Bresenham Noise

Masaki Kameya and John C. Hart

School of Electrical Engineering and Computer Science
Washington State University

{mkameya,hart}@eecs.wsu.edu

ABSTRACT

Procedural texture mapping is a powerful technique, and use of the Perlin noise function makes procedural textures appear so realistic and interesting. It is also an expensive technique, and executing a texturing procedure on a per-pixel basis is costly, and usually prevents the method from being used in real time applications. In this paper, we present a method for computing the Perlin noise function for procedural texturing using forward differencing. With this method, we can compute neighboring texture samples with a few additions instead of a number of multiplications thus reducing the computation time to real-time performance.

1. INTRODUCTION

Texture mapping is one of the easiest ways to add realism and complexity to a rendered image. In conventional texture mapping, a two-dimensional image is required, which can be taken from a photograph or a drawing, and is stored in memory. Coordinates of the texture image are associated with points on the object's surface, and the image is then applied to the surface of the object by the implied mapping. The main disadvantage of 2D texture mapping is that it requires large amounts of memory to store images, especially when a large variety of textures are used. Another disadvantage is that if the surface of the object changes dramatically, the correspondences between the texture coordinates and the object coordinates needs to be re-evaluated.

Procedural texturing typically uses solid texture coordinates. In solid texturing, three-dimensional texture coordinates are associated with three dimensional world coordinates of points on the surface. This correspondence is easier and more direct than parameterizing a surface with two-dimensional surface coordinates. When the object is drawn, the texture on the point is computed from the texturing function based on the corresponding texture coordinate. With procedural solid texturing, there is no need to store a texture image. Instead, texture is described by a mathematical function.

To add irregularity to the mathematically described texture, a random value is commonly added. This random value is called noise and it has some properties to make the texture appear more natural.

While procedural texturing is highly flexible, its disadvantage is its speed. Since the texture is computed pixel by pixel when the object is drawn, it requires significant computational power to compute the texture in real-time. Our objective for this research is to overcome this computational cost to realize real-time procedural texture mapping.

We used the parametric procedural texturing described in [3] to generate texture and value noise for the noise function. For the texture to be computed pixel by pixel, we have to evaluate a quadratic function and several noise functions for each pixel. In this paper, we utilize forward differencing to compute the noise function to execute this texturing function in real-time.

Our texturing model is described in Section 2. In Section 3 the noise function we utilized is described. In Section 4 we explain how to use forward differencing for our noise computation and its algorithm is described. Results are presented in Section 5.

2. TEXTURE MODEL

We use the parameterized procedural solid texturing model proposed in [3]. The texturing model maps solid texture coordinate $\mathbf{s} = (s, t, r)$ into a color space $\mathbf{c}$. Texture coordinates are associated with object coordinates by some function $(s, t, r) = S(x, y, z)$ where (x, y, z) is the object's three dimensional coordinate. The simplest example of function S is the identity mapping.

The texture model has the form $p(\mathbf{s}) = c\,(f(\mathbf{s}))$. Function f is a parameterized texturing function, which gives the index to the color space and c is a mapping from this index into a color space. Function f has two components represented as

$$f(\mathbf{s}) = q(\mathbf{s}) + \sum_i a_i n(T_i(\mathbf{s})) \tag{1}$$

where q is a quadric classification function and n is a noise function,

The quadric function controls the basic shape of the texture

$$q(\mathbf{s}) = As^2 + 2Bst + 2Csr + 2Ds + Et^2 + 2Ftr + 2Gt + Hr^2 + 2Ir + J \quad (2)$$

where s, t, r is a texture coordinate, A to J are parameters for controlling textures. By changing the parameter of q, we can make wide variety of textures including ramps, concentric cylinders and concentric spheres.

The noise function adds irregularity to the texture. Function n is a noise function which implements value noise. The parameter a_i controls the amplitude of the noise function, T_i controls the frequency and the phase. In our model, fixed number of noise function is used, typically four.

For example, a marble texture can be described as

$$f(\mathbf{s}) = r + \sum_{i}^{4} 2^{-i} n(2^i s, 2^i t, 2^i r). \quad (3)$$

Another example is wood texture and this is described as

$$f(\mathbf{s}) = s^2 + t^2 + n(4s, 4t, r). \quad (4)$$

More example are described in [3].

3. NOISE FUNCTION

The noise function is used to add irregularity to a procedural texture. The properties of a noise function is important factor to achieve realistic textures. The output of a noise function should be random, but correlated. Noise functions are repeatable pseudorandom functions with a known frequency bandwidth, do not exhibit obvious periodicities, stationary and isotropic [1]. Several noise functions which satisfy these properties have been studied so far [4],[5],[6],[7]. Lattice value noise is the one of the most popular implementation of noise for procedural texturing.

To generate lattice value noise, uniformly distributed random numbers are assigned to integer coordinates in texture space. These points form an integer lattice in texture coordinates. The cube formed by eight adjacent integer lattice points is called a cell. The noise value within the cell is computed from the random values at the eight corner points of the cell by interpolation. This interpolation is usually as smooth as possible. In our case we need to be as efficient as possible, so we use tri-linear interpolation.

Given a point p whose texture coordinate is (s, t, r), noise is computed as follows. First, we find in which cell the point resides. We then compute the random values assigned to the eight vertices of the cell. We then interpolate these eight values to determine the noise value at the given point p.

Let the coordinates s_0, t_0, r_0 be the floor of s, t, r respectively, and likewise s_1, t_1, r_1 be their respective ceiling. Let the eight vertices of the cell be $N_0, N_1, \cdots, N_7$. Their coordinates are $N_0 : (s_0, t_0, r_0), N_1 : (s_1, t_0, r_0), \cdots, N_6 : (s_1, t_1, r_1), N_7 : (s_0, t_1, r_1)$, where $\{s/t/r\}_1 = \{s/t/r\}_0 + 1$. Let noise value at the vertex N_i be $n_i, i = 0, \cdots, 7$. (Figure1)

After obtaining the random values on each of the eight vertices, noise is interpolated in terms of each coordinate. The

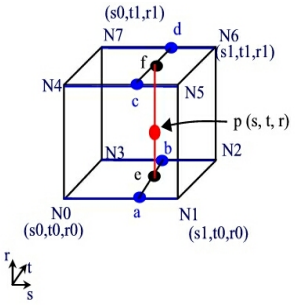


Figure 1: The cell in the texture coordinate.

noise value at point a is linearly interpolated from N_0 and N_1 in terms of s coordinate, i.e.

$$n_a = n_0 + (n_1 - n_0)\frac{s - s_0}{s_1 - s_0} \quad (5)$$

The noise value is computed similarly at points b, c, d. The noise value is computed at point e and f by interpolating n_a and n_b, n_c and n_d in terms of the t coordinate, respectively.

$$n_e = n_a + (n_b - n_a)\frac{t - t_0}{t_1 - t_0} \quad (6)$$

Finally, the noise value at point p is computed from noise value at e and f.

$$noise = n_e + (n_f - n_e)\frac{r - r_0}{r_1 - r_0} \quad (7)$$

Since $s_1 = s_0 + 1, t_1 = t_0 + 1 and r_1 = r_0 + 1$, equation (5),(6),(7) can be rewritten as

$$\begin{aligned} n_a &= n_0 + (n_1 - n_0)(s - s_0) \\ n_e &= n_a + (n_b - n_a)(t - t_0) \\ noise &= n_e + (n_f - n_e)(r - r_0) \end{aligned} \quad (8)$$

$s - s_0$ represents relative position from s_0 and s satisfies $s_0 \le s \le s_1 = s_0 + 1$. Therefore, $(s - s0)$ can be replaced by s_f which represents the fractional part of s and same as t and r. Then equation (8) can be rewritten as:

$$n_a = n_0 + (n_1 - n_0)s_f \quad (9)$$

$$n_e = n_a + (n_b - n_a)t_f \quad (10)$$

$$noise = n_e + (n_f - n_e)r_f \quad (11)$$

where s_f, t_f and r_f represents the fraction part of s, t and r respectively. Incorporating equation (9) and (10) to (11),

$$noise = a + bs_f + ct_f + dr_f + es_f t_f + ft_f r_f + gr_f s_f + hs_f t_f r_f \quad (12)$$

where coefficients a to h are the constants determined by $n_i, i = 0, \cdots, 7$, i.e. the cells which point p resides. This equation shows that trilinear interpolation is a cubic function of the coordinates.

4. TEXTURE COMPUTATION

Scan conversion algorithm draw a polygon from one edge to the another along each scan line. For example, at the scan line j, a polygon may be drawn by scanning from (i_0, j)

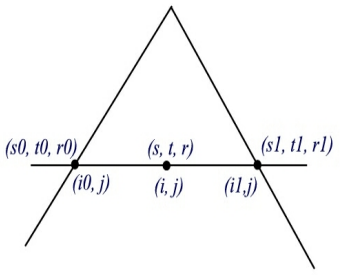


Figure 2: Texture coordinate and scan line

to (i_1, j), as shown in Figure 2. Let the corresponding texture coordinate of these points be (s_0, t_0, r_0), and (s_1, t_1, r_1). (Note these are not the same values as the corner lattice points in the previous section.) The texture coordinates (s, t, r) of each pixel on this scanline of the polygon are linearly interpolated between these two points are linearly interpolated from them.

The derivative of the texture coordinates along the scan line are

$$(ds/di, dt/di, dr/di) = \left(\frac{s_1 - s_0}{i_1 - i_0}, \frac{t_1 - t_0}{i_1 - i_0}, \frac{r_1 - r_0}{i_1 - i_0}\right) \quad (13)$$

The texture coordinates of the point between the edges is now computed as

$$(s, t, r) = (s_0 + \Delta i ds/di, t_0 + \Delta i dt/di, r_0 + \Delta i dr/di) \quad (14)$$

Given the texture coordinates at pixel (i, j), we can compute the color by evaluating the texturing function (1). Generally, this texture is computed by pixel by pixel. That is, the texture is computed every step of scan conversion by computing the texture coordinate and evaluating the texturing equation. If we compute the texture color only on at the enpoints of the scanline then linearly interpolate the resulting color, we will miss the detail of the texture across large polygons.

To make procedural texturing comutation in real-time, we apply the forward differencing technique to the noise function.

4.1 Forward Difference

The computation of successive values of a polynomial can be done efficiently with forward differencing [2]. In computer graphics, this has been used for drawing lines or curves. One of example is line drawing on the screen, such as Bresenham's line or circle drawing algorithms.

We utilize forward differencing to hasten the computation of successive noise values across the scanline, because computation of the noise function is expensive and can be represented as a cubic function of the coordinates for linearly interpolated noise.

The texture coordinates on the scanline are linearly interpolated and the derivative of the texture coordinate along the scan line, $ds/di, dt/di, dr/di$ are computed for each scan line. Suppose the texture coordinate at (i_0, j) is (s_0, t_0, r_0). Then the texture coordinate at $(i_0 + \Delta i, j)$ on the same scan line is computed by $(s_0 + \Delta i ds/di, t_0 + \Delta i dt/di, r_0 + \Delta i dr/di)$.

Incorporating screen coordinate i and the derivatives of texture coordinates $ds/di, dt/di$ and dr/di to equation (12) , noise can be represented as a function of Δi. By replacing (s_0, t_0, r_0) with (s, t, r) and regard i as the Δi,

$$\begin{aligned} noise(\Delta i) &= a + b(s + \Delta i ds/di) \\ &+ c(t + \Delta i dt/di) + d(r + \Delta i dr/di) \\ &+ e(s + \Delta i ds/di)(t + \Delta i dt/di) \\ &+ f(t + \Delta i dt/di)(r + \Delta i dr/di) \\ &+ g(r + \Delta i dr/di)(s + \Delta i ds/di) \\ &+ h(s + \Delta i ds/di)(t + \Delta i dt/di)(r + \Delta i dr/di) \\ &= \alpha i^3 + \beta i^2 + \gamma i + \delta \end{aligned} \quad (15)$$

where α, β, γ and δ are constants determined by constants a to h and s, t and r. Calculation of constans is described in Appendix.

Therefore, the function *noise* is a cubic polynomial in terms of i. Instead of evaluating this function for each pixel position (i, j), we can use forward differencing to avoid a lot of expensive multiplication, which is replaced by small number of setup multiplication and successive additions.

In scan conversion algorithms, the screen coordinate of i is incremented by one for each step. Thus noise can be computed iteratively with forward difference as

$$noise(i + 1) = noise(i) + \Delta n(i) \quad (16)$$

$\Delta n(i)$ can be represented as

$$\begin{aligned} \Delta n(i) &= noise(i + 1) - noise(i) \\ &= \alpha(i + 1)^3 + \beta(i + 1)^2 + \gamma(i + 1) + \delta \\ &\quad -(\alpha i^3 + \beta i^2 + \gamma i + \delta) \\ &= 3\alpha i^2 + i(3\alpha + 2\beta) + \alpha + \beta + \gamma \end{aligned} \quad (17)$$

Since equation (17) contains quadratic term of i, Δn is a function of i. Therefore, we apply forward difference to $\Delta n(i)$.

$$\begin{aligned} \Delta n(i + 1) &= \Delta n(i) + \Delta(\Delta n(i)) \\ &= \Delta n(i) \Delta^2 n(i) \end{aligned} \quad (18)$$

Using similar computation in refeqn:deltan to $\Delta n(i)$, we have

$$\Delta^2 n(i) = 6\alpha i + 6\alpha + 2\beta \quad (19)$$

This still contains i, therefore we apply forward difference once more.

$$\begin{aligned} \Delta^3 n(i) &= \Delta^2 n_(i + 1) - \Delta^2 n(i) \\ &= 6\alpha \end{aligned} \quad (20)$$

Initial values of the forward differences are obtained by plugin $i = 0$. Then we have

$$\begin{aligned} noise_0 &= \delta \\ \Delta n_0 &= \alpha + \beta + \gamma \\ \Delta^2 n_0 &= 6\alpha + 2\beta \\ \Delta^3 n_0 &= = 6\alpha \end{aligned} \quad (21)$$

Then *noise* is iteratively computed as

$$\begin{aligned} noise &+= \Delta n \\ \Delta n &+= \Delta^2 n \\ \Delta^2 n &+= \Delta^3 n \end{aligned} \tag{22}$$

4.2 Noise Computation Algorithm

In this section, the algorithm to compute noise value with forward difference for scan conversion is described. Noise is computed by equation (22) in each step of scan conversion. We have to decide the constant a to h and α, β, γ and δ.

Constants a to h depend on the noise value at the lattice. We have to locate which cell the current point is in. Once it is located and these constants are computed, same constants can be used as long as scan line proceed within the same cell. If scan line reached another cell, these constants have to be re-computed. Constants α, β, γ and δ depend on constants a to h and derivatives of the texture coordinate along the scan line $dsdi, dtdi$ and $drdi$. These derivatives are constants during one scan line and they are updated when it has changed.

In summary, forward differences in equation (22) are re-computed when

- scan line reaches different cell or
- scan line has changed

Pseudo code of noise computation algorithm is shown below.

```
noise_compuation {

    for j = min_edge to max_edge {
       Compute dsdi, dtdi, drdi
       Compute a, b, c, d, e, f, g, h
       Compute alpha, beta, gamma, delta
       Compute n, dn, d2n, d3n

       for i = edge_left to edge_right{

           if ( (i,j) is in the same cell ){
               n   += dn
               dn  += d2n
               d2n += d3n
           } else {
               Compute a, b, c, d, e, f, g, h
               Compute alpha, beta, gamma, delta
               Compute n, dn, d2n, d3n
           }
       }
    }
}
```

With simple example of edge information and texture coordinate, we explain the algorithm. Suppose we want to render square on the screen and corresponding texture coordinate is assigned as in figure 3. In this figure, dashed line shows the integer texture coordinate, i.e. a lattice and let $r = 0$. Suppose now we scan from point A to B. The arrow shows the current scan line. $dsdi, dtdi$ and $drdi$ can be computed from the distance between point A and B and corresponding texture coordinate. Point A resides in the cell whose left bottom texture coordinate is $(0, 0, 0)$. Compute the coefficients from the random value assigned to this cell and compute forward differences. Then plot the pixel along the scan line. At the point P, scan line enter the different cell from the one which point A resides. The left bottom coordinate of the cell is $(1, 0, 0)$. Therefore we have to re-compute the coefficients and the forward difference. After reaching point B, scan line proceed to next line, i.e $j = j+1$. At this time, derivatives of texture coordinate ($dsdi$ etc) might change depend on the texture coordinate assignment. Therefore, these derivatives might be re-computed for every scan line. Then we re-compute the coefficients and forward differences.

By using forward differences, evaluation of polynomial to compute noise at each pixel is reduced to a couple of addition. Foward differences remain same as long as the scan line proceeds the same cell and derivatives of the texture coordinate along the scan line remains same. In other words, if the texture coordinate is very dense, then overhead of computing the coefficients dominate the noise computation. In this case, it is much slower than the way of computing the noise function pixel by pixel.

5. SIMULATION

We measured our method against per-pixel computation. Both techniques were implemented and measured on a Windows NT 4.0 Pentium III 500Mhz PC, using the C programming language. We measure the time to render a 256×256 pixel image of a cube, shown in Figures 4 and 5. We used the following function to synthesize the texture for these images

$$\text{Wood} \quad : \quad f(\mathbf{s}) = 10s^2 + 10r^2 + noise(5s, 5t, r) \tag{23}$$

$$\text{Marble} \quad : \quad f(\mathbf{s}) = r + \sum_{i=0}^{3} noise(2^{-i}s, 2^{-i}t, 2^{-i}r) \tag{24}$$

In both cases, the vertex on the bottom of the cube image

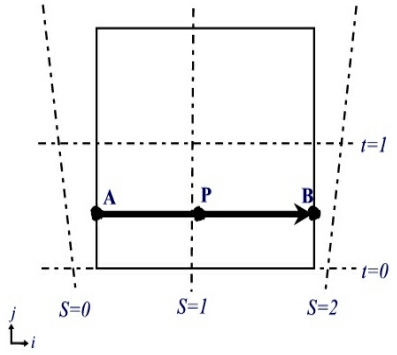


Figure 3: Coefficients are recomputed when the cell has changed.

Table 1: Rendering time comparison

Texture	Pixel by pixel (sec)	Forward diff.(sec)
Wood	0.1322	0.1211
Marble	0.2353	0.2053

was the origin of the texture coordinate system. The top vertex of the image in Figure 4 was assigned $(1, 1, 1)$. Since the wood texture uses a single unscaled octave of noise, only one noise cell is accessed in this image.

In figure 5, the top vertex was assigned the point $(2, 2, 2)$ in texture coordinates. Since the marble texture uses four octaves of noise, this example accesses $16^2 = 256$ cells of noise per displayed face, for a total of 768 noise cells.

We measured the computation time for the per-pixel implementation and our forward differencing version by averaging 10 runs, to avoid anomalies due to the operating system. The results are shown in table 1. These results show that our current prototype implementation of the forward differencing method renders the images about 10% faster than pixel by pixel computation.

To indicate the correctness of the algorithm, the images using both algorithm are shown in figure 6 and 7. Differences of these two images are shown in figure 8. In these figures, white pixels indicate no error, and black pixels indicate there is an error. Actual value is only 1 out of 0 to 255 range.

Also noise value when scan line proceeds 256 pixels in texture coordinate from (0,0,0) to (2,2,2) is shown in figure 9. Figure 10 and 11 shows the difference of noise value for the images 6 and 7 of one scan line. Dashed line shows the edge of the cell. At the first pixel of the each cell, there is no difference at all. However, as scan line proceed the cell, the difference is getting bigger. Since the range of noise value is 0 to 1, the difference is relatively small. Therefore, we can not see the any noticeable differences from rendered images.

These figures show that our algorithm correctly produces images as pixel by pixel noise computation algorithm.

The only error we would expect to see is the accumulation error due to imprecision of the higher derivative.

6. CONCLUSIONS

In this paper, we presented a method to compute the noise function for procedural texturing more efficiently by using forward differencing. This method can compute a noise function faster than per-pixel computation.

Our implementation is limited to trilinearly-interpolated noise cells, which yield a cubic polynomial in the texture coordinates. Most noise implementations use tricubic interpolation of noise cells, which would require a degree nine polynomial and thus nine forward difference variables. It is not yet clear whether such high-degree forward differencing is an effective and efficient choice for noise synthesis.

We limited our presentation to the noise function, but the entire texturing function can also be forward differenced to make texturing computation faster. We suspect forward differencing will play a significant role as the graphics community explores real time implementations of increasingly complex lighting and texturing.

7. ACKNOWLEDGEMENT

This research was supported by the Evans & Sutherland Computer Corporation, and was performed using the facilities of the Image Research Laboratory at Washington State University.

8. REFERENCES

[1] D. Ebert, K. Musgrave, D. Peachey, K. Perlin, and Worley. Texturing and modeling: A procedural approach 2nd edition. 1998.

[2] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. Computer graphics, principles and practice, 2nd edition. 1990. Held in Reading, Massachusetts.

[3] J. C. Hart, N. Carr, M. Kameya, S. A. Tibbitts, and T. J. Coleman. Antialiased parameterized solid texturing simplified for consumer-level hardware implementation. *1999 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 45–53, August 1999.

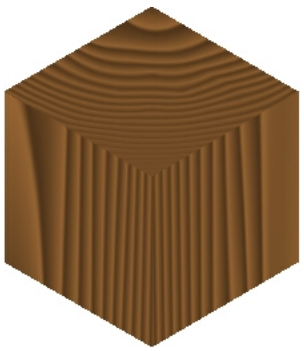


Figure 4: Wood Texture

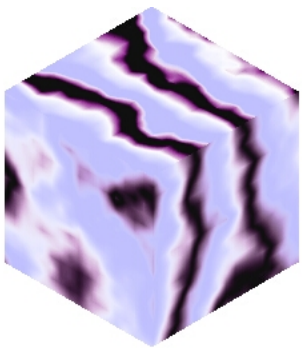
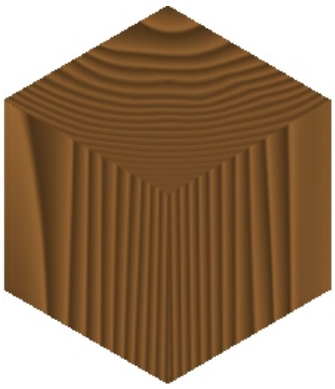
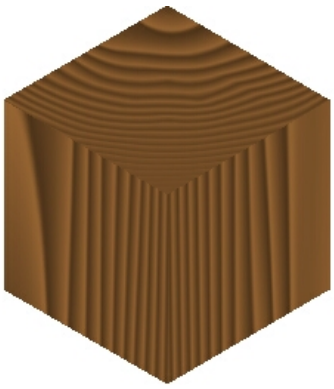


Figure 5: Marble Texture

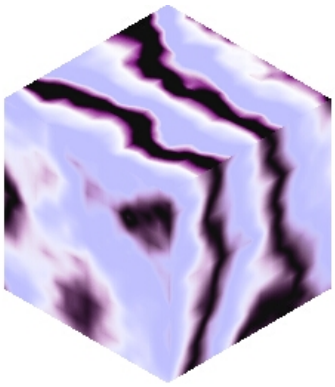


(a) Pixel by pixel

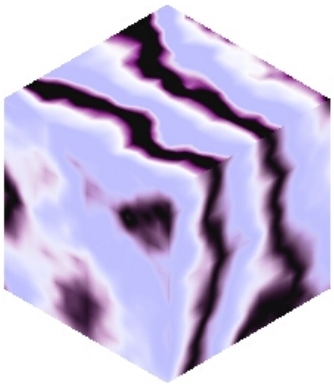


(b) Forward difference

Figure 6: Image comparison : Wood Texture

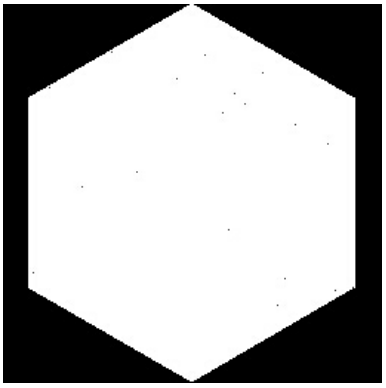


(a) Pixel by pixel

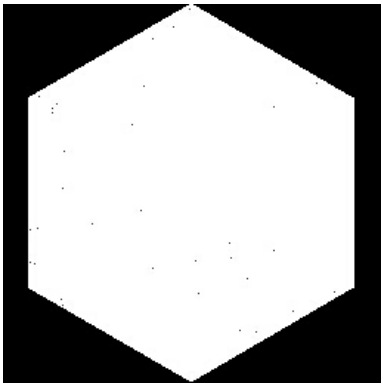


(b) Forward difference

Figure 7: Image comparison : Marble Texture

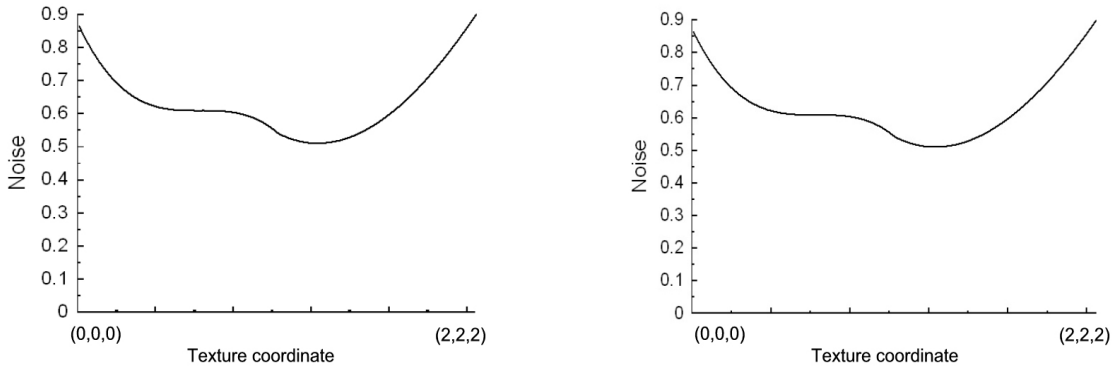


(a) Wood



(b) Marble

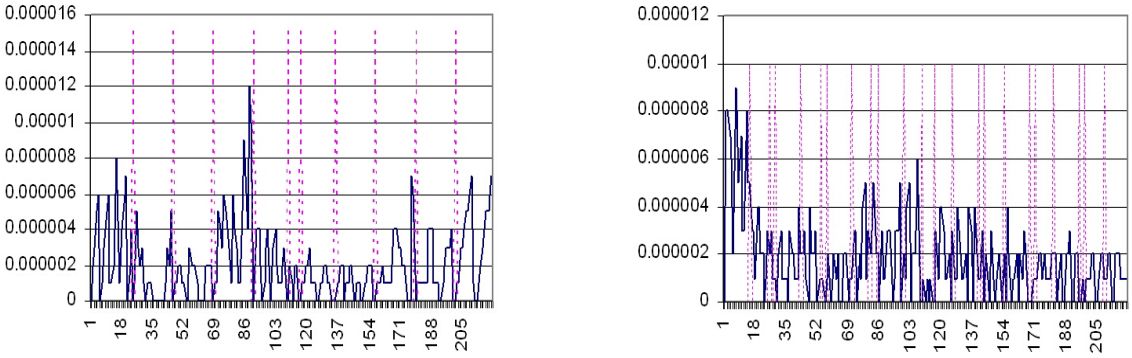
Figure 8: Difference between pixel by pixel and forward diff.



(a) Pixel by pixel

(b) Forward difference

Figure 9: Noise value comparion



(a) Wood

(b) Marble

Figure 10: Noise value difference between pixel by pixel and forward diff.

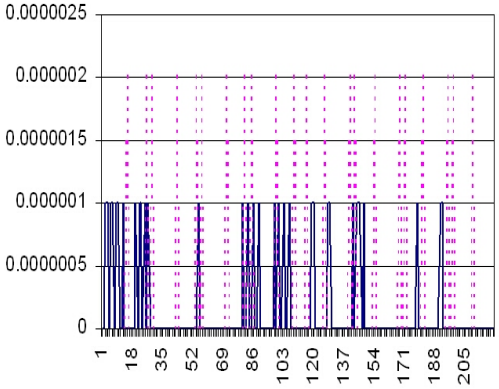


Figure 11: Noise value difference: Marble octave 4 only

[4] J.-P. Lewis. Algorithms for solid noise synthesis. *Computer Graphics (Proceedings of SIGGRAPH 89)*, 23(3):263–270, July 1989.

[5] K. Perlin. An image synthesizer. *Computer Graphics (Proceedings of SIGGRAPH 85)*, 19(3):287–296, July 1985.

[6] J. J. van Wijk. Spot noise-texture synthesis for data visualization. *Computer Graphics (Proceedings of SIGGRAPH 91)*, 25(4):309–318, July 1991.

[7] S. P. Worley. A cellular texture basis function. *Proceedings of SIGGRAPH 96*, pages 291–294, August 1996.

Appendix

The coefficients a to h in equation 12 are as follows:

$$\begin{aligned} a &= n_0 \\ b &= n_1 - n_0 \\ c &= n_3 - n_0 \\ d &= n_4 - n_0 \\ e &= n_0 - n_1 + n_2 - n_3 \\ f &= n_7 - n_4 - n_3 + n_0 \\ g &= n_5 - n_4 - n_1 + n_0 \\ h &= n_4 - n_5 + n_6 - n_7 - e \end{aligned}$$

where $n_i, i = 0, \cdots 7$ are random value at vertices of the cell. The coefficients α to δ can be computed as follows:

$$\begin{aligned} \alpha &= h\frac{ds}{di}\frac{dt}{di}\frac{dr}{di} \\ \beta &= e\frac{ds}{di}\frac{dt}{di} + f\frac{dt}{di}\frac{dr}{di} + g\frac{dr}{di}\frac{ds}{di} \\ &\quad +h(s\frac{dt}{di}\frac{dr}{di} + t\frac{dr}{di}\frac{ds}{di} + r\frac{ds}{di}\frac{dt}{di}) \\ \gamma &= b\frac{ds}{di} + c\frac{dt}{di} + d\frac{dr}{di} + e(s\frac{dt}{di} + t\frac{ds}{di}) + f(t\frac{dr}{di} + r\frac{dt}{di}) \\ &\quad +g(r\frac{ds}{di} + s\frac{dr}{di}) + h(sr\frac{dt}{di} + tr\frac{ds}{di} + st\frac{dr}{di}) \\ \delta &= a + bs + ct + dr + est + ftr + grs + hstr. \end{aligned}$$