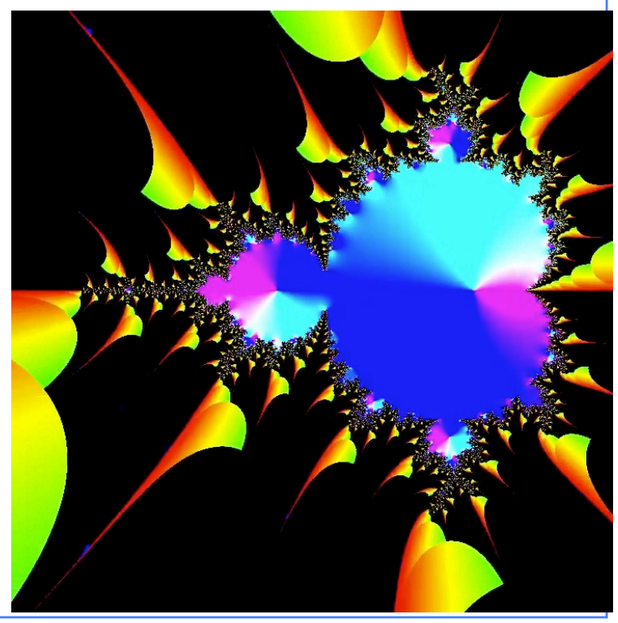
TSBK07/TSBK11

Computer Graphics

Ingemar Ragnemalm, ISY



Lecture 14

Random numbers

Noise

Gradient noise

Fractal Brownian Motion

Terrain generation

Guest lecture: Diffusion

Lecture questions

1. How can you produce random numbers on the GPU?

2. What significant features does gradient noise have?

3. How can you produce FBM with FFT?

4. What is lacurarity?

5. Which terrain generation method has the lowest computational complexity?

6. What are the advantages of gradient noise?

Random numbers

Important source of interesting patterns and other variations

Randomness is chaos?

Randomness with structure

"Random Number Generation is too Important to be Left to Chance" (Robert R. Coveyou 1970)

What is random?

Pick a "random" number - never truly random

People never pick the same number twice. Randomness does!

Roll a die

Is the die fair?

But how can we make a computer "roll a die"?

Pseudo-random numbers

Computers are intrinsically deterministic

How can we make randomness?

Pseudo-random number generation

Random library functions

Pseudo-random number functions are in most run-time libraries

High quality random functions, long sequencies

rand() and srand() in the standard libraries

random() and srandom()

Not cryptographically secure; irrelevant for us.

How about rand(), random()...?

But...

The random number libraries are generally *sequential*!

Not useable in a shader.

Also questionable in the parallel image generation model.

In shaders, we need another solution!

Random numbers in GLSL

GLSL runs on the GPU

Built-in random functions never worked (!)

Typical ways to get noise in GLSL:

- Noise texture, pre-generated noise
- Feed shader continuously with numbers from host
- Render to texture
- Truncated trigonometric numbers
- Hash functions

Truncated trigonometric numbers ("Trash bits hash")

Smart trick with built-in functions

M is a number >> 1

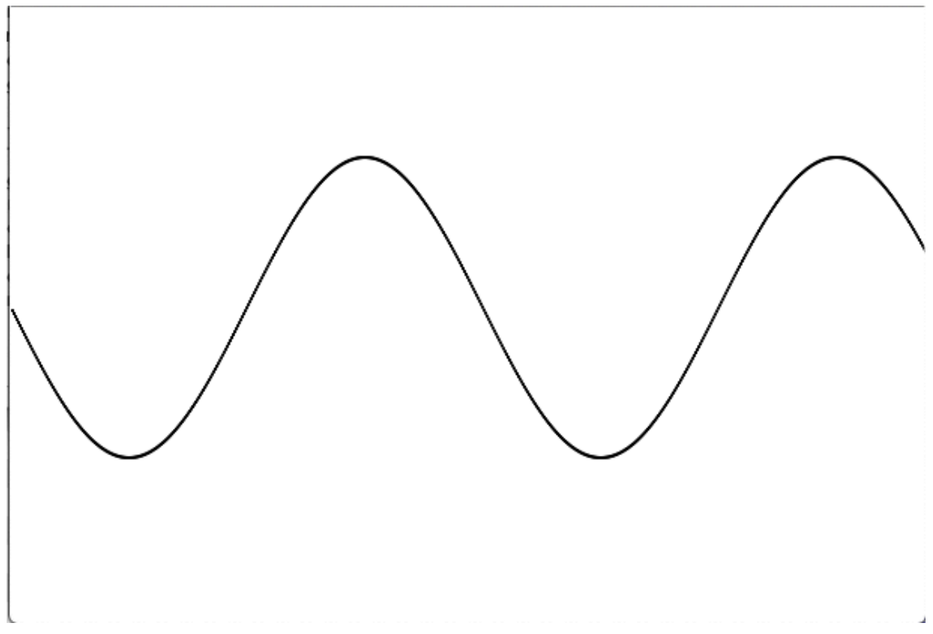
n = sin(x) * M

r = n - (int)n

Plain sin function

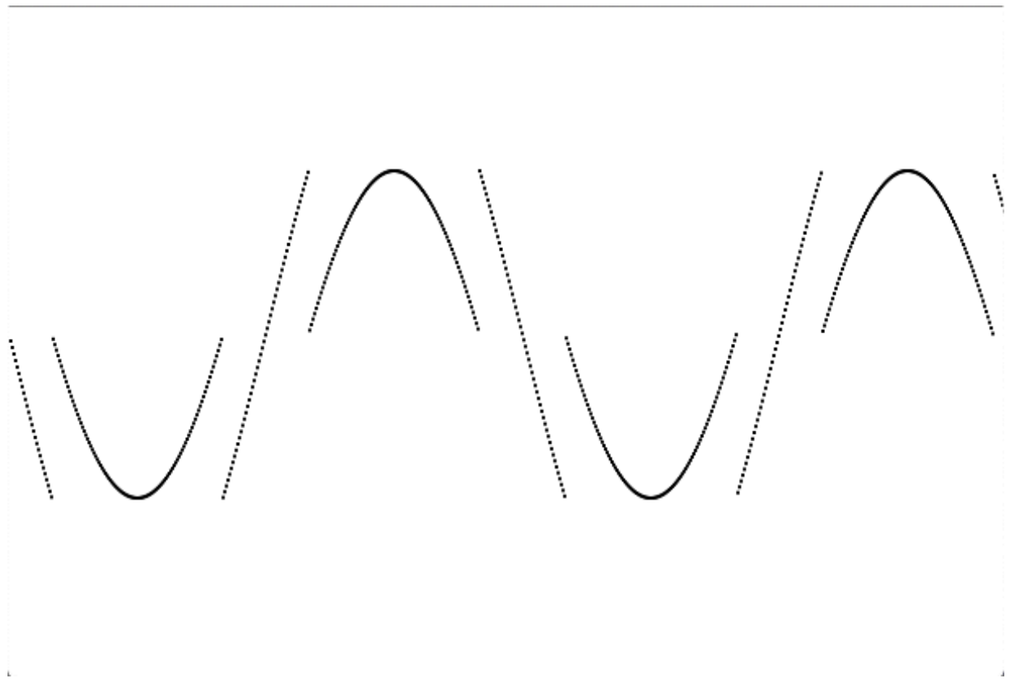
f := sin(x);

Nothing random about it



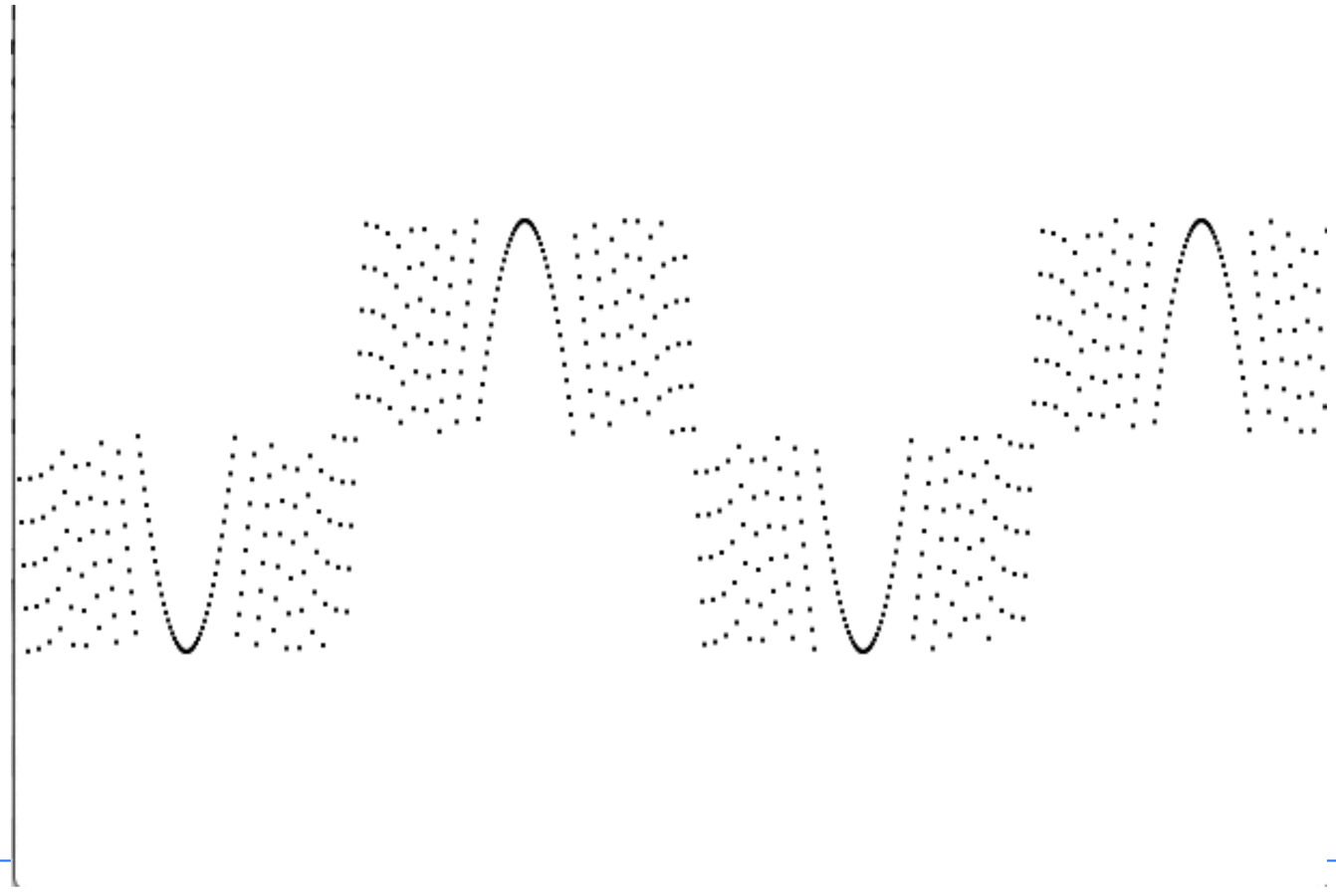
Multiply by 2, take the fraction part

f := frac(sin(x) * 2);



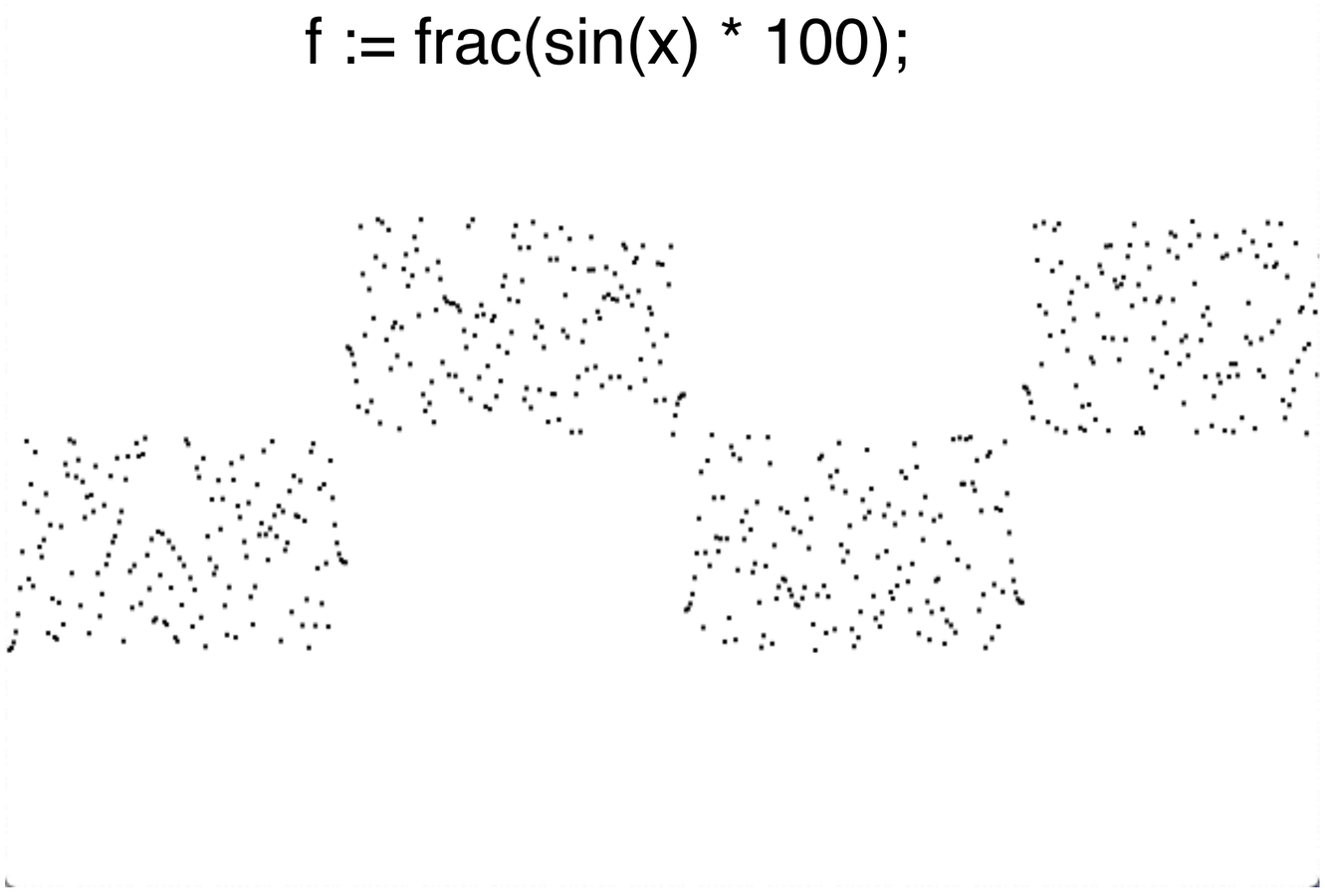
But if we multiply by more...

f := frac(sin(x) * 10);



...and even more...

f := frac(sin(x) * 100);



abs() to keep on one side and go high

f := frac(sin(x) * 100000);

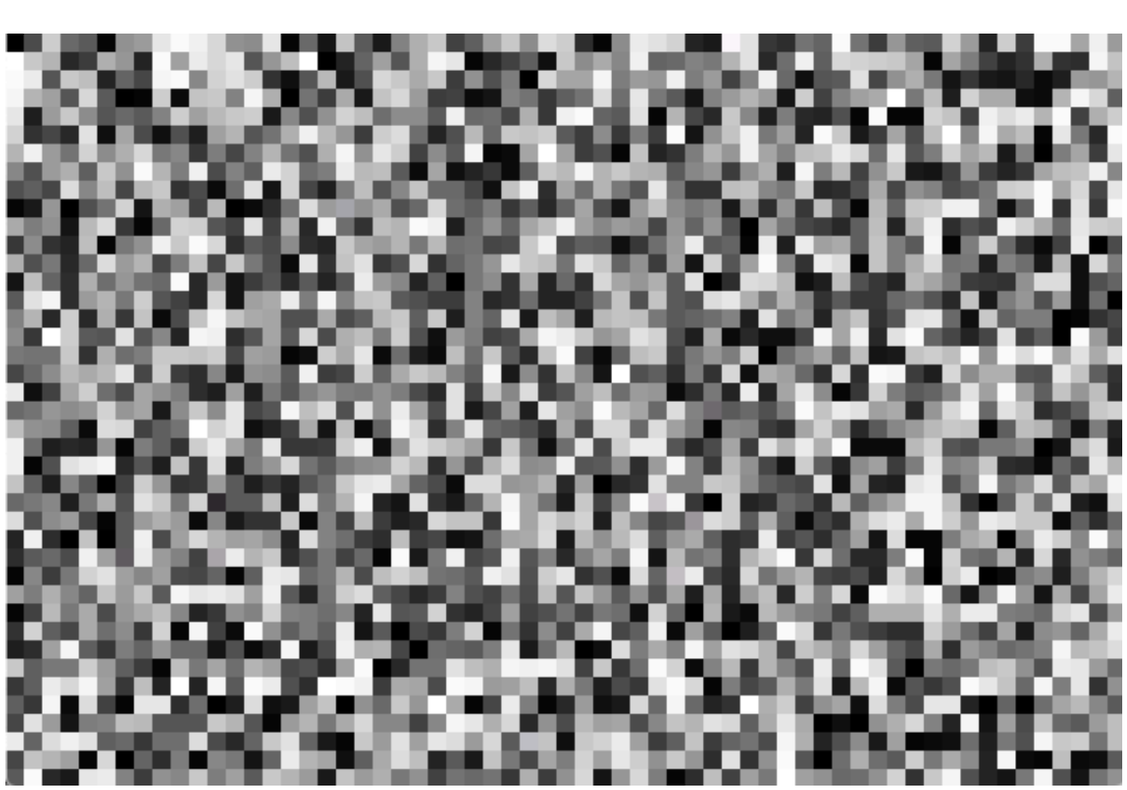
A random number generator!

We now have a non-sequential random number generator!
Only the pixel position is needed!

Just never take steps by π!

Random pixel values

Random intensity



Random numbers by hashes

Drawbacks with truncated trigonometric functions:

- The result is platform/machine dependent!
- High computational cost

Stefan says: "This is an unpredictable, ugly hack, and you really shouldn’t use it."

Instead: Take the seed and jumble the numbers (even bits) to create a pseudo random number.

One-dimensional hash

A hash can work with xor functions and shifts.

Example from the book we wrote: PCG hash

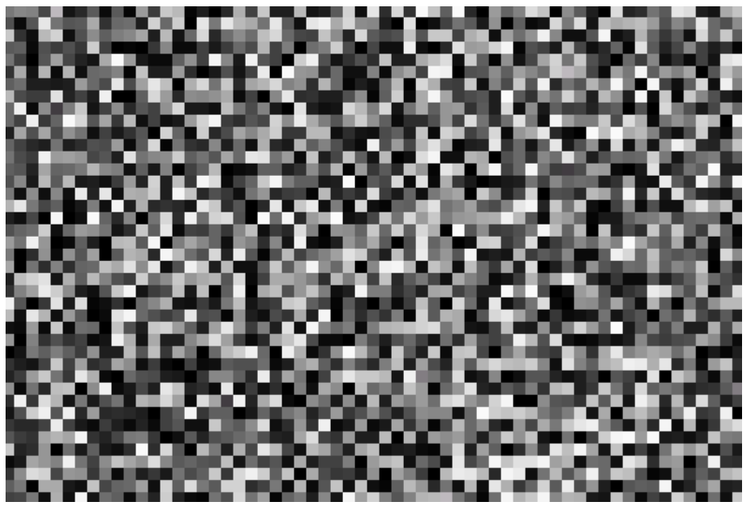
```
uint pcg_hash(uint input)
{
    uint state = input * 747796405u + 2891336453u;

    uint word = ((state >> ((state >> 28u) + 4u)) ^ state) * 277803737u;

    return (word >> 22u) ^ word;
}
```

Two-dimensional hash

For creating images, we need two dimensions.

One way to use 1D hashes is to call them twice, once for each coordinate.

hash(x , y) = hash(hash(x)+ y),



pcg_hash in 2D, perfectly OK?

Permutation polynoms

Modulo-based hash instead of xor-based. A permutation polynomial is a polynomial modulo a prime number.

The quadratic polynomial

$$p(x)=(2Ax^2+Bx) \bmod A^2$$

where A is a prime number, are guaranteed to be permutation polynomials

Stefan suggests A=17, B=10:

$$hash = (34x^2 + 10x) \bmod 289$$

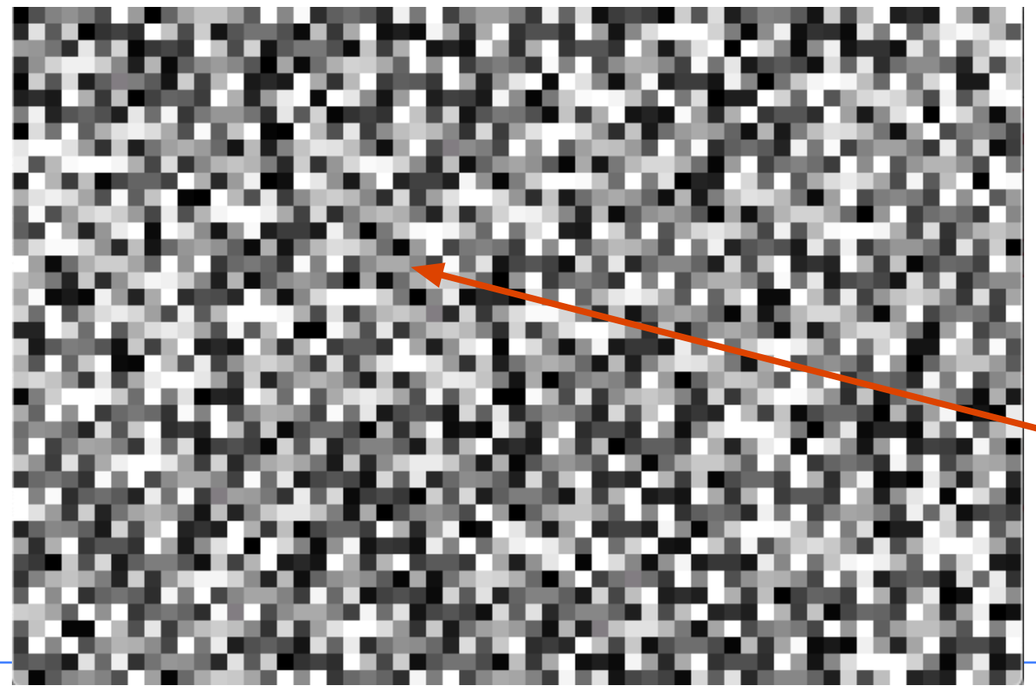
Usage in 2D

The hash function must be called *twice* to get both coordinates!

$$hash(i) = (34i^2 + 10i) \bmod 289$$

```
hash(hash(x)+y);
```

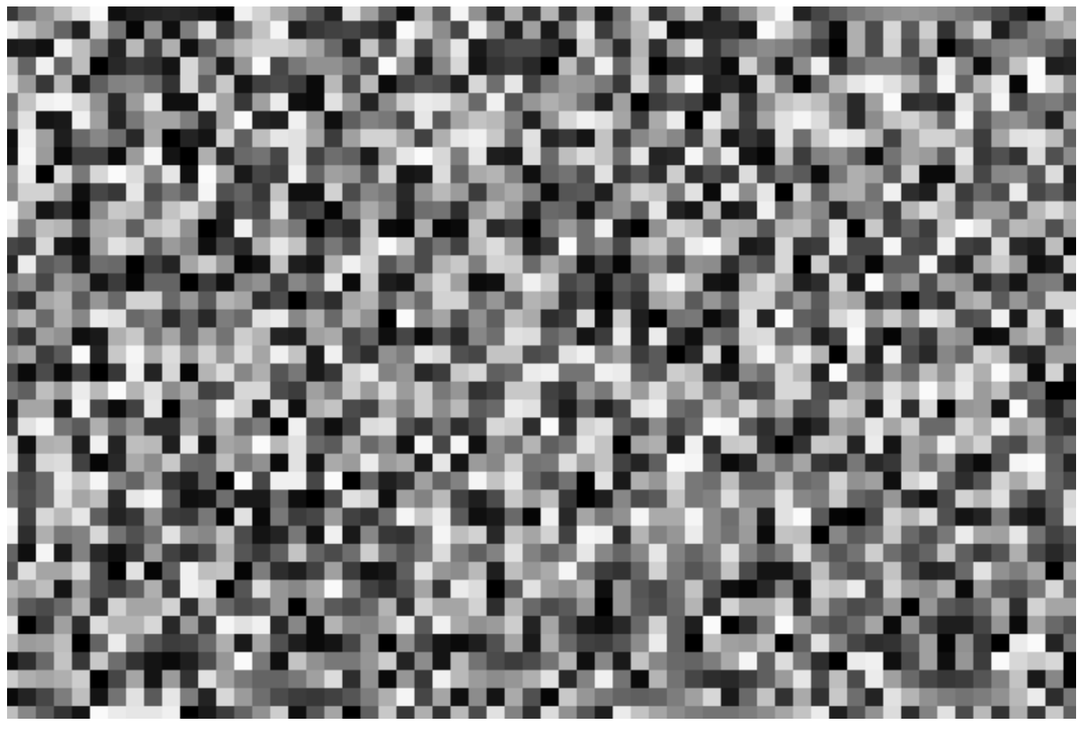
This creates a nice 2D randomness:



But... is that gray area truly random?

Messing with hash functions some more

How about this?



What is magic with 289? Nothing.

It is just a hack but ...

h := (141*x*x*y + 213*y*y*x) mod 289;

Conclusions on random number generation

Work from positions to get a repeatable randomness

Add additional seed (e.g. time) if it should vary

Truncated harmonic functions are popular but not good.

Hash functions are faster and more stable.

Uses of randomness

Random patterns

Random geometry

Random movement

Random location

and more...

Gradient noise (Perlin noise)

We need smooth noise!

Random numbers for pixel values is pretty good... but tends to be "blocky" with simple interpolation.

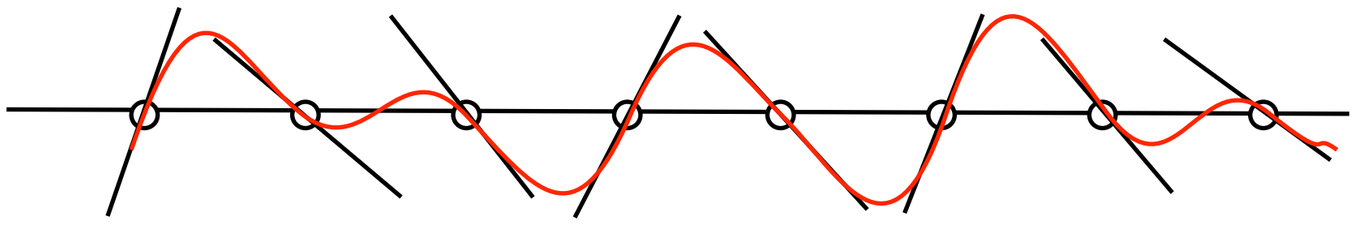
High quality interpolation is good but is computationally costly.

An easier way to get smooth noise: Random gradients.

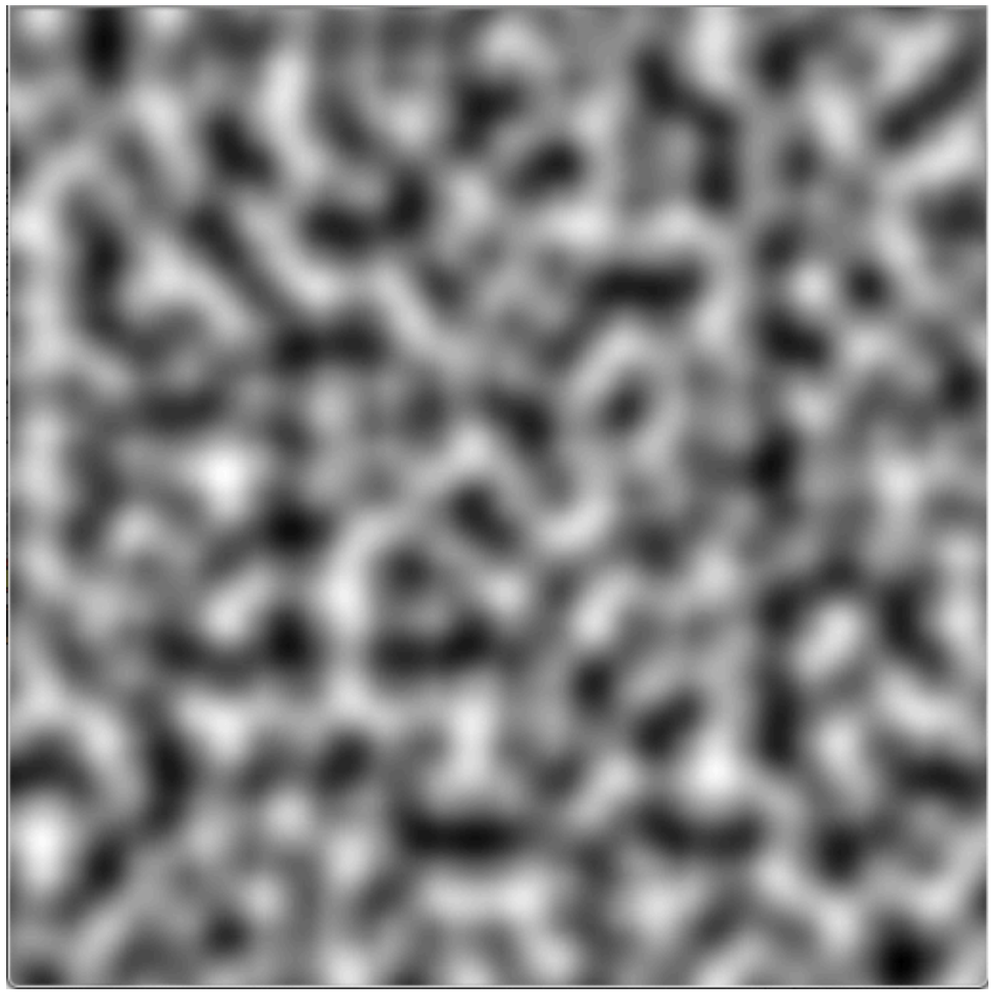
Gradient noise

If the values are used for gradients instead of height,
we get gradient noise. (Perlin noise.)

The function is interpolated to match the gradients.



Gradient/Perlin noise



Implementation

Not Perlin's but still gradient noise. Brief version by Inigo Quilez

```
vec2 random2(vec2 st)
{
    st = vec2( dot(st,vec2(127.1,311.7)),
               dot(st,vec2(269.5,183.3)) );
    return -1.0 + 2.0*fract(sin(st)*43758.5453123);
}

// Gradient Noise by Inigo Quilez - iq/2013
// https://www.shadertoy.com/view/XdXGW8
float noise(vec2 st)
{
    vec2 i = floor(st);
    vec2 f = fract(st);

    vec2 u = f*f*(3.0-2.0*f);

    return mix( mix( dot( random2(i + vec2(0.0,0.0) ), f - vec2(0.0,0.0) ),
                     dot( random2(i + vec2(1.0,0.0) ), f - vec2(1.0,0.0) ), u.x),
                mix( dot( random2(i + vec2(0.0,1.0) ), f - vec2(0.0,1.0) ),
                     dot( random2(i + vec2(1.0,1.0) ), f - vec2(1.0,1.0) ), u.x), u.y);
}
```

fract-based random numbers

Integer and fractional parts

Spline/smoothstep

Interpolate 4 nearest

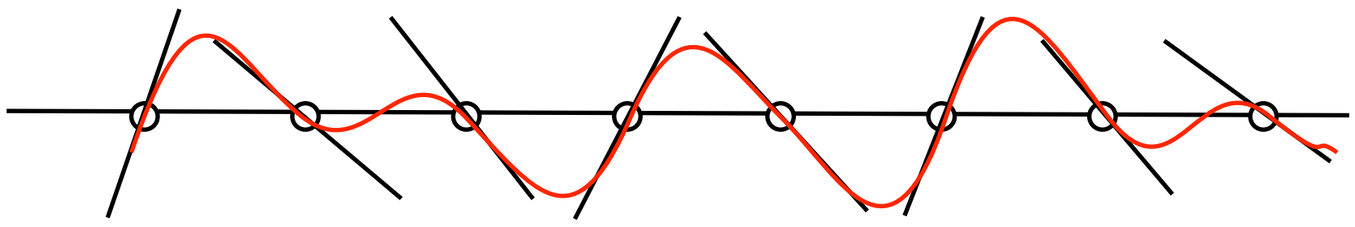
Artifacts

Gradient noise is incomplete! It "locks" 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!

Usually ignored.



Smooth noise = limited bandwidth

Ideally a smooth circle in frequency space

Limited information but can be combined with more.
More about that later.

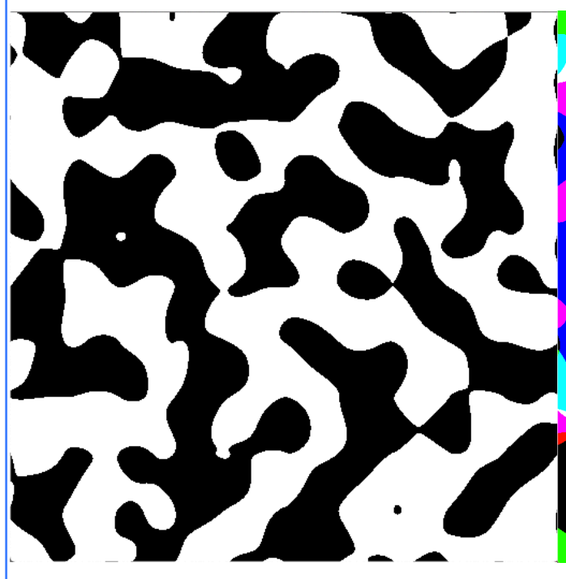
Usages of smooth noise

Treshold for interesting shapes

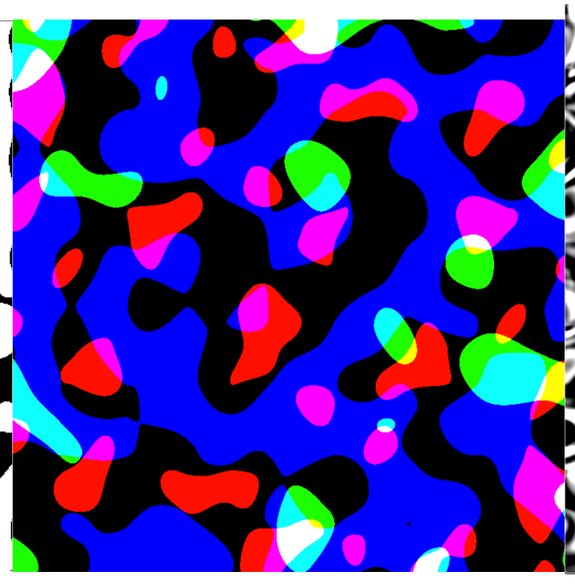
Shape modulation, use an existing shape and modulate the positions

Combine with other data (including other noise) for interesting effects

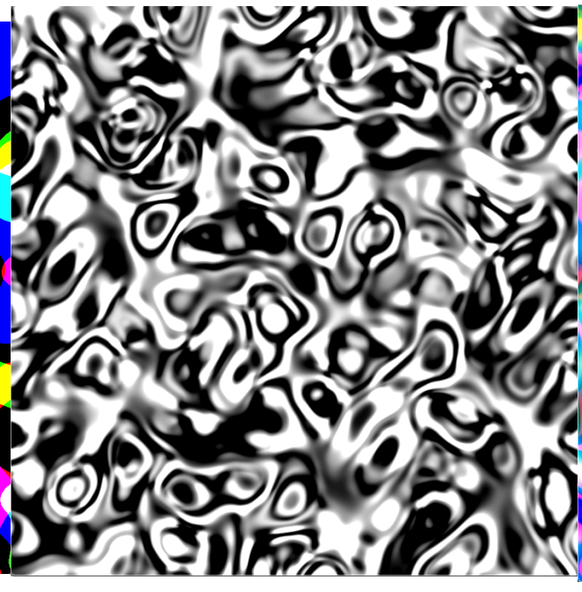
Just messing about...

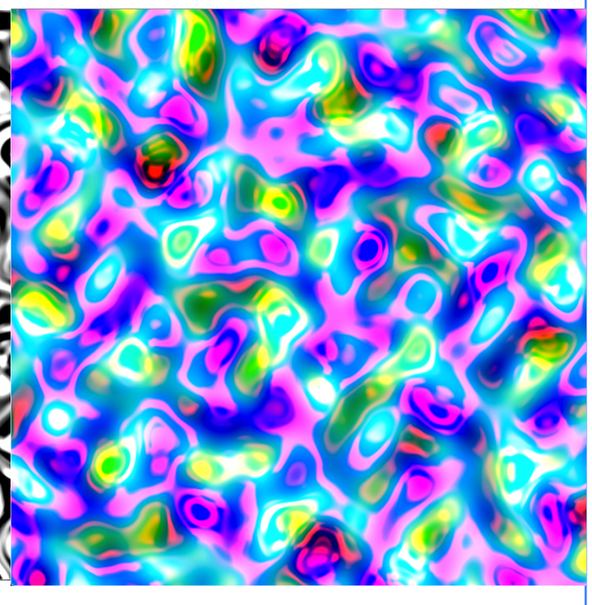


Plain noise, truncated



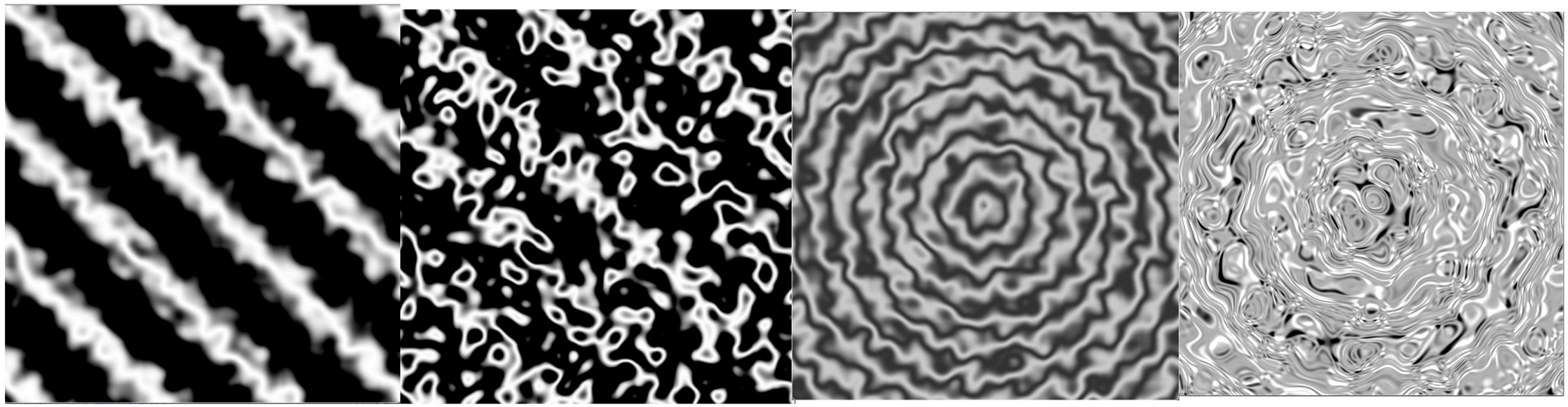
Multiple layers of noise





Noise of noise

Mix with other functions, here sin:



Noise modulating linear sin

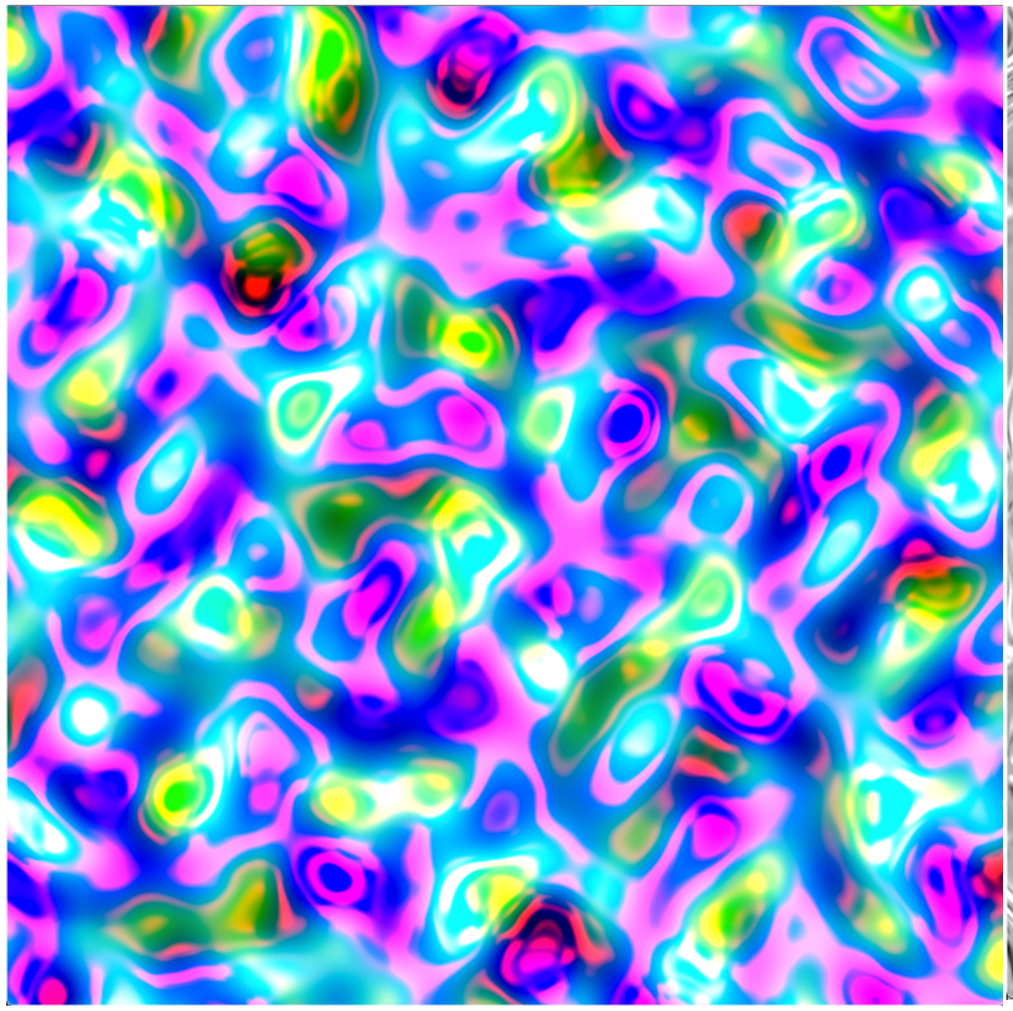
Noise modulating linear sin+

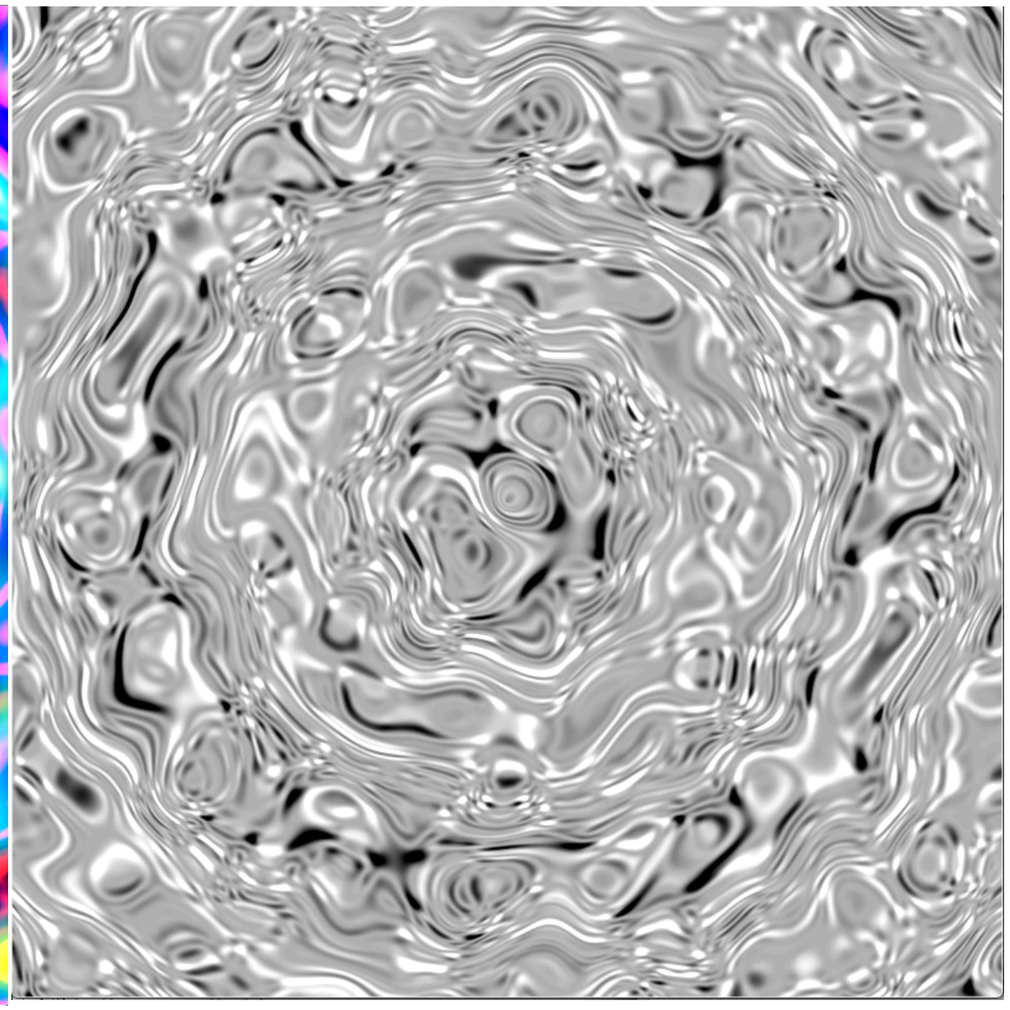
Noise modulating radial sin

A serious mess

***Controlled noise* gives it structure!**

Seriously... what are these?





But we can do even better:

Simplex noise

Gradient noise based on triangles/tetrahedrons. Ken Perlin's replacement for "Perlin noise" (gradient noise on quads).

Advantage: Fewer points affect each output.

Also applied other computing strategies for higher performance.

Fractal terrain generation

Statisticaly self-similar fractal ("Fractal Brownian Motion")

but also

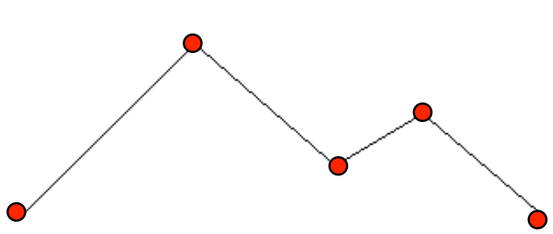
Application of noise functions

Three approaches:

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

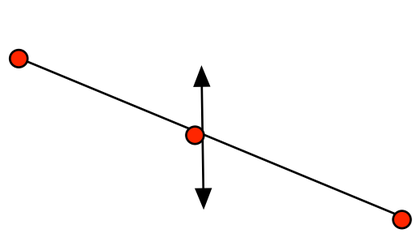
Random midpoint-displacement

Good for fractal terrain generation



Initiator

Desired rough overall shape



Generator

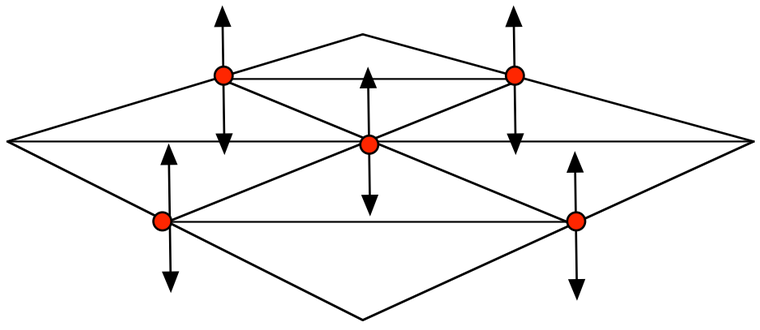
Find midpoint, displace along y only

7 iterations

Fractal terrain generation for 2D heightmap

Split a square to four

Displace midpoints of each side and middle



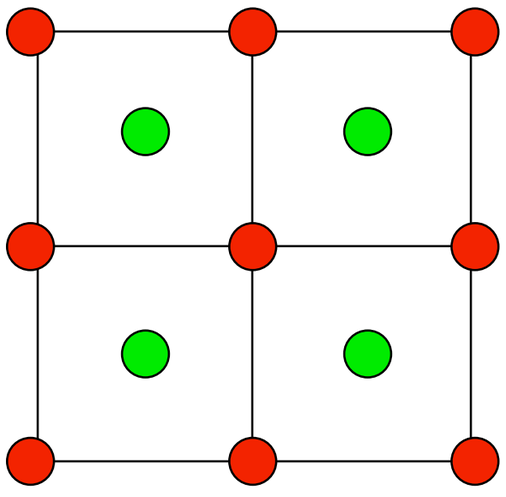
But: What dependencies? Any outside the square?

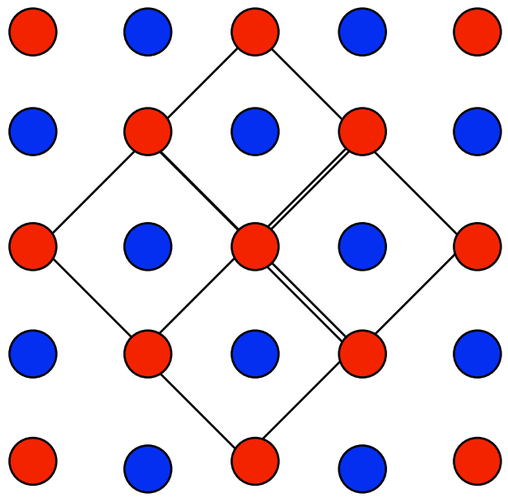
Edge points must match neighbor patches

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 length of the side of the square

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

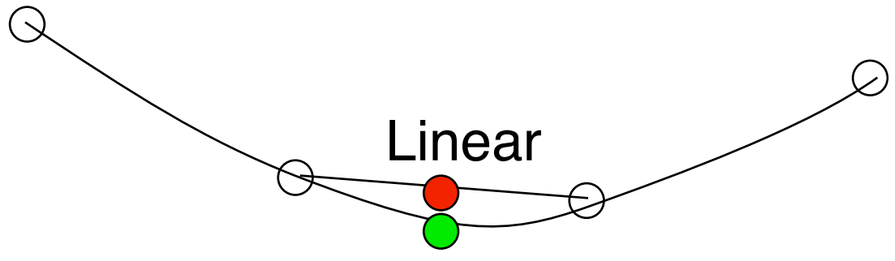
(Not by 2 for every two phases! Popular misconception!)

Diamond-square algorithm filtering

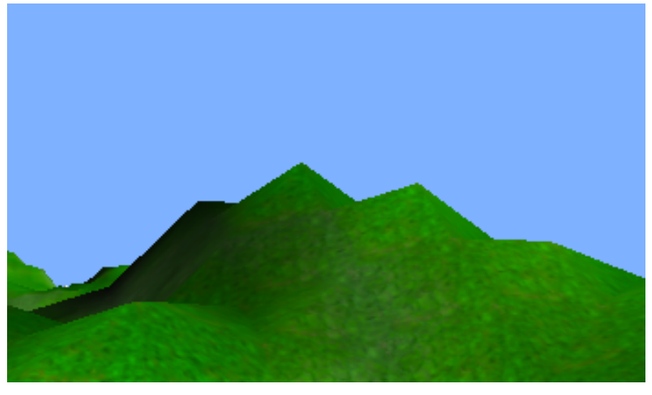
Important feature! We are reconstructing a signal from samples! (And then add HF detail.)

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



Jagged peaks caused by bad filter

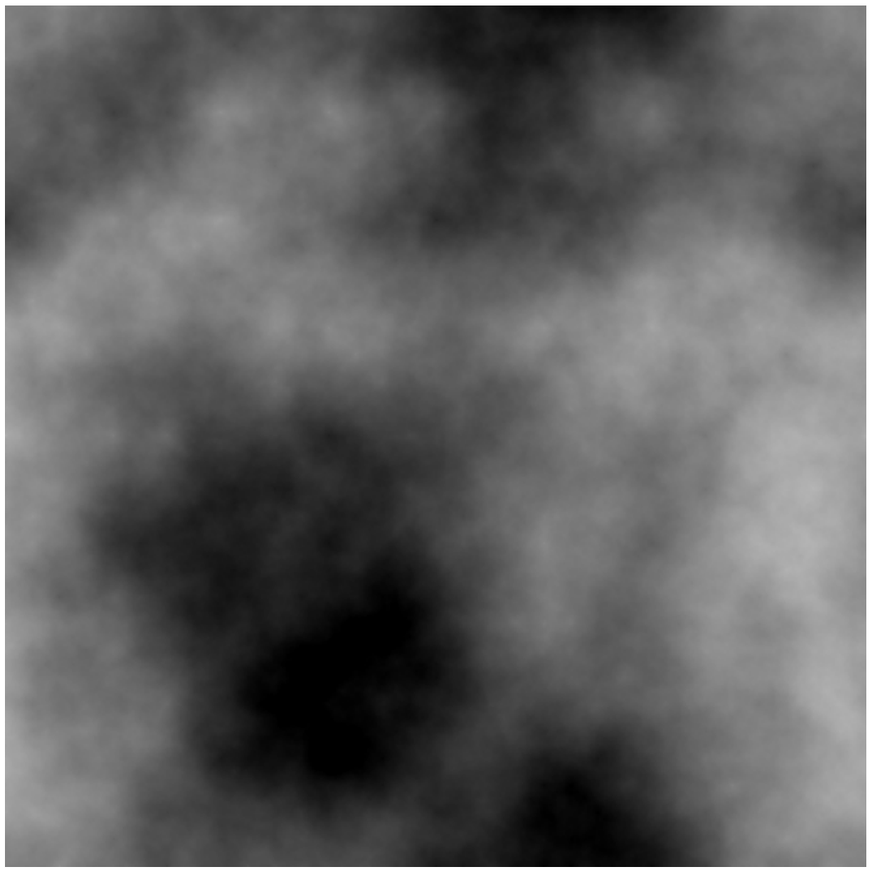
Computational complexity of Diamond-square

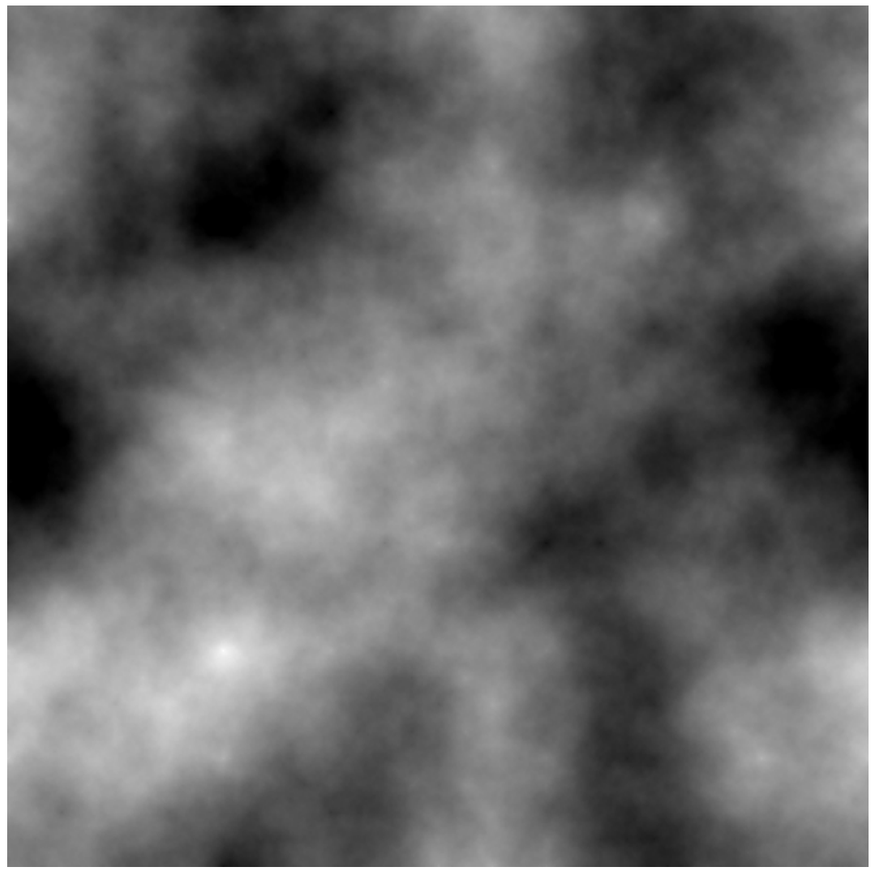
Every point is visited only once!

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

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!

Noise functions

Fractals and noise functions are closely related

Noise can look natural... but when?

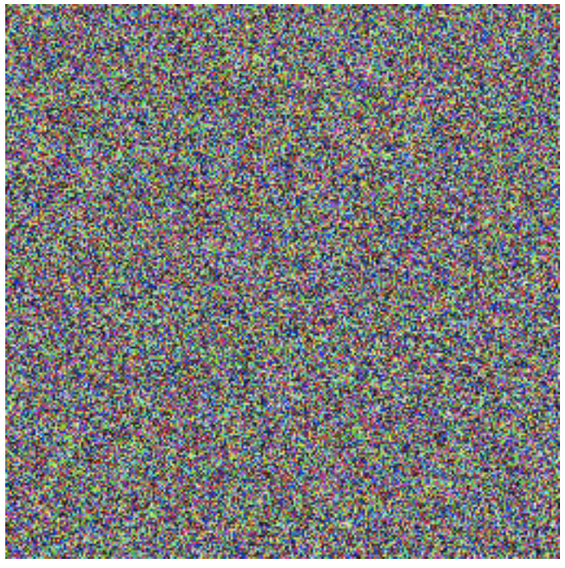
- white noise
- filtered noise
- gradient noise

White noise

Same amplitude in all frequencies

Useless as it is?

Can be processed to something better.

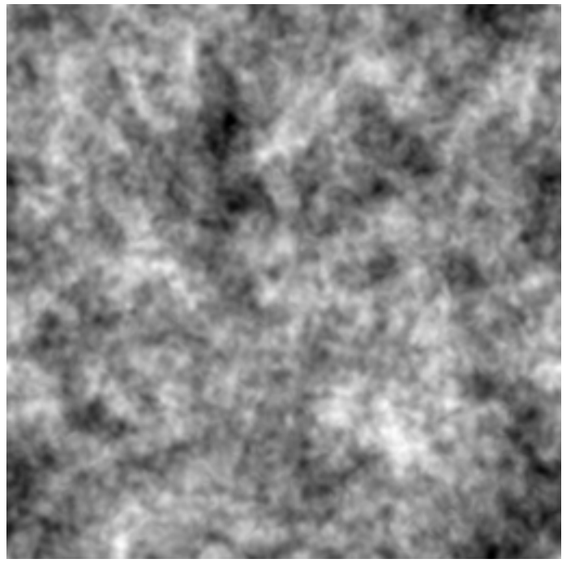


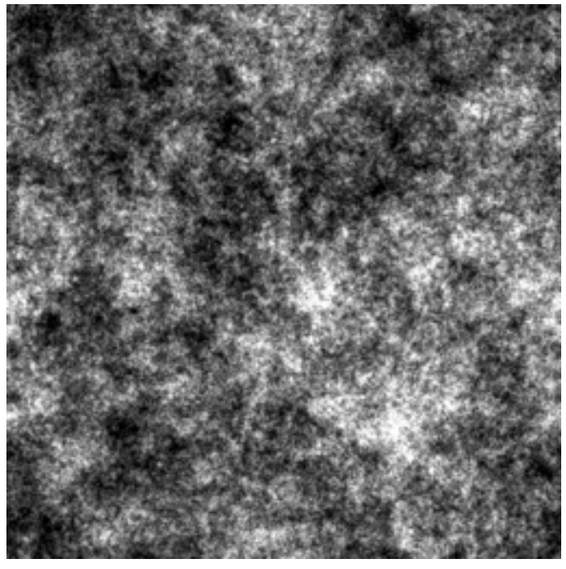
Colored noise

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!)

Therefore other methods became popular: Simplex noise, gradient noise.

Colored noise by filtering in the frequency plane

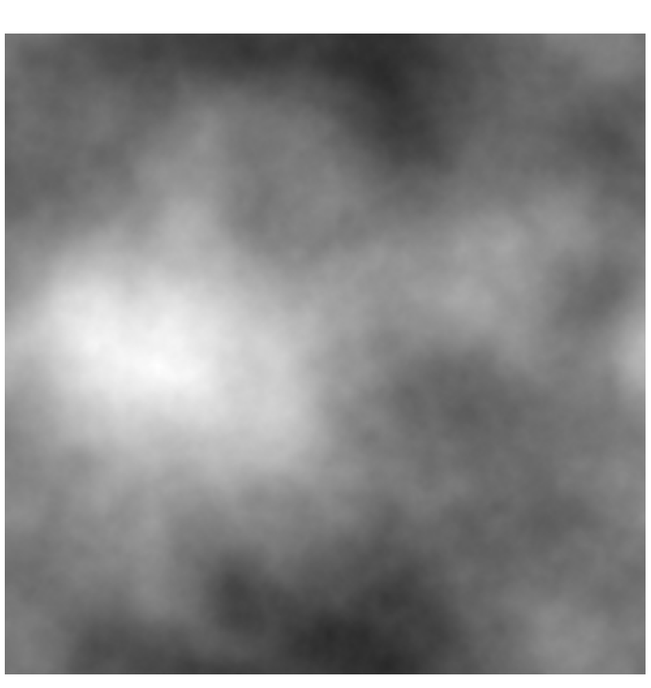
Fill frequency space (2D) with random numbers
(white noise)

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

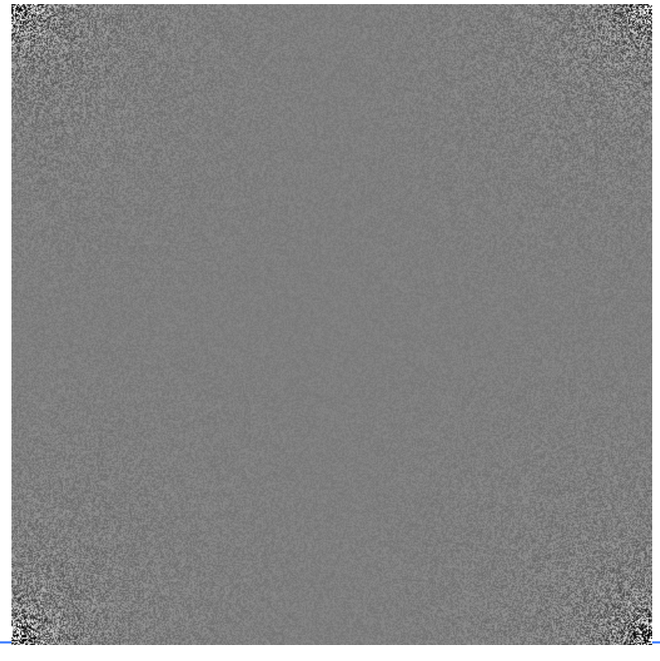
Convert to spatial image with FFT
(FHT, actually, Fast Hartley Transform)

Filter white noise by 1/f

Example



Frequency space:



Advantages of frequency plane filtered noise

Extremely good precision for the frequency behavior, valuable for e.g. water

Repeating patterns, good for textures

FFT/FHT well researched, highly optimized implementations = fast

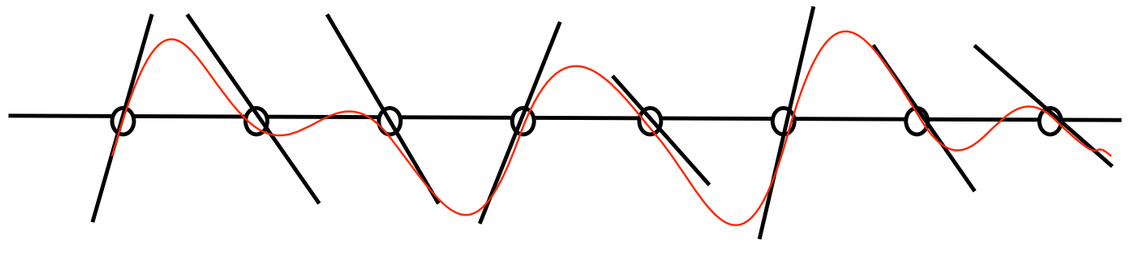
Gradient noise

Band limited noise using random gradients in a grid.

Interpolates between grid points

Most famous: Perlin/gradient noise.

Newer: Simplex noise, PSRD noise



Band limited noise

Not very useful... but combine several frequency bands!

A frequency band is often called an *octave*.

Octave

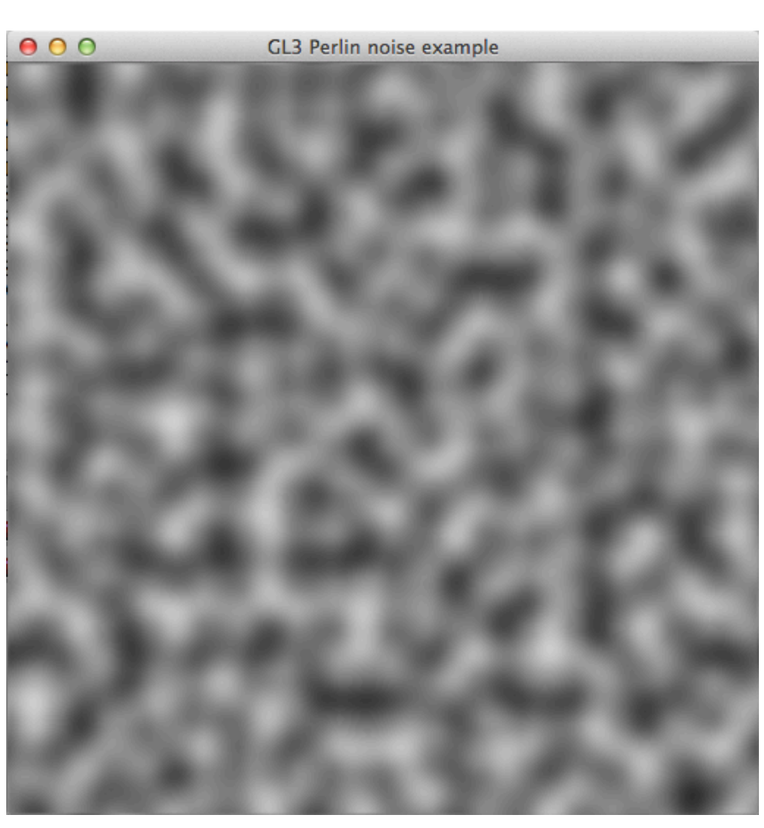
Usually a music term.

One octave = 2x frequency!

One octave is exactly at half of the string length!

Gradient/Perlin noise

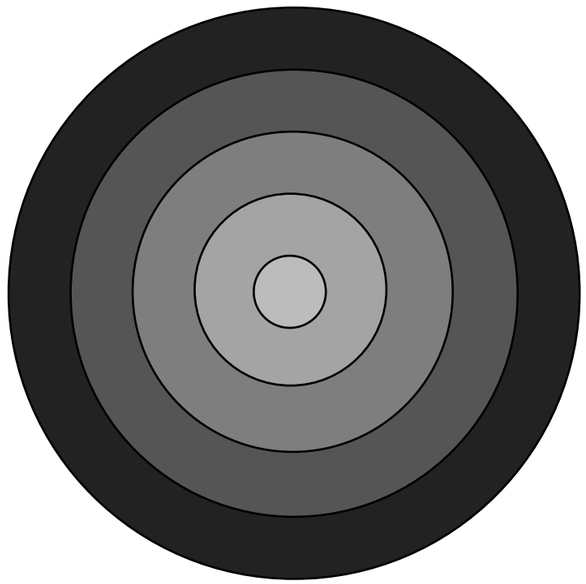
Single octave



Multi-octave Perlin

Multiple "rings" in frequency space

Each ring scaled after the 1/f rule!

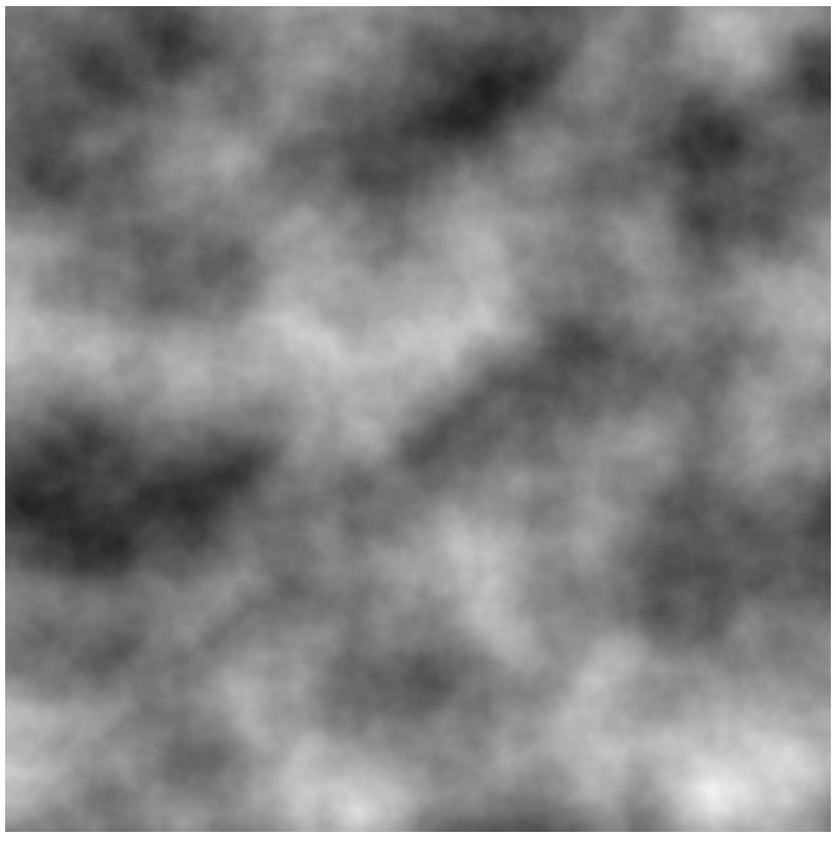


```
void makeperlintexture()
{
   int x, y;
   float i[3];
   char val;

   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 (FHT)

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.

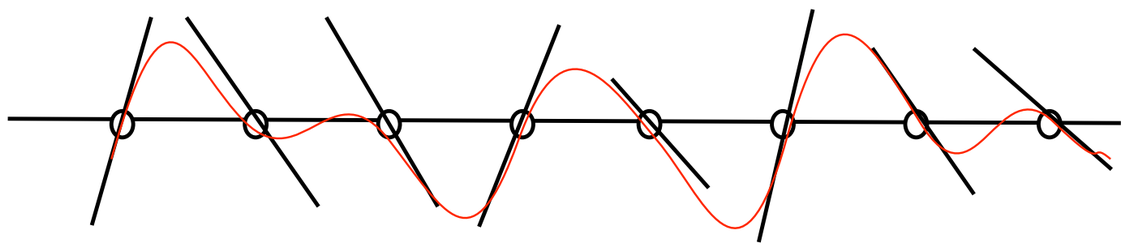
One pass Gradient noise faster... but don’t we need many?

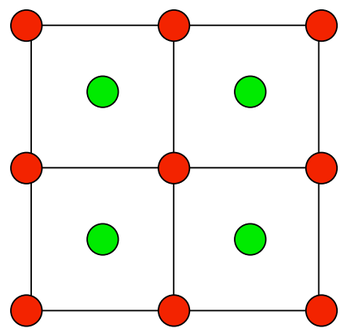
Artifacts

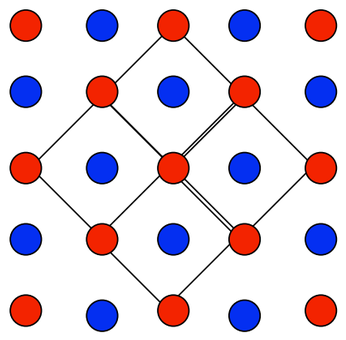
Diamond square and gradient noise are incomplete! They both "lock" in certain points, only producing certain phases of the signal.

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

Water

Etc...

Feature comparison

Scalability: Diamond square is very easy to scale to more detail but need extra fixes to expand to new patches.

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

Frequency plane filtering can not be expanded.

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

Parallelization: Gradient noise independent of neighbors - efficient to parallelize!

Your application needs may decide.

Adapt frequency behavior for gradient noise to other needs

Lacunarity and gain

Gradient noise can be adjusted by changing the frequency variation (lacunarity) as well as the amplitude (gain).

General advice: Stay close to 2 and 1/2!

Complexity comparison

Produce an NxN image

Diamond square: $O(N^2)$

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

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

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

All produce similar results except single octave gradient noise.

Filtered diamond square and FP filter improves quality but raises the constant!

Parallelism

Parallel implementations of Diamond 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!

What more can we do with the terrain?

- Add water, calculate rivers and lakes
- Erosion effects (esp along rivers)
- Roads
- Support caves, steep cliffs and chasms
- 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, even if not having the lowest computational complexity!

Other methods are better for repeating textures.