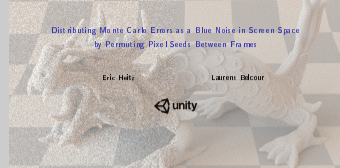


Distributing Monte Carlo Errors as a Blue Noise in Screen Space by Permuting Pixel Seeds Between Frames

Eric Heitz

Laurent Belcour

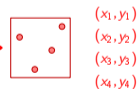
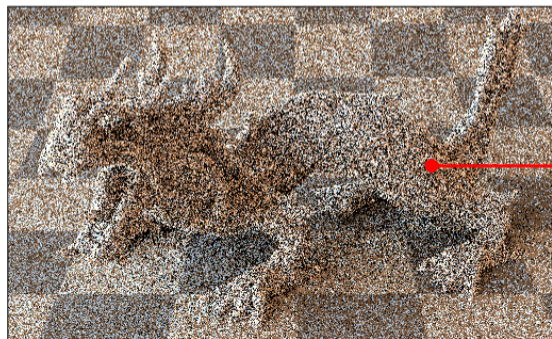


Introduction

In each pixel, we use a sequence of random numbers for Monte Carlo integration:

$$\text{pixel value} = \frac{1}{n} \sum_{i=1}^n f(x_i, y_i)$$

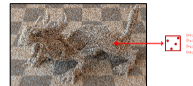
f = light transport



In each pixel we use a sequence of random numbers for Monte Carlo integration:

$$\text{pixel value} = \frac{1}{n} \sum_{i=1}^n f(x_i, y_i)$$

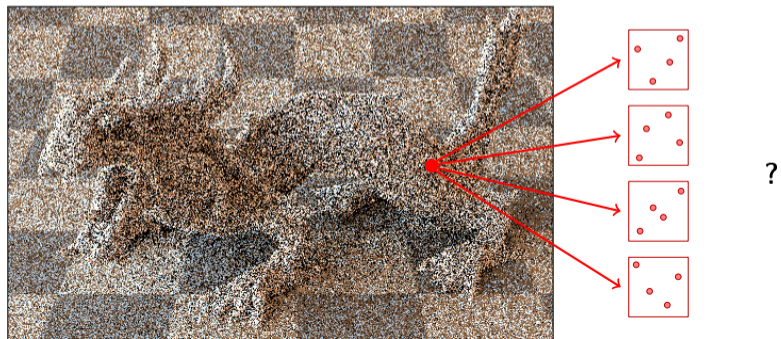
f = light transport



Our context is a classic Monte Carlo forward path tracer where each pixel is estimated using a sequence of random numbers.

Introduction

Is there a clever way to assign a sequence to each pixel?



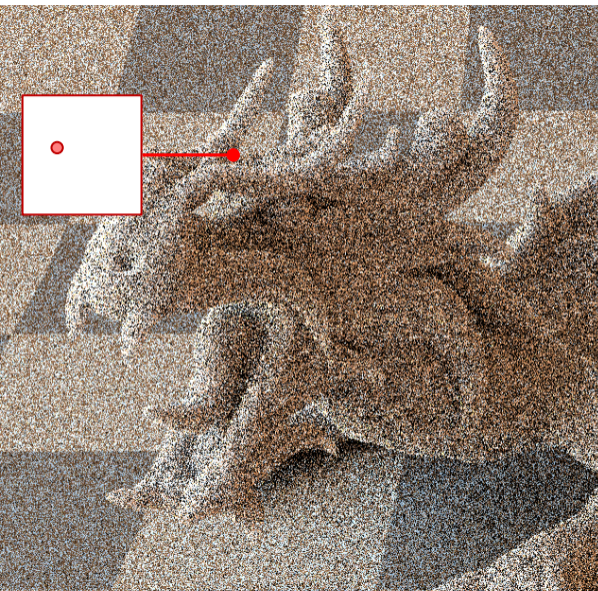
Is there a clever way to assign a sequence to each pixel?



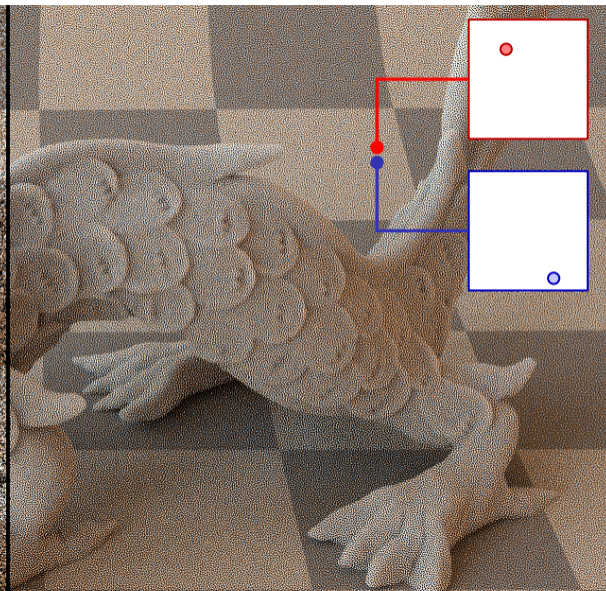
Many papers focus on the nature of the sequence itself. In this paper, we are interested in another question: the choice of the sequence for each pixel.

In a classic path tracer, a sequence is chosen randomly for each pixel. It is the random choice of the sequence that randomize the Monte Carlo errors in the image and produces noise. Can we do better than that?

Random



Optimized difference of neighboring sequences



Same number of samples per pixel, same rendering time.

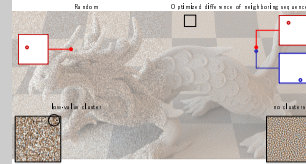
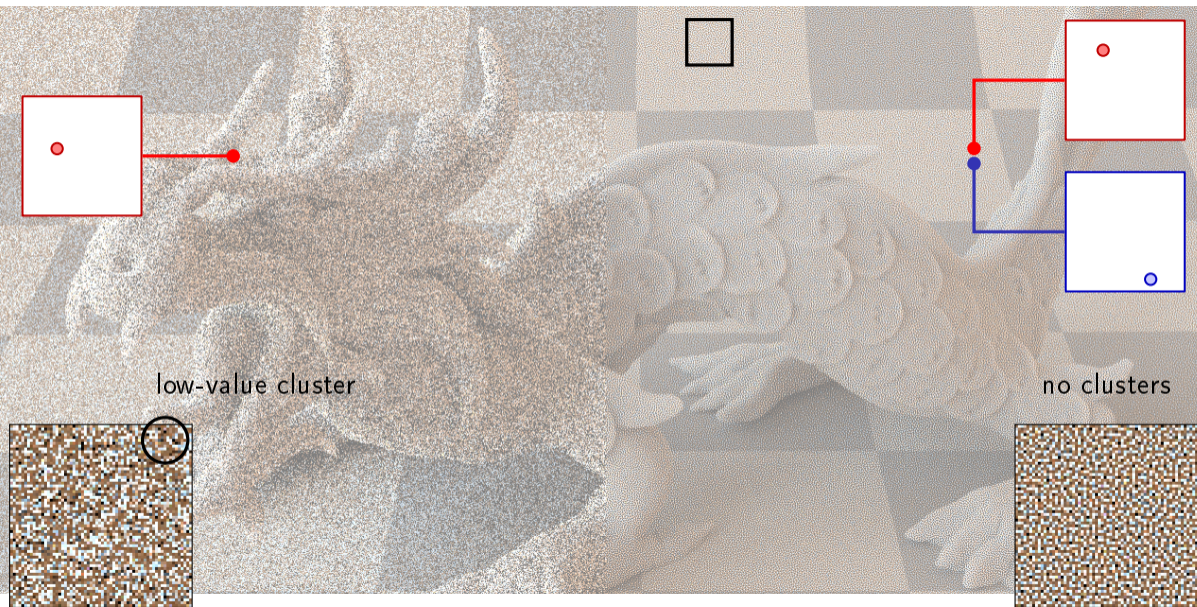
The potential hidden behind this question is the difference between the two parts of this images. Apparently, the right part looks much better than the left part.

On the left, each pixel uses a sequence chosen randomly. On the right, the distribution of the sequences has been optimized such that neighboring pixels have sequences that are as different as possible.

Same number of samples per pixel, same rendering time.

Random

Optimized difference of neighboring sequences



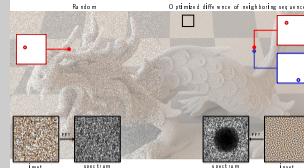
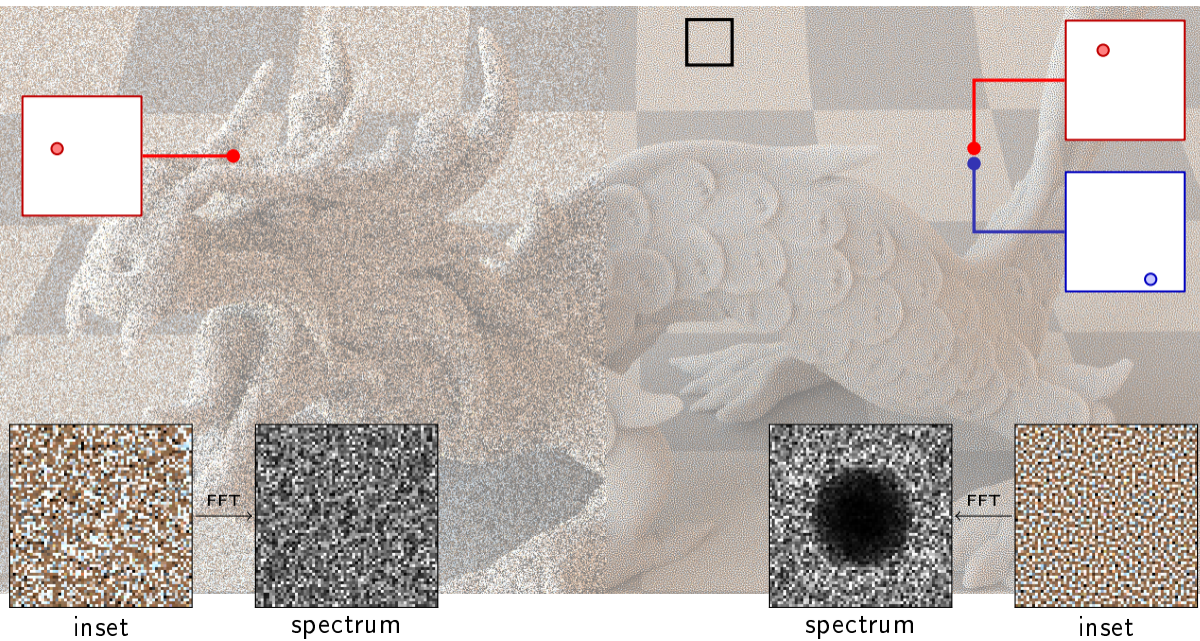
To get a better understanding of the difference in visual quality, let's zoom in a small region of the image.

On the left, when the sequences are chosen purely randomly, we always obtain clusters of pixels with very low (or high) errors. It is this clustering effect that contributes the most to our human perception of noisiness.

On the right, since we have chosen the sequences to maximize the difference between neighboring pixels, we prevent clusters from aggregating together. This is why the image looks less noisy.

Random

Optimized difference of neighboring sequences



However, the amount of noise (the Monte Carlo errors) is really the same in the two images. It is just that it is organized differently. This difference can be characterized by looking at the spectrum of the errors in a small neighborhood.

In the classic random case, the spectrum is statistically flat, i.e. we obtain white-noise errors where all the frequencies are equally represented.


In the optimized case of the right, preventing the clusters from appearing is basically killing the low-frequencies of the spectrum and we obtain blue-noise errors.

This is why the problem we are going to explore is called “*Distributing Monte Carlo errors as a blue noise in screen space*”.

Introduction

white
noise

 $L^2 = 17800$

blue
noise

 $L^2 = 17800$

Blue-noise errors are better perceptually.

white
noise

 $L^2 = 17800$

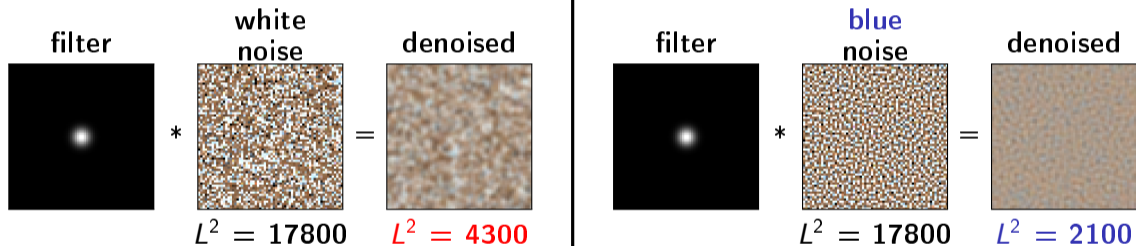
blue
noise

 $L^2 = 17800$

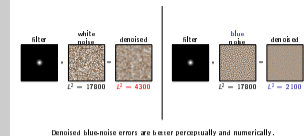
Blue-noise errors are better perceptually.

Note again that the improvement due to the blue-noise error distribution is only perceptual. If we compute the numerical errors of the two images, they are statistically exactly the same.

Introduction

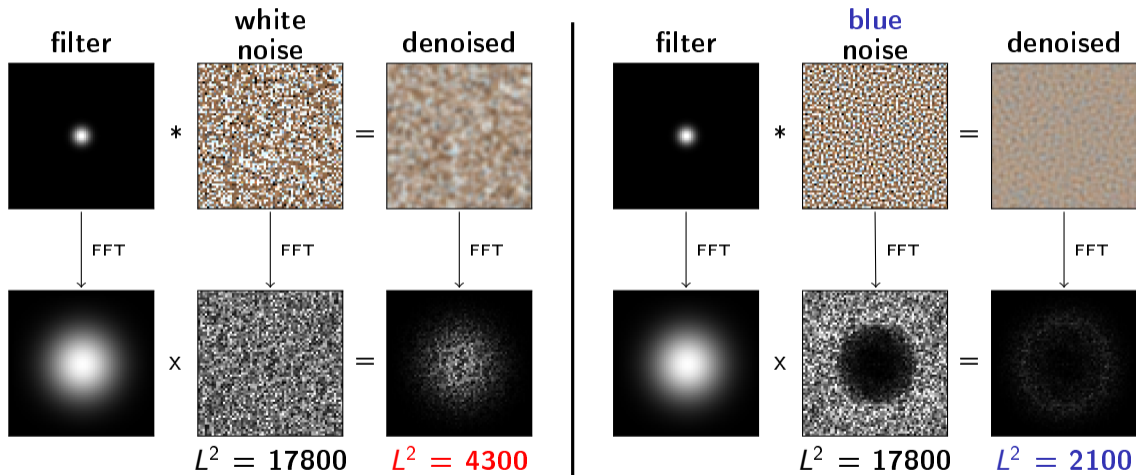


Denoised blue-noise errors are better perceptually and numerically.

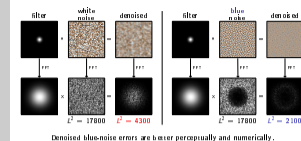


However, one almost never displays a raw noisy image directly. Monte Carlo rendered images are often denoised before being displayed. An interesting effect of the blue-noise error distribution is that it makes the error go down after denoising. Hence, when it is coupled with a denoiser, this concept is no more just a perceptual effect, it becomes a numerical improvement as well.

Introduction



Denoised blue-noise errors are better perceptually and numerically.



We can understand why by looking again in the spectral domain. A denoiser can be seen locally as the convolution with a low-pass filter. In the spectral domain, it becomes a multiplication with the spectrum of the filter.

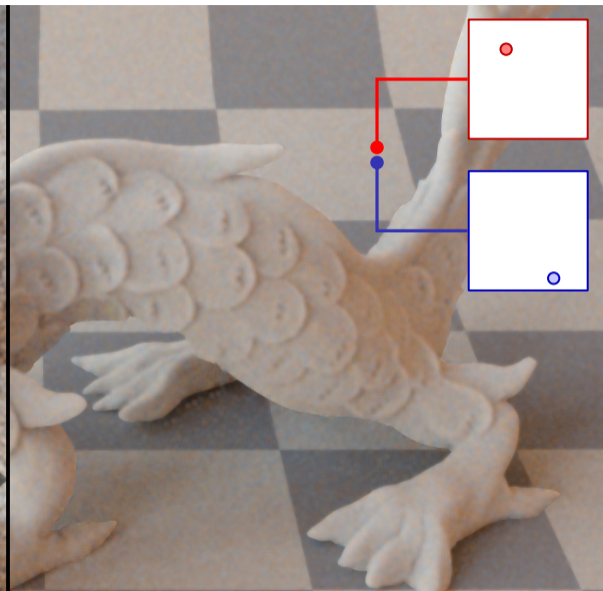
Applying the low-pass filter on the white-noise errors removes the high-frequencies but the energy of the errors located in the low-frequencies remain.

Applying the low-pass filter on the blue-noise errors removes almost all the energy of the errors since it is only located in the high-frequencies. This is why the errors becomes lower numerically after the filtering process.

Random + denoising



Optimized + denoising



Same number of samples per pixel, same rendering time, same denoiser.



Same number of samples per pixel, same rendering time, same denoiser.

This is the dragon image from before after denoising. The difference in quality is impressive.

In summary, the concept of distributing Monte Carlo errors as a blue-noise in screen space is something that improves the visual fidelity of raw noisy images and that boosts the performance of a denoiser.

Introduction

[Georgiev&Fajardo2016]

Blue-noise Dithered Sampling

Ryusei Georgiev
Solid Angle

Marcos Fajardo
Solid Angle

Keywords: Monte Carlo, sampling, blue noise, dithering

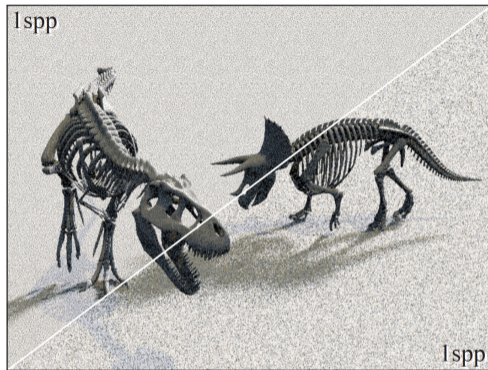
Concepts & competing methodologies → dithering:
The visual fidelity of a Monte Carlo rendered image depends not only on the magnitude of the pixel estimation error but also on its distribution over the image. In this work, state-of-the-art methods use high-quality stratified sampling patterns, which are randomly scattered or shifted to decorrelate the individual pixel estimates. While the white-noise error distribution produced by random pixel distributions is very pleasing, it is far from being perceptually optimal. We show that visual fidelity can be significantly improved by instead correlating the pixel estimates in a way that minimizes the low-frequency content in the output image. Inspired by digital halftoning, our blue-noise dithered sampler can produce substantially more faithful images, especially at low sampling rates.

Blue-noise dithering: In digital halftoning, dithering is the intentional application of noise to randomize the error from quantizing a continuous tone image [1] and [2]. A well-known approach is to threshold the pixels using a blue-noise mask, which is a 2D array of random values arranged such that the frequency spectrum of any thresholded gray-level is isotropic and devoid of low frequencies. That is, neighboring pixels get very different thresholds, and similar thresholds are assigned to pixels far apart.

Dithered sampling: Our idea is to apply the concepts of dithering to correlate pixel estimates in Monte Carlo rendering ray tracing. Given a d -dimensional sampling pattern, we iteratively shift it for every pixel, but rather than choosing the shift randomly, as done traditionally, we look it up in a blue-noise sample mask filled over the image. The value of each pixel in such a pre-computed mask is a d -dimensional vector, and for $d = 1$ the mask is very similar to a halftoning mask. In this setting, traditional random-offset pixel decorrelation is equivalent to using a white-noise sample mask.

References

[1] D. J. Stammers, "The dithering of digital or hard copies of part or all of this work for personal or internal use, not for resale or general distribution, is permitted, provided that copies are not made or distributed for profit or commercial advantage and that copies bear the notice and the full citation on the first page. Copyright for the third party component of this work must be honored. For all other uses, contact the copyright owner. © 2016 Copyright held by the author(s). SIGGRAPH '16, July 24–28, 2016, Austin, TX. ISBN 978-1-60558-240-9. DOI: 10.1145/2831578.2831620



New concept introduced in
Blue-noise Dithered Sampling

by Georgiev and Fajardo, Siggraph Talk 2016



New concept introduced in
Blue-noise Dithered Sampling
by Georgiev and Fajardo, Siggraph Talk 2016

This concept has been in the air for a very long time. For instance, it was mentioned in an early paper of Mitchell in 1991.

But the first people who brought it to life for the first time and made something practical out of it were Georgiev and Fajardo. They presented an idea called Blue-noise Dithered Sampling (BNDS) in a SIGGRAPH Talk in 2016.

Introduction

[Georgiev&Fajardo2016]

Blue-noise Dithered Sampling

Ryusei Georgiev
Solid Angle

Marcos Fajardo
Solid Angle

1D sample mask Our dithered 1D sampling 2D sample mask Our dithered 2D sampling

1000 pixel per spec Random pixel distribution Random pixel distribution Random pixel distribution

Figure 1: Our method uses blue-noise dither masks filled over the image to correlate the samples between pixels and thus minimize the low-frequency content in the distribution of the estimation error. Unlike usually replacing the missing of error, this correlation produces images with higher visual fidelity than traditional random pixel distributions, especially when using a small number of samples per pixel (spp).

Keywords: Monte Carlo, sampling, blue noise, dithering

Concepts of competing methodologies → dithering:
The visual fidelity of a Monte Carlo rendered image depends not only on the magnitude of the pixel estimation error but also on its distribution over the image. In this end, state-of-the-art methods use high-quality stratified sampling patterns, which are randomly shuffled or shifted to decorrelate the individual pixel estimates. While the white-noise error distribution produced by random pixel distributions is very pleasing, it is far from being optimally optimal. We show that visual fidelity can be significantly improved by instead correlating the pixel estimates in a way that minimizes the low-frequency content in the output image. Inspired by digital halftoning, our blue-noise dithered sampling can produce substantially more faithful images, especially at low sampling rates.

Blue-noise dithering: In digital halftoning, dithering is the intentional application of noise to randomize the error from quantizing a continuous tone image [1] and [2]. A sufficient approach is to threshold the pixels using a blue-noise mask, which is a 2D array of scalar values arranged such that the frequency spectrum of any thresholded grey-level is isotropic and devoid of low frequencies. That is, neighboring pixels get very different thresholds, and similar thresholds are assigned to pixels far apart.

Dithered sampling: Our idea is to apply the concepts of dithering to correlate pixel estimates in Monte Carlo distributions for tracing. Given a d -dimensional sampling pattern, we iteratively shift it for every pixel, but rather than choosing the offset randomly, as done traditionally, we look it up in a blue-noise sample mask filled over the image. The value of each pixel in such a pre-computed mask is a d -dimensional vector, and for $d = 1$ the mask is very similar to a halftoning mask. In this setting, traditional random pixel distributions is equivalent to using a white-noise sample mask.

DISCLAIMER: This document is made digital or hard copies of part or all of this work for personal or classroom use, or internal or personal use, or the personal or internal use of specific clients, is granted by ACM for users registered with ACM, provided that the copyright notice, this notice, and the full citation on the first page are preserved. Copyright for this part of the work must be retained. For all other uses, contact the author(s). © 2016 Copyright held by the author(s). SIGGRAPH 16, July 24–28, 2016, Austin, TX. ISBN 978-1-4503-3915-9/16/07. DOI: 10.1145/2831539.2831540

Optimal at 1 sample per pixel in low dimensions.

→ For real-time or previz.

Vanishes at high sample counts and high dimensions.

→ Not for offline path tracing.



Optimal at 1 sample per pixel in low dimensions.
→ For real-time or previz.

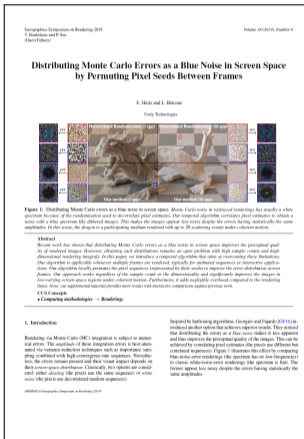
Vanishes at high sample counts and high dimensions.
→ Not for offline path tracing.

BNDS produces terrific results in simple cases (typically direct illumination) and low sample counts. However, the effect vanishes at higher sampling count and higher dimensionalities of the rendering integrand.

Because of this limitation, BNDS is something that is only meant to improve preview images or real-time path tracing. It won't improve a final beauty render in an offline path tracer.

Introduction

[Heitz&Belcour2019]



New theoretical formulation + temporal algorithm.

Scales to high sample counts and high dimensions.

→ For offline path tracing.



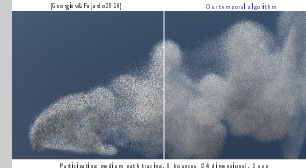
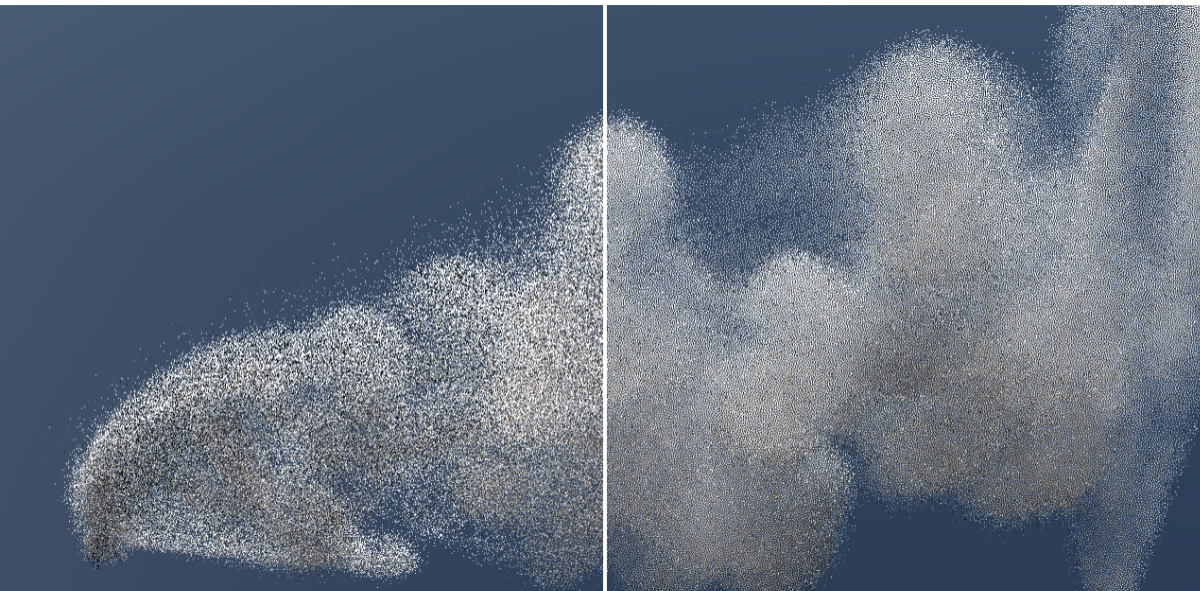
New theoretical formulation + temporal algorithm.

Scales to high sample counts and high dimensions.
→ For offline path tracing.

The ambition of our paper is to bring the concept of distributing MC errors as a blue noise to true path tracing by overcoming these limitations. To do this, we introduce a theoretical formulation and a practical temporal algorithm that approximates it.

[Georgiev&Fajardo2016]

Our temporal algorithm

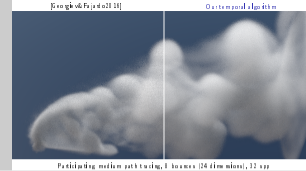
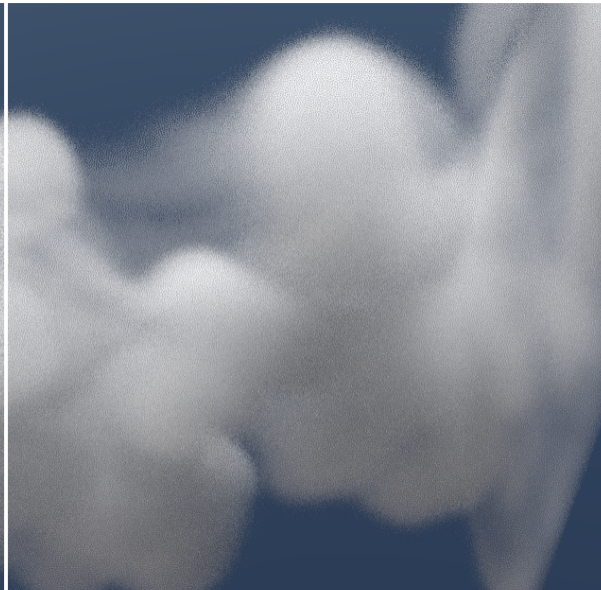
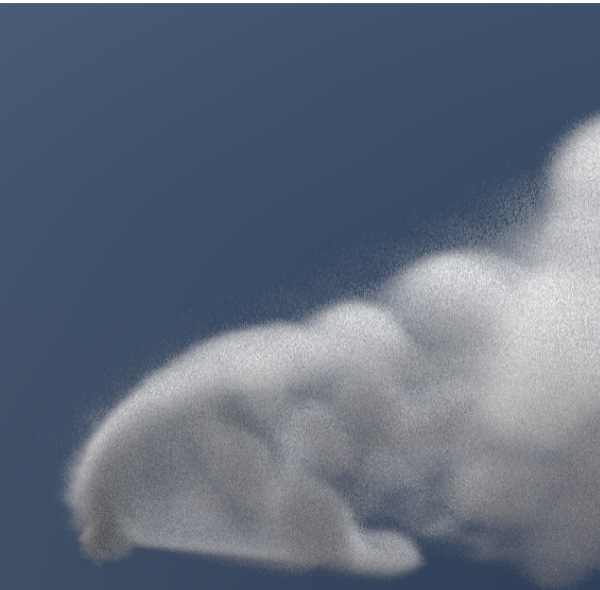


Here is a comparison of BNDS and our temporal algorithm. This scene is interesting because it is a participating medium rendered with path tracing, i.e. the dimensionality is very high. Because of this, BNDS of Georgiev and Fajardo does not improve the error distribution compared to a classic white noise. Our temporal algorithm is able to improve the result despite the high dimensionality (zoom in to see how the clusters are prevented in our image.)

Participating medium path tracing, 8 bounces (24 dimensions), 1 spp

[Georgiev&Fajardo2016]

Our temporal algorithm



This image is the same scene rendered at 32 spp instead of 1 spp. It shows that our algorithm scales not only in terms of dimensionality but also in terms of sample count.

Participating medium path tracing, 8 bounces (24 dimensions), 32 spp

Blue-noise Dithered Sampling [Georgiev&Fajardo2016]

Before talking about our algorithms, we would like to share some insights we gather regarding BNDS.

Blue-noise Dithered Sampling [Georgiev&Fajardo2016]



halftoning



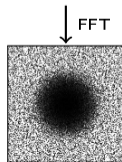
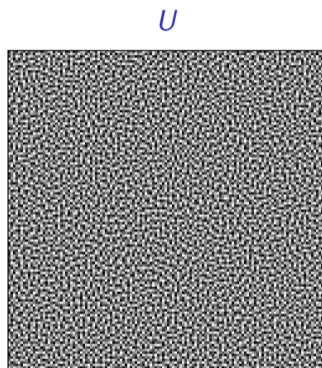
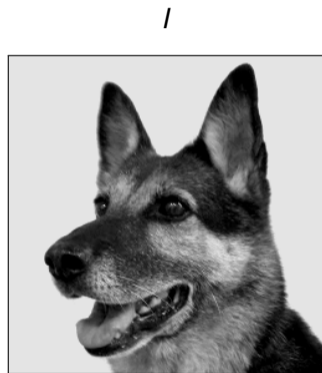
How to simulate shades of gray using black dots?

The original inspiration of Georgiev and Fajardo came from dithering algorithms for digital halftoning.

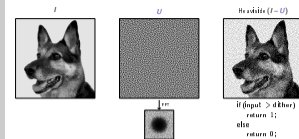
Given a grayscale image, how to obtain a binary image such that the interleaving of black and white somehow simulates the same shades of gray?

How to simulate shades of gray using black dots?

Blue-noise Dithered Sampling [Georgiev&Fajardo2016]



```
if(input > dither)
  return 1;
else
  return 0;
```

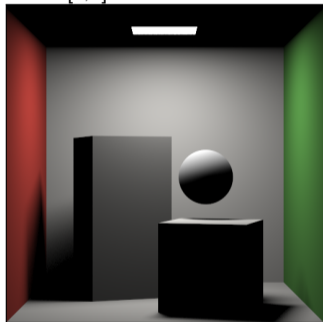


A classic technique to do that is to compare each grayscale pixel with a random number. If the random number is smaller store white, black otherwise. Furthermore, in the halftoning community, it is well-known that using a blue-noise texture to feed the random numbers achieves the best results.

One of the best way to obtain such a blue-noise texture is the void-and-cluster algorithm [Ulichney1983].

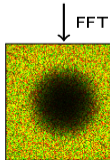
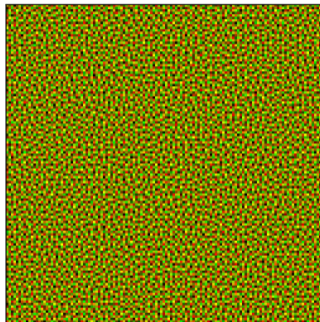
Blue-noise Dithered Sampling [Georgiev&Fajardo2016]

$$\int_{[0,1]^2} f(x,y) dx dy$$

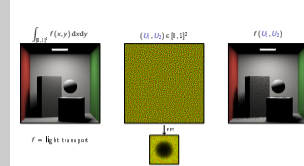
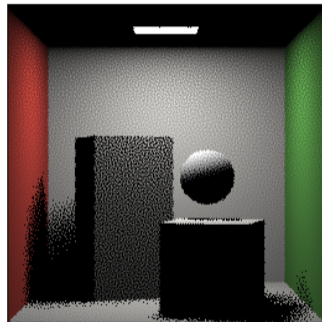


$f = \text{light transport}$

$$(U_1, U_2) \in [0,1]^2$$



$$f(U_1, U_2)$$



The idea of Georgiev and Fajardo is: why don't we do the same for Monte Carlo rendering?

Each pixel of a Monte Carlo rendered image is the result of an integral estimates using random numbers. If we feed the random numbers using a blue-noise texture, we will distribute the errors as a blue noise, exactly like halftoned images.

Actually, to do that, we need blue-noise textures that contain random vectors of the same dimensionality as the rendering integrand. For instance, in this image the light transport is 2D (the direct illumination of an area light) so we need a blue-noise texture with 2 channels per pixel.

Blue-noise Dithered Sampling [Georgiev&Fajardo2016]

[Georgiev&Fajardo2016]

Blue-noise Dithered Sampling
Hrayr Georgiev
Marcos Fajardo

Figure 1: Our method uses blue-noise dither masks that cover the image to correlate the samples between pixels and that minimize the frequency content in the distribution of the quantization error. Without actually reducing the amount of error, this correlation produces images with higher visual fidelity than traditional random-point dithering, especially when using a small number of samples per pixel (e.g., 1-4).

Keywords: Monte Carlo sampling, blue noise, dithering, concepts of sampling methodologies in rendering.

The visual fidelity of a Monte Carlo rendered image depends not only on the magnitude of the final estimator error but also on its distribution over the image. To this end, state-of-the-art methods use high-quality stratified sampling patterns, which are carefully randomized or dithered to decorrelate the individual pixel estimates. While the white-noise image error distribution produced by random point dithering is very pleasing, it is far from being perceptually optimal. We show that visual fidelity can be significantly improved by instead considering the pixel estimates as a set that minimizes the low-frequency content in the output noise. Inspired by digital halftoning, our blue-noise dithered sampling can produce substantially more faithful images, especially at low sampling rates.

Blue-noise dithering: In digital halftoning, dithering is the iterative application of noise to randomize the error from quantizing a continuous-tone image [1] and [2]. A dithered image is produced by thresholding the pixels using a blue-noise mask, which is a 2D array of random values arranged in a regular grid. The values of any thresholded gray level in isotropic and diagonal low frequencies. That is, neighboring pixels get very different thresholds, and similar thresholds are assigned to pixels far apart.

Dithered sampling: Our idea is to apply the concept of dithering to control-point estimates in Monte Carlo distribution ray tracing. Given a d -dimensional sampling pattern, we iteratively shift it for every pixel, but rather than choosing the offset randomly, we choose randomly, we look it up in a blue-noise sample mask that covers the image. The values of every pixel in such a pre-computed mask is a d -dimensional noise, and for $d = 1$ the mask is very similar to halftoning mask. In this setting, randomized random-point dithering is equivalent to using a white-noise sample mask.

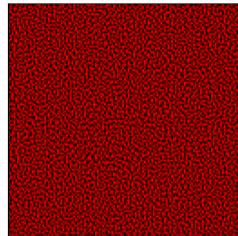
References

[1] J. D. VAN ANTWERP, 2004. *Principles of Digital Halftoning*. Second Edition. CRC Press.

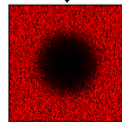
[2] MICHAEL, D. F. 1991. Spatially optimal sampling for distributed tone rendering. *ACM SIGGRAPH Computer Graphics*, 21, 4, 302-311.

[3] URSCHER, R. A. 1995. The variance-covariance method for dithering gray generation. *Proc. SPIE*, 1825, 325-342.

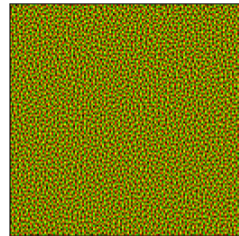
$U_1 \in [0, 1]$



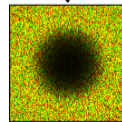
FFT



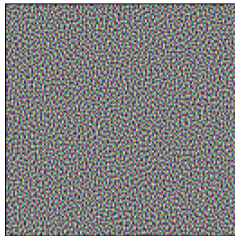
$(U_1, U_2) \in [0, 1]^2$



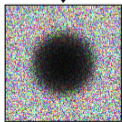
FFT



$(U_1, U_2, U_3) \in [0, 1]^3$



FFT



How to produce higher-dimensional blue-noise textures?

$U_1 \in [0, 1]$ $(U_1, U_2) \in [0, 1]^2$ $(U_1, U_2, U_3) \in [0, 1]^3$

This is precisely what this paper is about. It introduces an algorithm that computes these textures for an arbitrary dimensionality D (here $D = 1, 2, 3$).

How to produce higher-dimensional blue-noise textures?

Blue-noise Dithered Sampling [Georgiev&Fajardo2016]

[Georgiev&Fajardo2016]

Blue-noise Dithered Sampling
 Hrayr Georgiev, Solid Angle, Maxon Fajardo, Solid Angle

Figure 1: Our method uses blue-noise dithered masks that cover the image to correlate the samples between pixels and that minimize the frequency content in the distribution of the estimation error. Without actually reducing the amount of error, this correlation produces images with higher visual quality than traditional random pixel dithering, especially when using a small number of samples per pixel (ppp).

Keywords: Monte Carlo sampling, blue noise, dithering, concepts of sampling methodologies, dithering.

The visual quality of a Monte Carlo rendered image depends not only on the amount of the pixel resolution covered also on its distribution over the image. To this end, some-of-the-art methods use high quality stratified sampling patterns, which are carefully randomized or dithered to decorrelate the individual pixel estimates. While the white-noise image error distribution produced by random pixel dithering is very pleasing, it is far from being perceptually optimal. We show that visual fidelity can be significantly improved by instead considering the pixel estimates as a way that minimizes the low-frequency content in the output noise. Inspired by digital halftoning, our blue-noise dithered sampling can produce visually rich non-halftone images, especially at low sampling rates.

Blue-noise dithering: In digital halftoning, dithering is the iterative application of noise to randomize the error from quantization of a continuous-tone image [Zi and Avery 2008]. An effective approach is to dithered the pixels using a blue-noise mask, which is a 2D array of uniform energy spread that the sampling process can use to decorrelate the gray level in isotropic and diagonal low frequencies. That is, neighboring pixels get very different thresholds, and similar thresholds are assigned to pixels far apart.

Dithered sampling: Our idea is to apply the concept of dithering to correlated pixel estimates in Monte Carlo distribution ray tracing. Given a d -dimensional sampling pattern, we iteratively shift it for every pixel, but rather than choosing the offset randomly, we choose randomly, we look it up in a blue-noise sample mask that covers the image. The value of every pixel to render is pre-computed results in a d -dimensional vector, and for $d = 1$ the mask is very similar to halftoning mask. In this setting, randomized random pixel dithering is equivalent to using a white-noise sample mask.

References

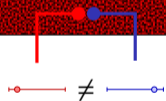
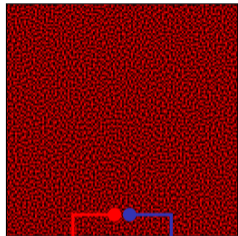
[Zi, 2008] Zi, H. *Randomness in color digital halftone images*. In: *Color and Halftone*, pp. 1-12. Springer, 2008.

[Liu, D., and ARY, C. 2008. *Modern Digital Halftoning*. Second Edition. CRC Press.

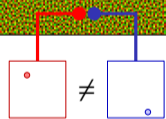
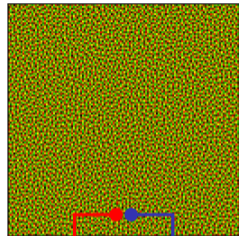
[Mitschke, D. D. 1991. *Optimally dithered sampling for distributed ray tracing*. *ACM SIGGRAPH Computer Graphics*, 25, 4, 353-362.

[Lichtenberg, R. A. 1993. *The variational-charts method for dithered gray generation*. *Proc. SPIE 1913*, 325-341.

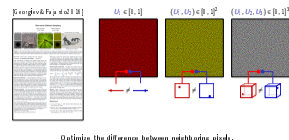
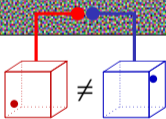
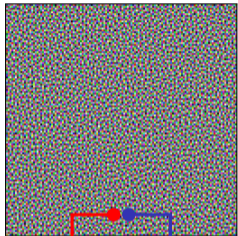
$U_1 \in [0, 1]$



$(U_1, U_2) \in [0, 1]^2$



$(U_1, U_2, U_3) \in [0, 1]^3$



Optimize the difference between neighboring pixels.

The algorithm is an optimization system. Each pixel is going to store a D -dimensional vector. The algorithm optimizes the location of the vectors such that they are always as different as possible from the neighboring ones.

When this constraint is optimized, we effectively obtain a blue-noise texture.

Optimize the difference between neighboring pixels.

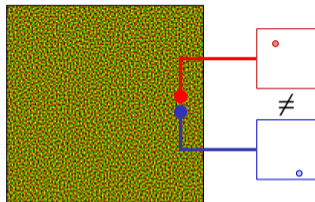
Blue-noise Dithered Sampling [Georgiev&Fajardo2016]



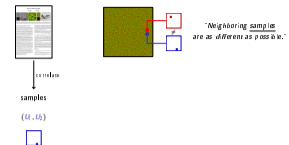
correlate

samples

(U_1, U_2)

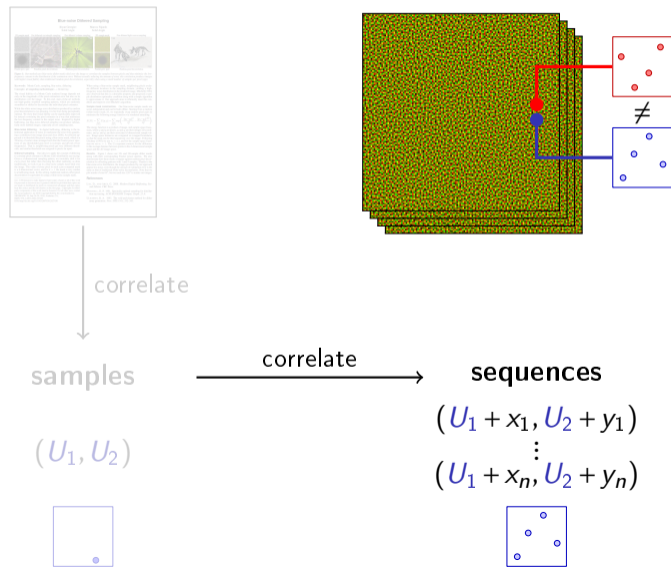


*“Neighboring samples
are as different as possible.”*

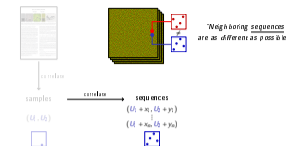


So, this paper is about optimizing the negative correlation between neighboring pixels. When we say “negative correlation” it just means “as different as possible”.

Blue-noise Dithered Sampling [Georgiev&Fajardo2016]



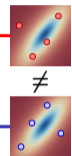
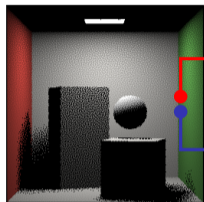
“Neighboring sequences are as different as possible.”



When we need a sample count higher than 1, Georgiev and Fajardo propose to use a unique sequence $(x_1, y_1), \dots, (x_n, y_n)$ for all the pixels and offset it by the blue-noise samples. The offsetting is done modulo 1, i.e. the samples that go out of the unit square are warped back on the other side of the square. This is also called “Cranley Patterson rotation”, or just “toroidal shift”.

By doing this we obtain a bunch a blue-noise textures where each texture represent one offsetted sample of the sequence.

Blue-noise Dithered Sampling [Georgiev&Fajardo2016]



“Neighboring errors are as different as possible.”

correlate

samples

(U_1, U_2)



correlate

sequences

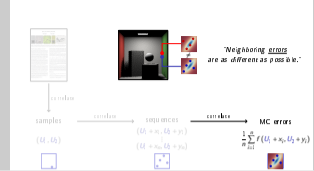
$(U_1 + x_1, U_2 + y_1)$
 \vdots
 $(U_1 + x_n, U_2 + y_n)$



correlate

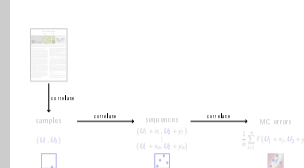
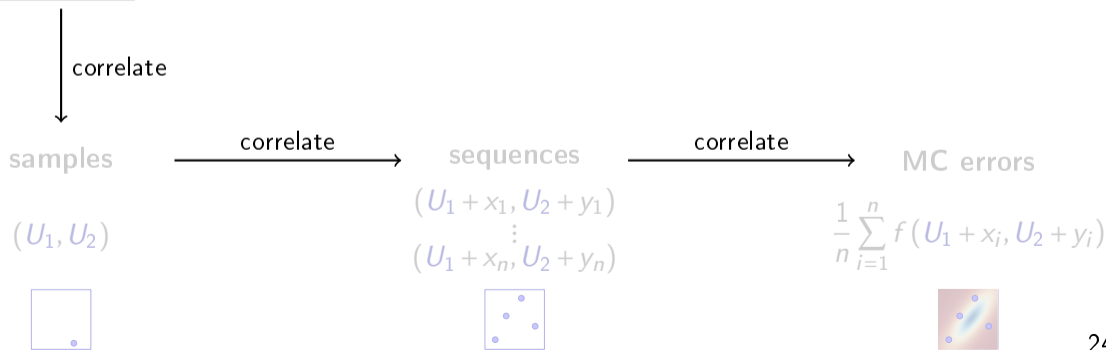
MC errors

$\frac{1}{n} \sum_{i=1}^n f(U_1 + x_i, U_2 + y_i)$



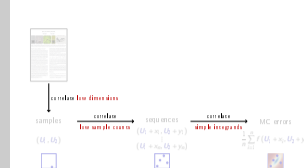
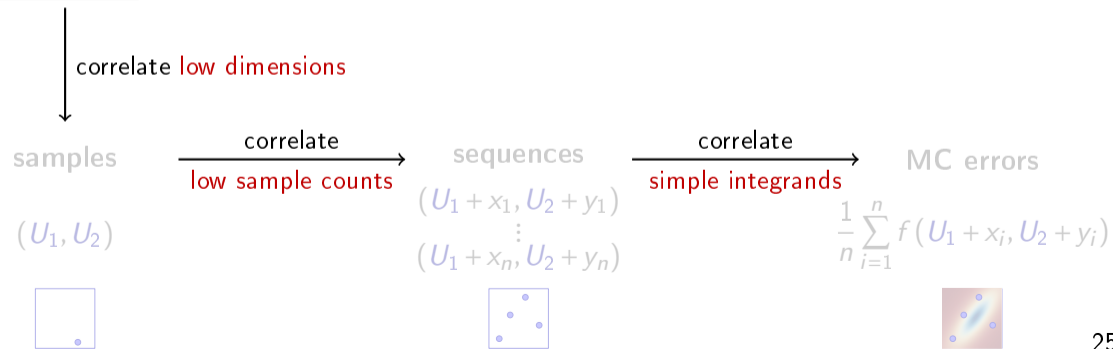
Finally, the sequences are used to compute a Monte Carlo estimate of the light transport in each pixel, which produces an error. Since neighboring sequences are as different as possible, we assume that their resulting MC errors are also as different as possible.

Blue-noise Dithered Sampling [Georgiev&Fajardo2016]



Once we explain it in this way, we can see that BNDS relies on a chain of correlations. The correlations originally optimized for the samples are transferred to the sequences and finally they get transferred to the MC errors, which is what we want.

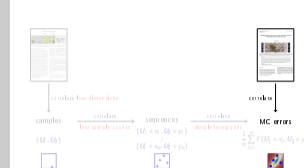
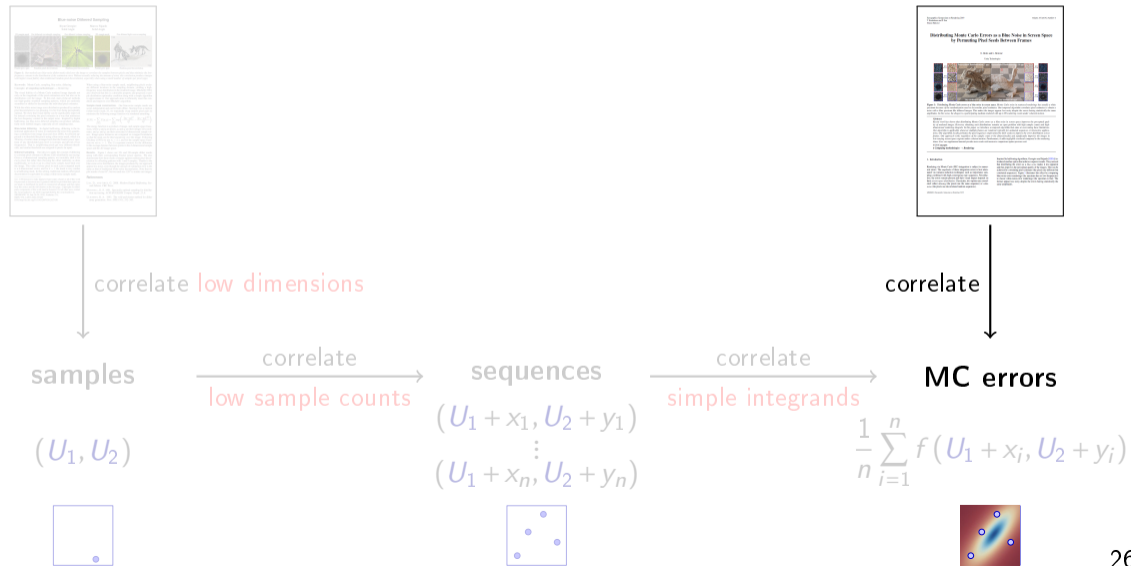
Blue-noise Dithered Sampling [Georgiev&Fajardo2016]



However, while the chain of correlations works very well in simple cases, there are also many reasons for it to break down.

For instance, the original correlation of the samples does not work very well in high dimensions. This is a classic problem with blue noise in general. Another problem is that the correlation of the samples does not transfer well to the sequences when the number of elements in the sequence is high. The longer the sequence, the less it preserves the correlation. Finally, the light-transport function f preserves the correlation of the sequences only if it is simple enough. If f has many discontinuities, oscillations, etc. the correlation is not successfully transferred to the errors.

Our approach

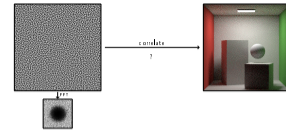
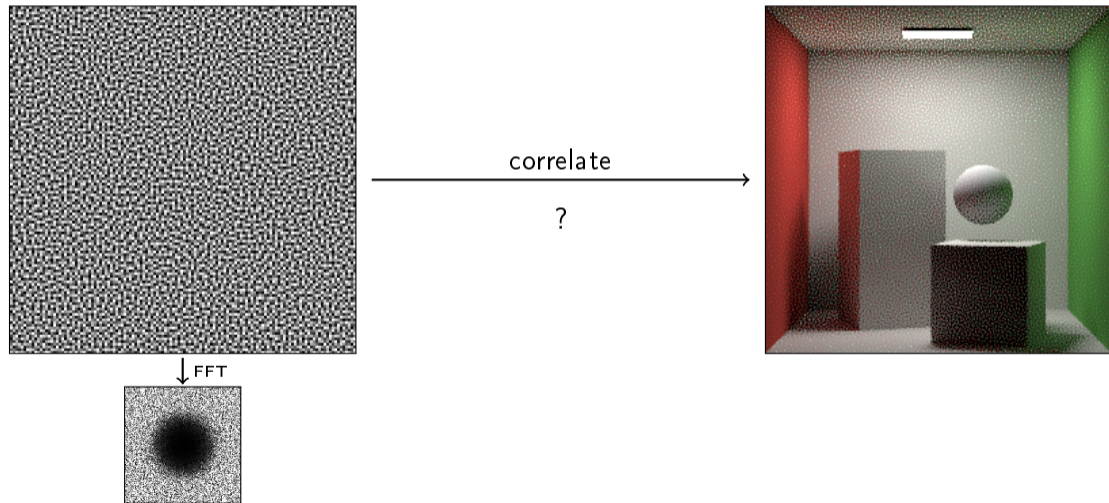


The approach of our paper is that if we want to correlate the errors, then we should design a technique that optimizes the errors directly instead of optimizing something else and hoping that it will be successfully transferred to the errors.

How can we directly correlate Monte Carlo errors?

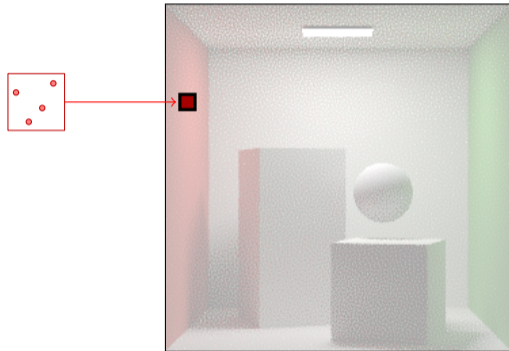
This leads us to this question: how can we directly correlate Monte Carlo errors?

How can we directly correlate Monte Carlo errors?

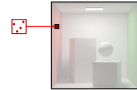


What we mean by correlating MC errors directly is that we would like to be able to transfer the correlation of a blue-noise texture directly to the errors of the rendered image, without additional intermediate steps.

How can we directly correlate Monte Carlo errors?



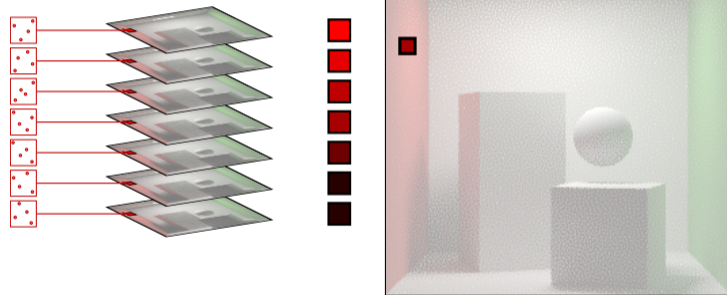
We compute a pixel value using a sequence of random numbers.



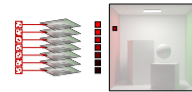
We compute a pixel value using a sequence of random numbers.

First, let's remember how a pixel value is computed. Typically, a sequence is chosen randomly and used to estimate the pixel value.

How can we directly correlate Monte Carlo errors?



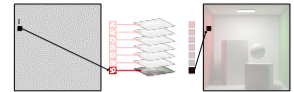
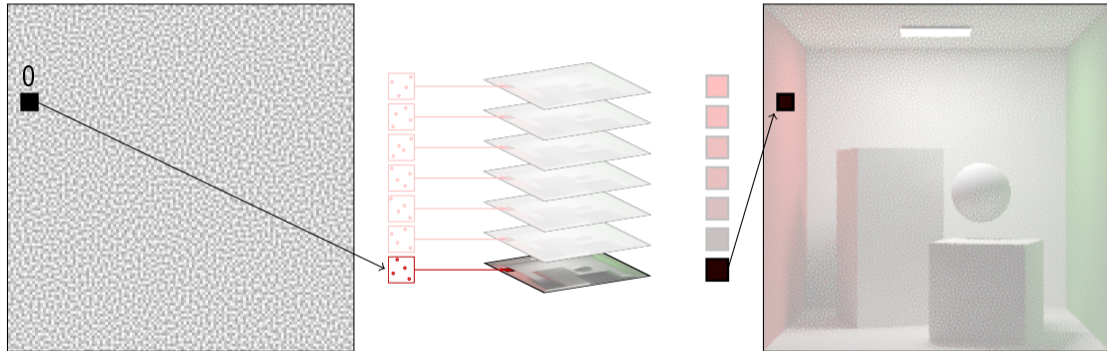
Consider all the potential values for this pixel and sort them.



Consider all the potential values for this pixel and sort them.

There is an infinity of different sequences that we could use to estimate the pixel. Let's consider all the potential sequences that we could use and the values that they would produce for this pixel. All these values are valid candidate for this pixel when it is rendered at 4 spp. Let's consider them all and store them in a sorted list.

How can we directly correlate Monte Carlo errors?

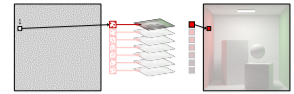
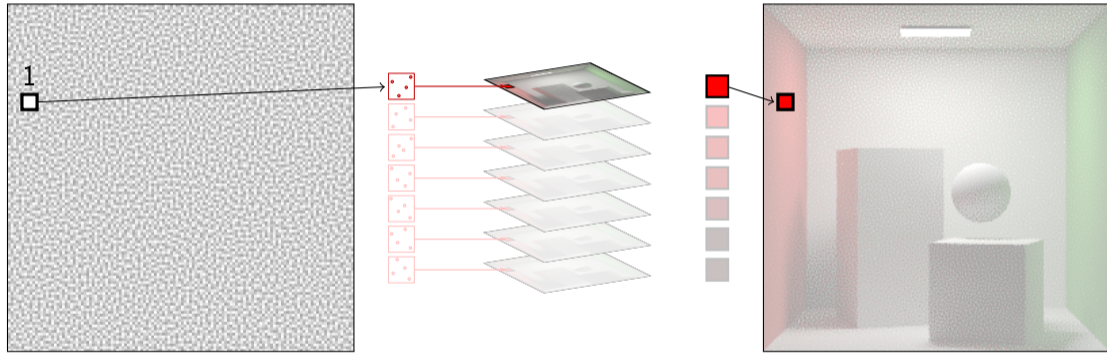


Sample the sorted list using the random number provided by the blue-noise texture.

What we are going to do is to choose one of these values based on the value of this pixel in the blue-noise texture.

For instance, if the blue-noise texture has a very small value, we choose a small element in the sorted list.

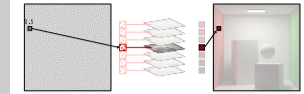
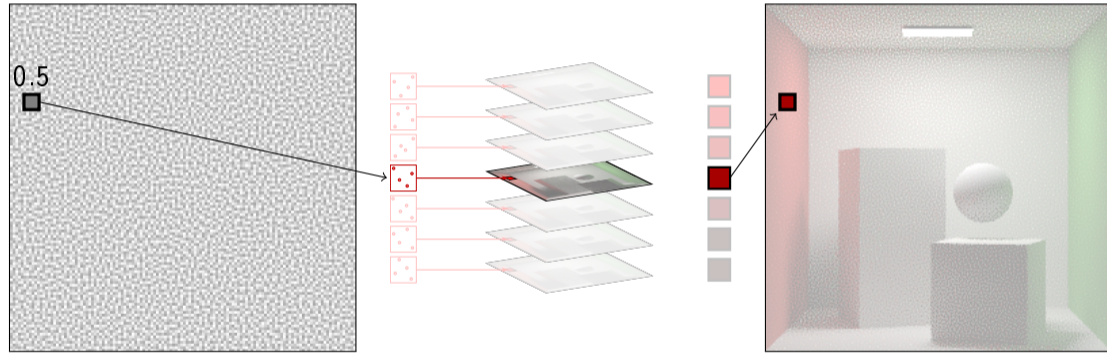
How can we directly correlate Monte Carlo errors?



Sample the sorted list using the random number provided by the blue-noise texture.

If the blue-noise texture has a very high value, we choose a high element in the sorted list.

How can we directly correlate Monte Carlo errors?

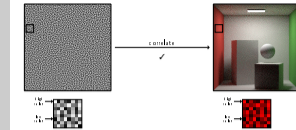
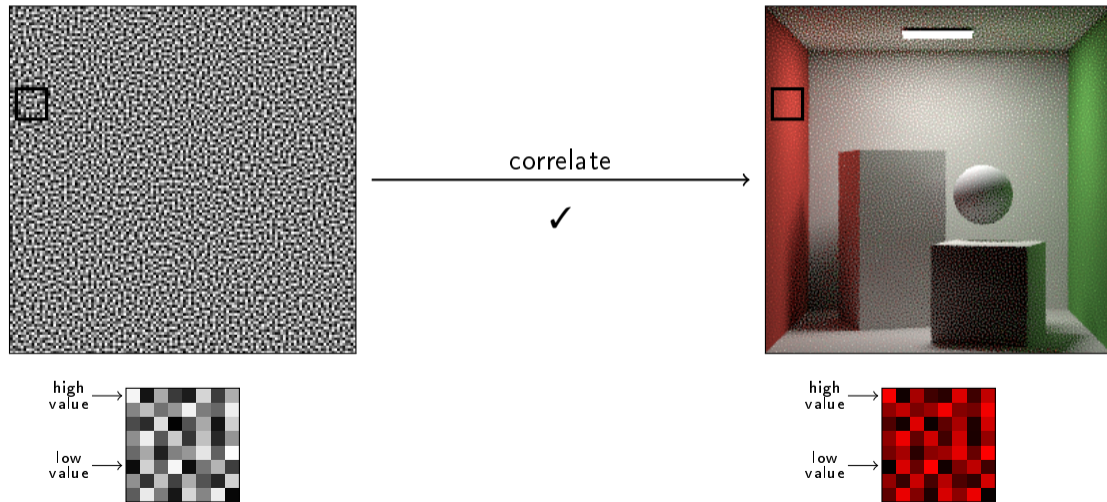


Sample the sorted list using the random number provided by the blue-noise texture.

If the blue-noise texture has a median value, we choose a median element in the sorted list.

We are just sampling the sorted list using the random number provided by the blue-noise texture.

How can we directly correlate Monte Carlo errors?

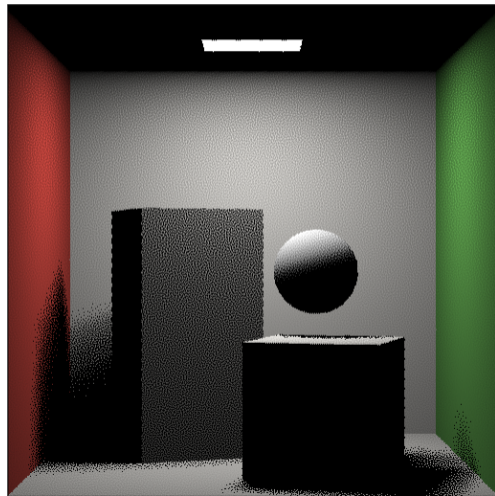
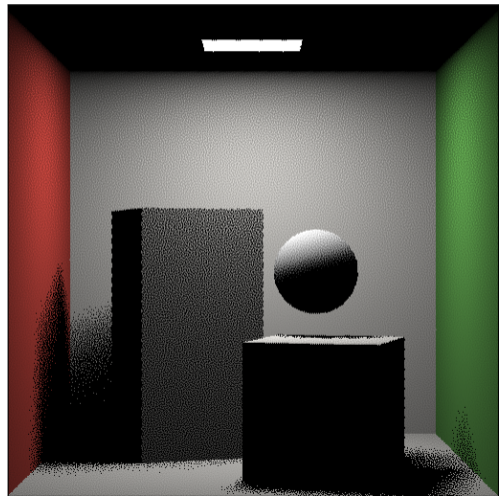


By doing this, we are effectively transferring the spatial correlations of the blue-noise texture to the errors of the rendered image. When the blue-noise texture has a small value, the rendered image has a small error, etc.

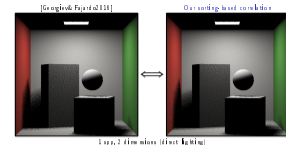
How can we directly correlate Monte Carlo errors?

[Georgiev&Fajardo2016]

Our sorting-based correlation



1 spp, 2 dimensions (direct lighting)

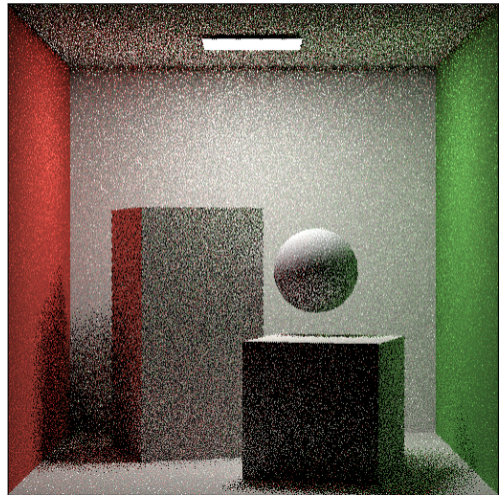


Let's compare this idea to BNDS.

In this scene, the sample count is only 1, the dimensionality is low, and the integrand is simple (a small area light). This is a case where the chain of correlations assumed by BNDS works well and the two formulations achieve pretty much the same quality.

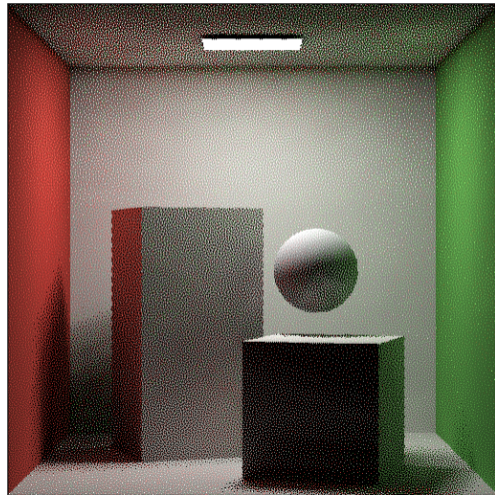
How can we directly correlate Monte Carlo errors?

[Georgiev&Fajardo2016]

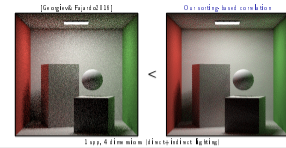


1 spp, 4 dimensions (direct+indirect lighting)

Our sorting-based correlation



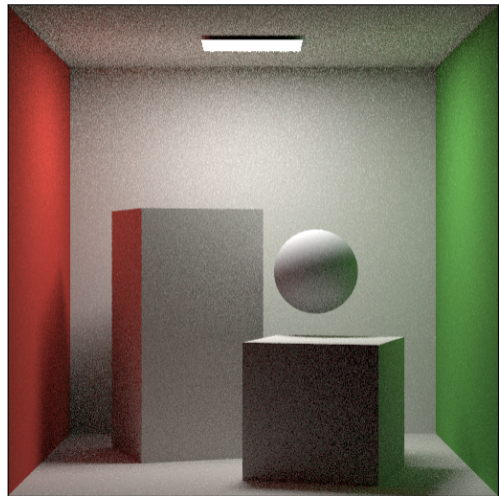
>



However, if we add some global illumination, the dimensionality increases and BNDS breaks down while our formulation still achieves a nice blue-noise distribution of the errors.

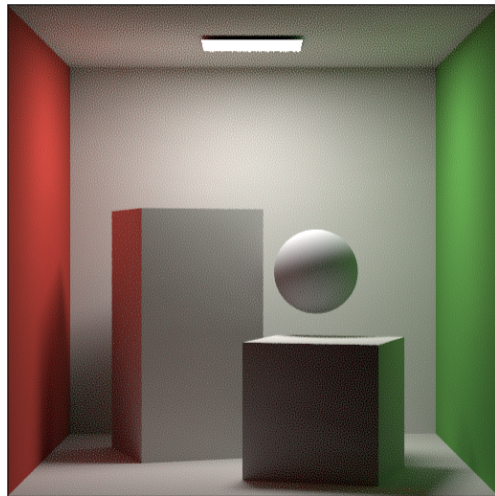
How can we directly correlate Monte Carlo errors?

[Georgiev&Fajardo2016]

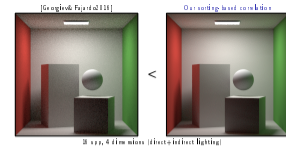


16 spp, 4 dimensions (direct+indirect lighting)

Our sorting-based correlation



<

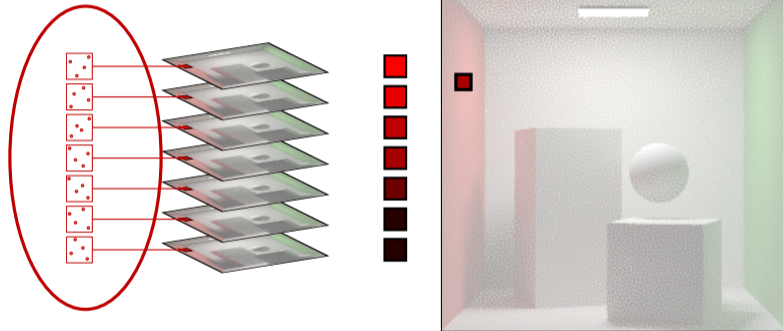


In this example, we also increased the number of samples per pixel. The image produced by our formulation looks like it is almost converged in this case.

This shows that our formulation based on the sorted list scales in terms of dimensionality and sample count.

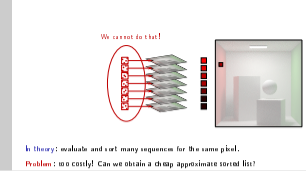
How can we directly correlate Monte Carlo errors?

We cannot do that!



In theory: evaluate and sort many sequences for the same pixel.

Problem: too costly! Can we obtain a cheap approximate sorted list?



The problem is that this idea is not practical. To compute these images, we need to render the pixels many times. For the same rendering time, one could just render a converged image without visible error!

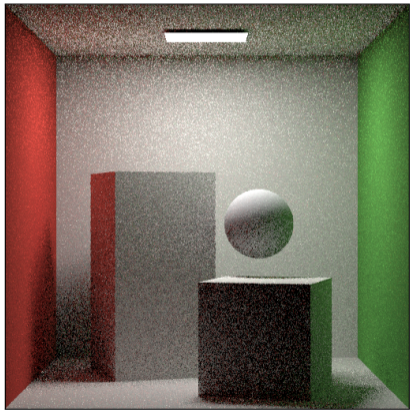
In order to make this approach practical, we need to find a way to predict the sorted list without actually rendering the pixels many times.

Temporal Algorithm

This is where our temporal algorithm comes into play.

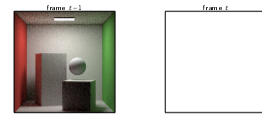
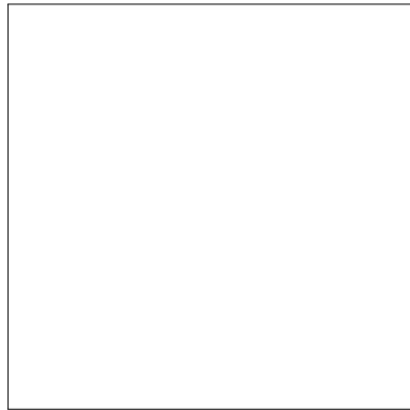
Temporal Algorithm

frame $t-1$



We just rendered the previous frame.

frame t

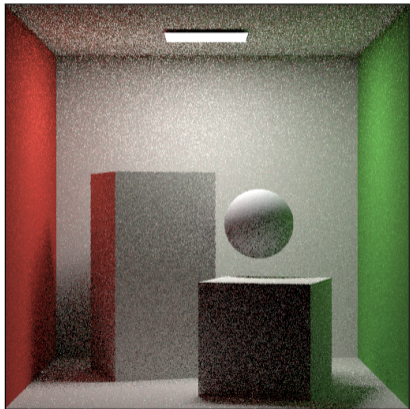


We just rendered the previous frame.

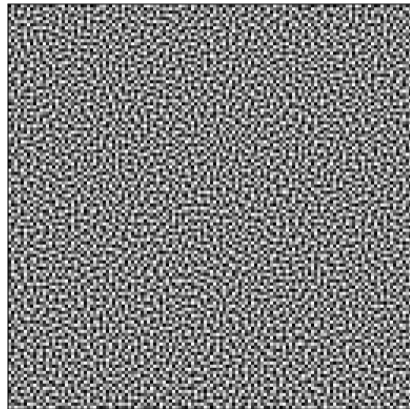
The context of our temporal algorithm is the following: we just rendered the previous frame and we are about to render the next one.

Temporal Algorithm

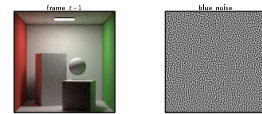
frame $t-1$



blue noise



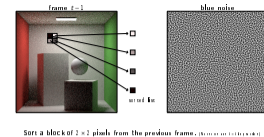
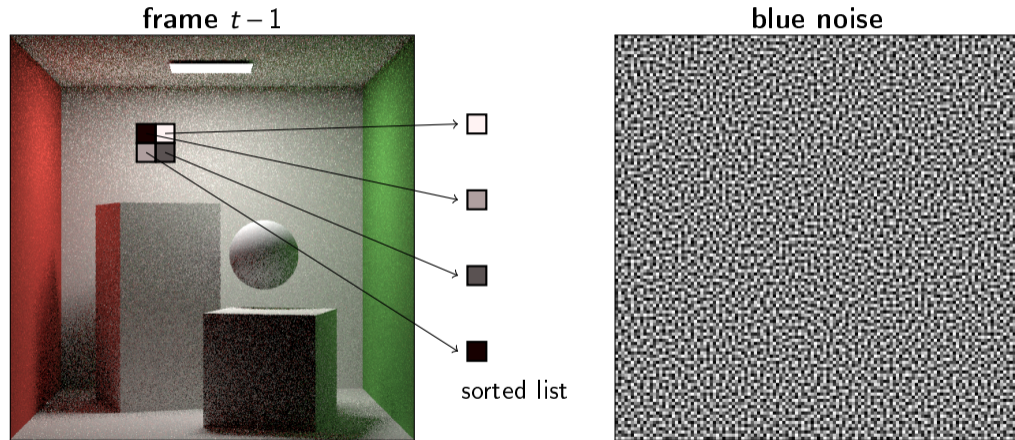
We want to correlate the next frame as this blue noise texture.



We want to correlate the next frame as this blue noise texture.

What we would like to do is to force the new frame to have the same correlation as this blue-noise texture.

Temporal Algorithm

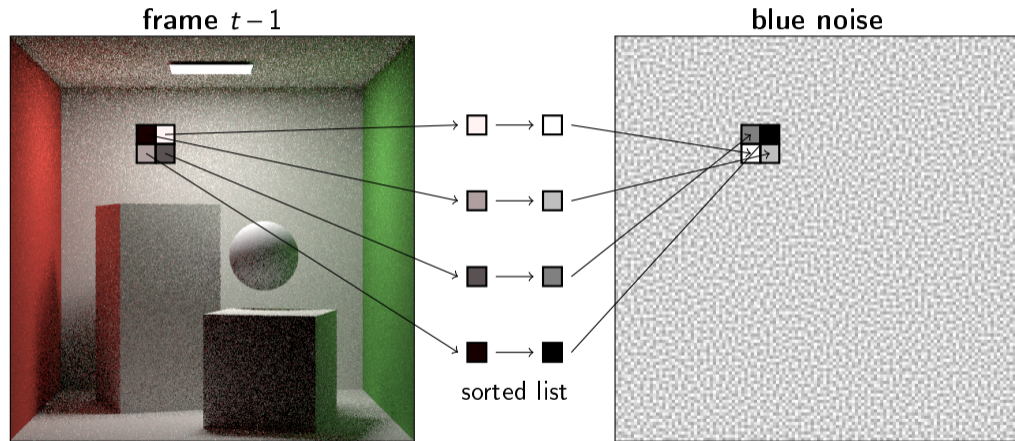


We divide the image space in small blocks (in this illustration it is 2x2 but in practice we always do 4x4).

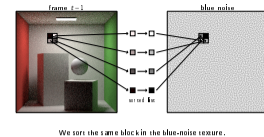
For each block, we sort the pixel values in the previous frame...

Sort a block of 2×2 pixels from the previous frame. (Note: we use 4×4 in practice)

Temporal Algorithm

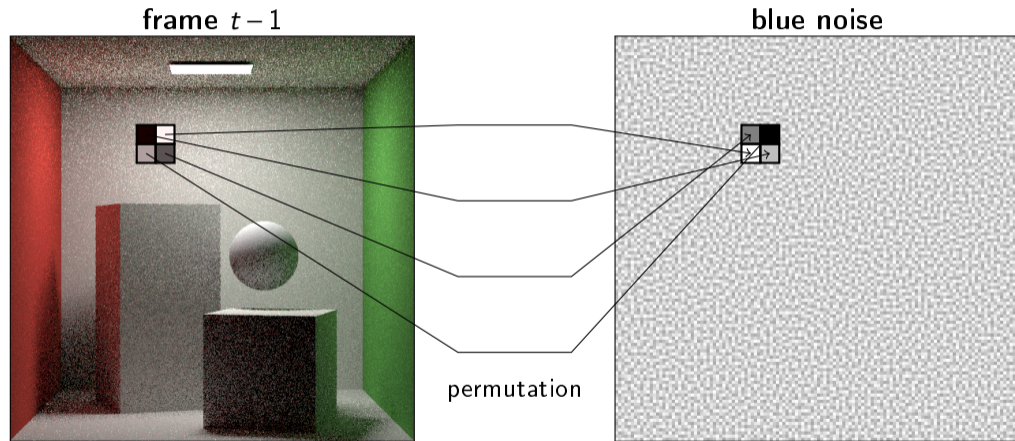


We sort the same block in the blue-noise texture.

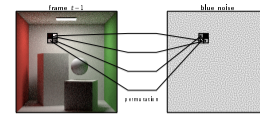


... and in the blue-noise texture.

Temporal Algorithm



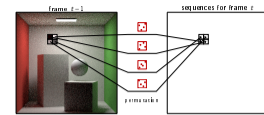
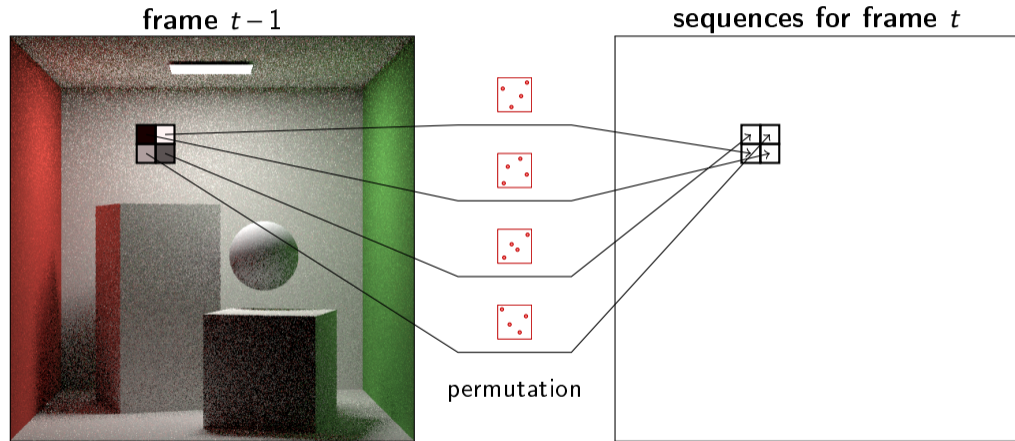
We obtain a permutation of the pixels in the block.



We obtain a permutation of the pixels in the block.

By putting the two sorted list side by side we obtain a permutation of the pixel coordinates inside the block.

Temporal Algorithm



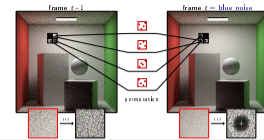
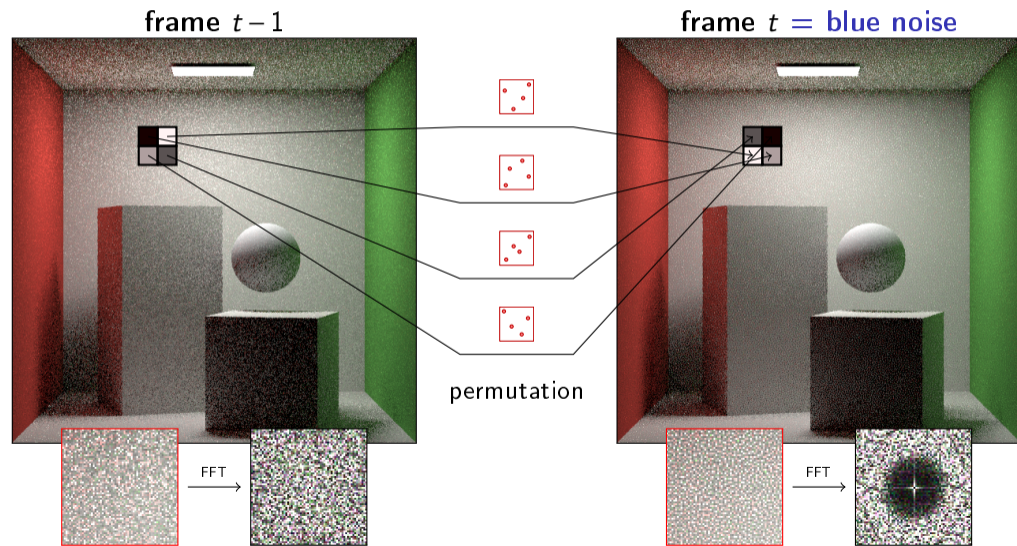
We apply this permutation on the sequences before rendering the next frame.

The idea is to apply this permutation on the sequences that produces the pixel values in the previous frame.

For instance, the sequence that produced the smallest value in the previous frame will be relocated to where we want the smallest value to be located in the next frame, etc.

We apply this permutation on the sequences before rendering the next frame.

Temporal Algorithm



Now that the sequences to use for the new frame are decided, all we have to do is press the rendering button. What we obtain is a frame that has the same blue-noise correlations as the blue-noise texture.

Temporal Algorithm

For each 4x4 block

```
sort(FramePixels[1..16])
```

```
sort(BlueNoisePixels[1..16])
```

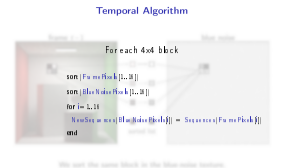
```
for i=1..16
```

```
  NewSequences(BlueNoisePixels[i]) = Sequences(FramePixels[i])
```

```
end
```

Our algorithm is very simple to implement. All we do is, between two frames, we divide the image in small blocks and sort their values to obtain a permutation that we apply on the sequences.

To be fair, there are some more technical details to our algorithm than just this but this is really its core component and by far the costliest operation. The details are in the paper.



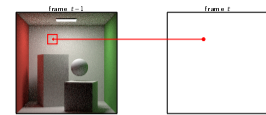
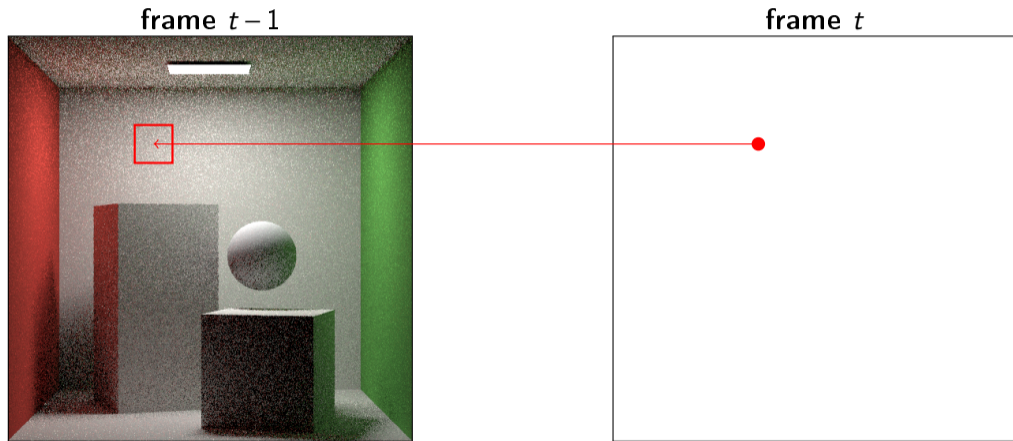
Temporal Algorithm

The hidden power of `std::sort` for Monte Carlo rendering.

The hidden power of `std::sort` for Monte Carlo rendering.

Let's look at some animated results. The animations are provided in the supplemental material.

Temporal Algorithm

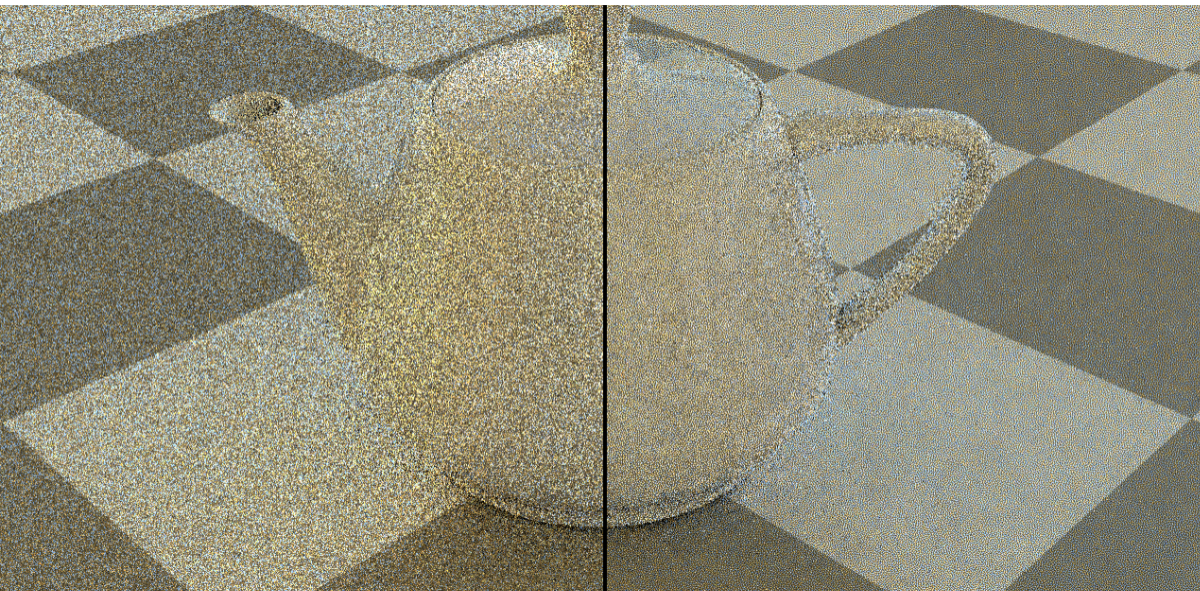


Spatio-temporal coherence: neighboring pixels in the previous frame are similar.

Note that our temporal algorithm makes the assumption that a sequence produces a similar value in neighboring pixels in the next frame. Of course, this is not always true: when the camera moves, or at the edge of an object, this assumption is violated.

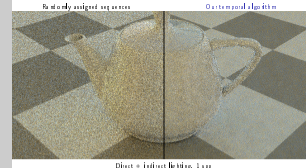
Spatio-temporal coherence: neighboring pixels in the previous frame are similar.

Randomly assigned sequences



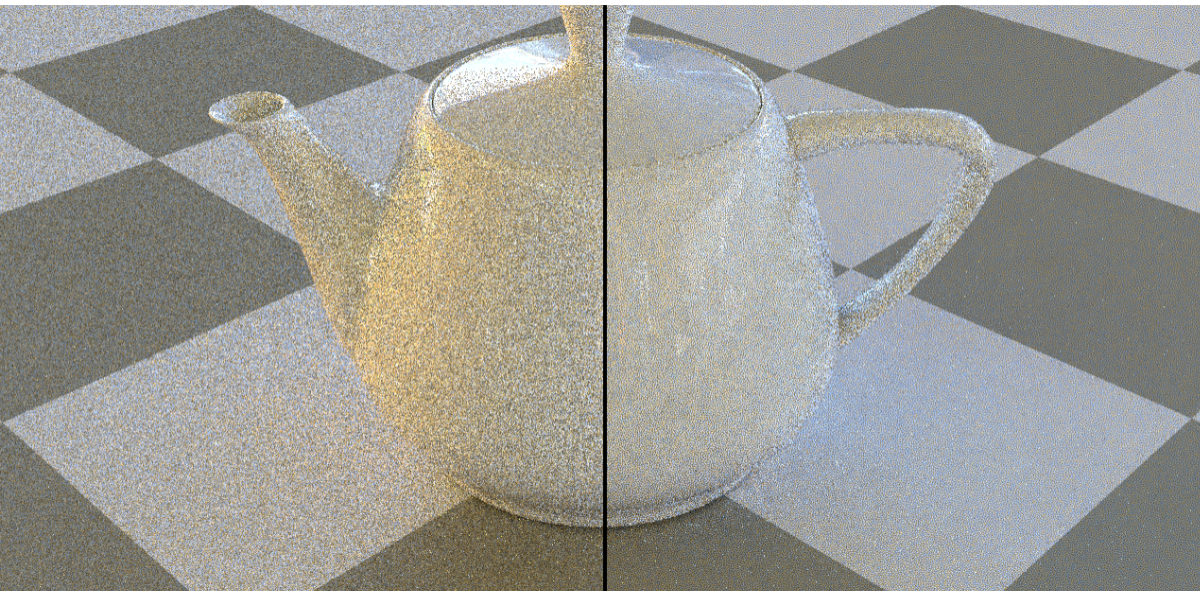
Direct + indirect lighting, 1 spp

Our temporal algorithm



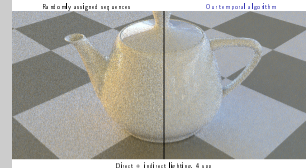
This image shows the consequences of the spatio-temporal similarity assumption. In the regions that are smooth and coherent (e.g. the checkerboard in the background or the center part of the teapot) the blue-noise distribution of the errors looks good. However, at the border of the teapot the error is poorly distributed, as a classic white noise.

Randomly assigned sequences



Direct + indirect lighting, 4 spp

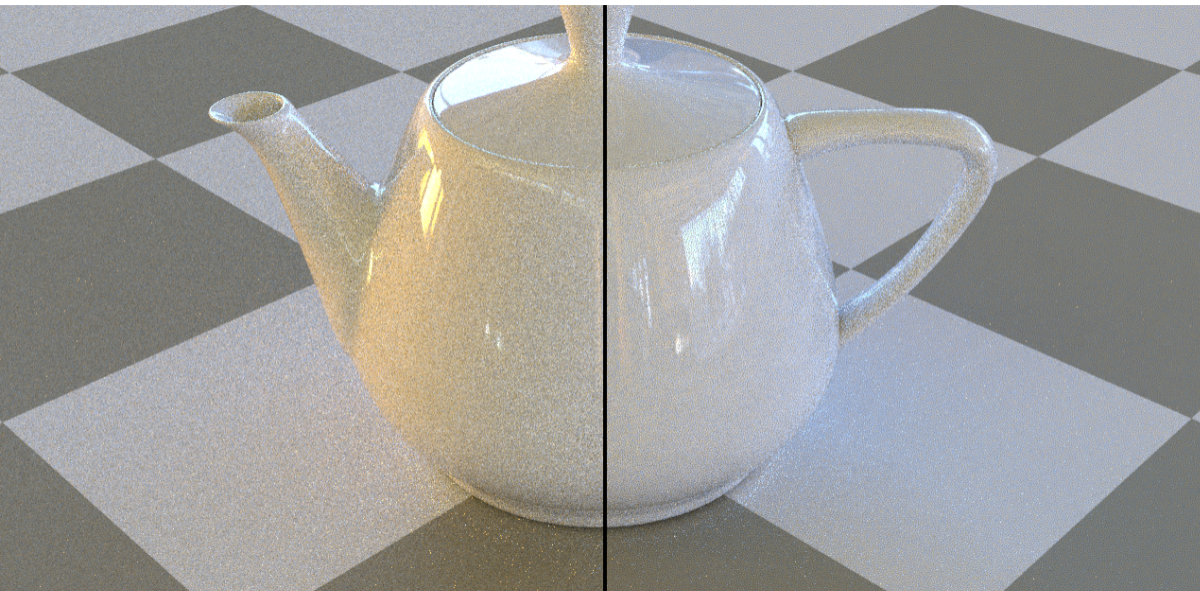
Our temporal algorithm



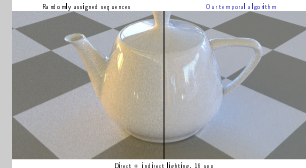
This image shows the consequences of the spatio-temporal similarity assumption. In the regions that are smooth and coherent (e.g. the checkerboard in the background or the center part of the teapot) the blue-noise distribution of the errors looks good. However, at the border of the teapot the error is poorly distributed, as a classic white noise.

Randomly assigned sequences

Our temporal algorithm



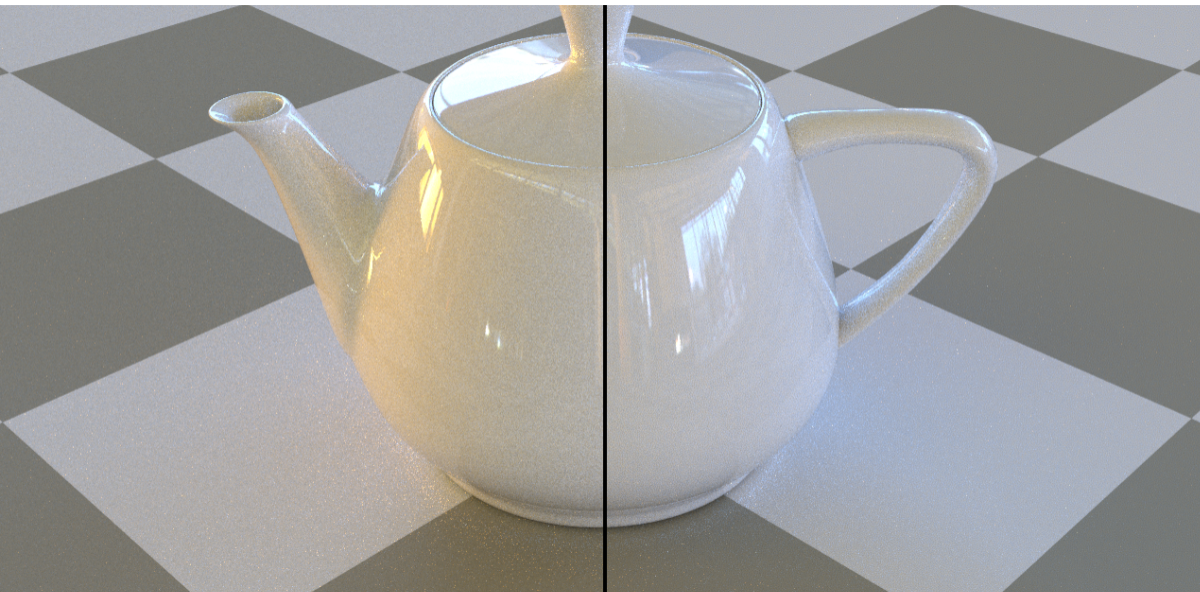
Direct + indirect lighting, 16 spp



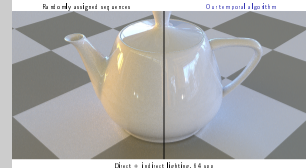
This image shows the consequences of the spatio-temporal similarity assumption. In the regions that are smooth and coherent (e.g. the checkerboard in the background or the center part of the teapot) the blue-noise distribution of the errors looks good. However, at the border of the teapot the error is poorly distributed, as a classic white noise.

Randomly assigned sequences

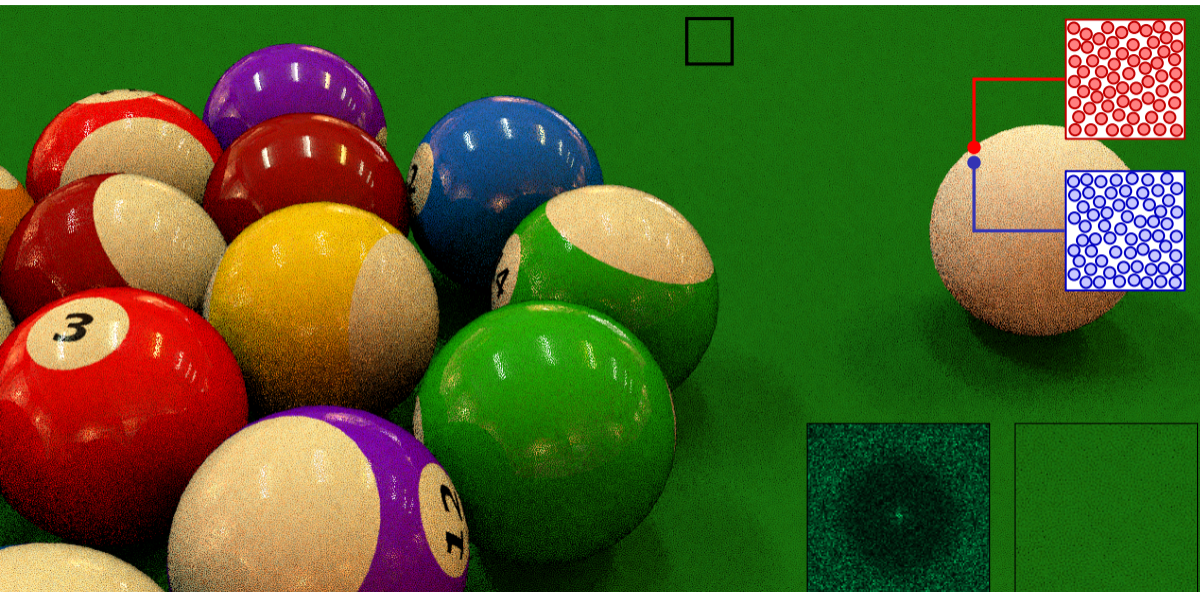
Our temporal algorithm



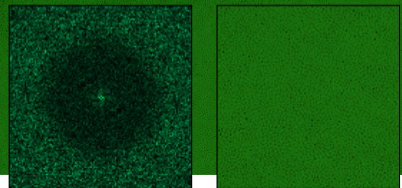
Direct + indirect lighting, 64 spp



This image shows the consequences of the spatio-temporal similarity assumption. In the regions that are smooth and coherent (e.g. the checkerboard in the background or the center part of the teapot) the blue-noise distribution of the errors looks good. However, at the border of the teapot the error is poorly distributed, as a classic white noise.

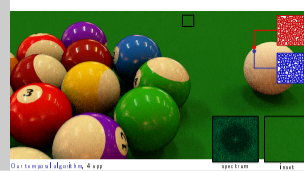


Our temporal algorithm, 4 spp



spectrum

inset



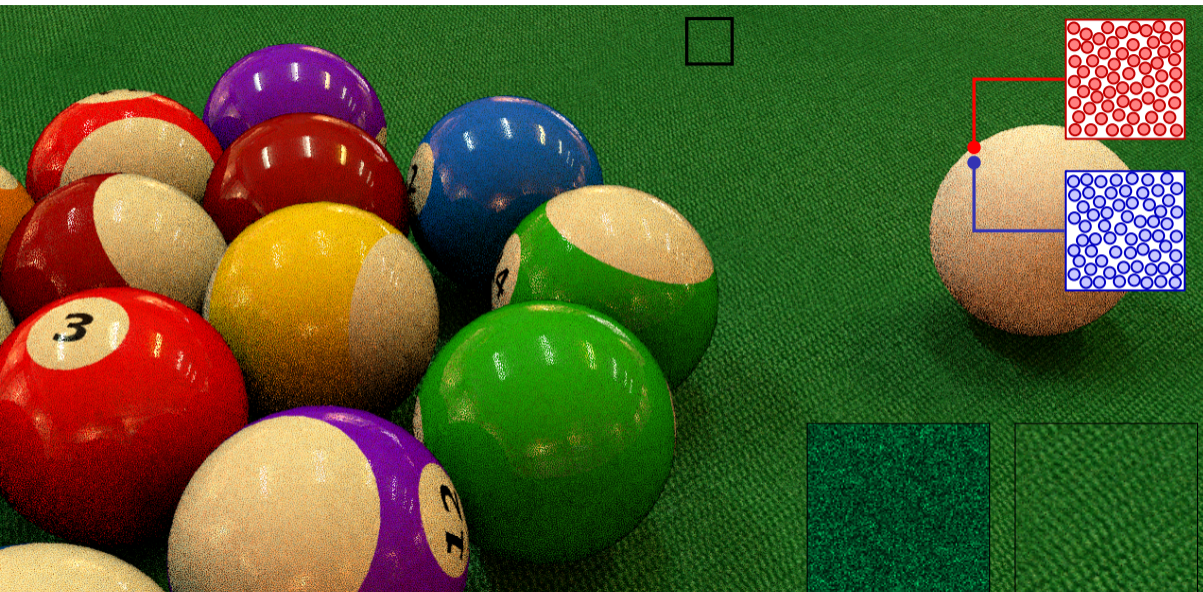
Our temporal algorithm, 4 spp

spectrum

inset

We applied our method on this snooker table with and without a texture.

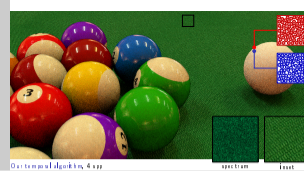
Without the texture, our temporal algorithm manages to distributed the errors as a blue noise on the table.



Our temporal algorithm, 4 spp

spectrum

inset



Our temporal algorithm, 4 spp

spectrum

inset

However, with a texture, our method fails. In this case, the errors are distributed as a white noise.

Temporal Algorithm

Quality

Spatio-temporally coherent regions: [blue-noise](#)

Spatio-temporally incoherent regions: white-noise (worst case = classic rendering)

Performance at 1080p

CPU Intel i7-5960X (1 thread): 1035 ms

CPU Intel i7-5960X (16 thread): 64 ms

GPU NVIDIA 2080: 0.60 ms

Main selling points: [negligible overhead](#) and [safe to use](#) (results can only be better).

Quality
Spatio-temporally coherent regions: [blue-noise](#)
Spatio-temporally incoherent regions: white-noise (worst case = classic rendering)

Performance at 1080p
CPU Intel i7-5960X (1 thread): 1035 ms
CPU Intel i7-5960X (16 thread): 64 ms
GPU NVIDIA 2080: 0.60 ms

Main selling points: [negligible overhead](#) and [safe to use](#) (results can only be better).

The main selling point of our algorithm is that it never worsen the images. In the best case, they are improved, in the worst case, we obtain a classic white-noise error distribution.

Furthermore, our algorithm is extremelly fast since it does nothing besides sorting small sorted lists. We originally meant it for offline rendering but it turned out it can also be considered for realtime rendering.

Given that our algorithm is safe to use and very cheap, there is almost no reason for not using it.

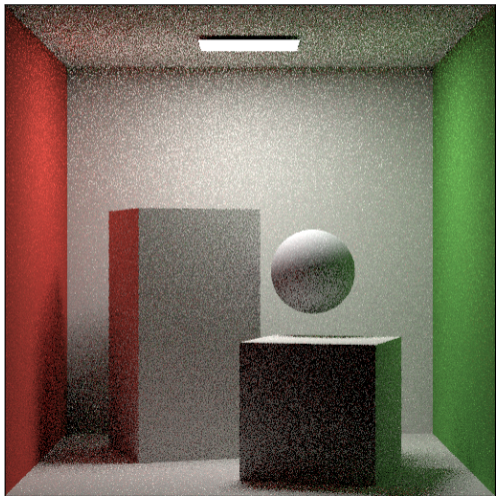
Are we there yet?

Are we there yet?

So... is this it or is there more to this research topic?

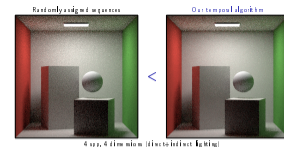
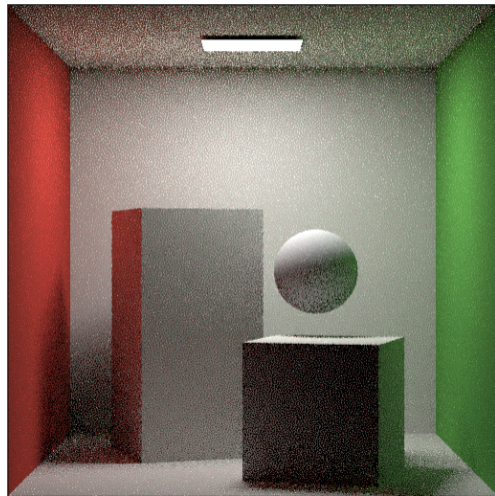
Are we there yet?

Randomly assigned sequences



4 spp, 4 dimensions (direct+indirect lighting)

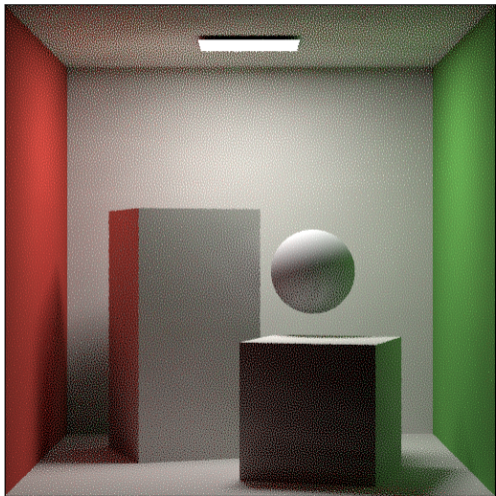
Our temporal algorithm



Our temporal algorithm does indeed increase the quality compared to a classic randomization.

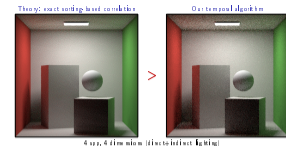
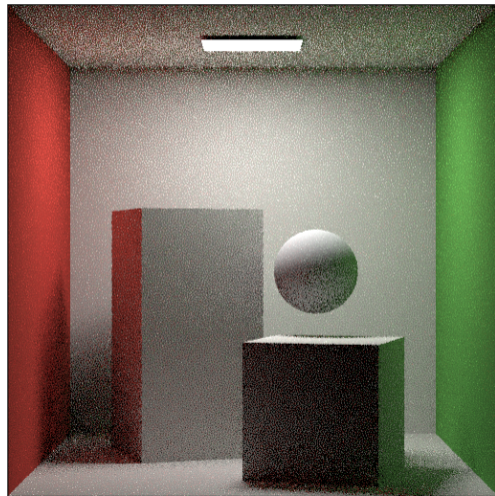
Are we there yet?

Theory: exact sorting-based correlation



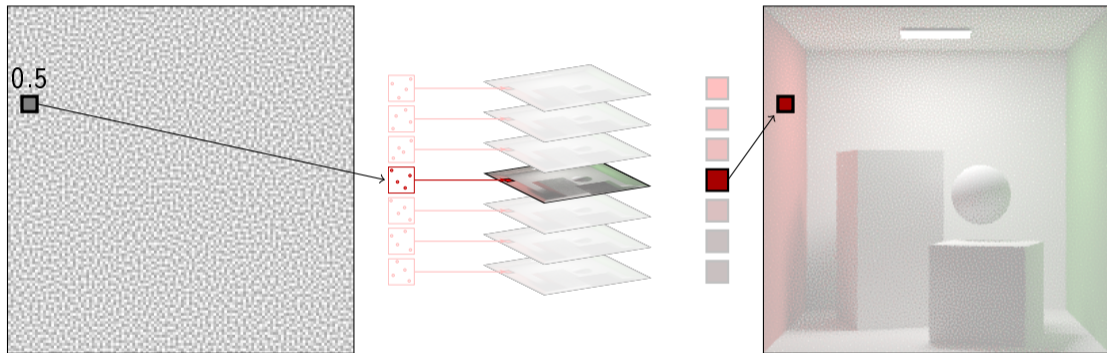
4 spp, 4 dimensions (direct+indirect lighting)

Our temporal algorithm



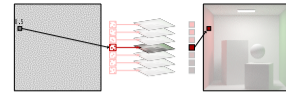
However, compared to the exact sorted-list formulation introduced before, the gap in quality is still large!

Are we there yet?



Theory: sort values computed for the **same** pixel.

Practice: sort values computed for **multiple neighboring** pixels in the **previous frame**.



Theory: sort values computed for the **same** pixel.

Practice: sort values computed for **multiple neighboring** pixels in the **previous frame**.

In theory, we should compute this sorted list for each pixel independently but in practice we approximated it by sharing pixel values over blocks of pixels from the previous frame. The difference between theory and practice is responsible for the difference in quality between the two previous images.

Are we there yet?

[Heitz&Belcour2019]

Theory: how to correlate MC errors directly regardless of the sample count and the dimensionality.

→ **The most interesting part of the paper!**

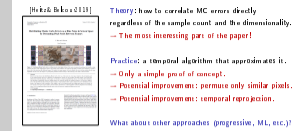
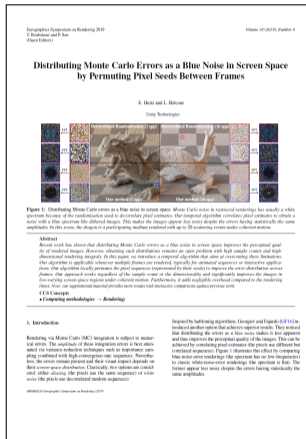
Practice: a temporal algorithm that approximates it.

→ **Only a simple proof of concept.**

→ **Potential improvement: permute only similar pixels.**

→ **Potential improvement: temporal reprojection.**

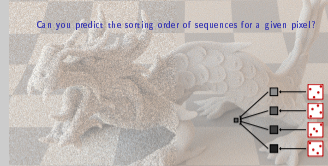
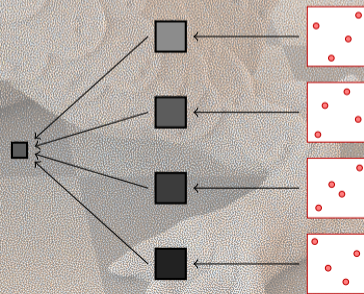
What about other approaches (progressive, ML, etc.)?



In our opinion, the most interesting part of our paper is the one that introduces the theoretical formulation based on the sorted list (or equivalently as a histogram in the paper). To show the potential of the idea, we wanted to make a simple proof of concept and this is what the temporal algorithm is for. It already produces some nice results but it is not hard to imagine some improvements such as a temporal reprojection or preventing permuting pixels that are not on the same objects, for instance.

We also believe that it should be possible to design non-temporal approaches. For instance, is it possible to design a progressive sequence construction that would rank as desired in the sorted list? Or is it possible to use Machine Learning to predict the sorting orders of multiple sequences for a given pixel? This problem is very open and we don't have any preconception of what would be the right approach.

Can you predict the sorting order of sequences for a given pixel?



As a conclusion, we would like to leave you with this question. If you can predict how sequences would perform on a pixel (you only have to predict the sorting orders, not the accurate values that they would produce) then you will be able to produce Monte Carlo rendered images with a terrific blue-noise distribution of their errors and this has the potential to bring the quality of your images to another level.

Thanks for your attention!