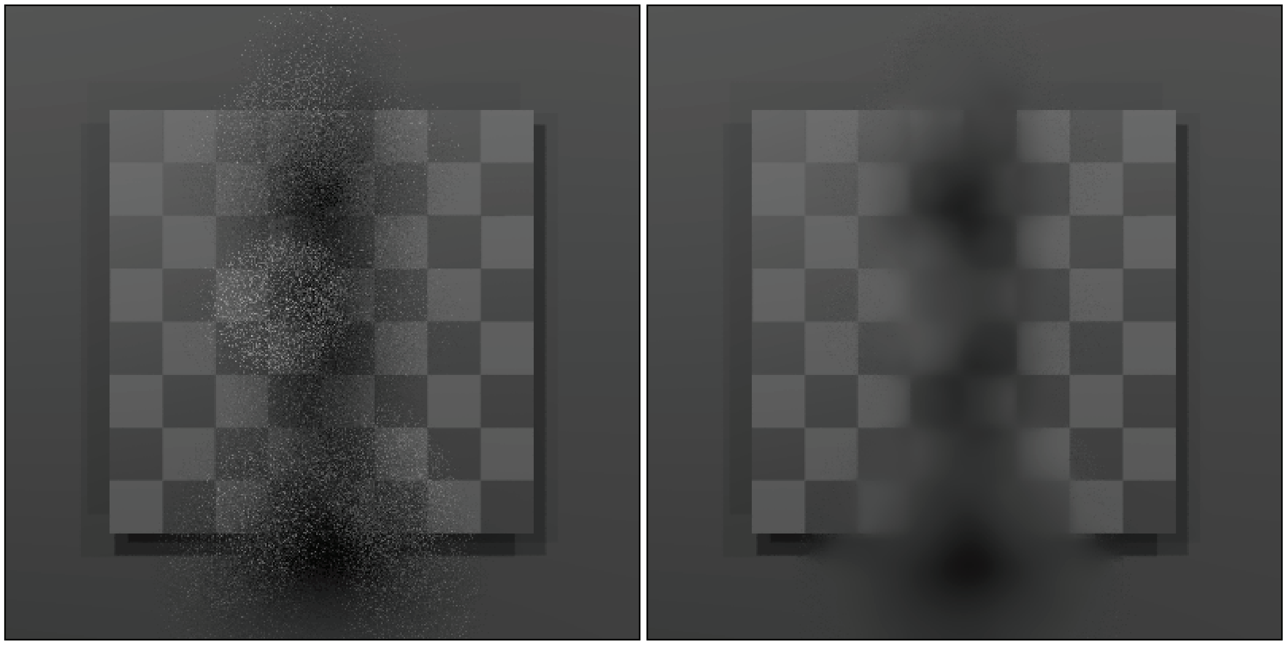
(a) Input Monte Carlo (8 spp) **(b)** Our approach (RPF)

Fig. 9. These balls have a glossy reflection, demonstrating that our algorithm is able to perform reasonably with interesting BRDFs.



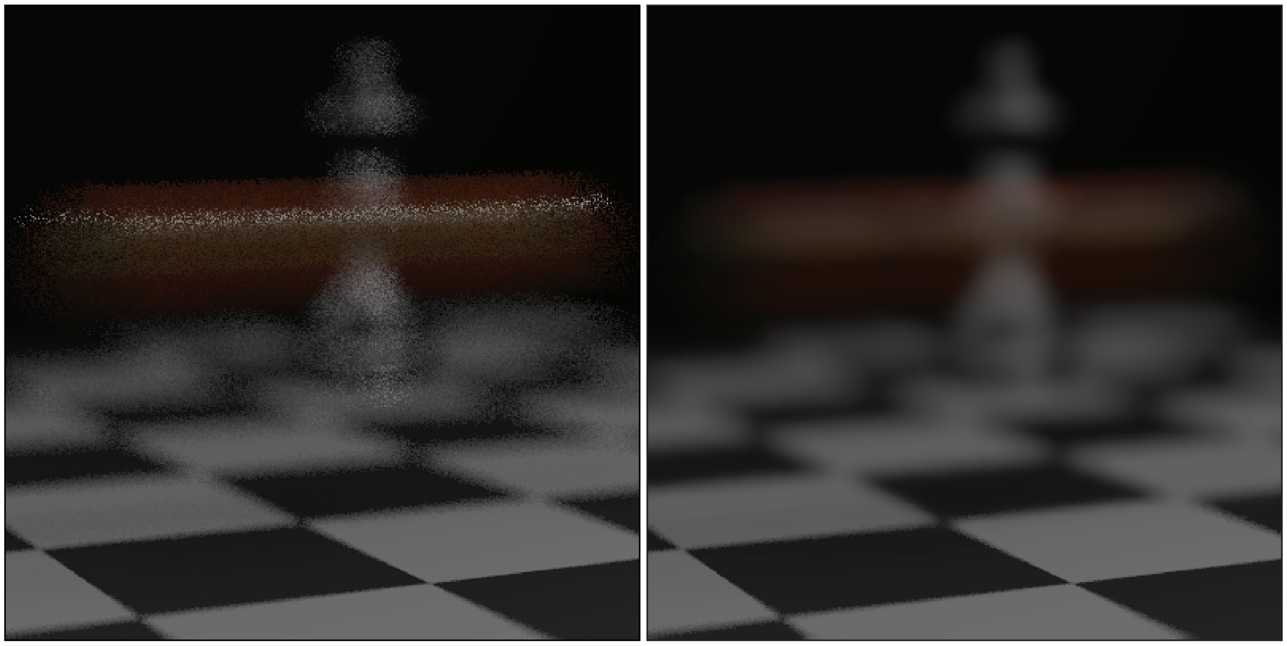
(a) Input Monte Carlo (8 spp) **(b)** Our approach (RPF)

Fig. 10. This DRAGONS scene features a fairly extreme depth-of-field effect by using a large aperture. Our algorithm produces a smooth result in the out-of-focus regions but is able to preserve the in-focus detail.



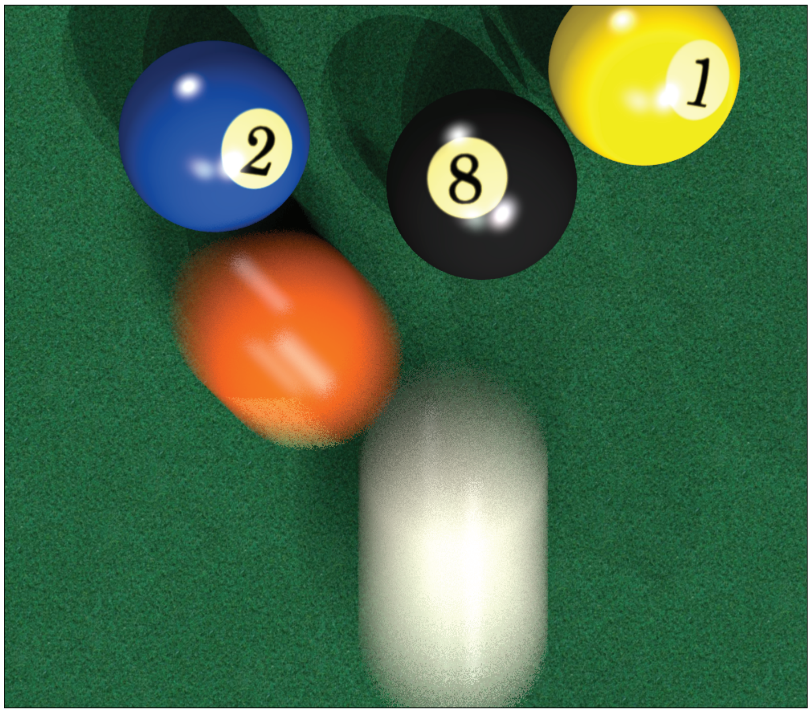
(a) Input Monte Carlo (8 spp) **(b)** Our approach (RPF)

Fig. 11. This scene has some chess pieces extremely out of focus in front of a checkered background. We see that our algorithm can blur the noise away in the out-of-focus regions while keeping the in-focus features sharp, even if they lie in the circle of confusion of the out-of-focus objects.

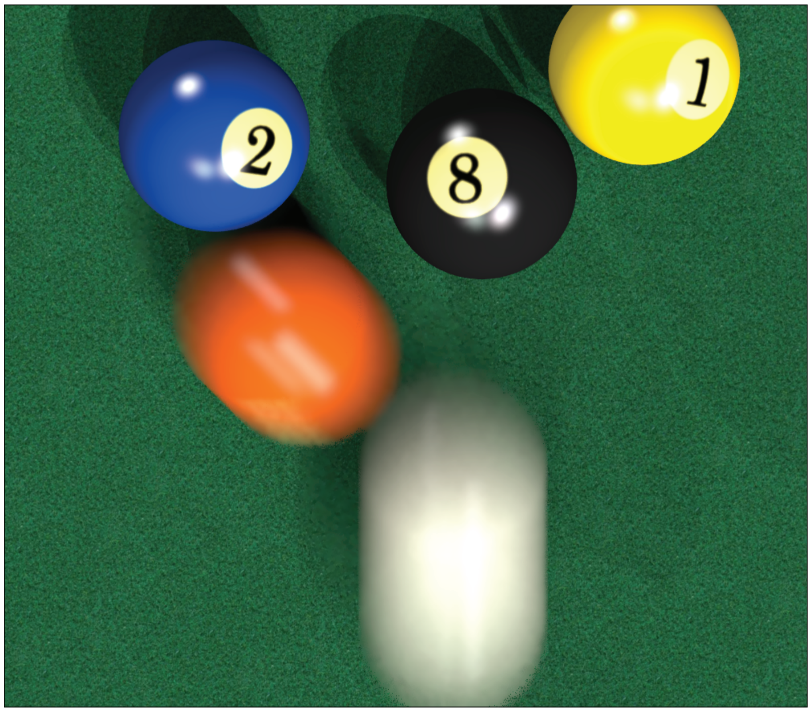


(a) Input Monte Carlo (8spp) **(b)** Our approach (RPF)

Fig. 12. This scene combines a depth-of-field effect with motion blur to demonstrate that we can denoise both effects simultaneously.

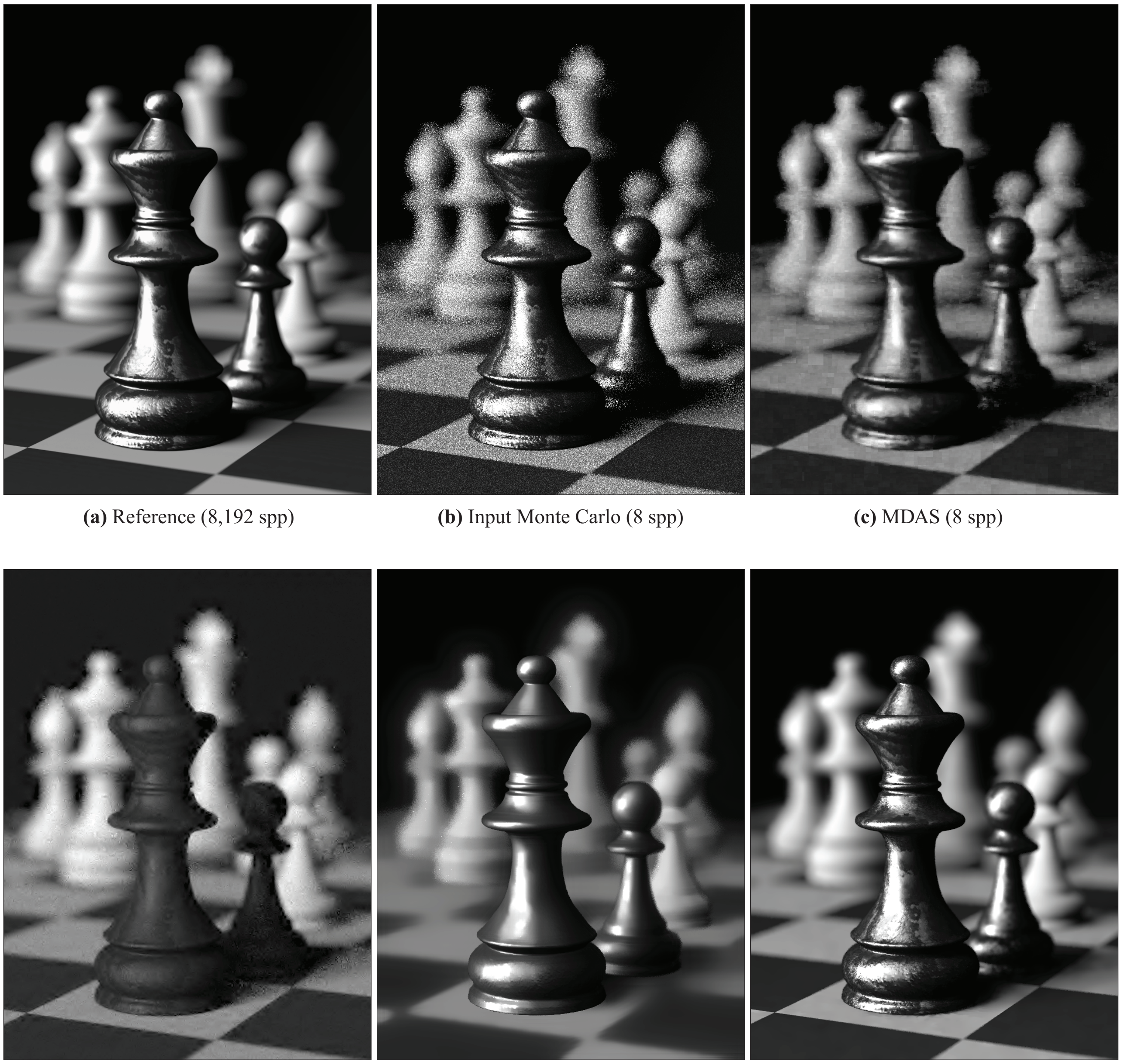


(a) Input Monte Carlo (8spp)



(b) Our approach (RPF)

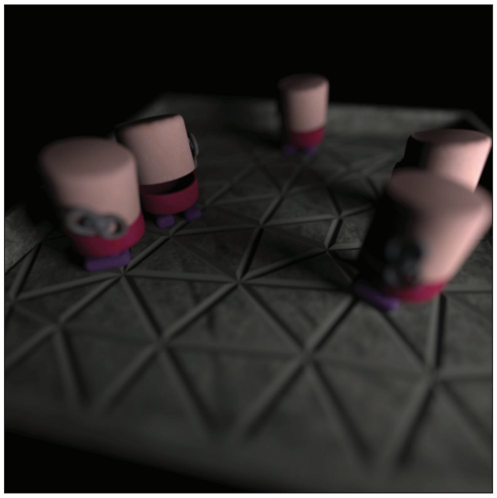
Fig. 13. This figure shows how our approach can remove the motion-blur noise in the POOLBALL scene.

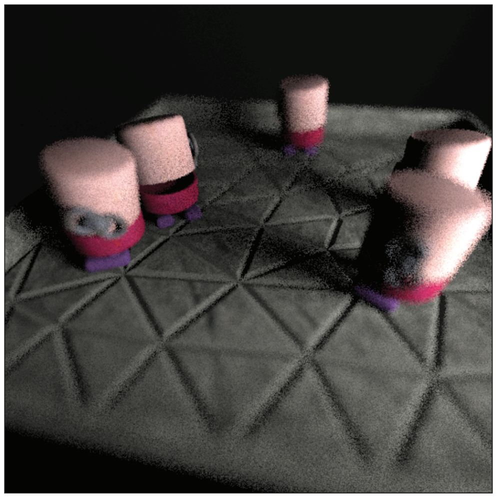


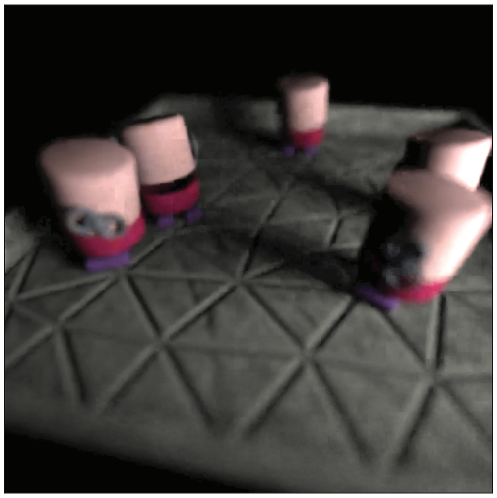
(a) Reference (8,192 spp) **(b)** Input Monte Carlo (8 spp) **(c)** MDAS (8 spp)

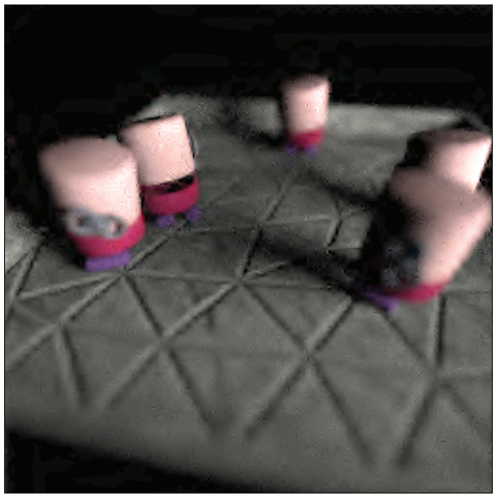
(d) AWR (8 spp) **(e)** À-Trous **(f)** Our approach (RPF)

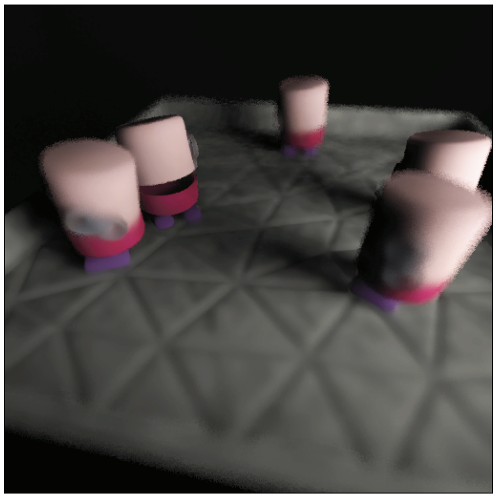
Fig. 14. CHESS scene with depth of field and an area light source. This is a larger version of Fig. 9 in the original paper. The results of multidimensional adaptive sampling (MDAS) [Hachisuka et al. 2008] in **(c)** and adaptive wavelet rendering (AWR) [Overbeck et al. 2009] in **(d)** use the implementations of the respective authors. The À-Trous method was proposed by Dammertz et al. [2010].


(a) Reference Monte Carlo (8,192 spp)


(b) Input Monte Carlo (8 spp)


(c) MDAS (8 spp)


(d) AWR (8 spp)


(e) À-Trous

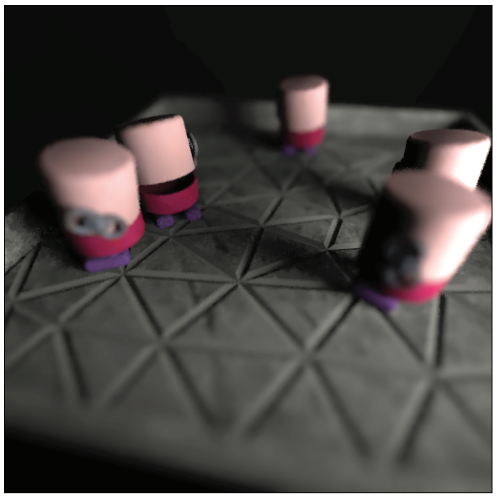
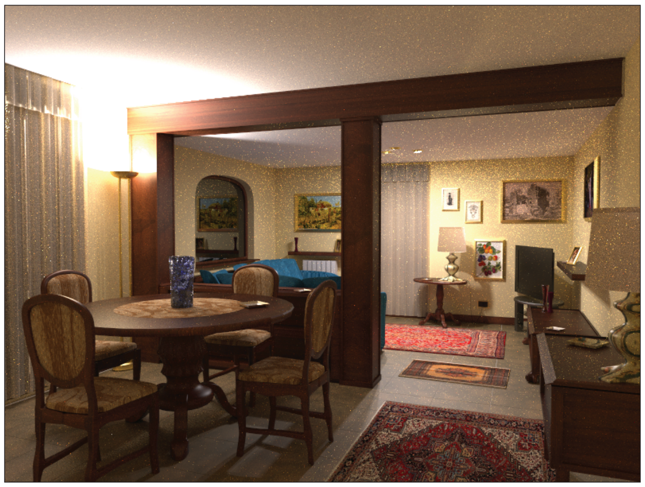
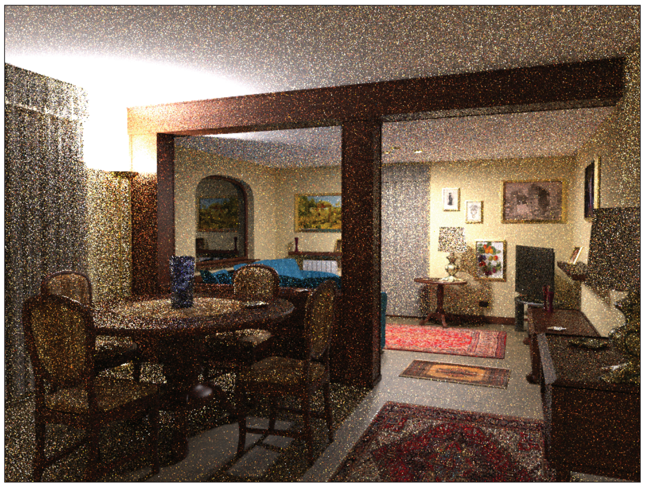

(f) Our approach (RPF)

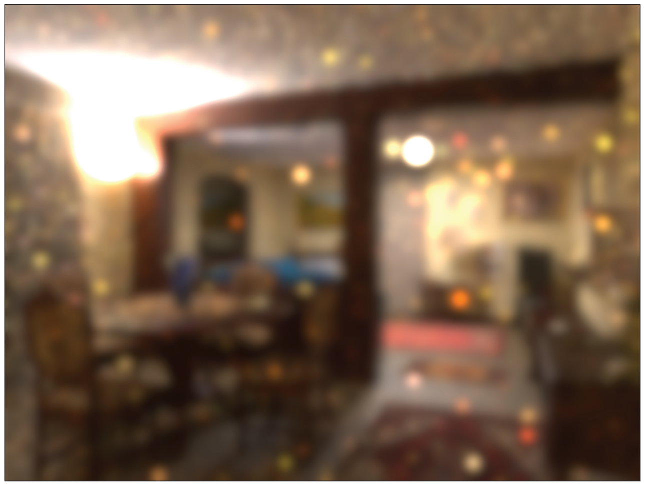
Fig. 15. This TOASTERS scene has an area light source and depth-of-field effect. It is a larger version of Fig. 10 in the original paper.



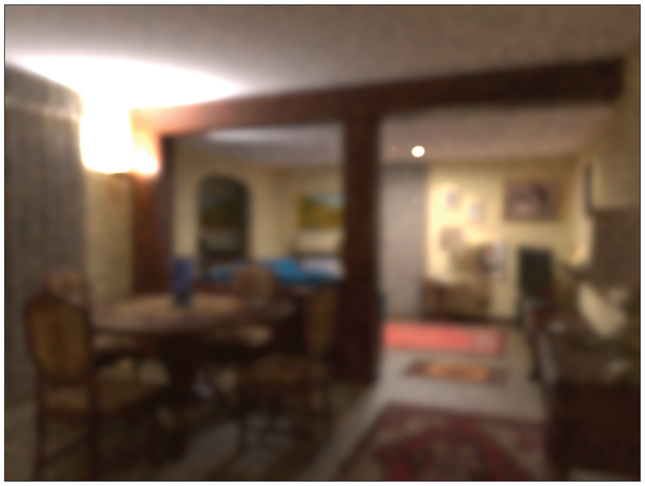
(a) Reference Monte Carlo (8,192spp)



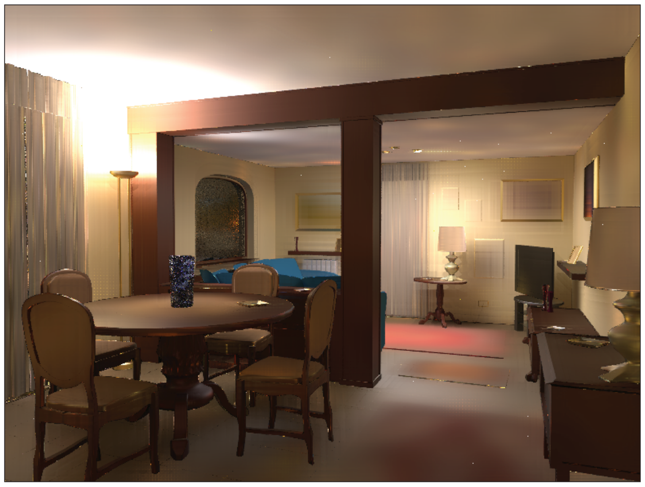
(b) Input Monte Carlo (8spp)



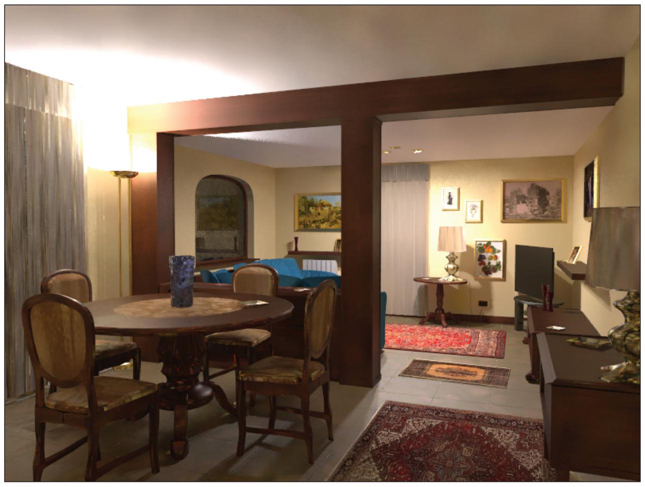
(c) Standard Gaussian blur



(d) Monte Carlo denoising

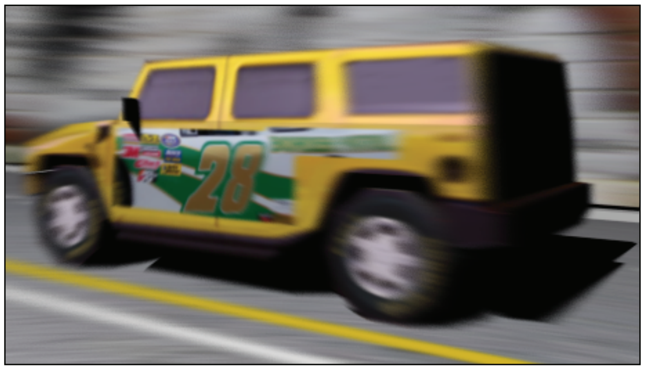


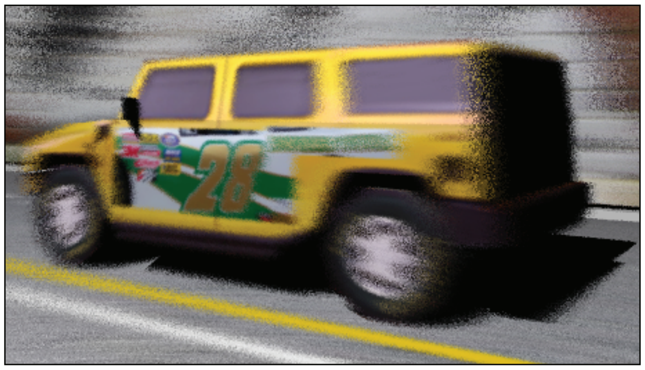
(e) À-Trous

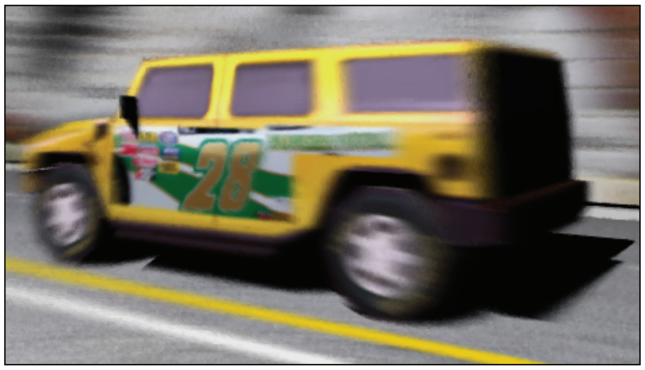


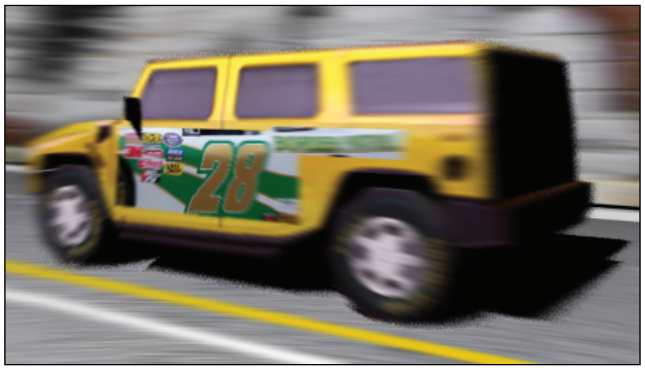
(f) Our approach (RPF)

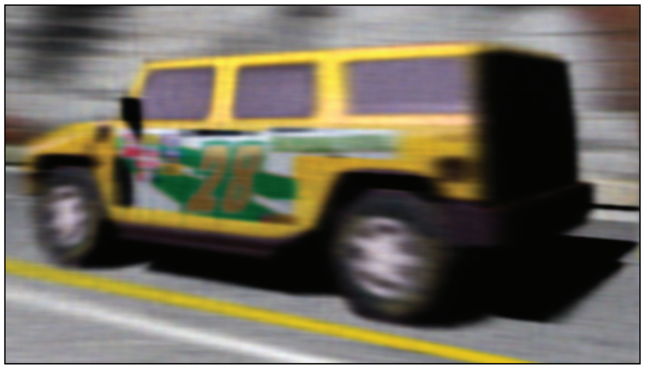
Fig. 16. Larger versions of the path-traced PERSIAN ROOM scene, shown in our original paper in Fig. 11. The Monte Carlo denoising method was proposed by Xu and Pattanaik [2005].


(a) Reference Monte Carlo (256 spp)


(b) Input Monte Carlo (4 spp)


(c) MDAS (4 spp)


(d) Sheared-filter motion blur (SFMB) (4 spp)


(e) Compressive integration

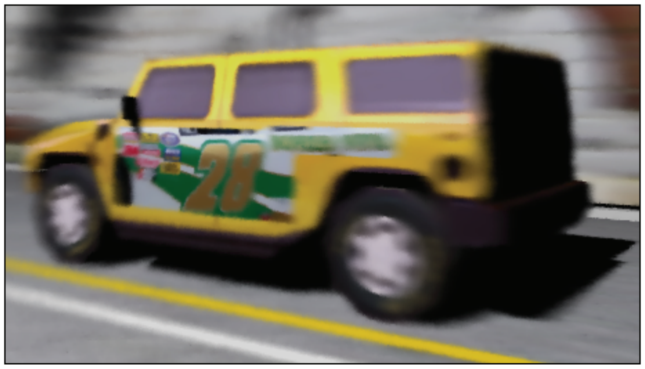
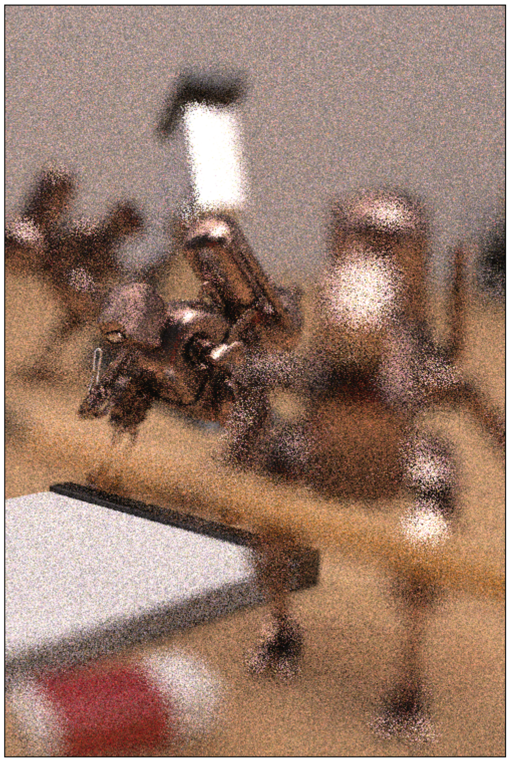
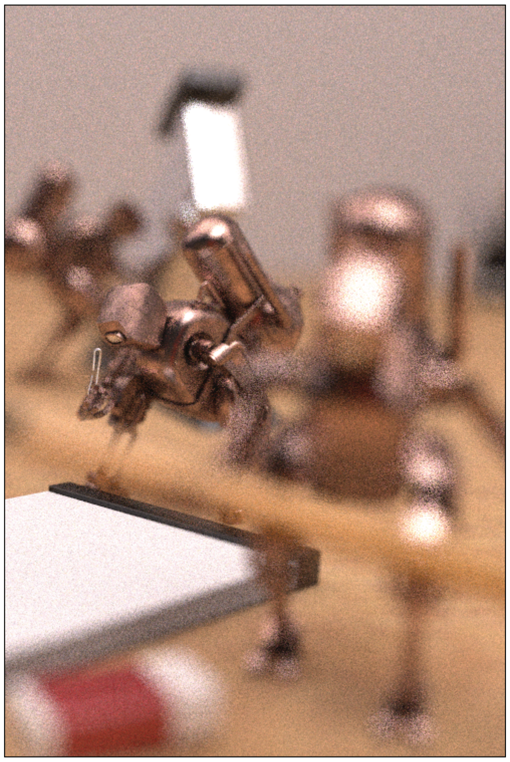

(f) Our approach (RPF)

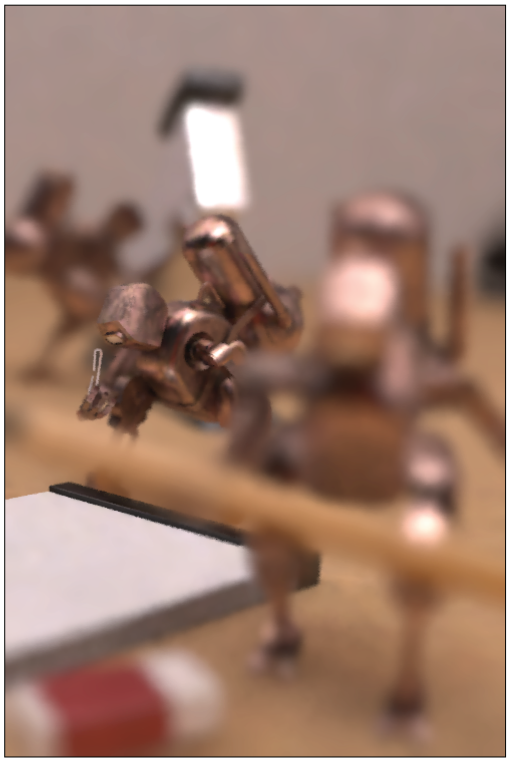
Fig. 17. Here we show the full images for the motion-blurred CAR scene whose insets are shown in Fig. 12 in the original paper. Note the artifacts of the sheared-filter motion blur (SFMB) approach of Egan et al. [2009], in particular around the edges of the shadow under the car, and compare their result to ours. Our algorithm is competitive even though their algorithm is specifically designed to handle motion blur and ours is a general technique. The results from MDAS, SFMB, and compressive integration [Sen and Darabi 2010; 2011a] all use the implementations provided by the respective authors.



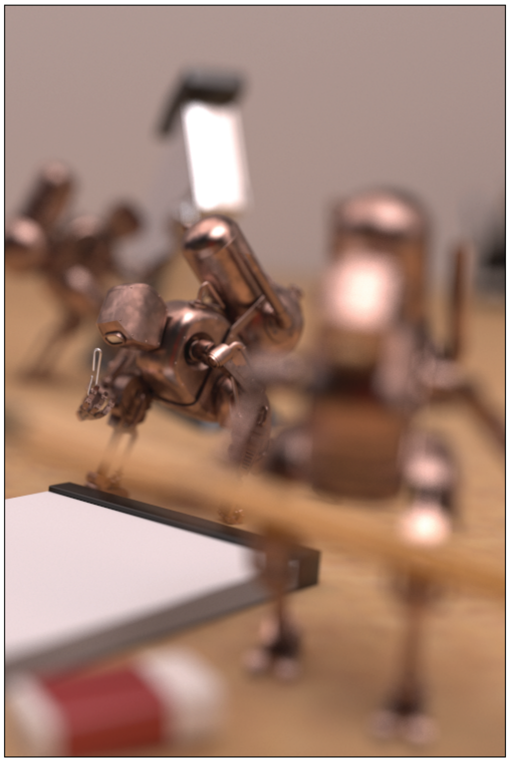
(a) Input Monte Carlo (8 spp)



(b) Equal time Monte Carlo (64 spp)



(c) Our approach (RPF)



(d) Reference Monte Carlo (8,192 spp)

Fig. 18. This figure shows larger images for the ROBOTS scene with depth of field and path-tracing, shown in Fig. 17 in the paper.