High-Performance By-Example Noise using a Histogram-Preserving Blending Operator

Eric Heitz, Fabrice Neyret

High-Performance By-Example Noise using a Histogram-Preserving Blending Operator

ERIC HEITZ, Unity Technologies
FABRICE NEYRET, CNRS/LJK, Inria

Fig. 1. High-performance by-example noise. *We implement our algorithm in a fragment shader that requires no more than 4 texture fetches and a few computations, and can be efficiently integrated into a rendering engine.*

We propose a new by-example noise algorithm that takes as input a small example of a stochastic texture and synthesizes an infinite output with the same appearance. It works on any kind of random-phase inputs as well as on many non-random-phase inputs that are stochastic and non-periodic, typically natural textures such as moss, granite, sand, bark, etc. Our algorithm achieves high-quality results comparable to state-of-the-art procedural-noise techniques but is more than 20 times faster.

Our approach is conceptually simple: we partition the output texture space on a triangle grid and associate each vertex with a random patch from the input such that the evaluation inside a triangle is done by blending 3 patches. The key to this approach is the blending operation that usually produces visual artifacts such as ghosting, softened discontinuities and reduced contrast, or introduces new colors not present in the input. We analyze these problems by showing how linear blending impacts the histogram and show that a blending operator that preserves the histogram prevents these problems.

The main requirement for a rendering application is to implement such an operator in a fragment shader without further post-processing, i.e. we need a histogram-preserving blending operator that operates only at the pixel level. Our insight for the design of this operator is that, with Gaussian inputs, histogram-preserving blending boils down to mean and variance preservation, which is simple to obtain analytically. We extend this idea to non-Gaussian inputs by "Gaussianizing" them with a histogram transformation and "de-Gaussianizing" them with the inverse transformation after the blending operation. We show how to precompute and store these histogram transformations such that our algorithm can be implemented in a fragment shader.

2577-6193/2018/8-ART31 $15.00
https://doi.org/10.1145/3233304

CCS Concepts: • **Computing methodologies** → **Texturing**;

Additional Key Words and Phrases: noise synthesis, procedural texturing

ACM Reference Format:
Eric Heitz and Fabrice Neyret. 2018. High-Performance By-Example Noise using a Histogram-Preserving Blending Operator . *Proc. ACM Comput. Graph. Interact. Tech.* 1, 2, Article 31 (August 2018), 25 pages. https://doi.org/10.1145/3233304

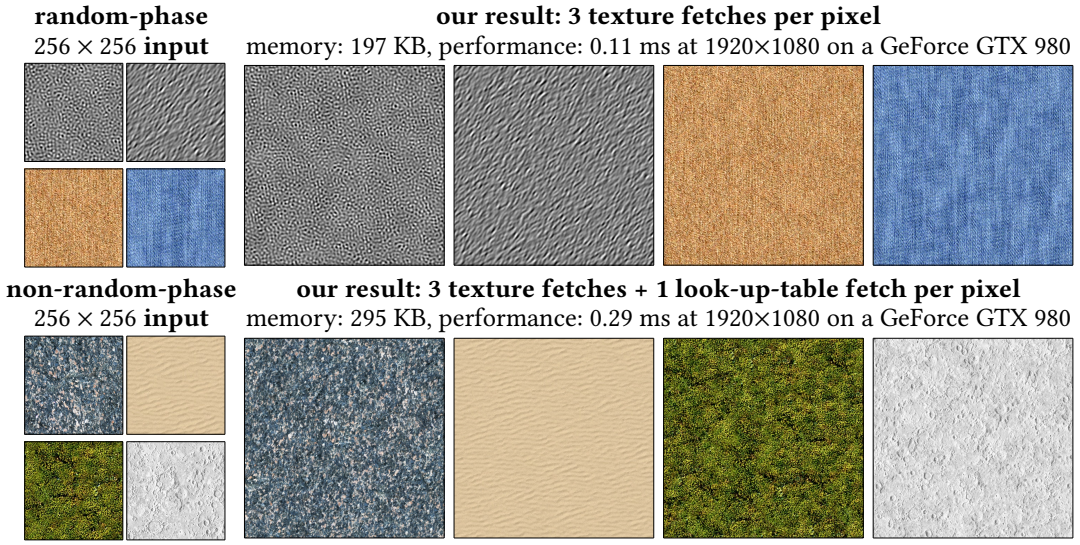

Fig. 2. Our method is 20× faster than Texton Noise [Galerne et al. 2017a], currently the fastest method for random-phase noise, and 75× faster than Local Random Phase Noise [Gilet et al. 2014], the state-of-the-art method for non-random-phase noise, and it achieves results of comparable quality.

1 INTRODUCTION

Automatic texture generation is required when a large amount of detailed textured surfaces are involved in a 3D scene. One approach is to build procedural textures from scratch using an analytic function such as Perlin [1985], Worley [1996] or Gabor noise [Lagae et al. 2009] and to let the user tune its parameters to reach a target aspect, which is a very difficult task for many real-world materials [Lagae et al. 2010]. An opposite approach consists of providing an example image and to resynthesize similar-looking textures [Wei et al. 2009]. Recently, several approaches have bridged the two worlds with *by-example noise* methods that generate noise with the same power spectrum density as a target texture [Galerne et al. 2012; Gilet et al. 2012]. These approaches come at the price of reduced performance since they build on the already costly Gabor noise and narrow the reachable texture space to *random-phase* noise, sometimes also called *Gaussian textures*.

The major research axes around by-example noise synthesis are thus *increasing speed* and *widening the generative space* that can be achieved by these methods. With respect to these criteria, we identify two state-of-the-art previous works. Regarding speed, *Texton Noise* [Galerne et al. 2017a] dramatically minimizes the cost of by-example random-phase noise synthesis, lowering it to about 30 texture fetches, and is currently the fastest method for this class of textures. On the other axis, *Local Random Phase Noise* [Gilet et al. 2014] remains the first attempt at synthesizing *non-random-phase* noise, i.e. it partially reproduces the structure present in the input if this structure can be identified as a periodic pattern.

In this paper, we propose a new by-example noise method that challenges these state-of-the-art previous works on their respective strengths:

- **Speed:** our method generates random-phase noise 20 times faster than *Texton Noise* [Galerne et al. 2017a], which is currently the fastest method for this class of textures. In the first row of Figure 2, we reproduce random-phase noise seen in recent previous works [Galerne et al. 2012, 2017a; Lagae et al. 2009] with unprecedented performance.
- **Generative space:** our method covers a wide set of non-random-phase input as shown in the second row of Figure 2. The richness of the generative space covered by our method is comparable to the one of *LRPN* [Gilet et al. 2014]. In general, our method achieves superior results on non-periodic and stochastic-looking inputs while LRPN works better with inputs that have a strong repetitive component. For this class of textures, our method is 75 times faster than LRPN. Furthermore, our method does not require any tuning as opposed to LRPN that has several arbitrary parameters to be set up by the user.

1.1 Tiling and blending

Our method achieves this performance because the evaluation of our noise consists of blending no more than 3 patches chosen randomly in the input using a dedicated blending operator, as shown in Figure 3. Besides performance, preserving the structure of the input is the main reason why we designed a method that fetches and blends the input only 3 times. Indeed, the fewer patches are blended, the more their structure is preserved.

The crux of the problem is in the ability to blend while preventing visual artifacts, such as ghosting or reduced contrast, that appear with linear blending, as shown in Figure 4(a). These artifacts reveal the underlying grid due to the variations of the blending weights. In Section 3, we explain the problems arising with linear blending in terms of their impact on the histogram of the input. What makes our approach viable is that we blend the patches in a way that guarantees that the histogram of the input is preserved in the output.

However, getting an acceptable and usable result is more complicated than just preserving the histogram. For instance, in Figure 4(b) we apply a histogram-equilization algorithm on the output of Figure 4(a). The result has the right global histogram but still exhibits local contrast variations that reveal the underlying grid. More importantly, such an approach is not practical in a rendering context, such as in Figure 1. Indeed, we aim to design a procedural texturing algorithm that works entirely in a fragment shader but the result of Figure 4(b) is obtained by applying a *global* histogram-transfer method, which is a post-process applied on the output texture. In the context of a rendering application, a procedural texture is computed on the fly in a fragment shader and cannot be post-processed.

To overcome this problem, in Section 3 we introduce a blending operator that preserves the histogram with this rendering-specific constraint in mind: the operation is entirely defined at the *pixel level*, i.e. we blend pixels inside a fragment shader such that its output directly belongs to the right histogram without requiring any further post-process. This new blending operator is the key to ensuring that the method can be implemented in a fragment shader without producing grid-revealing contrast variations, as in Figure 4(c). In Section 4, we propose an efficient implementation of our blending operator, which we use in Section 5 for our high-performance noise method. Finally, we discuss our results and compare them to the state-of-the art concurrent methods in Section 6.

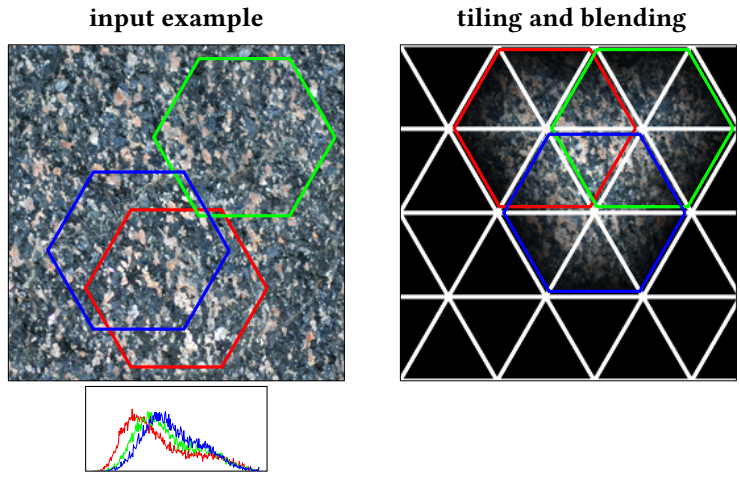

Fig. 3. Tiling and blending. *For each pixel of the output, we evaluate our noise by blending 3 patches randomly chosen from the input. We use a regular lattice that yields continuous blending weights and whose vertices are associated with random offsets in the input. The crux of the problem is to define a blending operator that prevents visual artifacts (see Figure 4).*

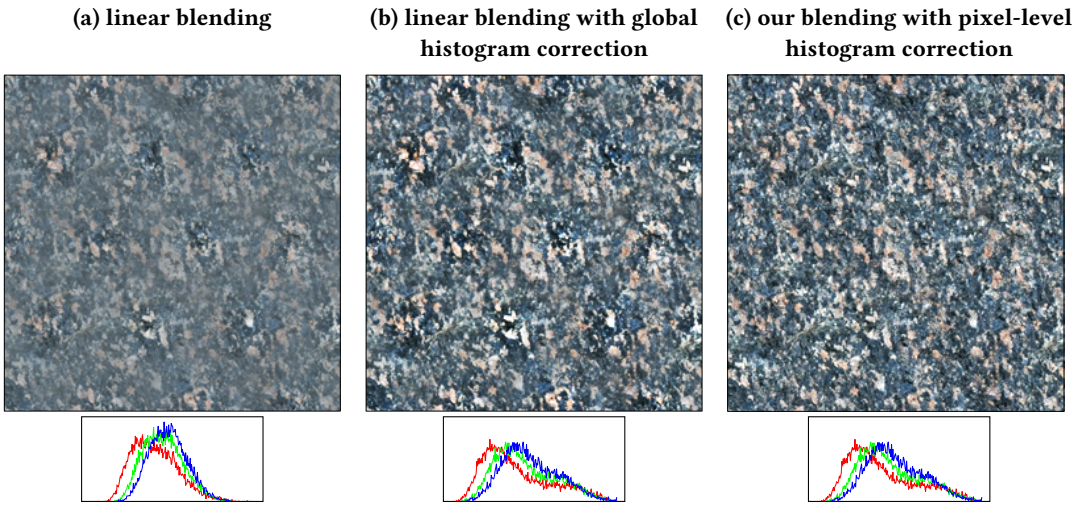

Fig. 4. Results of tiling and blending. *(a) Classic linear blending on a grid introduces non-stationary contrast that reveals the grid pattern. (b) Applying a global histogram correction on the result cannot be done in the context of real-time rendering and does not solve the non-stationarity problem: the grid-like artifacts are still visible. (c) Our blending operator ensures that the histogram is correct on a pixel-level basis. This prevents artifacts and makes the algorithm usable in a fragment shader.*

2 PREVIOUS WORK

Random-phase noises. Random-phase noises are defined entirely by their power spectrum density, i.e. by the amplitudes of their spectrum regardless of the phases. Since the structure information of an image is encoded in the phase, the generative space of random-phase noises is limited to structureless stochastic patterns [Lagae et al. 2010]. Gabor Noise [Lagae et al. 2009] was the first method to provide an accurate control over the spectrum of the noise and Gabor Noise by Example [Galerne et al. 2012] manages to automatize this control with an input example. However, evaluating these noises is costly when the power spectrum is complex since it is necessary to average many kernels to guarantee both good spatial and frequency coverage. This complexity was recently cut down by Texton Noise [Galerne et al. 2017a]. Its idea is to use a precomputed kernel, called a texton, that encodes all of the complexity of the target spectrum. This kernel is convolved with a field of randomly positioned points. Thanks to this, the cost of the method becomes independent of the complexity of the spectrum. However, the method requires 30 texture fetches which is still prohibitive for many real-time applications. Our method retains the idea of averaging samples from an input texture but brings the cost down to only 3 texture fetches. Thanks to this, it reproduces all of the aforementioned random-phase noises while being orders of magnitude faster. Furthermore, the generative space of our method is larger since it works on many non-random-phase textures.

Lattice noises vs. shot noises. Either analytic or image-based, a crucial difference between noise algorithms is their underlying structure: some interpolate or tile along lattices — this is typically how Perlin noise works [Perlin 1985, 2001, 2002] — while others splat multiple kernels along Poisson point distributions — typically, all of the sparse convolution methods [Galerne et al. 2012, 2017a; Lagae et al. 2009; Lewis 1984; van Wijk 1991]. This largely impacts the reconstruction cost, especially for analytic noises, which have to average many kernels. Unstructured schemes also use a more complicated data structure for accessing neighboring data contributing to the pixel. Acceleration structures draw on a lattice [Worley 1996] to organize the points neighborhood, and a multi-scale one for multi-frequency synthesis [Galerne et al. 2012]. But this still multiplies, by about 9 (in 2D), the number of convolution points to consider at render time. Interestingly, the Gabor Noise family of approaches originated from convolution textures, but the recent lattice-based formulation of Gilet et al. [2014] showed that the same result can be obtained faster and more simply. For all of these reasons, our method draws on a lattice as well, even if it also relates to convolution methods by blending several randomly offset primitives. More precisely, we use the equilateral-triangle lattice, or *simplex grid*, as in [Perlin 2001]. The advantage of this lattice is that evaluating the noise requires computing and blending the contributions of only 3 vertices instead of 4, which further improves performance.

Stitching, tiling, and patching by example. Our method is fundamentally different from typical tiling techniques that usually arrange precomputed textures according to some constraints that ensure continuity at the edges of the tiles [Cohen et al. 2003; Lagae and Dutré 2006; Neyret and Cani 1999; Vanhoey et al. 2013]. Texture tiling, even made aperiodic, shows spatial correlations because the same set of precomputed textures is repeated. The advantage of our method is that since we blend the patches, we do not have continuity constraints to meet. Moreover, thanks to our histogram-preserving blending operator, blending patches from the input example creates a continuous textural field with new content along each tile. Stitching source texture patches as in [Efros and Freeman 2001; Praun et al. 2000] only works for examples with many discontinuities in the content and for which repetitivity is not an issue, which is inappropriate for stochastic noise.

Preserving texture statistics. In terms of texture processes, there are several kinds of statistics that might be targeted for preservation. The shot noise method [Galerne et al. 2017a; Lagae et al. 2009; Lewis 1984] aims to control the power spectrum. It applies a multiplicative factor to the noise to obtain the desired variance, which is a constant related to the expected density of kernels. Our spatially varying, variance-preserving blending relates to the local variance-preserving blending scheme of Yu et al. [2011] for unstructured splats. We show that their approach works perfectly if the input has a Gaussian histogram, which fits the case of random-phase textures. We bring non-Gaussian histogram conservation to the existing set of tools by introducing a histogram transformation that first "Gaussianizes" the input example, so that we can rely on Yu et al.'s method when blending our patches.

Histogram transformations. Our "Gaussianization" process is equivalent to a *color transfer* that forces the histogram of the input to match a target histogram, such as the algorithm proposed by Reinhard et al. [2001]. Several options are possible for achieving this The most obvious one consists of applying three 1D histogram equalizations on each RGB channel. However, this does not account for inter-channel correlations, so wrong colors might appear in the result. Reinhard et al. [2001] use a color space in which the channels tend to be decorrelated for natural images. An improved option is obtained by effectively decorrelating the channels of the input by aligning them with the eigenvectors of its covariance matrix [Heeger and Bergen 1995]. However, while these methods work very well in practice, they remain approximate because decorrelated does not mean independent: correlation is only a first-order measure of dependence. The state-of-the-art approaches for histogram transfer are currently based on *Optimal Transport (OT)* for images [Bonneel et al. 2011; Morovic and Sun 2003] as well as for enriching stochastic textures [Galerne et al. 2017b]. Indeed, the optimal-transport plan between two histograms accounts for their full 3D shapes without introducing approximations. Our procedural-noise method is orthogonal to the choice of the histogram-transfer method and deciding which one to use is just an implementation choice. In this paper, we use the state-of-the-art optimal-transport solution.

3 HISTOGRAM-PRESERVING BLENDING

We begin this section by studying the effect of linear blending on the histogram through the statistical properties of the combination of random variables. We then describe our *histogram-preserving blending* operator, which is the key component of our procedural-noise algorithm.

Histogram and random variables. We consider a histogram $\mathcal{H}$ that is exactly a multivariate Probability Density Function (PDF) in $\mathbb{R}^d$ and N multivariate random variables $\mathbf{X}_1, .., \mathbf{X}_N$ that are independent and identically distributed in this histogram/PDF:

$$\mathbf{X}_1, .., \mathbf{X}_N \sim \mathcal{H}. \tag{1}$$

In the next section, the random points are distributed in 3D RGB space, but for this preliminary study we consider an arbitrary D-dimensional Cartesian space. We use $\mathbb{E}[\mathbf{X}]$ and $\mathbf{cov}[\mathbf{X}]$ to denote the expectation and the covariance of these random variables, respectively.

Weights. We consider N non-negative weights $w_1, .., w_N$ such that their sum is normalized:

$$\sum_{n=1}^{N} w_n = 1. \tag{2}$$

Histogram-preserving blending. The focus of the following sections is to find an operator that blends the random variables $\mathbf{X}_i$ with an importance driven by their respective weights w_i. The larger its weight, the more a variable should contribute to the result and a variable should not impact the

result at all if its weight is 0. Furthermore, we want the blending operator to be histogram-preserving, i.e. the result should also be distributed according to $\mathcal{H}$. The blending operators described in the next sections are compared in Figure 5.

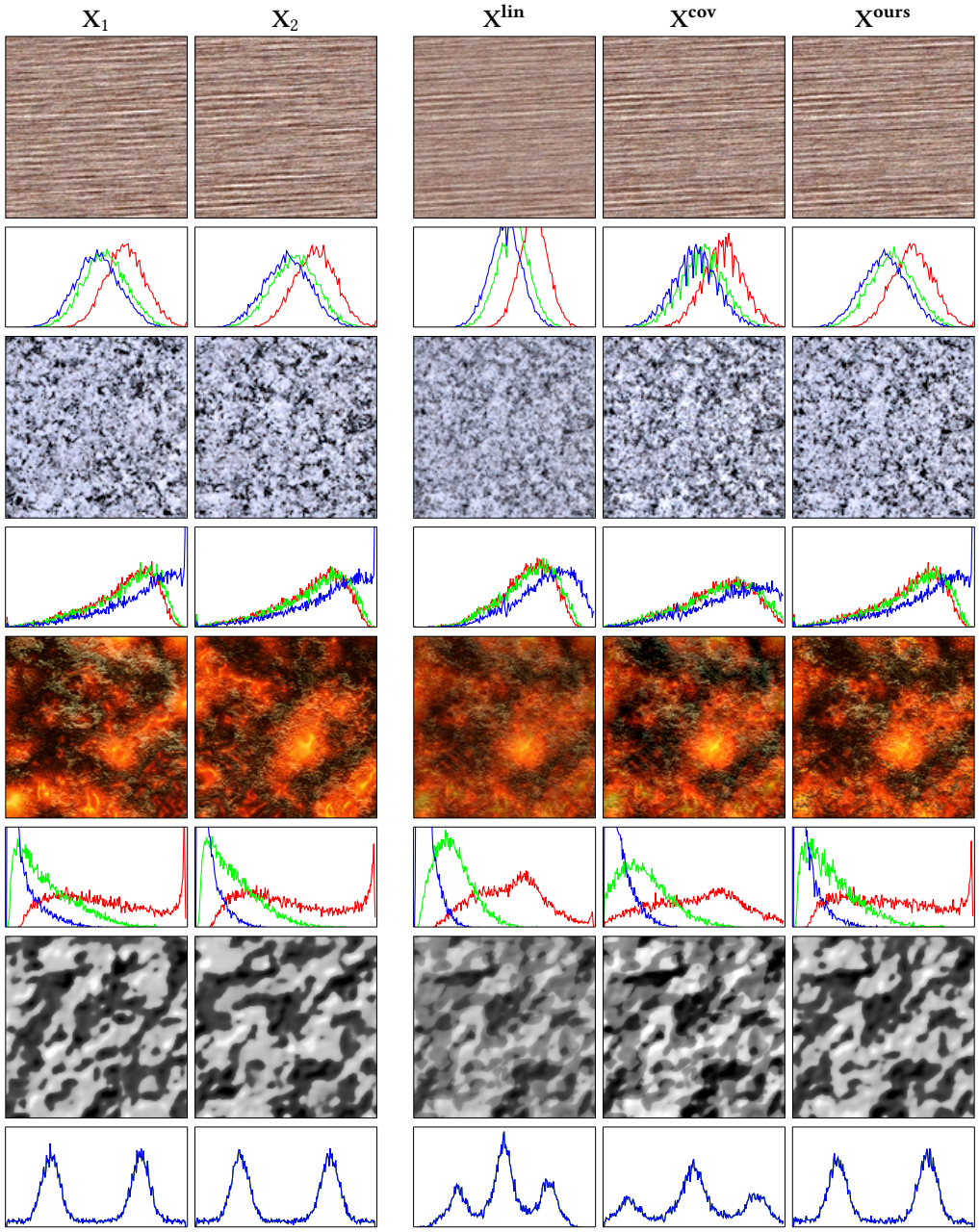

Fig. 5. Comparison of the blending operators. *We use the blending operators to blend each pixel of the two images,* $\mathbf{X}_1$ *and* $\mathbf{X}_2$*, with weights* $w_1 = w_2 = \frac{1}{2}$*.* $\mathbf{X}^{\boldsymbol{lin}}$ *is the classic linear blending (3.1),* $\mathbf{X}^{\boldsymbol{cov}}$ *is the variance-preserving blending (3.2), and* $\mathbf{X}^{\boldsymbol{ours}}$ *is our histogram-preserving blending (3.3).*

3.1 Recap on Linear Blending

The simplest way to blend the random variables is to compute their weighted average:

$$\mathbf{X}^{\mathbf{lin}} = \sum_{n=1}^{N} w_n \, \mathbf{X}_n. \tag{3}$$

Properties of the linearly blended result. Since the sum of the weights is 1, the linear blending operator preserves the expectation:

$$\mathbb{E}[\mathbf{X}^{\mathbf{lin}}] = \mathbb{E}[\mathbf{X}]. \tag{4}$$

Since the input samples are chosen independently, the variance of the expected histogram is the variance of the input scaled by the sum of their squared weights:

$$\mathbf{cov}\left[\mathbf{X}^{\mathbf{lin}}\right] = W^2 \, \mathbf{cov}\left[\mathbf{X}\right], \tag{5}$$

$$W = \sqrt{\sum_{n=1}^{N} w_n^2}. \tag{6}$$

Finally, since the variables are independent, their linear combination has the effect of convolving their histograms:

$$\begin{aligned} \mathcal{H}^{\mathbf{lin}}(\mathbf{X}) &= \mathcal{H}_1(\mathbf{X}) * \ldots * \mathcal{H}_N(\mathbf{X}) \\ &= \frac{1}{w_1} \mathcal{H}\left(\frac{\mathbf{X}}{w_1}\right) * \ldots * \frac{1}{w_N} \mathcal{H}\left(\frac{\mathbf{X}}{w_N}\right) \end{aligned} \tag{7}$$

Note that if some of the weights evaluate to 0, the associated histogram can just be removed from the above expression because $\lim_{w \to 0} \mathcal{H}\left(\frac{\mathbf{X}}{w}\right) / w = \delta(\mathbf{X})$, which is the identity element for the convolution.

Discussion. As predicted by the Central limit theorem, averaging multiple independent variates impacts the histogram in two ways: it reduces its variance; the effect of these multiple convolutions is that they do not preserve complex histogram features such as discontinuities or multimodality. This results in several problems that can be observed in Figure 5. Because of the variance reduction, the linear blending operator $\mathbf{X}^{\mathbf{lin}}$ lowers the contrast. Furthermore, due to the changed shape of the histogram, the result might contain new colors that were not present in the original data. This is especially visible in the third example where a greenish color appears (zoom in the left-bottom corner) and in the last example where a new tone of gray results in particularly visible ghosting artifacts.

3.2 Recap on Variance-Preserving Blending

Linear blending preserves the expectation of the random variables but changes their histogram. A straightforward improvement consists of rescaling the result around the expectation in order to obtain the correct variance:

$$\mathbf{X}^{\mathbf{cov}} = \frac{\sum_{n=1}^{N} w_n \, \mathbf{X}_n - \mathbb{E}[\mathbf{X}]}{W} + \mathbb{E}[\mathbf{X}]. \tag{8}$$

This operator has been used by Yu et al. [2011] for blending texture sprites attached to moving particles.

Properties. By construction, this blending preserves both the expectation:

$$\mathbb{E}[\mathbf{X}^{\mathbf{cov}}] = \mathbb{E}[\mathbf{X}], \tag{9}$$

and the covariance:

$$\mathbf{cov}[\mathbf{X}^{\mathbf{cov}}] = \mathbf{cov}[\mathbf{X}]. \tag{10}$$

However, its histogram is the histogram of $\mathbf{X}^{\mathbf{lin}}$ scaled by the variance factor W:

$$\mathcal{H}^{\mathbf{cov}}(\mathbf{X}) = W\,\mathcal{H}^{\mathbf{lin}}(W\ (\mathbf{X} - \mathbb{E}[\mathbf{X}]) + \mathbb{E}[\mathbf{X}]). \tag{11}$$

Hence, this blending operation does not prevent the shape of the histogram from being affected by the multiple convolutions.

Exact histogram preservation in the Gaussian case. In the special case of random variables $\mathbf{G}_1, .., \mathbf{G}_N$ independent and identically distributed according to a Gaussian distribution $\mathcal{G}$, i.e.

$$\mathbf{G}_1, .., \mathbf{G}_N \ \sim\ \mathcal{G}, \tag{12}$$

the variance-preserving blending happens to be also exactly histogram-preserving:

$$\mathbf{G}^{\mathbf{cov}} = \left(\frac{\sum_{n=1}^{N} w_n\,\mathbf{G}_n - \mathbb{E}[\mathbf{G}]}{W} + \mathbb{E}[\mathbf{G}]\right) \quad \Rightarrow \quad \mathbf{G}^{\mathbf{cov}} \ \sim\ \mathcal{G}. \tag{13}$$

Indeed, the Gaussian distribution is stable under convolution. It means that the result of the blending is going to be distributed as a Gaussian. Furthermore, a Gaussian distribution is entirely determined by its expectation and covariance matrix and the variance-preserving blending guarantees that the expectation and the covariance are preserved. The result is thus going to be distributed exactly in the same Gaussian distribution.

Discussion. In Figure 5, the variance-preserving blending operator $\mathbf{X}^{\mathbf{cov}}$ produces a perfect result on the **wood** example because the input and the blended data have a Gaussian histogram and the variance-preserving blending is also histogram-preserving in this case. However, in general, it overshoots (it creates too dark and too bright pixels) as visible in the second and last examples. Furthermore, it does not prevent wrong colors or ghosting from appearing in the result. In conclusion, variance-preserving blending is an improvement over linear blending that happens to solve the Gaussian case perfectly. However, it does not address the general case satisfactorily.

3.3 Our Histogram-Preserving Blending Operator

Our idea for achieving a histogram-preserving blending operator is to transform the general problem such that the special case of Equation (13) applies. More specifically, we transform the random variables to make them Gaussian so that the variance-preserving blending preserves this Gaussian distribution, then we apply the inverse histogram transformation to obtain the original histogram.

Histogram transformation. Let us consider an operator $\mathcal{T}$ that maps random variables $\mathbf{X}$ to random variables $\mathbf{G} = \mathcal{T}(\mathbf{X})$, such that if $\mathbf{X}$ is distributed in histogram $\mathcal{H}$ then $\mathbf{G}$ is distributed in a Gaussian histogram $\mathcal{G}$, i.e.

$$\mathbf{X} \ \sim\ \mathcal{H} \quad \Rightarrow \quad \mathcal{T}(\mathbf{X}) \ \sim\ \mathcal{G}, \tag{14}$$

and that the mapping is invertible:

$$\mathbf{G} \ \sim\ \mathcal{G} \quad \Rightarrow \quad \mathcal{T}^{-1}(\mathbf{G}) \ \sim\ \mathcal{H}. \tag{15}$$

The effect of such an operator is to "Gaussianize" the input image, as shown in Figure 6. For now, we assume that such an operator $\mathcal{T}$ exists and can be computed. We discuss the choice and the implementation of such an operator in Section 4.

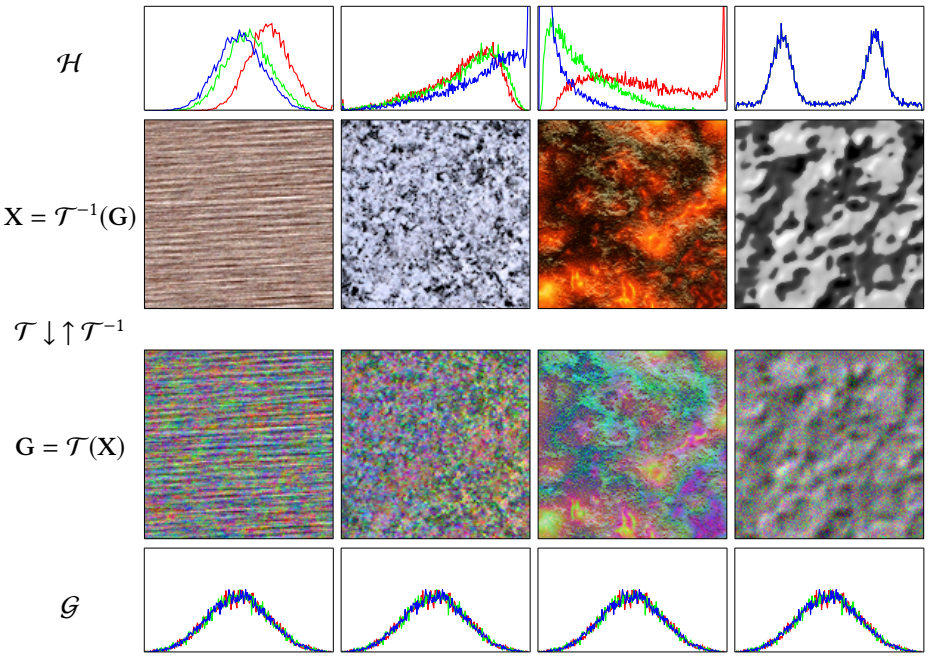

Fig. 6. Histogram transformation with the operator $\mathcal{T}$.

Histogram-preserving blending. Our blending operates in the following way. First, we use the transformation to make the random variables follow a Gaussian distrbution:

$$\mathbf{G}_n = \mathcal{T}(\mathbf{X}_n) \quad \Rightarrow \quad \mathbf{G}_n \sim \mathcal{G}, \tag{16}$$

on which we apply the variance-preserving blending operator of Equation (13), such that the result is distributed in the same Gaussian distribution: $\mathbf{G}^{\mathbf{cov}} \sim \mathcal{G}$. Finally, we apply the inverse transformation on the result to obtain a random variable from the original histogram:

$$\mathbf{X}^{\mathbf{ours}} = \mathcal{T}^{-1}(\mathbf{G}^{\mathbf{cov}}) \quad \Rightarrow \quad \mathbf{X}^{\mathbf{ours}} \sim \mathcal{H}. \tag{17}$$

The complete expression of our blending operator is

$$\mathbf{X}^{\mathbf{ours}} = \mathcal{T}^{-1}\left(\frac{\sum_{n=1}^{N} w_n \mathcal{T}(\mathbf{X}_n) - \mathbb{E}[\mathbf{G}]}{W} + \mathbb{E}[\mathbf{G}]\right). \tag{18}$$

Computation. Note that the only information required by our operator is the histogram transformation $\mathcal{T}$ and its inverse $\mathcal{T}^{-1}$. If these transformations are available, then our operator works the same for any number of points and any values for the weightings. The problem is thus to find such a transformation $\mathcal{T}$.

Discussion. In Figure 5, our histogram-preserving blending operator $\mathbf{X}^{\mathbf{ours}}$ produces the best results as all of the colors from the result are present in the input. The improvement is most clearly visible in the last example, because its histogram is bimodal. In this case, forcing the histogram to be correct prevents the ghosting artifacts that appear with the other blending options. Furthermore, it also prevents the appearance of parasitic colors, such as in the third example.

4 HISTOGRAM TRANSFORMATIONS

Our histogram-preserving blending operator in Equation (18) is based on a histogram transformation $\mathcal{T}$ and its inverse $\mathcal{T}^{-1}$, i.e. the ability to "Gaussianize" an input image and reverse this transformation, as shown in Figure 6. In this section, we explain how to apply $\mathcal{T}$ on the input (Section 4.2) and bake its inverse $\mathcal{T}^{-1}$ into a look-up table (Section 4.3).

Choosing a histogram transformation. As explained in Section 2 (*histogram transformations*), many options are available to compute a histogram transformation. Choosing one or another is orthogonal to our procedural-noise method presented in Section 5. In this paper, we choose to use the highest-quality method available in the state of the art, which is based on *Optimal Transport (OT)*. This method prevents potential artifacts due to histogram approximations and does not come with any drawbacks except for longer precomputation time (see Table 1), which we deem acceptable. The reader can trivially replace this method by a simpler one if desired.

4.1 Optimal-Transport Formulation

We consider an input image whose pixel values are noted $\mathbf{X}_{i,j}$ and that has a histogram $\mathcal{H}$. Our goal is to find a transformation $\mathcal{T}$ of this image that "Gaussianizes" it, i.e. that produces an image whose pixels $\mathbf{G}_{i,j} = \mathcal{T}(\mathbf{X}_{i,j})$ have a Gaussian histogram $\mathcal{G}$ and whose image-space structure is preserved as much as possible, i.e. the Gaussian version of the image has the same recognizable features, as in the results of Figure 6. We formulate this constraint as an L^2 error between the input and the result:

$$\text{error} = \min_{\mathcal{T}} \sum_{i,j} \left(\mathcal{T}(\mathbf{X}_{i,j}) - \mathbf{X}_{i,j}\right)^2 . \tag{19}$$

The error of Equation (19) is exactly the L^2 *Earth Mover's distance* between the input image $\mathbf{X}$ and the resulting image $\mathcal{T}(\mathbf{X})$, and the mapping $\mathcal{T}$ that minimizes this error can be solved using an *Optimal Transport (OT)* solver such as the *Fast Transport* library developed by Bonneel [2016].

4.2 Computation of the Transformation

Our Gaussianization of the input image works conceptually in the following way: we generate an image of random RGB values distributed as a Gaussian, and we permutate its pixels such that it has the same structure as the input image. The best permutation is what the Lagrangian optimal-transport solver computes.

Recap on sampling Gaussian random variables. The following steps require generating random variables from a Gaussian distribution. We use the 3D Gaussian distribution of average value $\frac{1}{2}$ and variance $\frac{1}{6^2}$ in each direction, such that the random values fit well on the interval $[0, 1]$ with 8-bit precision. Since this multivariate Gaussian distribution is separable, each dimension can be sampled separately. For this purpose, we use the inverse-transform method [Devroye 1986, Section 2.1: The inversion method], i.e. for each RGB component we generate a uniform random number $\mathbf{U} \in [0, 1]$ to which we apply the inverse of the Cumulative Distribution Function (CDF) of this Gaussian distribution:

$$\mathbf{G} = \frac{1}{2} + 6\sqrt{2}\,\text{erfinv}\,(2\,\mathbf{U} - 1) . \tag{20}$$

Using this equation, we generate an image of random Gaussian variables $\mathbf{G}_{k,l}$ of the same size as the input image. The question is how to choose their respective pixel coordinates k, l to obtain a Gaussian image that has the same structure as the input.

Organizing the Gaussian pixel values. With this discretized formulation, we interpret $\mathcal{T}$ as a permutation of the Gaussian pixel coordinates and we rewrite Equation (19):

$$\text{error} = \min_{\mathcal{T}} \sum_{i,j} \left(\mathbf{G}_{k,l=\mathcal{T}(i,j)} - \mathbf{X}_{i,j}\right)^2. \tag{21}$$

This problem is illustrated with a 2D example in Figure 7. The blue points are the RGB pixel values $\mathbf{X}_{i,j}$ of the input image and the red points are the Gaussian random variables $\mathbf{G}_{k,l}$ that we need to pair up with the pixel values. Minimizing the error in Equation (21) means finding a one-to-one mapping $(k,l) = \mathcal{T}(i,j)$ that minimizes the sum of the squared distances between the red-blue pairs. This is precisely what a Lagrangian optimal-transport solver computes. The mapping computed by the solver thus yields the pixel coordinates of the random Gaussian variables and we obtain the Gaussianized version of the image as in Figure 6.

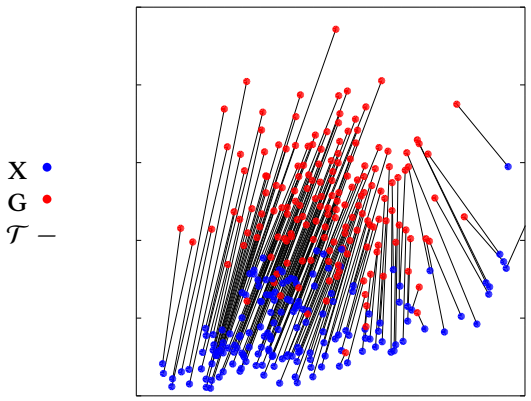

Fig. 7. *Lagrangian optimal transport in RGB space.* This illustration is 2D but the actual computations occur in 3D RGB space where each axis is associated with one color channel. The pixel values of the input image correspond to a point set in RGB space (blue points) that we map to a point set obtained by sampling a Gaussian distribution (red points). We use an optimal-transport solver to compute the mapping that minimizes the L^2 error.

4.3 Computation of the Inverse Transformation

The last step of our histogram-preserving blending in Equation (17) is the inversion of the histogram transformation to obtain a blended image that has the same histogram $\mathcal{H}$ as the input image. i.e. we consider an image of blended Gaussian values $\mathbf{G}^{\mathbf{cov}}_{i,j}$ to which we want to apply the inverse transformation $\mathcal{T}^{-1}$.

Pseudo-inverse transformation. We cannot directly reuse and invert the discrete mapping $\mathcal{T}$ that is the solution of Equation (21) because the blended Gaussian values $\mathbf{G}^{\mathbf{cov}}_{i,j}$ are not the same as $\mathbf{G}_{i,j}$: they are statistically distributed in the same Gaussian distribution but they are not exactly the same point set. We will thus use the optimal transport solver a second time on another point set to compute the inverse $\mathcal{T}^{-1}$. Because of this difference, $\mathcal{T}^{-1}$ is not strictly the inverse of $\mathcal{T}$. However, this is only due to the discrete formulation of our problem. In the continuous limit of our discrete formulation, this transformation is exactly the inverse of $\mathcal{T}$. In practice, since we use point sets of tens of thousands of points, the numerical difference is negligible.

Direct approach. A direct approach for inverting the transformation could be to use the optimal transport solver a second time to compute the best mapping between the blended Gaussian values to RGB values chosen randomly from the input image: $\mathcal{T}^{-1}\left(\mathbf{G}^{\mathbf{cov}}_{i,j}\right) = \mathbf{X}_{k,l}$ in the same way as in Equation (21). This approach works in theory but it cannot be performed on the fly, thus it is not suitable for real-time rendering.

Efficient approach using a precomputed look-up table. In order to obtain an efficient evaluation of the inverse transformation $\mathcal{T}^{-1}$ at run time, we precompute a mapping and store it in a 3D look-up table that can be fetched at run-time. More specifically, the look-up table maps 3D Gaussian values to RGB values $\mathbf{X}$ from the input image. We parameterize the look-up table with the inverse-transform mapping: the coordinates of each cell of the table yield a 3D random uniform value in $[0,1]^3$ that we use to sample a 3D Gaussian value using Equation (20). We thus obtain a point set of Gaussian random points in RGB space and we use the optimal-transport solver to map them to the same number of pixel values $\mathbf{X}$ chosen randomly from the input image. We store the values $\mathbf{X}$ in the cells of the look-up table given by the optimal-transport mapping.

Run time. At run time, we use the blended Gaussian value $\mathbf{G}^{\mathbf{cov}}$ to fetch the look-up table by inverting the parameterization of Equation (20), i.e. for each RGB component of $\mathbf{G}^{\mathbf{cov}}$ we compute

$$\mathbf{U} = \frac{1}{2} + \frac{1}{2}\,\mathrm{erf}\left(\frac{\mathbf{G}^{\mathbf{cov}} - \frac{1}{2}}{6\sqrt{2}}\right). \tag{22}$$

and we fetch the look-up table at the 3D location $\mathbf{U}$, which returns a value $\mathbf{X}^{\mathbf{ours}}$ from the histogram $\mathcal{H}$ of the input image.

Choosing a size for the look-up table. We always use a 32^3 look-up table. While a width of 32 texels might appear too low to encode the dynamic of the input, the 32^3 look-up table is actually large enough because we use the inverse-transform parameterization of Equation (20) that is *area-preserving* [Devroye 1986]. It means that the look-up table has no dead spots: all the cells are fetched with the same probability such that the dynamics of the Gaussian histogram is spread *uniformly* over the $32^3 = 32768$ cells. In practice, this is more than enough for a high-quality result.

5 HIGH-PERFORMANCE PROCEDURAL NOISE

In this section, we develop the implementation of our high-performance noise following the ideas developed in the previous sections.

5.1 Overview of our Method

Figure 8 provides an overview of our method.

Tiling and splatting scheme. We use the equilateral-triangle lattice and weighting scheme that was introduced in Simplex noise [Perlin 2001]. With this partitioning of the texture space, each vertex is associated with a hexagonal tile chosen randomly in the input image such that each point is covered by exactly 3 tiles. Each tile is weighted by a function falling to 0 at the borders and such that the sum of the 3 weights equals 1 everywhere. With this tiling scheme, each pixel requires exactly 3 texture fetches to be blended.

Blending. We blend the 3 tiles in a way that preserves the histogram of the input image. For this, we precompute a Gaussianized version of the input image and we represent the inverse transformation $\mathcal{T}^{-1}$ as a precomputed look-up table, as explained in Sections 4.2 and 4.3. These 3 Gaussian samples are thus blended using the variance-preserving operator of Equation (13), and converted back to source histogram with one look-up in table $\mathcal{T}^{-1}$.

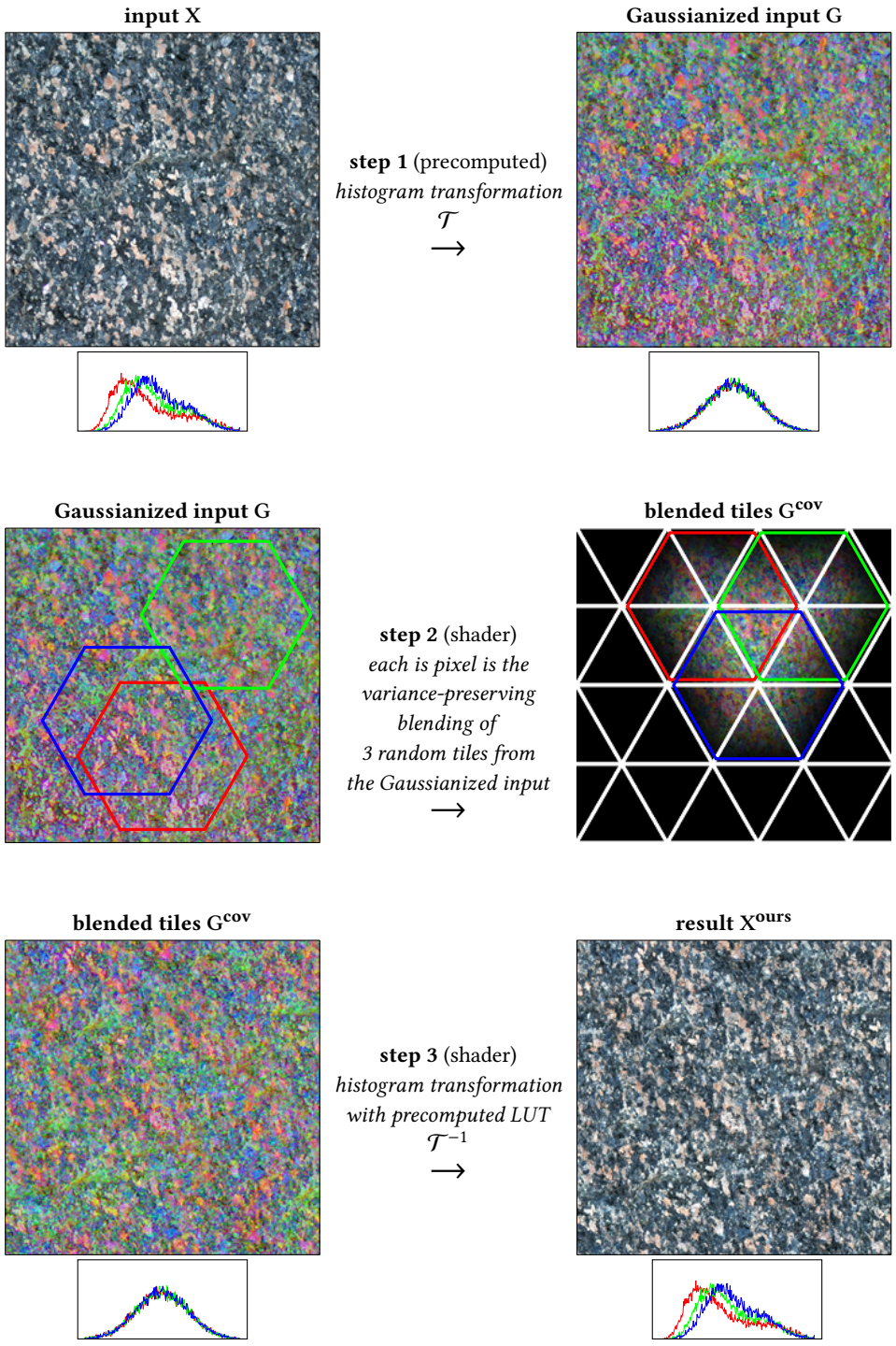

Fig. 8. Overview of our procedural-noise method.

5.2 Precomputations and Storage

Following Sections 4.2 and 4.3, we precompute and store two sets of data: the Gaussian version **G** of the input for **step 1** and the look-up table that encodes $\mathcal{T}^{-1}$ for **step 3**. For all our results, we use an input image of size 256^2 and a look-up table of size 32^3, as summarized in Table 1. Note that we use a single-threaded and unoptimized optimal-transport solver, so we might obtain a significant speed up with an optimized, multi-threaded library [Bonneel 2016].

symbol	name	type	size	precision	memory	time
G	**texGaussian**	2D texture	256^2	8-bit RGB	197KB	4 min
$\mathcal{T}^{-1}$	**lookUpTable**	3D texture	32^3	8-bit RGB	98KB	1 min

Table 1. Precomputations and storage. The timings are provided for a Intel(R) Core(TM) i7-5960X CPU.

5.3 Stationarity Condition on the Input

An assumption of our method is that all the relevant statistical and frequency content of the input is captured by the tiles, i.e. we assume that the input is stationary at the scale of a tile. We typically use tiles with a radius approximately half the width of the input, as in Figure 8, which means that the input should not have frequencies of periods larger than this size. In the example of Figure 9, the bark example on the left has intensity variations that cannot be captured by the size of the tile. This results in an undesirable low-frequency pattern. The input should thus be chosen or modified such it meets this condition. The second input in Figure 9 has been computed by removing the low frequencies from the first input. With this modification we obtain a satisfying result.

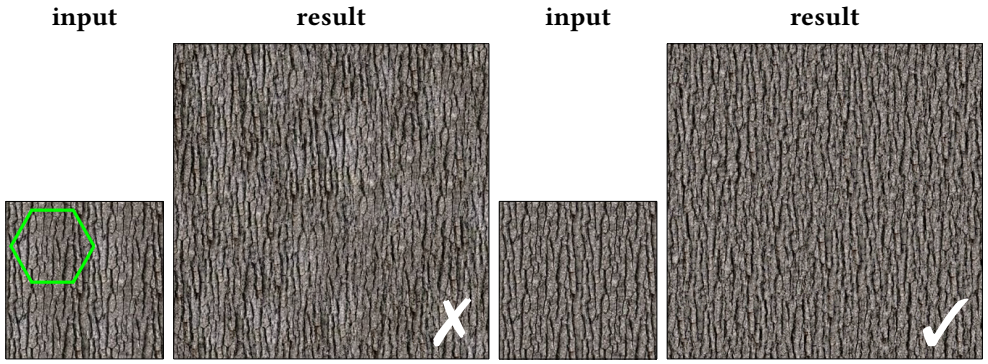

Fig. 9. Stationarity condition on the input. *We assume that the frequency content of the input is captured by the size of a tile. If this assumption is not met, our result contains an undesirable low-frequency pattern.*

5.4 Run time

We assume that noise is evaluated in a fragment shader for a given pixel position, and more precisely, invoked as **noise**(**uv**) where **uv** is some standard texture parameterization. In the following, we explain the computations of our shader implementation summarized in Algorithm 1.

Algorithm 1 Fragment shader at run time

Input: texture-space position **uv**

get triangle vertices v_1, v_2, v_3 and weights w_1, w_2, w_3 at **uv**
$\text{uv}_1 = \textbf{uv} + \textbf{hash}(v_1)$
$\text{uv}_2 = \textbf{uv} + \textbf{hash}(v_2)$
$\text{uv}_3 = \textbf{uv} + \textbf{hash}(v_3)$
$\text{duvdx} = \textbf{dFdx}(\textbf{uv})$
$\text{duvdy} = \textbf{dFdy}(\textbf{uv})$
$\mathbf{G}_1 = \textbf{texture2DGrad}(\textbf{texGaussian}, \text{uv}_1, \text{duvdx}, \text{duvdy})$
$\mathbf{G}_2 = \textbf{texture2DGrad}(\textbf{texGaussian}, \text{uv}_2, \text{duvdx}, \text{duvdy})$
$\mathbf{G}_3 = \textbf{texture2DGrad}(\textbf{texGaussian}, \text{uv}_3, \text{duvdx}, \text{duvdy})$
$\mathbf{G}^{\mathbf{cov}} = \frac{w_1\,\mathrm{G}_1 + w_2\,\mathrm{G}_2 + w_3\,\mathrm{G}_3 - \left(\frac{1}{2},\frac{1}{2},\frac{1}{2}\right)}{\sqrt{w_1^2 + w_2^2 + w_3^2}} + \left(\frac{1}{2},\frac{1}{2},\frac{1}{2}\right)$
$\mathbf{U} = \frac{1}{2} + \frac{1}{2}\,\mathrm{erf}\left(\frac{\mathrm{G}^{\mathbf{cov}} - \frac{1}{2}}{6\sqrt{2}}\right)$
$\mathbf{X}^{\mathbf{ours}} = \textbf{texture3D}(\textbf{lookUpTable}, \mathrm{U})$
return $\mathbf{X}^{\mathbf{ours}}$

Triangular grid: vertices and weights. The **uv** position in texture space falls within one triangle of the simplex grid. For the computation of the vertices of this triangle and their associated blending weights, we follow the procedure described by Perlin [2001].

Random offsets. We use the IDs of the 3 triangle vertices to compute a random offset for fetching the input Gaussian texture. We compute this offset with a **hash** function. Many options are available for this purpose, for instance the implementation of Gabor noise [Lagae et al. 2009] provides one.

Fetching the Gaussianized input. We fetch the Gaussianized texture with hardware MIPmapping [Williams 1983] and anisotropic filtering like a conventional texture. Note that the hardware uses screen-space derivatives to compute the MIPmap level and parameterize its anisotropic filter. Typically, these derivatives are computed with the finite differences between neighboring pixels of the **uv** positions passed as argument to the **texture2D** function. In our case, these screen-space derivatives are broken by the random offsets if neighboring pixels are not in the same triangle. To avoid this problem, we compute the derivatives of **uv** *before* adding the random offsets and we pass them explicitly to the **texture2DGrad** function. Note that we found that the calculations related to the grid do not result in visible aliasing at a distance. This is because the blending weights vary slowly in comparison to the content of the input texture, which is due to the assumption made in Section 5.3. Indeed, as the pixel-footprint size becomes larger than the size of the tile, the MIPmapped content of the fetched texture becomes a constant color and the result is the same whatever the values of the blending weights.

Variance-preserving blending of the Gaussian points. We compute $\mathbf{G}^{\mathbf{cov}}$ by blending the 3 fetched Gaussian values $\mathbf{G}_1$, $\mathbf{G}_2$, and $\mathbf{G}_3$ with their respective simplex-grid weights by applying Equation (8). Note that by construction of the Gaussian input, we have $\mathbb{E}[\mathbf{G}] = \left(\frac{1}{2},\frac{1}{2},\frac{1}{2}\right)$. This provides us with a stationary texture $\mathbf{G}^{\mathbf{cov}}$ with Gaussian histogram.

Fetching the look-up table. We use Equation (22) to map the Gaussian $\mathbf{G}^{\mathbf{cov}}$ to the uniform parameterization $\mathbf{U}$ of the look-up table $\mathcal{T}^{-1}$ and we fetch it to obtain our result $\mathbf{X}^{\mathbf{ours}}$ distributed in the histogram $\mathcal{H}$ of the input example. This is our final output color.

6 RESULTS AND DISCUSSIONS

In this section, we discuss our results and compare them to the state-of-the-art concurrent methods. All of the timings are given for rendering a full-screen quad on an NVIDIA GeForce GTX 980 at 1920×1080 resolution. Note that our supplemental material contains our full-image results at 1920×1080 resolution, which is useful for observing our result at the scale of dozens of tiles.

6.1 Random-Phase Noise

In Figure 10 and 11 , we show that our algorithm can be used to generate random-phase noise an order of magnitude faster than *Texton Noise*, which is currently the fastest method for this purpose. To produce these results, we used a 256^2 image produced by the texton-noise algorithm as our input.

Simplification for Gaussian inputs. Obviously, if the input already has a Gaussian histogram, we can use it as is with no need to compute the histogram transformations $\mathcal{T}$ and $\mathcal{T}^{-1}$. This means that we do not need to compute **step 1** and **step 3**, i.e. no look-up table to store and fetch, and our algorithm boils down to **step 2**, i.e. tiling the Gaussian input on the simplex grid with the variance-preserving blending operator.

Performance. This special case is extremely simple to implement and requires only 3 texture fetches for the tiling over the simplex grid and it runs at 0.11 ms per frame, which is 20 times faster than Texton Noise that on average uses 30 texture fetches and more complicated calculations. As with all point-convolution methods, they use an acceleration structure to determine contributing points. This requires visiting the neighbor cells, so the number of random points to evaluate is multiplied by 4 in the Texton Noise implementation. About 120 random hash values — a costly function on GPU — must then be evaluated at each pixel, while we need only 3. We used the OpenGL code provided by Galerne et al. [2017a] to benchmark the Texton Noise performance.

Power spectrum. In Section 5.3, we make the assumption that frequencies of the input image are captured within the size of the tiles. Thanks to this property, the frequency content of the input can be reproduced by our tiling scheme, and the multiplication by the blending weights has negligible impact on the spectrum.

Hardware bilinear-interpolation pattern on magnification. The hardware bilinear-interpolation pattern used to fetch the Gaussianized input texture becomes visible when our result is highly magnified, as with a conventional texture. Analytic noise primitives or Texton Noise have the advantage of producing a smoother result even at high magnification. However, in practice the cost of procedural-noise methods often prevents practitioners from using them on the fly, so noise textures end up being precomputed offline and stored as conventional textures instead. This means that, for many practitioners, the bilinear interpolation pattern is acceptable — mainly because if a texture happens to be highly magnified, it means that it its resolution is too low — while performance can prevent adoption. We thus believe that the bilinear-interpolation pattern visible at large magnification is a small price to pay for the performance achieved by our method.

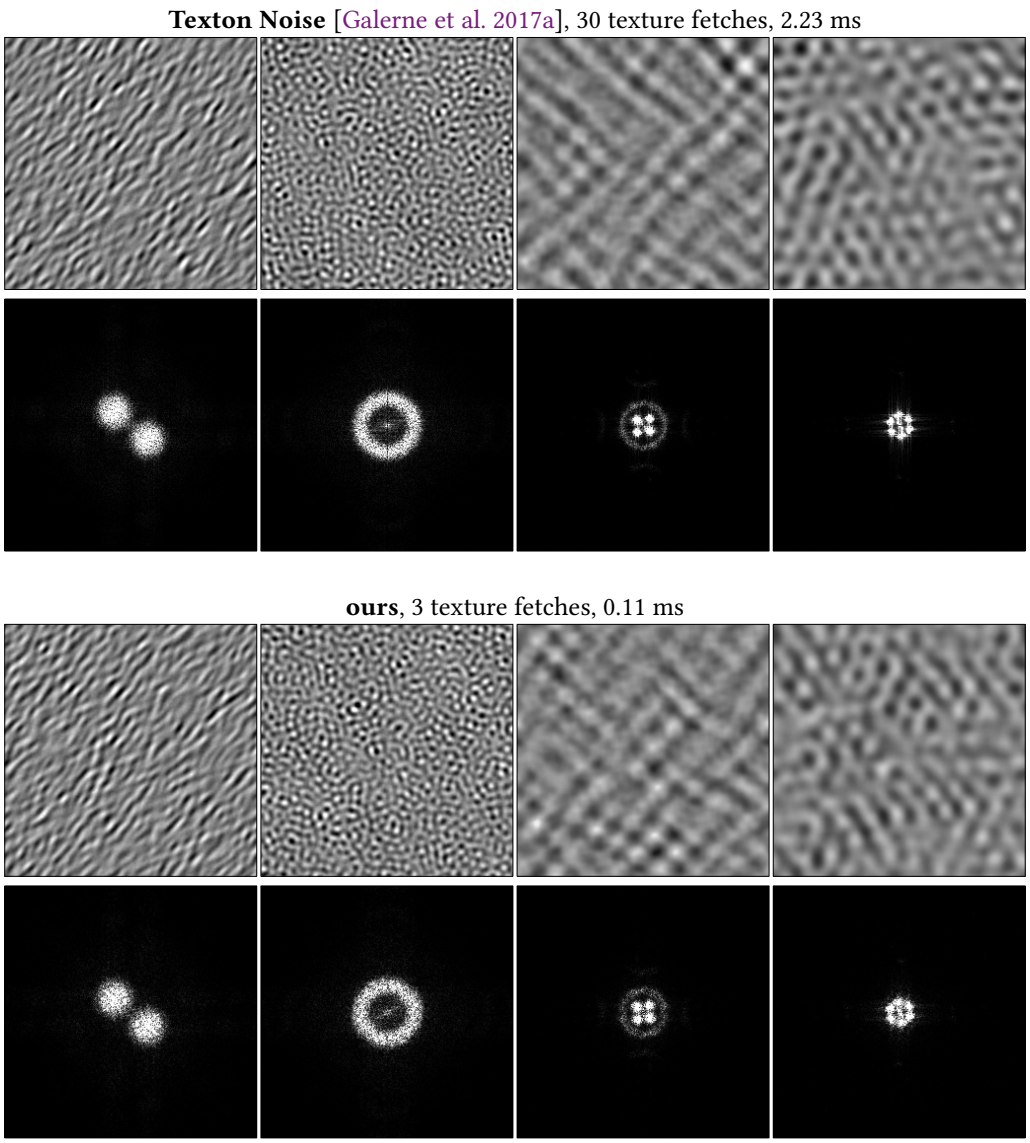

Fig. 10. Random-phase noise and their respective power spectra. *We reproduce the same results as previous random-phase noise methods with higher performance. We use a* 256^2 *image of the target noise as our input and we only need to compute* ***step 2*** *of our algorithm, i.e. we use only 3 texture fetches. We measured the performance by rendering a full-screen quad at* 1920×1080 *resolution on a GeForce GTX 980.*

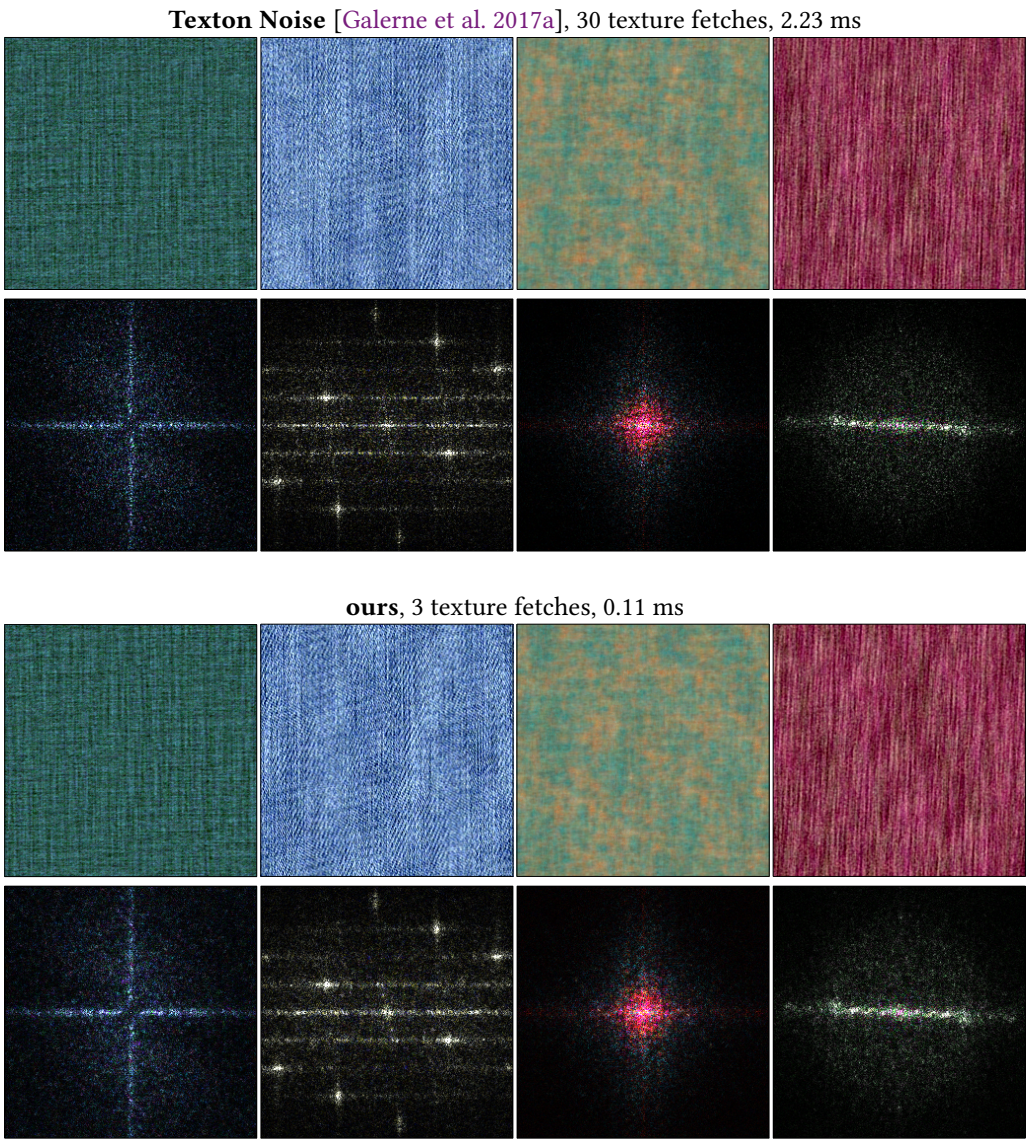

Fig. 11. Random-phase noise and their respective power spectra. *We reproduce the same results as previous random-phase noise methods with higher performance. We use a* 256^2 *image of the target noise as our input and we only need to compute* ***step 2*** *of our algorithm, i.e. we use only 3 texture fetches. We measured the performance by rendering a full-screen quad at* 1920×1080 *resolution on a GeForce GTX 980.*

6.2 Non-Random-Phase Noise

In addition to Texton Noise, Figures 12, 13 and 14 compare our method to *Local Random Phase Noise (LRPN)* [Gilet et al. 2014], which is the state-of-the-art method for non-random-phase noise.

Performance. Compared to our simplified random-phase noise algorithm, our complete implementation uses the Gaussianized version of the input image and the look-up table and it takes 0.29 ms per frame. This is because we have additional computation and a dependent look-up fetch. Since we did not have an implementation of LRPN to hand, we used the scaling factor reported by Galerne et al. [2017a] to get an estimate of their performance on our hardware. This is a factor of 10 in comparison to Texton Noise, which means that we might expect 22 ms for their method in our configuration. Our method is thus 75 times faster.

Power spectrum. In contrast to the random-phase case, we cannot make predictions about the spectra of our result. This is because the non-linear histogram transformations $\mathcal{T}$ and $\mathcal{T}^{-1}$ make the study of the spectrum complicated. In practice, we observed that the spectra are similar, as shown in Figure 12. This is because the histogram transformations, despite being non-linear, preserve (as much as possible) the structure of images, as explained in Section 4.1. In general, our method seems better than LRPN at preserving the spectrum of the input.

Qualitative comparison. In Figure 12, as expected, Texton Noise does not perform well for non-random-phase input images. We obtained the LRPN results by running the implementation provided by the authors. Since their method uses several parameters provided by the user, we have tuned the parameters for each example and kept the best results achieved by LRPN. We can see that our method works significantly better on stochastic examples, where the phase information contributes mostly to produce discontinuities and sharp features instead of patterns. This is because their method is based on the idea of extracting a periodic component of the spectrum of the input. If this repetitive component does not exist, LRPN boils down to random-phase noise, which is what happens on these examples except for the sand, and might even be worse than random-phase noise algorithms such as Texton Noise because of their particular color transformations, e.g. in the moss example (bottom right) some blue pixels appear. If the repetitive component identified by LRPN is overestimated, the result will have discontinuities, which is what happens with the sand example (bottom left). In contrast, our method performs well on this patternless category of texture. Not only do our results show accurate reproduction of the global histogram but also its inter-channel precise correlation. This higher quality shows in the images, especially when zooming in on details.

We obtained the results of Figure 14 by taking the examples and results of the LRPN article and applying our method to them. These examples are not purely stochastic, as some structure and organization is clearly visible. In this case, our method appears to increase the frequency of the pattern. This is especially visible in the cracked skin example (bottom left). However, their method produces high-frequency noise that isn't present in the input, and the colors seem wrongly quantized (zoom in on the top-right example). Our results are free of spurious high-frequency noise and have the right colors thanks to the histogram preservation.

We cannot make a general statement as to whether our method or LRPN is better, since they have different strengths and weaknesses depending on the images. Our results lead us to believe that it is safe to claim that the generative space of our method is as wide as the one of LRPN, while our method is 75 times faster and does not require user assistance to set up parameters.

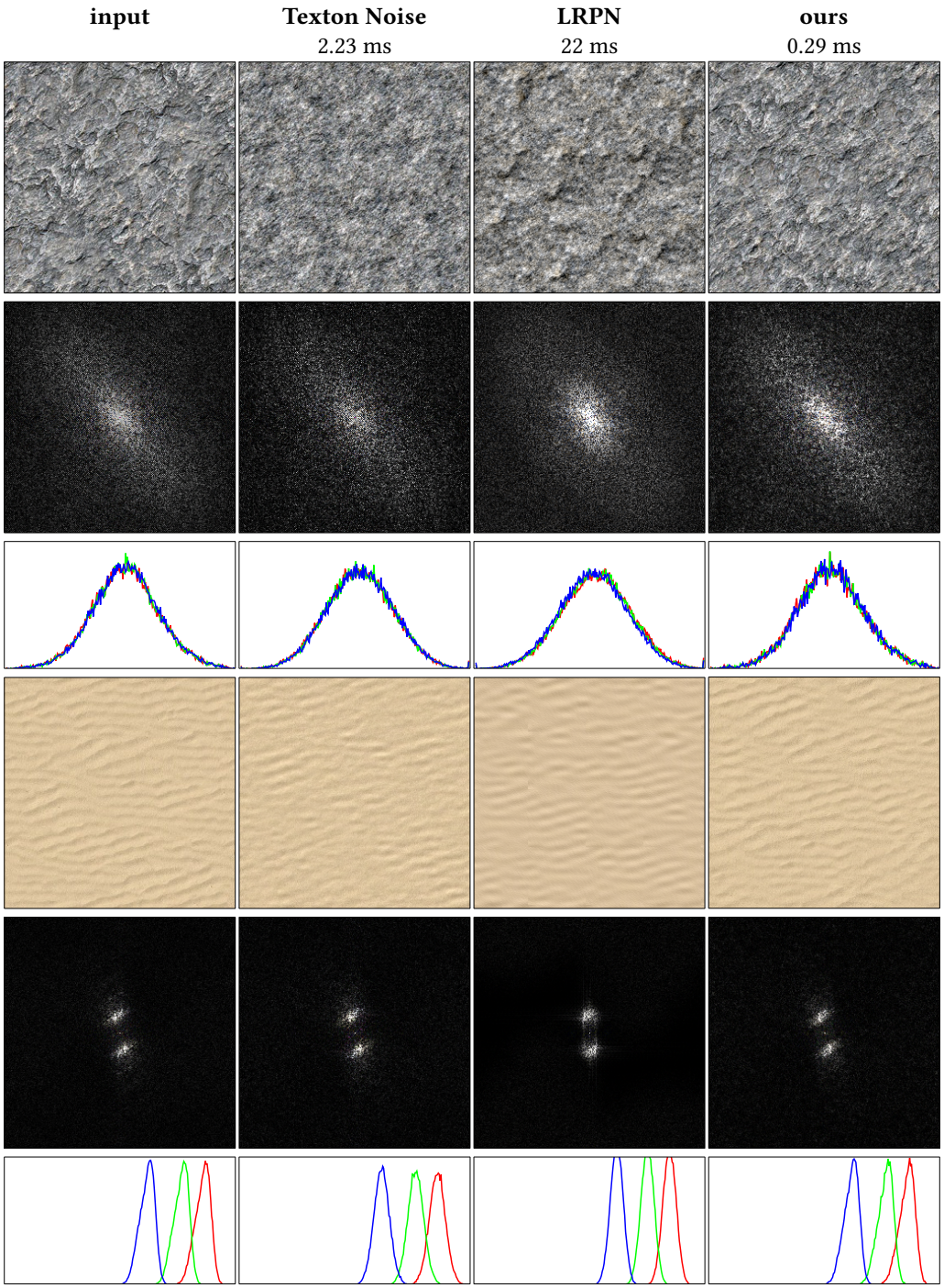

Fig. 12. Non-random-phase noise. *We compare our method to Texton Noise [Galerne et al. 2017a] and Local Random Phase Noise (LRPN) [Gilet et al. 2014]. For each image, we show its spectrum and histogram. We measured the performance by rendering a full-screen quad at* 1920×1080 *resolution on a GeForce GTX 980.*

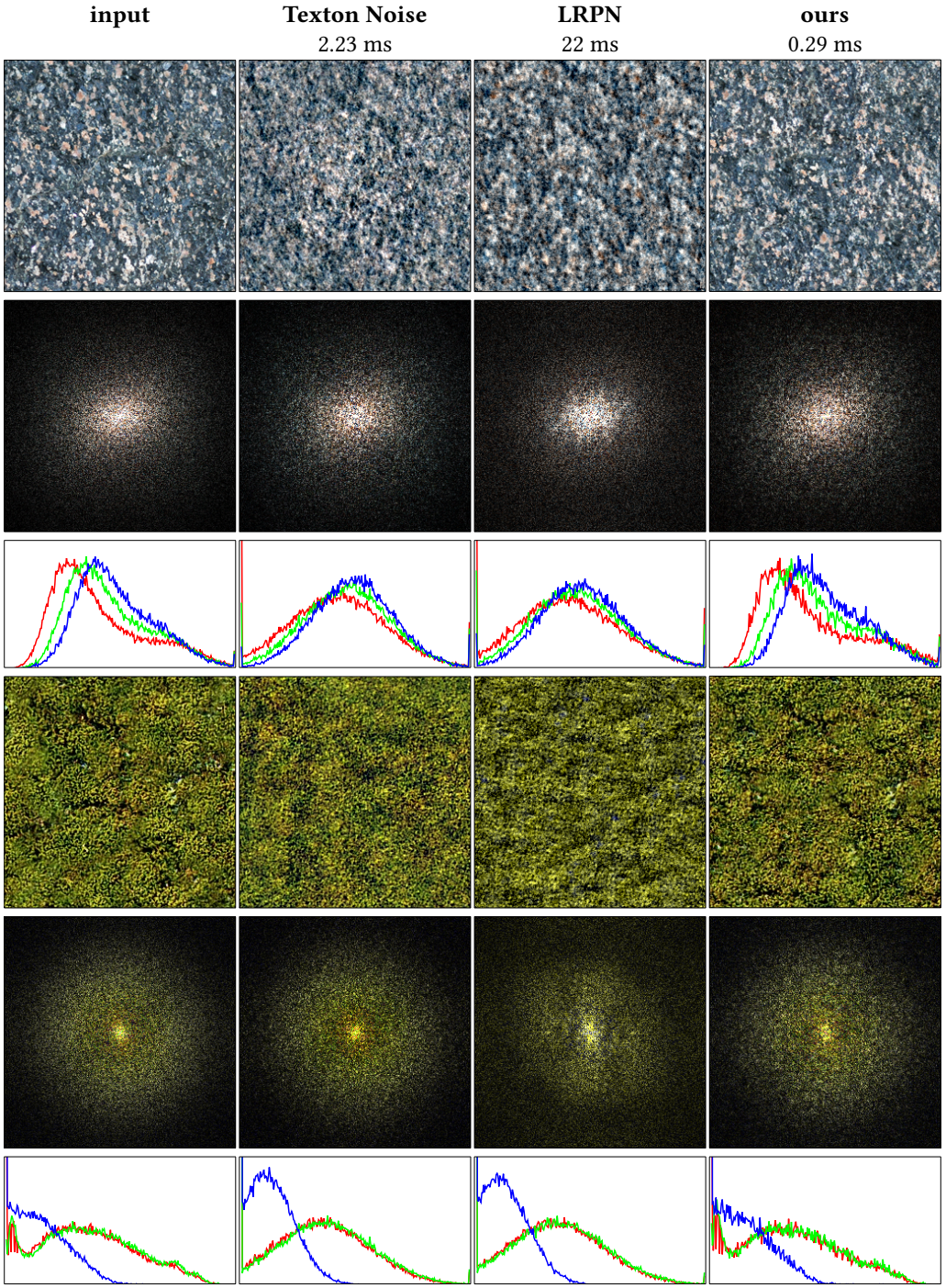

Fig. 13. Non-random-phase noise. *We compare our method to Texton Noise [Galerne et al. 2017a] and Local Random Phase Noise (LRPN) [Gilet et al. 2014]. For each image, we show its spectrum and histogram. We measured the performance by rendering a full-screen quad at* 1920×1080 *resolution on a GeForce GTX 980.*

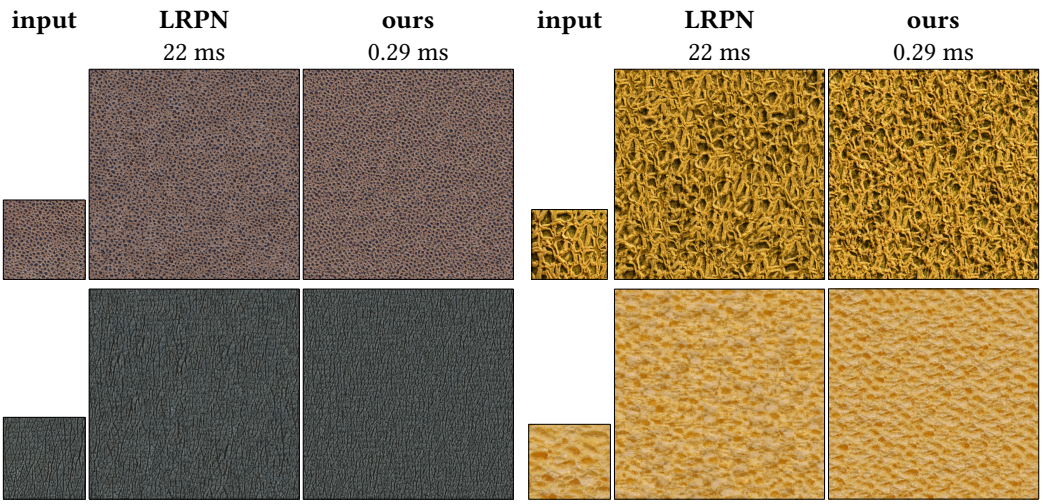

Fig. 14. Non-random-phase noise. *We compare our method to LRPN on the examples provided in their article [Gilet et al. 2014]. We measured the performance by rendering a full-screen quad at* 1920×1080 *resolution on a GeForce GTX 980.*

Failure cases. Figure 15 shows typical failure cases of our method if the input contains strong patterns or is repetitive.

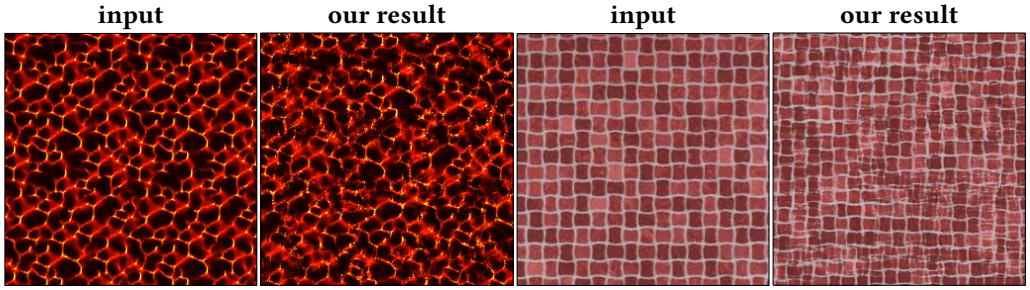

Fig. 15. Failure cases of our method. *Our method does not produce plausible results if the input presents a strong pattern-like organization or, worse, repetition.*

7 CONCLUSION AND FUTURE WORK

We have presented a high-performance and accurate method for on-the-fly generation of procedural noise that reproduces the main characteristics of an example noise image: power spectrum, stationary statistics and color histogram with correlations. It accurately reproduces all of the random-phase results of Texton Noise [Galerne et al. 2017a] and Gabor Noise by Example [Galerne et al. 2012] 20 times faster than the former, which was already dramatically more efficient than the previous work. Indeed our method could be used directly to boost simple stationary Gabor noise. More interestingly, our method can also faithfully reproduce many examples containing structure, while still preserving statistics, spectrum and accurate color histogram. This is thanks to our histogram-preserving blending operator, which greatly extends the generative space of by-example procedural noise.

Our run-time implementation is easy to implement and requires no data structure except for the Gaussianized example image and a 32^3 look-up table. As for Texton Noise, it relies on the GPU for storing, interpolating and filtering. At the cost of only 3 random hashes, 3 weight evaluations, and 3+1 texture fetches we believe we have proposed the first by-example procedural noise that is efficient enough for demanding real-time applications such as games and simulators. Figure 1 shows a scene rendered in a video-game engine in which we implemented our method as a custom material shader.

The main limitation is that not all possible structures are well reproduced, but this relates to the limits of what a *stochastic* texture is: long-correlated and recognizable features are likely beyond this scope. Moreover, in contrast to Texton noise as well as the Gabor-like family, the bilinear interpolation pattern can be visible at high magnification as with a conventional texture.

Finally, we believe that our histogram-preserving blending operator could benefit other applications as well. For instance, in Figure 16 we show how it can be used to improve triplanar mapping. Another interesting future work is managing more than 3 color channels, which could be used to reproduce displacement or bump that accounts for correlation with colors, or any other attribute.

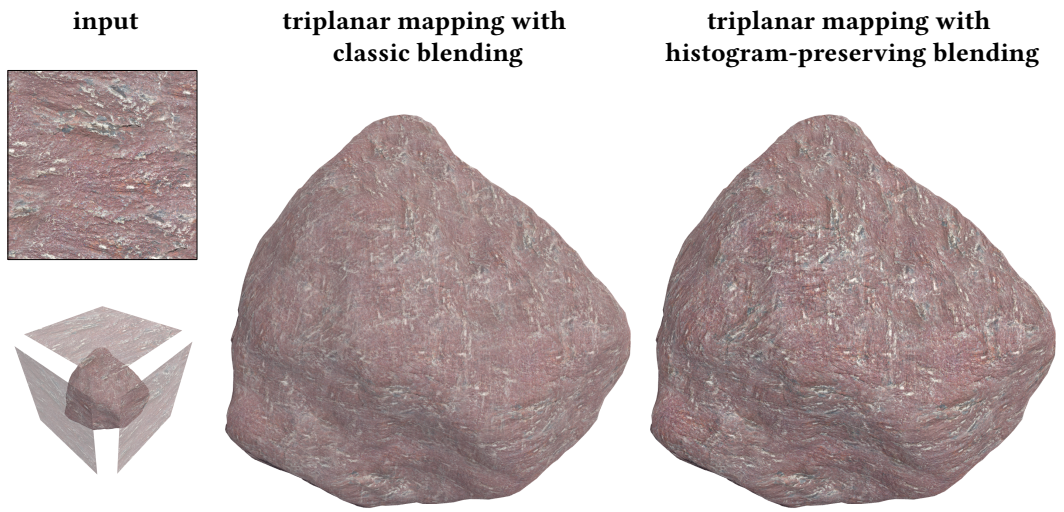

Fig. 16. Other application example: improving triplanar mapping with histogram-preserving blending. *Triplanar mapping textures uv-free meshes by projecting three textures on the X, Y, Z planes and blending them using the surface normal components as blending weights. A typical problem of triplanar mapping is that blending lowers contrast and produces ghosting artifacts. Replacing the classic blending by our histogram-preserving blending solves this problem and is straightforward. Furthermore, since triplanar mapping already fetches the input three times, the overhead of using our blending operator is only the final look-up table fetch.*

REFERENCES

Nicolas Bonneel. 2016. FastTransport. (2016). http://liris.cnrs.fr/ nbonneel/FastTransport/.

Nicolas Bonneel, Michiel van de Panne, Sylvain Paris, and Wolfgang Heidrich. 2011. Displacement Interpolation Using Lagrangian Mass Transport. *ACM Trans. Graph.* 30, 6, Article 158 (2011), 12 pages.

Michael F. Cohen, Jonathan Shade, Stefan Hiller, and Oliver Deussen. 2003. Wang Tiles for Image and Texture Generation. *ACM Trans. Graph.* 22, 3 (2003), 287–294.

Luc Devroye. 1986. *Non-Uniform Random Variate Generation.*

Alexei A. Efros and William T. Freeman. 2001. Image Quilting for Texture Synthesis and Transfer *(SIGGRAPH '01)*. 341–346.

Bruno Galerne, Ares Lagae, Sylvain Lefebvre, and George Drettakis. 2012. Gabor Noise by Example. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2012)* 31, 4 (2012), 73:1–73:9.

B. Galerne, A. Leclaire, and L. Moisan. 2017a. Texton Noise. *Computer Graphics Forum* (2017).

Bruno Galerne, Arthur Leclaire, and Julien Rabin. 2017b. Semi-Discrete Optimal Transport in Patch Space for Enriching Gaussian Textures. In *Geometric Science of Information (Lecture Notes in Computer Science)*, Vol. 10589.

G. Gilet, J-M. Dischler, and D. Ghazanfarpour. 2012. Multiple Kernels Noise for Improved Procedural Texturing. *Vis. Comput.* 28, 6-8 (June 2012), 679–689.

Guillaume Gilet, Basile Sauvage, Kenneth Vanhoey, Jean-Michel Dischler, and Djamchid Ghazanfarpour. 2014. Local Random-phase Noise for Procedural Texturing. *ACM Trans. Graph.* 33, 6, Article 195 (2014), 11 pages.

David J. Heeger and James R. Bergen. 1995. Pyramid-based Texture Analysis/Synthesis. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '95)*. 229–238.

Ares Lagae and Philip Dutré. 2006. An Alternative for Wang Tiles: Colored Edges versus Colored Corners. *ACM Transactions on Graphics* 25, 4 (2006), 1442–1459.

Ares Lagae, Sylvain Lefebvre, Rob Cook, Tony DeRose, George Drettakis, D.S. Ebert, J.P. Lewis, Ken Perlin, and Matthias Zwicker. 2010. State of the Art in Procedural Noise Functions. In *EG 2010 - State of the Art Reports*.

Ares Lagae, Sylvain Lefebvre, George Drettakis, and Philip Dutré. 2009. Procedural Noise using Sparse Gabor Convolution. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2009)* 28, 3 (2009), 54–64.

John-Peter Lewis. 1984. Texture Synthesis for Digital Painting. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '84)*. 245–252.

Ján Morovic and Pei-Li Sun. 2003. Accurate 3D Image Colour Histogram Transformation. *Pattern Recogn. Lett.* 24, 11 (July 2003), 1725–1735.

Fabrice Neyret and Marie-Paule Cani. 1999. Pattern-based Texturing Revisited. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '99)*. 235–242.

Ken Perlin. 1985. An Image Synthesizer. In *Computer Graphics (SIGGRAPH '85 Proceedings)*, B. A. Barsky (Ed.), Vol. 19. 287–296.

Ken Perlin. 2001. Noise hardware. (2001). Real-time shading languages, SIGGRAPH 2001 Course.

Ken Perlin. 2002. Improving Noise. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '02)*. 681–682.

Emil Praun, Adam Finkelstein, and Hugues Hoppe. 2000. Lapped Textures *(SIGGRAPH '00)*. 465–470.

Erik Reinhard, Michael Ashikhmin, Bruce Gooch, and Peter Shirley. 2001. Color Transfer Between Images. *IEEE Comput. Graph. Appl.* 21, 5 (2001), 34–41.

Jarke J. van Wijk. 1991. Spot Noise Texture Synthesis for Data Visualization. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '91)*. 309–318.

Kenneth Vanhoey, Basile Sauvage, Frédéric Larue, and Jean-Michel Dischler. 2013. On-the-fly Multi-scale Infinite Texturing from Example. *ACM Trans. Graph.* 32, 6, Article 208 (2013), 10 pages.

Li-Yi Wei, Sylvain Lefebvre, Vivek Kwatra, and Greg Turk. 2009. State of the Art in Example-based Texture Synthesis. In *Eurographics 2009, State of the Art Report, EG-STAR*.

Lance Williams. 1983. Pyramidal Parametrics. *SIGGRAPH Comput. Graph.* 17, 3 (1983), 1–11.

Steven P. Worley. 1996. A Cellular Texture Basis Function. In *SIGGRAPH 96 Conf. Proc.* 291–294.

Qizhi Yu, Fabrice Neyret, Eric Bruneton, and Nicolas Holzschuch. 2011. Lagrangian Texture Advection: Preserving Both Spectrum and Velocity Field. *IEEE Transactions on Visualization and Computer Graphics* 17, 11 (2011), 1612–1623.

ACKNOWLEDGMENTS

The authors would like to thank Thomas Deliot for implementing the algorithm in the Unity engine, Kenneth Vanhoey for sharing LRPN results and Stephen Hill for his numerous corrections.