

# 9. Noise Hardware

## Introduction

Perlin Noise has been a mainstay of computer graphics since 1985 [EBERT98],[FOLEY96],[PERLIN85], being the core procedure that enables procedural shaders to produce natural appearing materials. Now that real-time graphics hardware has reached the point where memory bandwidth is more of a bottleneck than is raw processing power, it is imperative that the lessons learned from procedural shading be adapted properly to real-time hardware platforms.

The original implementation of Noise is fairly simple. First I will show how it is constructed, and various ways it is used in shader programs to get different kinds of procedural textures. There will be pictures and animations. Then I'll talk about several approaches to real-time implementation, so that Noise can be used most effectively in game hardware and other platforms that should support Noise-intensive imagery at many frames per second.

## What's Noise?

Noise appears random, but isn't really. If it were really random, then you'd get a different result every time you call it. Instead, it's "pseudo-random" - it gives the *appearance* of randomness.

Noise is a mapping from  $\mathbb{R}^n$  to  $\mathbb{R}$  - you input an n-dimensional point with real coordinates, and it returns a real value. Currently the most common uses are for  $n=1$ ,  $n=2$ , and  $n=3$ . The first is used for animation, the second for cheap texture hacks, and the third for less-cheap texture hacks. Noise over  $\mathbb{R}^4$  is also very useful for time-varying solid textures, as I'll show later.

Noise is *band-limited* - almost all of its energy (when looked at as a signal) is concentrated in a small part of the frequency spectrum. High frequencies (visually small details) and low frequencies (large shapes) contribute very little energy. Its appearance is similar to what you'd get if you took a big block of random values and blurred it (ie: convolved with a gaussian kernel). Although that would be quite expensive to compute.

First I'll outline the *ideal* characteristics of a Noise function, then I'll review how the original Noise function matches these characteristics (sort of). After that I'll outline how one might implement the original Noise function in hardware, in an optimized way. Finally, I'll show a radically different approach I've been taking to implementing Noise, with an eye toward standardizing on a visually better primitive that can also run much faster in hardware.

There are two major issues involved in this adaptation: (i) A common procedural shading abstract *machine language*, to enable the capabilities that were first introduced in [Perlin85], and subsequently adapted by the motion picture special effects industry, and (ii) a standard, fast, robust, differentiable and extensible Noise function contained in the instruction set of this abstract machine language. This chapter addresses the second of these two issues.

The *ideal* Noise can be separated from the shortcomings of any particular implementation which aims to approximate this ideal. [Perlin85] outlined a number of characteristics for an ideal Noise. Ideally a hardware-implemented standard would conform to this ideal, without suffering from any shortcomings.

The traditional Noise algorithm, while very useful, had some shortcomings that would be of particular consequence in a real-time setting and in a hardware implementation. I'll describe a new method that suffers from none of these shortcomings. Shortcomings which are addressed include:

- *Lack of a single standard reference implementation:* Unlike the situation to date with software versions of Noise, all implementations should ideally produce the same result for the same input, up to the inherent limitation imposed by limited bit depth, on all platforms and implementations.
- *Requiring many multiplies:* The original formulation of Noise required, as a subset of its component calculations, that a gradient be evaluated at each corner of a cube surrounding the input point. Each gradient evaluation requires an inner product, which costs three multiplies, at each of eight cube vertices, for a total of 24 multiplies. A multiply is expensive in its use of hardware, relative to such other operations as bit manipulation and addition. In a hardware implementation, it would be greatly advantageous to redefine Noise so that it does not require this large number of multiplies.
- *Visually significant anisotropy:* The Noise function is ideally a directionally insensitive (isotropic) signal. However, the original implementation, because it consists of adjoining 3D tricubic patches, contains visible directional artifacts which are an unavoidable consequence of its underlying algorithm. This is also the case for approximations that mimic that implementation, such as nVidia's recent Noise patch [NVIDIA00].

Specifically, when these implementations of Noise are applied to a rotated domain, a casual viewer of the result can easily pick out the orientation of the rotated coordinate grid. Ideally, it should be impossible for a casual viewer to infer the orientation of the rotated coordinate system, when presented with the texture image produced by Noise applied to a rotated domain.

- *Gradient artifacts:* The original Noise function uses a piecewise cubic blending function  $3t^2 - 2t^3$  in each dimension. When the Noise function uses this blending function, then visually noticeable discontinuities appear in the derivative of the Noise function along the 3D tricubic patch boundaries of the Noise function's domain, since the derivative of the derivative of this blending function contains a piecewise constant term.
- *Difficulty of computing a derivative:* The original Noise algorithm contains an associated derivative function which is difficult to compute algorithmically, since it consists of a product of a linear function with three cubic splines. In non-real time applications, the Noise derivative has generally been approximated by evaluating differences of Noise at closely spaced sample points along the three coordinate axes. This has required evaluating the Noise function four times. In a hardware implementation such an approach would not only consume valuable gates, but would be impractical for any method that used less than full floating point precision, since the use of difference methods to compute derivatives requires high bit depth. It is desirable for a hardware Noise standard to possess an associated derivative function that can be computable analytically, at a cost of a relatively modest number of gates. This is particularly important when using Noise to compute normal perturbations and other effects that use the derivative of Noise, as opposed to its value.
- *Need for table memory:* The original Noise algorithm relied on a number of table lookups, which are quite reasonable in a software implementation, but which in a hardware implementation are expensive and constitute a cost bottleneck, particularly when multiple instances of the Noise function are required in parallel. Ideally a Noise implementation should not rely on the presence of tables of significant size.

- *Memory-limited extent of the volume tile:* Noise is generally defined within a repeating volumetric tile. In previous implementations, the extent of this tile has been limited by table size. Ideally Noise should be based on a virtual volume which is scalable, inexpensively, to any extent.
- *Expense of generalizing to higher dimensions:* The original implementation of Noise was based on a cubic lattice. Moving to higher dimensions causes computation cost to more than double with each additional dimension, since it requires moving to an n-dimensional hypercube lattice. In hardware, this cost would be measured as a product of gate count and number of successive instruction cycles. For example, the cost of Noise over four dimensions is at least twice the cost of Noise over three dimensions. Quite soon it will be desirable to extend the standard from 3D Noise to 4D Noise (to account for time-varying volume textures), and thereafter to 5D Noise (to specify textured BRDFs). It is important to address this issue now.
- *Lack of separation between signal and reconstruction:* The original Noise presented the pseudo-random gradient field (its "signal") and its tricubic interpolation scheme (its "reconstruction filter") as a single functional object. It would be greatly advantageous for a method to allow these two operations to be cleanly separable, so that other signals which share the same hardware and abstract machine language can also use this reconstruction filter.

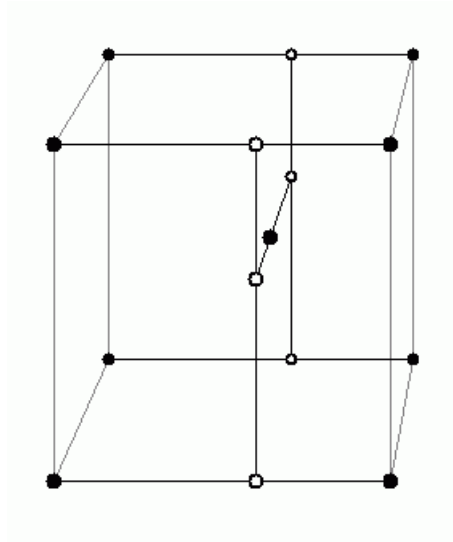
In non-real time applications, in which perfection of the final result is far more important than is processing budget, such as is the case in the use of Noise for motion picture special effects, it is possible to "fudge" some these artifacts by applying Noise multiple times. For example, as I mentioned in my previous chapter, the procedural shaders for the scene depicting ocean waves in the recent film "The Perfect Storm" combined about 200 shader procedures, each of which invoked Perlin Noise. In contrast, for real-time applications, where the cost of every evaluation counts, it is crucial that Noise itself be artifact free, that its derivative be directly computable, and that it incur a relatively small computational cost.

### **The original algorithm:**

To make Noise run fast, I originally implemented it as a pseudo-random spline on a regular cubic grid lattice. Now we'll examine this approach in some detail.

1. Given an input point
2. For each of its neighboring grid points:
  - Pick a "pseudo-random" direction vector
  - Compute linear function (dot product)
3. Linearly combine with a weighted sum, using a cubic ease curve in each dimension, such as  $3t^2-2t^3$ , as the interpolant.

The 8 neighbors in three dimensions:



In three dimensions, there are eight surrounding grid points. To combine their respective influences we use a trilinear interpolation (linear interpolation in each of three dimensions).

In practice this means that once we've computed the  $3t^2-2t^3$  cross-fade function in each of x,y and z, respectively, then we'll need to do seven linear interpolations to get the final result. Each linear interpolation  $a+t(b-a)$  requires one multiply.

In the diagram, the six white dots are the results of the first six interpolations: four in x, followed by two in y. Finally, one interpolation in z gives the final result

To compute the pseudo-random gradient, we can first precompute a table of permutations  $P[n]$ , and a table of gradients  $G[n]$ :

$$G = G[ ( i + P[ ( j + P[k]) \text{ mod } n ] ) \text{ mod } n ]$$

A speedier variation would be to let  $n$  be a power of two. Then, except for bit masking (which is essentially free), the computation reduces to:

$$G = G[ i + P[ j + P[ k ] ] ]$$

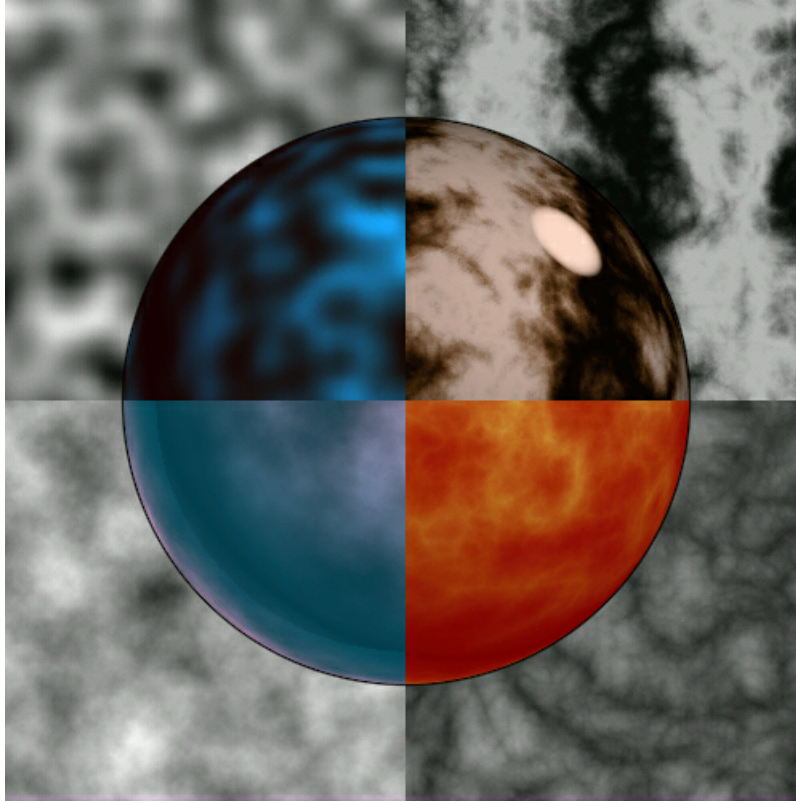
## Using in expressions to get texture

For review, here is a very short set of representative examples to show how Noise can be used to make interesting textures.

- Eg: fractal sums
- $1/f$  noise: rock, mountains, ...
- $1/f \text{ abs}(\text{noise})$ : fire, marble, clouds, ...

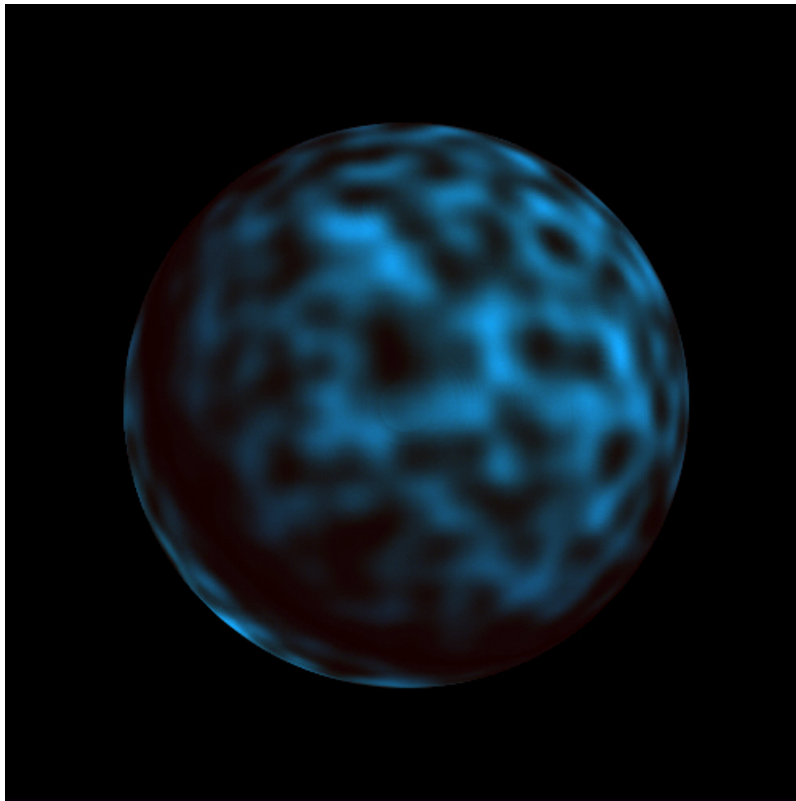
noise

$\sin(x + \sum 1/f(|\text{noise}|))$

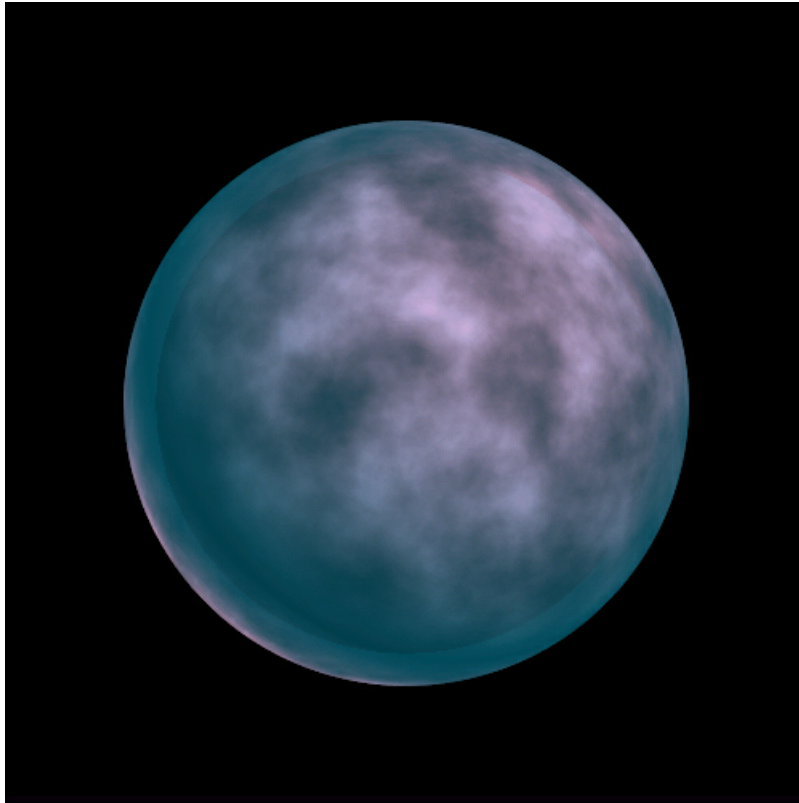


$\sum 1/f(\text{noise})$

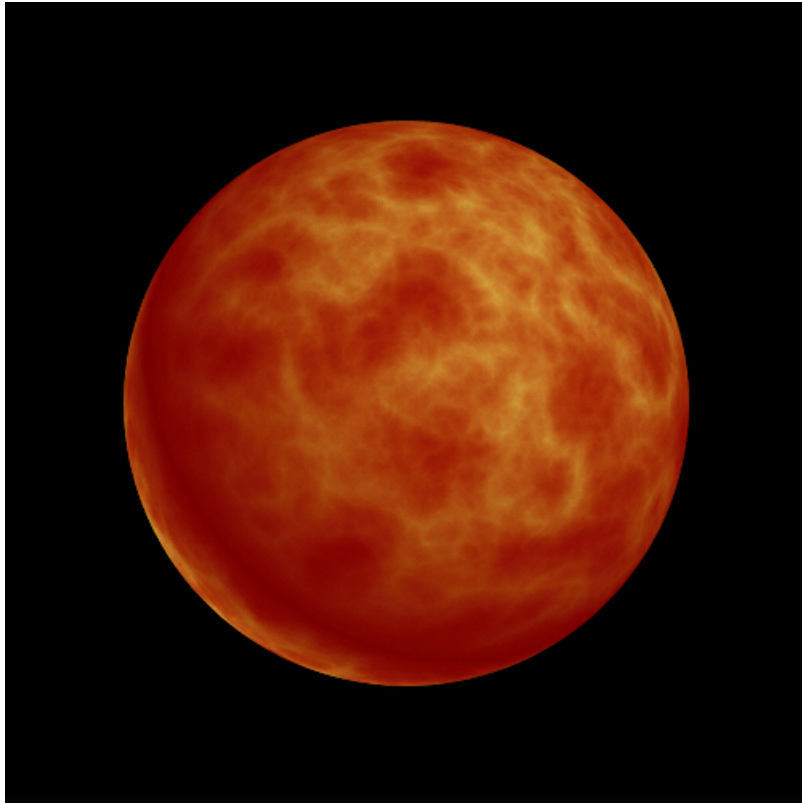
$\sum 1/f(|\text{noise}|)$



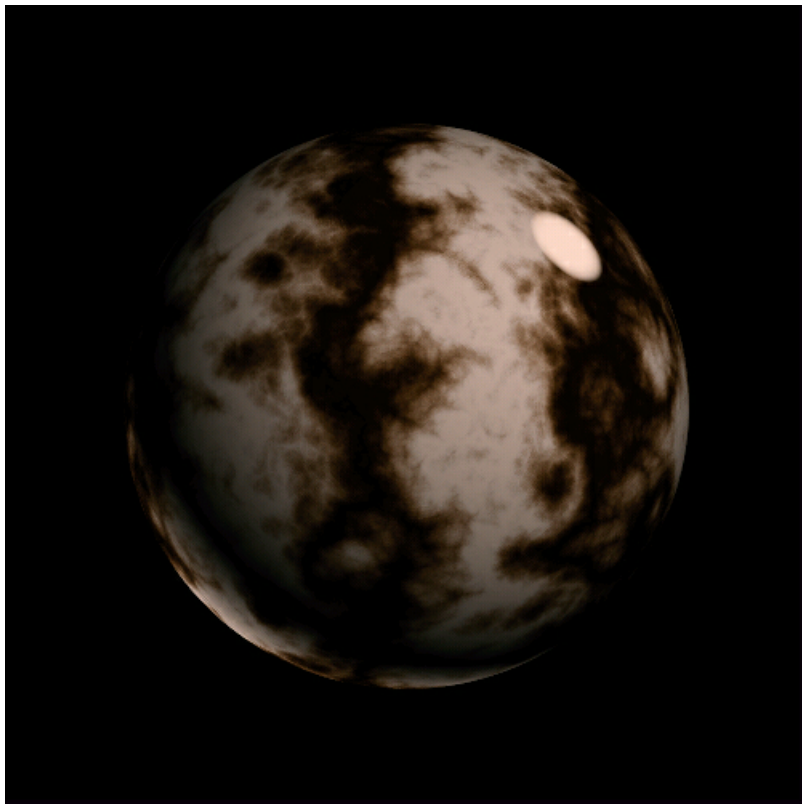
noise



sum  $1/f(\text{noise})$



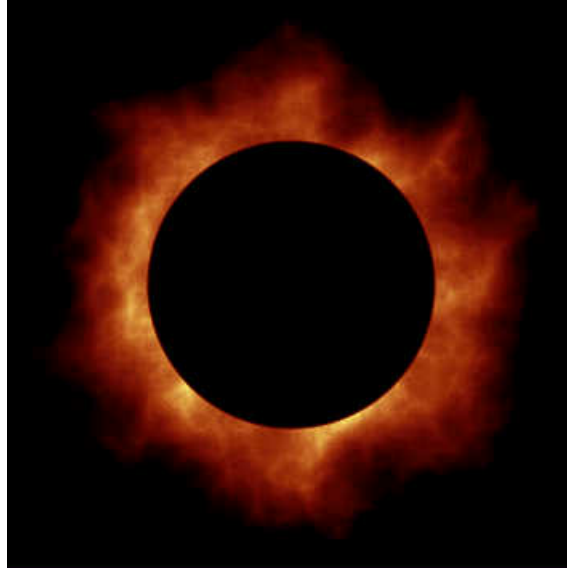
sum  $1/f(|\text{noise}|)$



$$\sin(x + \text{sum } 1/f(|\text{noise}|))$$

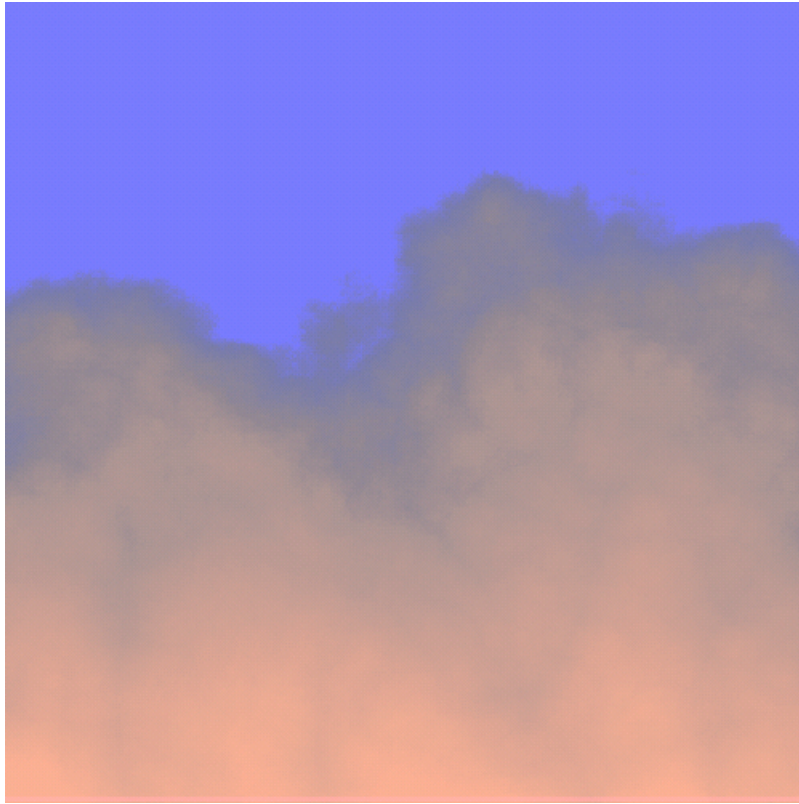
### Using 3D noise to animate 2D turbulent flow

You can also create time-varying animations by using the third dimension of Noise to modulate a texture over time. The following are single frames from pseudo-turbulent animations that were created by dragging, over time, a three dimensional Noise field through the plane  $z=0$ . At each animation frame, the intersection of Noise with the  $(x,y)$  plane was used to create a flow-perturbation texture. The coherently pseudo-random movement of Noise creates time-coherent pseudo-random flame-like or cloud-like appearance of flow.



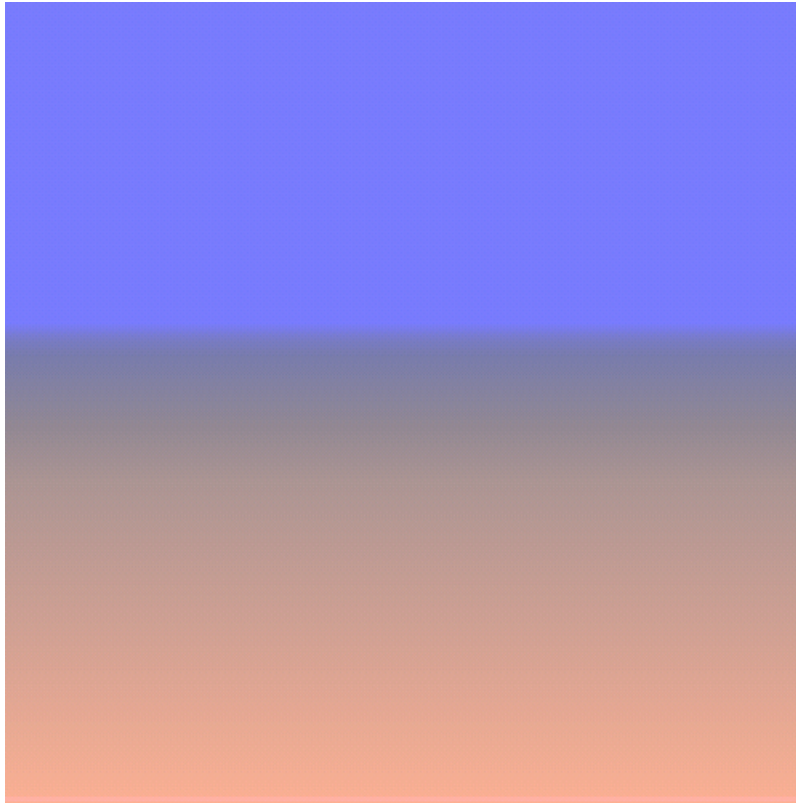
Flame: Noise scales in  $x,y$ , translates in  $z$





### Clouds: Noise translates in x and z

Creating these textures is really quite simple. Below is an image of the gradient field used to create the cloud texture, before perturbation:



Clouds without  $1/f(\text{noise})$  perturbation

## Optimizing for hardware

Recently I've been looking at the question: what would it take to port the Noise function to a direct hardware implementation? There have been some attempts to speed up Noise by relying on non-traditional instruction sets for various computing chips, first by Intel [INTEL96], to capitalize on their MMX instruction set, then more recently by nVidia [NVIDIA00], to make use of the capabilities of their Vertex processing instruction set (although this only helps for triangle vertices, not for texture samples).

But ideally one would like to implement Noise as a primitive operation, directly in hardware, so that it may be invoked as many times as possible, at the texture sample evaluation level, without becoming a computational bottleneck, and perhaps even for volume filling textures [PERLIN89]. What would this require?

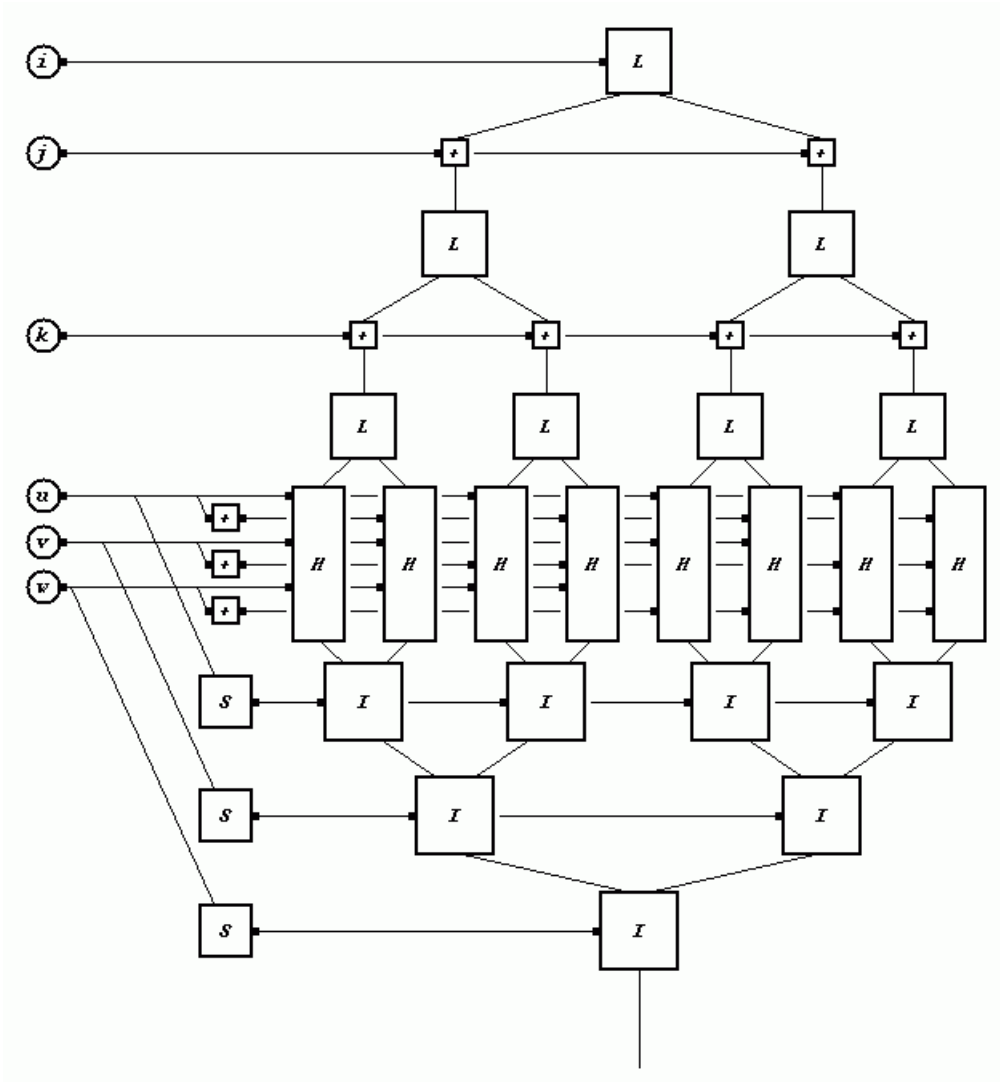
I found that it could be fairly inexpensive, if you worked with a restricted bit depth. For example, if you assume an 8.8 bit input in each of X,Y,Z (ie: each component is in a tiling integer lattice which is 256 units on a side, and there are 256 domain steps within each unit), and an 8 bit output range, then you can implement Noise in under 10K gates, which is a remarkably small number.

The basic approach is to do a pipelined architecture, as in the diagram below. In that diagram:

- $i,j,k$  represent the integer (most significant 8 bits) of the input,
- $u,v,w$  represent the fractional (least significant 8 bits) of the input,
- $L$  represents a computational unit that produces a hash value for one integer component,

- $H$  represents a unit that uses this hash value, together with the three fractional components, to produce a contribution from each of the eight corners of the surrounding unit cube,
- $S$  represents a unit that computes the cubic interpolant function for each dimension,
- $I$  represents a linear interpolator.

The module can be pipelined so that a new Noise evaluation can be calculated at each clock cycle, with a latency of about twenty clock cycles.



## A better method:

Ultimately though, the goals I outlined earlier are not satisfied by my original approach to Noise. So I've been developing a new approach. This method of implementing Noise conforms better to the ideal Noise specification of [Perlin85]. While providing the same general "look" as previous versions of Noise, it also:

- provides a single uniform standard result on any platform,
- is visually isotropic, unlike the original algorithm,
- does not require significant table space to compute good pseudo-random gradients,

- can have an arbitrarily large extent for its repeating virtual tile, at very low cost,
- does not require multiplies to evaluate gradient at surrounding grid vertices,
- does not produce visible grid artifacts,
- does not produce visible artifacts in the derivative,
- is cheaper to compute than is the original algorithm,
- allows for a direct analytic computation of derivative at reasonable cost,
- can be generalized to higher dimensions at relatively small computational expense.

I'll describe the method in two parts:

- A pseudo-random signal generator primitive, and
- A reconstruction primitive.

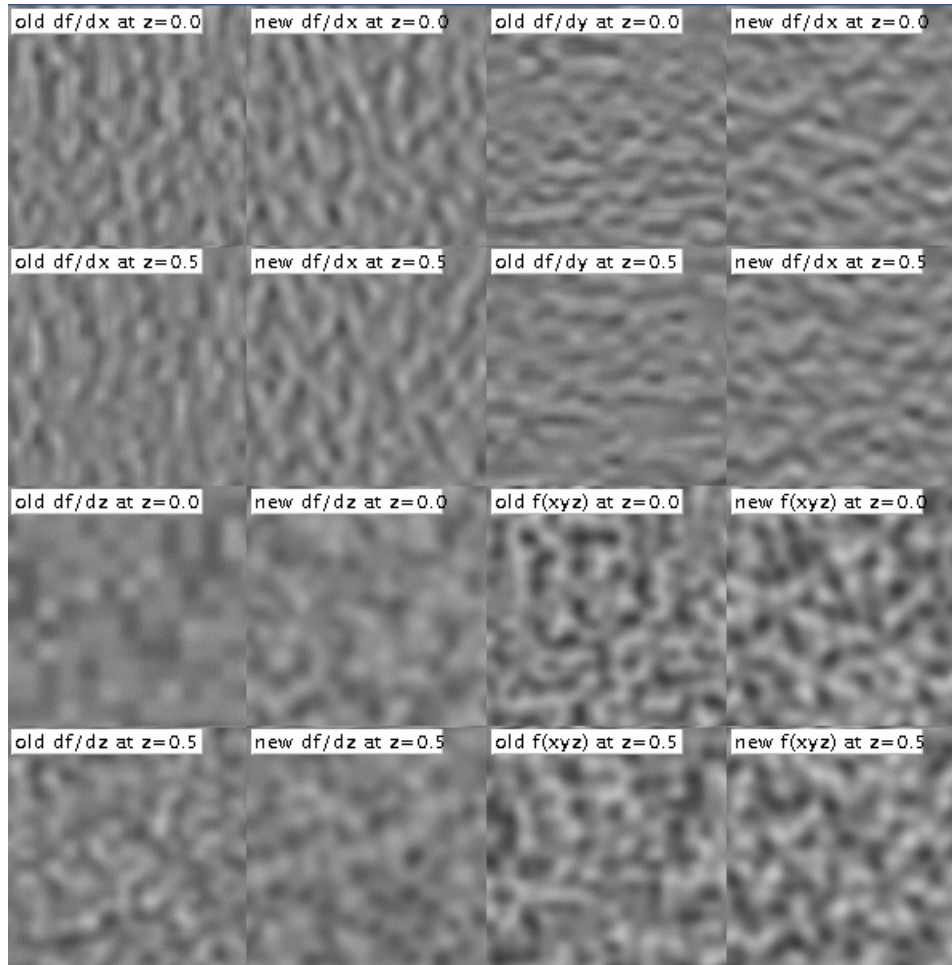
This separation enables other signal generators that reside on the same graphics hardware chip to share the same reconstruction hardware and API.

The image below is a side-by-side visual comparison of "traditional" Noise with the method described here. The four quadrants of the image represent, respectively:

$$\begin{aligned} \mathbf{df/dx} &= \mathbf{d(Noise(xyz))/dx} & \mathbf{df/dy} &= \mathbf{d(Noise(xyz))/dy} \\ \mathbf{df/dz} &= \mathbf{d(Noise(xyz))/dz} & \mathbf{f(xyz)} &= \mathbf{Noise(xyz)} \end{aligned}$$

Each of these quadrants is divided into four sub-quadrants. These represent, respectively:

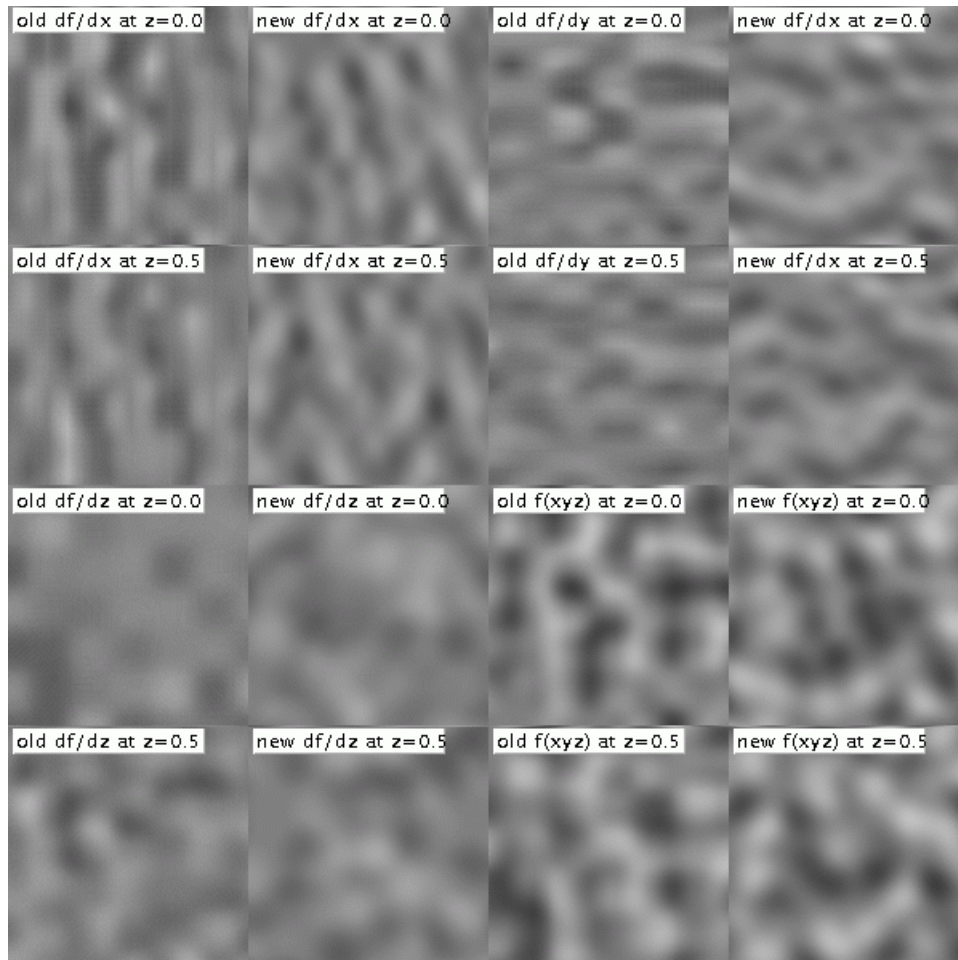
$$\begin{aligned} \mathbf{old\ Noise\ at\ z = 0} & \quad \mathbf{new\ Noise\ at\ z = 0} \\ \mathbf{old\ Noise\ at\ z = 0.5} & \quad \mathbf{new\ Noise\ at\ z = 0.5} \end{aligned}$$



The old and new Noise look roughly the same when evaluated at  $z=0$ , the major visual difference being that the new Noise implementation is visually isotropic. Specifically, if the picture is arbitrarily rotated, it is not possible for an observer examining any subportion of the resulting texture to infer, by visual examination, the original orientation of the image produced by new Noise, whereas it is possible for an observer to infer this orientation for the image produced by the old Noise.

Note also the appearance of the derivative with respect to  $z$ . In the old Noise this degenerates into an image of fuzzy squares.

Below is the same comparison, this time with the domain magnified by a factor of four. Note the artifacts in  $df/dx$  and in  $df/dy$  in old Noise, which appear as thin vertical streaks in  $df/dx$  and as thin horizontal streaks in  $df/dy$ . This is due to the use of the piecewise cubic interpolant function  $3t^2 - 2t^3$ , whose derivative is  $6t - 6t^2$ , which contains a linear term. The presence of this linear term causes the derivative of the Noise function to be discontinuous at cubic cell boundaries.



## Components of the new method:

The new method is a result of combining several different ideas together. When these ideas are used in combination, the result is a far superior Noise function that satisfies all of the requirements outlined above:

1. Rather than using a table lookup scheme to compute the index of a pseudo-random gradient at each surrounding vertex, the new method uses a bit-manipulation scheme that uses only a very small number of hardware gates.
2. Rather than using a cubic interpolation grid, the new method uses a simplicial grid. This confers two advantages during reconstruction:
  1. Only four component evaluations need be done per Noise evaluation (one per contributing vertex), rather than the eight evaluations required in a cubic lattice, and
  2. The axis aligned visual impression of a grid structure is replaced by a far less visually noticeable simplicial packing structure.

Rather than using a tricubic interpolation function, this interpolation scheme uses a spherically symmetric kernel, multiplied by a linear gradient, at each component surrounding vertex. This confers three advantages:

1. The new method contains no directional artifacts due to interpolation function;
2. The new method contains no directional or discontinuity artifacts in gradient;
3. Using the new method, it is practicable to compute the derivative function directly.

Rather than using inner products, with their attendant (and expensive) multiplies, to convert each index into an actual pseudo-random gradient, the new reconstruction method uses a method that produces more visually uniform results, and is easier to integrate into a derivative calculation, while requiring no multiplies at all.

Each of these changes is now described in more detail:

## The new pseudo-random generator primitive:

### Computing index of pseudorandom gradient

Given an integer lattice point  $(i,j,k)$ , the new method uses a bit-manipulation algorithm to generate a six bit quantity. This six bit quantity is then used to generate a gradient direction. The six bit quantity is defined as the lower six bits of the sum:

$$b(i, j, k, 0) + b(j, k, i, 1) + b(k, i, j, 2) + b(i, j, k, 3) + \\ b(j, k, i, 4) + b(k, i, j, 5) + b(i, j, k, 6) + b(j, k, i, 7)$$

where  $b()$  uses its last argument as a bit index into a very small table of bitPatterns.

```
define b(i, j, k, B) :
  patternIndex = 4 * bitB(i) + 2 * bitB(j) + bitB(k)
  return bitPatterns[patternIndex]
```

and where the bit pattern table is defined as:

```
bitPatterns[] = { 0x15, 0x38, 0x32, 0x2c, 0x0d, 0x13, 0x07, 0x2a }
```

### Using index to derive pseudorandom gradient

The new method converts a six bit pseudo-random index into a visually uniform gradient vector which is easy to integrate into a derivative calculation and which requires no multiplies to compute. The key innovation is to use values of only zero or one for the gradient magnitude.

The specific new technique is as follows: The six bit index is split into *(i)* a lower three bit quantity, which is used to compute a magnitude of either zero or one for each of  $x, y$  and  $z$ , and *(ii)* an upper three bit quantity, which is used to determine an octant for the resulting gradient (positive or negative sign in each of  $x, y$ , and  $z$ ).

*(i)* Magnitude computation, based on the three lower bits:

If  $\text{bit}_1 \text{bit}_0 = 0$ , then let  $(p,q,r) = (x,y,z)$ . Otherwise, let  $(p,q,r)$  be a rotation of the order of  $(x,y,z)$  to  $(y,z,x)$  or  $(z,x,y)$ , as  $\text{bit}_1 \text{bit}_0 = 1$  or  $2$ , respectively, and set either  $q$  or  $r$  to zero as  $\text{bit}_2 = 0$  or  $1$ , respectively. The resulting possible rotations are shown in the table below:

<b>bit<sub>2</sub>bit<sub>1</sub>bit<sub>0</sub></b>		<b>bit<sub>2</sub>bit<sub>1</sub>bit<sub>0</sub></b>	
<b>000</b>	p=x q=y r=z	<b>100</b>	p=x q=y r=z
<b>001</b>	p=y q=z r=0	<b>101</b>	p=y q=0 r=x
<b>010</b>	p=z q=x r=0	<b>110</b>	p=z q=0 r=y
<b>011</b>	p=x q=y r=0	<b>111</b>	p=x q=0 r=z

(ii) Octant computation, based on the three upper bits:

Once p,q,r have been determined, invert the sign of p if bit<sub>5</sub>=bit<sub>3</sub>, of q if bit<sub>5</sub>=bit<sub>4</sub>, and of r if bit<sub>5</sub>=(bit<sub>4</sub>!=bit<sub>3</sub>), then add together p,q, and r. The resulting possible gradient values are shown in the table below:

<b>bit<sub>5</sub>bit<sub>4</sub>bit<sub>3</sub></b>		<b>bit<sub>5</sub>bit<sub>4</sub>bit<sub>3</sub></b>	
<b>000</b>	-p-q+r	<b>100</b>	p+q-r
<b>001</b>	p-q-r	<b>101</b>	-p+q+r
<b>010</b>	-p+q-r	<b>110</b>	p-q+r
<b>011</b>	p+q+r	<b>111</b>	-p-q-r

In this way, a gradient vector is defined using only a small number of bit operations and two additions. In particular, the computation of gradient requires no multiply operations. This contrasts very favourably with previous implementations of Noise, in which three multiply operations were required for each gradient computation (one multiply in each of the three component dimensions).

## The new reconstruction primitive:

### Simplex grid:

Rather than placing each input point into a cubic grid, based on the integer parts of its (x,y,z) coordinate values, the input point is placed onto a simplicial grid as follows:

1. Skew the input point (x,y,z) to:

```
define skew((x,y,z) -> (x',y',z')) :
  s = (x+y+z)/3
  (x',y',z') = (x+s,y+s,z+s)
```

This skew transformation linearly scales about the origin, along the x=y=z axis, bringing the point (1,1,1) to the point (2,2,2).

2. Use the integer coordinates in the skewed space to determine a surrounding unit cube whose corner vertex with lowest coordinate values is:

```
(i',j',k') = ( floor(x'), floor(y'), floor(z') )
```

This corner point can be converted back to the original unskewed coordinate system via:

```
define unskew((i',j',k') -> (i,j,k)) :
```



$$s' = (i' + j' + k') / 6$$

$$(i, j, k) = (i - s', j - s', k - s')$$

Also consider the original coordinates relative to the unskewed image of the cube corner:

$$(u, v, w) = (x - i, y - j, z - k)$$

3. Find the simplex containing the point. Relative to (i,j,k), the skewed image of relative point (u,v,w) will lie in one of the six simplices:

$$\begin{aligned} & \{ (0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1) \} \\ & \{ (0, 0, 0), (1, 0, 0), (1, 0, 1), (1, 1, 1) \} \\ & \{ (0, 0, 0), (0, 1, 0), (1, 1, 0), (1, 1, 1) \} \\ & \{ (0, 0, 0), (0, 1, 0), (0, 1, 1), (1, 1, 1) \} \\ & \{ (0, 0, 0), (0, 0, 1), (1, 0, 1), (1, 1, 1) \} \\ & \{ (0, 0, 0), (0, 0, 1), (0, 1, 1), (1, 1, 1) \} \end{aligned}$$

Each of these simplices can be defined as an ordered traversal A,B,C from vertex (0,0,0) to vertex (1,1,1) of a unit cube in the skewed space, where { A,B,C } is some permutation of { (1,0,0),(0,1,0),(0,0,1) }. For example, the last simplex above can be defined as an z,y,x traversal, since its first transition A = (0,0,1), its second transition B = (0,1,0), and its third transition C = (0,0,1).

Which simplex contains the input point is determined by the relative magnitudes of u, v and w. For example, if  $w > v$  and  $v > u$ , then the first transition will be in the z dimension, so A = (0,0,1), and the second transition will be in the y dimension, so B = (0,1,0). In this case, the point lies within the simplex whose traversal order is z,y,x.

The four surrounding vertices of the simplex can now be defined as:

$$\{ (i, j, k), (i, j, k) + \text{unskew}(A), (i, j, k) + \text{unskew}(A+B), (i, j, k) + \text{unskew}(A+B+C) \}$$

## Spherical kernel

If the input point is positioned (u,v,w) from a given simplex vertex, then the contribution from that vertex to the final result will be given by:

$$t = 0.6 - (u^2 + v^2 + w^2)$$

$$\text{if } t > 0 \text{ then } 8(t^4) \text{ else } 0$$

## Hardware integration:

The new method can be implemented as a set of pipelined hardware logic gates, in a way that would be very straightforward to design using an FPGA, given the reference implementation below.

Any implementation needs to choose the number of bits of accuracy desired for both input and for output. This choice of bit-depth will vary the number of hardware gates required, but does not in any significant way modify the underlying technique.

In a pipelined implementation, it is very straightforward to maintain a high performance relative to the number of hardware gates used in the implementation, by pipelining the input. This guarantees that the circuitry which implements each different part of the method is always in active use.

The hardware circuitry that implements the new method can make use of an efficiently pipelined parallel implementation, as follows:

1. A supervisory process or driver fills an array with a sequence of (x,y,z) tuples to be evaluated;
2. Noise is invoked by pipelining these input values into a section of logic circuitry that implements it;
3. The resulting sequence, of Noise derivative/value tuples  $(f_x, f_y, f_z, f)$ , is placed into an output array;
4. The supervisory driver reads out this array of results, and moves on to the next operation within the algorithmic sequence of texture synthesis.

## Generalization to higher dimensions:

It is straightforward to generalize this approach to higher dimensions. In n dimensions, a hypercube can be decomposed into  $n!$  simplices, where each simplex corresponds to an ordering of the edge traversal of the hypercube from its lower vertex  $(0,0,\dots,0)$  to its upper vertex  $(1,1,\dots,1)$ . For example, when  $n=4$ , there are 24 such traversal orderings. To determine which simplex surrounds the input point, one must sort the coordinates in the difference vector  $(u_1,\dots,u_n)$  from the lower vertex of the surrounding skewed hypercube to the input point.

For a given n, the "skew factor" f should be set to  $f = (n+1)^{1/2}$ , so that the point  $(1,1,\dots,1)$  is transformed to the point  $(f,f,\dots,f)$ . In addition, the exact radius and amplitude of the hypersphere-shaped kernel centered at each simplex vertex need to be tuned so as to produce the best visual results for each choice of n.

Previous implementations of Noise, since they were defined on a cubic grid, required a successive doubling of the number of grid points that need to be visited, for each increment in the number of dimensions n. The computational complexity, in terms of vector operations, required to evaluate Noise in n dimensions was therefore  $O(2^n)$ . Since this is exponential in the number of dimensions, it is not practical beyond a few dimensions.

In contrast, the new implementation, since it is defined on a simplicial grid, requires only an increment in the number of grid points that need be visited, for each increment in the number of dimensions n. The computational complexity, in terms of vector operations, required to evaluate the new implementation of Noise in n dimensions is therefore  $O(n)$ . Since this is only polynomial in the number of dimensions, it is practical even for higher dimensions.

To compute the computational complexity in terms of arithmetic operations, both of the above figures need to be multiplied by  $O(n)$ , since the length of each contributing vector operation, and therefore the computational cost of each vector operation, is n, increasing linearly with the number of dimensions. Therefore the computational complexity of previous Noise implementations in n dimensions is  $O(n \cdot 2^n)$ , whereas the computational complexity of the new Noise implementation, in n dimensions is  $O(n^2)$ .

The important conclusion to be drawn from this analysis is that this new implementation of Noise, in contrast to previous implementations, is practical in even high dimensional spaces, because it is a computation of only *polynomial* complexity, not of *exponential* complexity. For example, the cost of computing Noise in 10 dimensions using previous implementations is approximately  $O(10 \cdot 2^{10}) =$

$O(10240)$ , whereas the cost using the new implementation is approximately  $O(10 * 10) = O(100)$ . In this case, a computational advantage factor of 100 is demonstrated. A reference Java implementation of the new Noise algorithm is given below, in Appendix B.

## Appendix A

```

/* coherent noise function over 1, 2 or 3 dimensions */
/* (copyright Ken Perlin) */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define B 0x100
#define BM 0xff

#define N 0x1000
#define NP 12 /* 2^N */
#define NM 0xffff

static p[B + B + 2];
static float g3[B + B + 2][3];
static float g2[B + B + 2][2];
static float g1[B + B + 2];
static start = 1;

static void init(void);

#define s_curve(t) ( t * t * (3. - 2. * t) )

#define lerp(t, a, b) ( a + t * (b - a) )

#define setup(i,b0,b1,r0,r1)\
    t = vec[i] + N;\
    b0 = ((int)t) & BM;\
    b1 = (b0+1) & BM;\
    r0 = t - (int)t;\
    r1 = r0 - 1.;

double noise1(double arg)
{
    int bx0, bx1;
    float rx0, rx1, sx, t, u, v, vec[1];

    vec[0] = arg;
    if (start) {
        start = 0;
        init();
    }

    setup(0, bx0,bx1, rx0,rx1);

    sx = s_curve(rx0);

    u = rx0 * g1[ p[ bx0 ] ];
    v = rx1 * g1[ p[ bx1 ] ];

    return lerp(sx, u, v);
}

float noise2(float vec[2])
{

```

```

int bx0, bx1, by0, by1, b00, b10, b01, b11;
float rx0, rx1, ry0, ry1, *q, sx, sy, a, b, t, u, v;
register i, j;

if (start) {
    start = 0;
    init();
}

setup(0, bx0,bx1, rx0,rx1);
setup(1, by0,by1, ry0,ry1);

i = p[ bx0 ];
j = p[ bx1 ];

b00 = p[ i + by0 ];
b10 = p[ j + by0 ];
b01 = p[ i + by1 ];
b11 = p[ j + by1 ];

sx = s_curve(rx0);
sy = s_curve(ry0);

#define at2(rx,ry) ( rx * q[0] + ry * q[1] )

q = g2[ b00 ] ; u = at2(rx0,ry0);
q = g2[ b10 ] ; v = at2(rx1,ry0);
a = lerp(sx, u, v);

q = g2[ b01 ] ; u = at2(rx0,ry1);
q = g2[ b11 ] ; v = at2(rx1,ry1);
b = lerp(sx, u, v);

return lerp(sy, a, b);
}

float noise3(float vec[3])
{
    int bx0, bx1, by0, by1, bz0, bz1, b00, b10, b01, b11;
    float rx0, rx1, ry0, ry1, rz0, rz1, *q, sy, sz, a, b, c, d, t, u, v;
    register i, j;

    if (start) {
        start = 0;
        init();
    }

    setup(0, bx0,bx1, rx0,rx1);
    setup(1, by0,by1, ry0,ry1);
    setup(2, bz0,bz1, rz0,rz1);

    i = p[ bx0 ];
    j = p[ bx1 ];

    b00 = p[ i + by0 ];
    b10 = p[ j + by0 ];
    b01 = p[ i + by1 ];
    b11 = p[ j + by1 ];

    t = s_curve(rx0);
    sy = s_curve(ry0);
    sz = s_curve(rz0);

#define at3(rx,ry,rz) ( rx * q[0] + ry * q[1] + rz * q[2] )

```

```

q = g3[ b00 + bz0 ] ; u = at3(rx0,ry0,rz0);
q = g3[ b10 + bz0 ] ; v = at3(rx1,ry0,rz0);
a = lerp(t, u, v);

q = g3[ b01 + bz0 ] ; u = at3(rx0,ry1,rz0);
q = g3[ b11 + bz0 ] ; v = at3(rx1,ry1,rz0);
b = lerp(t, u, v);

c = lerp(sy, a, b);

q = g3[ b00 + bz1 ] ; u = at3(rx0,ry0,rz1);
q = g3[ b10 + bz1 ] ; v = at3(rx1,ry0,rz1);
a = lerp(t, u, v);

q = g3[ b01 + bz1 ] ; u = at3(rx0,ry1,rz1);
q = g3[ b11 + bz1 ] ; v = at3(rx1,ry1,rz1);
b = lerp(t, u, v);

d = lerp(sy, a, b);

return lerp(sz, c, d);
}

static void normalize2(float v[2])
{
    float s;

    s = sqrt(v[0] * v[0] + v[1] * v[1]);
    v[0] = v[0] / s;
    v[1] = v[1] / s;
}

static void normalize3(float v[3])
{
    float s;

    s = sqrt(v[0] * v[0] + v[1] * v[1] + v[2] * v[2]);
    v[0] = v[0] / s;
    v[1] = v[1] / s;
    v[2] = v[2] / s;
}

static void init(void)
{
    int i, j, k;

    for (i = 0 ; i < B ; i++) {
        p[i] = i;

        g1[i] = (float)((random() % (B + B)) - B) / B;

        for (j = 0 ; j < 2 ; j++)
            g2[i][j] = (float)((random() % (B + B)) - B) / B;
        normalize2(g2[i]);

        for (j = 0 ; j < 3 ; j++)
            g3[i][j] = (float)((random() % (B + B)) - B) / B;
        normalize3(g3[i]);
    }

    while (--i) {
        k = p[i];
        p[i] = p[j = random() % B];
        p[j] = k;
    }
}

```

```

    for (i = 0 ; i < B + 2 ; i++) {
        p[B + i] = p[i];
        g1[B + i] = g1[i];
        for (j = 0 ; j < 2 ; j++)
            g2[B + i][j] = g2[i][j];
        for (j = 0 ; j < 3 ; j++)
            g3[B + i][j] = g3[i][j];
    }
}

```

## Appendix B:

A complete implementation of a function returning a value that conforms to the new method is given below as a Java class definition:

```

public final class Noise3 {
    static int i,j,k, A[] = {0,0,0};
    static double u,v,w;
    static double noise(double x, double y, double z) {
        double s = (x+y+z)/3;
        i=(int)Math.floor(x+s); j=(int)Math.floor(y+s); k=(int)Math.floor(z+s);
        s = (i+j+k)/6.; u = x-i+s; v = y-j+s; w = z-k+s;
        A[0]=A[1]=A[2]=0;
        int hi = u>=w ? u>=v ? 0 : 1 : v>=w ? 1 : 2;
        int lo = u< w ? u< v ? 0 : 1 : v< w ? 1 : 2;
        return K(hi) + K(3-hi-lo) + K(lo) + K(0);
    }
    static double K(int a) {
        double s = (A[0]+A[1]+A[2])/6.;
        double x = u-A[0]+s, y = v-A[1]+s, z = w-A[2]+s, t = .6-x*x-y*y-z*z;
        int h = shuffle(i+A[0],j+A[1],k+A[2]);
        A[a]++;
        if (t < 0)
            return 0;
        int b5 = h>>5 & 1, b4 = h>>4 & 1, b3 = h>>3 & 1, b2= h>>2 & 1, b = h & 3;
        double p = b==1?x:b==2?y:z, q = b==1?y:b==2?z:x, r = b==1?z:b==2?x:y;
        p = (b5==b3 ? -p : p); q = (b5==b4 ? -q : q); r = (b5!=(b4^b3) ? -r : r);
        t *= t;
        return 8 * t * t * (p + (b==0 ? q+r : b2==0 ? q : r));
    }
    static int shuffle(int i, int j, int k) {
        return b(i,j,k,0) + b(j,k,i,1) + b(k,i,j,2) + b(i,j,k,3) +
            b(j,k,i,4) + b(k,i,j,5) + b(i,j,k,6) + b(j,k,i,7) ;
    }
    static int b(int i, int j, int k, int B) { return T[b(i,B)<<2 | b(j,B)<<1 | b(k,B)
    static int b(int N, int B) { return N>>B & 1; }
    static int T[] = {0x15,0x38,0x32,0x2c,0x0d,0x13,0x07,0x2a};
}

```

---

## References:

[EBERT98] *Texturing and Modeling; A Procedural Approach*, Second Edition; Ebert D. et al, AP Professional; Cambridge 1998c;

[FOLEY96] *Computer Graphics: Principles and Practice*, C version, Foley J., et al, ADDISON-WESLEYD 1996,

[INTEL96], *Using MMX[tm] Instructions for Procedural Texture Mapping* Intel Developer Relations Group, Version 1.0, November 18, 1996, <http://developer.intel.com/drg/mmx/appnotes/proctex.htm>

[NVIDIA00] *Technical Demos - Perlin Noise* [http://www.nvidia.com/Support/DeveloperRelations/Technical Demos](http://www.nvidia.com/Support/DeveloperRelations/TechnicalDemos), Disclosed 11/10/2000.

[PERLIN89] Perlin, K., and Hoffert, E., *Hypertexture*, 1989 Computer Graphics (proceedings of ACM SIGGRAPH Conference); Vol. 23 No. 3.

[PERLIN85] Perlin, K., *An Image Synthesizer*, Computer Graphics; Vol. 19 No. 3.

