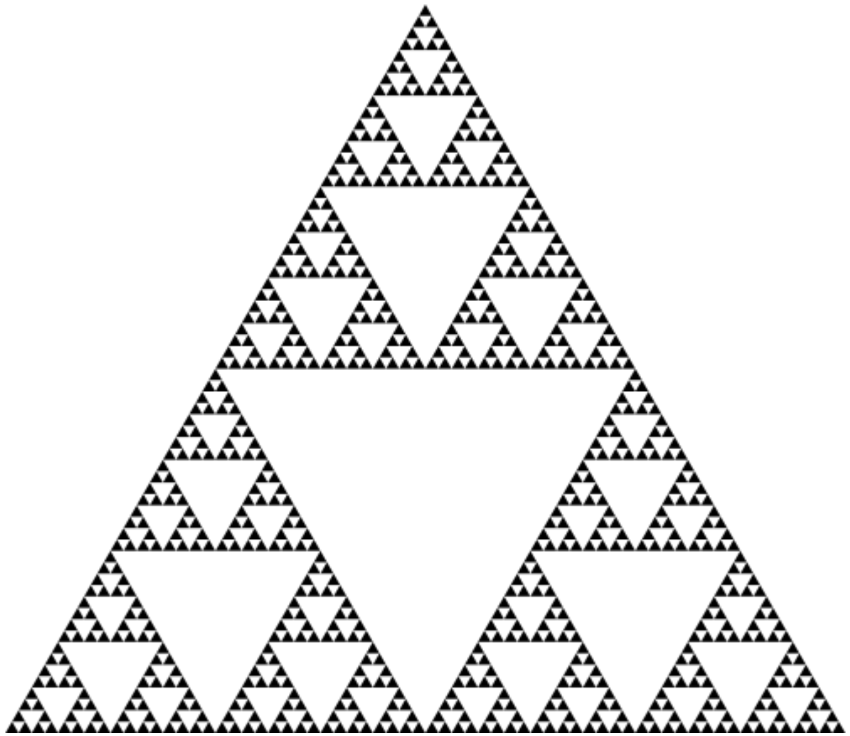
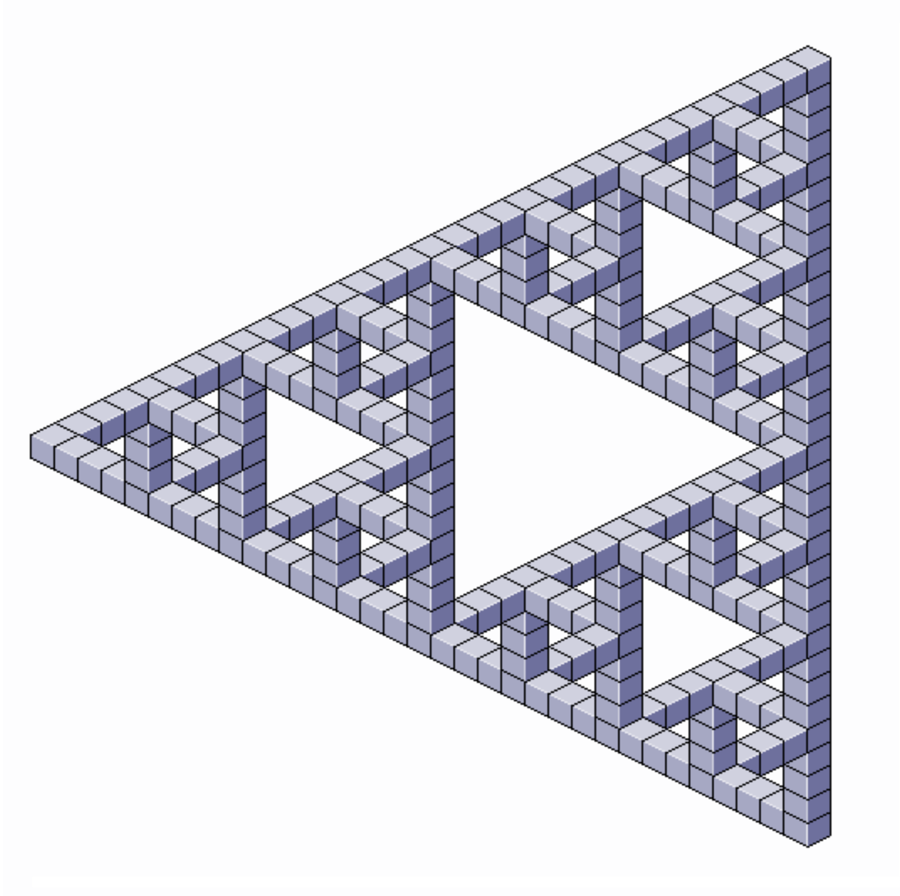
Computer Graphics

Procedural Methods - Noise & Terrain

Mark Pauly

Geometric Computing Laboratory





source

Procedural Techniques

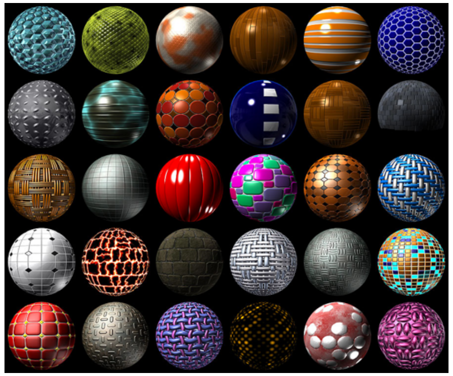
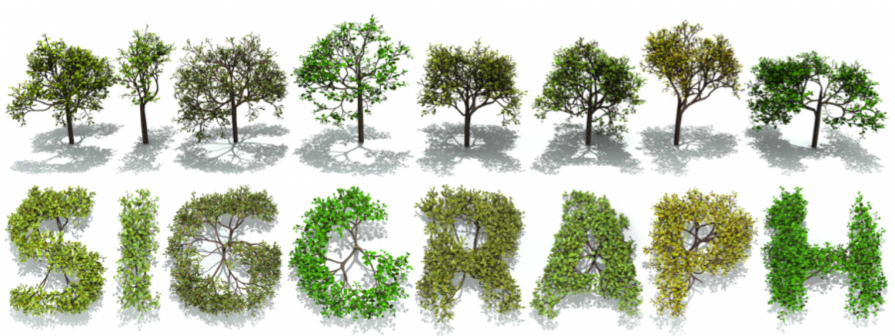
- Algorithms, functions, code segments that generate computer graphics objects
  - textures
  - geometry
  - reflection models
  - motion
  - etc.
- Program code vs. data

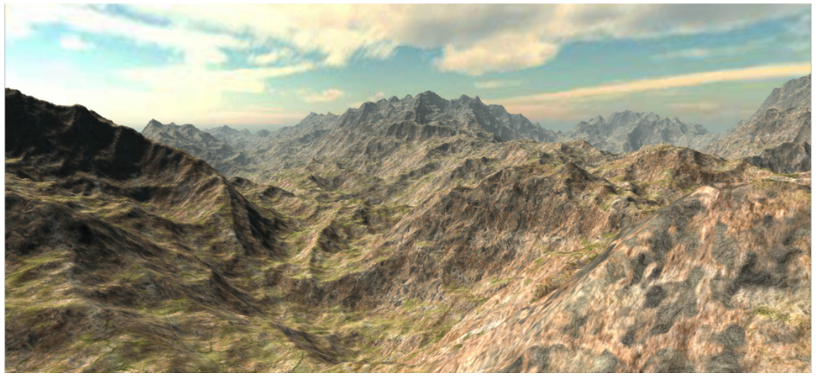
Procedural Techniques

- Why?
  - abstraction
  - automatic generation
  - compact representations
  - infinite detail
  - parametric control
  - flexibility
- Particularly suitable for models resulting from processes that are repeating, self-similar, or random

Procedural Techniques

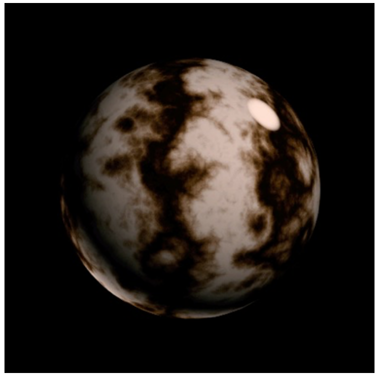
- Ubiquitous in graphics
    - texturing, modeling, animation, etc.




Overview

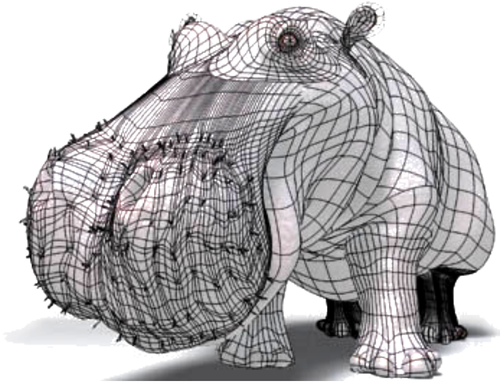
- Today:
  - noise functions
  - texture & terrain synthesis



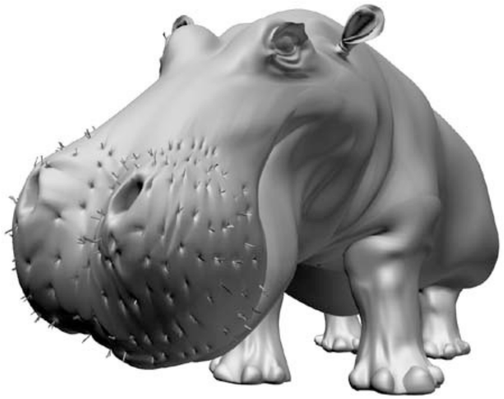
- Later:
  - procedural modeling with L-Systems
  - basic plant modeling

Materials & Texture

- Recall: textures add visual detail without raising geometric complexity



Geometry



+Lighting

+Texture

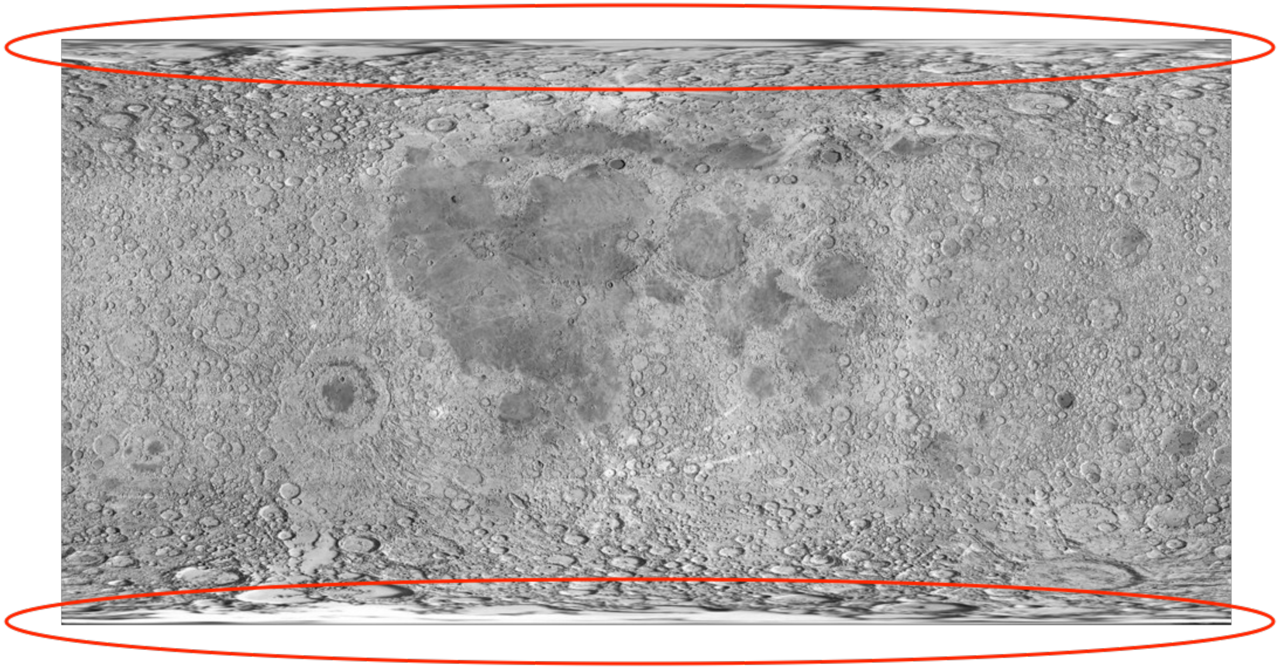
Images from http://www.3drender.com/jbirn/productions.html

Materials & Texture

- Control much more than just colors:
  - reflectance (diffuse + specular colors/coefficients)
  - normal vector (normal mapping, bump mapping)
  - geometry (displacement mapping)
  - opacity (alpha mapping)
  - reflection/illumination (environment mapping)
  - ...

Solid Textures

- Often it's better to attach 3D, volumetric textures
  - Avoid surface texture distortion due to parametrization



  - Assign consistent material inside the object too (e.g., for fracture)

Solid Textures

How Do We Acquire Textures?

- Photograph/scan materials
- Manually paint
- Download online
- …

Material Acquisition via Scanning

- More difficult than just taking a picture
  - Must factor out lighting effects
  - Post-process to extract normal maps, ensure tiling, etc.
- Limited by scanner size

More Problems with Acquired Textures

- Physical extent limited by storage size
  - Particularly problematic for solid textures…
- Repeating to fill more space causes visible artifacts:

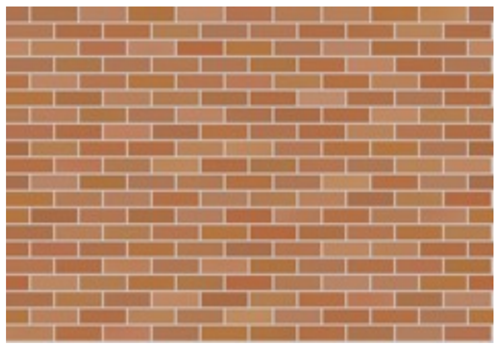
Blender Stack Exchange

Procedural Approach

- Instead of using image data, define the texture with code.
  - Simple example:

$$\text{color} = \text{vec3}(0.5 * \sin(x) + 0.5)$$

  - Trivial extension to solid textures…
- Easily create repetitive patterns:

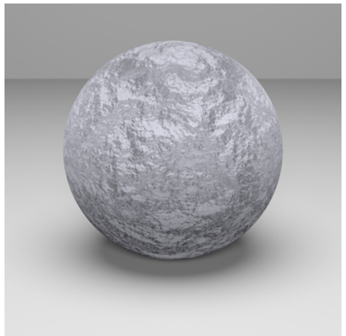



Procedural Approach

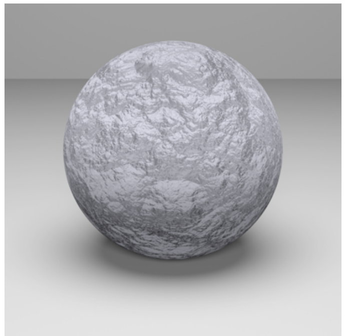
- Instead of using image data, define the texture with code.
  - Simple example:

$$\text{color} = \text{vec3}(0.5 * \sin(x) + 0.5)$$

  - Trivial extension to solid textures...
- Easily create repetitive patterns
- We'll see how to create patterns with structured randomness:



fBm



turbulence

Procedural Approach

- Why?
  - automatic generation on the fly
  - compact representations
  - infinite detail
  - unlimited extent
  - parametric control
- Particularly suitable for models resulting from processes that are repeating, self-similar, or random
- Challenges: artistic control, debugging, efficiency

Procedural Synthesis Examples

Created using Terragen

Procedural Synthesis Examples

Created using Terragen

Procedural Synthesis Examples

Created using MojoWorld Generator

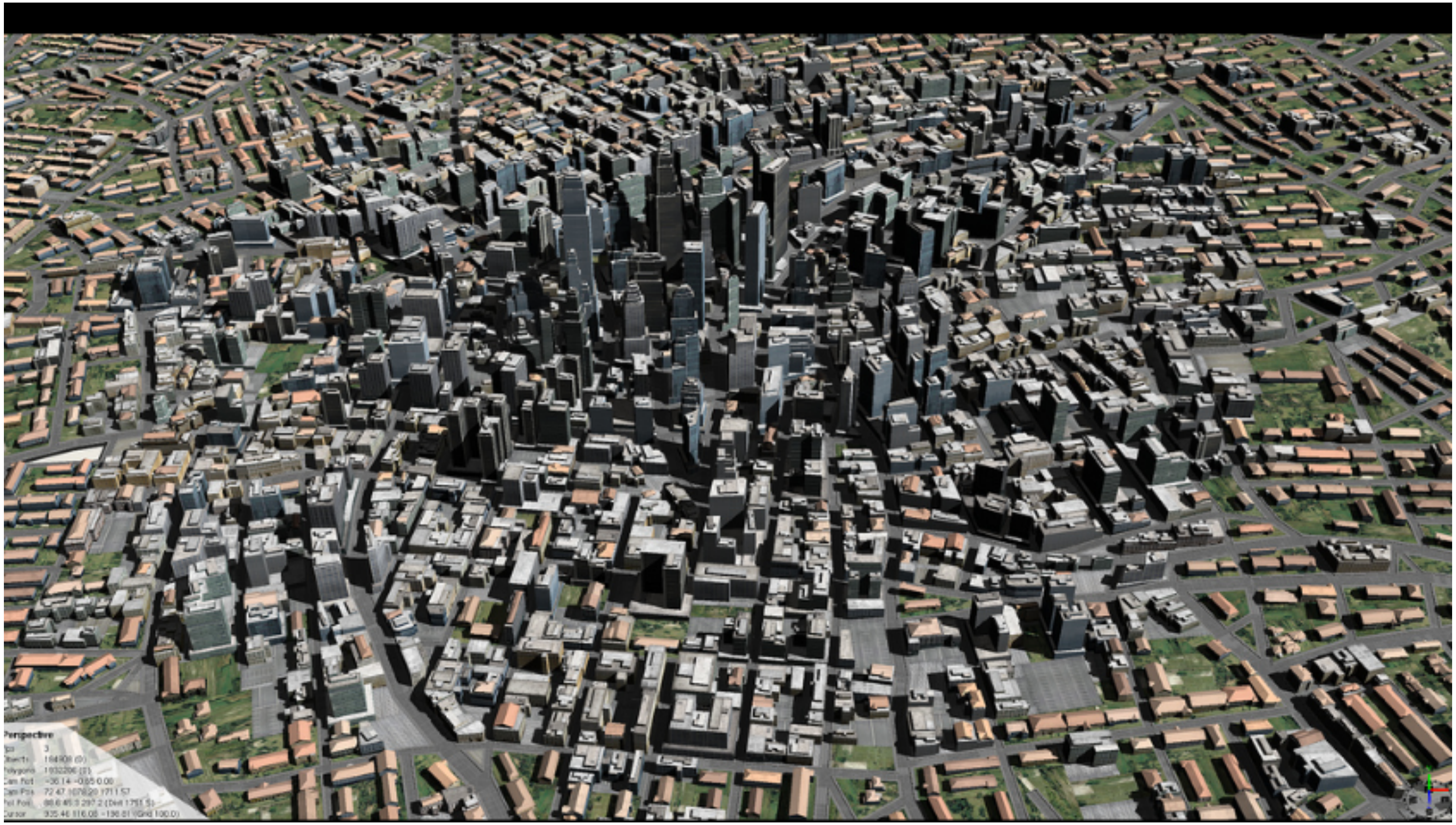
Procedural Synthesis Examples

Created using Vue Infinite

Procedural Synthesis Examples

Created using Vue Infinite

Procedural Synthesis Examples



Created using Esri CityEngine

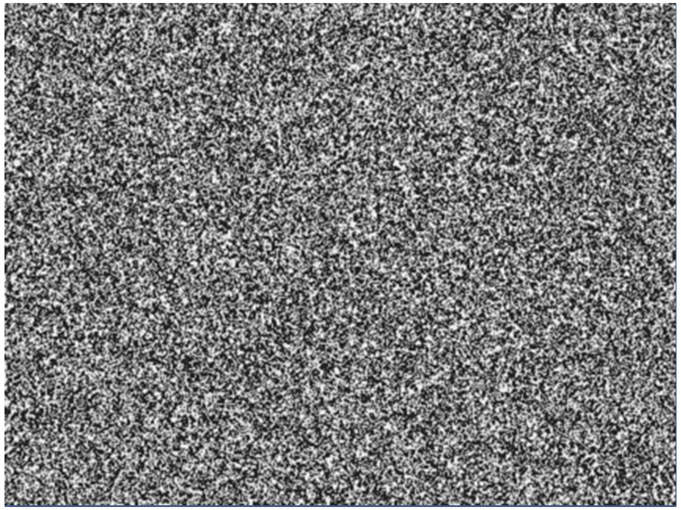
How to Model a Mountain Terrain?

- Simulate the complex physical process that created it?
- Mimic its qualitative features?

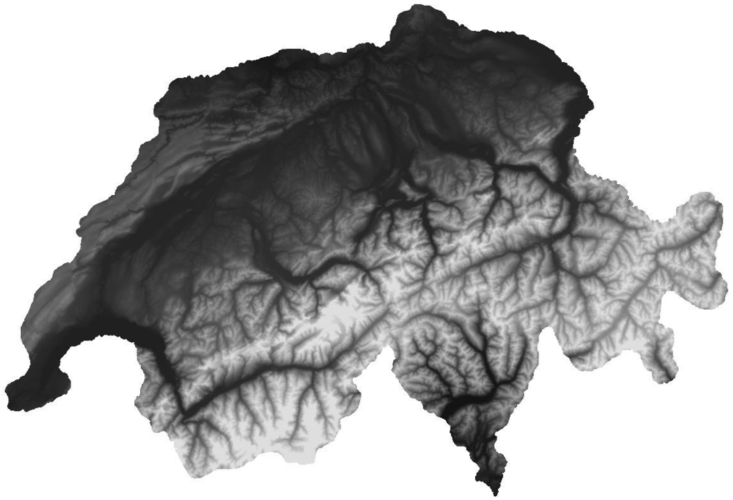
wikipedia

Randomness

- Computers are good at faking randomness
- But randomness alone isn't what we want



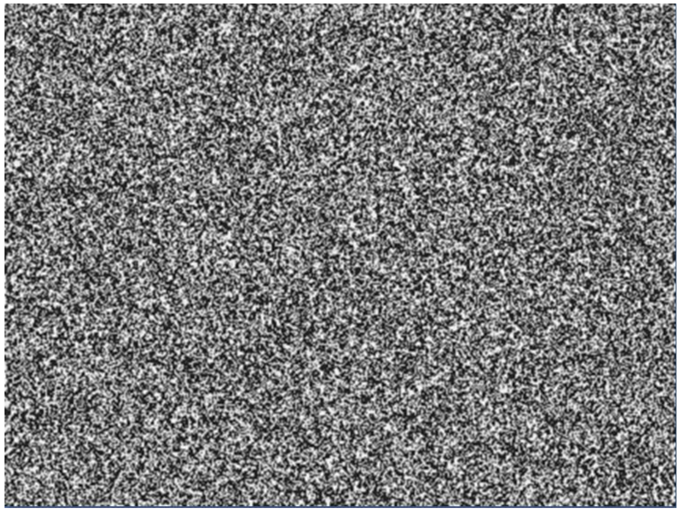
white noise



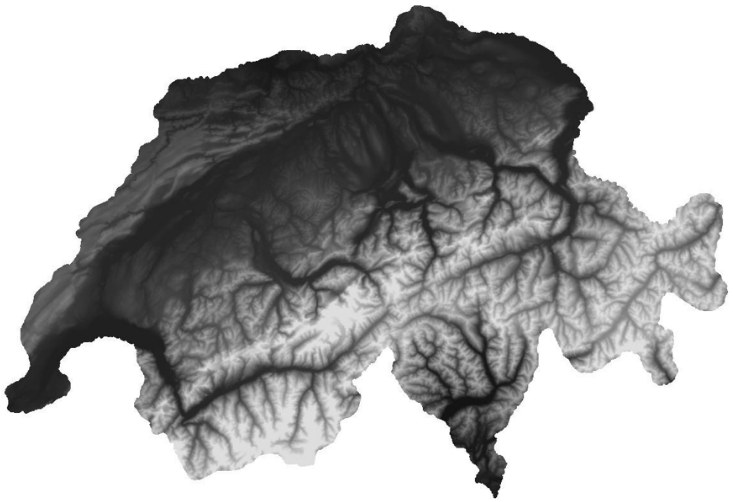
more natural pattern

Problems with Pure Randomness

- Neighboring samples are uncorrelated
  - Natural phenomena lead to more structure
- Get a different result every time
  - When an artist finishes setting up a scene, they don't want it to change.



white noise



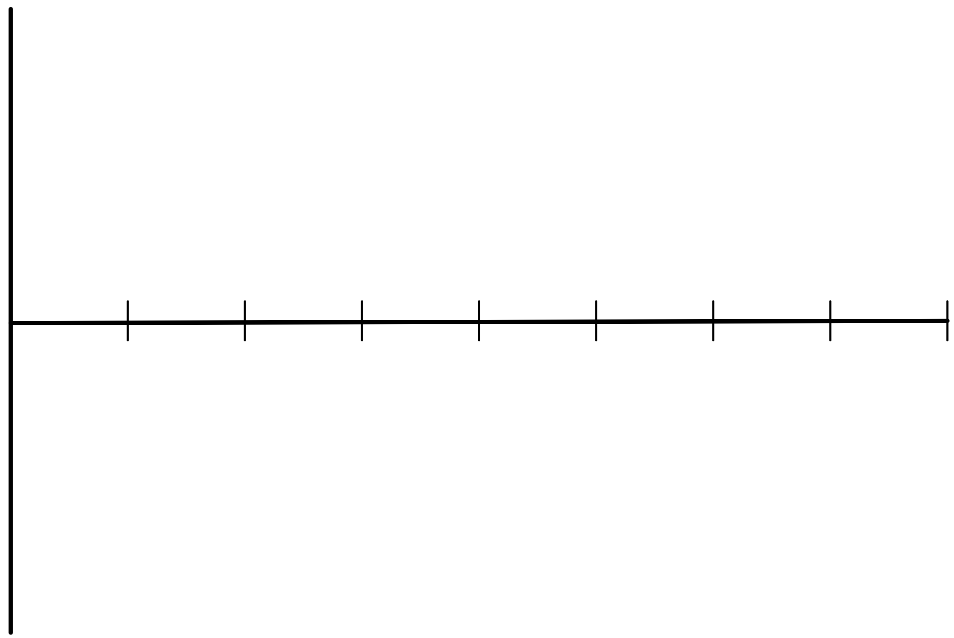
more natural pattern

Noise Functions

- Function $\mathbb{R}^n \rightarrow [-1, 1]$, where $n = 1, 2, 3 \ldots$
- Desirable properties
  - No obvious repetition
  - Rotation invariance
  - band-limited
    - frequencies stay finite
    - more structure than white noise
  - efficient to compute
  - reproducible
- Fundamental “primitive” or building block of most procedural synthesis approaches

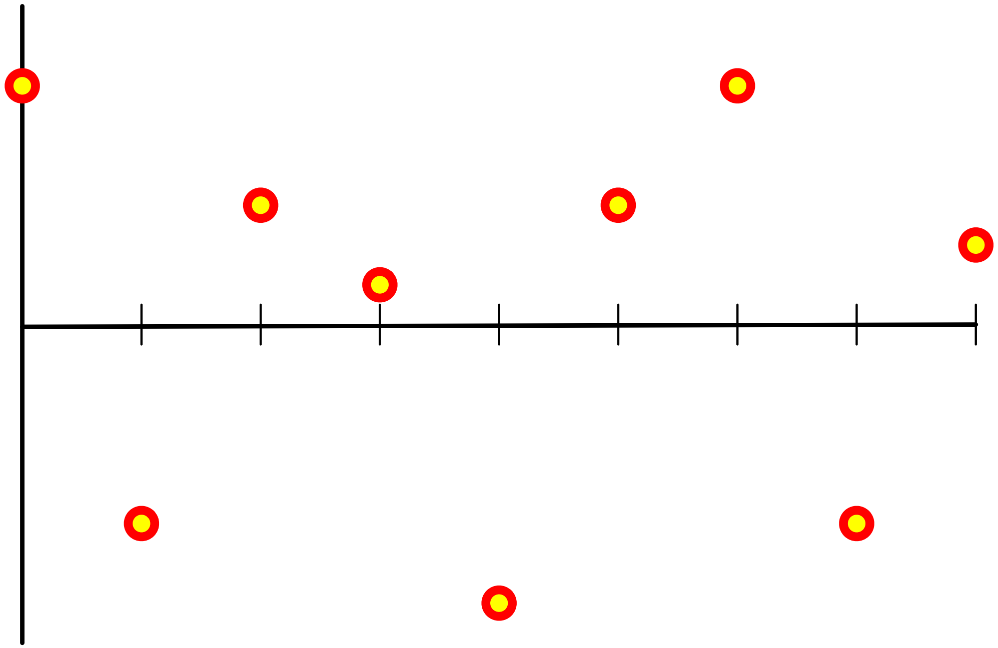
Noise Functions

- Simple example: value noise
  - Generate random value on the grid points of an integer lattice
  - Interpolate these values throughout the grid



Noise Functions

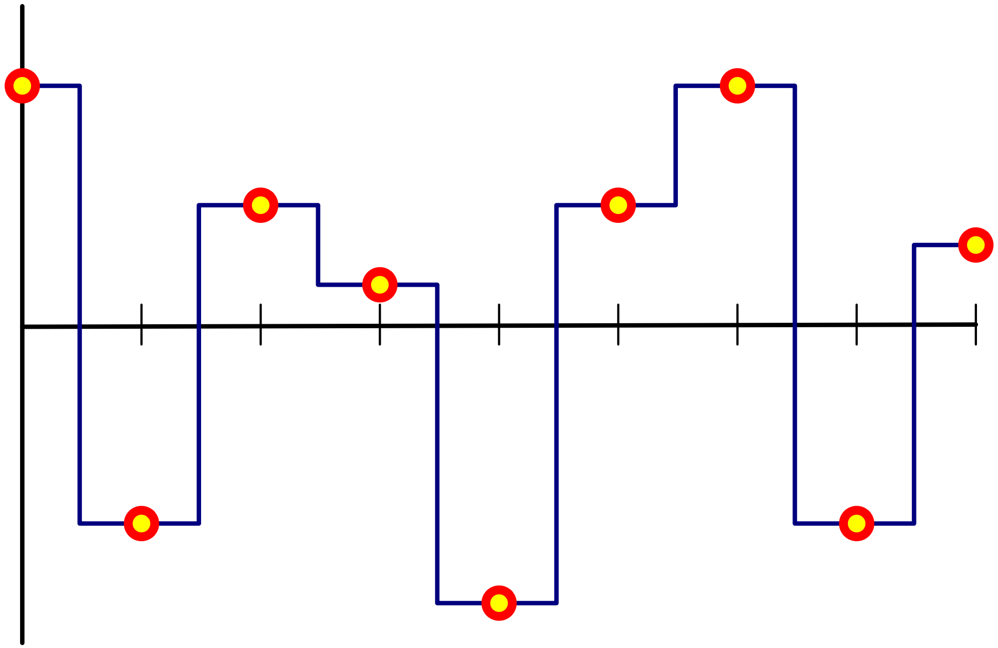
- Simple example: value noise
  - Generate random value on the grid points of an integer lattice
  - Interpolate these values throughout the grid



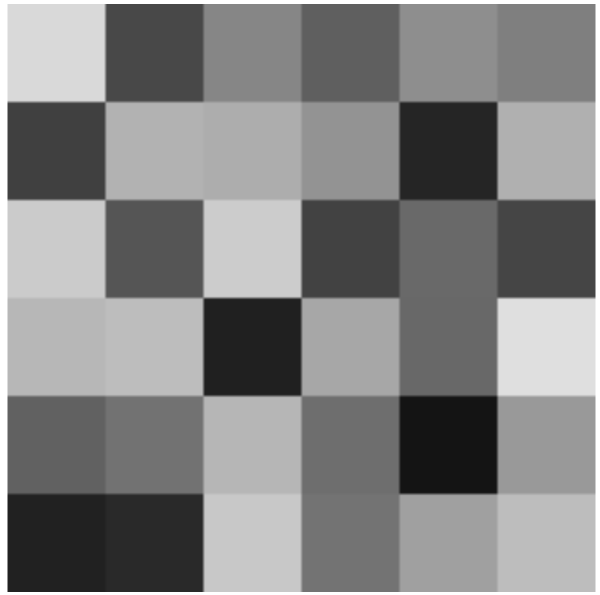
random values on the grid

Noise Functions

- Simple example: value noise
  - Generate random value on the grid points of an integer lattice
  - Interpolate these values throughout the grid

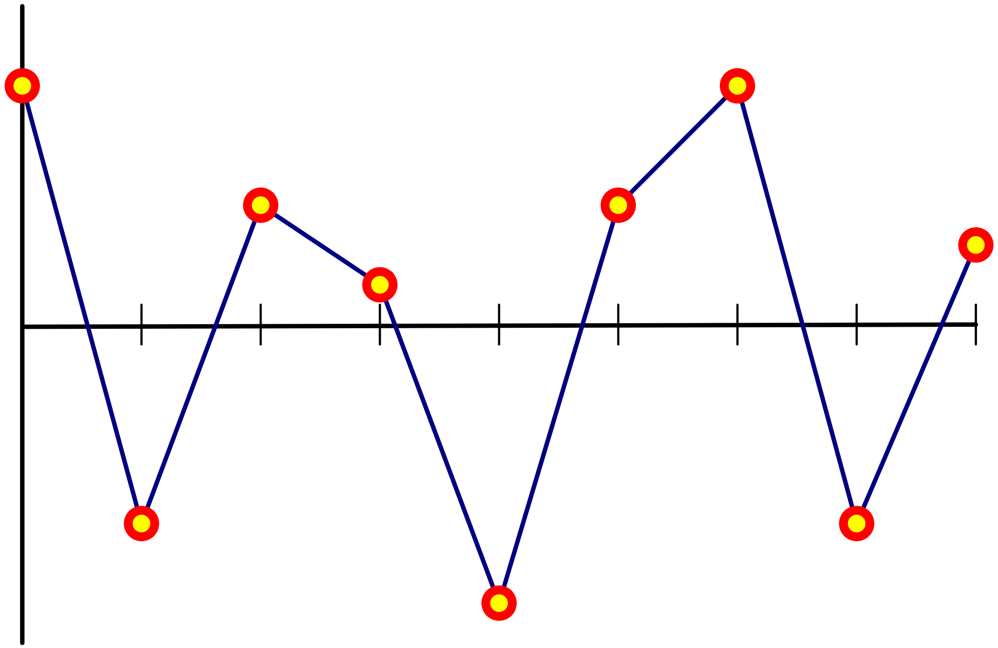


piecewise constant interpolation (nearest)

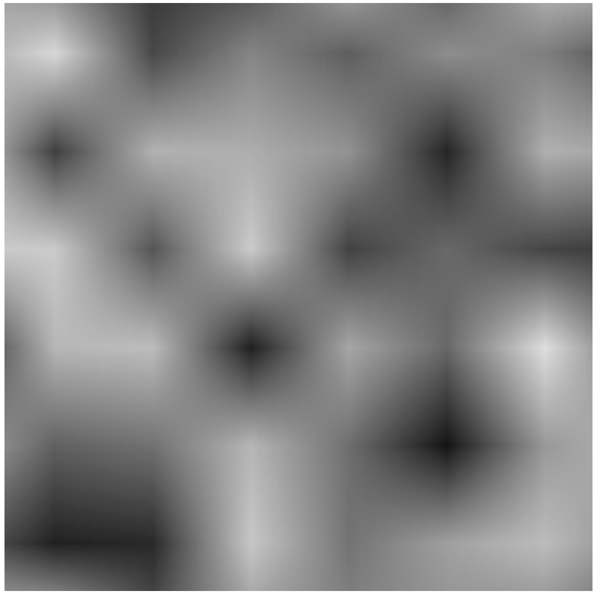


Noise Functions

- Simple example: value noise
  - Generate random value on the grid points of an integer lattice
  - Interpolate these values throughout the grid

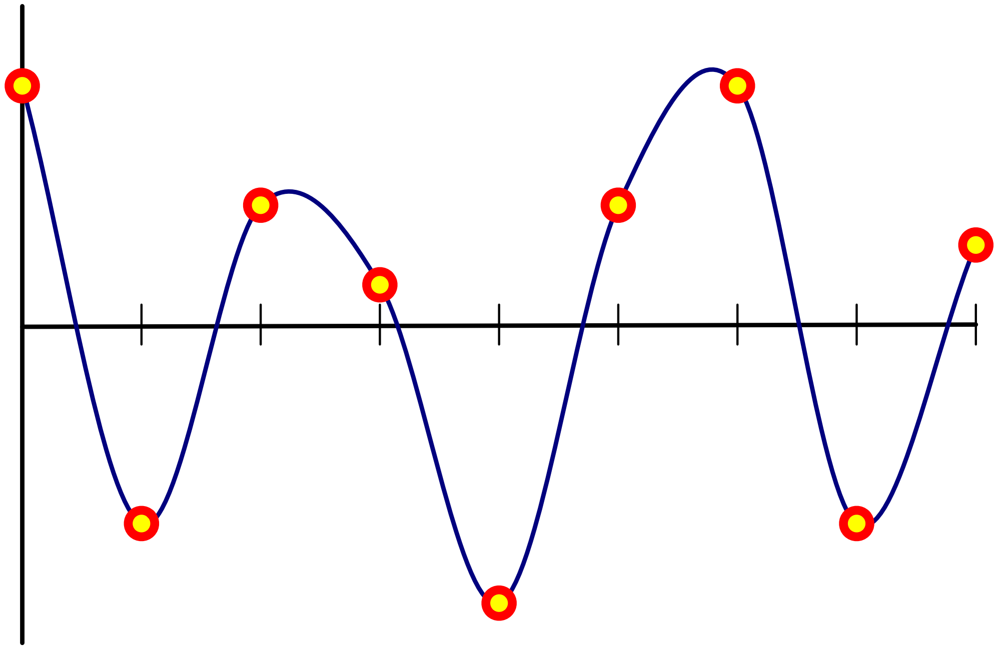


piecewise (bi-, tri-)linear interpolation



Noise Functions

- Simple example: value noise
  - Generate random value on the grid points of an integer lattice
  - Interpolate these values throughout the grid



piecewise cubic interpolation

highest frequency is limited by the lattice resolution!

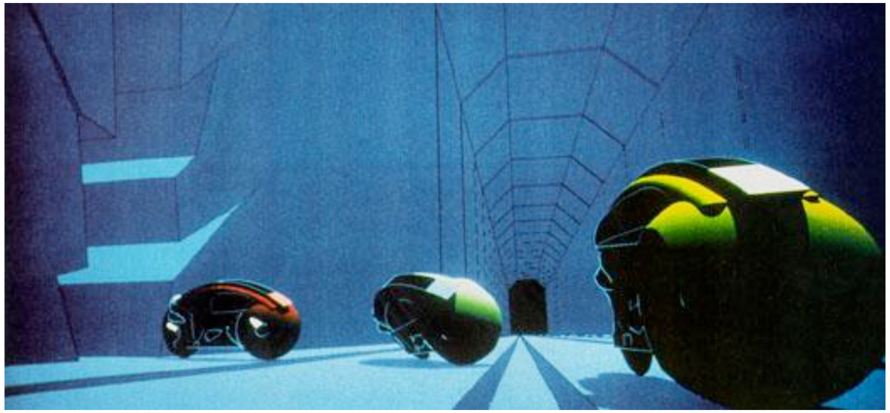
Value Noise Issues

1. Cubic looks best (most organic), but it is expensive
   - Linear interpolation combines the 2^n nearest lattice values
   - Cubic interpolation combines the 4^n nearest lattice values…
2. Repeatability
   - New random numbers every time you regenerate the values!
3. Memory use
   - Cannot store an infinite number of random grid values

- Solution to 2 & 3:
  - Pre-compute a table of ~512 random values
  - Use a hash function to map lattice locations to table indices

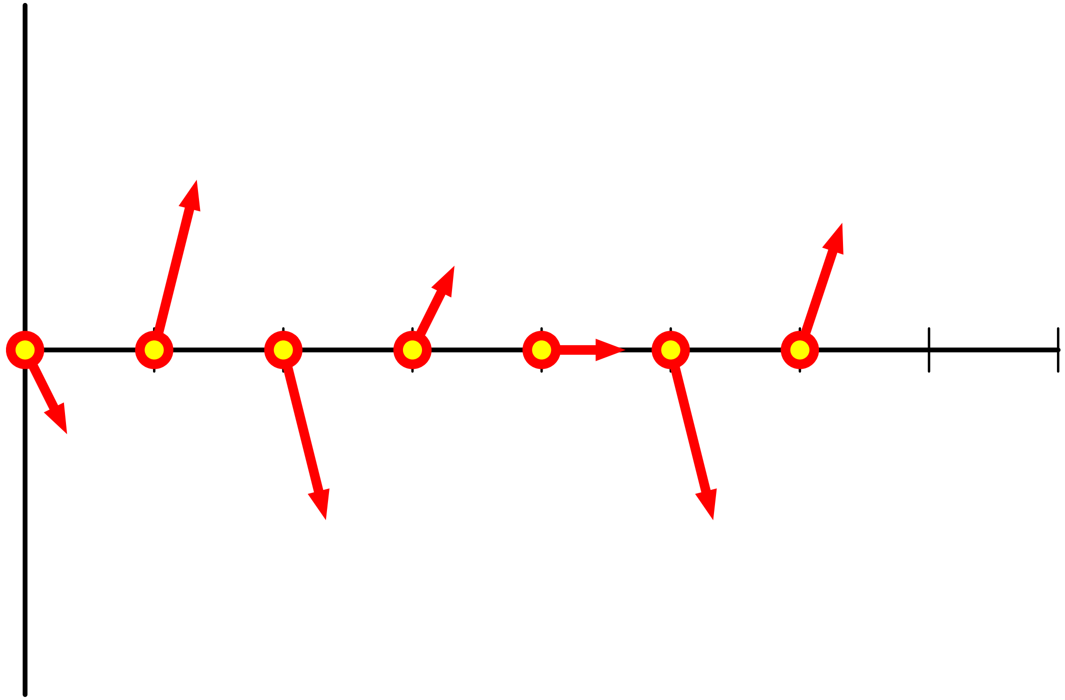
Perlin Noise

- Invented by Ken Perlin in 1982
  - First used in the movie Tron
- Also called gradient noise



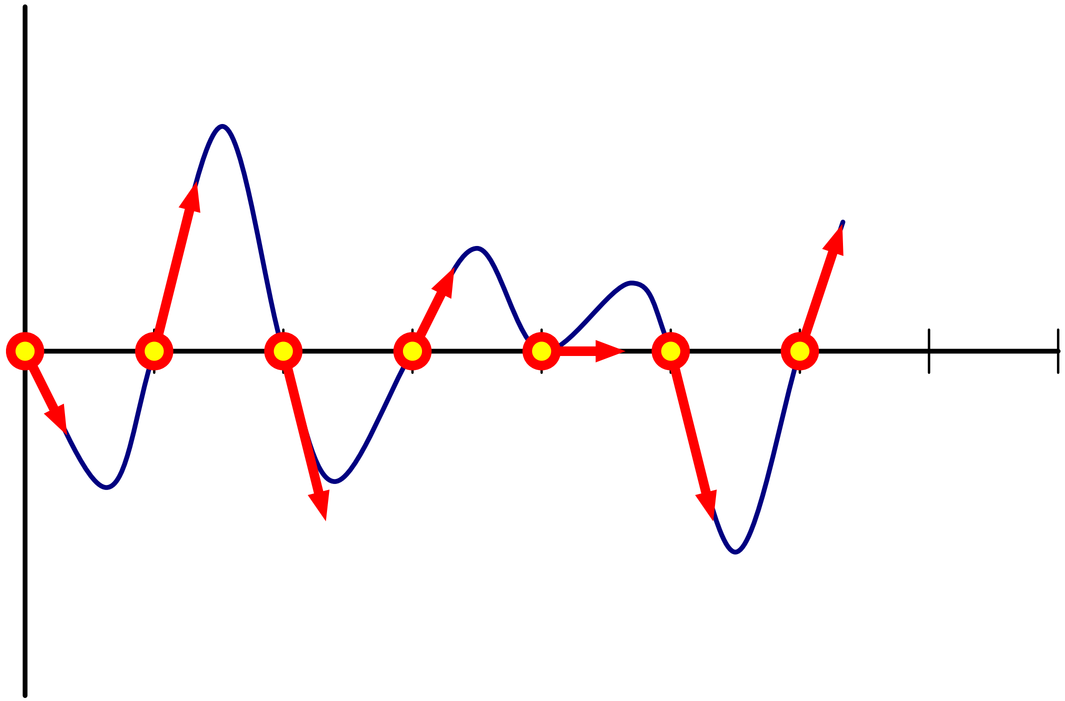
Classic Perlin Noise (1980s)

- Generate random *gradients* on the grid:

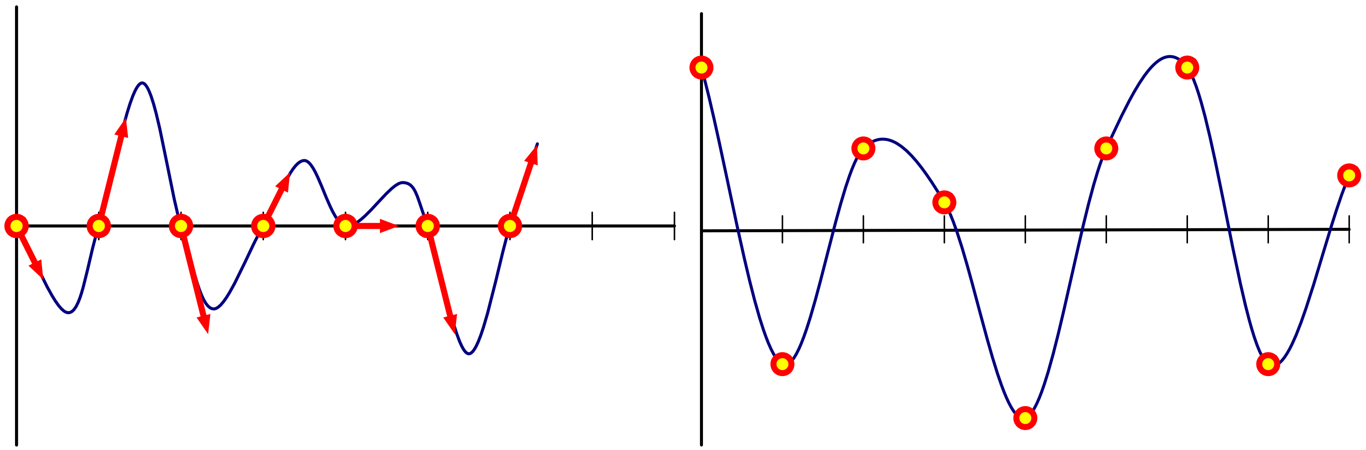


Classic Perlin Noise (1980s)

- Interpolate these gradients with Hermite interpolation



Perlin Noise vs Cubic Value Noise

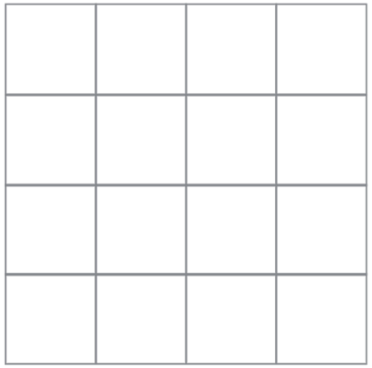


Perlin (Gradient) Noise *Cubic Value Noise*

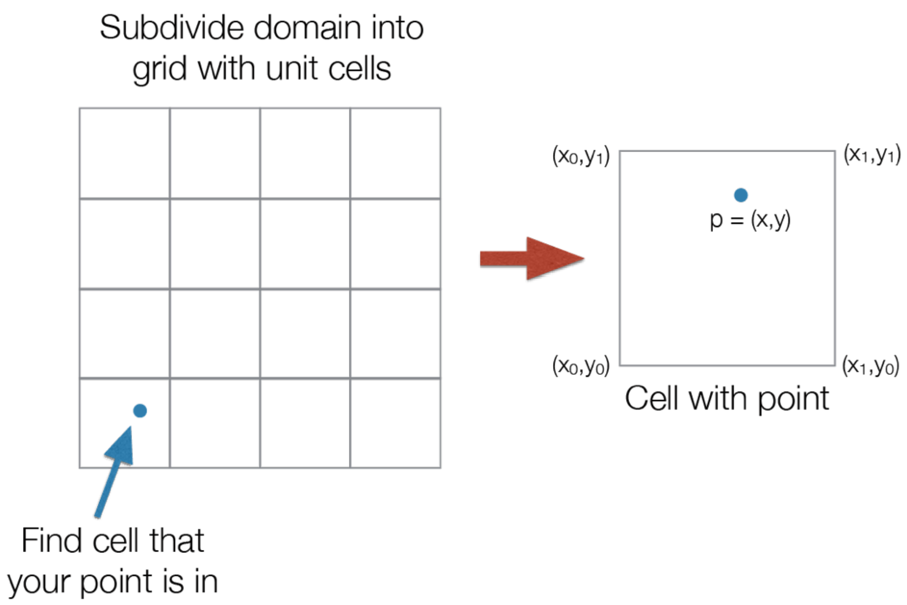
- Advantage of Perlin Noise: efficiency
  - Get cubic interpolation with only 2^n nearest gradients, not 4^n values
- Potential downside
  - Value at grid location are always zero
  - To overcome this, can combine gradient and value noise: generate gradient and value sample for each lattice point and use Hermite interpolation.

2D Perlin Noise Example

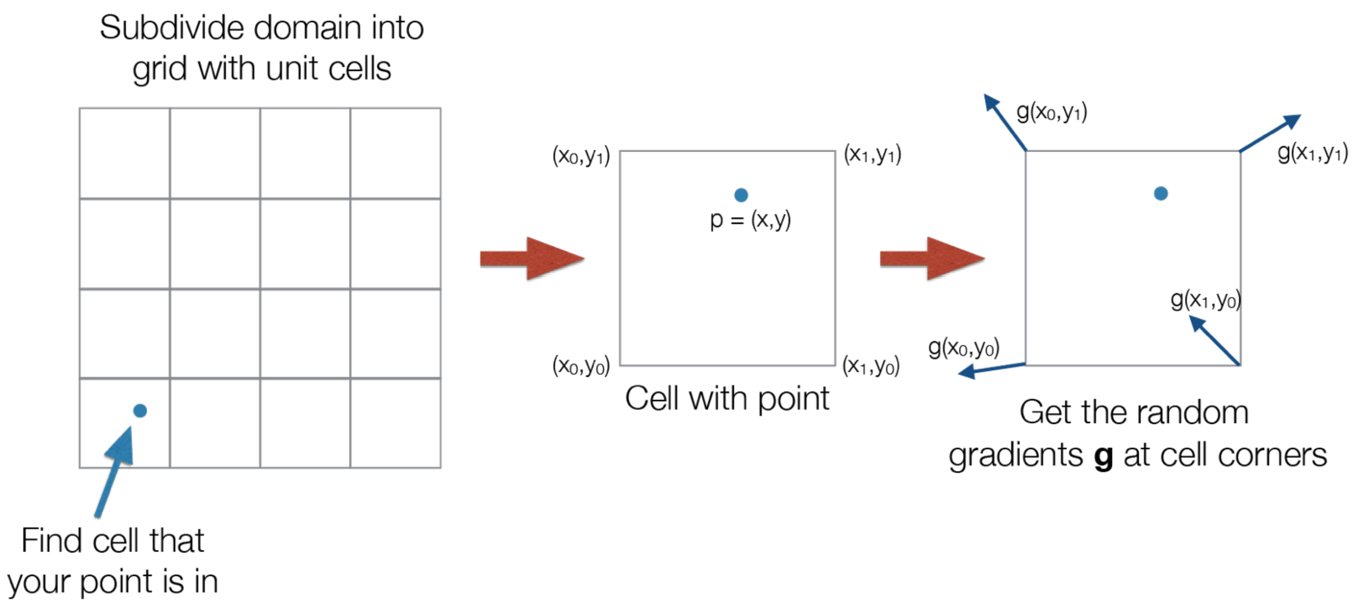
Subdivide domain into
grid with unit cells



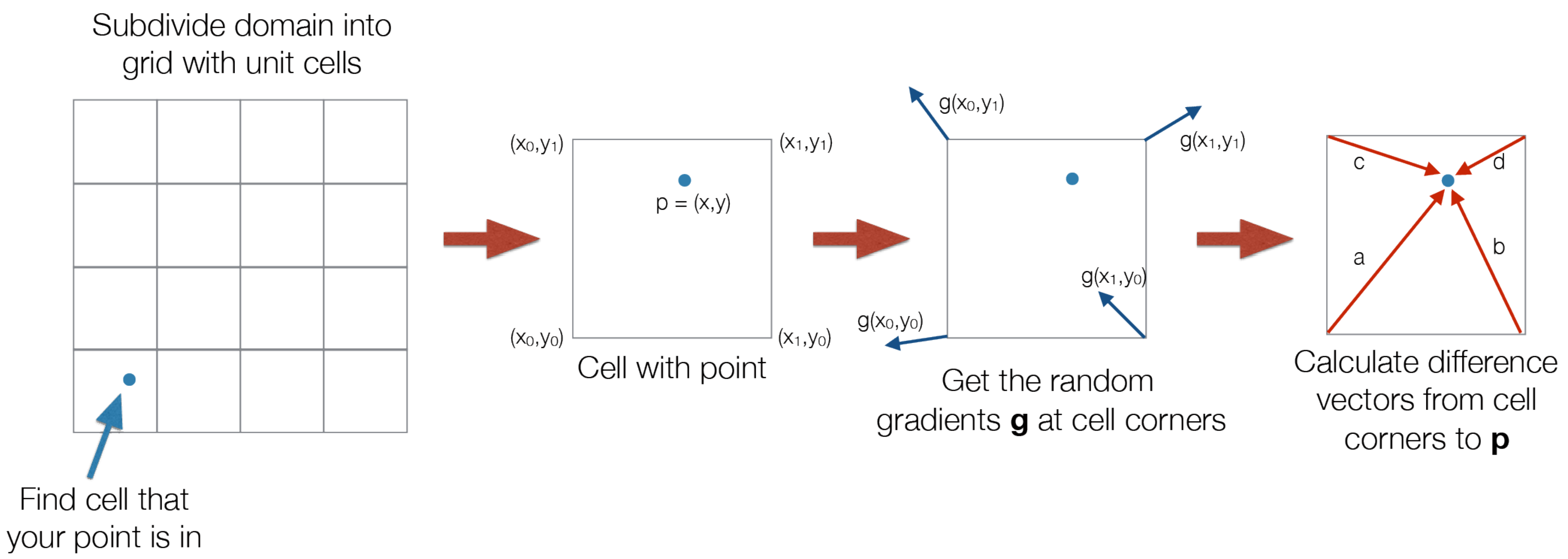
2D Perlin Noise Example



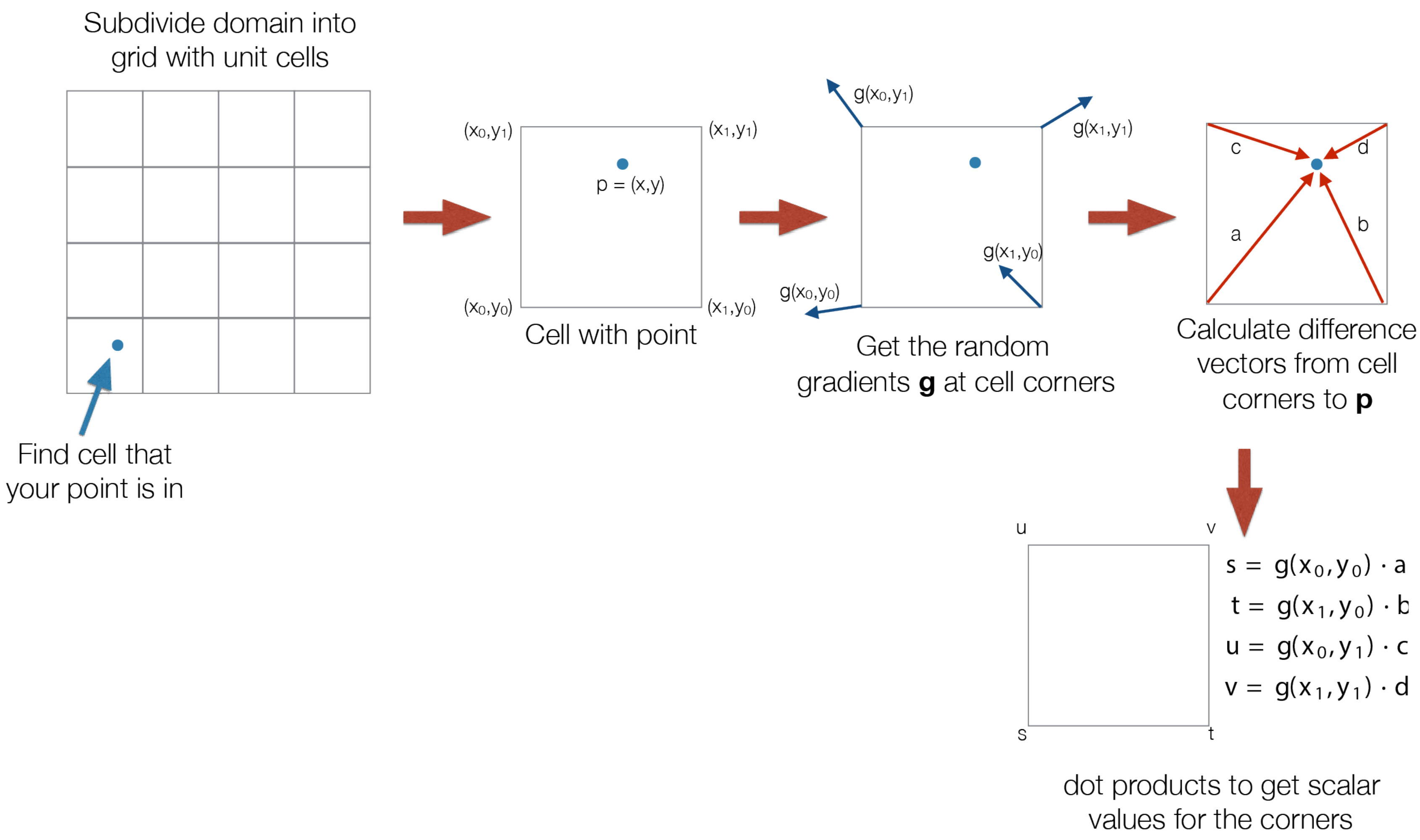
2D Perlin Noise Example



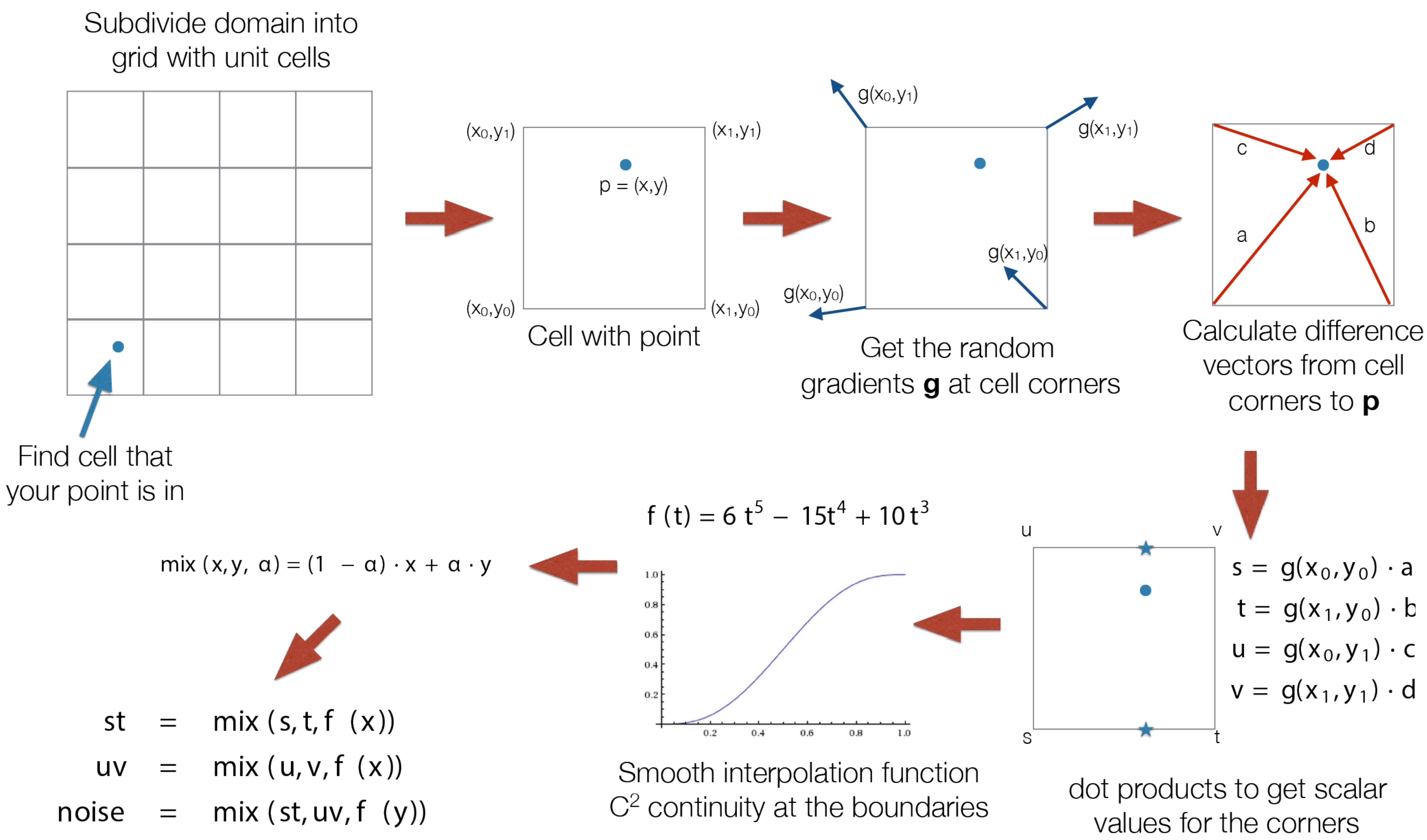
2D Perlin Noise Example



2D Perlin Noise Example

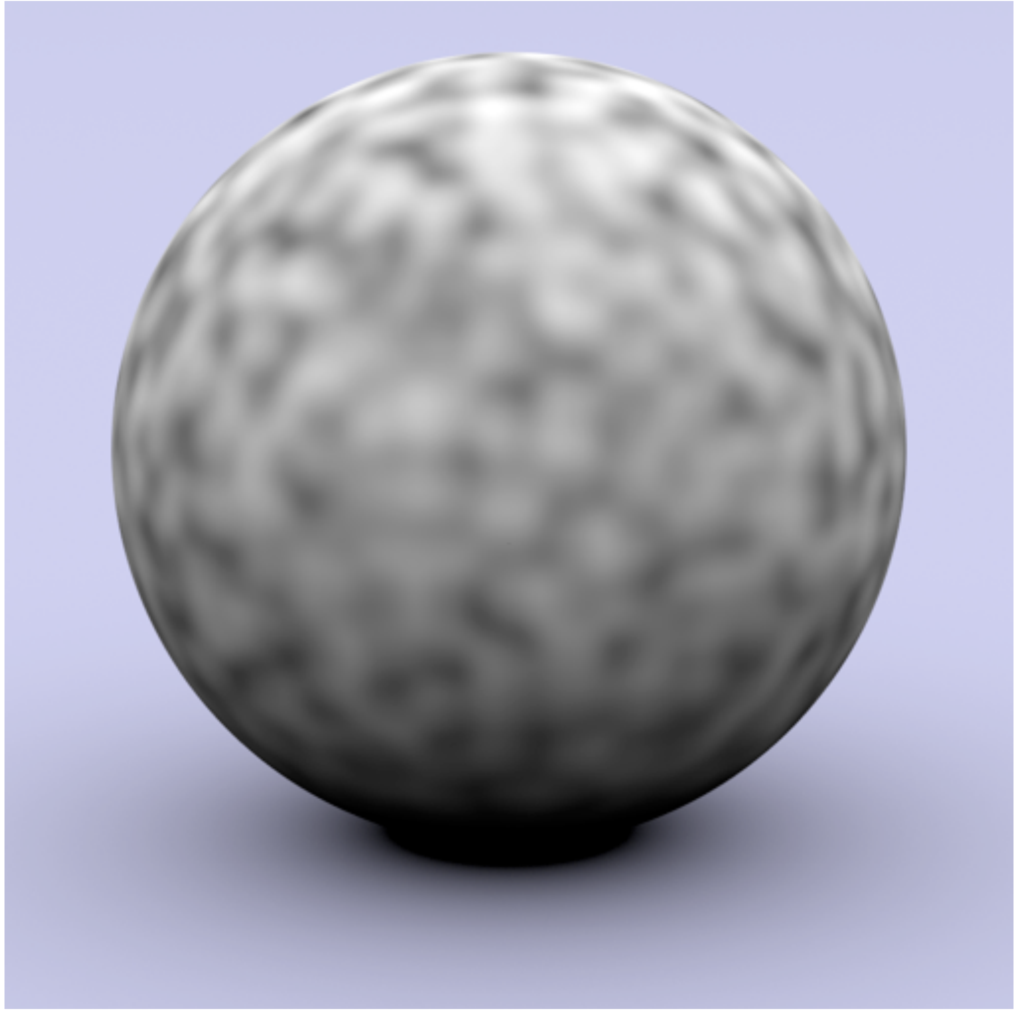


2D Perlin Noise Example



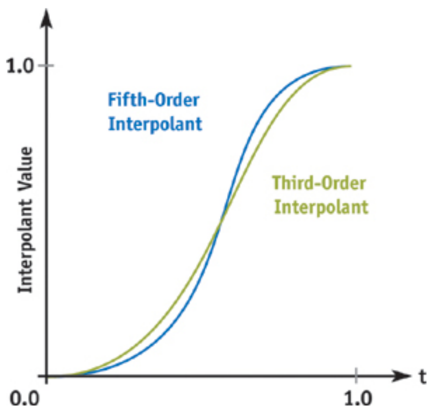
2D Perlin Noise

3D Perlin Noise



Classic vs. Improved Perlin Noise

- Short 2002 paper improving efficiency/visual quality
- New version:
  - Randomly chose from only 12 pre-defined gradient vectors, (Human vision is sensitive to statistical orientation anomalies, but not the orientation granularity)
  - Interpolate the corners' linear functions with $6t^5 - 15t^4 + 10t^3$ instead of $3t^2 - 2t^3$ (avoid discontinuities in second derivative)



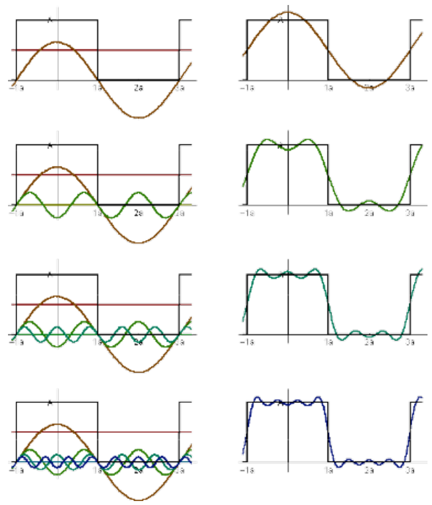
Improved Perlin Noise Implementation

```
// JAVA REFERENCE IMPLEMENTATION OF IMPROVED NOISE - COPYRIGHT 2002 KEN PERLIN.

public final class ImprovedNoise {
   static public double noise(double x, double y, double z) {
      int X = (int)Math.floor(x) & 255,                  // FIND UNIT CUBE THAT
          Y = (int)Math.floor(y) & 255,                  // CONTAINS POINT.
          Z = (int)Math.floor(z) & 255;
      x -= Math.floor(x);                                // FIND RELATIVE X,Y,Z
      y -= Math.floor(y);                                // OF POINT IN CUBE.
      z -= Math.floor(z);
      double u = fade(x),                                // COMPUTE FADE CURVES
             v = fade(y),                                // FOR EACH OF X,Y,Z.
             w = fade(z);
      int A = p[X  ]+Y, AA_ = p[A]+Z, AB_ = p[A+1]+Z,      // HASH COORDINATES OF
          B = p[X+1]+Y, BA_ = p[B]+Z, BB_ = p[B+1]+Z;      // THE 8 CUBE CORNERS,

      return lerp(w, lerp(v, lerp(u, grad(p[AA_  ], x  , y  , z   ),  // AND ADD
                                     grad(p[BA_  ], x-1, y  , z   )), // BLENDED
                             lerp(u, grad(p[AB_  ], x  , y-1, z   ),  // RESULTS
                                     grad(p[BB_  ], x-1, y-1, z   ))),// FROM  8
                     lerp(v, lerp(u, grad(p[AA_+1], x  , y  , z-1 ),  // CORNERS
                                     grad(p[BA_+1], x-1, y  , z-1 )), // OF CUBE
                             lerp(u, grad(p[AB_+1], x  , y-1, z-1 ),
                                     grad(p[BB_+1], x-1, y-1, z-1 ))));
   }
   static double fade(double t) { return t * t * t * (t * (t * 6 - 15) + 10); }
   static double lerp(double t, double a, double b) { return a + t * (b - a); }
   static double grad(int hash, double x, double y, double z) {
      int h = hash & 15;                      // CONVERT LO 4 BITS OF HASH CODE_
      double u = h<8 ? x : y,                 // INTO 12 GRADIENT DIRECTIONS.
             v = h<4 ? y : h==12||h==14 ? x : z;
      return ((h&1) == 0 ? u : -u) + ((h&2) == 0 ? v : -v);
   }
   static final int p[] = new int[512], permutation[] = { 151,160,137,91,90,15,131,13,201,95,96,53,194,233,7,225,140,36,103,30,69,142,8,99,37,240,21,10,23,
   190, 6,148,247,120,234,75,0,26,197,62,94,252,219,203,117,35,11,32,57,177,33,88,237,149,56,87,174,20,125,136,171,168, 68,175,74,165,71,134,139,48,27,166,77,146,158,231,83,111,229,122,60,211,133,230,220,105,9
   102,143,54, 65,25,63,161, 1,216,80,73,209,76,132,187,208, 89,18,169,200,196,135,130,116,188,159,86,164,100,109,198,173,186, 3,64,52,217,226,250,124,123,
   5,202,38,147,118,126,255,82,85,212,207,206,59,227,47,16,58,17,182,189,28,42,223,183,170,213,119,248,152, 2,44,154,163, 70,221,153,101,155,167, 43,172,9,
```

Perlin Noise

- Parameters
  - Change amplitude: e.g. $10 * \texttt{noise}(x)$
  - Change frequency: e.g. $\texttt{noise}(10 * x)$
- Many other possible ways to implement a basic noise function
  - Simplex noise (use triangles/tetrahedra instead of voxel grid)
  - Sparse Gabor convolution
  - etc.

Spectral Synthesis



- Building a complex function $f_s(x)$ by summing weighted contributions from a scaled primitive function $f(x)$

$$f_s(x) = \sum_i w_i f(s_i x)$$

- Weight (amplitude) w_i, frequency scaling s_i

- Example: Fourier basis

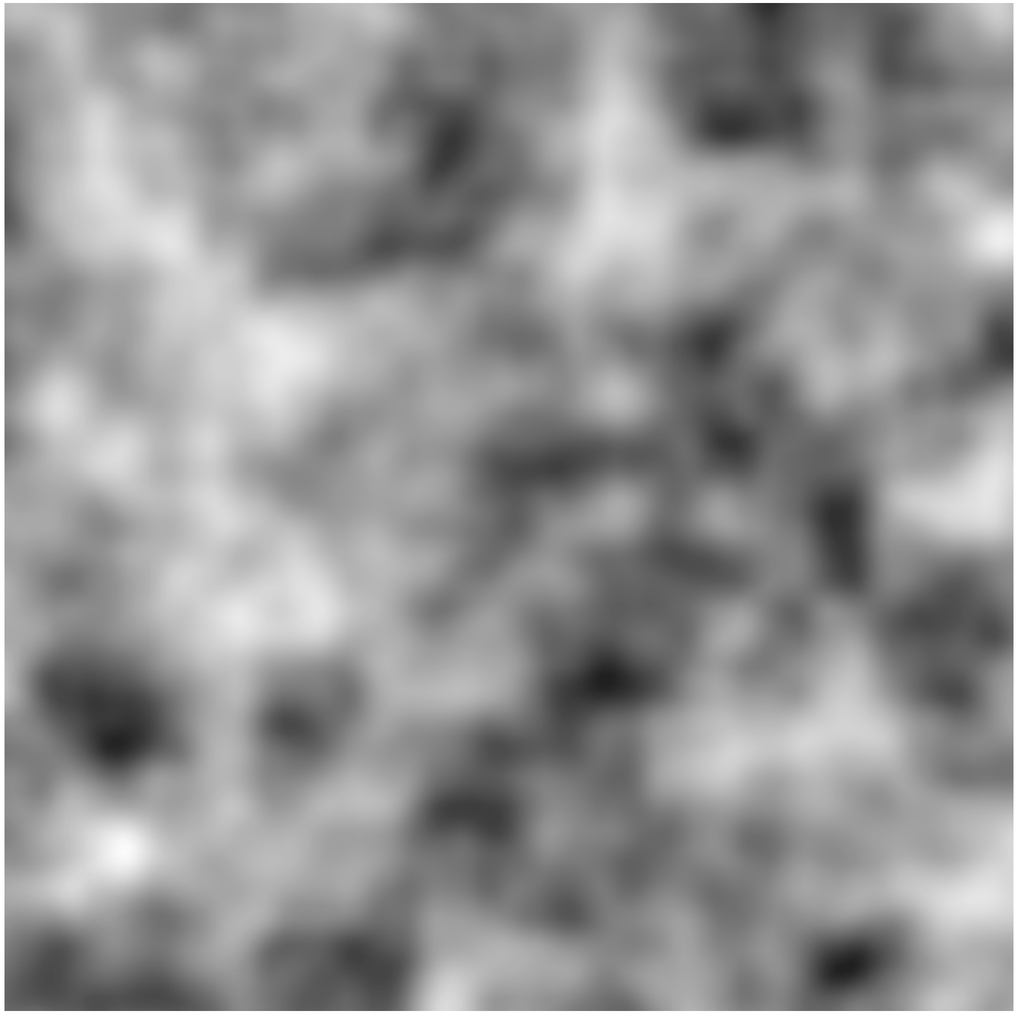
$$f_s(x) = w_0 + w_1 \cos(x) + w_2 \cos(3x) + w_3 \cos(5x) + w_4 \cos(7x) + \ldots$$

Fractal Brownian Motion (fBm)

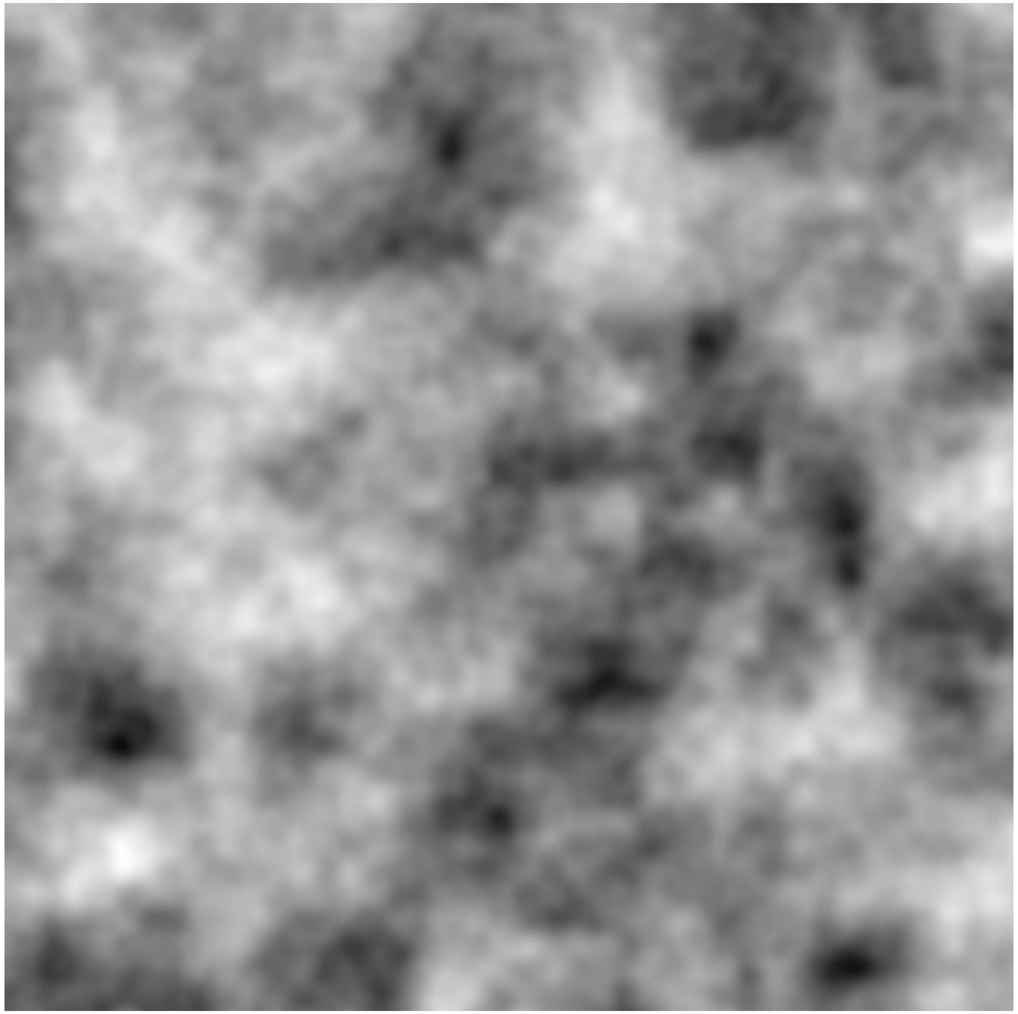
- Spectral synthesis of noise function
  - Progressively higher frequency
  - Progressively smaller amplitude
- Typically Perlin noise is used
- Each term in the summation is called an *octave*

fBm - 1 Octave

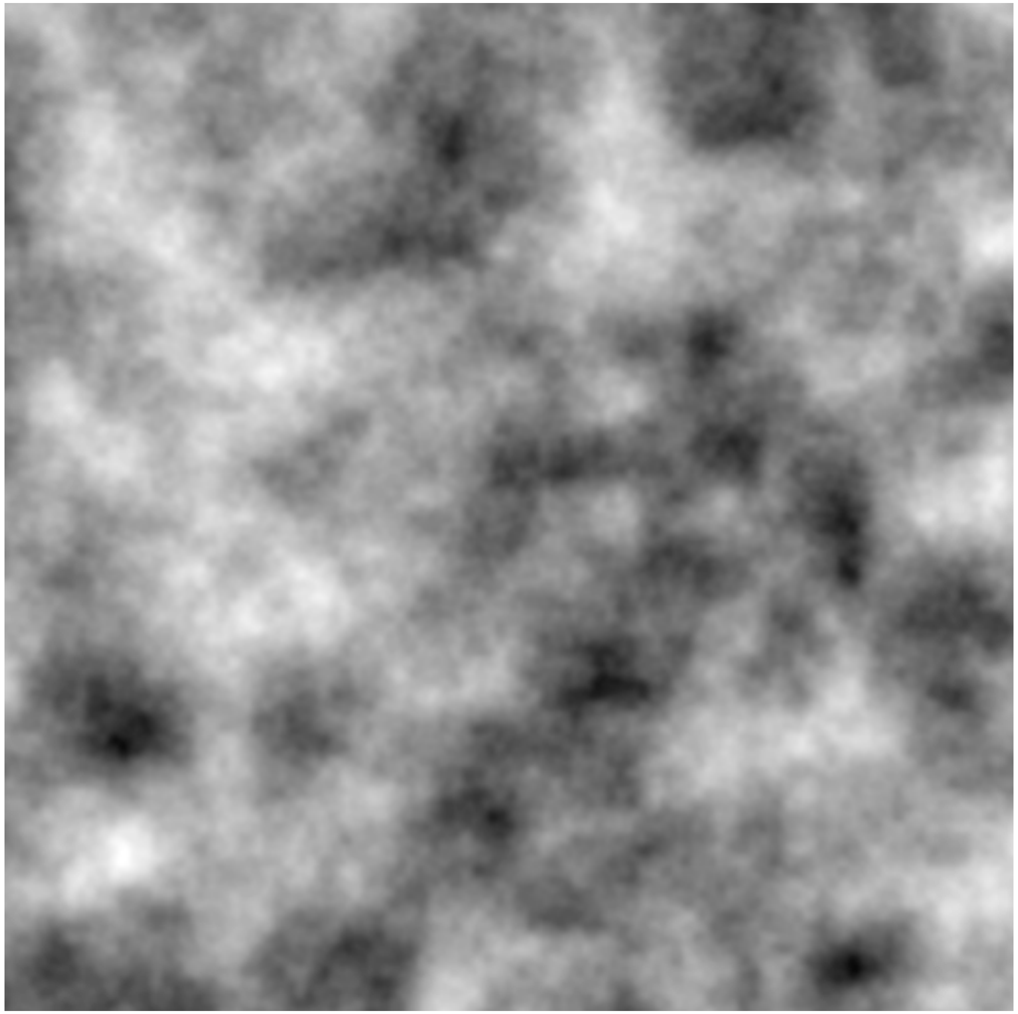
fBm - 2 Octave



fBm - 3 Octave

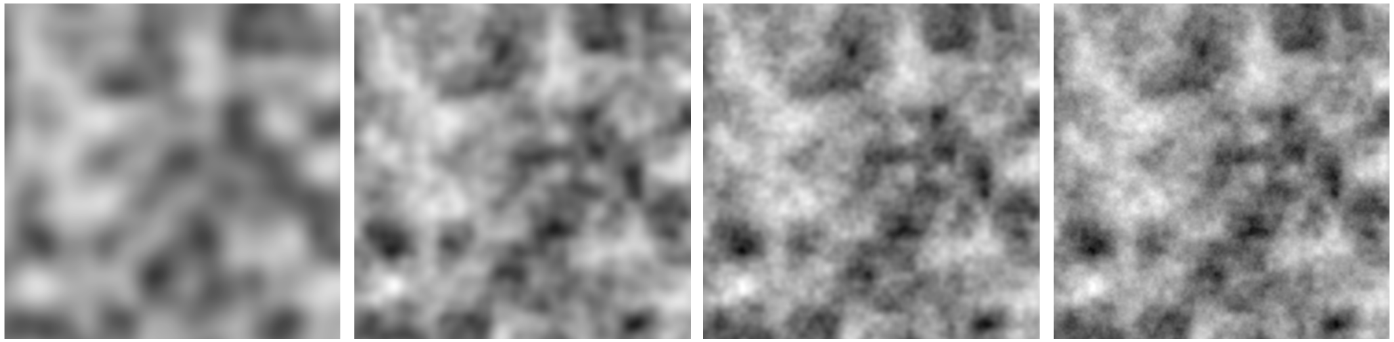


fBm - 4 Octave



Fractal Brownian Motion (fBm)

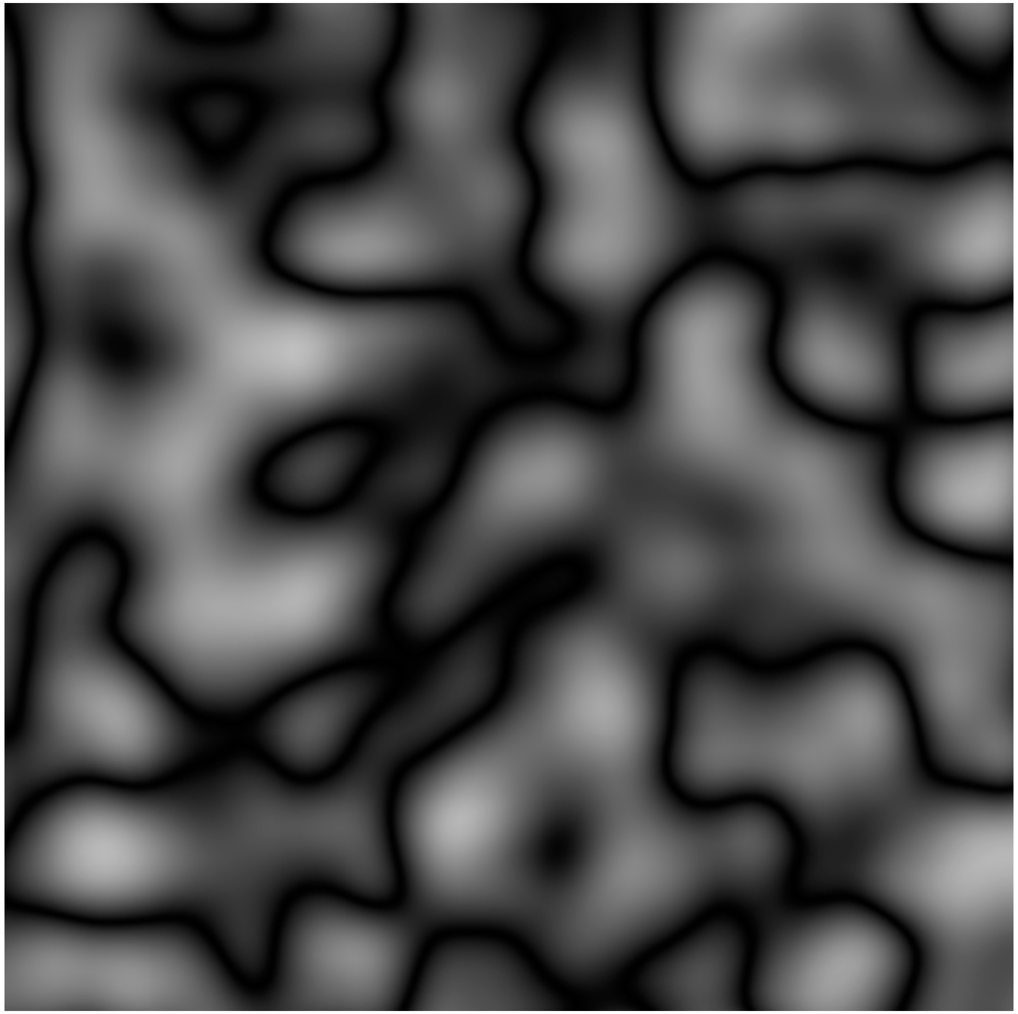
- Spectral synthesis of noise function
  - Progressively smaller frequency
  - Progressively smaller amplitude
- Typically Perlin noise is used
- Each term in the summation is called an *octave*
- Each octave typically doubles frequency and halves amplitude



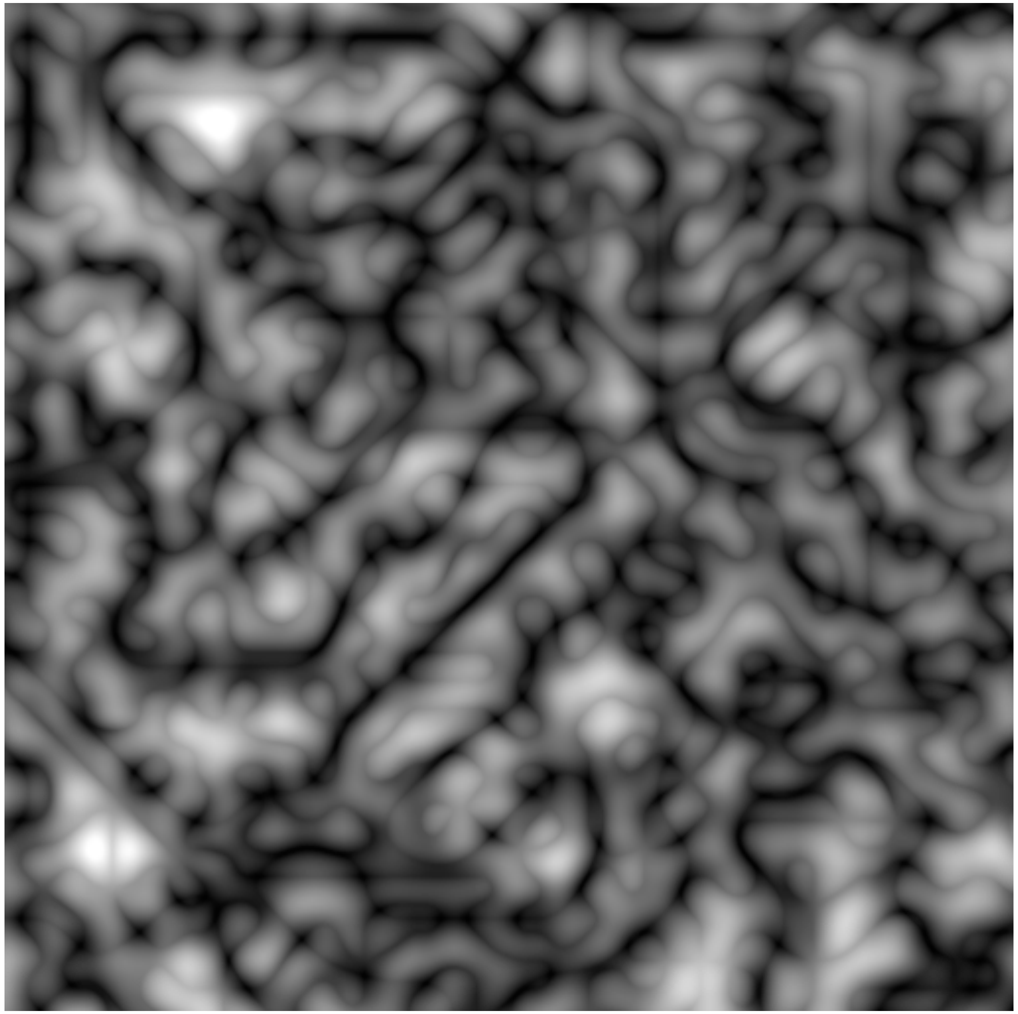
“Turbulence”

- Another common compound noise function
- Same as fBm, but sum the *absolute value* of the noise function

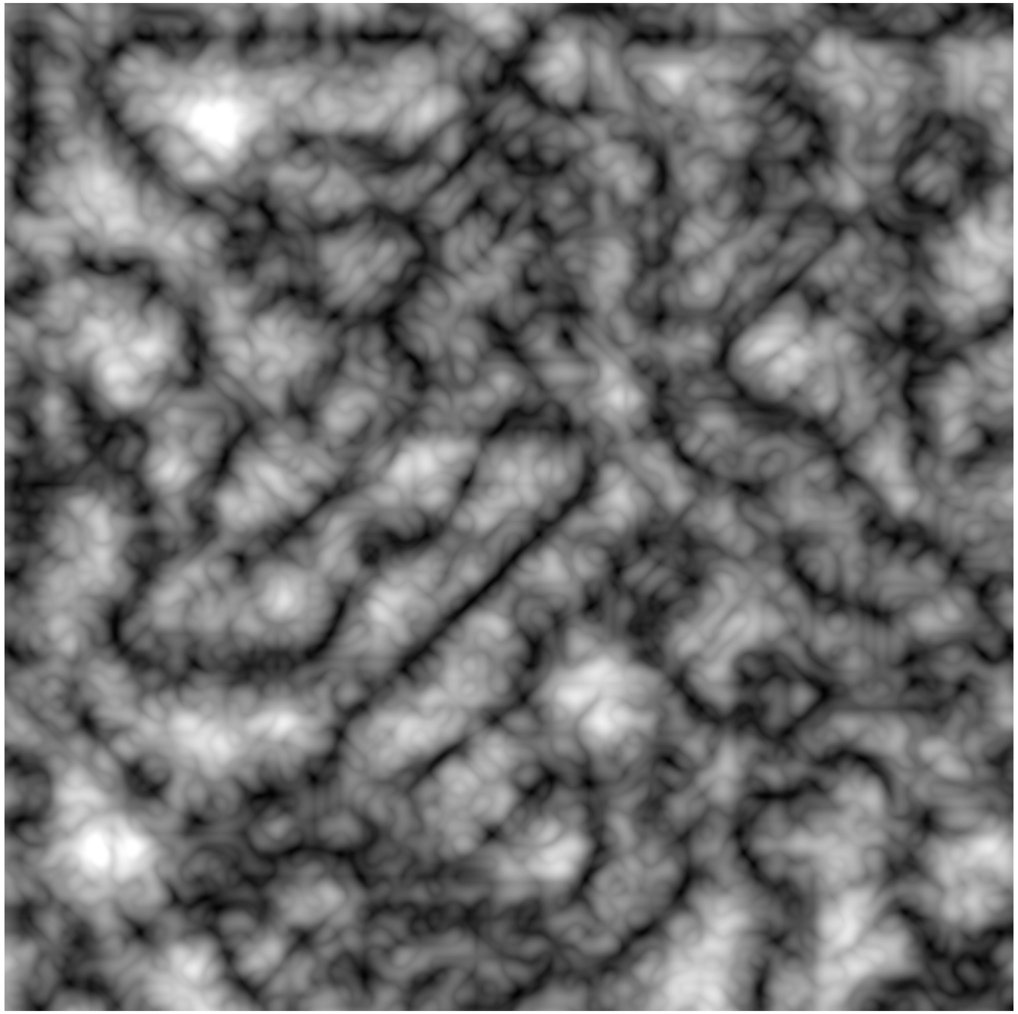
Turbulence - 1 Octave



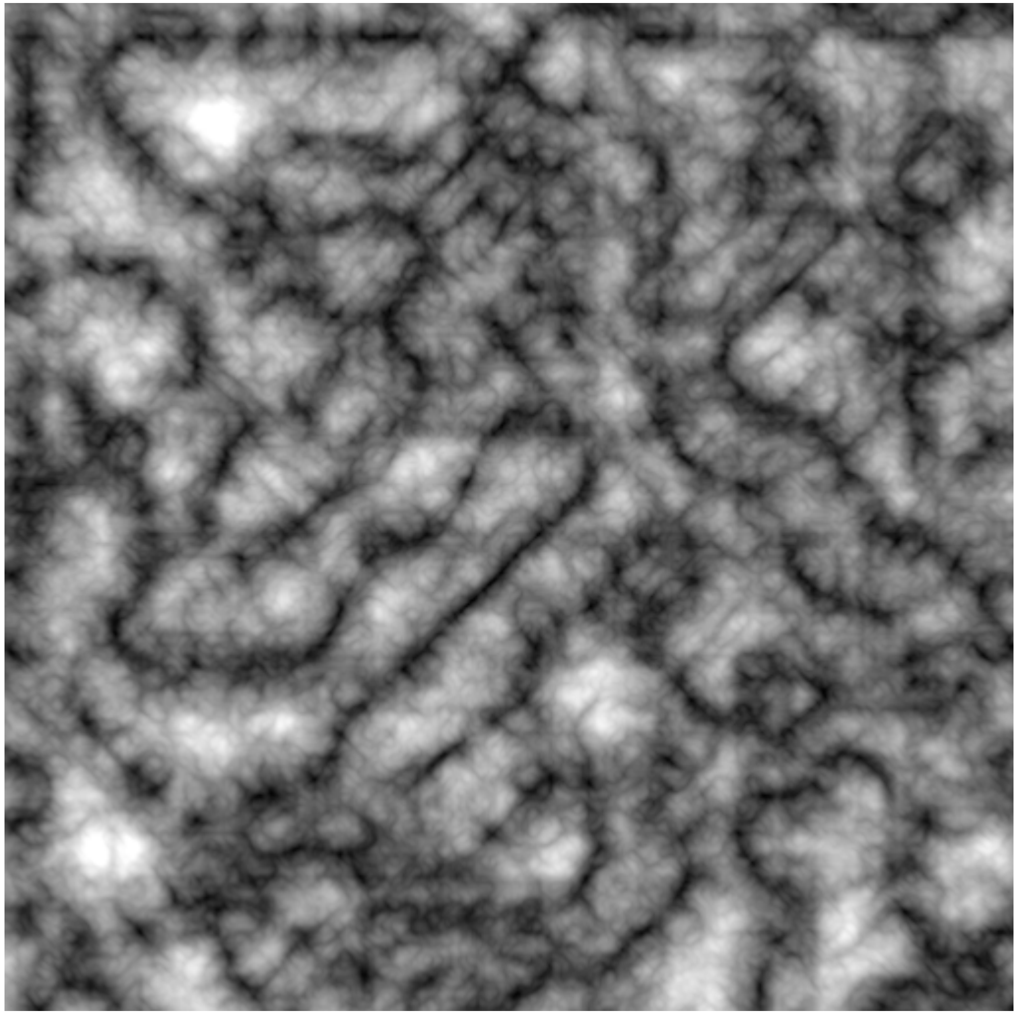
Turbulence - 2 Octave



Turbulence - 3 Octave

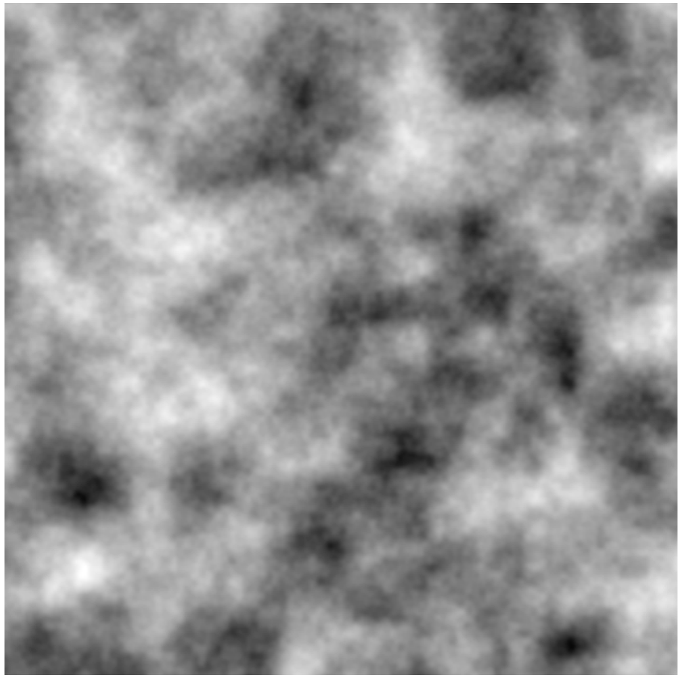


Turbulence - 4 Octave

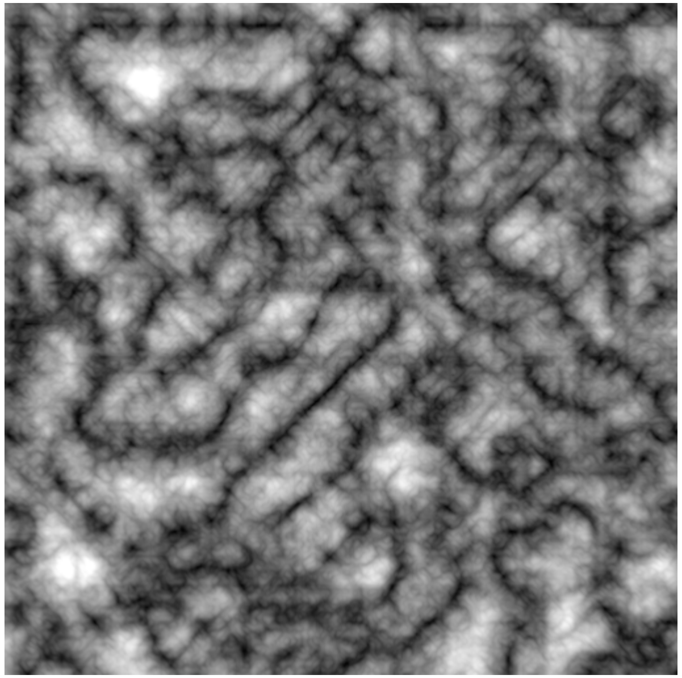


FBm vs Turbulence

Both useful primitives for emulating natural materials

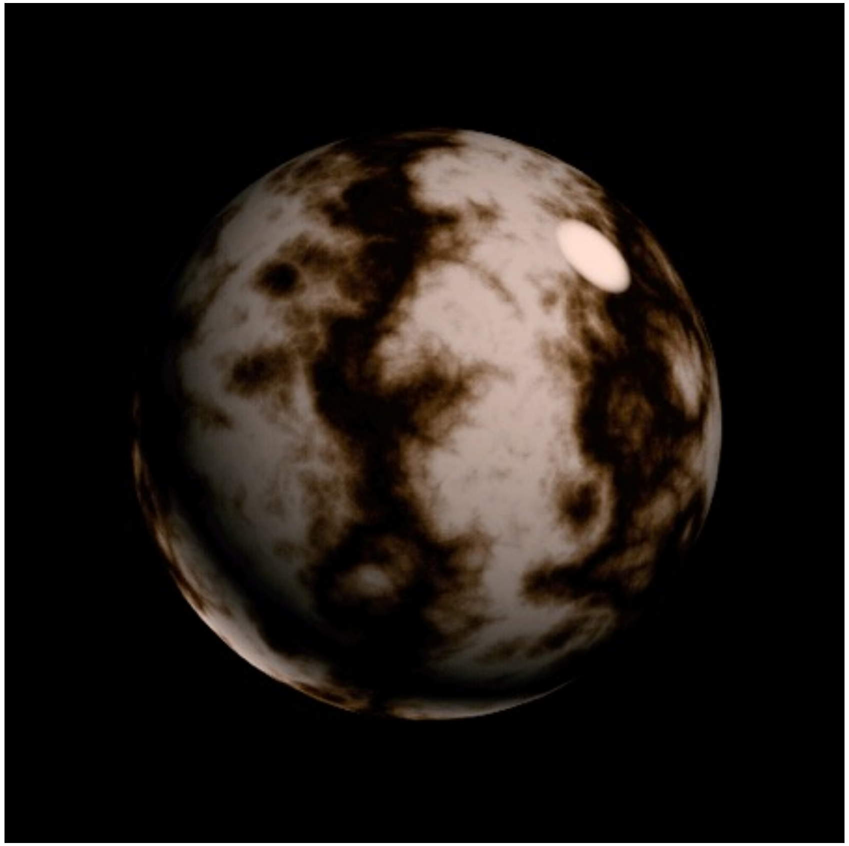


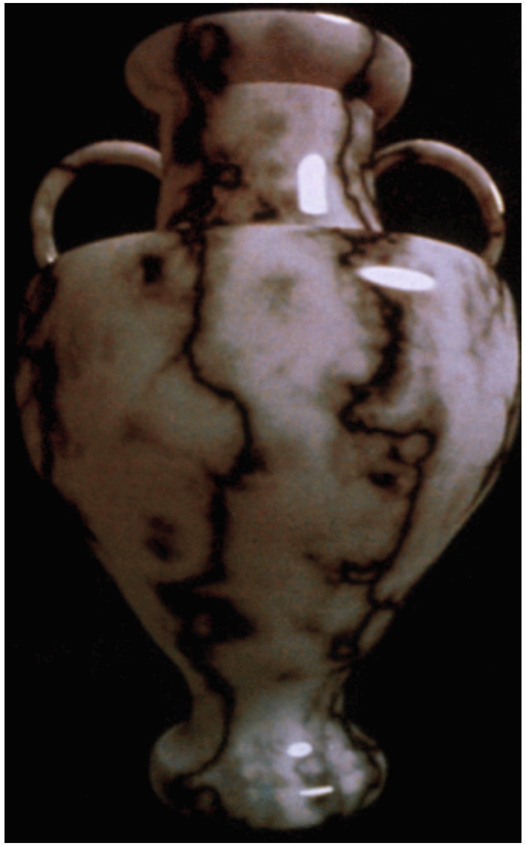
fBm



turbulence

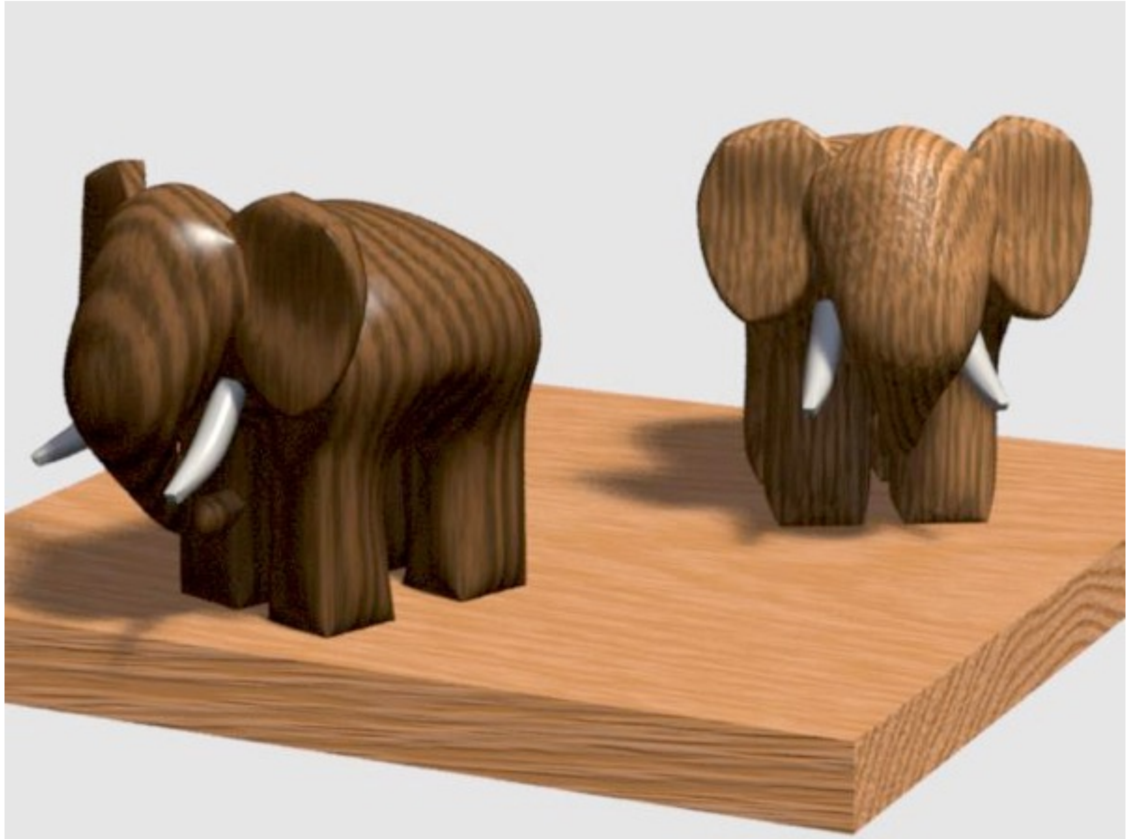
Marble





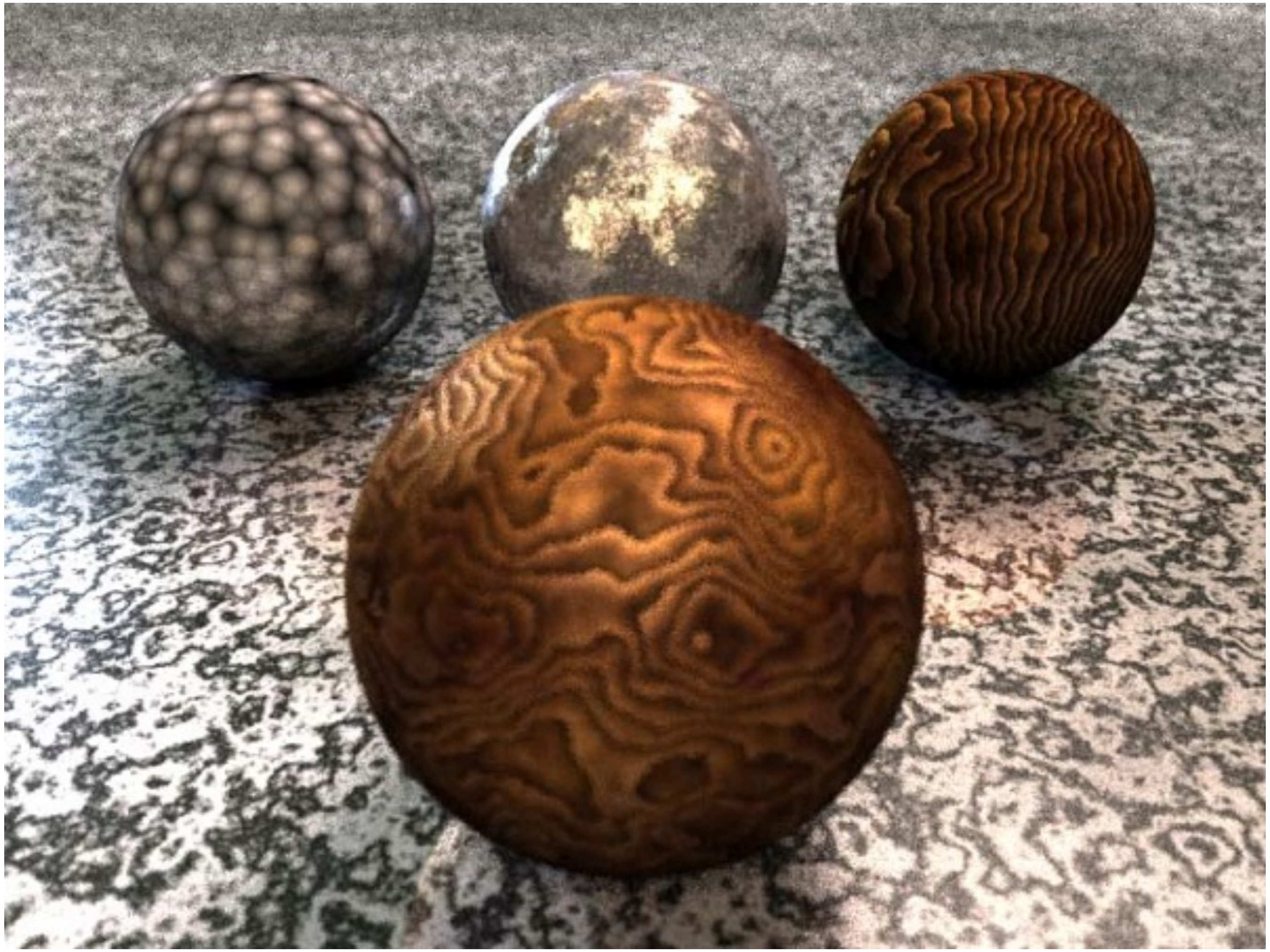
$$\texttt{color} = \sin(x + \texttt{turbulence}(x, y, z))$$

Wood



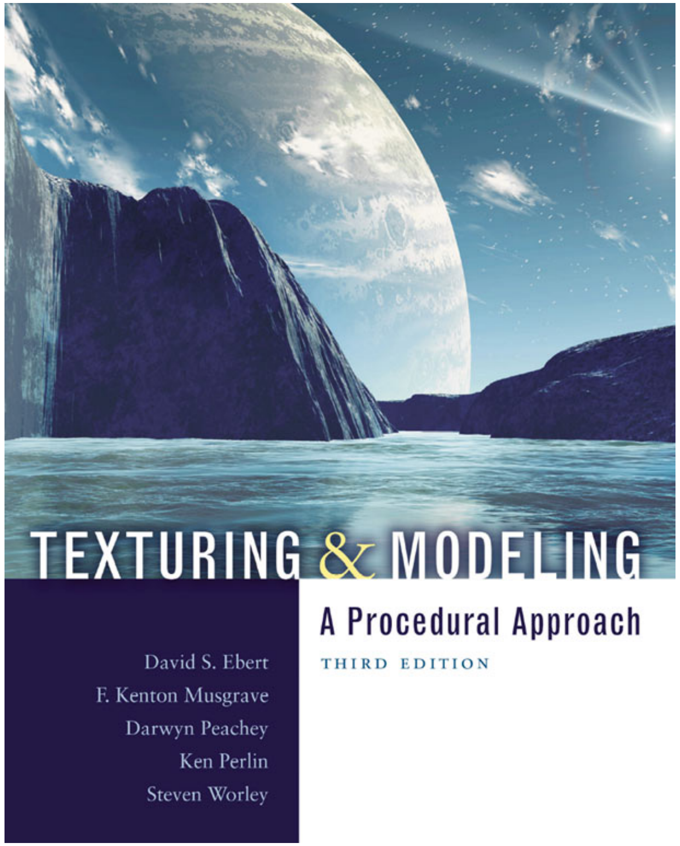
$$\texttt{color} = \sin\left(\sqrt{x^2 + y^2} + \texttt{fbm}(x, y, z)\right)$$

And More...



And More...

Literature


TEXTURING & MODELING
A Procedural Approach
THIRD EDITION
David S. Ebert
F. Kenton Musgrave
Darwyn Peachey
Ken Perlin
Steven Worley