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

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

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

\[ f = \text{light transport} \]

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?

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?
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.
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.
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”.
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.
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.

Denoised blue-noise errors are better perceptually and numerically.
Random + denoising
Optimized + denoising

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.
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

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

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.
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.)
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.
Before talking about our algorithms, we would like to share some insights we gather regarding BNDS.
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?
Blue-noise Dithered Sampling [Georgiev&Fajardo2016]

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].

\[ \text{Heaviside}(I - U) \]

\[
\begin{align*}
\text{if}(\text{input} > \text{dither}) & \quad \text{return 1; } \\
\text{else} & \quad \text{return 0;}
\end{align*}
\]
Blue-noise Dithered Sampling [Georgiev & Fajardo 2016]

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

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

\(f(U_1, U_2)\)

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.
How to produce higher-dimensional blue-noise textures?

This is precisely what this paper is about. It introduces an algorithm that computes these textures for an arbitrary dimensionality $D$ (here $D=1,2,3$).
Blue-noise Dithered Sampling [Georgiev & Fajardo 2016]

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.
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 $(x_1, y_1), \ldots, (x_n, y_n)$ 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 of 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.”

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.
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.

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

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

We compute a pixel value using a sequence of random numbers.
How can we directly correlate Monte Carlo errors?

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?

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

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

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 & Fajardo 2016]

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.

1 spp, 4 dimensions (direct + indirect lighting)
How can we directly correlate Monte Carlo errors?

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!

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.

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

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

This is where our temporal algorithm comes into play.
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

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 $2 \times 2$ but in practice we always do $4 \times 4$).

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

Sort a block of $2 \times 2$ pixels from the previous frame. (Note: we use $4 \times 4$ in practice)
Temporal Algorithm

frame $t-1$

blue noise

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.

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.
Tempor al 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.
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.

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

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

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 extremely 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?

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

Randomly assigned sequences

Our temporal algorithm

4 spp, 4 dimensions (direct+indirect lighting)

Our temporal algorithm indeed increases the quality compared to a classic randomization.
Are we there yet?

Theory: exact sorting-based correlation

Our temporal algorithm

4 spp, 4 dimensions (direct+indirect lighting)

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

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!