Anisotropic Noise

Alexander Goldberg Matthias Zwicker Frédo Durand
University of California, San Diego MIT CSAIL

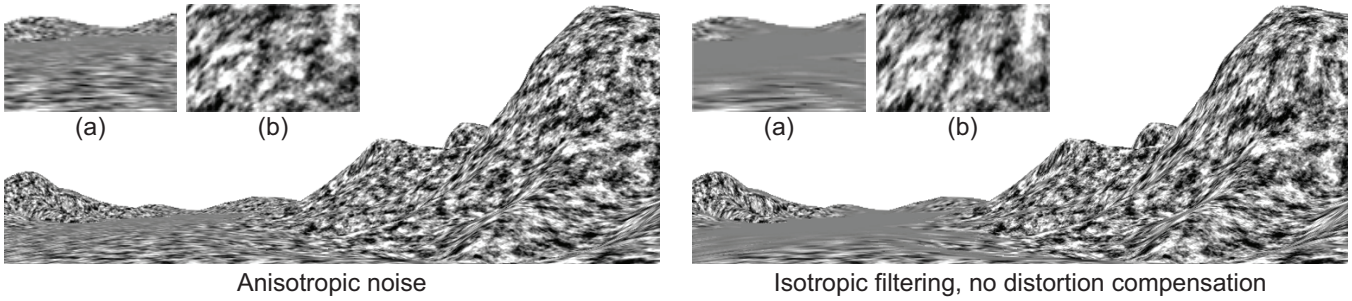


Figure 1: *We present a technique for fast, high quality rendering of noise textures in interactive applications. We perform anisotropic filtering, which leads to higher image quality compared to isotropic filtering, as shown by the difference between close-ups (a). Our method uses 2D noise tiles and we present a technique to compensate for parametric distortions to achieve similar effects as with solid texturing. Our technique avoids texture distortions as shown by the difference between close-ups (b).*

Abstract

Programmable graphics hardware makes it possible to generate procedural noise textures on the fly for interactive rendering. However, filtering and antialiasing procedural noise involves a tradeoff between aliasing artifacts and loss of detail. In this paper we present a technique, targeted at interactive applications, that provides high-quality anisotropic filtering for noise textures. We generate noise tiles directly in the frequency domain by partitioning the frequency domain into oriented subbands. We then compute weighted sums of the subband textures to accurately approximate noise with a desired spectrum. This allows us to achieve high-quality anisotropic filtering. Our approach is based solely on 2D textures, avoiding the memory overhead of techniques based on 3D noise tiles. We devise a technique to compensate for texture distortions to generate uniform noise on arbitrary meshes. We develop a GPU-based implementation of our technique that achieves similar rendering performance as state-of-the-art algorithms for procedural noise. In addition, it provides anisotropic filtering and achieves superior image quality.

1 Introduction

Noise functions are widely used in computer graphics to efficiently generate complex textures resembling natural phenomena. Essentially, these textures consist of a sum of band-limited noise images. By applying carefully designed functions to the noise bands, one can produce a variety of textures that resemble wood, marble, or clouds, etc. [Peachey 2003]. Noise functions are also often added to other textures to provide detail and a more natural look.

Perlin [1985] originally proposed to generate the noise bands procedurally. His approach amounts to evaluating a low-pass filter on the fly when the texture is sampled. With programmable graphics hardware it is possible to use procedural noise for interactive rendering [Miné and Neyret 1999; Hart 2001; Green 2005; Olano 2005]. This is desirable because only minimal texture memory must be allocated for noise evaluation. However, it is difficult to perform high-quality antialiasing of procedural noise. Antialiasing is typically achieved by truncating the noise frequencies to the Nyquist limit of the display. In theory, this can be done simply by omitting the summation of high frequency noise bands. Unfortunately, because Perlin noise is not narrowly band-limited, there is always a trade-off between aliasing and blurriness.

Lewis [1989] has addressed this problem with two alternative algorithms that allow the user to control the noise spectrum more precisely. Cook and DeRose [2005] point out, however, that Lewis' approach is not sufficient to solve the loss-of-detail vs. aliasing problem. Instead, they propose to construct noise bands using wavelet analysis. Wavelet noise bands are more narrow-band than Perlin noise and allow for more accurate antialiasing. An alternative approximation of isotropic antialiasing is to pre-filter the color lookup table as proposed by Hart et al. [1999]. It is well known, however, that high-quality texture antialiasing requires *anisotropic filtering*, which is not supported by these techniques.

In this paper we propose a technique, targeted at interactive applications, to efficiently generate and render noise textures with high-quality anisotropic antialiasing. As shown in Figure 1, our technique provides higher-quality antialiasing than isotropic filters such as wavelet noise. It also avoids the use of 3D textures.

We generate noise textures by tiling the frequency domain into oriented subbands. Typically, we use between four and eight bands at different orientations. Our approach is similar to wavelet noise in that we precompute texture tiles instead of sampling noise procedurally. However, the frequency content of our tiles is strongly oriented and anisotropic. This allows us to perform anisotropic filtering during rendering, leading to higher image quality. Our approach is also based solely on 2D textures, which reduces the storage requirements compared to 3D textures by an order of magnitude. If

we were to naïvely generate noise in 2D texture space, however, distortions of the surface parameterization would lead to nonuniform frequencies on the surface. We address these problems by analyzing and compensating for the parametric distortions (Figure 1, compare uniform noise features on the left to stretched texture on the right). We locally compute weighted combinations of the oriented subbands that approximate a uniform frequency spectrum on the surface. We present a GPU implementation of our technique that includes high-quality anisotropic antialiasing. In addition, it achieves similar rendering performance as state-of-the-art procedural algorithms.

In summary, we make the following contributions:

- We propose a novel construction of noise textures by tiling the frequency domain into oriented subbands.
- We introduce a method to generate uniform 2D noise on parameterized surfaces. Our approach compensates for distortions of the parameterization.
- We develop an algorithm for anisotropic filtering of noise textures based on oriented noise bands.
- We demonstrate a GPU implementation that provides anisotropic filtering and higher image quality than procedural techniques. Our implementation produces good results with a single 256×256 RGBA texture tile, requiring only 256KB of memory.

Our technique has three major components: In Section 2 we describe how we construct oriented 2D noise tiles using frequency domain filtering. The main feature of this approach is that we can approximate arbitrary noise spectra by simply computing weighted sums of the texture tiles. In Section 3 we propose a method to generate uniform noise on arbitrary meshes with two-dimensional texture coordinates. This ensures that our noise textures look like solid textures, without exhibiting distortion artifacts due to the underlying 2D parameterization. In Section 4 we present our rendering algorithm with anisotropic antialiasing. We show that this improves image quality over state-of-the-art isotropic noise filtering. Finally, we provide an implementation of our technique for programmable graphics hardware in Section 5.

2 Frequency Domain Noise Construction

The main idea of our approach is to construct noise subbands using frequency domain filtering. We generate noise bands that are not only narrowly band-limited in scale, but they also have a preferred orientation. We next discuss a frequency domain decomposition that partitions the frequency domain into oriented subbands. Then we show how we construct noise textures for each subband.

Frequency Domain Decomposition. We partition the frequency domain into subbands using steerable filters [Simoncelli and Freeman 1995; Portilla and Simoncelli 2000]. These filters provide a number of properties that are crucial for our application. First, each filter defines a subband that is tightly *localized in scale and orientation*. Second, the filters implement an *invertible transform*. This implies that we can exactly recover a signal from its decomposition into subbands. Finally, the filters are *steerable* in orientation. This essentially means that we can linearly interpolate between the filters to generate a filter with the exact same profile, but at an intermediate orientation. This property is useful because it avoids interpolation artifacts when linearly blending the subbands as in Sections 3 and 4. Ashikmin and Shirley [2002] also exploited steerable filters to smoothly interpolate illumination textures.

We can express the subband filters in the frequency domain explicitly in a polar-separable form $D_{i,j}(r,\theta) = G_i(\theta)H(2^j r)$, where i and j are indices for orientation and scale respectively. The angular and radial parts are defined as

$$G_i(\theta) = \begin{cases} \alpha_K^2 \left[cos\left(\theta - \frac{\pi i}{K}\right)\right]^{2(K-1)}, & |\theta - \frac{\pi i}{K}| < \frac{\pi}{2} \\ 0, & \text{otherwise} \end{cases}$$

$$H(r) = \begin{cases} \cos^2\left(\frac{\pi}{2}\log_2\left(\frac{4r}{\pi}\right)\right), & \frac{\pi}{4} < r < \frac{\pi}{2} \\ 4, & r \le \frac{\pi}{4} \\ 0, & r \ge \frac{\pi}{4} \end{cases} \quad (1)$$

where K is the number of orientations, $i \in [0, K-1]$, and $\alpha_K = 2^{i-1}(K-1)!/\sqrt{K[2(K-1)]!}$. Note that we use the square of the filters proposed by Portilla and Simoncelli [2000]. This is because they apply the filters twice in their scheme, first in the analysis and then in the synthesis step. Therefore, our filters correspond to a combined analysis-synthesis sequence in their framework.

Spectral Noise Synthesis. We now describe how to precompute noise textures in the *spatial* domain using the frequency domain filters discussed above. Our approach is illustrated in Figure 2:

1. Create a frequency spectrum F consisting of complex valued, uniform white noise in the Fourier domain.
2. Decompose the uniform noise spectrum into oriented subbands by multiplying with the filters $D_{i,j}$ described above. We obtain noise subbands $N_{i,j} = FD_{i,j}$, which are narrowly band-limited in scale and orientation.
3. Perform an inverse Fourier transform of each individual subband $N_{i,j}$ to get the band-limited spatial domain noise images $n_{i,j}$.

Because the frequency domain decomposition is invertible, the frequency spectrum of the sum of all spatial domain noise images corresponds to the uniform white noise that we used as an input in the first step. By summing only select subband images we can quickly generate noise images with custom-tailored frequency spectra. Note that we sum the subband images directly in the spatial domain. Therefore, this approach is well suited for interactive applications and GPU implementation.

For example as shown in Figure 3, summing all orientations of the same scale leads to noise that is isotropically band-limited, similar to wavelet noise [Cook and DeRose 2005]. However, we can also generate noise images with anisotropic spectra. This is the core idea of our algorithms that we present Sections 3 and 4.

3 Steerable Noise on Surfaces

Mapping 2D textures onto 3D surfaces will almost always introduce parametric distortions of the texture. These distortions cause variations in the frequency spectra of noise textures, which are visually disturbing. In this section we present a method to combat parametric distortions by generating noise bands that exhibit uniform spectra on the surface. In addition, our method can be used to *steer* noise to obtain user specified and spatially varying spectra. We compute weighted sums of the precomputed noise subbands to approximate the desired spectra. Our approach leads to uniform noise even on meshes with highly distorted parameterizations. This

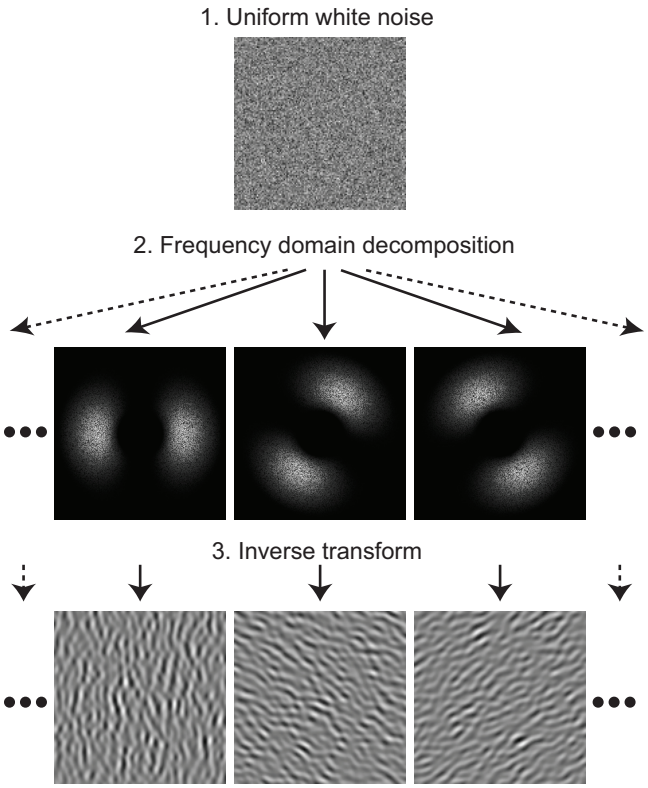


Figure 2: *Illustration of spectral noise generation. The frequency domain decomposition has three orientations. We show three oriented subbands at the same scale and their corresponding spatial domain images, which we store as textures.*

allows us to achieve similar effects as solid texturing, without revealing the underlying 2D parameterization.

Alternatively, we could extend the frequency domain decomposition approach from Section 2 to 3D. This would allow us to evaluate 3D noise textures that do not suffer from parametric distortions. However, this approach would increase the number of oriented noise subbands by an order of magnitude. We would have to store and access dozens of 3D textures, which would not be practical for hardware-accelerated interactive rendering.

3.1 Distortion Compensation

Our goal is to generate uniform 2D noise textures on parameterized triangle meshes. We compensate for parametric distortions on a per-triangle basis as illustrated in Figure 4. Let us define a local coordinate system with coordinates s and t on the plane of each triangle (Figure 4a). We strive to generate uniform noise with an isotropic, band-limited spectrum in (s,t) coordinates as shown in Figure 4c. This target spectrum is identical for each triangle, such that the noise texture will be uniform over the whole triangle mesh. For some applications it may also be interesting to use spatially varying or anisotropic target spectra. In this case, the user defines the target spectra on a per triangle basis. We show an example of this in Section 6.

We represent parametric distortions by linear mappings $(u;v) = \mathbf{T}(s;t)$ from local (s,t) to (u,v) texture coordinates. Here $\mathbf{T}$ is a 2×2 matrix and $(s;t)$ denotes a column vector. Note that we ignore translations. We compute $\mathbf{T}$ from the correspondence of (s,t) and (u,v) coordinates at each triangle vertex. Our goal now is to compute a spectrum in texture space that will match the target spectrum when it is mapped to object space. When a noise texture is mapped from texture to object space using $(s;t) = \mathbf{T}^{-1}(u;v)$, its spectrum is distorted by $\mathbf{T}^T$ and the amplitudes of the spectrum are scaled by a factor $\sqrt{|\mathbf{T}^{-1}|}$ (Figure 4d). The scaling factor follows from Parseval's theorem. Note that, under affine deformations of the texture, the integral of the squared noise values over each triangle in the spatial domain remains constant. We generate noise that approximates the distorted spectrum by computing a weighted sum of the subband textures that we precomputed in Section 2. We present a least-squares and a heuristic approach to determine the subband weights.

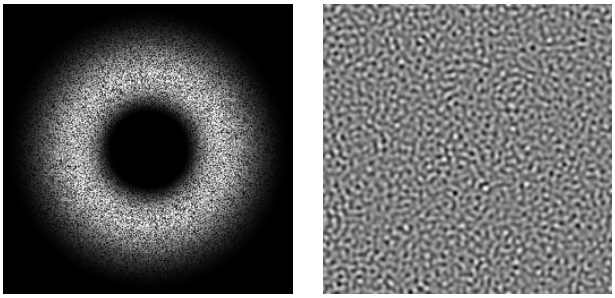


Figure 3: *We obtain noise with custom-tailored frequency spectra by summing subband images. Here we add three orientations of the same scale, shown in Figure 2, to get perfectly isotropic noise.*

Least Squares Approximation. Let us denote the isotropic target spectrum in object space by E. Note that E is the same for all triangles. We also denote the frequencies corresponding to texture coordinates u, v by ϕ, ψ. We write the least squares optimization to approximate the distorted spectrum in texture space as

$$\min_{\gamma_{i,j}} \iint \left(\sqrt{|T^{-1}|} E(\mathbf{T}^T(\phi;\psi)) - \sum_{i,j} \gamma_{i,j} D_{i,j}(\phi,\psi) \right)^2 d\phi d\psi,$$

where $D_{i,j}$ are the filters forming the frequency domain decomposition from Section 2. We solve this system for each triangle in a preprocess. We evaluate the integrals numerically. Note that each filter $D_{i,j}$ corresponds to a precomputed spatial domain texture $n_{i,j}$. Therefore, we obtain the noise texture with the desired spectrum by summing the spatial domain tiles, i.e., the texture is $\sum_{i,j} \gamma_{i,j} n_{i,j}$.

Heuristic Approximation. The disadvantage of the least squares method is that we cannot use it for dynamically deforming meshes. In this case we need to update the weights on the fly while triangles deform. As an alternative, we use a heuristic approach that is very fast to compute. We associate a center $\phi_{i,j}, \psi_{i,j}$ with each subband and simply define the weights as $\gamma_{i,j} = \sqrt{|T^{-1}|} E(\mathbf{T}^T(\phi_{i,j};\psi_{i,j}))$. We compare the least squares and the heuristic approximation in Section 3.2.

In Figure 5, we illustrate the least-squares and heuristic approximation of the target spectrum shown in Figure 4d. We used noise subbands at three different scales and six orientations.

Multiple Octaves. One target band as shown in Figure 4 is usually not sufficient to generate interesting noise textures. Instead, one uses a weighted sum of several *octaves* of noise. We need to

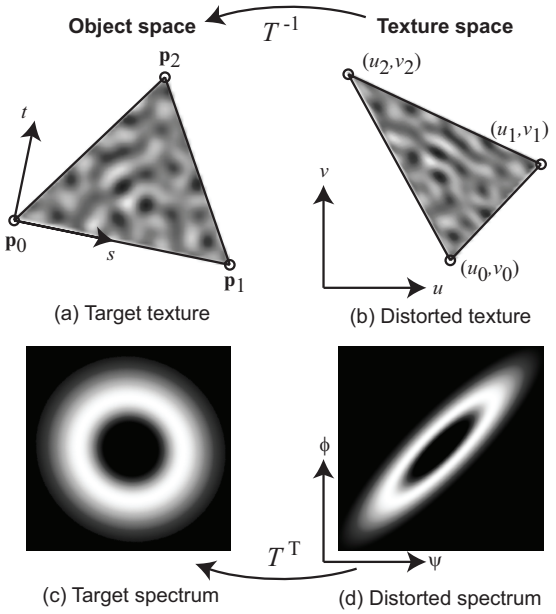


Figure 4: *Distortion compensation: Figure (a) shows the desired uniform noise texture in object space; the corresponding target spectrum is in Figure(c). The surface parameterization leads to distortion in texture space, as in Figure (b). The corresponding noise spectrum in texture space is shown in (d). This is the spectrum we need to generate to obtain uniform noise in object space.*

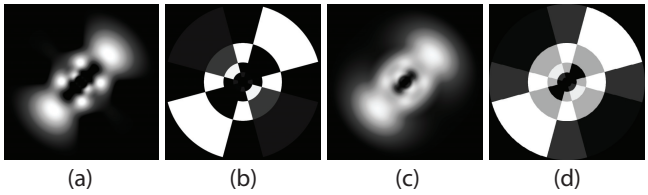


Figure 5: *Approximations of the target spectrum in Figure 4d for distortion compensation. (a) Spectrum of the least squares approximation and its corresponding filter weights (b); (c) spectrum of the heuristic approximation and its corresponding filter weights (d).*

compute the distortion compensation weights only for a single octave and can easily derive the weights for all other octaves from it.

Each octave has a band-limited spectrum as in Figure 4c, with the frequencies scaled by a factor of two from one octave to the next. Let us assume we have computed the distortion compensation weights for the lowest octave. We denote them by $\gamma_{0,i,j}$, where 0 is the index of the octave, and i and j are the indices for scale and orientation. Because the scales of our subband decomposition increase by a factor of two as well, we obtain the subband weights for the next higher octave as $\gamma_{1,i,j} = \gamma_{0,i-1,j}$. In general we have $\gamma_{n,i,j} = \gamma_{0,i-n,j}$. If $i - n < 0$ we set the weight to zero. Note that regardless of the number of scales required to represent a single octave of noise on a mesh, each additional desired octave increases the total number of scales by one.

3.2 Results and Comparisons

Figure 6 illustrates our distortion compensation approach using a fractal noise texture consisting of two octaves of noise. We map a

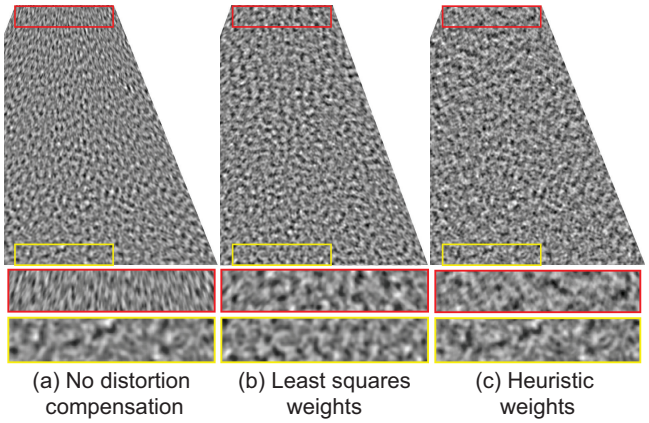


Figure 6: *Illustration of the distortion compensation technique using fractal noise with two octaves. (a) Without distortion compensation, undesirable compression of the noise texture appears. (b) Distortion compensation using least-squares fitting, and (c) using the heuristics. The results are visually comparable.*

square region of texture space, i.e., $(u, v) \in [0, 1] \times [0, 1]$ onto a trapezoid. This leads to undesirable compression of the noise texture as shown in Figure 6a. We compare distortion compensation using least-squares fitting and our heuristics in Figures 6b and 6c. We used a frequency domain decomposition with five scales and four orientations. The heuristics produces results that are visually comparable to least-squares fitting. Therefore, unless otherwise noted, all images in this paper were generated using the heuristic method.

4 Anisotropic Filtering

In this section we describe an algorithm for high-quality, anisotropic antialiasing of noise textures based on oriented subbands. Our algorithm builds on a frequency domain formulation of texture filtering, which we describe in Section 4.1. We present an efficient, constant time approximation of this approach using precomputed subband images in Section 4.2. We discuss results of our approximation in Section 4.3.

4.1 Frequency Domain Texture Filtering

We derive our approach from an analytic formulation of anisotropic texture filtering introduced by Heckbert [1989], which we summarize first. Heckbert showed that we can formulate anisotropic filtering as a convolution in texture space. We write this as

$$\begin{aligned} P(x_0, y_0) &= \int f(u,v) g(M(x_0, y_0) - (u, v)) du dv \\ &= (f \otimes g)(M(x_0, y_0)). \end{aligned} \quad (2)$$

Here, $P(x_0, y_0)$ is the anisotropically filtered texture value at a pixel x_0, y_0. We denote the texture by $f(u, v)$, where u, v are texture coordinates. In addition, $(u, v) = M(x, y)$ is the mapping from image to texture coordinates. Heckbert derives the low pass filter $g(u, v)$ in texture space from the low pass filter $h(x, y)$ in image space as

$$g(u, v) = \left|\mathbf{J}^{-1}\right| h(\mathbf{J}^{-1}(u; v)).$$

Here, $\mathbf{J}$ is the 2×2 Jacobian matrix of M defined by

$$\mathbf{J} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{\partial u}{\partial y} \\ \frac{\partial v}{\partial x} & \frac{\partial v}{\partial y} \end{bmatrix}_{x_0, y_0},$$

and the notation $\mathbf{J}(\cdot;\cdot)$ is a multiplication with a column vector.

We now express the convolution in Equation 2 as a multiplication in the frequency domain. Let us denote the Fourier transform pair of the low-pass filter in image space by $h \leftrightarrow H$. This implies that we also have the Fourier transform pair $g \leftrightarrow G$ for the low-pass filter in texture space, where the Frequency domain filter is $G = H(\mathbf{J}^T(\psi, \phi))$. Therefore, the frequency domain formulation of Equation 2 is

$$f \otimes g \leftrightarrow FG = F(\psi, \phi) H(\mathbf{J}^T(\psi; \phi)), \tag{3}$$

where F is the Frequency domain representation of the texture.

We efficiently evaluate the frequency domain low pass filter in texture space using Gaussians. The Gaussian low-pass filter in image space is

$$h(x, y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(x^2+y^2)},$$

where we typically use $\sigma = 1$. The corresponding frequency domain filter in texture space is

$$H(\mathbf{J}^T(\psi; \phi)) = e^{-\frac{\sigma^2}{2}(\psi;\phi)^T \mathbf{J}\mathbf{J}^T(\psi;\phi)}. \tag{4}$$

4.2 Constant Time Approximation

We now derive an efficient, constant time approximation of Equation 3 based on precomputed subband textures in the *spatial domain*. This leads to a technique similar to the constant time filtering methods by Fournier and Fiume [1988] and Gotsman [1994]. They also rely on a set of basis filters to approximate arbitrary low pass filters. Our approach, however, uses different basis filters that are tailored to generate noise textures.

We approximate the frequency domain low-pass filter in Equation 3 as a weighted sum of the filters $D_{i,j}$ from Section 2,

$$H(\mathbf{J}^T(\psi; \phi)) \approx \sum_{i,j} \lambda_{i,j} D_{i,j}(\psi, \phi), \tag{5}$$

where $\lambda_{i,j}$ are suitable weights. We can determine these weights using a least squares approximation or a simple heuristics similar as in Section 3. For the heuristics, we choose the weights as $\lambda_{i,j} = H(\mathbf{J}^T(\psi_{i,j}; \phi_{i,j}))$. We compare the two approaches in Section 4.3.

Substituting Equation 5 into Equation 3 we obtain

$$F(\psi, \phi) H(\mathbf{J}^T(\psi; \phi)) \approx \sum_{i,j} \lambda_{i,j} (F D_{i,j})(\psi, \phi). \tag{6}$$

Since we precomputed the Fourier transform pairs $n_{i,j} \leftrightarrow N_{i,j} = F D_{i,j}$, we immediately get the spatial domain counterpart of Equation 6,

$$(f \otimes g)(M(x_0, y_0)) \approx \sum_{i,j} \lambda_{i,j} n_{i,j}(M(x_0, y_0)). \tag{7}$$

This is the basis of our texture filtering algorithm, which is illustrated in Figure 7. At each pixel (x_0, y_0) we compute the frequency domain representation of the Gaussian low-pass filter in texture space as in Equation 4, and the approximation weights $\lambda_{i,j}$. We look up each subband image $n_{i,j}$ at the corresponding texture location $M(x_0, y_0)$ using bilinear interpolation. The final texture value is the weighted sum of the values from all subband images. The complexity of this approach is constant at each pixel, and it is proportional to the number of subbands that we use to represent the texture. Note that, in principle, this technique is applicable to arbitrary textures, not only noise textures.

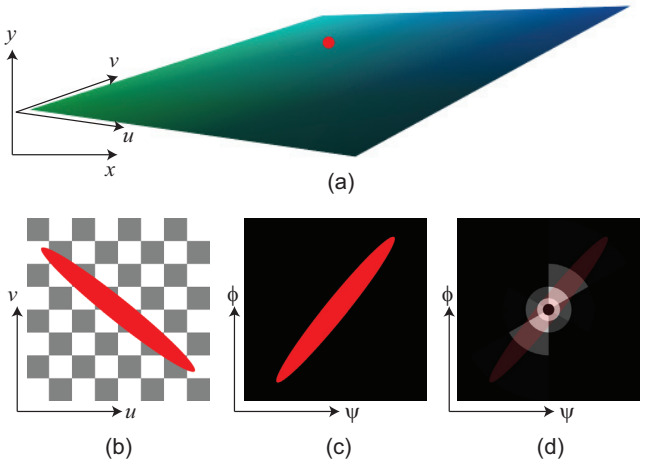


Figure 7: *Illustration of our anisotropic filtering algorithm. Image (a) visualizes a rendered plane, where the colors encode the magnitudes of the texture space partial derivatives with respect to screen space. Image (b) shows the low-pass filter kernel at the red pixel mapped to texture space, and (c) is a frequency space representation of this filter. Image (d) visualizes the subband weights $\lambda_{i,j}$ that we computed to approximate the frequency space representation of the filter.*

4.3 Results and Comparisons

We compare the results of least squares fitting and the heuristics in Figure 8. Note that the least squares approach is not suitable for interactive rendering because we would have to solve a small system of linear equations at each pixel. Our comparison shows, however, that the heuristic approach does not significantly reduce image quality.

We resampled a test image containing high frequencies under an affine mapping that includes shearing and non-uniform scaling. Figure 8a shows the reference solution using Gaussian filtering; the inset depicts the frequency spectrum of the filter. To apply our algorithm we precomputed a frequency domain decomposition of the test image with different numbers of oriented subbands. In Figure 8b we used a decomposition with eight orientations and least-squares fitting to compute the approximation weights. The result is practically indistinguishable from the reference solution. We used least squares fitting with four orientations in Figure 8c and with heuristic weights in 8d. Reducing the number of orientations increases aliasing artifacts. However, least squares fitting and heuristic weight computation are visually very similar. We provide Figure 8d as a reference showing bilinear resampling.

5 GPU Implementation

Rendering anisotropic noise is a three step process. We first synthesize and store noise tiles in an offline process as discussed in Section 2. We pack an oriented subband image into each chan-

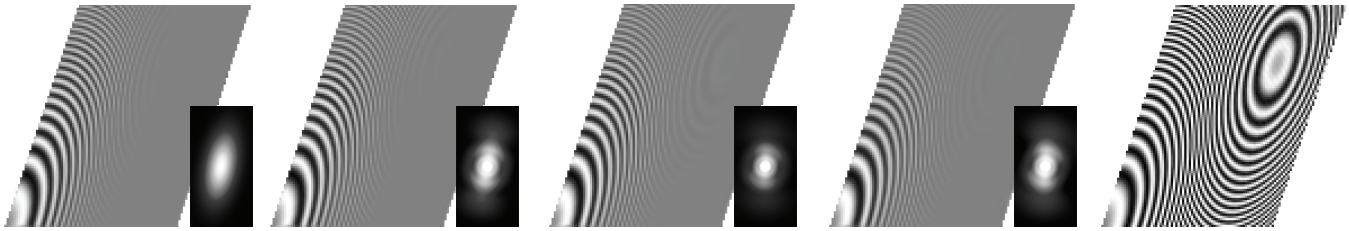


Figure 8: *We evaluate our least-squares fitting and heuristic weight computation in an image resampling application. We use Gaussian filtering as a reference for comparison. The insets depict the frequency spectra of the filters, and we provide the mean square error (MSE) to the reference. Least-squares fitting using eight orientations is very close to the reference. With four orientations more aliasing appears. While the heuristic weight computation is visually very close to the least-squares fit, the numerical error is significantly higher. We show a result of bilinear resampling for comparison.*

nel of a 32-bit RGBA image, yielding four orientations per texture. Note that we precompute noise subbands at a single scale only. We generate all other scales on the fly by simply scaling the precomputed textures. We have found that a single 256×256 four-orientation texture produces very good results, and eight orientations (using two textures) produces excellent results. We generate distortion compensation weights for each mesh on the CPU. We either use least-squares fitting in an offline process, or compute the weights with the heuristic approach at runtime. The noise tile textures and per-vertex distortion compensation weights are the input to the shader programs.

During rendering, we compute a nested sum of noise subbands at each pixel (x_0, y_0),

$$\sum_{k}^{Oct} f_k \sum_{i}^{Or} \sum_{j}^{Sc} \gamma_{f,i,j} \lambda_{i,j} n_{i,j}[M(x_0, y_0)].$$

The inner two sums compute a noise value of a single octave by iterating over all subbands. Here $n_{i,j}$ are the scaled subband images, and $M(x_0, y_0)$ are the texture coordinates at pixel (x_0, y_0). We weight each subband i, j with its distortion compensation weights $\gamma_{f,i,j}$ for octave k, and the weights $\lambda_{i,j}$ that approximate the low pass filter. In addition, f_k is a user specified function applied to noise at octave k. For example, fractal noise has $f_k(t) = p^k t$, where t is the noise value and p is the persistence, which is a scalar parameter that controls noise amplitude as a function of the frequency band.

In our *unoptimized* implementation, we evaluate the triple sum above in the pixel shader. If the function f_k is linear, as for example for fractal noise, we can swap the order of operations and sum over octaves first. This allows us to precompute a single distortion compensation weight for each subband and eliminate the loop over octaves from the pixel shader. We refer to this version as the *optimized* implementation.

For most real-world meshes the anisotropic filtering weights vary gradually across each triangle. Therefore, instead of computing them on a per-pixel basis, we compute them on a per vertex basis and interpolate them during rasterization. This requires calculating the Jacobian of the screen-space to texture-space mapping on a per-vertex basis. We accomplish this by precomputing and storing each triangle's texture- to object-space transformation. We concatenate it at runtime with the rest of the viewing transformation to obtain the full texture- to screen-space mapping. We use its Jacobian to compute the filter weights as described in Section 4.2. This approach leads to visually identical results as the per-pixel computation, except for meshes that are extremely coarsely triangulated.

Because our technique is based on 2D textures one could also rely on hardware anisotropic antialiasing instead of using our filtering weights. However, hardware antialiasing does not address parameterization distortion. One could resort to 3D texture tiles to avoid the distortion issues, but graphics hardware does not support proper anisotropic filtering for 3D textures. In addition, in our framework there is no measurable performance penalty for anisotropic filtering, since we compute the antialiasing weights in the *vertex shader* and premultiply them with the distortion compensation weights.

The heuristics to determine subband weights described in Section 4 uses only the center of the spectrum of each subband. This can lead to minor aliasing artifacts at grazing angles. We avoid these artifacts by weighting three radial points from each subband instead of one in the center. Since this computation occurs in a vertex shader, the additional performance overhead is negligible.

6 Results

We demonstrate the image quality achieved with our GPU implementation in Figure 9. The textures contain two octaves of noise. Figure 9a shows our result with four oriented subbands, and 9b uses eight orientations. The additional orientations lead to a noticeable increase in sharpness without introducing any aliasing. Figure 9c shows procedural noise for comparison. The noise pattern is not exactly the same as with our techniques, but we adjusted it to match as well as possible. We do not use any filtering, which leads to strong aliasing artifacts. They become even more evident in animations.

Table 10 reports the rendering performance of our GPU implementation. We rendered noise at 1680×1050 full screen resolution, which means that each frame consisted of 1.764 million noise shaded pixels. We used a single-core 3.0GHz Pentium 4 with 2 GBytes of memory and a GeForce 6800 Ultra GPU. We provide performance numbers for both the optimized and unoptimized implementation (Section 5). Note that the number of scales in the optimized version does not directly correspond to the number of octaves. We need more scales to generate a desired number of octaves depending on the severity of distortions. For the unoptimized implementation, we report performance depending on the number of scales per octave. The required number again depends on the severity of parameterization distortion. For the examples shown in this paper we never needed more than four scales per octave. We compared our technique to our own implementation of Olano's ap-

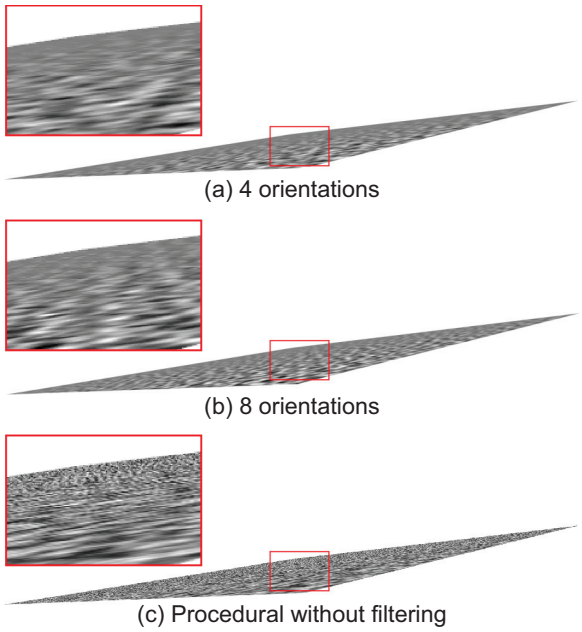


Figure 9: *We compare image quality with two octaves of fractal noise using our GPU implementation: (a) four oriented subbands and heuristic antialiasing weights, (b) eight oriented subbands, and (c) procedural noise without filtering.*

Optimized			Unoptimized		
S		*O*	2 *S/O*	3 *S/O*	4 *S/O*
2	465	1	465	377	307
3	377	2	220	167	125
4	307	3	165	115	80
5	263	4	99	76	58

Figure 10: *Rendering performance of anisotropic noise in frames per second. We distinguish between the optimized and the unoptimized implementation of anisotropic noise.* S *indicates the number of scales. For the unoptimized implementation,* O *indicates the number of octaves, and* S/O *is the number of scales per octave.*

proach [2005]. We found that the rendering performance of our optimized approach is slightly faster. The performance of our unoptimized technique is very similar to theirs, with an advantage for our approach on lower-end GPUs. Their approach, however, does not include antialiasing.

We can also perturb the noise textures using additional functions as described in Section 5. Figure 11 shows combinations of bump mapping, turbulence, and color look-up tables. Our distortion compensated noise can be the basis for many effects that are otherwise achieved with procedural solid noise. The main difference is that our approach requires a surface parameterization. For example, we can animate the noise by blending between several base textures. We use histogram matching to avoid blurriness in interpolated textures. We also shift texture coordinates over time to achieve the effect of moving flames. However, our antialiasing technique does not lead to correct results if the perturbation functions are nonlinear. This is a limitation that our technique shares with other approaches such as wavelet noise. Correct anisotropic filtering of procedural textures is a challenging open problem.

The frequency domain noise generation described in Section 2 leads to textures that are tileable on a torus topology as well as on the

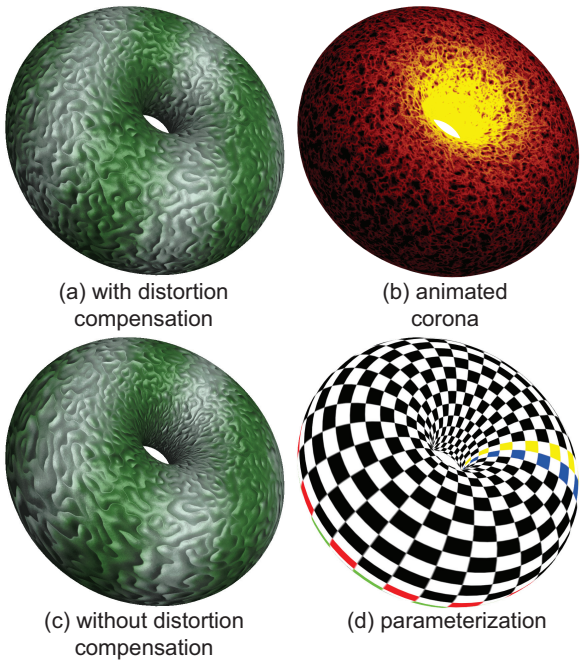


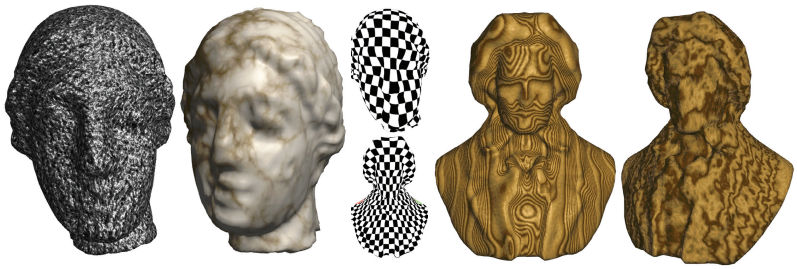
Figure 11: *Distortion compensated noise as a basis for advanced effects that perturb the noise bands: (a) bump-mapped turbulence with a color look-up, (b) turbulence as a displacement to look-up a color texture. We can also animate textures such as this one. In (c) we show a noise texture without un-distortion, which reveals the parametric distortions shown using a checkerboard texture in (d). Colored checkers indicate boundaries in the 2D texture.*

plane. We can also make the textures tileable on the sphere using standard techniques based on finding inconspicuous boundaries and Poisson image blending [Efros and Freeman 2001; Pérez et al. 2003]. In Figure 12 we show examples with spherical topology. The bust is parameterized using the scheme proposed by Gu et al. [2002]. The terrain texture can be modified interactively at high framerates.

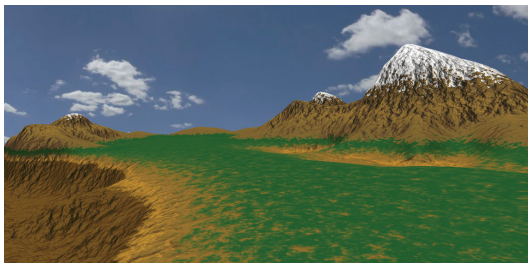
We can also steer noise explicitly by encoding a spatially varying, anisotropic target spectrum in an additional texture map, instead of using a uniform, isotropic target. The texture defines per triangle target spectra, which allows us to obtain interesting effects as shown in Figure 13. The orientation field can be manipulated interactively to modify the texture.

7 Conclusions

We presented a novel approach to render high-quality noise textures that is targeted at interactive applications. Similar to wavelet noise by Cook and DeRose, we generate noise tiles that are narrowly band-limited and, therefore, suitable for antialiasing. However, we construct the noise tiles directly in the frequency domain, and we use a partition of the frequency domain into oriented subbands. We use the oriented noise subbands as building blocks to approximate noise with desired spectra. Our approach is based solely on 2D textures, which are more suitable for interactive rendering and reduce memory requirements. We developed a technique to compensate for parametric distortions and generate noise with uniform frequencies on surfaces. This allows us to obtain effects similar to 3D solid texturing, without any visible distortions due to 2D texturing. We perform high quality anisotropic filtering using an efficient

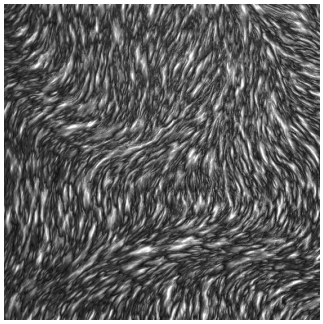


Noise textures on meshes with spherical topology



Procedural terrain

Figure 12: *We demonstrate meshes with spherical topology on the left. We avoid seams by preprocessing the textures to make them tileable on the spherical topology. We use bump mapping and color look-up tables. The terrain on the right also uses bump mapping, where the bump scale is modulated using a look-up table.*



Steerable Noise

Orientation Field

Figure 13: *We use the distortion compensation technique to generate noise with a spatially varying, anisotropic spectrum. The orientation field on the right encodes the direction of anisotropy.*

GPU implementation. Our approach provides higher image quality through anisotropic filtering than procedural techniques.

In the future, we will further investigate correct anisotropic filtering of non-linearly perturbed noise. This is a very challenging problem that has withstood a principled solution so far. We will also extend our system using Wang tiling to enable texturing of arbitrarily large geometry without periodic patterns. This could be particularly useful for terrain rendering. We believe that our approach could find widespread application for interactive rendering of noise textures because of its performance and its quality advantages.

Acknowledgments. We thank Paul Green for his preliminary experiments, the MIT pre-reviewers for their insightful feedback, Arash Keshmirian for help with the video, and Hugues Hoppe for the Venus and Beethoven meshes. This work was supported by an NSF CAREER award 0447561. Frédo Durand acknowledges a Microsoft Research New Faculty Fellowship and a Sloan Fellowship.

References

ASHIKHMIN, M., AND SHIRLEY, P. 2002. Steerable illumination textures. *ACM Trans. Graph. 21*, 1, 1–19.

COOK, R. L., AND DEROSE, T. 2005. Wavelet noise. *ACM Trans. Graph. 24*, 3, 803–811.

EFROS, A. A., AND FREEMAN, W. T. 2001. Image quilting for texture synthesis and transfer. In *Proceedings of SIGGRAPH '01*, 341–346.

FOURNIER, A., AND FIUME, E. 1988. Constant-time filtering with space-variant kernels. *SIGGRAPH Comput. Graph. 22*, 4, 229–238.

GOTSMAN, C. 1994. Constant-time filtering by singular value decomposition. *Comp. Graph. Forum 13*, 2, 153–163.

GREEN, S. 2005. *GPU Gems 2*. Addison-Wesley Professional, ch. Implementing Improved Perlin Noise.

GU, X., GORTLER, S. J., AND HOPPE, H. 2002. Geometry images. *ACM Trans. Graph. 21*, 3, 355–361.

HART, J. C., CARR, N., KAMEYA, M., TIBBITTS, S. A., AND COLEMAN, T. J. 1999. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 45–53.

HART, J. C. 2001. Perlin noise pixel shaders. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, 87–94.

HECKBERT, P. 1989. *Fundamentals of texture mapping and image warping*. Master's thesis, University of California, Berkeley.

LEWIS, J. P. 1989. Algorithms for solid noise synthesis. In *Proceedings of SIGGRAPH '89*, 263–270.

MINÉ, A., AND NEYRET, F. 1999. Perlin textures in real time using OpenGL. Tech. rep., RR-3713, INRIA.

OLANO, M. 2005. Modified noise for evaluation on graphics hardware. In *Eurographics Symp. on Graphics Hardware*, 105 –110.

PEACHEY, D. 2003. *Texturing and Modeling: A Procedural Approach*, 3 ed. Morgan Kaufman Publishers Inc., ch. Building procedural textures.

PÉREZ, P., GANGNET, M., AND BLAKE, A. 2003. Poisson image editing. *ACM Trans. Graph. 22*, 3, 313–318.

PERLIN, K. 1985. An image synthesizer. In *Proceedings of SIGGRAPH '85*, 287–296.

PORTILLA, J., AND SIMONCELLI, E. P. 2000. A parametric texture model based on joint statistics of complex wavelet coefficients. *Int'l Journal of Computer Vision 40*, 1 (October), 49–71.

SIMONCELLI, E. P., AND FREEMAN, W. T. 1995. The steerable pyramid: A flexible architecture for multi-scale derivative computation. In *Second Int'l Conf. on Image Proc.*, 444–447.