Perlin Noise Function

Purposes of noise functions

- Mimic Natural Phenomena
- Add complexity to a texture to make it less boring

Useful Functions

linear interpolate from a to b by a factor of t

```
float lerp( float t, float a, float b )
{
	return a + t*( b - a );
}
```

an s shaped blending function equal to 3t^2 - 2t^3

```
float sCurve( float t )
{
	return t*t*(3-2*t);
}
```

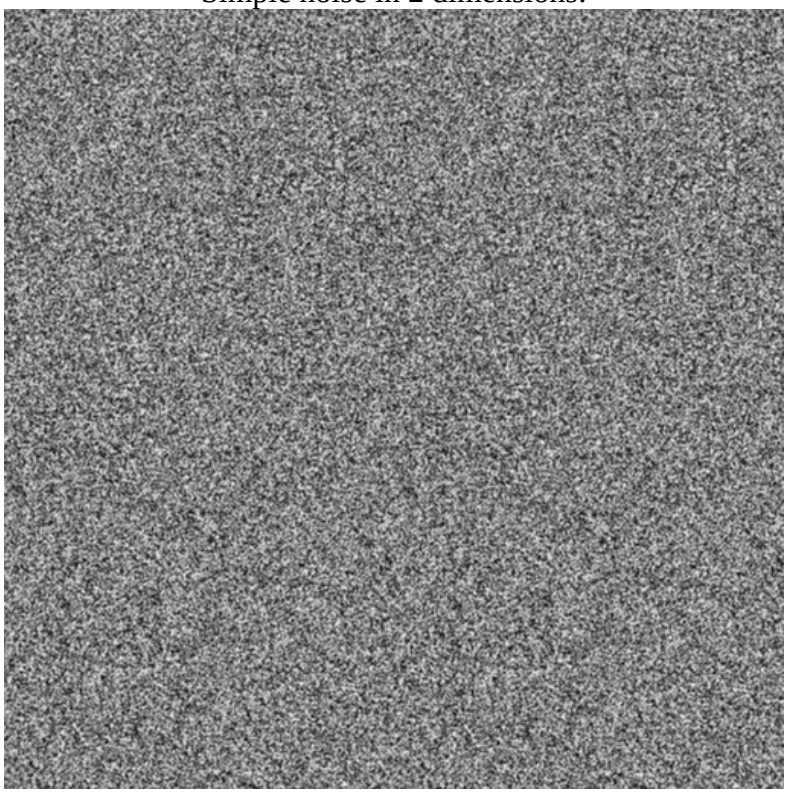
Simple Noise Function

These simple noise functions have some desirable properties.

```
float simpleNoise1D( int x )
{
	x = (x<<13) ^ x;
	return ( 1.0 - ( (x * (x * x * 15731 + 789221) + 1376312589) &
0x7fffffff) / 1073741824.0);
}
```

```
float simpleNoise2D( int x, int y )
{
	int n = x + y * 257;
	n = (n<<13) ^ n;
	return ( 1.0 - ( (n * (n * n * 15731 + 789221) + 1376312589) &
0x7fffffff) / 1073741824.0);
}
```

Simple noise in 2 dimensions:



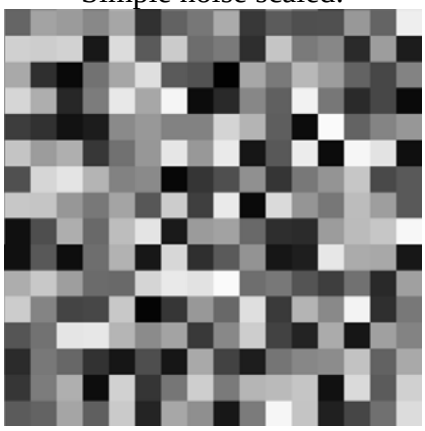
Desirable properties of simple noise

- Fast
- Produces the same output for any given input (consistent). Can be used to generate textures on the fly.

Problems with simple noise

- Not countinuous when scaled (incoherent).

Simple noise scaled:



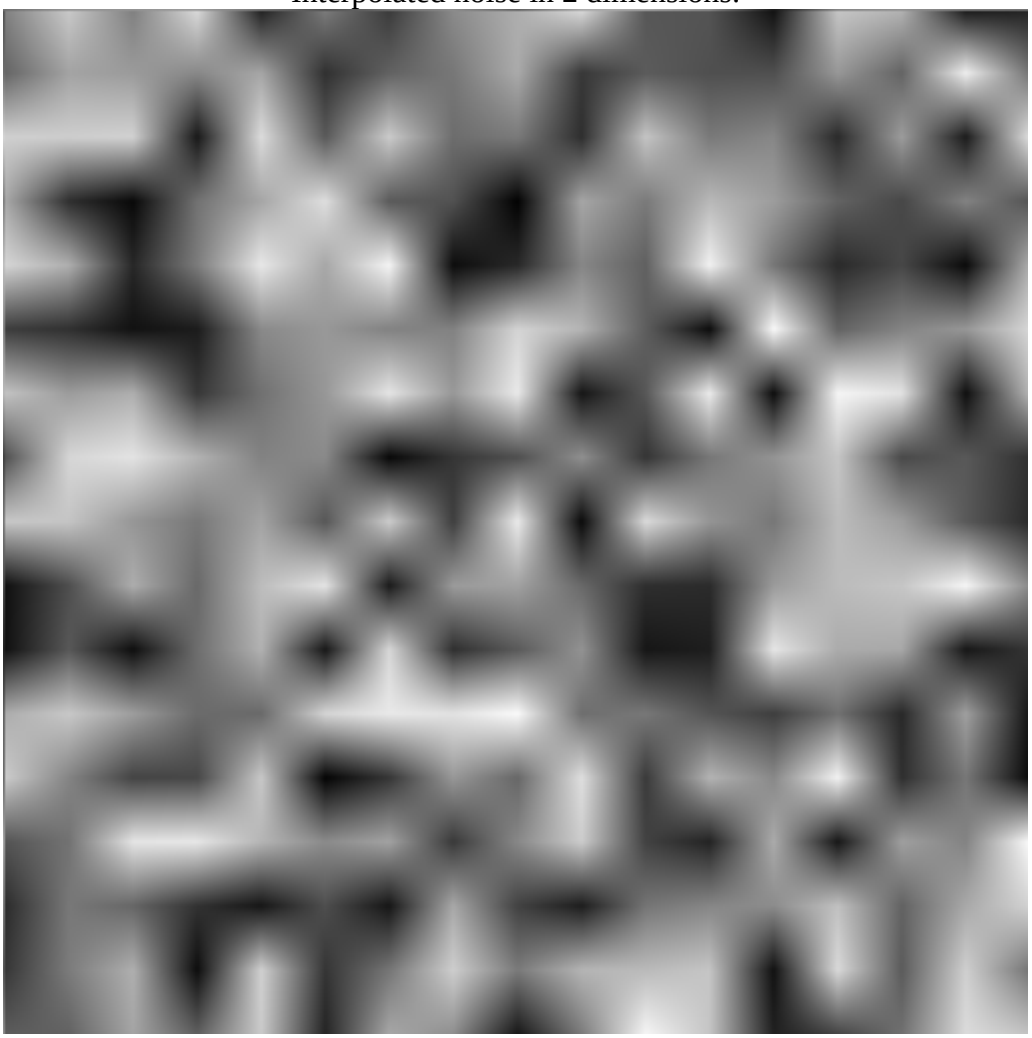
Interpolated Noise Function

These noise functions use linear interpolation to make the noise coherent.

```
float lerpNoise1D( float x )
{
        int xi;     //integer component
        float xf;     //floating point component
        float l0, l1; //uninterpolated values

        xi = (int)x;
        xf = x - (float)xi;

        l0 = simpleNoise1D( xi );
        l1 = simpleNoise1D( xi+1 );

        return lerp( xf, l0, l1 );  //interpolate the noise values
}
```

```
float lerpNoise2D( float x, float y )
{
        int xi,yi;              //integer component
        float xf,yf;              //floating point component
        float l00, l01, l10, l11; //uninterpolated values
        float l0, l1;             //partially interpolated values

        xi = (int)x;
        xf = x - (float)xi;
        yi = (int)y;
        yf = y - (float)yi;

        l00 = simpleNoise2D( xi  , yi   );
        l01 = simpleNoise2D( xi+1, yi   );
        l10 = simpleNoise2D( xi  , yi+1 );
        l11 = simpleNoise2D( xi+1, yi+1 );

        l0 = lerp( xf, l00, l01 );  //interpolate across the x axis
        l1 = lerp( xf, l10, l11 );

        return lerp( yf, l0, l1 );  //interpolate across the y axis
}
```

Interpolated noise in 2 dimensions:



Desirable properties of interpolated noise

- Continuous when zoomed in (coherent)

Problems with interpolated noise

- Still looks unnatural due to ridges at regular intervals.

S-curve Interpolated Noise Function

These noise functions use s-curve weighted interpolation to make the noise coherent.

```
float smoothNoise1D( float x )
{
        int xi;     //integer component
        float xf;     //floating point component
        float xs;     //s-curve function result
        float l0, l1; //uninterpolated values

        xi = (int)x;
        xf = x - (float)xi;
        xs = sCurve(xf);

        l0 = simpleNoise1D( xi );
        l1 = simpleNoise1D( xi+1 );

        return lerp( xf, l0, xs );  //interpolate the noise values
useing the s-curve value
}
```

```
float smoothNoise2D( float x, float y )
{
        int xi,yi;                //integer component
        float xf,yf;              //floating point component
        float xs,ys;              //s-curve function results
        float l00, l01, l10, l11; //uninterpolated values
        float l0, l1;             //partially interpolated values

        xi = (int)x;
        xf = x - (float)xi;
        xs = sCurve(xf);
        yi = (int)y;
        yf = y - (float)yi;
        ys = sCurve(yf);

        l00 = simpleNoise2D( xi  , yi   );
        l01 = simpleNoise2D( xi+1, yi   );
        l10 = simpleNoise2D( xi  , yi+1 );
        l11 = simpleNoise2D( xi+1, yi+1 );

        l0 = lerp( xs, l00, l01 );  //interpolate across the x axis
using the s curve
        l1 = lerp( xs, l10, l11 );

        return lerp( ys, l0, l1 );  //interpolate across the y axis
using the s curve
}
```

S-curve interpolated noise in 2 dimensions:

Comments on s-curve interpolated noise

- This results in a smooth curve, but the results still look artificial.
- Peaks and valleys will always occur at regular intervals
- Humans seem to be good at detecting peaks and valleys in a pattern

The Perlin noise function.

Perlin Noise takes a different approach to natural looking noise. Instead of defining the value of the noise function at regular intervals, the slope of the noise function is defined at regular intervals. This leads to peaks/valleys forming at irregular intervals.

Perlin noise in one dimension:

At each integer location, a linear weight function is defined. For non-integers values, the results weight functions are interpolated and weighted with the scurve.

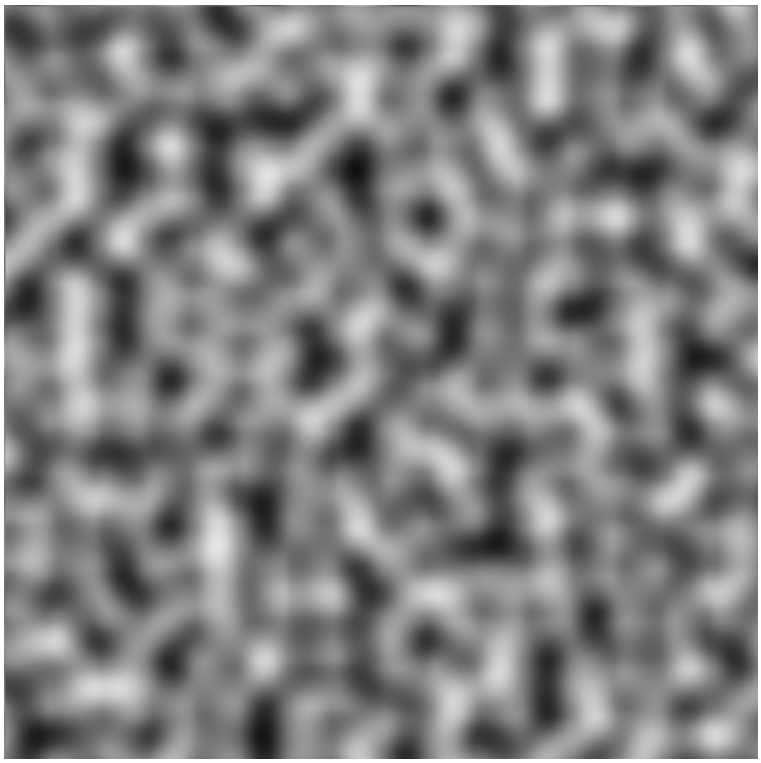
Perlin noise in higher dimensions:

At the grid points (integer valued coordinates), the slope of the noise function is defined a an n-dimensional vector where n is the dimensionality of the noise function. This slope vector is more commonly referred to as the gradient vector, and is randomly generated for each grid point. The length is usually normalized to 1. The sample vector is the vector extending from the grid point to the location being sampled. The weight function is simply the dot product of the gradient vector and the sample vector. The weights are interpolated in n dimensions using the s curve, just as in one dimension.

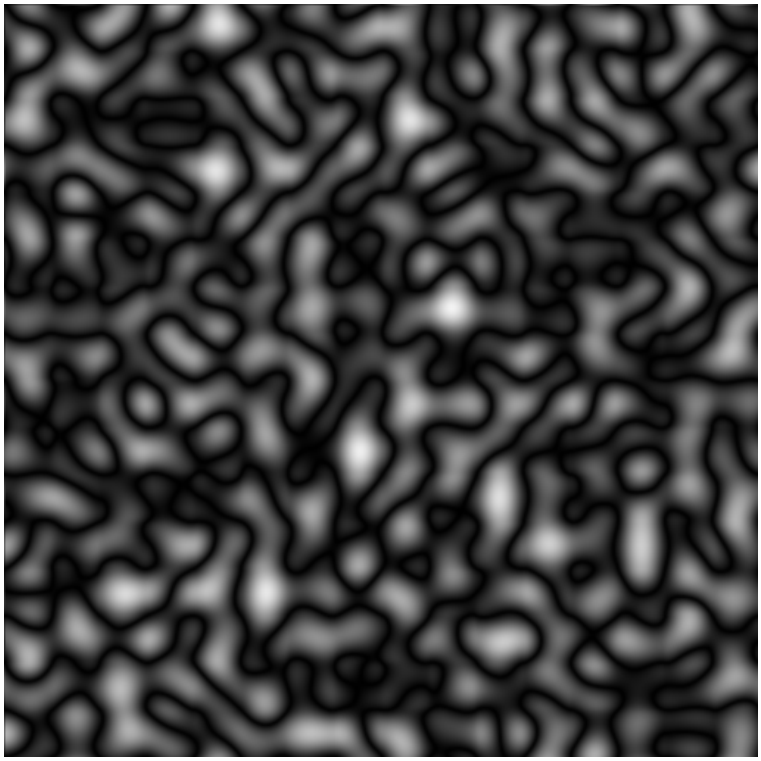
Psuedocode for Perlin noise

```
float perlinNoise( vector sampleVec )
begin
        foreach integer grid vector h surrounding sampleVec:
        begin
                g = psuedorandom gradient vector generated from h
                v = sampleVec - g;
                w[h] = g.v;
        end

        return s-curve weighted interpolation of all w[] values
end
```

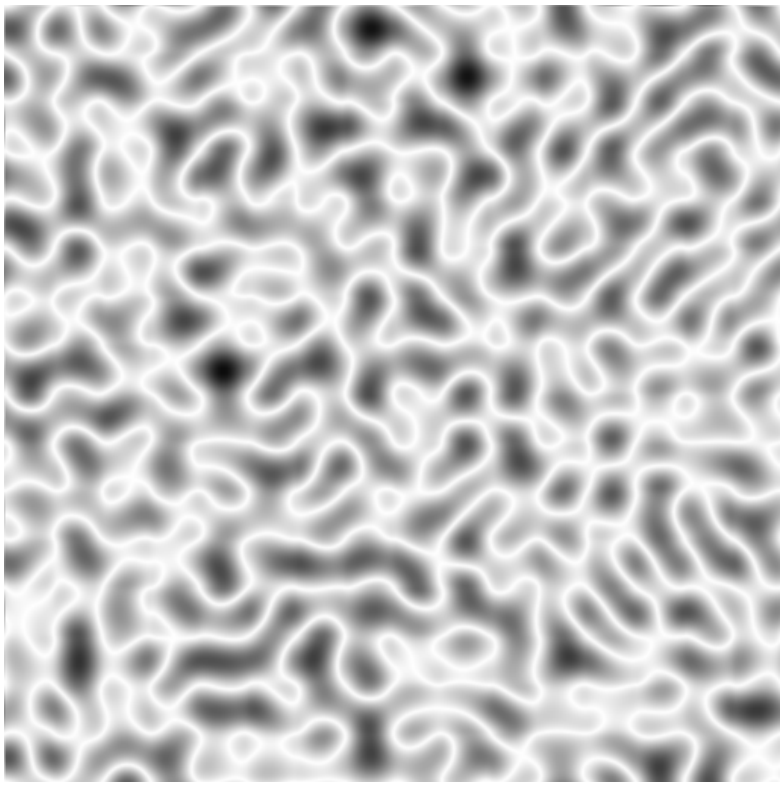
Perlin noise in 2 dimensions:



Absolute value of the Perlin noise function:



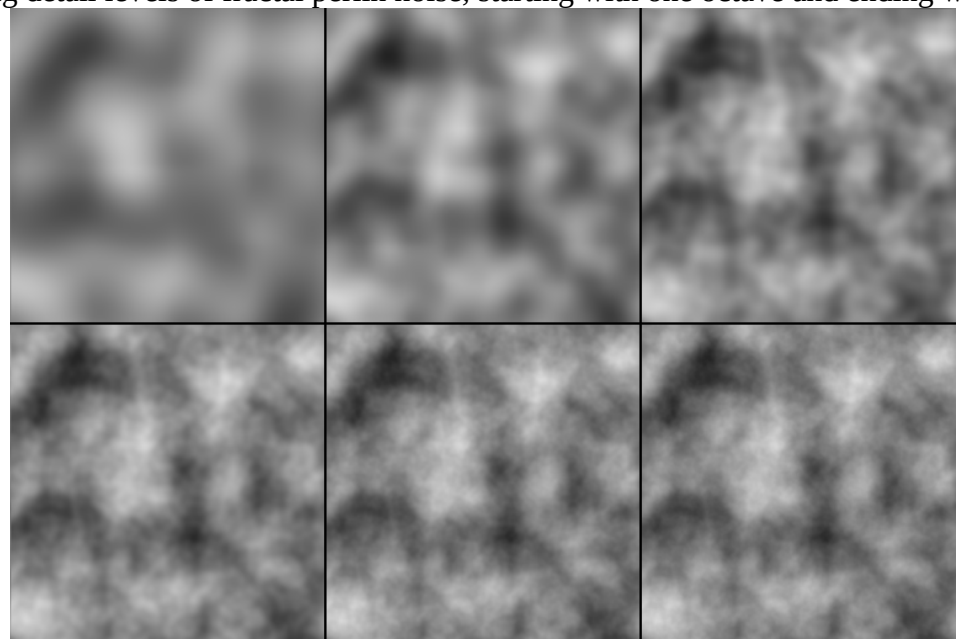
Inverse of the absolute value of the Perlin noise function:



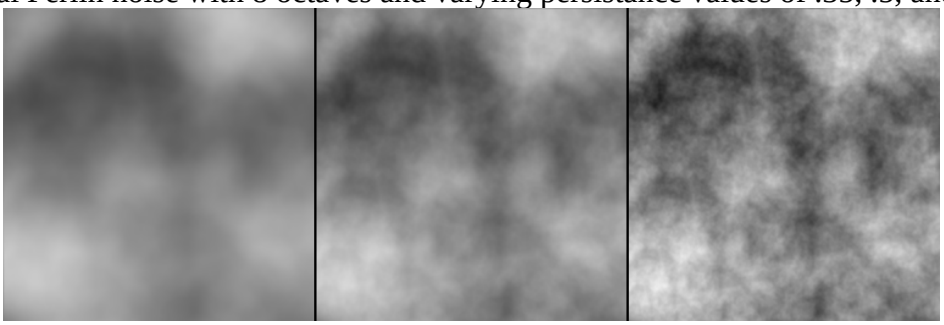
Fractal noise

Simple Perlin noise looks nice, and can simulate some natural phenomena such as water surfaces and smooth, rolling hills. Fractal noise adds more complexity, and produces results that are more visually appealing. Fractal noise is generated by summing together several levels of a noise function. These levels are often called octaves. The frequency of each octave is twice that of the previous octave. The amplitude of each octave is some multiple p of the previous octave. The multiple p is known as the persistence.

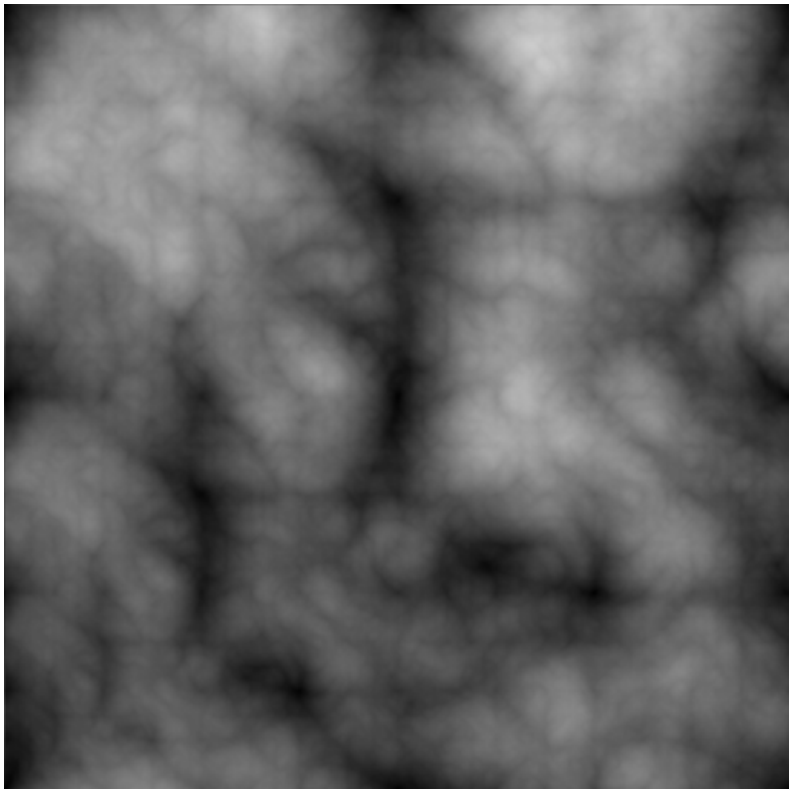
Varying detail levels of fractal perlin noise, starting with one octave and ending with six:



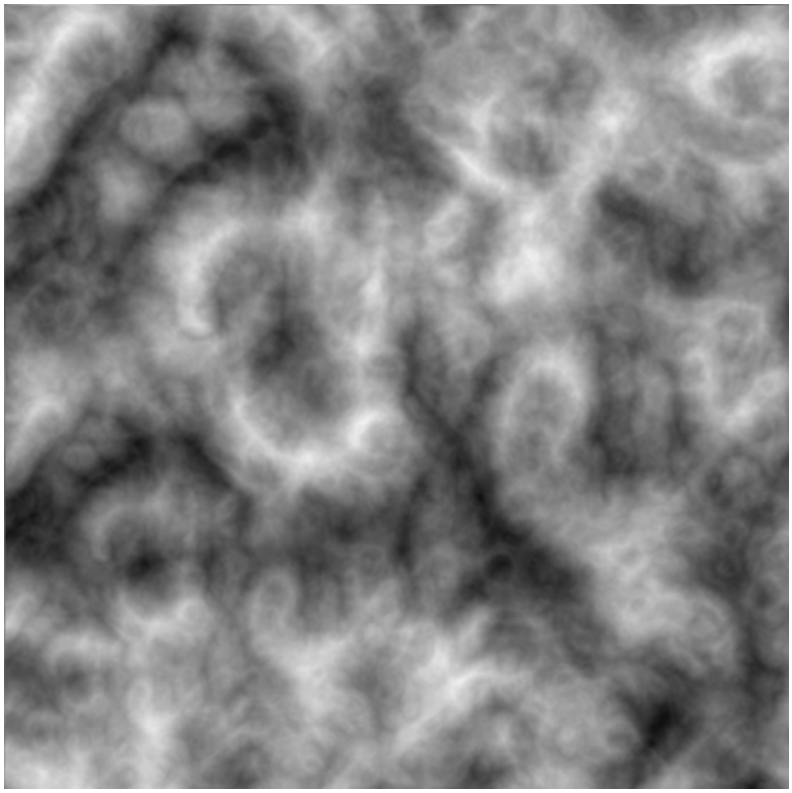
Fractal Perlin noise with 8 octaves and varying persistance values of .35, .5, and .65:



Fractal perlin noise using sums of absolute values:



Fractal sum of reflected Perlin noise:



Optimizing Perlin Noise

To sample n dimensional perlin noise, 2^n psuedorandom gradient vectors are required, each with n components. This means a total of $n*2^n$ psuedorandom numbers must be generated per sample. In addition, the noise looks best when the gradient vectors are normalized to length 1. Using the psuedorandom number generator found in the simpleNoise functions would be very slow when run $n*2^n$ times per sample. A permute array is a much faster way to generate random vectors.

First, create a gradient array that contains 256 normalized n dimensional gradient vectors. I'll call this array g[]. This only needs to be created once when the program starts.

Next, create an array of 256 unsigned bytes. Fill this array with each integer from 0 to 255 in a random, permuted order. This array will be called p[]. This also only needs to be done once when the program starts.

One dimensional fast psuedorandom gradient vector

To generate a psuedorandom index given an integer x, simply return p[x mod 256]. Since p is full of indexes in random order, this essentially generates a random index quickly. This index is then used to read a vector from g[]. This can be sped up even further by taking advantage of the fact that x mod 256 is equivalent to x & 255.
Psuedocode for one dimensional random gradient vector

```
vector randGrad1D( int x, byte p[], vector g[] )
begin
	int i = p[x & 255];
	return g[i];
end
```

N dimensional fast psuedorandom gradient vector

To generate a psuedorandom gradient from a higher dimension, just repeatedly permute the coordinates for each coordinate. Psuedocode for two and three dimensional random gradient vector

```
vector randGrad2D( int x, int y, byte p[], vector g[] )
begin
	int i = p[x & 255];
	int j = p[(y+i) & 255];
	return g[j];
end
```

```
vector randGrad3D( int x, int y, int z, byte p[], vector g[] )
begin
	int i = p[x     & 255];
	int j = p[(y+i) & 255];
	int k = p[(z+j) & 255];
	return g[k];
end
```