Wavelet Noise

Robert L. Cook Tony DeRose
Pixar Animation Studios

Abstract

Noise functions are an essential building block for writing procedural shaders in 3D computer graphics. The original noise function introduced by Ken Perlin is still the most popular because it is simple and fast, and many spectacular images have been made with it. Nevertheless, it is prone to problems with aliasing and detail loss. In this paper we analyze these problems and show that they are particularly severe when 3D noise is used to texture a 2D surface. We use the theory of wavelets to create a new class of simple and fast noise functions that avoid these problems.

CR Categories: I.3.3 [Picture/Image generation]: Antialiasing—[I.3.7]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

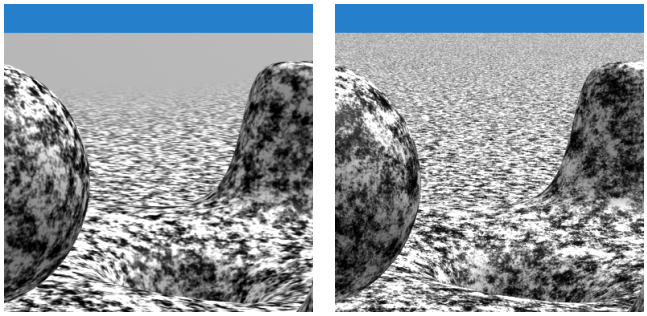
Keywords: Multiresolution analysis, noise, procedural textures, rendering, shading, texture synthesis, texturing, wavelets.

1 Introduction

Ever since the introduction of Perlin noise in [Perlin 1985], noise functions have been important for creating textures in 3D computer graphics. Perlin constructs patterns from *bands* of noise, each of which is limited to a range of frequencies. The bands are used as building blocks to construct complex noise patterns, where the frequency spectrum is controlled by weighting the contributions of different bands. For example, fractal noise can be produced by making the amplitude of each band inversely proportional to its frequency. Because this approach uses a single weight per band, it does not support detailed spectral shaping. By contrast, [Lewis 1989] introduced a Wiener interpolation method that is capable of constructing a texture with an arbitrary spectrum. This provides more control over the characteristics of the noise, but is considerably more expensive.

Despite a considerable amount of research into the definition and construction of noise (see [Peachey 2003] for an excellent survey), Perlin's original version of noise has continued to be the workhorse of the industry. There are good reasons for this: it is fast, it is simple, and the bands provide sufficient spectral control for most applications.

Each Perlin band is intended to be band-limited so that it contains only frequencies in a power-of-2 range. But each band actually contains a much wider range of frequencies; this is discussed in detail in [Lewis 1989] and is evident in the Fourier transforms in Figure 8. These weak band limits lead to some serious problems. A Perlin band near the Nyquist limit contains both frequencies that are low enough to be representable (i.e., they contain detail that should be in the image) and frequencies that are high enough to be unrepresentable (i.e, they can cause aliasing). Excluding the band causes loss-of-detail artifacts, but including it causes aliasing artifacts. Balancing this tradeoff between loss of detail and aliasing has been a constant source of frustration for shader writers at Pixar and elsewhere. Because aliasing is usually more unacceptable than loss of detail in feature film production, bands are attenuated aggressively. An unfortunate consequence of this is that as you zoom into a scene the texture detail becomes visible later than the geometry detail, so the texture doesn't appear to be tied to its geometry. Instead, the texture appears to fade in unnaturally as if there were a haze that obscured only some aspects of the surface appearance and only when it was farther away. This is illustrated in Figure 1. A sparse convolution method was introduced in [Lewis 1989] to improve the band-limited character of noise, but it too does not completely solve the loss-of-detail vs. aliasing problem.



(a) (b)

Figure 1: *A comparison between images created using (a) Perlin noise and (b) wavelet noise. Image (a) represents best practices use of Perlin noise at Pixar to achieve the optimal tradeoff between detail and aliasing; notice how much detail is missing at high spatial frequencies in the far distance.*

There is, however, an even more fundamental problem — one that plagues every existing noise creation method. When rendering, it is common to texture 2D surfaces by sampling a 3D noise function, but the resulting 2D texture will in general *not* be band-limited, *even if the 3D function is perfectly band-limited.* This means that the loss-of-detail vs. aliasing tradeoff cannot be solved simply by constructing a band-limited 3D function. To our knowledge this problem has never even been identified, much less addressed.

In this paper, we introduce a new noise technique called *wavelet noise* that addresses all of these issues. Our approach is based on the observation that Perlin's construction using sums of scaled and attenuated versions of a band-limited function was an early example of what has come to be known as a *multiresolution* function. Subsequent to Perlin's work, wavelet analysis (cf. [Chui 1992; Stollnitz et al. 1996]), also known as multiresolution analysis, has emerged as a powerful way to analyze and construct such functions. It is therefore natural to use this approach to analyze and construct noise.

Wavelet noise is almost perfectly band-limited, providing good detail with minimal aliasing, as demonstrated in Figure 1. Moreover, 3D wavelet noise can be used to texture a 2D surface in a way that maintains its band-limited character. The technique is also easy to implement and fast (an implementation is provided in the Appendices).

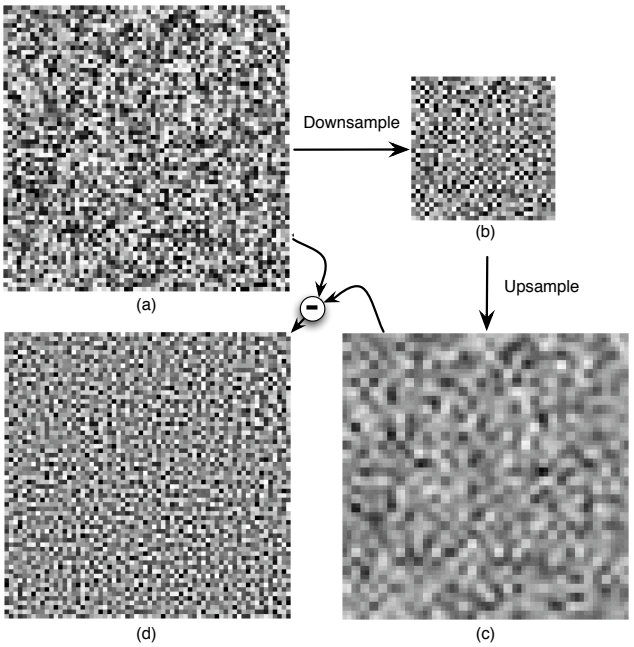


Figure 2: *(a). Image R of random noise, (b) Half-size image $R^{\downarrow}$, (c) Half-resolution image $R^{\downarrow\uparrow}$, (d) Noise band image $N = R - R^{\downarrow\uparrow}$.*

2 Overview

To provide a context for the remainder of the paper and to emphasize the simplicity of the basic algorithm, we begin by presenting a high-level overview, using the two-dimensional case and glossing over the details. The essence of our algorithm consists of the following four steps, which are illustrated in Figure 2:

1. Create an image R filled with random noise. (Figure 2a)
2. Downsample R to create the half-size image $R^{\downarrow}$. (Figure 2b)
3. Upsample $R^{\downarrow}$ to a full size image $R^{\downarrow\uparrow}$. (Figure 2c)
4. Subtract $R^{\downarrow\uparrow}$ from the original R to create N. (Figure 2d)

N is thus created by taking R and removing the part that is representable at half-size. What's left is the part that is *not* representable at half-size, i.e., the band-limited part. N is then used just like Perlin noise to construct noise patterns. This construction is similar in spirit to the procedural band-pass pyramids of [Perlin and Velho 1995].

The mathematics for deriving and justifying this approach, particularly the filters used in the downsampling and upsampling steps, is presented in Section 3. Practical implementation considerations are then discussed in Section 4, and results are presented in Section 5.

3 Theory

In this section, we provide the mathematical underpinnings for our algorithm. For now, we work in one dimension; the extension to more dimensions is straightforward and is covered in Section 3.6.

Perhaps the most obvious way to construct band-limited noise would be to use spectral methods, which are based on sinc filters. Instead, the approach we take uses wavelets, which are more general than spectral methods in that they work with a wide variety of basis functions. This flexibility allows us to use basis functions that have a finite extent and that more closely match the filters used in renderers.

We construct our multiresolution noise $M(x)$ in a fashion similar to Perlin, that is, by summing scaled and attenuated versions of a noise band $N(x)$. Specifically,[1]

$$M(x) = \sum_{b=b_{min}}^{b_{max}} w_b N(2^b x) \tag{1}$$

where b indexes the band, and where the weights w_b are free variables used to control the spectral character of $M(x)$. We differ from Perlin in the definition and construction of $N(x)$.

We begin by modeling how noise is used in the rendering process. Renderers typically use a filter kernel to turn a scene function into pixels. In particular, the value of the i-th pixel is

$$Pixel(i) = \int S(x)K(x-i)dx \tag{2}$$

where $S(x)$ represents the scene being rendered and $K(x-i)$ is the renderer's filter kernel $K(x)$ translated so that it is centered at pixel i. $K(x)$ is usually a non-negative, locally supported function.

The scene function $S(x)$ typically contains a mixture of declarative data, such as the positions and orientations of geometric primitives, together with procedural components, such as procedural shaders. Since the scene can be an arbitrarily complicated function of noise, no noise function can possibly prevent aliasing under all conditions. (As an extreme example, consider a procedural shader that returns white if the low order bit of $M(x)$ is one, and black otherwise.) Instead, we devise a method that guarantees no aliasing or loss of detail under ideal conditions, and then show that it behaves well in all but the most pathological of non-ideal conditions.

3.1 Orthogonality

Under our ideal conditions, the scene being rendered, when restricted to the region contributing to pixel i, is locally well approximated by a scaled and translated version of multiresolution noise. If s is the scale of the scene at pixel i, and k is the offset, then our assumption is that $S(x) \approx M(2^s(x-k))$, in which case the value of i-th pixel is given by:

$$\begin{aligned} Pixel(i) &= \int M(2^s(x-k))K(x-i)dx && (3)\\ &= \int \sum_{b=b_{min}}^{b_{max}} w_b N(2^{s+b}(x-k))K(x-i)dx && (4)\\ &= \sum_{j=b_{min}+s}^{b_{max}+s} w_{j-s} \int N(2^j x-\ell)K(x-i)dx && (5) \end{aligned}$$

where for notational convenience, in the last equation we substituted $s+b \to j$ and $2^{s+b}k \to \ell$.

Our objective is to allow shader writers to safely truncate this summation without loss of detail and without aliasing. We assume without loss of generality that the resolution of the image corresponds to $j=-1$, so that the noise band at resolution $j=-1$ is fully representable but noise bands at resolutions $j \geq 0$ are not representable. We want these unrepresentable bands to have no contribution to the image, which means that:

$$\int N(2^j x-\ell)K(x-i)dx = 0 \tag{6}$$

for all $j \geq 0$ and for all i and ℓ. This condition means that the fine scale versions of $N(x)$ are *orthogonal* to the renderer's filter kernel $K(x)$. In other words, our method is based on the principle that the contribution of the noise function to the image should vanish exactly when aliasing would occur.

[1] A band-dependent translation is usually added to $2^b x$ to de-correlate the noise bands, but we can safely ignore this in our analysis.

We know of no technique that can find such an $N(x)$ for an arbitrary filter kernel $K(x)$ and arbitrary parameters j and ℓ. However, given relatively mild restrictions on $K(x)$, we can use wavelet analysis to achieve exact orthogonality when j and ℓ are integers. With this objective in mind, Sections 3.2 and 3.3 review some needed results from wavelet analysis[2] in preparation for constructing our orthogonal noise band $N(x)$ in Section 3.4. We then show in Section 3.5 that the deviation of our noise function from orthogonality is small in real-world situations.

3.2 Refinability and Upsampling

Much of computer graphics involves representing functions as a weighted sum of basis functions. For example, B-spline curves are represented as weighted sums of B-spline basis functions, and a monitor approximates an image as a weighted sum of the spot shapes of the pixels.

Starting with a basis function $\phi(x)$ centered at $x = 0$, other functions $F(x)$ can be built by taking linear combinations of $\phi(x)$ translated by integer amounts i:

$$F(x) = \sum_i f_i \phi(x-i) \tag{7}$$

where $\phi(x-i)$ is a version of $\phi(x)$ centered around $x = i$, and where the f_i are the coefficients of the representation.

The set of all functions that can be represented by varying the coefficients f_i forms a vector space that we call the resolution 0 space and denote by $\mathbf{S}^0$:

$$\mathbf{S}^0 := \{F(x) | F(x) = \sum_i f_i \phi(x-i)\} \tag{8}$$

A larger, resolution 1 space $\mathbf{S}^1$ of functions can be represented by scaling down the width of $\phi(x)$ by a factor of two (2^1), then translating along the half integers. Functions $G(x)$ in this space take the form

$$G(x) = \sum_i g_i \phi(2x-i) \tag{9}$$

A key idea in wavelet analysis is to ensure that all the functions in $\mathbf{S}^0$ are also in $\mathbf{S}^1$; that is, that $\mathbf{S}^0 \subset \mathbf{S}^1$. This guarantees that $\mathbf{S}^1$ enriches the space $\mathbf{S}^0$, rather than replacing it with a completely different set of functions. This subset relationship is guaranteed if there exist coefficients p_k that allow $\phi(x)$ to be written in terms of $\phi(2x-k)$:

$$\phi(x) = \sum_k p_k \phi(2x-k) \tag{10}$$

When such coefficients do exist, $\phi(x)$ is said to be *refinable*. Examples of refinable functions include uniform B-spline basis functions (of any degree), sinc functions, and Daubeschies basis functions. Figure 3 illustrates the refinability of uniform quadratic B-splines. Gaussians are good examples of functions that are not refinable.

We henceforth assume that $\phi(x)$ is refinable. Given the coefficients f_i that represent a function $F(x)$ in $\mathbf{S}^0$ as in Equation 7, there exist coefficients $f_i^{\uparrow}$ that represent that function exactly in $\mathbf{S}^1$:

$$F(x) = \sum_i f_i^{\uparrow} \phi(2x-i) \tag{11}$$

Using the refinement equation, Equation 10, it is straightforward to show that

$$f_i^{\uparrow} = \sum_k p_{i-2k} f_k \tag{12}$$

[2] A number of texts, such as [Chui 1992] and [Stollnitz et al. 1996], describe this material in more detail.

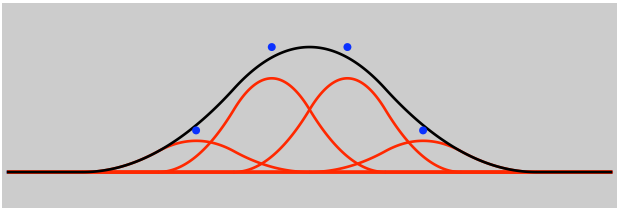


Figure 3: *Refinement of a uniform quadratic B-spline. The black curve is the basis function $\phi(x)$. The red curves are $\phi(2x-k)$, integer-offset half-width copies of $\phi(x)$ that have been attenuated by the refinement coefficient p_k. The blue dots show the p_k values $(1/4, 3/4, 3/4, 1/4)$ for this basis. This basis function is refinable because the sum of the red curves is exactly equal to the black curve.*

The sequence $(..., f_i^{\uparrow}, ...)$ is twice as long as the sequence $(..., f_k, ...)$, so Equation 12 represents an upsampling filter; this is in fact the upsampling filter used in Step 3 of our algorithm. This filter, defined solely by the refinement coefficients and hence by the basis function, describes how to convert a function represented at resolution 0 to its resolution 1 representation. For quadratic B-splines, all of the refinement coefficients p_k are 0 except for:

$$p_{-1} = 1/4, \quad p_0 = 3/4 \quad p_1 = 3/4, \quad p_2 = 1/4 \tag{13}$$

3.3 Wavelets and Downsampling

Above we established that refinability guarantees that every member of $\mathbf{S}^0$ can be represented exactly as a member of the higher-resolution space $\mathbf{S}^1$. The converse, however, is not true: not every function $G(x)$ in $\mathbf{S}^1$ can be represented exactly in the lower-resolution space $\mathbf{S}^0$; in general there is some lost detail. However, this loss of detail can be minimized in a least squares sense by dividing $G(x)$ into two parts:

$$G(x) = G^{\downarrow}(x) + D(x) \tag{14}$$

where

- $G^{\downarrow}(x)$ is the least squares best approximation to $G(x)$ in $\mathbf{S}^0$.
- $D(x)$ is the least squares residual and contains the information in $\mathbf{S}^1$ that *cannot* be represented in $\mathbf{S}^0$.

It follows that $D(x)$ and all of its integer translates are orthogonal to all functions in $\mathbf{S}^0$. That is,

$$\int D(x-\ell)\phi(x-i)dx = 0 \tag{15}$$

for all integers ℓ and i. The set of all such detail functions $D(x)$ forms a vector space called the wavelet space $\mathbf{W}^0$.

Given the coefficients g_i for $G(x)$, the coefficients $g_i^{\downarrow}$ for $G^{\downarrow}(x)$ can be determined using a standard result from wavelet analysis. Specifically, there exist so-called analysis coefficients a_k, again depending only on the basis function $\phi(x)$, such that

$$g_i^{\downarrow} = \sum_k a_{k-2i} g_k \tag{16}$$

This is a downsampling filter that projects the coefficient sequence $(..., g_k, ...)$ into a sequence $(..., g_i^{\downarrow}, ...)$ of half the length in a least squares optimal way. The analysis coefficients a_k comprise the downsampling filter of Step 2 in our algorithm. For quadratic B-splines, the sequence $(..., a_k, ...)$ is infinitely long, but it decays quickly and is well approximated by a finite sequence. The variable a in Appendix 1 contains this sequence for quadratic B-splines.

Because $\phi(x)$ is refinable, it can be represented exactly in terms of $\phi(2^j x)$ for all integers $j \geq 0$. From this and Equation 15, it can readily be shown that

$$\int D(2^j x-\ell)\phi(x-i)dx = 0 \tag{17}$$

for all integers $j \geq 0$ and all integers ℓ and i. This is exactly the condition required for our noise band $N(x)$ in Equation 6. Thus, *if we build our noise bands in the wavelet space* $\mathbf{W}^0$*, then they can be scaled to any resolution* j *and be guaranteed to have no effect on images at any resolution less than* j.

3.4 Constructing noise bands

Now we are ready to construct our noise band $N(x)$ so that it lies in the wavelet space $\mathbf{W}^0$ generated by the filter kernel $K(x)$. At this point, we must choose a particular filter kernel $K(x)$, which we take to be the uniform quadratic B-spline basis function $B(x)$. We do so for several reasons:

- It is a good approximation to the filter kernels used by most renderers (including Gaussians);
- it has a small support and low degree, which makes the evaluation of N(x) simple and computationally efficient; and
- it generates differentiable noise, meaning that noise gradients, which are also useful in procedural shading, are continuous.

One could, however, use this same technique to create noise for any other refinable basis function, such as uniform cubic B-splines or sincs.

What follows here is a more detailed and precise version of the algorithm presented in Section 2. Code to implement this algorithm (in three dimensions) is provided in Appendix 1.

1. **Create $R(x)$ using a random number generator.**

   Use a random number generator to create a coefficient sequence $R = (\ldots, r_i, \ldots)$. This sequence defines a function $R(x)$ in $\mathbf{S}^1$. Using the basis function $B(x)$, Equation 9 becomes:

   $$R(x) = \sum_i r_i B(2x - i) \tag{18}$$

   An example of such a function is shown in Figure 4(a).

2. **Compute $R^{\downarrow}(x)$ by downsampling $R(x)$.**

   As shown in Equation 14, $R(x)$ can be decomposed into two parts: a part in $\mathbf{S}^0$, denoted by $R^{\downarrow}(x)$, and a detail part in $\mathbf{W}^0$, which is the function $N(x)$ whose coefficients we seek:

   $$R(x) = R^{\downarrow}(x) + N(x) \tag{19}$$

   $$N(x) = R(x) - R^{\downarrow}(x) \tag{20}$$

   Using Equation 7, we get:

   $$R^{\downarrow}(x) = \sum_i r_i^{\downarrow} B(x - i) \tag{21}$$

   The coefficients $r^{\downarrow}$ may be computed from the coefficients r using Equation 16:

   $$r_i^{\downarrow} = \sum_k a_{k-2i} r_k \tag{22}$$

   An example of $R^{\downarrow}(x)$ is shown in Figure 4(b).

3. **Compute $R^{\downarrow\uparrow}(x)$ by upsampling $R^{\downarrow}(x)$.**

   Using Equations 18 and 21, we can express Equation 20 as:

   $$N(x) = \sum_i r_i B(2x - i) - \sum_i r_i^{\downarrow} B(x - i) \tag{23}$$

   The first sum in this equation is an $\mathbf{S}^1$ representation that uses the basis function $B(2x - i)$; the second sum is an $\mathbf{S}^0$ representation that uses the basis function $B(x - i)$. To combine these sums, we need them to have a common basis, which is possible because by upsampling we can represent the $\mathbf{S}^0$ part $R^{\downarrow}(x)$ exactly in $\mathbf{S}^1$ as:

   $$R^{\downarrow\uparrow}(x) = \sum_i r_i^{\downarrow\uparrow} B(2x - i) \tag{24}$$

   where the coefficients $r^{\downarrow\uparrow}$ may be computed by upsampling the coefficients $r^{\downarrow}$ using Equation 12:

   $$r_i^{\downarrow\uparrow} = \sum_k p_{i-2k} r_k^{\downarrow} \tag{25}$$

   An example of $R^{\downarrow\uparrow}(x)$ is shown in Figure 4(c).

4. **Compute $N(x)$ by subtracting $R^{\downarrow\uparrow}(x)$ from $R(x)$.**

   Using Equation 24, we can now rewrite Equation 23 as:

   $$N(x) = \sum_i r_i B(2x - i) - \sum_i r_i^{\downarrow\uparrow} B(2x - i) \tag{26}$$

   $$= \sum_i n_i B(2x - i) \tag{27}$$

   where the coefficients of $N(x)$ are given by $n_i = r_i - r_i^{\downarrow\uparrow}$. An example of the resulting noise band $N(x)$ is shown in Figure 4(d).

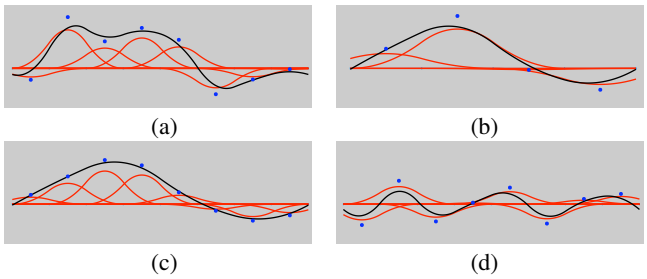


Figure 4: *An example of wavelet noise in one dimension: (a) a function $R(x)$ built using random coefficients with B-spline basis functions; (b) the downsampled version $R^{\downarrow}(x)$; (c) the re-upsampled version $R^{\downarrow\uparrow}(x)$; (d) the noise function $N(x) = R(x) - R^{\downarrow\uparrow}(x)$. Functions appear in black, their coefficients are plotted as blue dots, and the basis functions multiplied by their corresponding coefficients appear in red. Note that the basis functions in (b) are twice the width of those in the other figures, and that the function $R^{\downarrow}(x)$ in (b) and the function $R^{\downarrow\uparrow}(x)$ in (c) are identical.*

Once the coefficients n_i have been determined, a value of $N(x)$ for a given x can be computed using any evaluation method for quadratic B-splines (cf. [Farin 2002]). The code in Appendix 2 shows this computation in 3 dimensions.

As established in Section 3.3, this construction guarantees that all bands of noise $N(2^j x)$ for $j \geq 0$ are orthogonal to $\mathbf{S}^0$, and hence do not contribute to the image. Additionally, noise bands are orthogonal to each other, which makes spectral shaping more controllable.

3.5 Fractional scales and translates

The noise band $N(x)$ has been constructed so that $N(2^j x - \ell)$ contributes nothing to pixel values when ℓ is an arbitrary integer and when j is a non-negative integer. However, in practice j and ℓ are not always integers, and in these cases noise bands for which $j \geq 0$ will have some contribution to the pixel; omitting them from the summation will cause some detail to be lost. Fortunately the contribution falls off very rapidly as j increases.

As an indication of this falloff, we have numerically studied the contribution function

$$C(j, \ell) := \int N(2^j x - \ell) B(x) dx \tag{28}$$

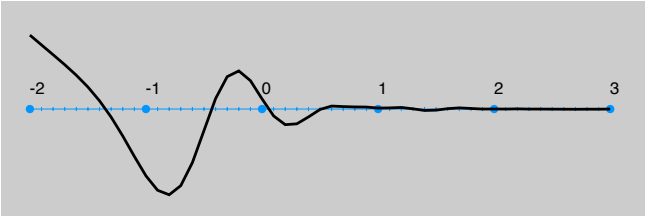


Figure 5: *The contribution function $C(j,1/2)$ for $-2 \le j \le 3$.*

as a continuous function of j and ℓ, using various instances of our noise band $N(x)$. For fixed j, the function reaches a maximum at $\ell = 1/2$. The rate of falloff of the maximum contribution can therefore be seen by plotting $C(j,1/2)$ as function of j, as is done in Figure 5. The falloff is rapid, so we can safely truncate the summation at $j = 0$ with minimal loss of detail. (In practice, to ensure that the contribution completely vanishes at $j = 0$ even for non-integer ℓ, we fade out the noise as we approach $j = 0$.)

3.6 Noise in multiple dimensions

The two-dimensional noise function $N(x,y)$ is constructed from a two-dimensional array n_{i_x,i_y} of coefficients that are blended together to form a uniform quadratic tensor-product B-spline:

$$N(x,y) = \sum_{i_x,i_y} n_{i_x,i_y} B(2x - i_x) B(2y - i_y) \tag{29}$$

The coefficient array $N = (..., n_{i_x,i_y}, ...)$ is obtained from an arbitrary array $R = (..., r_{i_x,i_y}, ...)$ by applying the downsampling filter and then the upsampling filter to each row of R to obtain an array $R^{\downarrow\uparrow}$. Next, the downsampling filter followed by the upsampling filter is applied to each of the columns of $R^{\downarrow\uparrow}$ to obtain an array ${}^{\uparrow\downarrow}R^{\downarrow\uparrow}$. This array is then subtracted from the initial array R to produce the noise coefficient array N. Generation of coefficients for noise in three and higher dimensions is accomplished in a similar fashion, as shown in the code in Appendix 1.

In analogy to the orthogonality condition for one-dimensional noise, the two-dimensional noise function $N(x,y)$ satisfies the two-dimensional orthogonality condition:

$$\int N(2^j x, 2^j y) B(x - i_x) B(y - i_y) dx\,dy = 0, \quad j \ge 0 \tag{30}$$

meaning that bands of noise for $j \ge 0$ can be safely truncated.

Similarly, in three dimensions, $N(x,y,z)$ satisfies

$$\int N(2^j x, 2^j y, 2^j z) B(x - i_x) B(y - i_y) B(z - i_z) dx\,dy\,dz = 0 \tag{31}$$

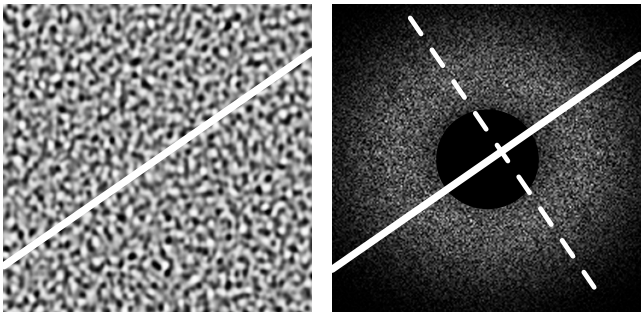
for $j \ge 0$, meaning that bands of noise for $j \ge 0$ can be safely truncated. This integral is relevant when 3D noise is used to texture a volume.

3.7 Projected noise

Texture values on a 2D surface are frequently determined by sampling a 3D noise function. This approach runs into a fundamental problem, however: a 2D slice through a 3D band-limited function is in general not band-limited. As far as we are aware, this fact does not seem to have been previously recognized in the texture synthesis literature.[3] As an example, Figures 8(a) and (b) show how 3D Perlin noise is somewhat band-limited, but a planar slice through it is not band-limited at all.

To see why this is so, we use the Fourier Slice Theorem (cf. [Malzbender 1993]), which states that the Fourier transform of a slice is obtained by integrating the Fourier transform along the direction perpendicular to the slice. For example, the 1D Fourier transform of the slice shown in Figure 6(a) is obtained by integrating the 2D Fourier transform shown in Figure 6(b) along lines perpendicular to each point on the white line. Notice that the slice can contain arbitrarily low frequencies because the low-frequency line integrals (i.e., near the origin of the Fourier transform) pick up contributions from non-zero high-frequency parts of the 2D Fourier transform. For example, the integral along the dotted line, which determines a low frequency, is in general non-zero since it passes through the higher frequencies in the band-pass.

[3] [Lewis 1989] observed this phenomenon in a 1D slice through 2D Perlin noise but attributed it to Perlin's interpolation method rather than the slicing process itself.



(a) (b)

Figure 6: *A band-limited function (a) and its Fourier transform (b). The Fourier transform of the white slice in (a) is obtained by integrating the 2D Fourier transform along the direction perpendicular to the white slice shown in (b).*

Fortunately, we can use the wavelet orthogonality machinery to address this issue. Consider first the case of a noise-axis-aligned planar slice such as the plane $2^j z = z_0$. Rather than texturing the plane at (x,y) with the 3D function $N(2^j x, 2^j y, z_0)$, we instead use the 2D function $N_{z_0}(2^j x, 2^j y)$, where

$$N_{z_0}(2^j x, 2^j y) := \int N(2^j x, 2^j y, 2^j z) B(z - z_0) dz \tag{32}$$

This shows how to project wavelet noise onto a surface: instead of simply point sampling the texture at the intersection point, we perform a line integral orthogonal to the surface, where the integrand is the 3D noise weighted by a 1D filter kernel with its center at the point of intersection. Using Equation 31, this definition guarantees the orthogonality condition we desire:

$$\int N_{z_0}(2^j x, 2^j y) B(x - i_x) B(y - i_y) dx\,dy = 0 \tag{33}$$

for all integer z_0 and $j \ge 0$.

Moreover, Equation 32 is a projection operator from the tri-variate wavelet space $\mathbf{W}^0(x,y,z)$ to the bi-variate wavelet space $\mathbf{W}^0(x,y)$. (This can be established by expanding $N(x,y,z)$ in a wavelet basis for $\mathbf{W}^0(x,y,z)$.) As a consequence, the other important property, band orthogonality, is also preserved: i.e., $N_{z_0}(2^i x, 2^i y)$ and $N_{z_0}(2^j x, 2^j y)$ are orthogonal whenever $i \ne j$.

In some cases, this axis-aligned projection is just what we want. For example, when projecting 4D noise (indexed by xyz and time) onto 3D images (xyz) in an animated sequence, the projection is along the time dimension so the integration is axis-aligned. But in many cases the projection is not axis-aligned, so the integration is not axis-aligned. In our numerical experiments that consider non-integer j and non-axis alignment, we have found that the deviation from orthogonality is small in the non-axis-aligned case and falls off rapidly in j in a fashion similar to that shown in Figure 5.

4 Implementation

In this section we examine a variety of issues that arise in practical implementations of wavelet noise.

4.1 Noise evaluation

In practice, renderers approximate the integral of Equation 2 using a quadrature formula. In the case where $S(x) = N(x)$ this becomes

$$Pixel(i) \approx \sum_{q=1}^{Q} B_q N(x_q) \tag{34}$$

where the quadrature weights B_q are typically just set to $B(x_q)$, but they could also be weights that provide higher order accuracy, using for instance Gaussian quadrature. If the renderer's quadrature formula is sufficiently accurate, we can use $N(x_q)$ directly. $N(x_q)$ is calculated by interpolating the coefficients n_i around $x = x_q$ using $B(x - x_q)$, which is 3 noise coefficients wide.

However, if the number of samples Q is too small, a more accurate value for the pixel can be obtained by replacing $N(x_q)$ by $\bar{N}(x_q)$, a weighted average of the value of $N(x)$ in the vicinity of x_q. In the limit of one sample ($Q = 1$), we are setting $Pixel(i) = \bar{N}(i)$, so $\bar{N}(i)$ would ideally be equal to the value of integral in Equation 2. The optimal computation of $\bar{N}(x_q)$ depends on the details of the quadrature calculation of the renderer, but a reasonable and convenient approximation is to use a widened version of $B(x)$ for the finest bands. For example, for $Q = 1$, we have had good success interpolating the noise coefficients using $B((1+2^j)(x - x_q))$, which is 6 noise coefficients wide at $j = 0$ and 4.5 wide at $j = -1$, and then gradually unwidening the basis function in the region $-2 < j \leq -1$ so that for $j \leq -2$, we use the regular, unwidened $B(x - x_q)$.

The projected noise integral in Equation 32 can be expressed in terms of the noise coefficients n as (taking $j = 0$ for simplicity):

$$N_{z_0}(x,y) = \sum_{i_x i_y i_z} n_{i_x i_y i_z} B(2x - i_x) B(2y - i_y) \beta(2z_0 - i_z) \tag{35}$$

where

$$\beta(z) = \int B(z') B(2z' - z) dz' \tag{36}$$

is a piecewise quintic whose support covers 9 noise coefficients. This quintic can be closely approximated by $\beta(z) \approx B(\frac{z+3/2}{2})$, which is a double-width quadratic B-spline, whose support covers 6 coefficients as shown in Figure 7. This allows us to combine projection and evaluation using a single asymmetric tri-variate basis function. Note that if the normal is not parallel to one of the axes of the noise coefficient grid, the computation will be less efficient because the support of the basis function will not be aligned with the grid. Code for this is included in Appendix 2.

4.2 Noise variance

Another issue with Perlin noise is that there is no model for the expected statistical distribution of the intensity values or how that distribution changes as a function of the number of bands and their weights. By contrast, as we show in this section, the statistical distribution of wavelet noise can be calculated and controlled.

Every step leading from the random variables r_i to the noise coefficients n_i is linear, meaning that the noise coefficients n_i are linear combinations of the r_i. A standard result from statistics says that a linear combination of independent, identically distributed Gaussian random variables is again a Gaussian random variable. Specifically, let $x_1, \ldots, x_n$ be Gaussian random variables with variance σ_x^2. The variable

$$y = \sum_{i=1}^{n} w_i x_i \tag{37}$$

has variance

$$\sigma_y^2 = \sigma_x^2 \sum_{i=1}^{n} w_i^2 \tag{38}$$

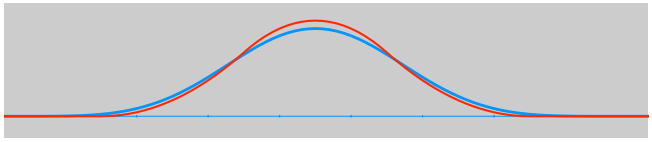


Figure 7: *The quintic projection function $\beta(z)$ (in blue) can be closely approximated by a quadratic basis function (in red).*

Thus if the r_i are Gaussian, the n_i will be too. There are various standard ways to generate such values r_i; we use the polar form of the Box-Muller transformation (cf. [Knuth 1997]) which gives us a Gaussian distribution with a mean of 0 and a variance of 1. The variance σ_n^2 of the n_i can then be determined analytically using the r_i weights that are equivalent to the upsample, downsample, subtract process; however, it is simpler to calculate σ_n^2 numerically from the n_i values.

$N(x)$ is computed as a linear combination of the n_i using a set of basis functions evaluated at x. As a result, the variance σ_N^2 of $N(x)$ differs from σ_n^2 by the sum of squares of evaluated basis functions. We could use a different variance for every x, but we have found it preferable to use the average variance. The former allows a bit more control over the variance; the latter preserves orthogonality. The average variance can be determined analytically, but once again it is simpler to evaluate it numerically. For quadratic B-splines, this average σ_N^2 is 0.265 for 2D noise, 0.210 for 3D noise, and 0.296 for 3D noise projected onto a 2D surface.

As bands of noise are added, the variance is further changed by the band weights w_b. The variance of the final multiresolution noise $M(x)$ is:

$$\sigma_M^2 = \sigma_N^2 \sum_{j=j_{min}}^{0} w_j^2 \tag{39}$$

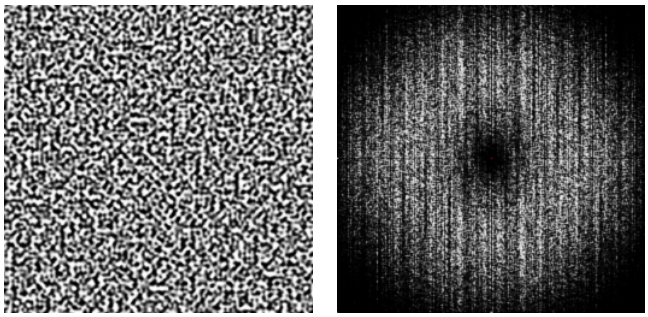
Because the values in $M(x)$ have a Gaussian distribution with a predictable variance, we can change the values to match any desired distribution. To match a Gaussian distribution with a different variance, we simply scale the results; for a white distribution, we use the *erf* function (cf. [Abramowitz and Stegun 1970]); otherwise we build a lookup table. This control is useful, but making the distribution non-Gaussian does violate the orthogonality assumption. Like any extreme transformations applied to the noise, abrupt transitions within the desired distribution can lead to aliasing during animation, but we have found that smoothly varying distributions do not introduce any noticeable artifacts.

4.3 Noise tiles

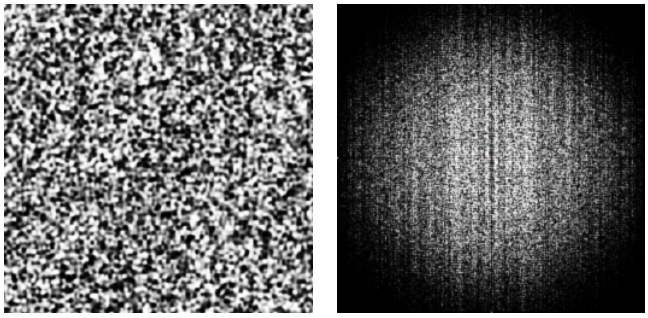
The algorithm as described so far is not very efficient because the width of the downsampling filter makes the generation of the noise coefficients expensive. Instead of incurring this expense for every noise sample during rendering, we can use a small pre-computed volume of noise coefficients and then tile our space with that volume.[4] The tile can optionally include a border the width of the basis function so that tile boundary checks can be avoided in the evaluation loop.

The downside of using noise tiles is that they introduce low-frequency repeating patterns into the noise. Fortunately, because the noise band contains only small-scale features, this large-scale repetition is often not objectionable and can just be ignored. For some applications, it may be appropriate to use a larger tile to make

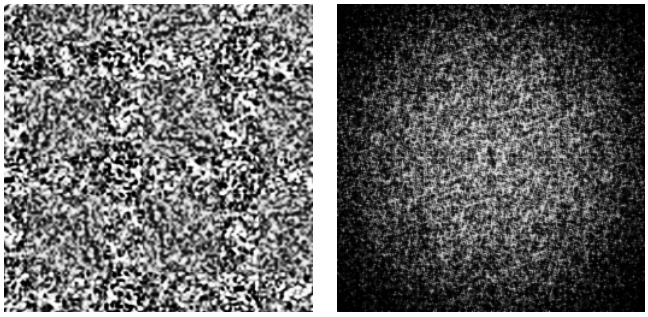
[4] Although it isn't immediately obvious, the tile size must be even; this is because the period of $R^{\downarrow\uparrow}(x)$ is equal to the period of $R(x)$ if the size is even, but twice the period of $R(x)$ if it is odd.



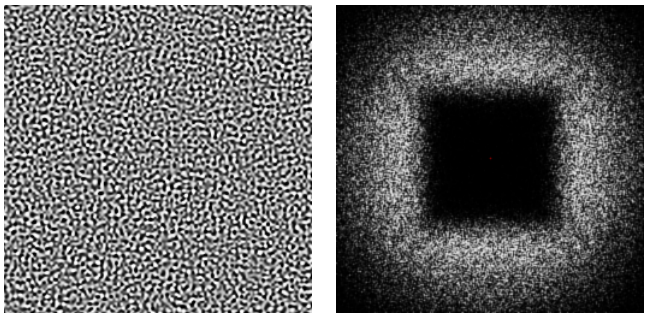
(a) 2d Perlin noise



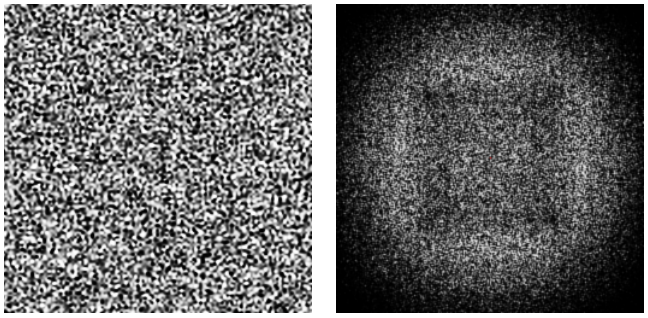
(b) 2D slice through 3D Perlin noise



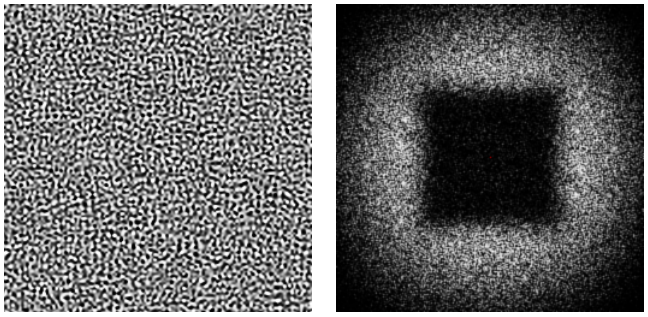
(c) 2D white noise



(d) 2d wavelet noise



(e) 2D slice through 3D wavelet noise



(f) 3D wavelet noise projected onto 2D

Figure 8: *Noise patterns (left) with their Fourier transforms (right). For Perlin noise, we use the RenderMan implementation of [Perlin 2002].*

the repetition less obvious. For other applications, the repeating pattern can be eliminated by using a hash function to select a different noise tile to use in each tile location. This strategy avoids repetition but creates discontinuities at the tile boundaries. These discontinuities can be avoided by overlapping the tiles slightly and blending from one tile to the other, but this creates another artifact: blending reduces the variance of the noise as noted in Section 4.2. Using Equation 38, if the tiles are blended using α of one tile and $1-\alpha$ of another, the variance of the blended value is reduced by a factor of $\alpha^2+(1-\alpha)^2$. This appears as areas of reduced contrast along the tile borders. We can correct this simply by dividing the blended tile values by the variance reduction. Since the α values are a function of the location within the tile, we can pre-multiply the noise coefficients by the variance-corrected α values at tile creation time, eliminating the run-time cost of the α multiplication and variance correction. To reduce storage, we can store a single tile pattern and create 48 different variations by permuting the x, y, and z tile indices (6 possibilities) and by stepping through the coefficients in reverse order in x, y, or z (2x2x2 possibilities). This requires storing only 4 numbers per variation: step sizes in x, y, and z and a pointer to the (0,0,0) location.

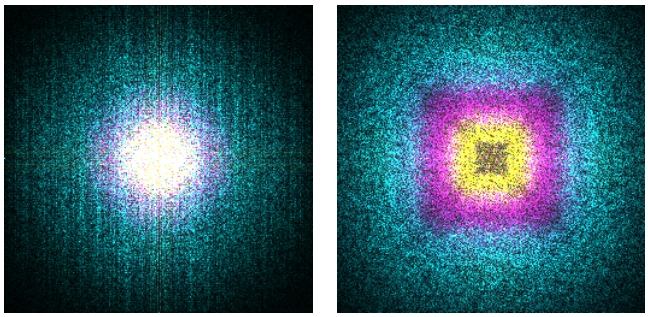
Finally, we note that the upsampling and downsampling filters treat the even and odd indexed coefficients differently, which introduces a subtle variation in variance. This variation is easily eliminated by adding two noise tiles offset by an odd number of coefficients in each dimension. This can be done at tile creation time so that there is no additional run-time cost. This was done prior to the computation of the variances listed in Section 4.2.

5 Results

5.1 Fourier transforms

A comparison of Perlin and wavelet noise patterns and their Fourier transforms is shown in Figure 8. The center of each Fourier transform corresponds to the DC term, with x frequencies increasing to the right and y frequencies increasing vertically. A bright spot at $x = 0.25$, $y = 0$, for example, would indicate a signal that repeats every 4 pixels in the x direction. The Fourier transform of a noise band should be dark where $|x|$ and $|y|$ are both < 0.5, with no other patterns. Wavelet noise (Figure 8(d)) has these characteristics. Perlin noise (Figure 8(a)), by contrast, contains significant energy in the low frequency areas near the center of the Fourier transform.

When we look at the Fourier transform of a 2D slice through 3D noise, Perlin noise (Figure 8(b)) becomes as non-band-limited as 2D white noise (Figure 8(c)). Wavelet noise (Figure 8(e)) fairs



(a) (b)

Figure 9: *Fourier transforms of 3 bands of 3D noise projected onto 2D with (a) Perlin noise and (b) wavelet noise. In both images, the middle band (magenta) should nestle exactly between the next higher band (cyan) and the next lower band (yellow).*

better, but low frequencies still leak through. When we project wavelet noise onto a 2D surface using the technique described in Section 3.7, however, the band limits are preserved (Figure 8(f)). Figure 9 shows the Fourier transforms of three adjacent bands of noise in different colors and illustrates the band-limiting character of wavelet noise vs. Perlin noise.

The square nature of the wavelet noise Fourier transform reflects the asymmetry of our separable quadratic B-spline basis function and is less evident for non-axis-aligned views. We have not found this asymmetry to be objectionable, but it can be reduced by either (1) averaging multiple tiles with different orientations or (2) using a higher B-spline, thus more closely approximating a Gaussian, which is symmetric.

5.2 Run-time cost

The run-time cost of noise depends on many factors of the specific implementation, but a rough guide to the performance is the number of noise coefficients involved in the interpolation calculation. The run-time cost of the basic wavelet noise algorithm involves 27 (3x3x3) noise coefficients compared to the 24 (2x2x2 triples) used in Perlin noise. So we would expect the cost of the two techniques to be similar.

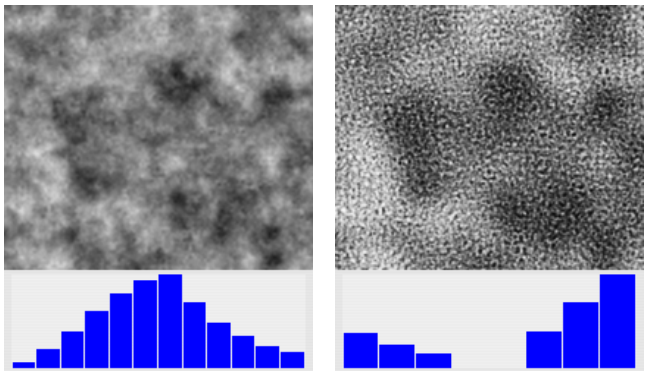
In timing tests on a 1.4 GHz Macintosh G4, however, our moderately optimized implementation of wavelet noise was about 30% faster than the highly optimized RenderMan implementation of Perlin noise (0.81 μsec vs. 1.06 μsec per noise evaluation). When we include tile meshing (Section 4.3) with a tile overlap of 1/16 of a tile, the cost of wavelet noise increased to 0.97 μsec, which is still about 10% faster than Perlin noise.

Wavelet noise uses more memory than Perlin noise, but even with a relatively large tile the memory requirements are small by today's standards. For example, a 128x128x128 tile uses 8 Mb compared to about 16 Kb for the RenderMan implementation of Perlin noise.

Using the optional parts of our technique adds additional cost. Projecting 3D noise onto a 2D surface (Sections 3.7 and 4.1) approximately doubles the cost in the axis-aligned case; the non-axis-aligned case is more expensive because the interpolation is less efficient. Basis function widening (Section 4.1) approximately triples the cost of the finest band, doubles the cost of the second band, and doesn't affect coarser bands. The decision of which of these options to use will depend on the application.

5.3 Examples

We have embedded our implementation of wavelet noise in a 2D testbed with controls for shaping the weights of the bands and for specifying the desired distribution of the result. Figure 10 shows two patterns produced by this program. Because of the independence of our noise bands, the spectral shape is accurately controlled by the band weights so that we have not found it necessary to provide a lacunarity control. The disadvantage of values of lacunarity less than 2 is that they make the bands overlap and lose their mutual orthogonality.



(a) (b)

Figure 10: *2D noise patterns with (a) 12 bands with a Gaussian distribution and (b) 8 bands with a white distribution. The blue bars are the band weights.*

We have also embedded our implementation of wavelet noise in a RenderMan shader DSO. Figures 1(a) and 1(b) are stills from two animations that compare Perlin and wavelet noise. The geometry is the same in both animations; they differ only in the version of noise used. The level of noise used in Figure 1(a) shows the amount of detail obtainable using the RenderMan version of Perlin noise. Figure 1(b) shows the same scene rendered using wavelet noise using the same noise band weights.

5.4 Infinite noise

Band weights usually correspond to fixed scales in world space. Another option we have found useful is to make the weights *relative* to each location's scale in screen space. This produces *infinite noise*, the ability to zoom into a scene indefinitely, with low frequencies gradually fading out as higher frequencies gradually fade in. We have found that this trick works remarkably well and that the fading out is not noticeable if it's done over a few bands (as in Figure 10(a), for example). We speculate that this is because it echoes the way human perception adapts to the average intensity of a scene.

6 Summary

The wavelet noise preprocessing calculations are:

- Create a tile of noise coefficients by filling the tile with random noise, then downsampling, upsampling, and subtracting (Section 3.4 and Appendix 1).
- (Optional) Add two such tiles together to correct for even vs. odd variance (Section 4.3 and Appendix 1).
- (Optional) Adjust the tile borders to accommodate tile meshing (Section 4.3).

The run-time calculations for each band b, given a location and an object scale s, are:

- Determine the noise resolution j from s and b (Section 3.1 and Appendix 2).
- (Optional) Use a hash function to determine which tile pattern(s) to use for this location (Section 4.3).

- (Optional) If the renderer undersamples, widen the basis function in the tangent directions in the last bands (Section 4.1).
- (Optional) Project the 3D noise onto the 2D surface by doubling the support of the basis function in the direction normal to the surface (Section 3.7 and Appendix 2).
- Evaluate the noise function by multiplying the noise coefficients by the basis function (Section 3.4 and Appendix 2).
- (Optional) Correct the noise for the desired distribution (Section 4.2 and Appendix 2).

In conclusion, we have shown that wavelets are well suited to producing noise for use in procedural textures. Because the bands are orthogonal, they provide a set of independent controls over the shape of the spectrum. The distribution of the final result can be predicted and controlled. Most importantly, the noise is truly band-limited, so that virtually all of the detail can be rendered with minimal aliasing, even when projecting 3D noise onto a 2D surface.

Acknowledgements

Thanks to Chris Bernardi and Eben Ostby for pointing out the problems with Perlin noise and getting us to look in this area in the first place. Loren Carpenter and Michael Kass helped with the Fourier analysis.

References

ABRAMOWITZ, M., AND STEGUN, I. A. 1970. Error function and fresnel integrals. In *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables*. Dover, ch. 7.1.1, 297.

CHUI, C. K. 1992. *An introduction to wavelets*. Academic Press Professional, Inc., San Diego, CA, USA.

FARIN, G. 2002. *Curves and surfaces for CAGD: a practical guide*, 5th ed. Morgan Kaufmann Publishers Inc.

KNUTH, D. E. 1997. *The Art of Computer Programming*, third ed., vol. 2. Addison-Wesley.

LEWIS, J. P. 1989. Algorithms for solid noise synthesis. In *SIGGRAPH '89: Proceedings of the 16th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 263–270.

MALZBENDER, T. 1993. Fourier volume rendering. *ACM Transaction on Graphics 12*, 3 (July), 233–250.

PEACHEY, D. 2003. Building procedural textures. In *Texturing and Modeling: A Procedural Approach*, third ed. Morgan Kaufmann Publishers Inc., ch. 2.

PERLIN, K., AND VELHO, L. 1995. Live paint: painting with procedural multiscale textures. In *SIGGRAPH '95: Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 153–160.

PERLIN, K. 1985. An image synthesizer. In *SIGGRAPH '85: Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 287–296.

PERLIN, K. 2002. Improving noise. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, ACM Press, New York, NY, USA, 681–682.

STOLLNITZ, E., DEROSE, T., AND SALESIN, D. 1996. *Wavelets for Computer Graphics*. Morgan Kaufmann Publishers.

Appendix 1. Noise tile generation

```
/* Note: this code is designed for brevity, not efficiency; many operations can be hoisted,
 * precomputed, or vectorized. Some of the straightforward details, such as tile meshing,
 * decorrelating bands and fading out the last band, are omitted in the interest of space.*/
static float *noiseTileData; static int noiseTileSize;
int Mod(int x, int n) {int m=x%n; return (m<0) ? m+n : m;}
#define ARAD 16
void Downsample (float *from, float *to, int n, int stride ) {
    float *a, aCoeffs[2*ARAD] = {
         0.000334,-0.001528, 0.000410, 0.003545,-0.000938,-0.008233, 0.002172, 0.019120,
        -0.005040,-0.044412, 0.011655, 0.103311,-0.025936,-0.243780, 0.033979, 0.655340,
         0.655340, 0.033979,-0.243780,-0.025936, 0.103311, 0.011655,-0.044412,-0.005040,
         0.019120, 0.002172,-0.008233,-0.000938, 0.003546, 0.000410,-0.001528, 0.000334};
    a = &aCoeffs[ARAD];
    for (int i=0; i<n/2; i++) {
        to[i*stride] = 0;
        for (int k=2*i-ARAD; k<=2*i+ARAD; k++)
            to[i*stride] += a[k-2*i] * from[Mod(k,n)*stride];
    }
}
void Upsample( float *from, float *to, int n, int stride) {
    float *p, pCoeffs[4] = { 0.25, 0.75, 0.75, 0.25 };
    p = &pCoeffs[2];
    for (int i=0; i<n; i++) {
        to[i*stride] = 0;
        for (int k=i/2; k<=i/2+1; k++)
            to[i*stride] += p[i-2*k] * from[Mod(k,n/2)*stride];
    }
}
void GenerateNoiseTile( int n, int olap) {
    if (n%2) n++;                                       /* tile size must be even */
    int ix, iy, iz, i, sz=n*n*n*sizeof(float);
    float *temp1=(float *)malloc(sz),*temp2=(float *)malloc(sz),*noise=(float *)malloc(sz);
    /* Step 1. Fill the tile with random numbers in the range -1 to 1. */
    for (i=0; i<n*n*n; i++) noise[i] = gaussianNoise();
    /* Steps 2 and 3. Downsample and upsample the tile */
    for (iy=0; iy<n; iy++) for (iz=0; iz<n; iz++) {     /* each x row */
         i = iy*n + iz*n*n;  Downsample( &noise[i], &temp1[i], n, 1 );
                             Upsample(   &temp1[i], &temp2[i], n, 1 );
    }
    for (ix=0; ix<n; ix++) for (iz=0; iz<n; iz++) {     /* each y row */
         i = ix   + iz*n*n;  Downsample( &temp2[i], &temp1[i], n, n );
                             Upsample(   &temp1[i], &temp2[i], n, n );
    }
    for (ix=0; ix<n; ix++) for (iy=0; iy<n; iy++) {     /* each z row */
         i = ix   + iy*n;    Downsample( &temp2[i], &temp1[i], n, n*n );
                             Upsample(   &temp1[i], &temp2[i], n, n*n );
    }
    /* Step 4. Subtract out the coarse-scale contribution */
    for (i=0; i<n*n*n; i++) {noise[i]-=temp2[i];}
    /* Avoid even/odd variance difference by adding odd-offset version of noise to itself.*/
    int offset=n/2; if (offset%2==0) offset++;
    for (i=0,ix=0; ix<n; ix++) for (iy=0; iy<n; iy++) for (iz=0; iz<n; iz++)
        temp1[i++] = noise[ Mod(ix+offset,n) + Mod(iy+offset,n)*n + Mod(iz+offset,n)*n*n ];
    for (i=0; i<n*n*n; i++) {noise[i]+=temp1[i];}
    noiseTileData=noise; noiseTileSize=n; free(temp1); free(temp2);
}
```

Appendix 2. Noise evaluation

```
float WNoise( float p[3]) {                         /* Non-projected 3D noise */
    int i, f[3], c[3], mid[3], n=noiseTileSize;     /* f, c = filter, noise coeff indices */
    float w[3][3], t, result =0;
    /* Evaluate quadratic B-spline basis functions */
    for (i=0; i<3; i++) {
        mid[i]=ceil(p[i]-0.5); t=mid[i]-(p[i]-0.5);
        w[i][0]=t*t/2; w[i][2]=(1-t)*(1-t)/2; w[i][1]=1-w[i][0]-w[i][2]; }
    /* Evaluate noise by weighting noise coefficients by basis function values */
    for(f[2]=-1;f[2]<=1;f[2]++) for(f[1]=-1;f[1]<=1;f[1]++) for(f[0]=-1;f[0]<=1;f[0]++) {
         float weight=1;
         for (i=0; i<3; i++) {c[i]=Mod(mid[i]+f[i],n); weight*=w[i][f[i]+1];}
         result += weight * noiseTileData[c[2]*n*n+c[1]*n+c[0]]; }
    return result;
}
float WProjectedNoise( float p[3], float normal[3]) {   /* 3D noise projected onto 2D */
    int i, c[3], min[3], max[3],  n=noiseTileSize;      /* c = noise coeff location */
    float support, result=0;
    /* Bound the support of the basis functions for this projection direction */
    for (i=0; i<3; i++) {
        support = 3*abs(normal[i]) + 3*sqrt((1-normal[i]*normal[i])/2);
        min[i] =  ceil( p[i] - (3*abs(normal[i]) + 3*sqrt((1-normal[i]*normal[i])/2)) );
        max[i] = floor( p[i] + (3*abs(normal[i]) + 3*sqrt((1-normal[i]*normal[i])/2)) ); }
    /* Loop over the noise coefficients within the bound. */
    for(c[2]=min[2];c[2]<=max[2];c[2]++) {
    for(c[1]=min[1];c[1]<=max[1];c[1]++) {
    for(c[0]=min[0];c[0]<=max[0];c[0]++) {
        float t, t1, t2, t3, dot=0, weight=1;
        /* Dot the normal with the vector from c to p */
        for (i=0; i<3; i++) {dot+=normal[i]*(p[i]-c[i]);}
        /* Evaluate the basis function at c moved halfway to p along the normal. */
        for (i=0; i<3; i++) {
            t = (c[i]+normal[i]*dot/2)-(p[i]-1.5); t1=t-1; t2=2-t; t3=3-t;
            weight*=(t<=0||t>=3)? 0 : (t<1) ? t*t/2 : (t<2)? 1-(t1*t1+t2*t2)/2 : t3*t3/2;}
        /* Evaluate noise by weighting noise coefficients by basis function values. */
        result += weight * noiseTileData[Mod(c[2],n)*n*n+Mod(c[1],n)*n+Mod(c[0],n)];
    }}}
    return result;
}
float WMultibandNoise( float p[3],float s,float *normal,int firstBand,int nbands,float *w) {
    float q[3], result=0, variance=0; int i, b;
    for (b=0; b<nbands && s+firstBand+b<0; b++) {
        for (i=0; i<=2; i++) {q[i]=2*p[i]*pow(2,firstBand+b);}
        result += (normal) ? w[b] * WProjectedNoise(q,normal) : w[b] * WNoise(q);}
    for (b=0; b<nbands; b++) {variance+=w[b]*w[b];}
    /* Adjust the noise so it has a variance of 1. */
    if (variance) result /= sqrt(variance * ((normal) ? 0.296 : 0.210));
    return result;
}
```