Fractal Brownian Motion

A model of signals/processes with dependecies between that makes the value vary so that it tends to be close to the previous values, but still with random variations.

It is self-similar in the sense that the probability distribution is uniform.

"As a centered Gaussian process, it is characterized by the stationarity of its increments and a medium- or long-memory property"

For our purposes: Fractal offsets of geometry

Fractal terrain generation

Statisticaly self-similar fractal (FBM)

but also

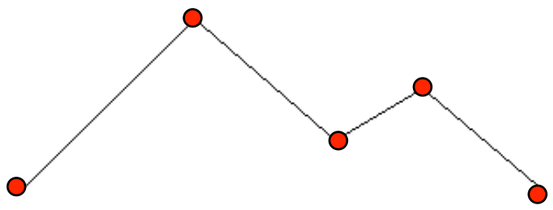
Application of noise functions

Fractal terrain generation

- Fractals: Midpoint displacement/heightfield refinement
- Signal processing: Frequency space filtered noise
- Noise functions: Multiple bands of band-limited noise functions (Perlin noise)

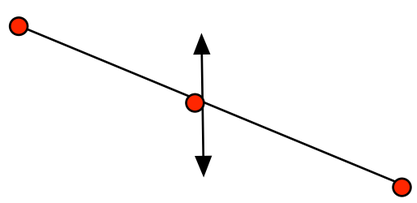
Random midpoint-displacement

Good for fractal terrain generation



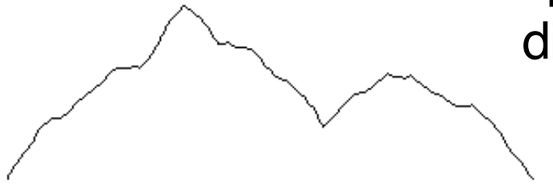
Initiator

Desired rough overall shape



Generator

Find midpoint, displace along y only

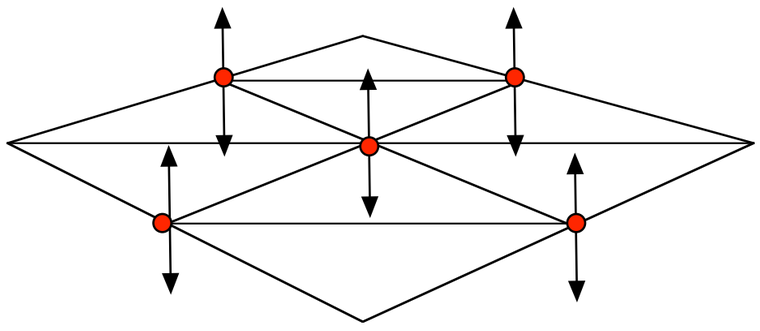


7 iterations

Fractal terrain generation for 2D heightmap

Split a square to four

Displace midpoints of each side and middle



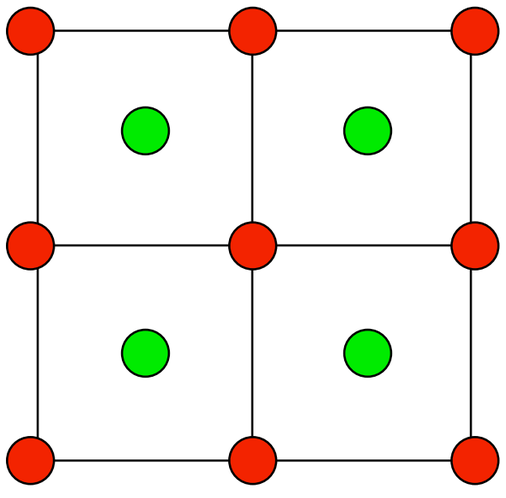
But: What dependencies? Any outside the square?

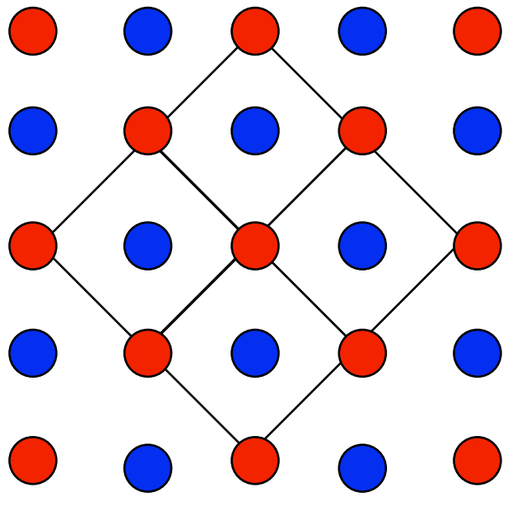
Edge points must match neighbor patches

Diamond-square algorithm

1) Midpoint from corners

2) Midpoint from resulting "diamonds"





Repeat to desired resolution

Diamond-square algorithm

Random offset at each stage

Proportional to size of the side of the square

=> Scale down by sqrt(2) for each phase!

(Not by 2 for every two phases! Popular misconception!)

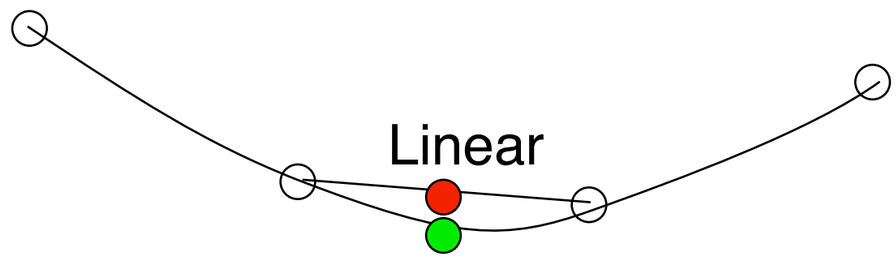
Downscaling = 1/f rule!

Diamond-square algorithm filtering

Important feature! We are reconstructing a signal from samples! (And then add HF detail.)

Simple and fast: Averaging (linear interpolation)

Better: Higher precision filter from larger neighborhood. Usual signal processing rules apply! Use a 4x4 neighborhood.



Better, e.g. cubic spline

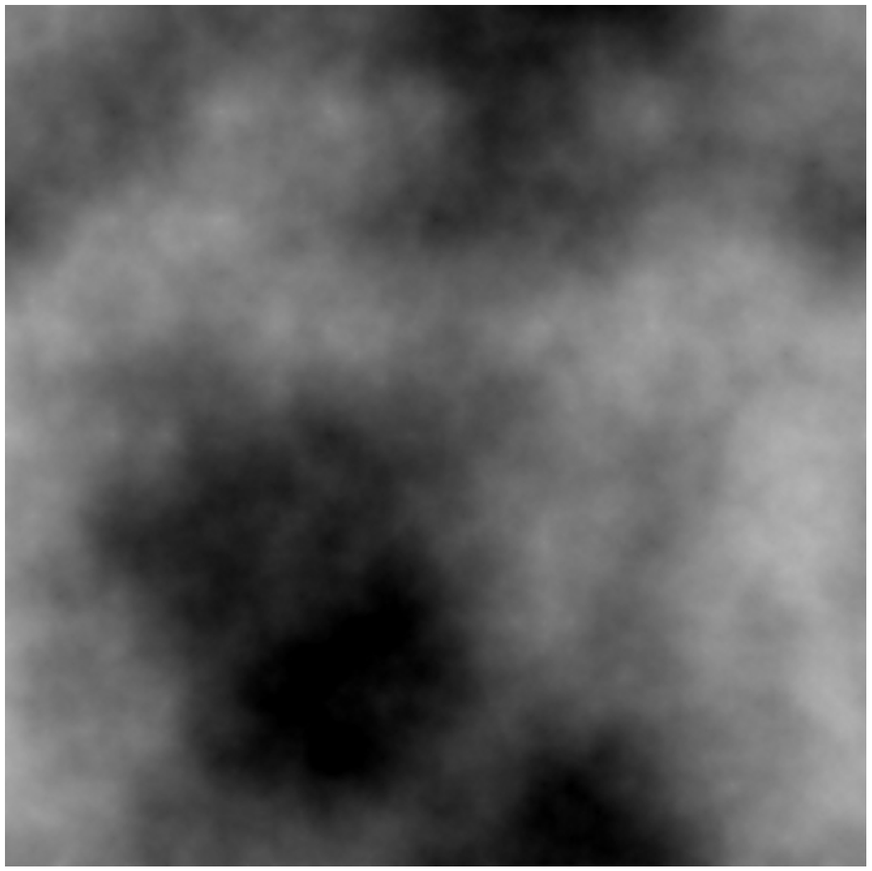
Computational complexity of Diamond-square

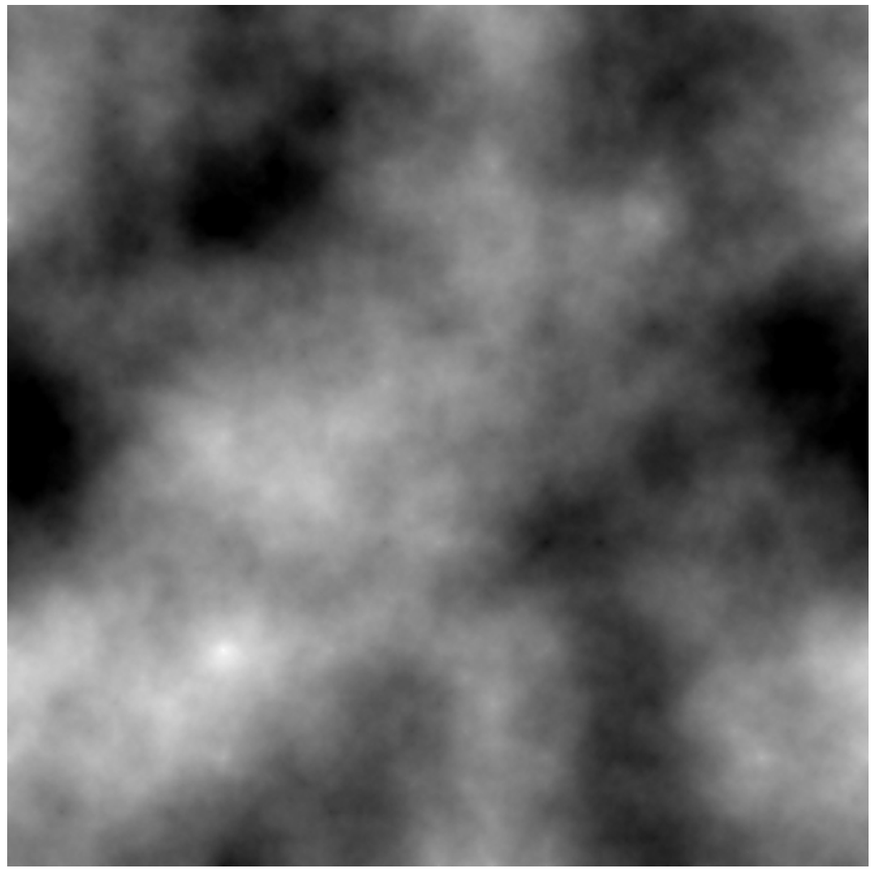
Every point is visited only once!

Hence, $O(N^2)$ for an N^2 image - linear!

Exceptionally fast in its simplest forms

A constant factor gets higher for better filtering





Diamond-square results

Visually pleasing but has artifacts if not properly filtered!

Pyramid approach

”Square-square” algorithm

Terrain level k is array of resolution 2^k x 2^k

The next level has 4x the resolution

Generate new 2x2 block from one, or (better) filter over a small neighborhood

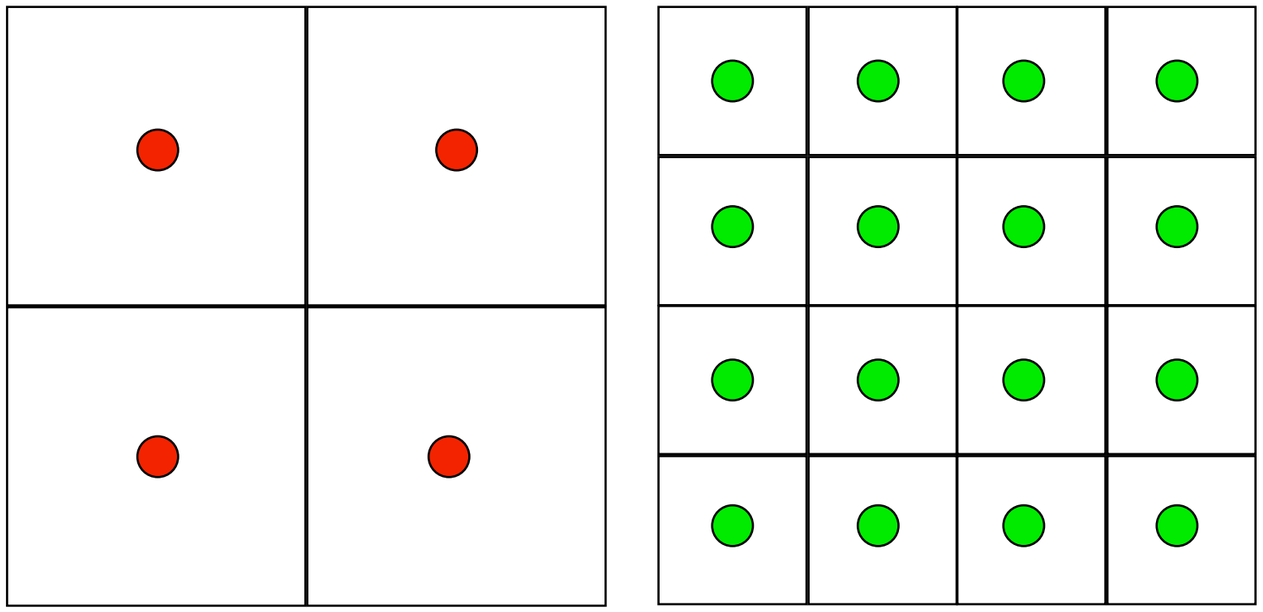
Add random offset to all values

Offset should be smaller for higher k

=> magnitude of frequency components inverse proportional to frequency!

Square square

Image upsampling + add noise



Again: Linear interpolation is simple and fast, better filter gives better result

Essentially: Create a resolution pyramid level by level!

Computational complexity of Square-square

Image size + 1/4 + 1/16... = 1 1/3!

Hence, $O(N^2)$ (linear) but with a higher constant than Diamond-Square

Additional benefit

Produces a resolution pyramid as a side effect! Can be saved and used for level-of-detail

This is possible with other methods but automatic here.

Noise functions

Fractals and noise functions are closely related

Noise can look natural... but when?

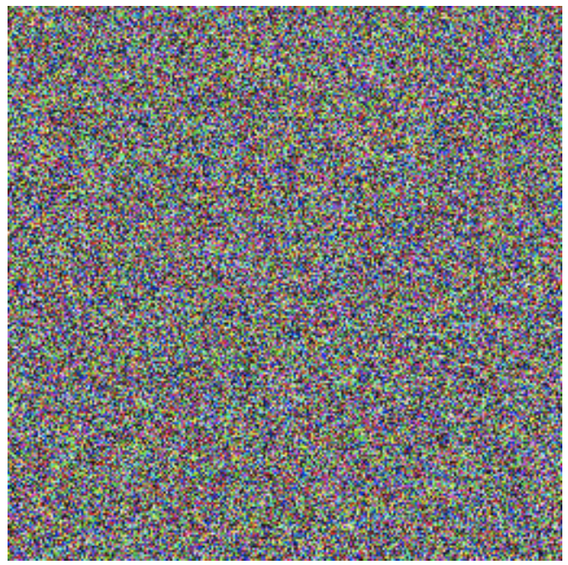
- white noise
- colored noise
- value noise
- gradient noise

White noise

Same amplitude in all frequencies

Useless as it is?

Can be processed to something better.

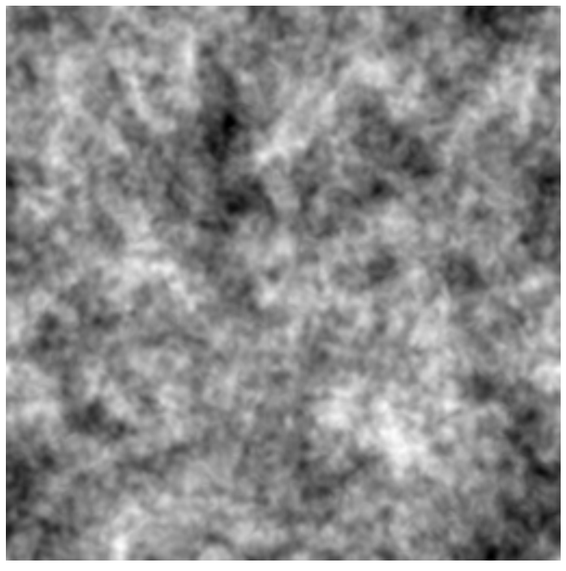


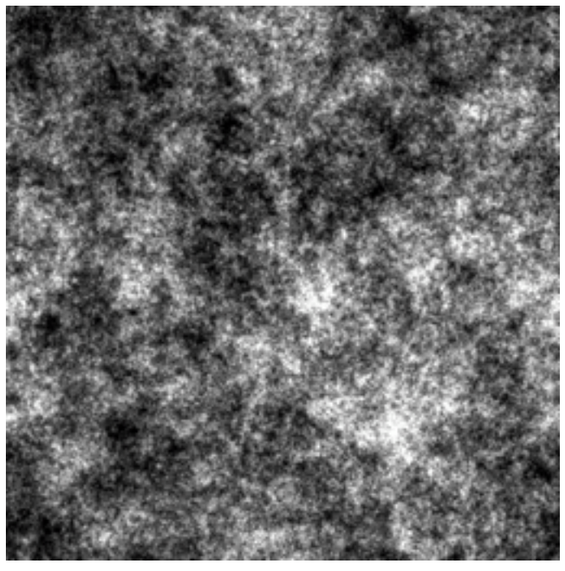
Colored noise

Amplitude varies with frequencies

With the right variation, it can look nice - natural!

The 1/f rule!





Colored noise

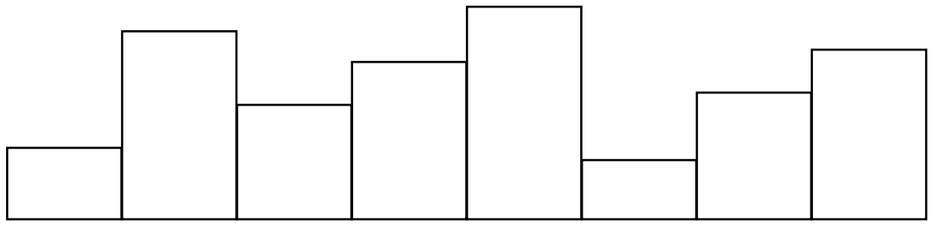
Can be processed with filters, e.g. frequency plane functions

Considered too computationally heavy. (Questionable!)

Value noise

If you just fill your pixels with values in some range, you get value noise (essentially white noise).

Value noise is perfectly useful after proper filtering, possibly combining several frequency bands.



Colored noise by filtering in the frequency plane

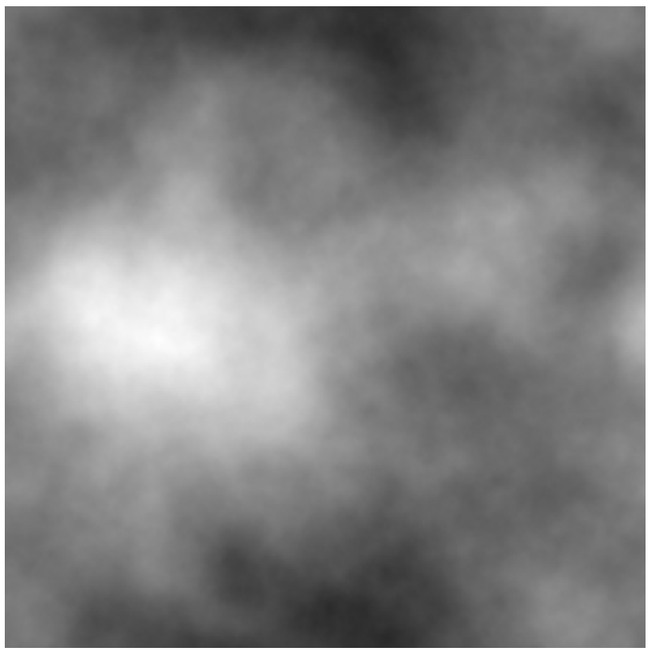
Fill frequency space (2D) with random numbers (white noise)

Filter by G(f) = F(f) * 1/|f|

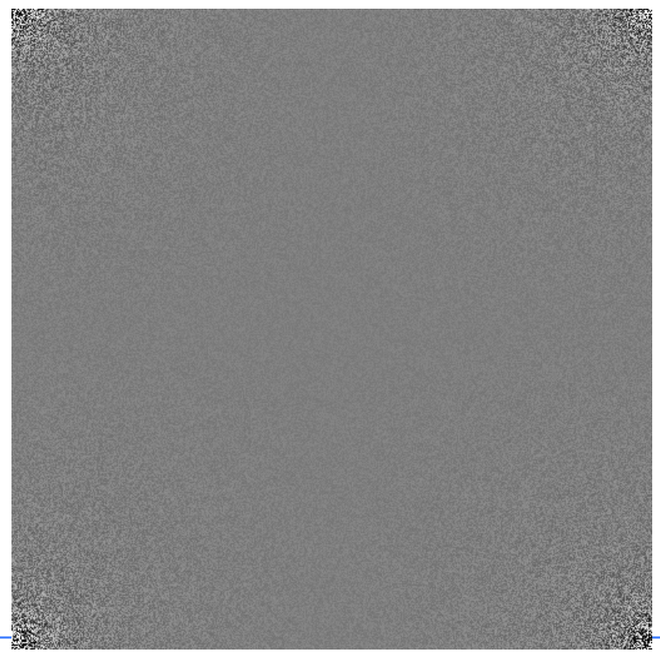
Convert to spatial image with FFT

Filter white noise by 1/f

Example

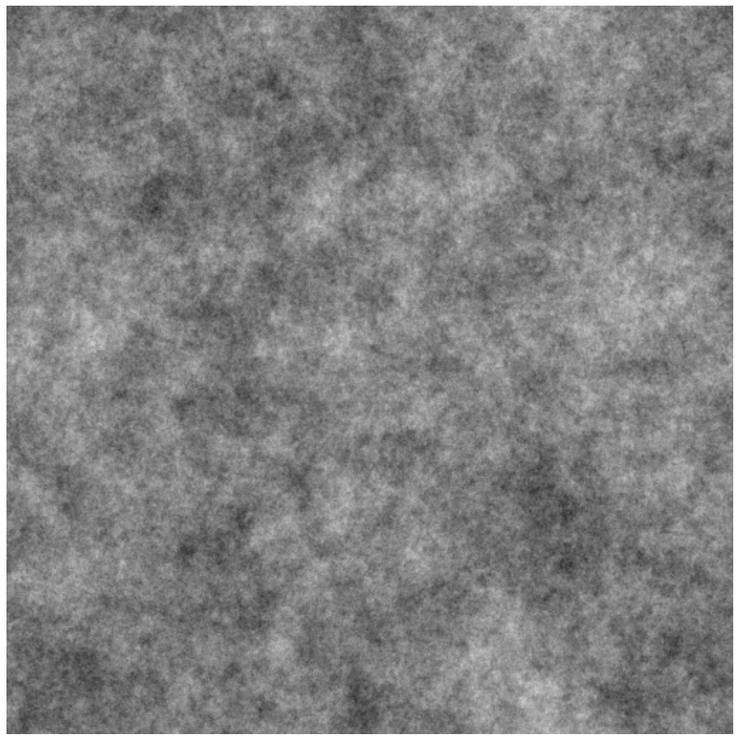


Frequency space:



Other falloffs than 1/|f|

Too much ramp



Too little ramp

Advantages of frequency plane filtered noise

Extremely good precision for the frequency behavior

Repeating patterns, good for textures

FFT well researched, highly optimized implementations = fast

Octave

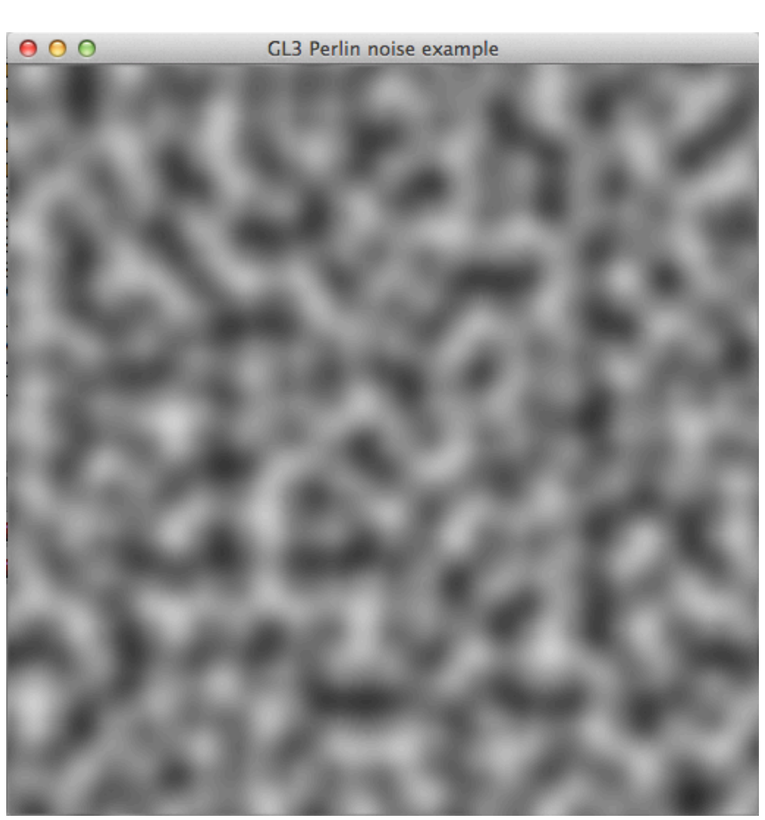
Usually a music term.

One octave = 2x frequency!

One octave is exactly at half of the string length!

Gradient/Perlin noise

Single octave



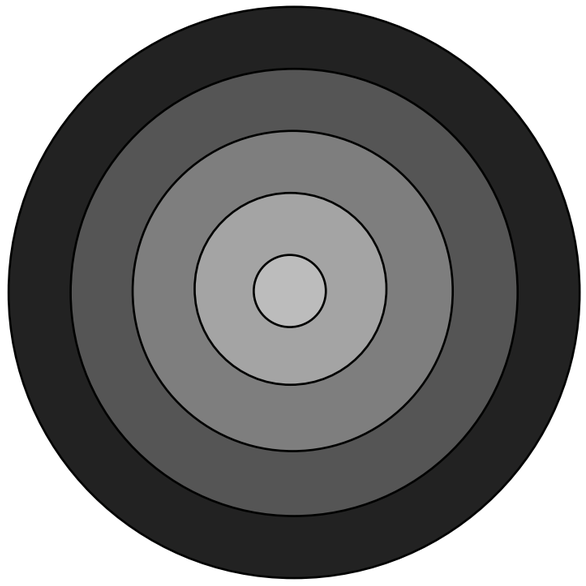
Single octave Perlin

```
void makeperlintexture()
{
   int x, y;
   float i[3];
   char val;

   for (x = 0; x < 128; x++)
   for (y = 0; y < 128; y++)
   {
      i[0] = x / 8.0;
      i[1] = y / 8.0;
      val = (char)(clamp(noise2(i) * 200.0, -127, 127))+128;

      ptex[x][y][0] = val;
      ptex[x][y][1] = val;
      ptex[x][y][2] = val;
   }
}
```

Multi-octave Perlin

Multiple "rings" in frequency space

Each ring scaled after the 1/f rule!

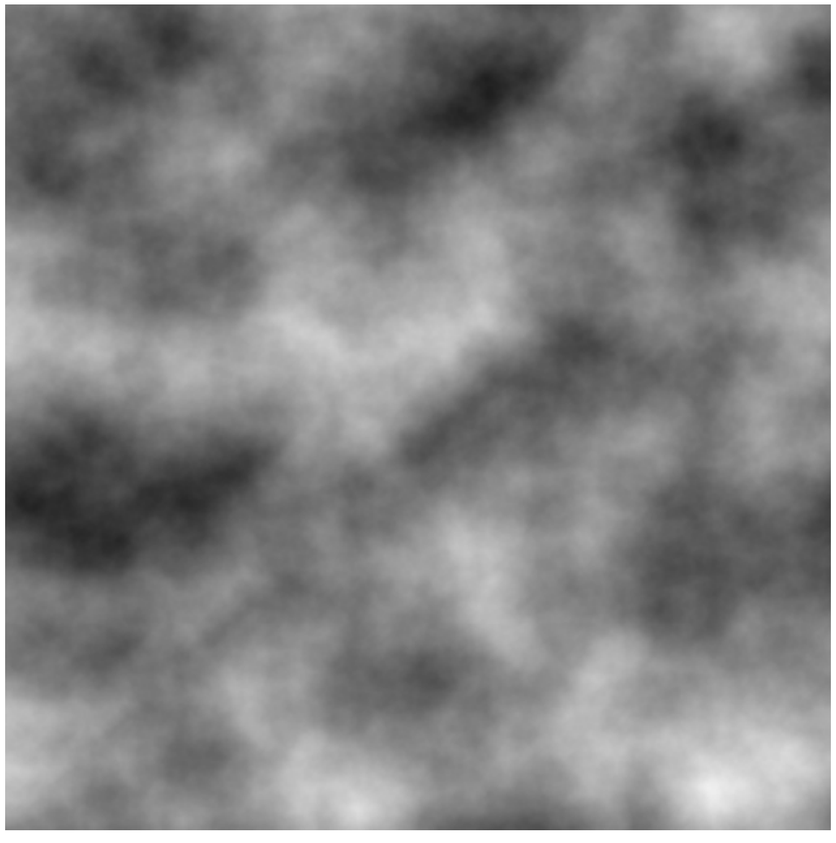


```
void makeperlintexture()
{
   int x, y;
   float i[3];
   char val;

   for (x = 0; x < 128; x++)
   for (y = 0; y < 128; y++)
   {
// 5 octaves:

      i[0] = x / 32.0;
      i[1] = y / 32.0;
      i[2] = 0.0;

      val = (char)(noise3(i) * 128.0)+128;

      i[0] = x / 16.0;
      i[1] = y / 16.0;
      i[2] = 2.0;

      val += (char)(noise3(i) * 64.0);

      i[0] = x / 8.0;
      i[1] = y / 8.0;
      i[2] = 3.0;

      val += (char)(noise3(i) * 32.0);

      i[0] = x / 4.0;
      i[1] = y / 4.0;
      i[2] = 4.0;

      val += (char)(noise3(i) * 16.0);

      i[0] = x / 2.0;
      i[1] = y / 2.0;
      i[2] = 5.0;

      val += (char)(noise3(i) * 8.0);

      ptex[x][y][0] = val;
      ptex[x][y][1] = val;
      ptex[x][y][2] = val;
   }
}
```

Twice the frequency - half the amplitude!

Result

Similar to the other methods



Gradient noise vs FFT

Some say FFT is slow. Questionable today!

Gradient noise claimed to be very fast. (Compared to what?)

Frequency space processing much simpler algorithm (simple weighting curve, based on 1/f, FFT) and great control, but requires O(NlogN) operations.

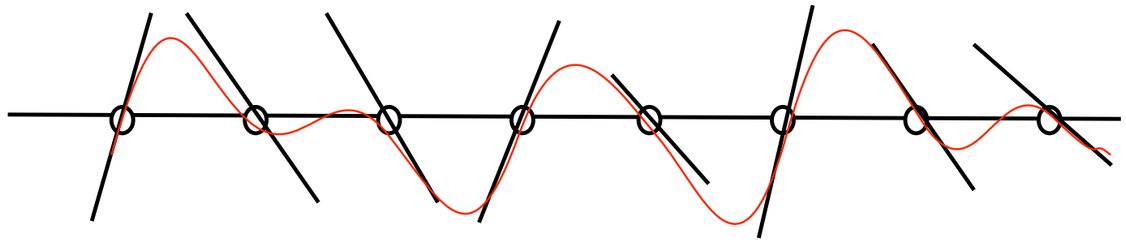
One pass Gradient noise faster... but don’t we need many?

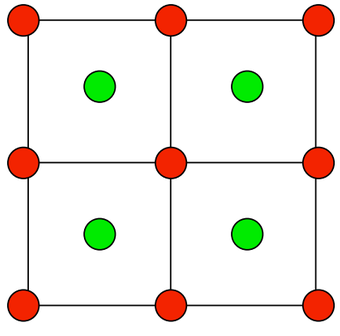
Artifacts

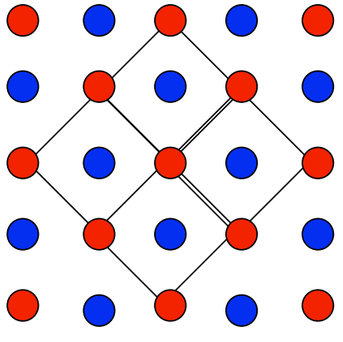
Diamond square and Perlin noise are incomplete! They both "lock" in certain points, only producing certain phases of the signal.

I.e. produce only the cosine part of a signal and skipping the sin!

This can be corrected by generating two sets of the signal, with a proper offset!







Applications of FBM

Terrains

Textures, texture detail

Smoke

Etc...

Feature comparison

Scalability: Diamond square and Square square very easy to scale to more detail but hard to expand to new patches.

Perlin easy to scale if you add additional octaves - which degrades performance. Strongest point: Easy to expand to new patches + easy to parallellize.

Control: Frequency plane filtering has extreme control. The others depend on weights on octaves.

Your application needs may decide.

Repeating textures

Frequency plane filtering creates repeating textures automatically!

Diamond square can, trivially

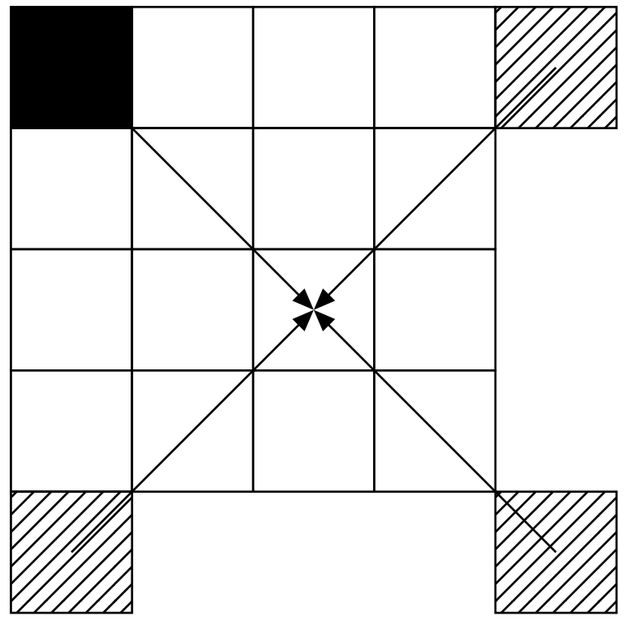
Perlin noise is repeating in special versions.

Good feature for texture generation.

Repeating texture from Diamond-Square

Generate from ONE corner instead of four!

The most natural way to run the algorithm!



Adapt frequency behavior (change amplitude) to other needs

The change of frequency per step is called *lacunarity* and is the change if frequency per step. Is is almost always 2.0. (The definition of "octave"!)

The scaling of amplitude per step is sometimes called *persistence* or *gain* but the course book just calls it H. At 0.5 it will match the 1/f rule.

How about other frequency behaviors?

Just change the amplitude variation per step!

Trivial modificaton for both Perlin and Diamond-Square, just change the scale. For frequency plane filtering, change the curve.

Useful for e.g. different biomes, like deserts.

Repeating texture from gradient noise

Requires a modified algorithm.

Popular method: Use 3D noise instead and map the texture to a torus in 3D.

This makes the noise non-uniform!

Not the best case for gradient noise! Use the modified 2D noise instead.

Performance comparison

Produce an NxN image

Diamond square: $O(N^2)$

Square square: $O(N^2)$

Single octave Perlin: $O(N^2)$

Multi octave Perlin: $O(N^2 \log N)$

Frequency plane filtered: $O(N^2 \log N)$

All produce similar results except single octave Perlin.

Parallelism

Parallel implementations of Diamond square, Square square and Frequency plane filtering all require multiple passes!

Perlin noise is calculated in one pass = fragment shader friendly!

This is where Perlin and similar shines! Locality!

(More on multi-pass processing next time.)

What more can we do with the terrain?

- Add water, calculate rivers and lakes
- Erosion effects (esp along rivers)
- Roads
- Vegetation and buildings
- Expand into new patches
- Multitexturing for different kinds of locations (slopes, height)
- Different generation for different climates/biomes (mountains, deserts...?)

Multi-patch terrains

Terrains are never drawn one triangle at a time.

But very large terrains should not be drawn as a single model!

- Too much geometry out of view
- Too much data loaded in VRAM

Split to patches!

- Use frustum culling
- Re-generate or swap to CPU/disk as needed
- LOD on the patches is a good idea

Conclusions of terrain generation

The backbone of procedural environment generation!

Fractal or noise? Same thing!

Higher frequencies - lower amplitudes. (Typical for natural images as well as a rule in fractals.)

Perlin and similar good for parallelism, but not the lowest computational complexity!

Other methods are better for repeating textures.