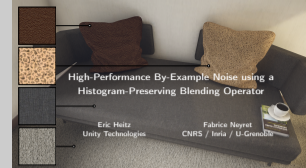


High-Performance By-Example Noise using a Histogram-Preserving Blending Operator

Eric Heitz
Unity Technologies

Fabrice Neyret
CNRS / Inria / U-Grenoble



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

Eric Heitz
Unity Technologies

Fabrice Neyret
CNRS / Inria / U-Grenoble



This one of Unity's tutorial game. Despite its simplicity, it is already facing a big rendering challenge: how to texture large areas with small-scale details?

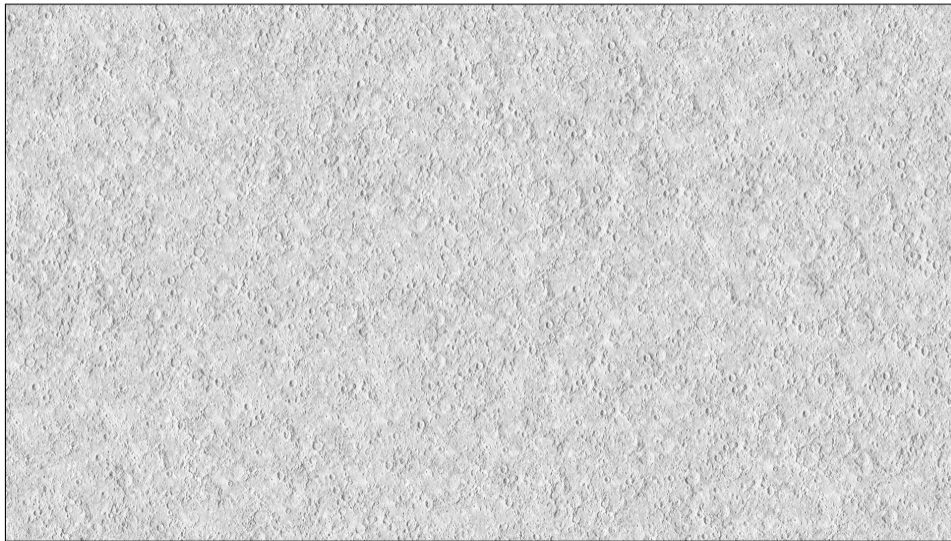
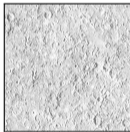
To do that with a reasonable memory budget, a classic approach consists in using tileable textures that can be seamlessly repeated. However, doing this results in a visible repetition artifact.



We would rather like to be able to texture large areas with details and without visible repetitions.

Introduction

example



example



Ideally, we would like the following workflow. The user provides a small piece of an example texture and we are looking for the ability to compute a texture with...

Introduction

example



Desired properties

- Non-repetitive (no repeated tiles)
- Infinite
- Same appearance as example

Technical requirements

- Fast
- Small memory footprint
- On the fly (fragment shader only)

example



Desired properties

- Non-repetitive (no repeated tiles)
- Infinite
- Same appearance as example

Technical requirements

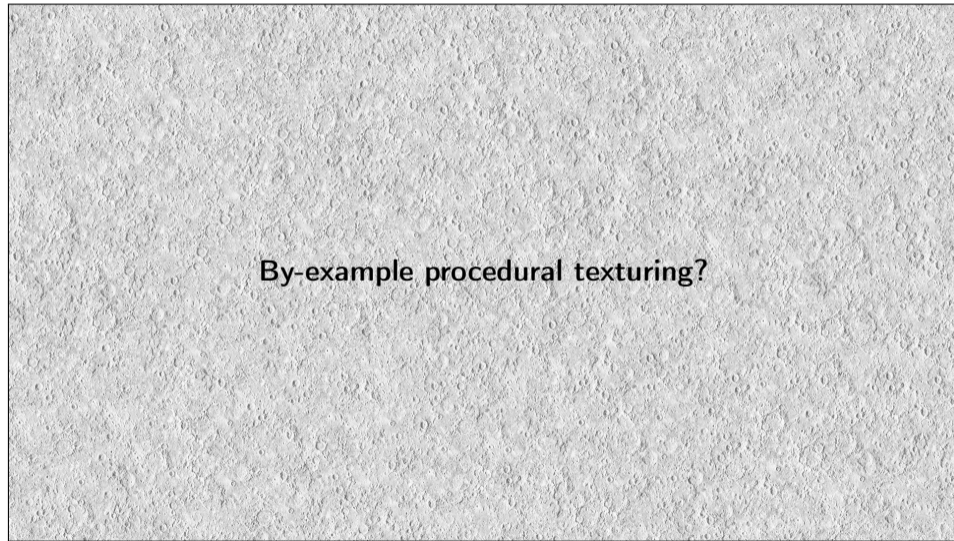
- Fast
- Small memory footprint
- On the fly (fragment shader only)

...the following properties. We want a potentially infinite output of the same appearance without visible repetitions. This excludes the obvious OpenGL repeat mode with a tileable texture but also aperiodic tiling approaches (such as Wang tiles) that still repeat the same features in an obvious manner.

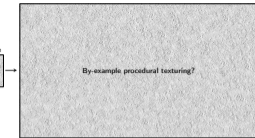
To be usable, such an approach should be at the same time fast and have a reasonable memory footprint. Ideally, it would be evaluated on the fly as a small piece of fragment-shader code.

Introduction

example



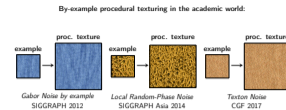
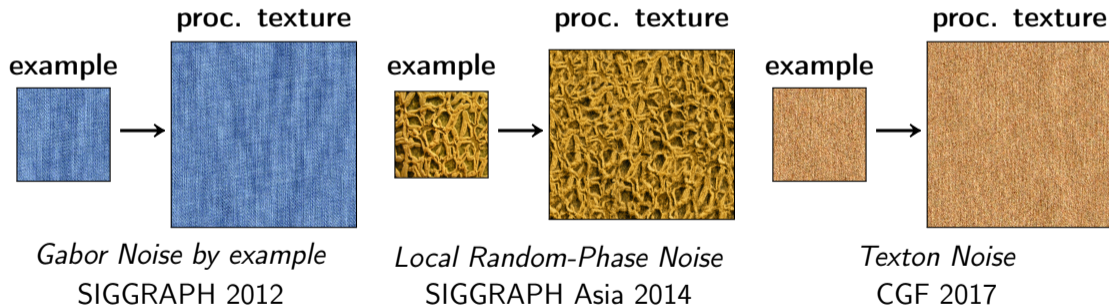
example



What are we looking for sounds like by-example procedural texturing.

Introduction

By-example procedural texturing in the academic world:



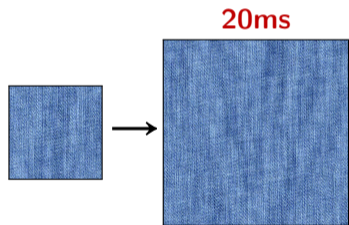
By-example procedural texturing is an existing research topic and there is some noticeable academic previous work. These three ones are probably the most emblematic ones.

Note that we are interested in procedural methods that can be evaluated in a real-time rendering engine, i.e. as a fragment shader. This excludes other texture synthesis methods, such as deep-learning for instance.

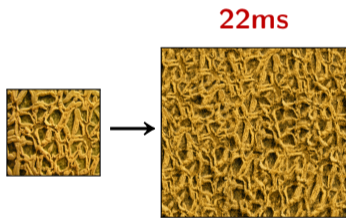
With this constraint in mind, the three methods shown here achieve pretty high-quality procedural textures.

Introduction

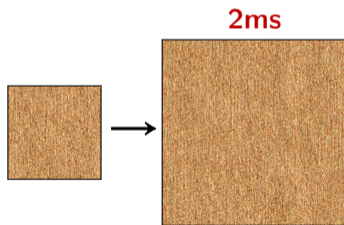
Too costly for games



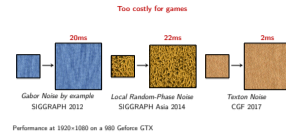
Gabor Noise by example
SIGGRAPH 2012



Local Random-Phase Noise
SIGGRAPH Asia 2014



Texton Noise
CGF 2017

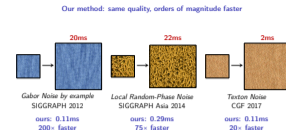
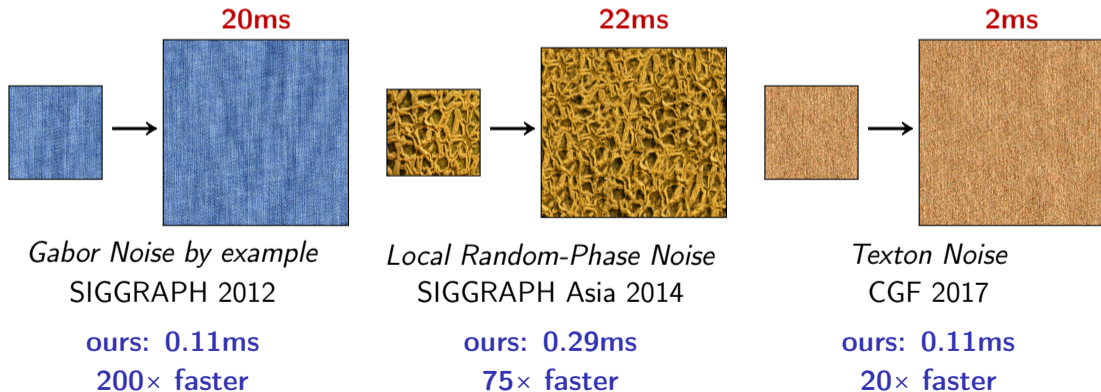


Unfortunately, these techniques are still way too costly to be really considered in practice.

Texton Noise is by far the fastest of these techniques. However, with 2ms at 1080p on a high-end graphic card, it is still way too costly for gaming real-time.

Introduction

Our method: same quality, orders of magnitude faster



In this paper, we propose a method that, we hope, really brings by-example procedural textures to real-time with a huge performance speed up. Our method competes with previous work in terms of quality but is orders of magnitude faster.

Introduction

Story

We tried the cheap low-quality approaches people use in practice.

We looked into the low-quality problem.

We fixed it.

It's going to be a journey in the world of blending.

Story

We tried the cheap low-quality approaches people use in practice.

We looked into the low-quality problem.

We fixed it.

It's going to be a journey in the world of blending.

The outline of the talk.

Cheap approach: texture bombing

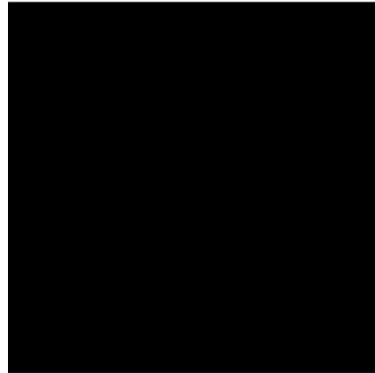
A popular procedural texturing method used in practice is Texture Bombing.

Cheap approach: texture bombing

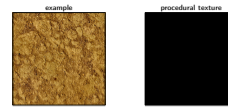
example



procedural texture



The user provides an input.

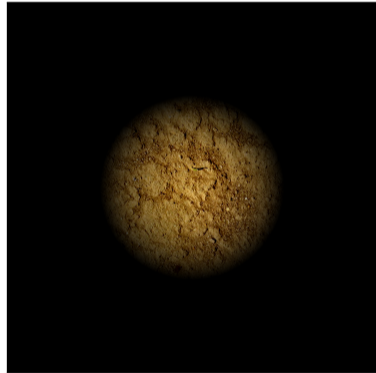


The user provides an input.

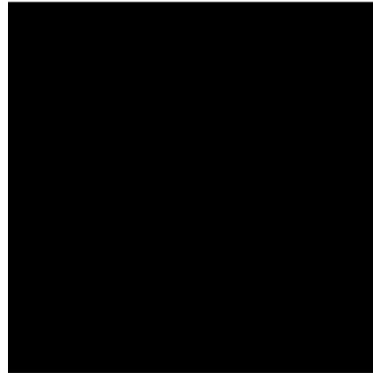
The user provides an example texture with the desired appearance, ...

Cheap approach: texture bombing

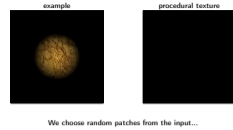
example



procedural texture



We choose random patches from the input...

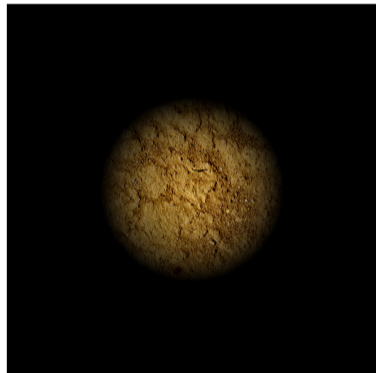


...from which random patches are chosen with a smoothing kernel.

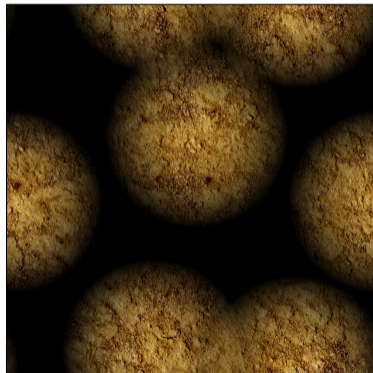
The most naive implementations use always the same patch but nothing prevents randomizing its position in the example texture.

Cheap approach: texture bombing

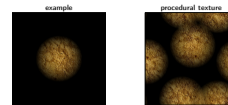
example



procedural texture



...and we splat them randomly...

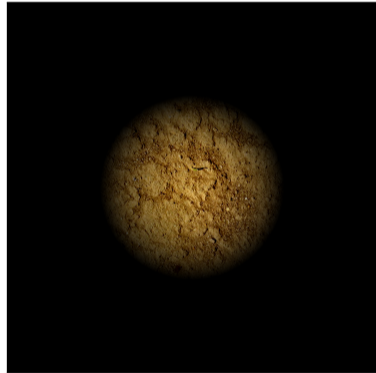


...and we splat them randomly...

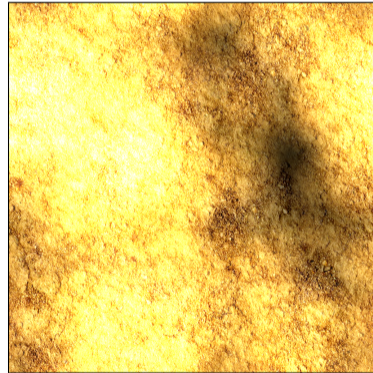
These random patches are splatted in the procedural texture. From a fragment-shader perspective, it means that for each pixel we need to find the contributing patches and do a texture fetch in the example texture.

Cheap approach: texture bombing

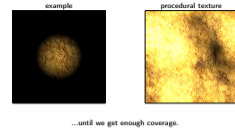
example



procedural texture



...until we get enough coverage.



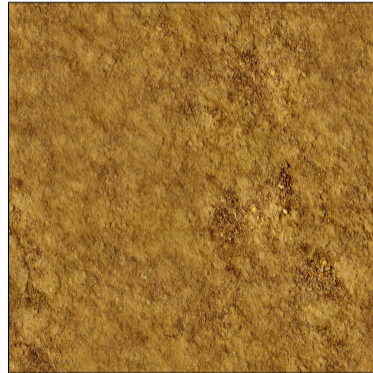
It is important to splat enough patches to get enough coverage in the procedural texture.

Cheap approach: texture bombing

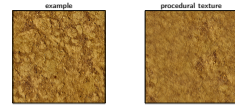
example



procedural texture



We divide each pixel by the density of patches.

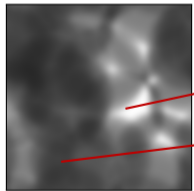


We divide each pixel by the density of patches.

The final value of the procedural texture is obtained by dividing by the patch density (the sum of the smoothing kernel). By doing this, we make sure that the average color of the pixels is effectively the average color of the example.

Cheap approach: texture bombing

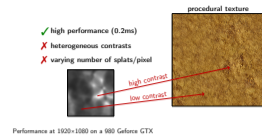
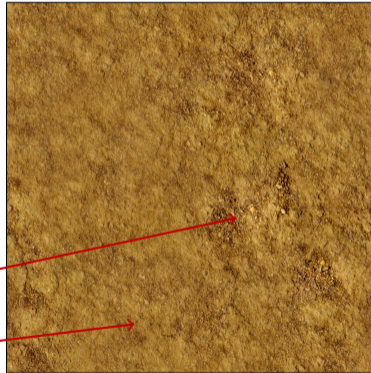
- ✓ high performance (0.2ms)
- ✗ heterogeneous contrasts
- ✗ varying number of splats/pixel



high contrast

low contrast

procedural texture



Performance at 1920x1080 on a 980 Geforce GTX

This method is simple enough to yield true real-time performance.

Unfortunately, there are some noticeable appearance problems. The most obvious one is that the contrast of the procedural texture is heterogeneous: some zones are less contrasted than others.

It seems reasonable to believe that this is due to the varying number of patches that are splatted per pixel. Can we fix this?

Improved cheap approach: tiling and blending

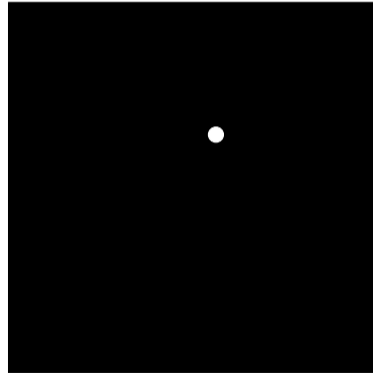
We are now going to describe an alternative tiling-and-blending algorithm that fixes this first issue. This is the basis of the algorithm our final procedural method is based on.

Improved cheap approach: tiling and blending

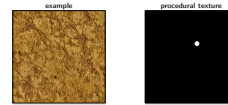
example



procedural texture



We want to evaluate the procedural texture at this point.



We want to evaluate the procedural texture at this point.

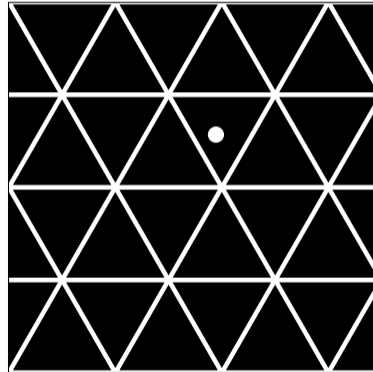
We are in the fragment shader and we want to evaluate the procedural texture at this point (the white dot).

Improved cheap approach: tiling and blending

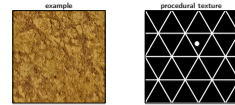
example



procedural texture



We partition the texture space on a triangle grid.



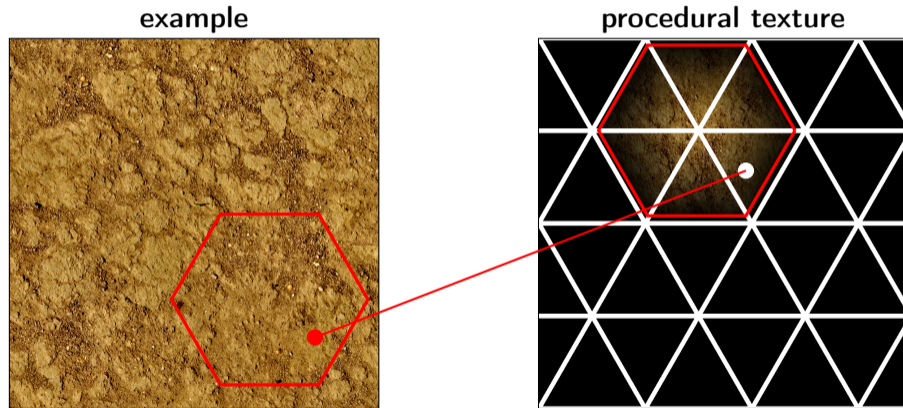
We partition the texture space on a triangle grid.

The first thing we do is partition the texture space on a triangle grid.

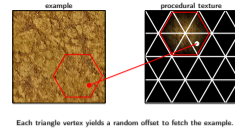
The fragment shader finds the triangle in which its uv coordinates are located. It computes the coordinates of its 3 vertices and its barycentric coordinates inside the triangle.

Note that we could also use a classic square grid. The advantage of the triangle grid is that the fragment shader will have to do the further computations only for 3 vertices instead of 4 on a square grid. Using a triangle grid is thus 33% faster.

Improved cheap approach: tiling and blending



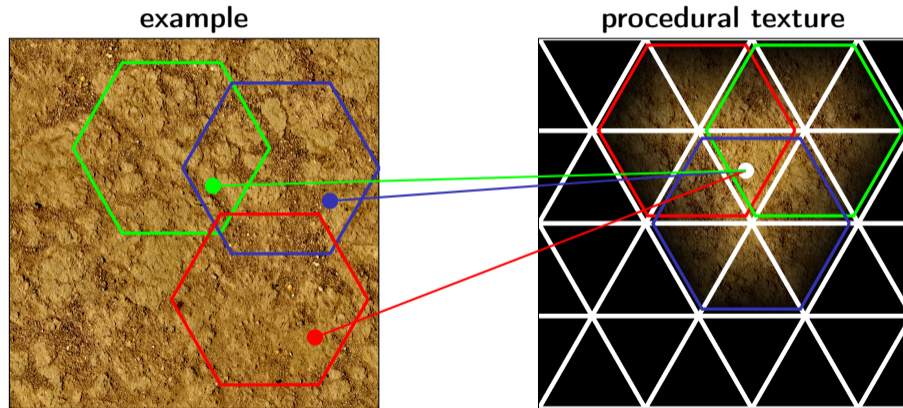
Each triangle vertex yields a random offset to fetch the example.



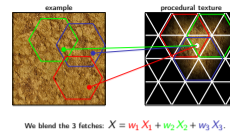
We use a hash function to associate a random offset with each vertex of the triangle grid. This random offset is used to fetch the example texture.

By doing this, we are virtually splatting hexagonal patches randomly chosen in the example and centered on the triangle vertices in the procedural texture.

Improved cheap approach: tiling and blending



We blend the 3 fetches: $X = w_1 X_1 + w_2 X_2 + w_3 X_3$.



We do that for each of the 3 vertices and we blend the result using the barycentric coordinates as blending weights in order to obtain smooth transitions between the patches.

Improved cheap approach: tiling and blending

Fragment shader

1. Get triangle vertices and blending weights

$$V_1, V_2, V_3 \quad w_1, w_2, w_3$$

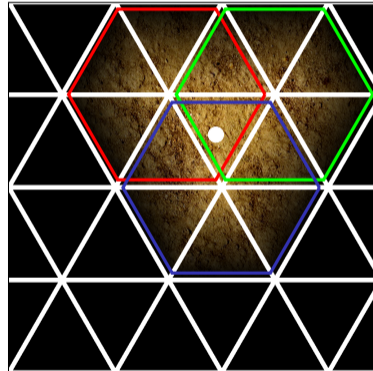
2. Compute random offsets at vertices
 $\text{offset}_i = \text{hash}(V_i)$

3. Fetch example with random offsets
 $X_i = \text{texture}(uv + \text{offset}_i)$

4. Blend

$$X = w_1 X_1 + w_2 X_2 + w_3 X_3$$

procedural texture



Fragment shader

1. Get triangle vertices and blending weights
 $V_1, V_2, V_3 \quad w_1, w_2, w_3$
2. Compute random offsets at vertices
 $\text{offset}_i = \text{hash}(V_i)$
3. Fetch example with random offsets
 $X_i = \text{texture}(uv + \text{offset}_i)$
4. Blend
 $X = w_1 X_1 + w_2 X_2 + w_3 X_3$

procedural texture

This is the fragment-shader code of this algorithm.

The advantage of this algorithm over texture bombing is that each pixel is computed using exactly 3 patches. We have thus solved the problem of the spatially-varying number of patches used per pixel.

Improved cheap approach: tiling and blending

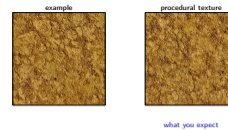
example



procedural texture



what you expect



We thus expect a nice-looking procedural texture with correct contrast.

Improved cheap approach: tiling and blending

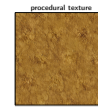
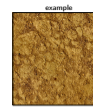
example



procedural texture



what you get

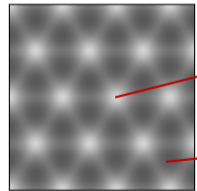


what you get

Unfortunately, the result is not satisfying either.

Improved cheap approach: tiling and blending

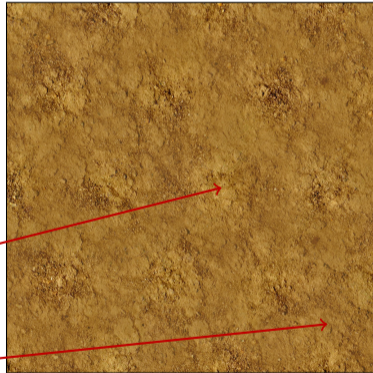
- ✓ high performance (0.1ms)
- ✓ constant number of splats/pixel
- ✗ still heterogeneous contrast!



high contrast

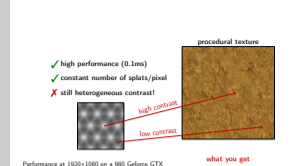
low contrast

procedural texture



what you get

Performance at 1920×1080 on a 980 Geforce GTX



By using only 3 patches per pixel, we obtained a nice side effect: the performance slightly increased compared to texture bombing. This is because 3 is smaller than the average number of patches used in texture bombing and because the algorithm is simpler and does not introduce divergence and branching operations.

Unfortunately, it did not solve the contrast problem and it is even worse: now the heterogeneous contrast produce a grid-revealing pattern. It means that the contrast problem is more complicated than just using the same number of patches per pixel.

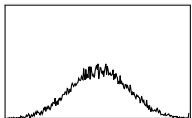
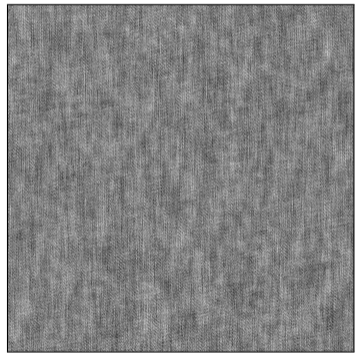
What's the problem with contrast?

What's the problem with contrast?

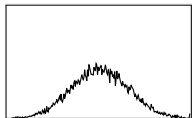
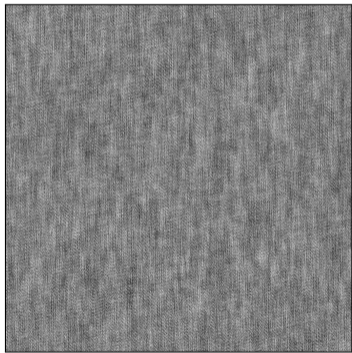
So, what's the problem with the contrast?

What's the problem with contrast?

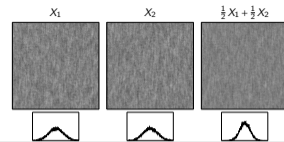
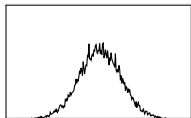
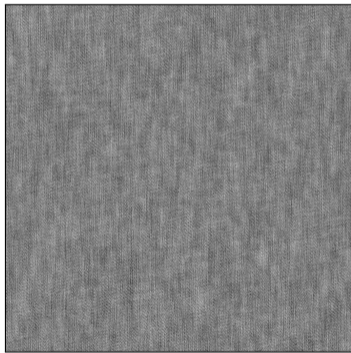
X_1



X_2



$\frac{1}{2} X_1 + \frac{1}{2} X_2$



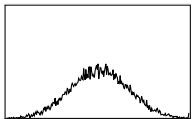
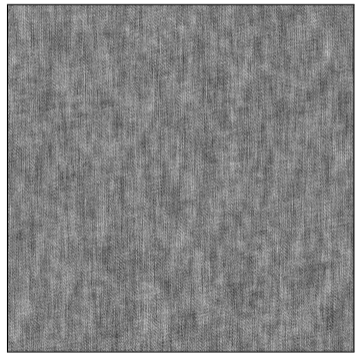
We will see that the contrast problem is related to the blending operation.

For now, we will leave the procedural texturing application on the side and we will investigate blending.

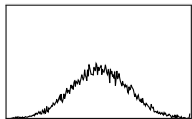
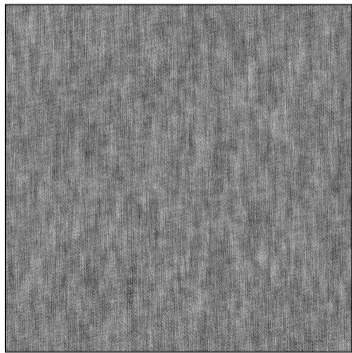
To do that, we look into the simple operation of blending two images. We can see that the blended image has slightly lower contrasts.

What's the problem with contrast?

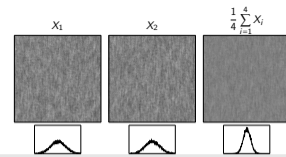
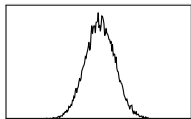
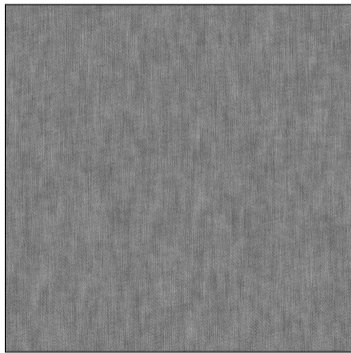
X_1



X_2



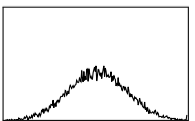
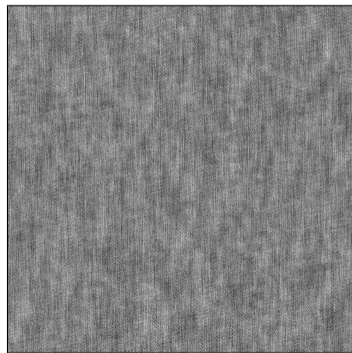
$$\frac{1}{4} \sum_{i=1}^4 X_i$$



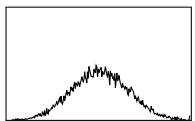
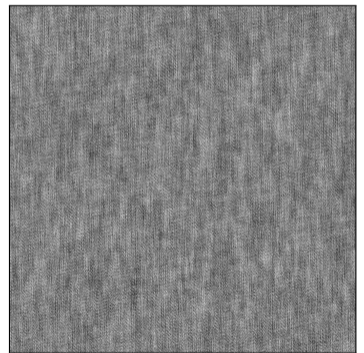
This becomes more obvious if we blend 4 images instead of 2.

What's the problem with contrast?

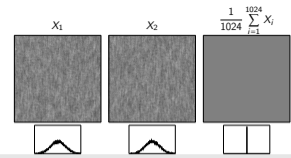
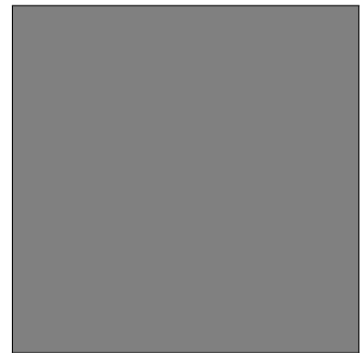
X_1



X_2



$$\frac{1}{1024} \sum_{i=1}^{1024} X_i$$



If we blend a large number of images, the result converges towards the average color of the images.

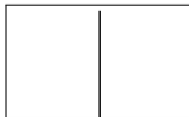
What's the problem with contrast?

$$\frac{1}{1024} \sum_{i=1}^{1024} X_i$$

contrast = variance

$$\text{var} \left(\frac{1}{N} \sum_{i=1}^N X_i \right) = \frac{1}{N} \text{var}(X_i)$$

Averaging reduces the variance.



contrast = variance

$$\text{var} \left(\frac{1}{N} \sum_{i=1}^N X_i \right) = \frac{1}{N} \text{var}(X_i)$$

Averaging reduces the variance.

A diagram showing a large gray square above a small white square with a vertical line, illustrating the reduction in contrast.

This is a well-known phenomenon. Contrast is measured by the variance of the image (or the standard deviation, the square-root of the variance).

The problem is that averaging things reduces their variance. The more we average, the more the variance is reduced, the less contrasted is the result.

Can we solve this contrast (variance) problem?

Can we solve this contrast (variance) problem?

Fortunately, the solution to this problem is not very complicated.

Can we solve this contrast (variance) problem?

$$X^{\text{blended}} = \sum_{i=1}^N w_i X_i \quad \leftarrow \text{linear blending}$$

$$\mathbb{E}[X^{\text{blended}}] = \mathbb{E}[X_i] \quad \checkmark$$

$$\text{var}(X^{\text{blended}}) = \underbrace{\left(\sum_{i=1}^N w_i^2 \right)}_{\text{less than 1}} \text{var}(X_i) \quad \times$$

$$X^{\text{blended}} = \sum_{i=1}^N w_i X_i \quad \leftarrow \text{linear blending}$$
$$\mathbb{E}[X^{\text{blended}}] = \mathbb{E}[X_i] \quad \checkmark$$
$$\text{var}(X^{\text{blended}}) = \underbrace{\left(\sum_{i=1}^N w_i^2 \right)}_{\text{less than 1}} \text{var}(X_i) \quad \times$$

In the general blending case, we consider a set of N images (or patches) that are blended with blending weights.

Because blending weights are normalized ($w_1 + \dots + w_N = 1$), the expectation (the average color) of the blended result is the same as the input.

However, if we do the math we can predict that the variance of the blended result is the variance of the input multiplied by the sum of the squared weights, which is less than one. The variance of the blended result is thus smaller as the variance of the input.

Can we solve this contrast (variance) problem?

1. linear blending

$$X^{\text{blended}} = \frac{\sum_{i=1}^N w_i X_i}{\sqrt{\sum_{i=1}^N w_i^2}}$$

2. fix variance

$$\begin{aligned} \mathbb{E}[X^{\text{blended}}] &\neq \mathbb{E}[X_i] && \times \\ \text{var}(X^{\text{blended}}) &= \text{var}(X_i) && \checkmark \end{aligned}$$

1. linear blending

$$X^{\text{blended}} = \frac{\sum_{i=1}^N w_i X_i}{\sqrt{\sum_{i=1}^N w_i^2}}$$

2. fix variance

$$\begin{aligned} \mathbb{E}[X^{\text{blended}}] &\neq \mathbb{E}[X_i] && \times \\ \text{var}(X^{\text{blended}}) &= \text{var}(X_i) && \checkmark \end{aligned}$$

One way to fix the variance is to scale the linearly blended result by the inverse of the lost standard deviation.

However, by scaling the result, we scale not only its variance but also its expectation, which means that the average color is not preserved anymore.

Can we solve this contrast (variance) problem?

1. linear blending

$$X^{\text{blended}} = \frac{\sum_{i=1}^N w_i X_i - \mathbb{E}[X_i]}{\sqrt{\sum_{i=1}^N w_i^2}} + \mathbb{E}[X_i]$$

2. center on 0

3. fix variance

4. center on expectation

$$\mathbb{E}[X^{\text{blended}}] = \mathbb{E}[X_i] \quad \checkmark$$

$$\text{var}(X^{\text{blended}}) = \text{var}(X_i) \quad \checkmark$$

1. linear blending

$$X^{\text{blended}} = \frac{\sum_{i=1}^N w_i X_i - \mathbb{E}[X_i]}{\sqrt{\sum_{i=1}^N w_i^2}} + \mathbb{E}[X_i]$$

2. center on 0

3. fix variance

4. center on expectation

$$\mathbb{E}[X^{\text{blended}}] = \mathbb{E}[X_i] \quad \checkmark$$

$$\text{var}(X^{\text{blended}}) = \text{var}(X_i) \quad \checkmark$$

In order to obtain at the same time the correct expectation (average color) and the correct variance, one can proceed in the following way:

1. Compute the linear blending.
2. Remove the average value such that the values are statistically centered around 0.
3. Fix the variance by scaling the values. Since the expectation at this point is 0 it remains 0 after scaling.
4. Offset the values to center them on the expectation again. Since, offsetting does not affect the variance, we obtain a blended value with the correct variance and the correct expectation.

Can we solve this contrast (variance) problem?

1. linear blending

2. center on 0

3. fix variance

4. center on expectation

$$X^{\text{blended}} = \frac{\sum_{i=1}^N w_i X_i - \mathbb{E}[X_i]}{\sqrt{\sum_{i=1}^N w_i^2}} + \mathbb{E}[X_i]$$

This is a **variance-preserving blending** operator.

1. linear blending

2. center on 0

3. fix variance

4. center on expectation

$$X^{\text{blended}} = \frac{\sum_{i=1}^N w_i X_i - \mathbb{E}[X_i]}{\sqrt{\sum_{i=1}^N w_i^2}} + \mathbb{E}[X_i]$$

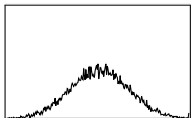
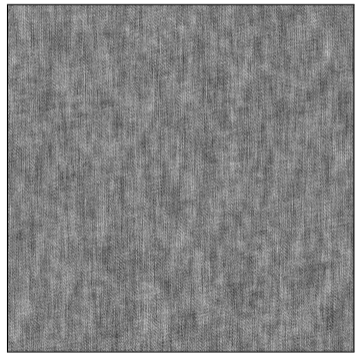
This is a variance-preserving blending operator.

This is called a variance-preserving blending operator because it preserves the variance in addition to the expectation (already preserved by linear blending).

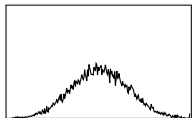
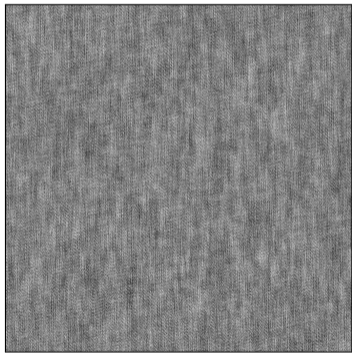
Note that we are not the inventor of variance-preserving blending. This concept appears in several previous papers and is simple enough to be considered common wisdom.

Can we solve this contrast (variance) problem?

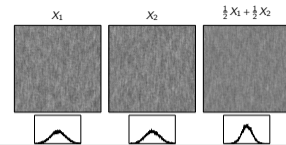
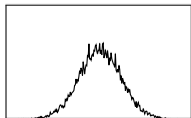
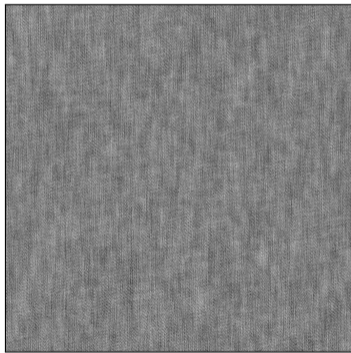
X_1



X_2



$\frac{1}{2} X_1 + \frac{1}{2} X_2$

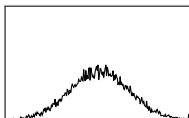
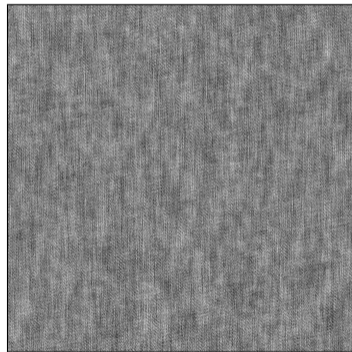


We can give it a try on this example.

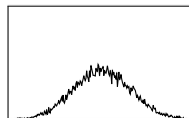
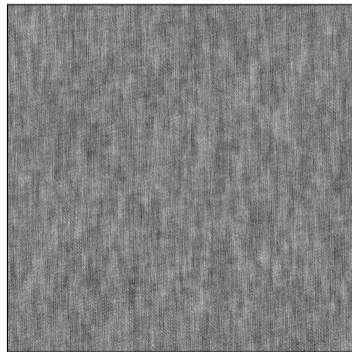
As we have seen before, linear blending reduces the contrast.

Can we solve this contrast (variance) problem?

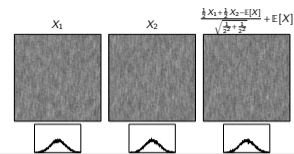
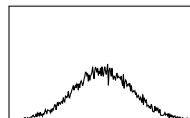
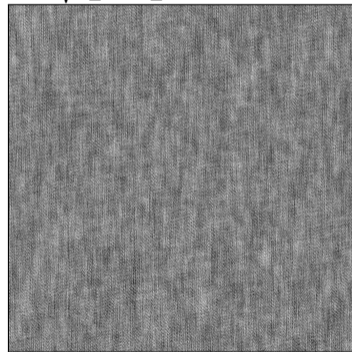
X_1



X_2



$$\frac{\frac{1}{2} X_1 + \frac{1}{2} X_2 - \mathbb{E}[X]}{\sqrt{\frac{1}{2^2} + \frac{1}{2^2}}} + \mathbb{E}[X]$$



But variance-preserving blending preserves the contrast.

The blended result looks good! With variance-preserving blending, we are thus able to solve this contrast problem.

Is it really just about contrast?

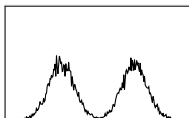
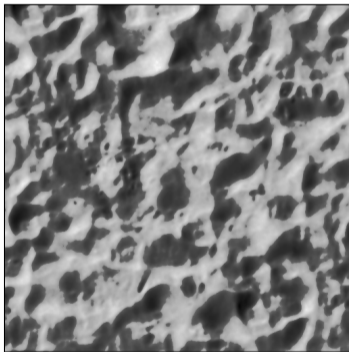
Is it really just about contrast?

But did we really solve the problem of blending?

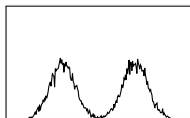
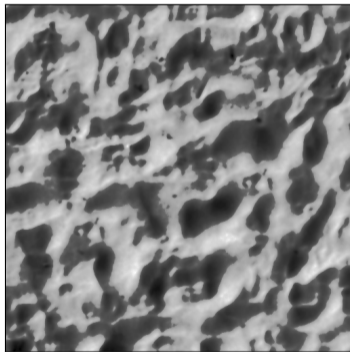
Is this going to work on more challenging examples?

Is it really just about contrast?

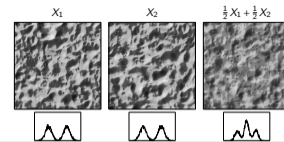
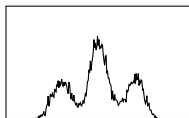
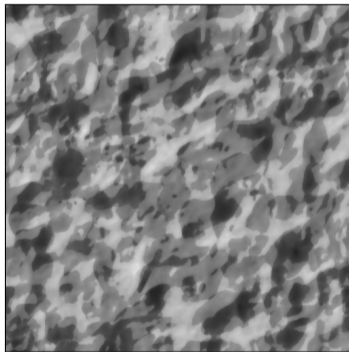
X_1



X_2



$\frac{1}{2} X_1 + \frac{1}{2} X_2$

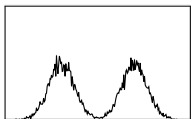
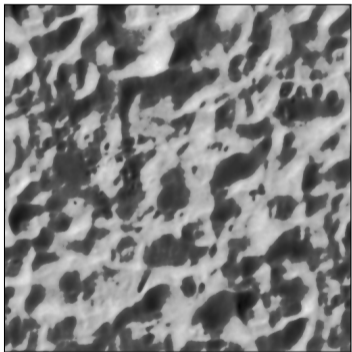


This example is more challenging because, as we can see in the histogram, it has two dominant shades of gray: the histogram is bimodal.

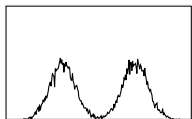
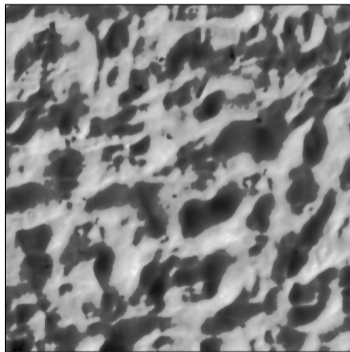
As expected, the linearly blended result has less contrast. But this is not the only problem: we also notice ghosting artifacts and new gray-scale levels that were not present in the inputs. This makes the blended result have a different appearance from the input.

Is it really just about contrast?

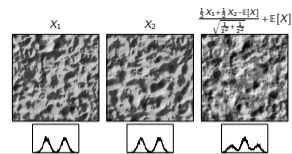
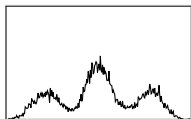
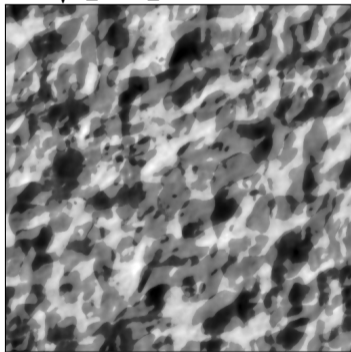
X_1



X_2

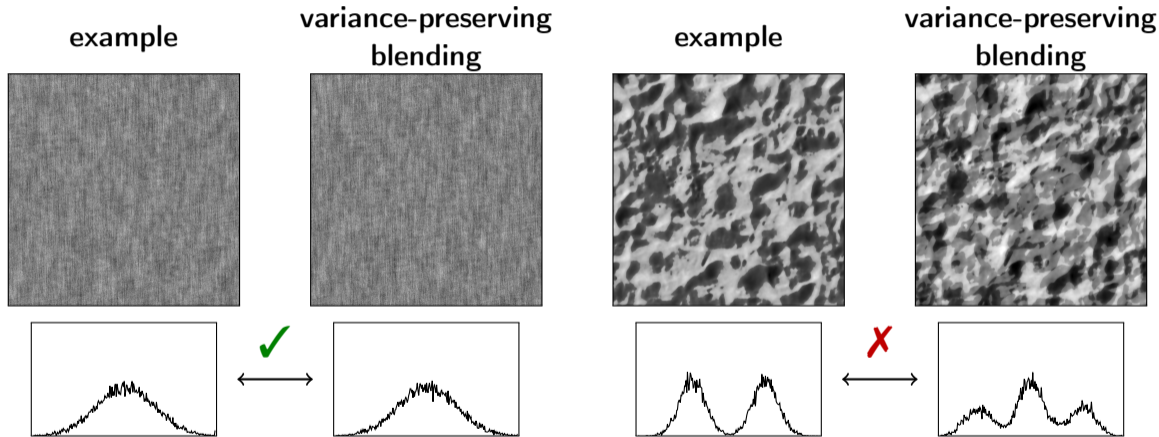


$$\frac{\frac{1}{2} X_1 + \frac{1}{2} X_2 - \mathbb{E}[X]}{\sqrt{\frac{1}{2^2} + \frac{1}{2^2}}} + \mathbb{E}[X]$$

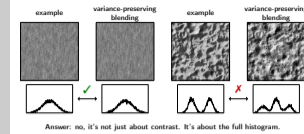


If we use variance-preserving blending, we fix the contrast: the blended image has now the same contrast as the input in the sense that its variance is the same. However, the appearance of result is still obviously wrong.

Is it really just about contrast?



Answer: no, it's not just about contrast. It's about the full histogram.

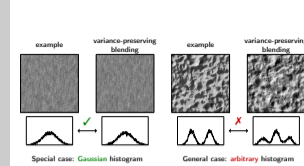
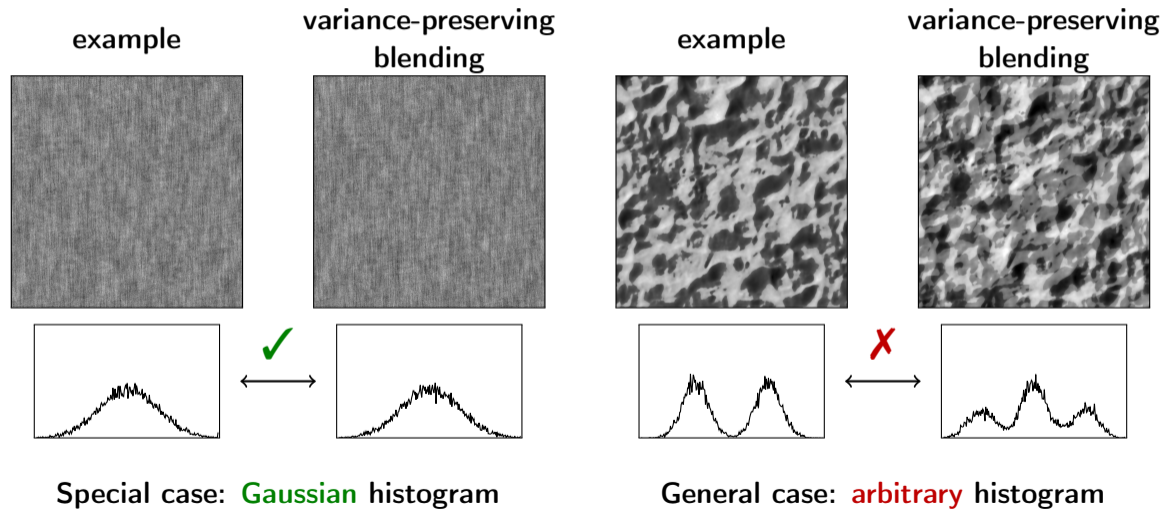


We had one example on which the variance-preserving blending performed well and one example on which it performed poorly. By looking at the histograms, we can see that the quality of the blended result seems to be related to how much its histogram is close to the histogram of the input.

In the first case, the histogram is exactly the same and the result looks good. In the second case, the histogram is different and the result looks wrong.

The histogram contains much more statistical information than just the variance. Obtaining the same histogram means that not only we obtain the same variance/contrast but also many more statistical descriptors. The closer the histograms, the higher the chances of approaching the appearance of the input.

Is it really just about contrast?



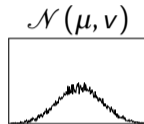
An interesting observation is that the variance-preserving blending was able to preserve the full shape of the histogram in the first example but not in the second.

This is because the first example is a special case: its histogram is Gaussian (also called Normal, or a bell-curve).

Is it really just about contrast?

Special case:

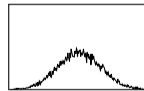
Independent **Gaussian** random variables: G_1, \dots, G_n




Blending weights: W_1, \dots, W_n

Variance-preserving blending = histogram-preserving blending

$$\frac{w_1 G_1 + \dots + w_n G_n - \mathbb{E}[G]}{\sqrt{w_1^2 + \dots + w_n^2}} + \mathbb{E}[G] \longrightarrow$$




Special case:

Independent **Gaussian** random variables: $G_1, \dots, G_n \longrightarrow$ 

Blending weights: W_1, \dots, W_n

Variance-preserving blending = histogram-preserving blending

$$\frac{w_1 G_1 + \dots + w_n G_n - \mathbb{E}[G]}{\sqrt{w_1^2 + \dots + w_n^2}} + \mathbb{E}[G] \longrightarrow$$


The Gaussian distribution is a special case where variance-preserving blending is the same as full histogram-preserving blending.

Proof: this is easy to demonstrate using a random-variable analogy. Adding random variables means convolving their distributions/histograms and the Gaussian distribution has a specific property: it is invariant under convolution.

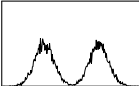
→ Take as many Gaussian distributions as you want and convolve them: the result is a Gaussian distribution.

→ Take as many Gaussian random variables as you want and add them: the result is a Gaussian random variable.

Furthermore, if we make sure that the output has the same expectation and the same variance (i.e. variance-preserving) we obtain the same Gaussian (it is parameterized only by expectation and variance).

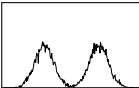
Is it really just about contrast?

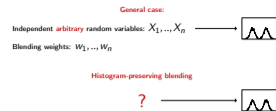
General case:

Independent **arbitrary** random variables: X_1, \dots, X_n \longrightarrow 

Blending weights: W_1, \dots, W_n

Histogram-preserving blending

? \longrightarrow 



However, we would like to have a solution for the general case. Our objective is to be able to blend arbitrary inputs, not only Gaussian. Hence, we are looking for an histogram-preserving blending operator that works with arbitrary input.

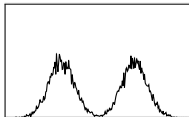
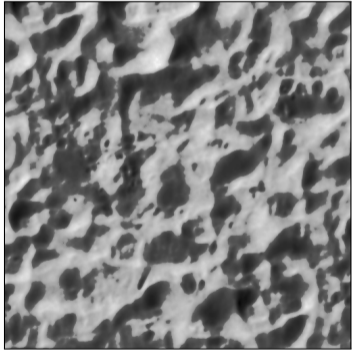
Histogram-preserving blending

We are now going to introduce our histogram-preserving blending operator.

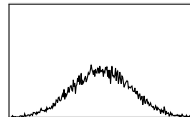
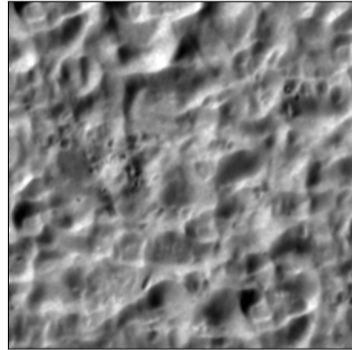
This is the main technical contribution of our paper.

Histogram-preserving blending

non Gaussian, hard to blend



Gaussian, simple to blend



non Gaussian, hard to blend



Gaussian, simple to blend



It starts with the following observation. Consider this input. It is hard to blend because it is non-Gaussian. But if it were Gaussian we would know how to blend it.

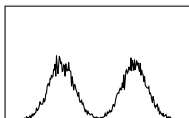
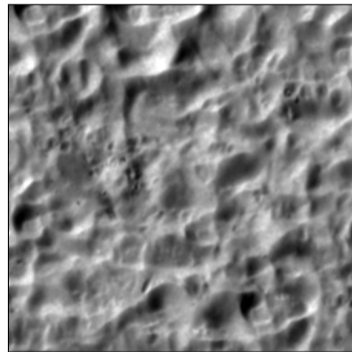
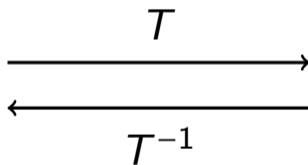
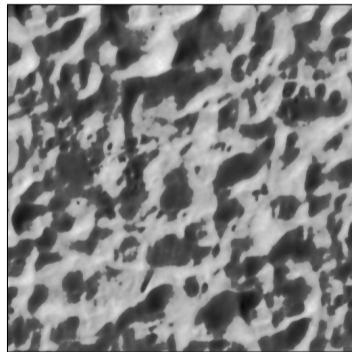
Can we make this happen?

Histogram-preserving blending

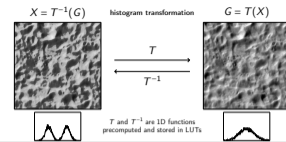
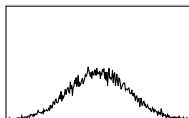
$$X = T^{-1}(G)$$

histogram transformation

$$G = T(X)$$



T and T^{-1} are 1D functions precomputed and stored in LUTs



Of course we can. All we have to do is a simple histogram transformation, which is a classic image-processing operation.

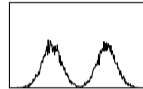
A histogram transformation can be seen as function T that maps non-Gaussian pixel values to Gaussian pixel values.

Once the user provides an example, we compute its histogram and its associated transformations T and T^{-1} , which we store in look-up tables (LUTs). For now on, we consider that these functions are always at our disposal.

Histogram-preserving blending

Input:

X_1, \dots, X_n



Input:

X_1, \dots, X_n



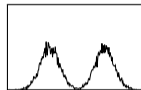
Our histogram-preserving blending works in the following way.

The input is a set of non-Gaussian values (in our procedural texturing application they would be color values from the patches chosen randomly in the example image).

Histogram-preserving blending

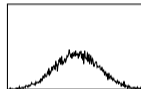
Input:

$$X_1, \dots, X_n$$



1. "Gaussianize" (fetch LUT):

$$T(X_1), \dots, T(X_n)$$



Input:

X_1, \dots, X_n



1. "Gaussianize" (fetch LUT):

$T(X_1), \dots, T(X_n)$

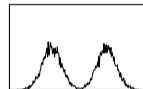


The first step consists in "Gaussianizing" these values by applying the histogram transformation (precomputed and available as a LUT). Now, we have a set of Gaussian values.

Histogram-preserving blending

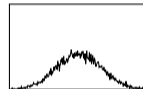
Input:

$$X_1, \dots, X_n$$



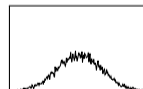
1. "Gaussianize" (fetch LUT):

$$T(X_1), \dots, T(X_n)$$



2. Variance-preserving blending:

$$\frac{w_1 T(X_1) + \dots + w_n T(X_n)}{\sqrt{w_1^2 + \dots + w_n^2}}$$



Input:	X_1, \dots, X_n	
1. "Gaussianize" (fetch LUT):	$T(X_1), \dots, T(X_n)$	
2. Variance-preserving blending:	$\frac{w_1 T(X_1) + \dots + w_n T(X_n)}{\sqrt{w_1^2 + \dots + w_n^2}}$	

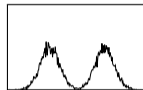
Since the values are now Gaussian we know how that we can blend them using variance-preserving blending. The blended result is distributed in the same Gaussian.

(Note that to simplify the notations, we use a Gaussian centered on 0. Hence, the null expectation does not appear in the expression).

Histogram-preserving blending

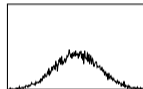
Input:

$$X_1, \dots, X_n$$



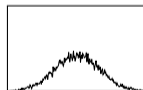
1. "Gaussianize" (fetch LUT):

$$T(X_1), \dots, T(X_n)$$



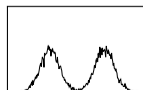
2. Variance-preserving blending:





$$\frac{w_1 T(X_1) + \dots + w_n T(X_n)}{\sqrt{w_1^2 + \dots + w_n^2}}$$



3. "unGaussianize" (fetch LUT):

$$T^{-1} \left(\frac{w_1 T(X_1) + \dots + w_n T(X_n)}{\sqrt{w_1^2 + \dots + w_n^2}} \right)$$



Input:	X_1, \dots, X_n	
1. "Gaussianize" (fetch LUT):	$T(X_1), \dots, T(X_n)$	
2. Variance-preserving blending:	$\frac{w_1 T(X_1) + \dots + w_n T(X_n)}{\sqrt{w_1^2 + \dots + w_n^2}}$	
3. "unGaussianize" (fetch LUT):	$T^{-1} \left(\frac{w_1 T(X_1) + \dots + w_n T(X_n)}{\sqrt{w_1^2 + \dots + w_n^2}} \right)$	

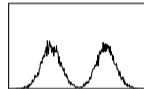
Finally, we remap the Gaussian result to the original histogram by using the inverse histogram transformation (also available as a LUT).

Histogram-preserving blending

General case:

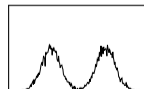
Independent **arbitrary** random variables: X_1, \dots, X_n

Blending weights: W_1, \dots, W_n



Histogram-preserving blending:

$$T^{-1} \left(\frac{w_1 T(X_1) + \dots + w_n T(X_n)}{\sqrt{w_1^2 + \dots + w_n^2}} \right)$$



General case:
Independent **arbitrary** random variables: X_1, \dots, X_n
Blending weights: W_1, \dots, W_n



Histogram-preserving blending:
 $T^{-1} \left(\frac{w_1 T(X_1) + \dots + w_n T(X_n)}{\sqrt{w_1^2 + \dots + w_n^2}} \right)$

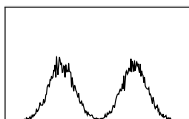


This expression is our histogram-preserving blending operator. It blends inputs with arbitrary histograms such that the output has statistically the same histogram.

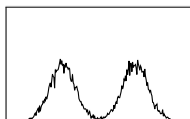
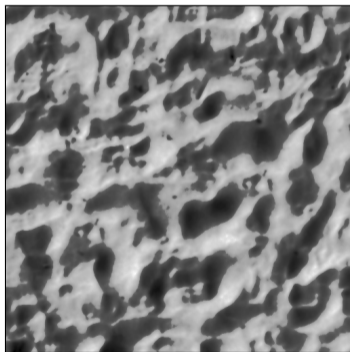
We are going to give it a try.

Histogram-preserving blending

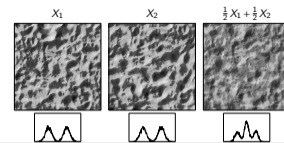
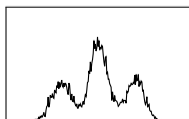
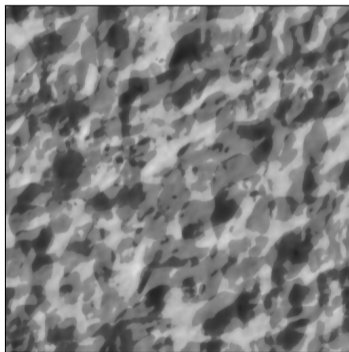
X_1



X_2



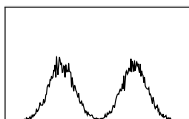
$\frac{1}{2} X_1 + \frac{1}{2} X_2$



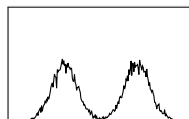
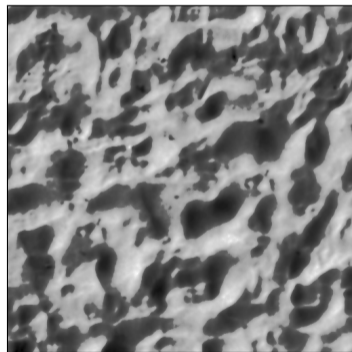
That's linear blending with the problems discussed before.

Histogram-preserving blending

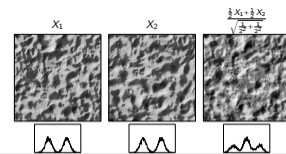
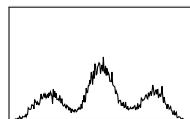
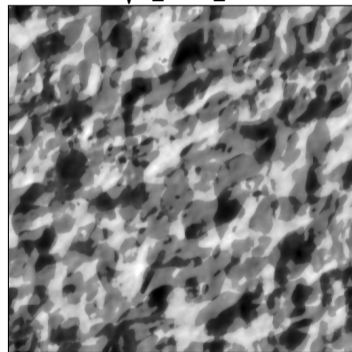
X_1



X_2



$$\frac{\frac{1}{2} X_1 + \frac{1}{2} X_2}{\sqrt{\frac{1}{2^2} + \frac{1}{2^2}}}$$



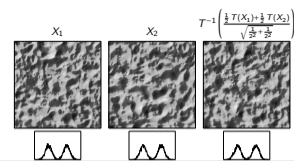
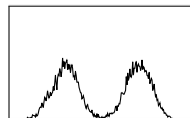
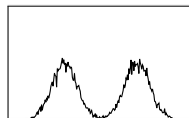
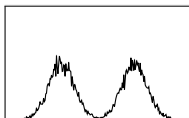
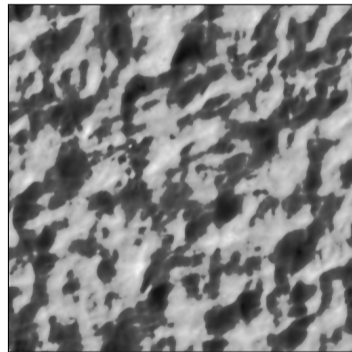
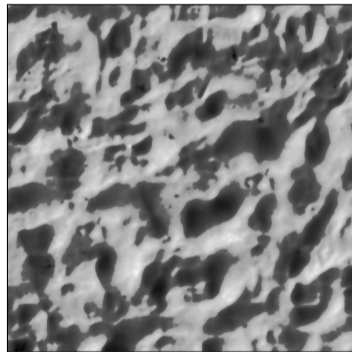
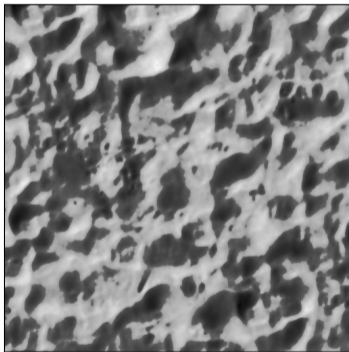
That's variance-preserving blending with the problems discussed before.

Histogram-preserving blending

X_1

X_2

$$T^{-1} \left(\frac{\frac{1}{2} T(X_1) + \frac{1}{2} T(X_2)}{\sqrt{\frac{1}{2^2} + \frac{1}{2^2}}} \right)$$



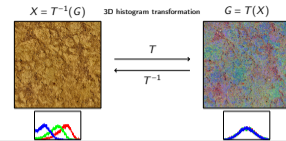
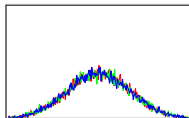
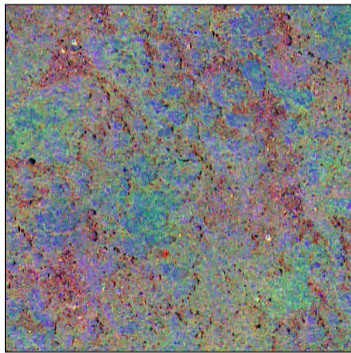
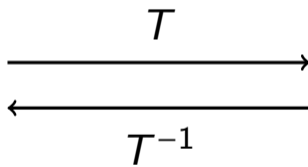
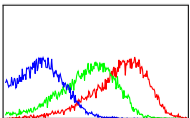
That's our histogram-preserving blending. The result looks much better. The wrong gray-scale levels have disappeared and the resulting image has the same appearance as the input.

Histogram-preserving blending

$$X = T^{-1}(G)$$

3D histogram transformation

$$G = T(X)$$



We are almost set up to go back to procedural texturing!

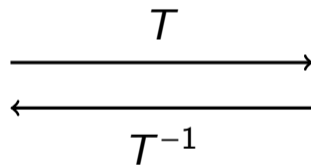
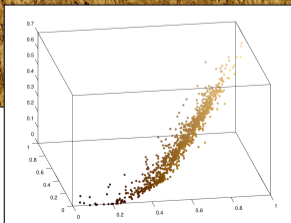
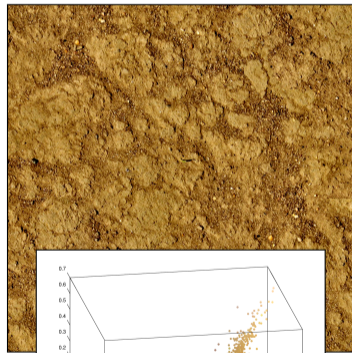
There is one last thing we need to discuss: multi-channel inputs. So far, we have made our analysis on mono-channel (gray-scale) images. With multi-channel images, the principle of our blending operator is the same but we have to be careful about how we compute the histogram transformation.

Histogram-preserving blending

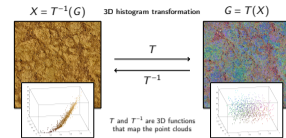
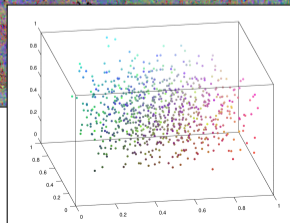
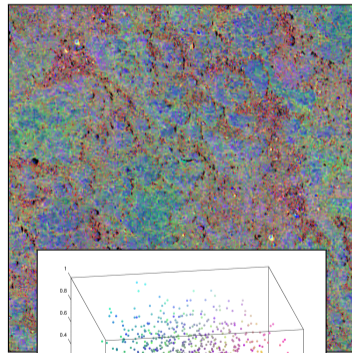
$$X = T^{-1}(G)$$

3D histogram transformation

$$G = T(X)$$



T and T^{-1} are 3D functions that map the point clouds



Indeed, the histogram of an RGB image should not be seen as 3 1D function but rather as one 3D function or a 3D point cloud. This function might have some inter-channel correlations, i.e. the full 3D shape is not just the composition of 3 1D shapes. Applying a 1D histogram transformation on each channel separately might lose some of these inter-channel correlations.

We thus need to use a truly 3D histogram transformation that is able to account for these correlations correctly. Fortunately, this is a well-documented image-processing problem and there are several existing solutions. We discuss them in the paper.

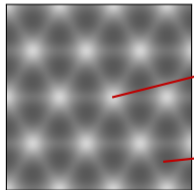
Besides carefully choosing the histogram transformation, the rest of the algorithm is the same as before.

Our procedural texturing method

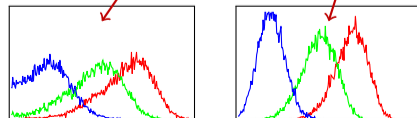
Now we have all the ingredients to fix our procedural texturing method.

Our procedural texturing method

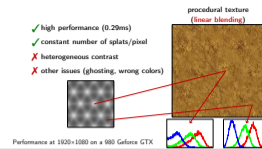
- ✓ high performance (0.29ms)
- ✓ constant number of splats/pixel
- ✗ heterogeneous contrast
- ✗ other issues (ghosting, wrong colors)



procedural texture
(linear blending)



Performance at 1920×1080 on a 980 Geforce GTX

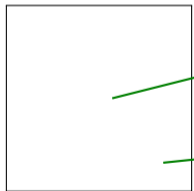


As we have seen in the introduction, this tiling-and-blending approach produces heterogeneous contrast that appears as a grid-revealing pattern.

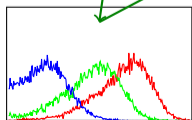
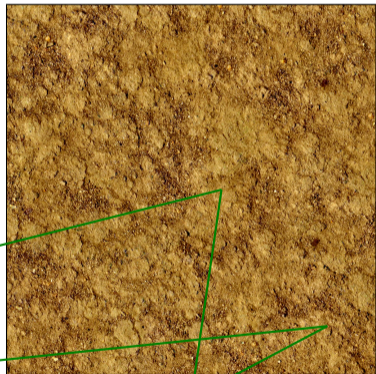
With our investigations on blending we now understand why: the expected histogram depends on the position within the triangle because the values of the blending weights impact the final shape of the histogram, decreasing the contrast and potentially introducing other artifacts, such as wrong gray-scale levels (or colors).

Our procedural texturing method

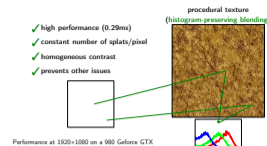
- ✓ high performance (0.29ms)
- ✓ constant number of splats/pixel
- ✓ homogeneous contrast
- ✓ prevents other issues



procedural texture
(histogram-preserving blending)



Performance at 1920×1080 on a 980 Geforce GTX

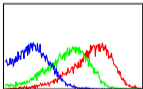


With our histogram-preserving blending, the expected histogram does not depend on the position in the triangle grid anymore and the procedural texture is artifact free!

We will now have an overview of our complete algorithm.

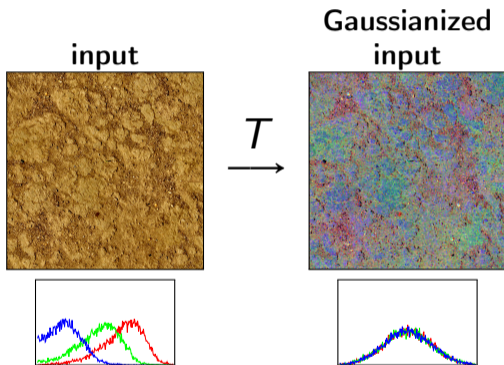
Our procedural texturing method

input

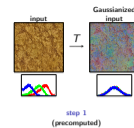


First, we get an input from the user.

Our procedural texturing method

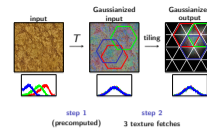
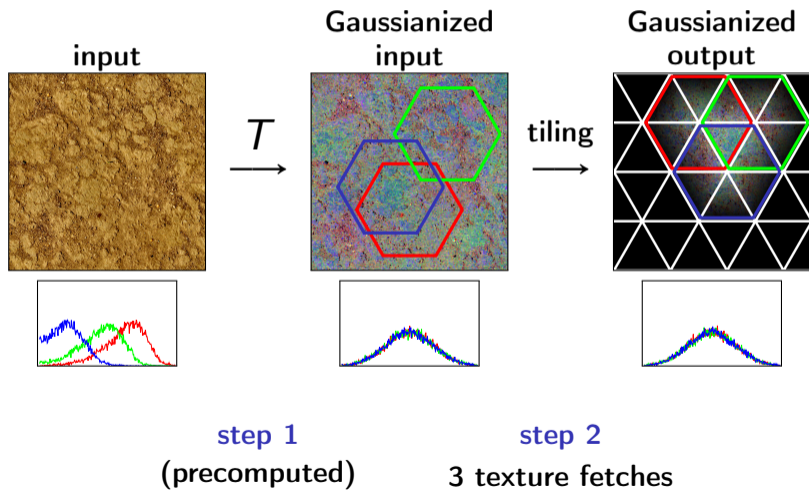


step 1
(precomputed)



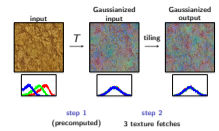
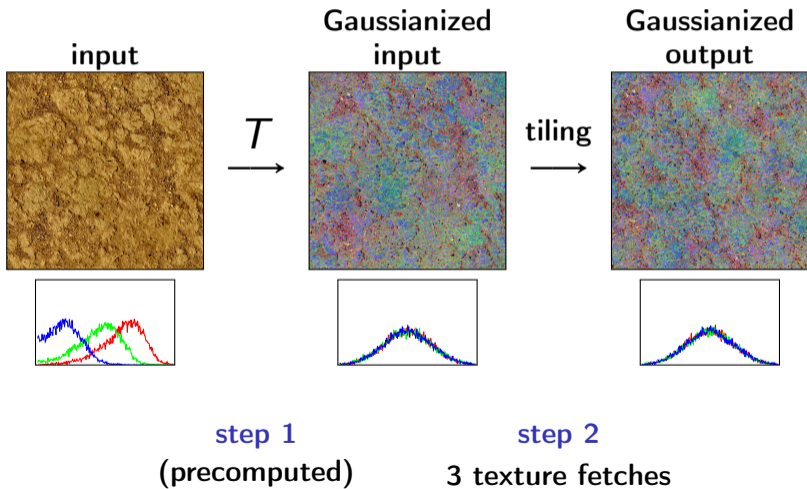
The first step consists in computing the histogram transformation and use it to "Gaussianize" the input. This is done on the CPU side as a preprocess. We store the Gaussian input in a texture on the GPU.

Our procedural texturing method



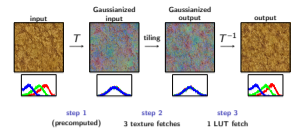
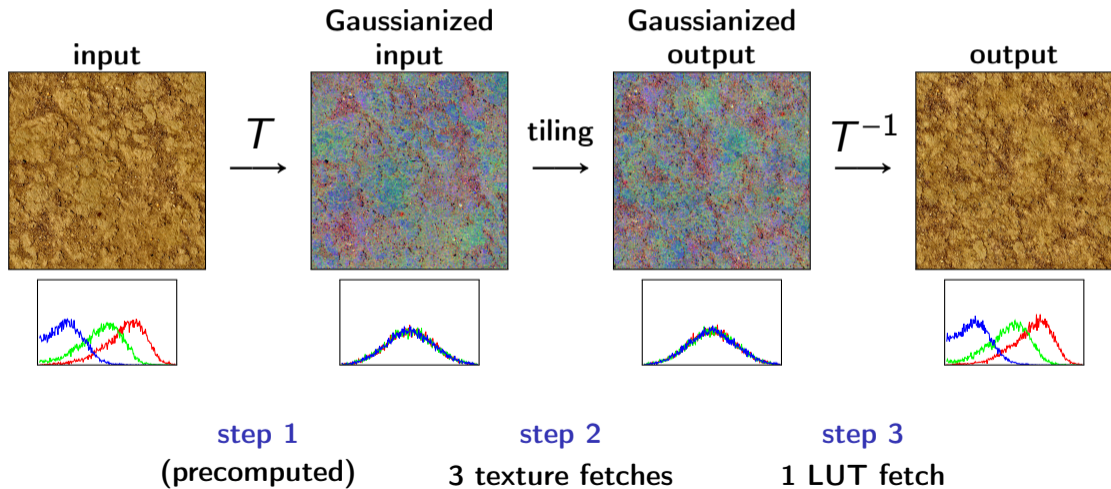
Now we enter the fragment shader and use the tiling-and-blending algorithm with the Gaussian input and variance-preserving blending.

Our procedural texturing method



At this point, the fragment shader has computed a Gaussian color value.

Our procedural texturing method

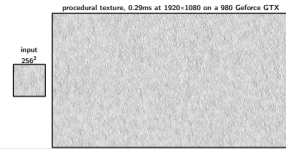
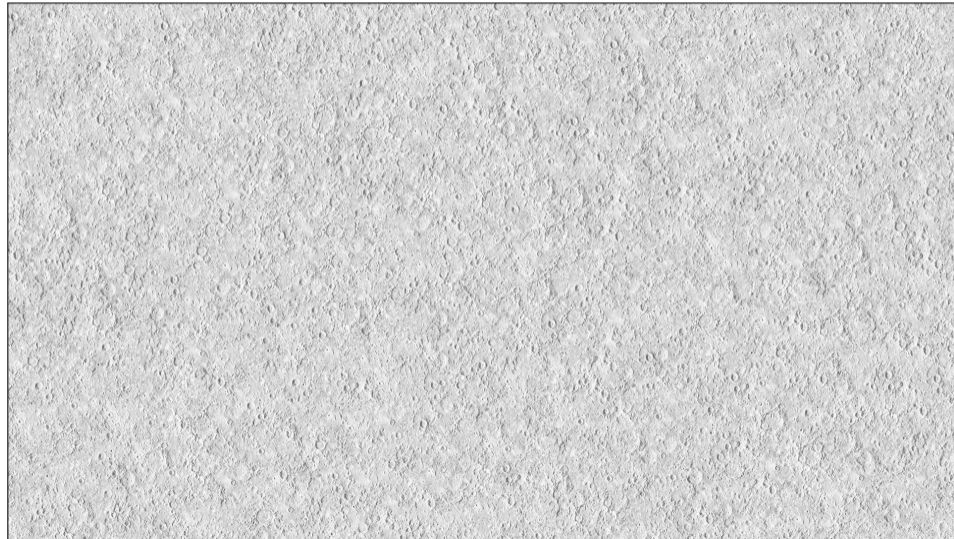


The final step of the fragment shader consists in fetching the inverse histogram transformation T^{-1} that we precomputed and stored in a LUT.

We obtain a color value that is statistically distributed in the same histogram as the input. This is the result returned by the fragment shader.

Our procedural texturing method

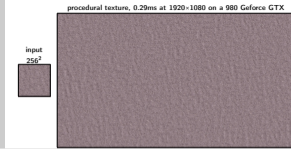
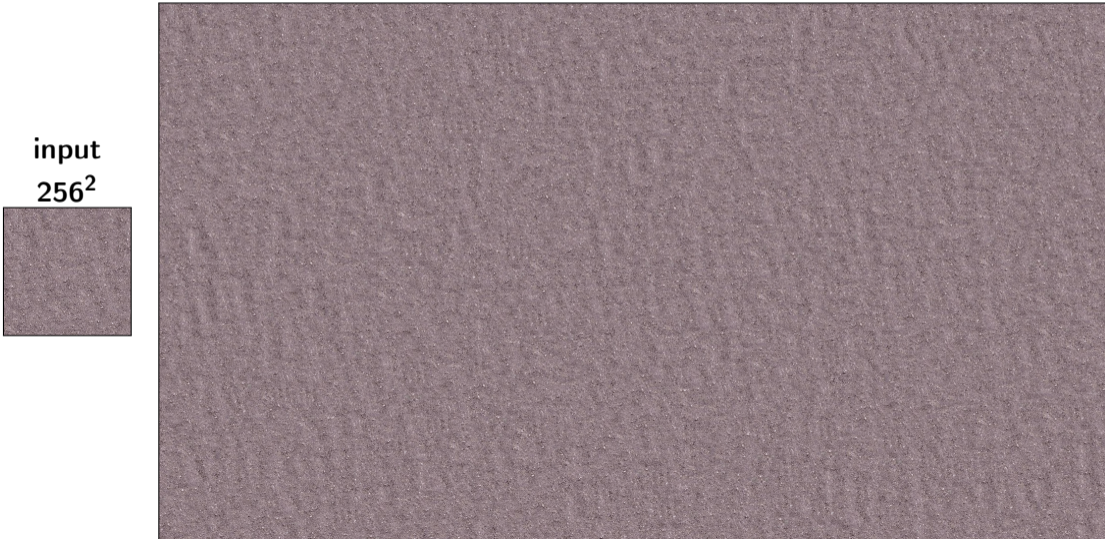
procedural texture, 0.29ms at 1920×1080 on a 980 Geforce GTX



Here are some examples obtained with our method.

Our procedural texturing method

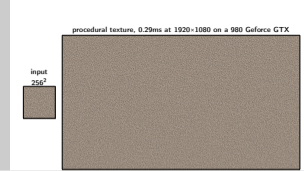
procedural texture, 0.29ms at 1920×1080 on a 980 Geforce GTX



Here are some examples obtained with our method.

Our procedural texturing method

procedural texture, 0.29ms at 1920×1080 on a 980 Geforce GTX

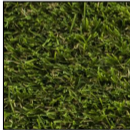


Here are some examples obtained with our method.

Our procedural texturing method

procedural texture, 0.29ms at 1920×1080 on a 980 Geforce GTX

input
256²



Here are some examples obtained with our method.

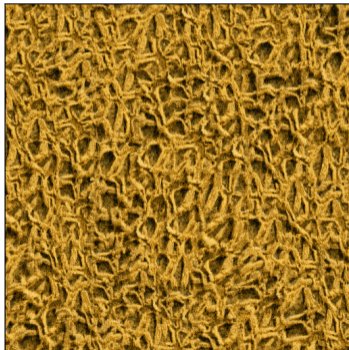
Our procedural texturing method

example



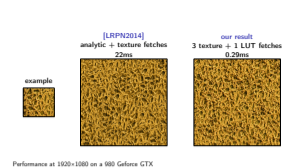
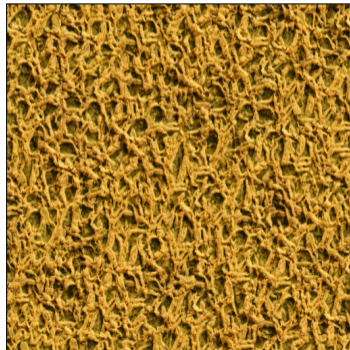
[LRPN2014]

analytic + texture fetches
22ms



our result

3 texture + 1 LUT fetches
0.29ms

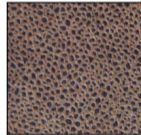


This is a comparison against LRPN on a stochastic but structured example. We achieve similar visual quality 75× faster.

Note that the example and the LRPN result are from the LRPN paper.

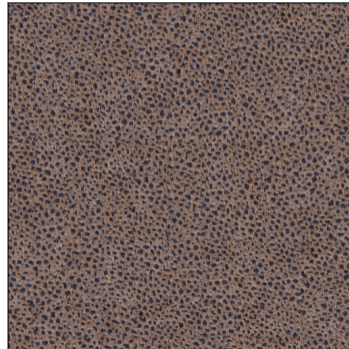
Our procedural texturing method

example



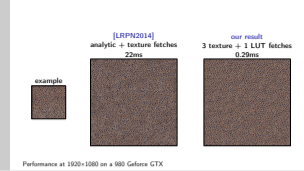
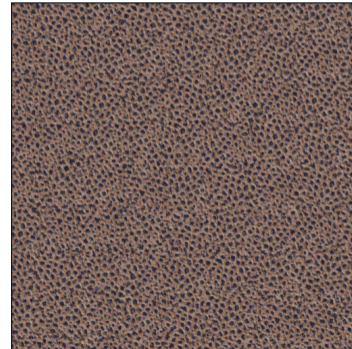
[LRPN2014]

analytic + texture fetches
22ms



our result

3 texture + 1 LUT fetches
0.29ms



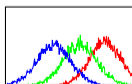
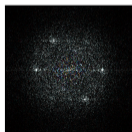
This is a comparison against LRPN on a stochastic but structured example. We achieve similar visual quality 75× faster.

Note that the example and the LRPN result are from the LRPN paper.

Our procedural texturing method

procedural texture, **0.29ms** **0.11ms** at 1920×1080 on a 980 Geforce GTX

Gaussian
example
 256^2

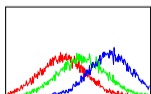
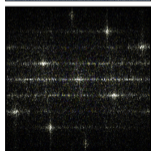
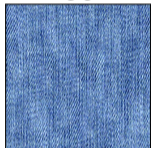


In the special case where the input is Gaussian, we can do a lot of simplifications to our algorithm. Indeed, we put a lot of work into making nonGaussian data become Gaussian. If they are already Gaussian, we can spare the calculations and the performance is 0.11ms instead of 0.29ms.

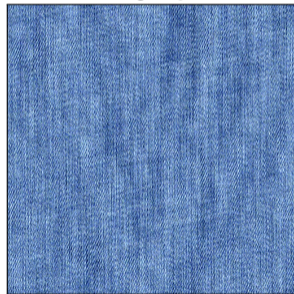
This special case is actually representative of an important class of textures: random-phase textures. They are textures that are defined only by their power spectrum and they are always Gaussian. These textures are the focus of Gabor Noise by Example and Texton Noise, for instance.

Our procedural texturing method

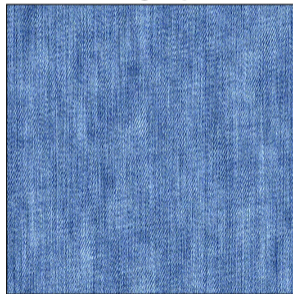
Gaussian example
256²



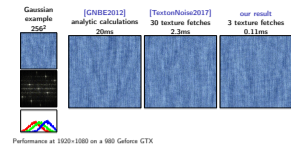
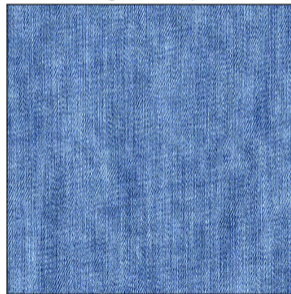
[GNBE2012]
analytic calculations
20ms



[TextonNoise2017]
30 texture fetches
2.3ms



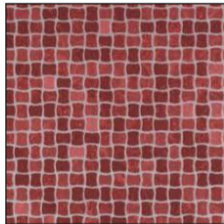
our result
3 texture fetches
0.11ms



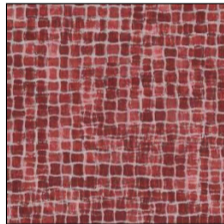
With Gaussian examples, our method computes results of the same quality with a 20× speed up compared to Texton Noise.

Our procedural texturing method

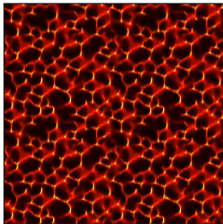
example



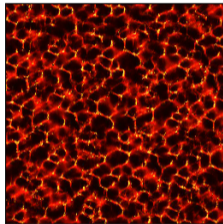
our result



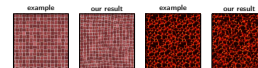
example



our result



Same failure cases as previous work: strong patterns and spatial correlations.



Same failure cases as previous work: strong patterns and spatial correlations.

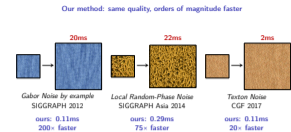
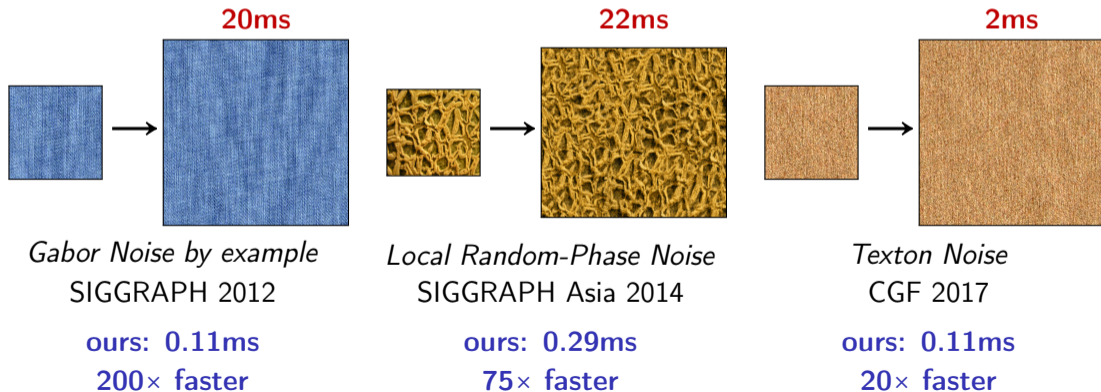
Our approach is meant for stochastic textures. If the input has strong spatial correlations (like repetition or strong patterns) our approach fails.

This is because our histogram-preserving blending operator is based on the assumption that the blending takes independent inputs. This is not the case with strong spatial correlations.

To be fair, these failure cases are not specific to our approach. Textures with strong patterns are challenging in general and we are not aware of any satisfying previous work for these cases. They remain an open problem for by-example procedural texturing.

Our procedural texturing method

Our method: same quality, orders of magnitude faster



In summary, we claim that our algorithm can produce the same textural generative space with the same quality as the previous work but with much higher performance.



Finally, what we obtained is super simple. We started from a very small piece of shader code that had some appearance problems and we fixed it, doing few modifications. The result is as fast as a cheap hack but outputs the same quality as the academic state of the art. With this kind of performance, we hope that we have brought by-example procedural textures to true real time.

This could be the last slide of the talk but it's not over...

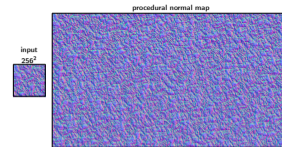
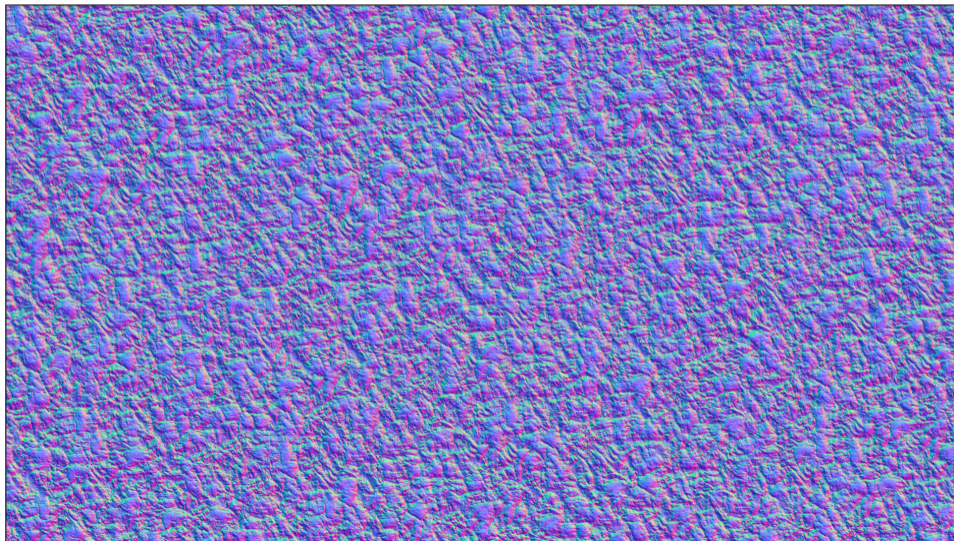
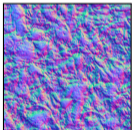
Other applications of histogram-preserving blending

Now we are done talking about procedural textures, we can tell you what this paper is *really* about. To us, this paper is not so much about procedural textures, the important contribution is rather the general concept of histogram-preserving blending, which, we believe, can be useful for many other applications.

Other applications of histogram-preserving blending

procedural normal map

input
 256^2



The first obvious extension would be to blend other things than color textures: normal maps, roughness maps, etc.

Other applications of histogram-preserving blending

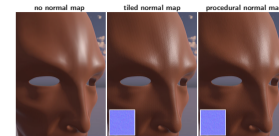
no normal map



tilled normal map



procedural normal map



In this example, we generate an infinite and non-repetitive procedural detail map using our method.

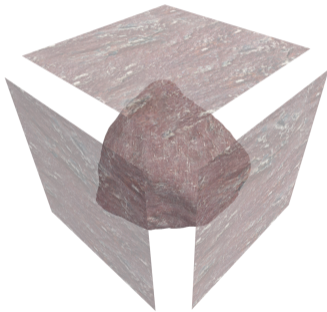
But this an obvious extension. Let's see some more interesting stuff.

Other applications of histogram-preserving blending

texture



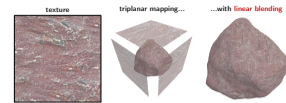
triplanar mapping...



...with **linear blending**



Application: texturing uv-free meshes with triplanar mapping.



Application: texturing uv-free meshes with triplanar mapping.

Triplanar mapping is a very interesting use case for histogram-preserving blending.

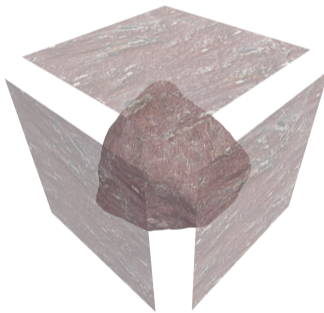
Consider this uv-free mesh. Triplanar mapping provides an easy way to texture it by projecting three textures on the X , Y and Z axes and blend them using the surface's normal coordinates. Triplanar mapping is usually done using linear blending and thus inherits its typical problems: low contrast, wrong colors, etc.

Other applications of histogram-preserving blending

texture



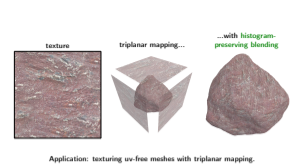
triplanar mapping...



...with histogram-preserving blending



Application: texturing uv-free meshes with triplanar mapping.



Our histogram-preserving blending operator can be used to obtain an triplanar-mapping result.

Other applications of histogram-preserving blending

Hashed Alpha Testing

Chris Wyman*
NVIDIA

Morgan McGuire
NVIDIA & Williams College

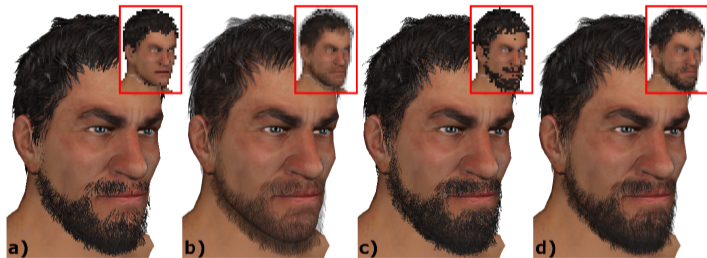
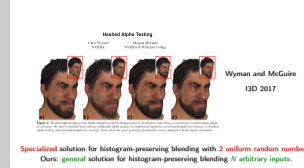


Figure 1: Hashed alpha testing avoids alpha-mapped geometry disappearing in the distance and reduces correlations in multisample alpha-to-coverage. We show a bearded man with (a) traditional alpha testing, (b) traditional, hardware-accelerated alpha-to-coverage, (c) hashed alpha testing, and (d) hashed alpha-to-coverage. Insets show the same geometry from further away, enlarged to better depict variations.

Wyman and McGuire

I3D 2017



Another interesting example we would like to mention is this stochastic alpha-testing paper. The method uses random numbers generated in object space and that are made coherent in screen-space. To achieve this, their algorithm interpolates between uniform random numbers. However, the authors noticed that the interpolated result was not uniform anymore and it was a problem for their algorithm. To solve this problem, they worked out a specialized solution to interpolate two uniform random numbers in such a way that the result is also a uniform random number. In other words, they worked out an histogram-preserving blending operator for this specific case. This example shows that the problem of histogram-preserving blending is relevant not only for texturing applications.

Our solution addresses the general case of histogram-preserving blending.

Specialized solution for histogram-preserving blending with **2 uniform random numbers**.
Ours: **general** solution for histogram-preserving blending **N arbitrary inputs**.

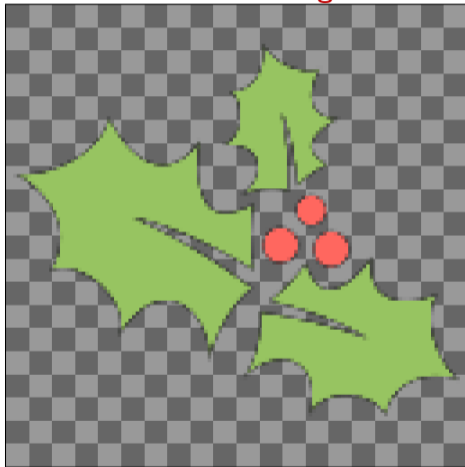
The importance of getting blending right

Blending is often trickier than what you think and linear blending is often not the right way.

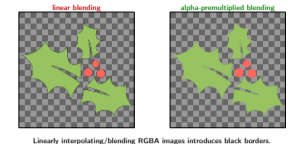
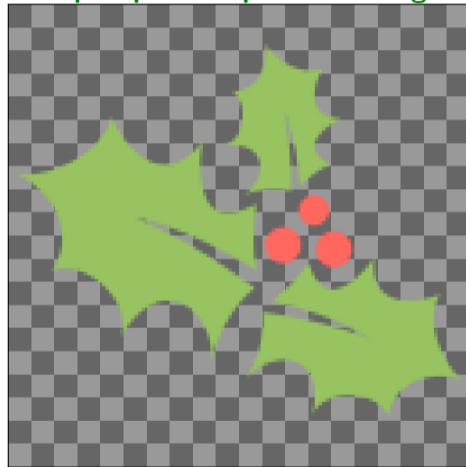
There is another non-linear blending operator that is super famous in computer graphics and that everyone should know about.

The importance of getting blending right

linear blending



alpha-premultiplied blending



Linearly interpolating/blending RGBA images introduces black borders.

It is premultiplied blending.

Its typical application is when you blend (interpolate, magnify, mipmap) RGBA data. If you blend them linearly, you might be mixing colors that do not have the same transparency. For instance, at the border, you are mixing the color of the opaque object with the (undefined and often set to 0) color of the transparent background. This results in wrong dark edges.

The solution is premultiplied-blending: multiply the RGB color data by their alpha value, then blend, and then divide by the blended alpha value. The result has the correct color.

But the applications of premultiplied-blending range way beyond RGBA data. It is pretty amazing to see how often people discover that a bug could be fixed using premultiplied blending.

The importance of getting blending right

Blending an HDR color into a U8 Buffer
Blogpost by Alan Wolfe 'demofox', July 2018



Problem: wrong exposition
Guilty: blending
Solution: premultiplied blending



Blending an HDR color into a U8 Buffer
Blogpost by Alan Wolfe 'demofox', July 2018

Problem: wrong exposition
Guilty: blending
Solution: premultiplied blending

This is an example from last month. In this blogpost, the problem of blending HDR color into a LDR buffer is studied. The naive approach with linear blending resulted in some exposition problems.

The solution: premultiplied blending.

The importance of getting blending right

The team color problem
Blogpost by Ben Golus, July 2018



Problem: color leaks
Guilty: blending
Solution: premultiplied blending



Another example, also from last month! How to dynamically change the color of some parts of an asset dynamically without changing the texture? Doing this with linear blending results in color-leaking artifacts.

The solution: premultiplied blending.

The importance of getting blending right



Marc Olano

@olanom

And yet again, premultiplied alpha is the answer. Premultiplied alpha is always the answer



Marc Olano
@olanom

And yet again, premultiplied alpha is the answer. Premultiplied alpha is always the answer

Premultiplied blending is the solution to so many practical bugs that sometimes it seems that it is always the solution. Why is that?

It's because blending is one of the most fundamental computer-graphics operation. We use blending everywhere and it seems so natural that we don't even pay attention to it. Unfortunately, naive linear blending is often not the right way. Blending is everywhere and sometimes trickier than you would assume. This is why it's such a big source of bugs.

Identifying the blending cases that require more than linear blending is thus of high importance.

The importance of getting blending right

Common wisdom

Whenever you blend **weighted** data, use **premultiplied blending**.

This prevents **leaking and mixing with undefined values**.

Common wisdom
Whenever you blend **weighted** data, use **premultiplied blending**.
This prevents **leaking and mixing with undefined values**.

The cases where premultiplied blending is required are the cases when you blend weighted data. Being aware of this is so important that you could call it "common wisdom".

The importance of getting blending right

Common wisdom

Whenever you blend **weighted** data, use **premultiplied blending**.

This prevents **leaking and mixing with undefined values**.

Main takeaway of this paper

Whenever you blend data chosen **randomly**, use **histogram-preserving blending**.

This prevents **lowering contrast, ghosting and wrong data to appear**.

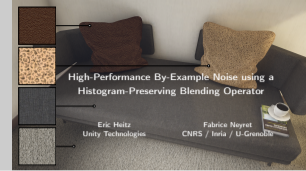
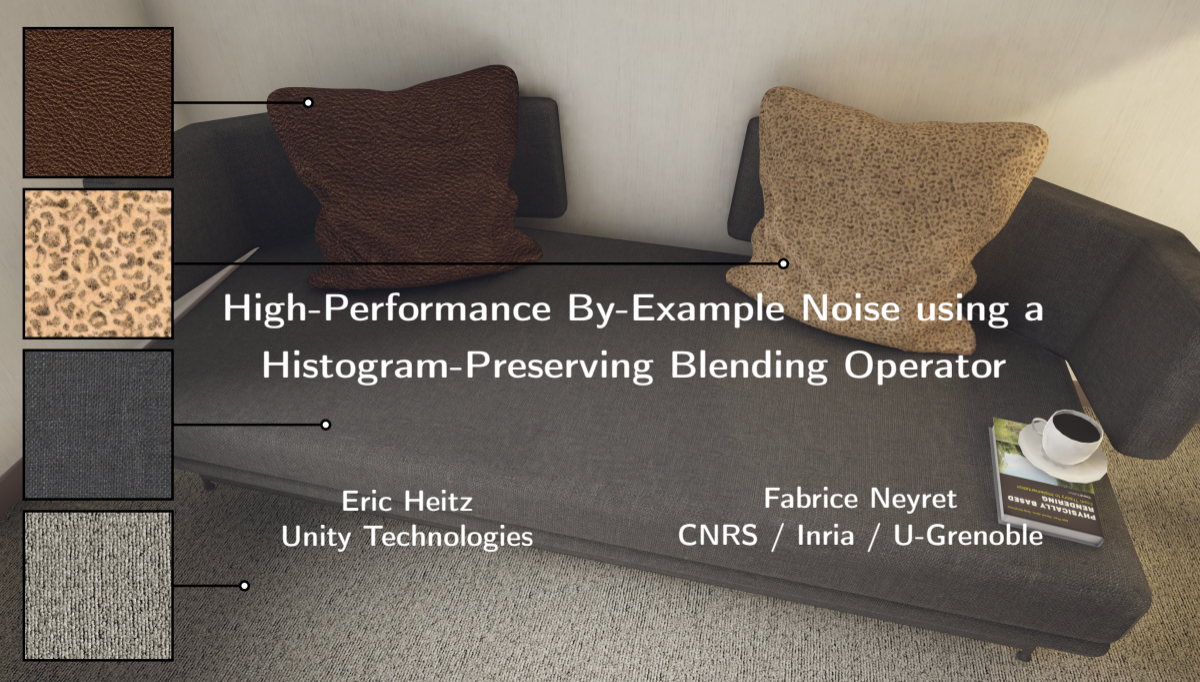
Common wisdom
Whenever you blend **weighted** data, use **premultiplied blending**.
This prevents **leaking and mixing with undefined values**.

Main takeaway of this paper
Whenever you blend data chosen **randomly**, use **histogram-preserving blending**.
This prevents **lowering contrast, ghosting and wrong data to appear**.

The main message of our paper has the same structure. Blending data that you chose randomly requires an histogram-preserving blending operator if you want to preserve their random properties. This is a problem you might encounter in many applications such as texturing, stochastic alpha testing, etc. We have proposed a simple solution for this problem (Gaussian is easy, make everything Gaussian) that scales to an arbitrary number of data distributed in arbitrary histograms.

We wrote a paper that pretended to be about procedural texturing but this concept is what it was truly about.

End of the story!



Eric Heitz and Fabrice Neyret thank Thomas Deliot for implementing the algorithm in the Unity engine, Kenneth Vanhoey for sharing LRPN results and Stephen Hill for his numerous corrections.