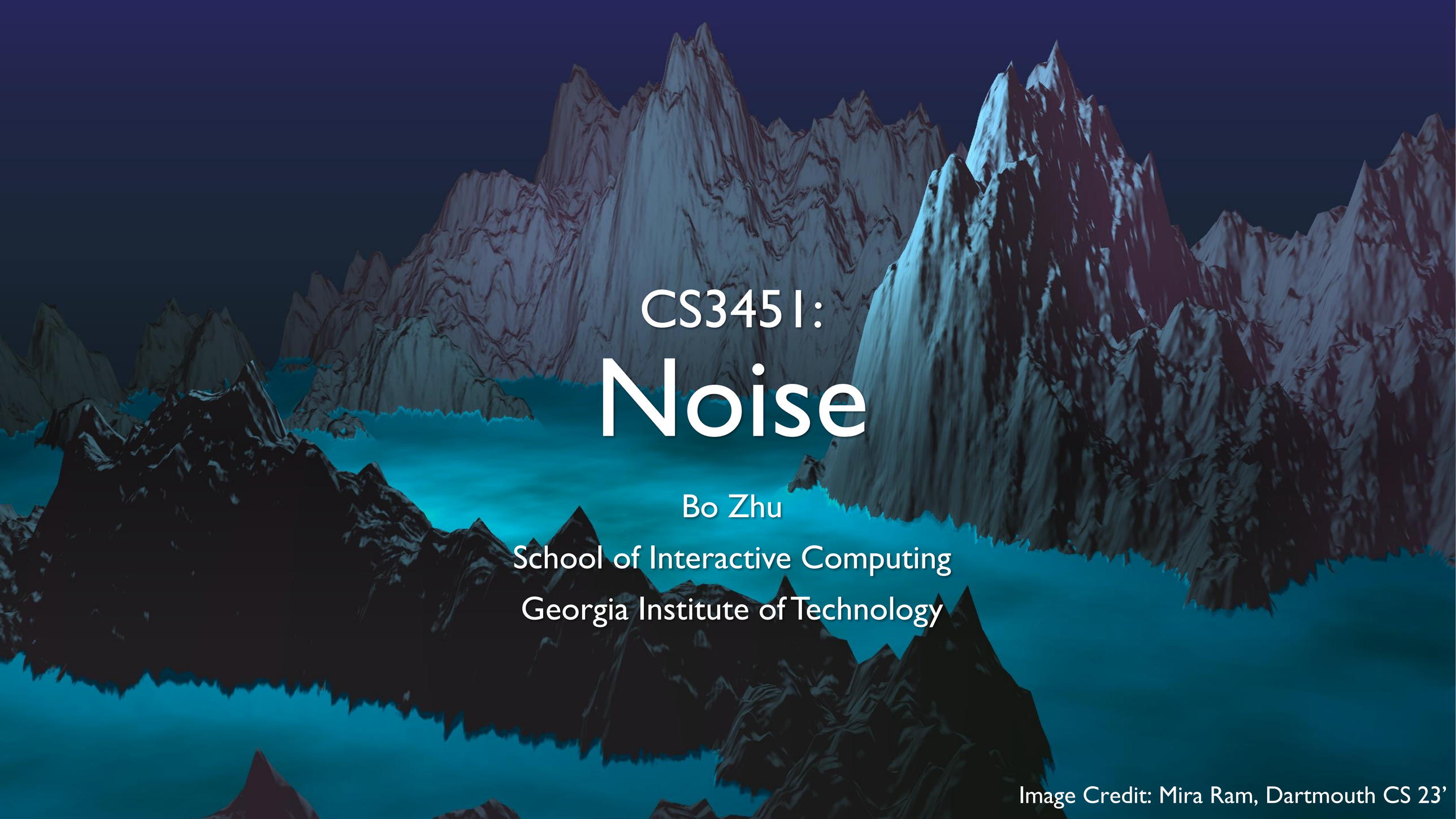


CS3451:
Noise

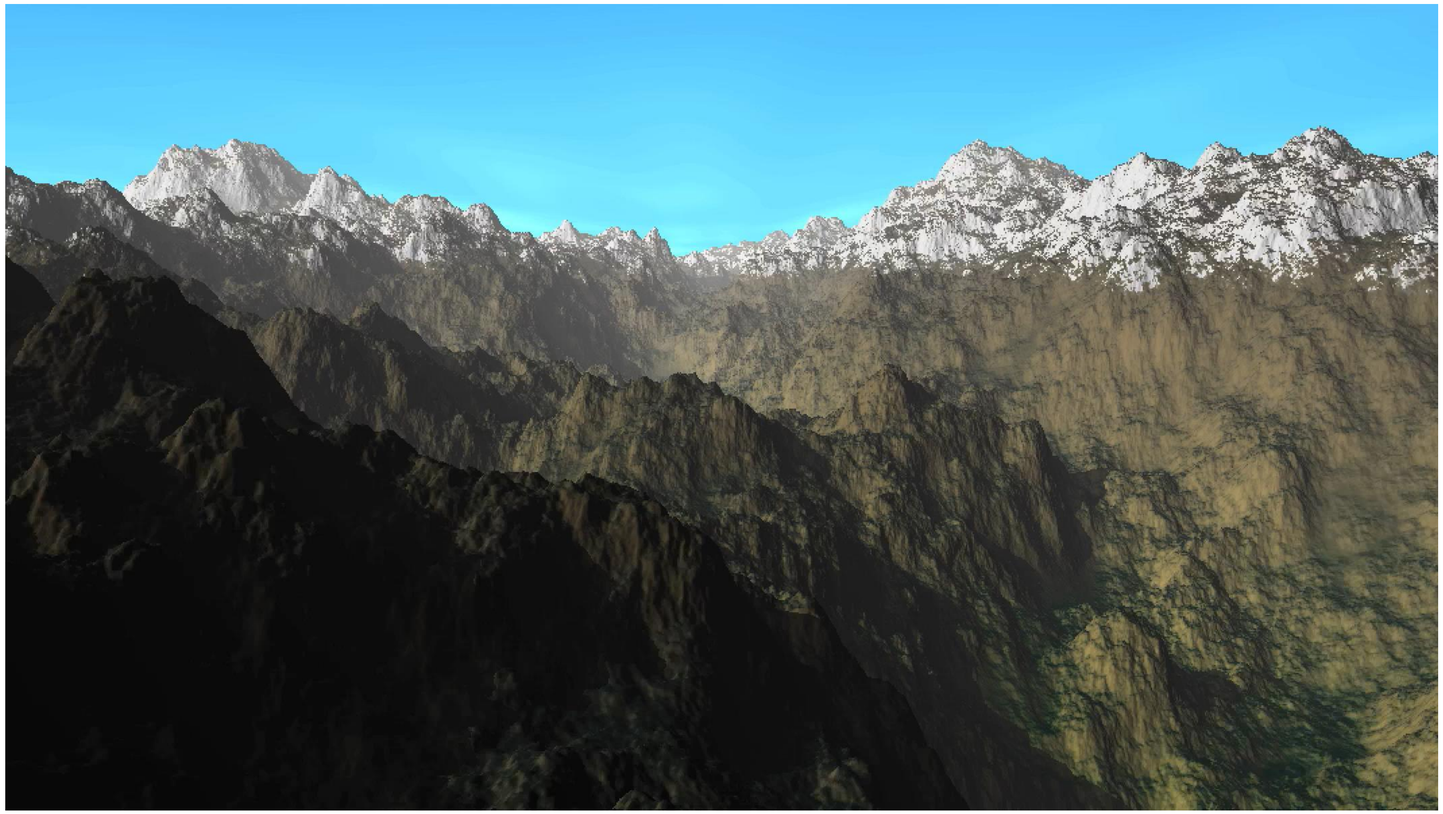
Bo Zhu

School of Interactive Computing

Georgia Institute of Technology

Image Credit: Mira Ram, Dartmouth CS 23'

Motivational Video:
Infinite Terrain Generation with Procedural Modeling



by Evan Z. Muscatel,
Dartmouth CS 21’

Inventor of our A6 ☺

Motivational Video:

Houdini Procedural Modeling Demo

https://www.youtube.com/watch?v=CE7H_P8q6QY

Motivational Video:

WonderJourney: Going from Anywhere to Everywhere

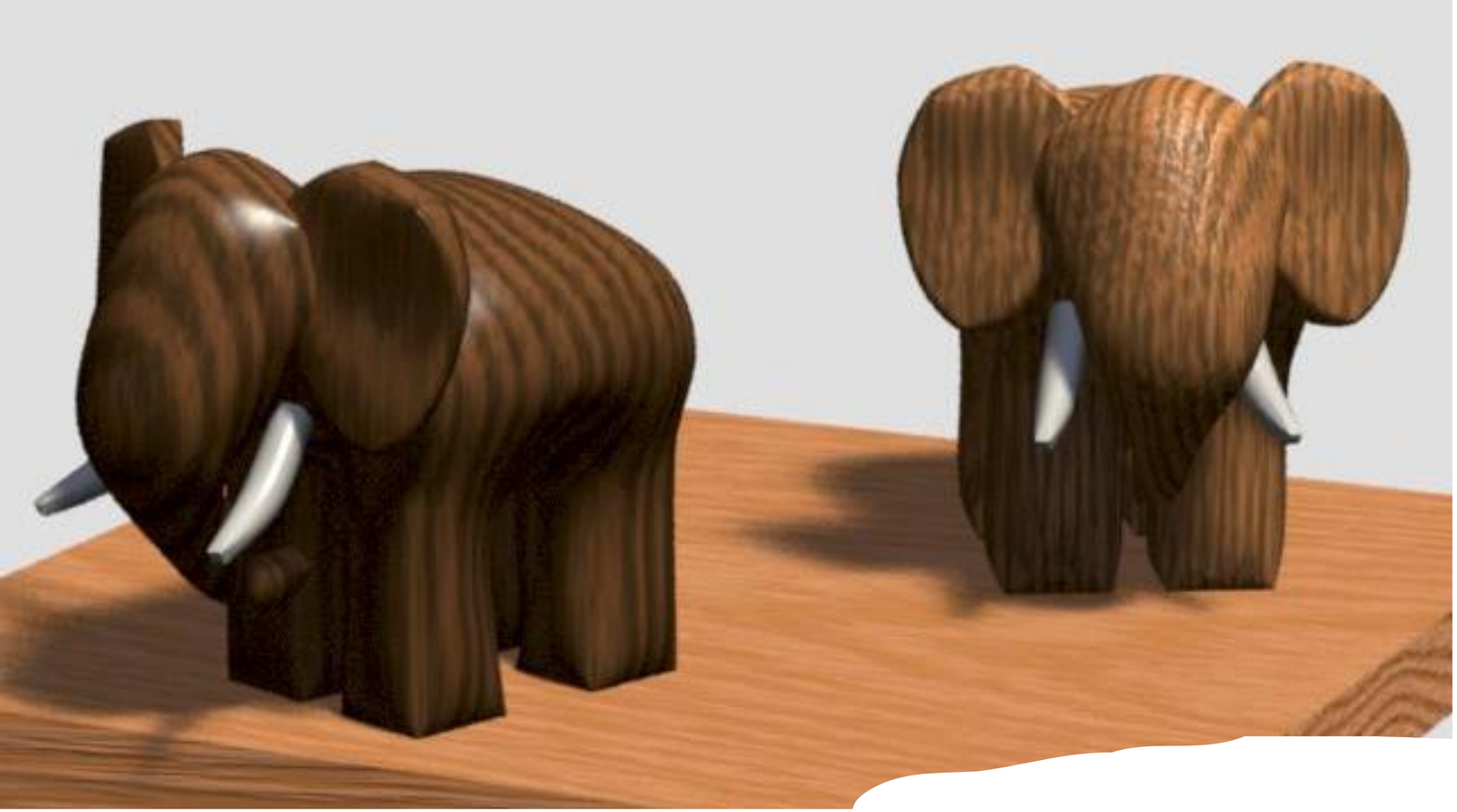
https://kovenyu.com/wonderjourney/

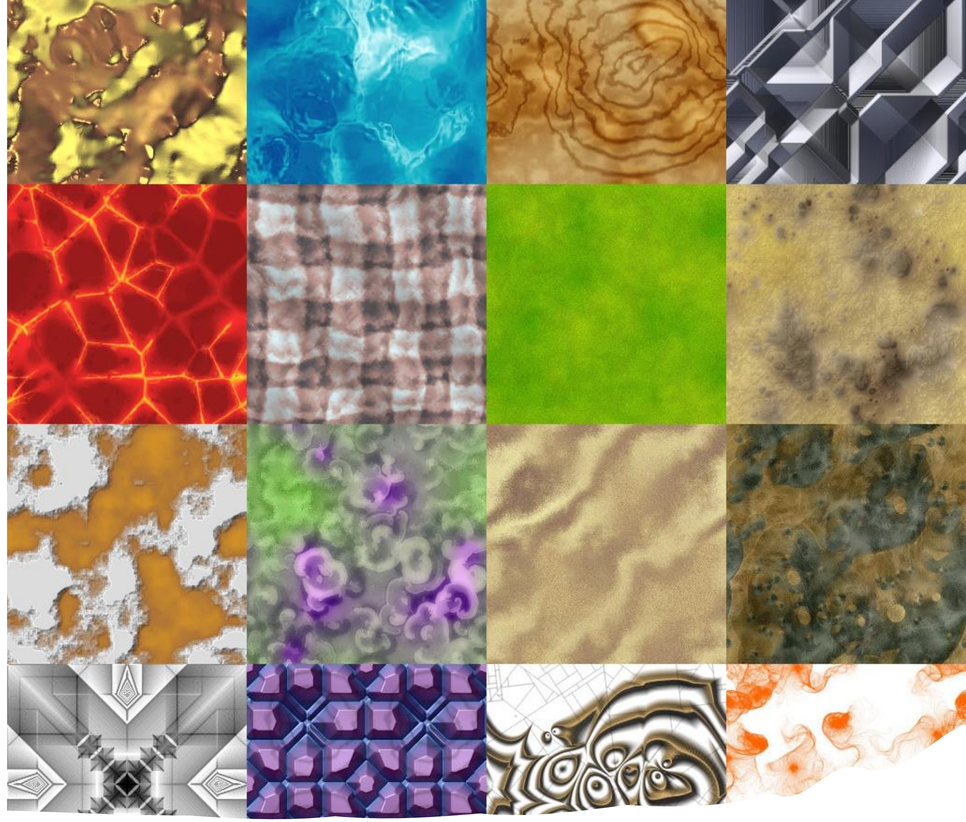
Motivational Video:

OpenAI Sora: World Generation Model

https://www.youtube.com/watch?v=anmuklFtu8U


We want to Define Rules
to Generate the World



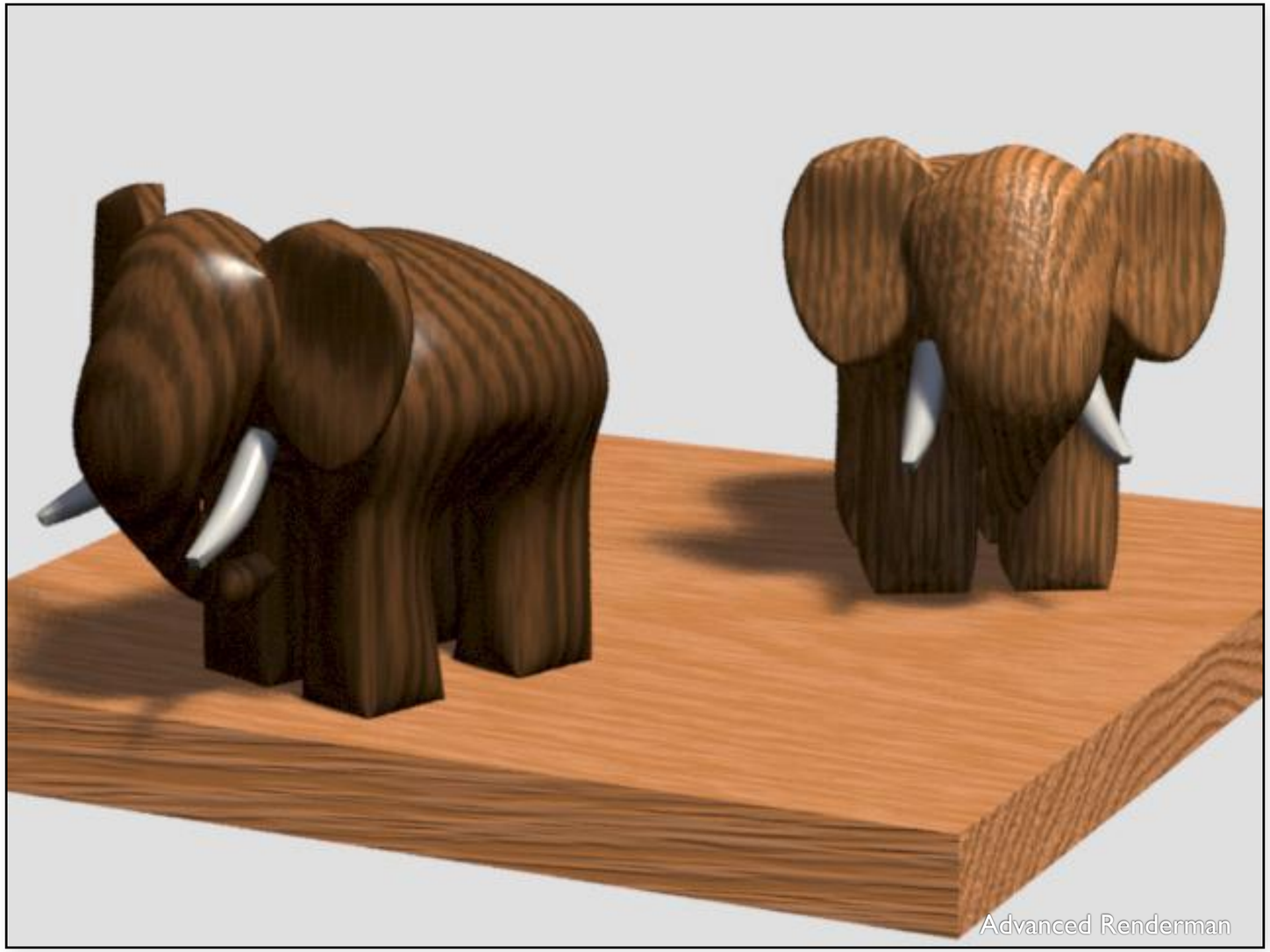


Pioneering Exploration in Graphics:

Generating Texture Images with **Simple** Math

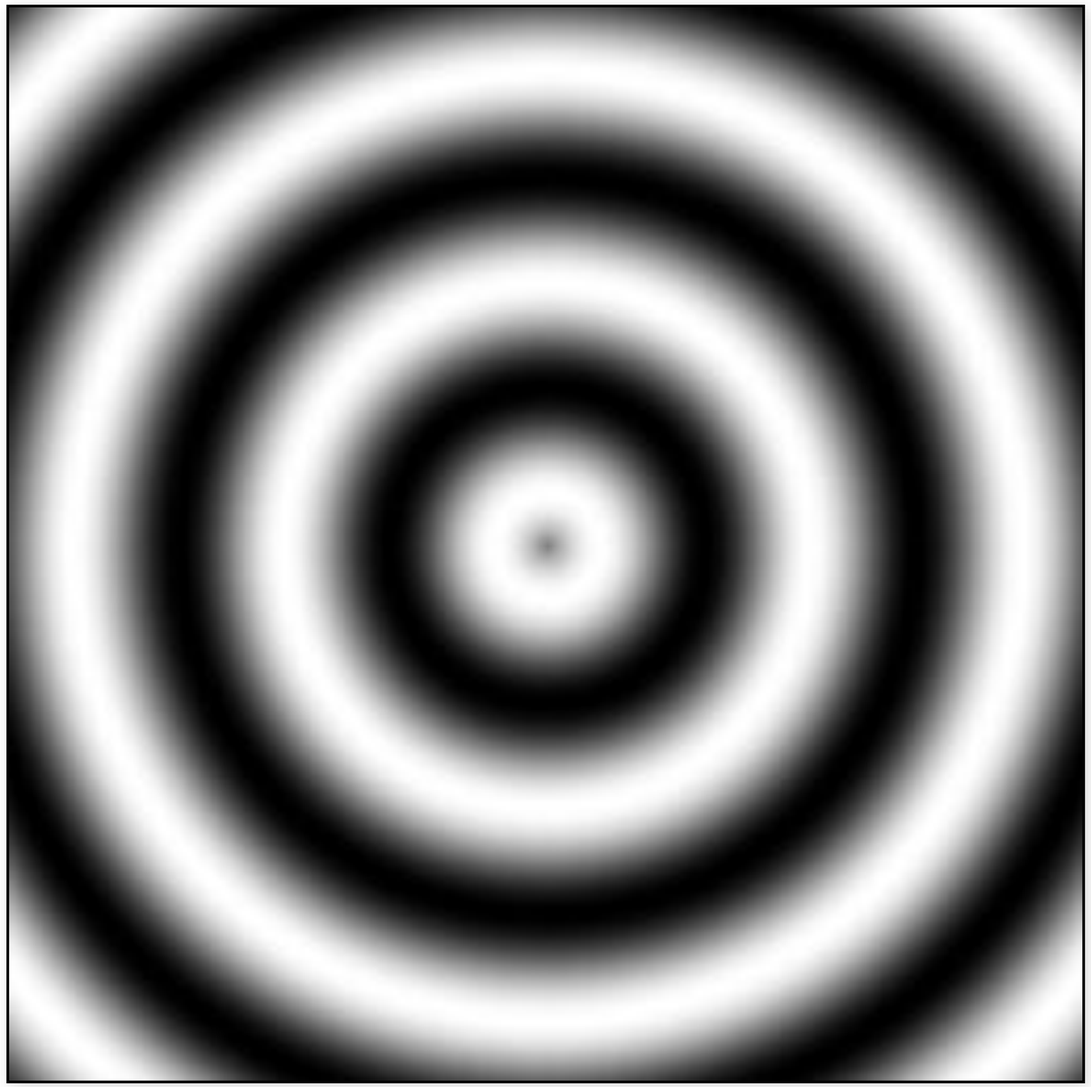
Wood

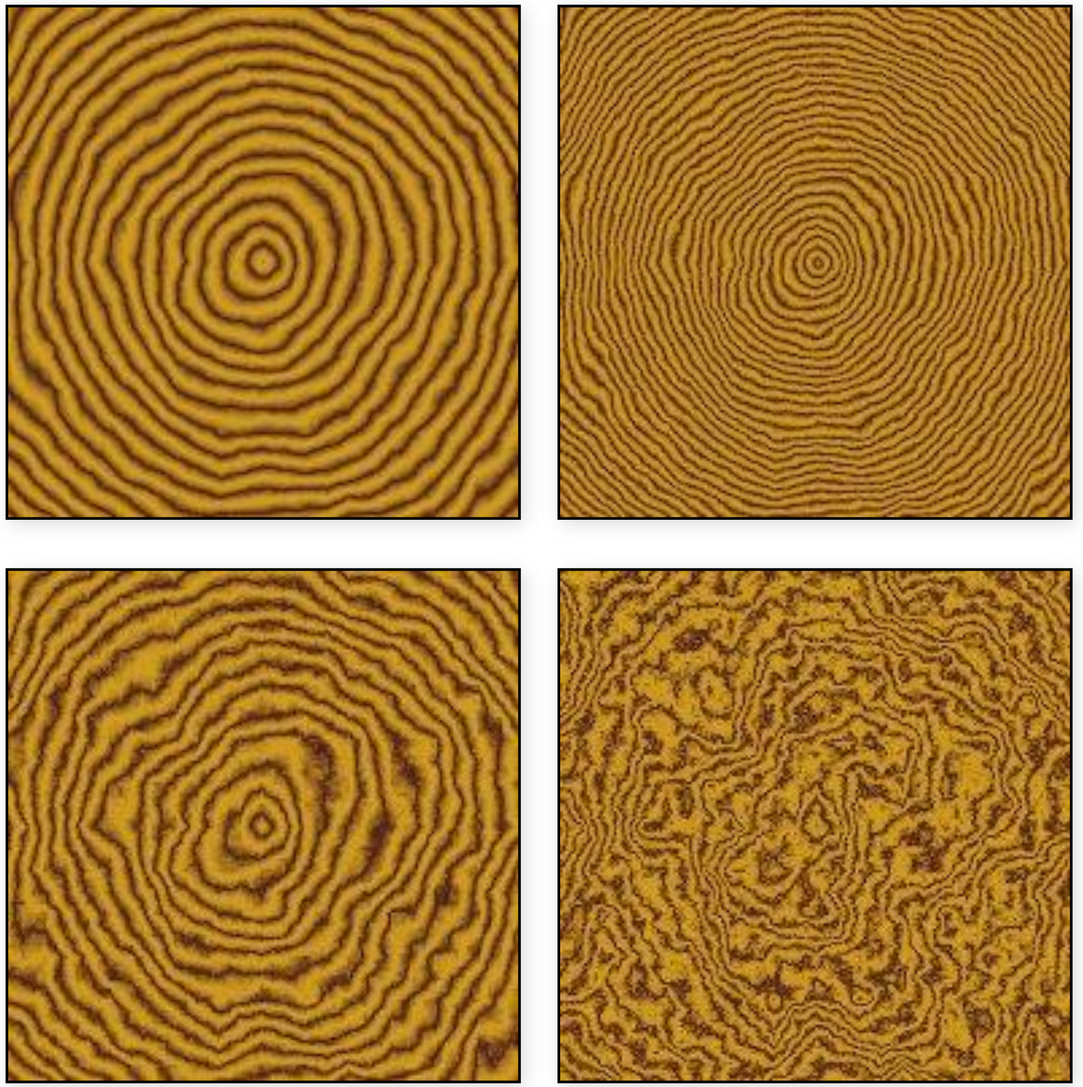
$$(1 + \sin(\mathrm{sqrt}(\mathbf{p}_x^2 + \mathbf{p}_y^2) + \mathrm{fBm}(\mathbf{p})))/2$$



Another Wood

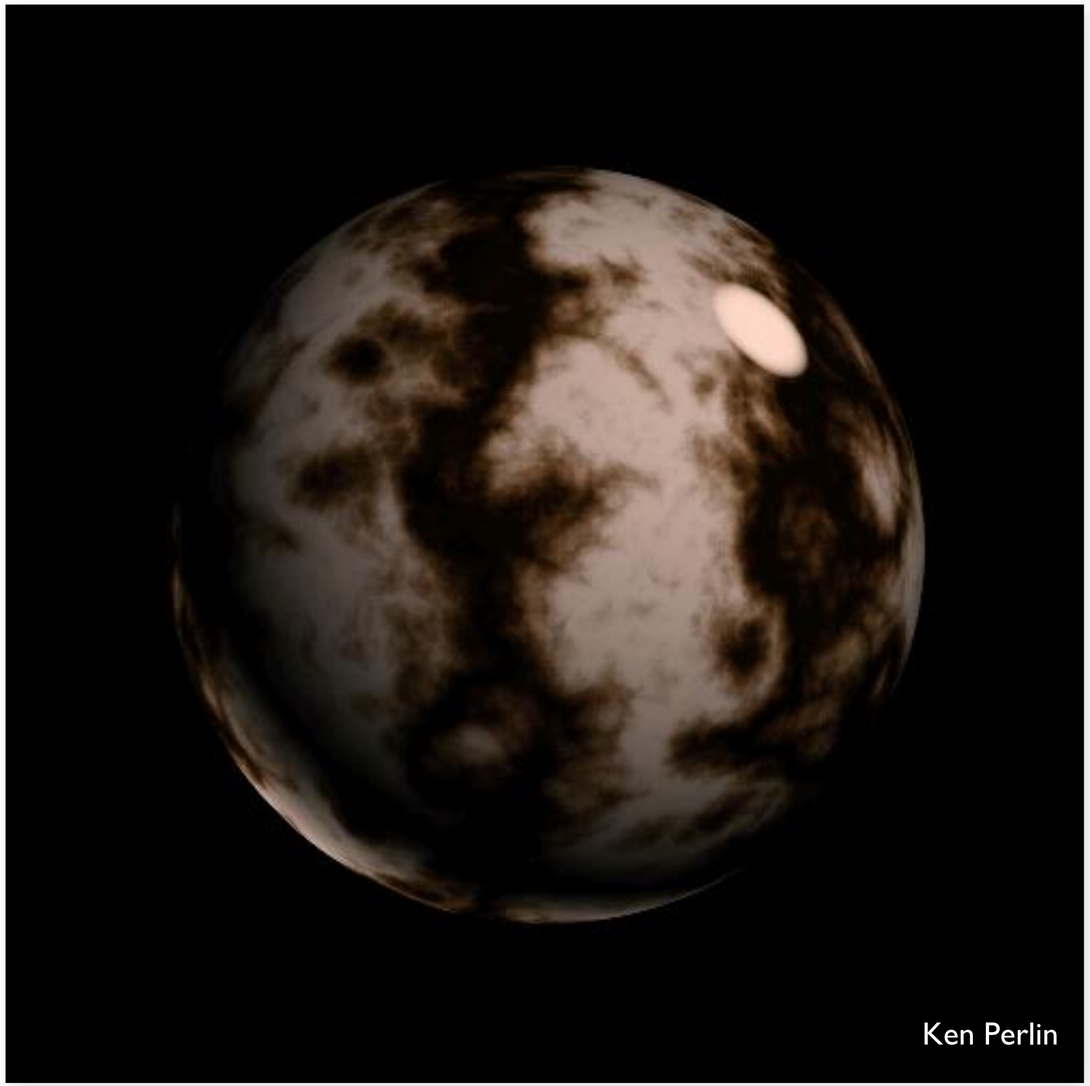
$(1 + \sin(\mathrm{sqrt}(\mathbf{p}_x^2 + \mathbf{p}_y^2) + \mathrm{fBm}(\mathbf{p})))/2$

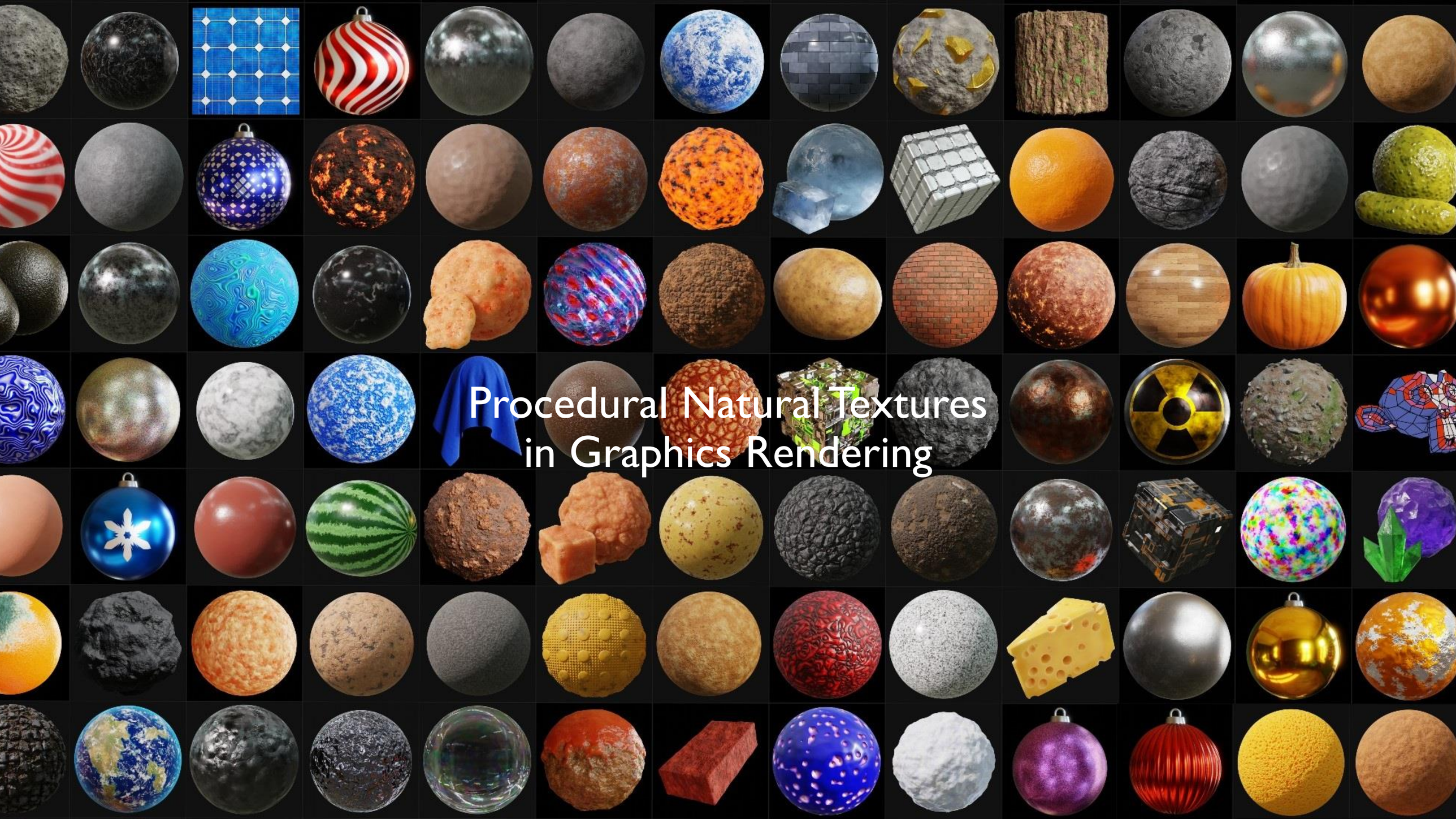




Marble

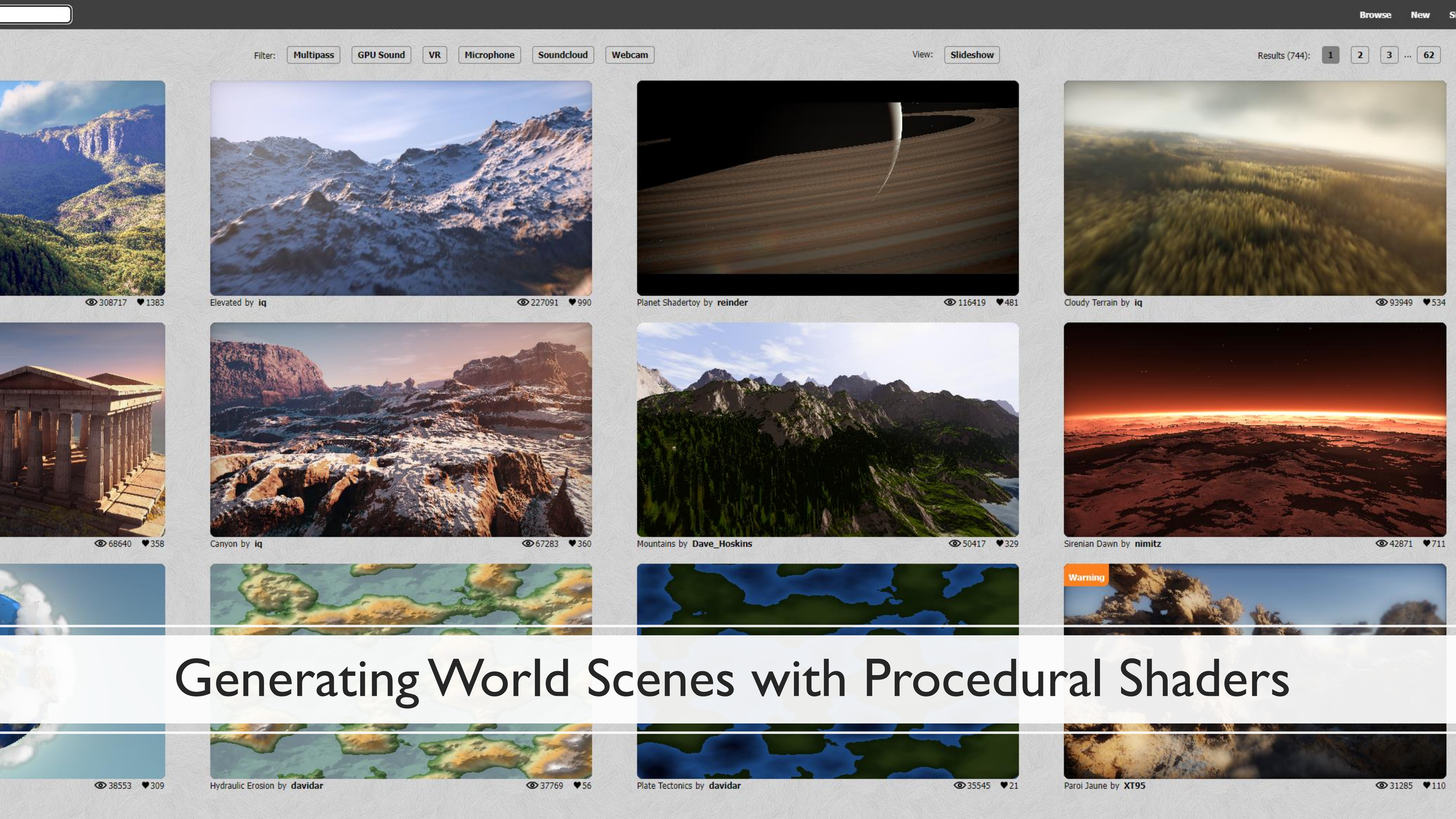
$$(1 + \sin(k_1\,\mathbf{p}_x + \text{turbulence}(k_2\,\mathbf{p}))/w)/2$$




Procedural Natural Textures
in Graphics Rendering

Procedural Natural Landscapes in Movies and Games


Procedural City Scenes
in Movies and Games



Generating World Scenes with Procedural Shaders

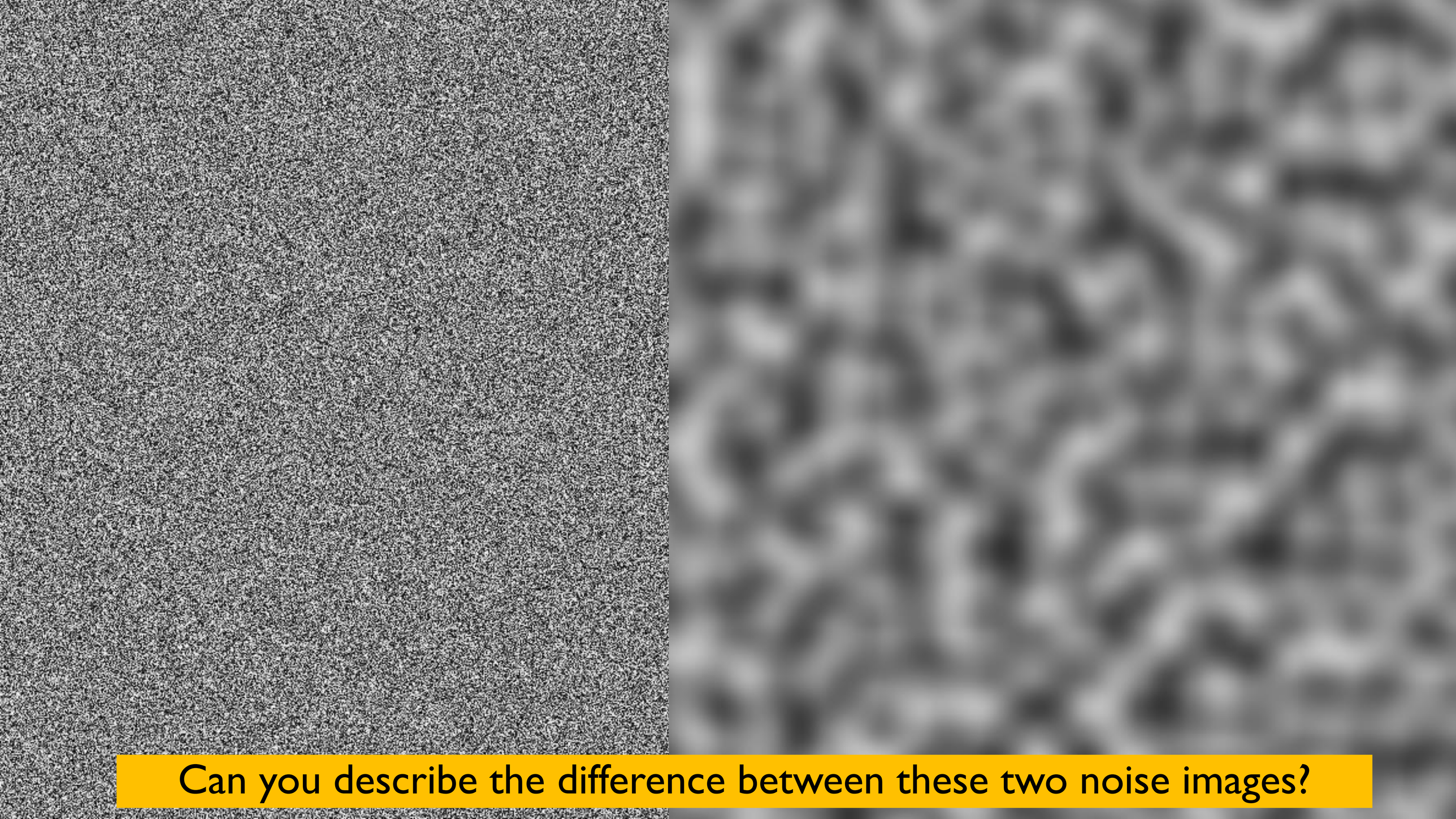
ShaderToy Demo: Procedural Terrain

https://www.shadertoy.com/view/4ttSWf

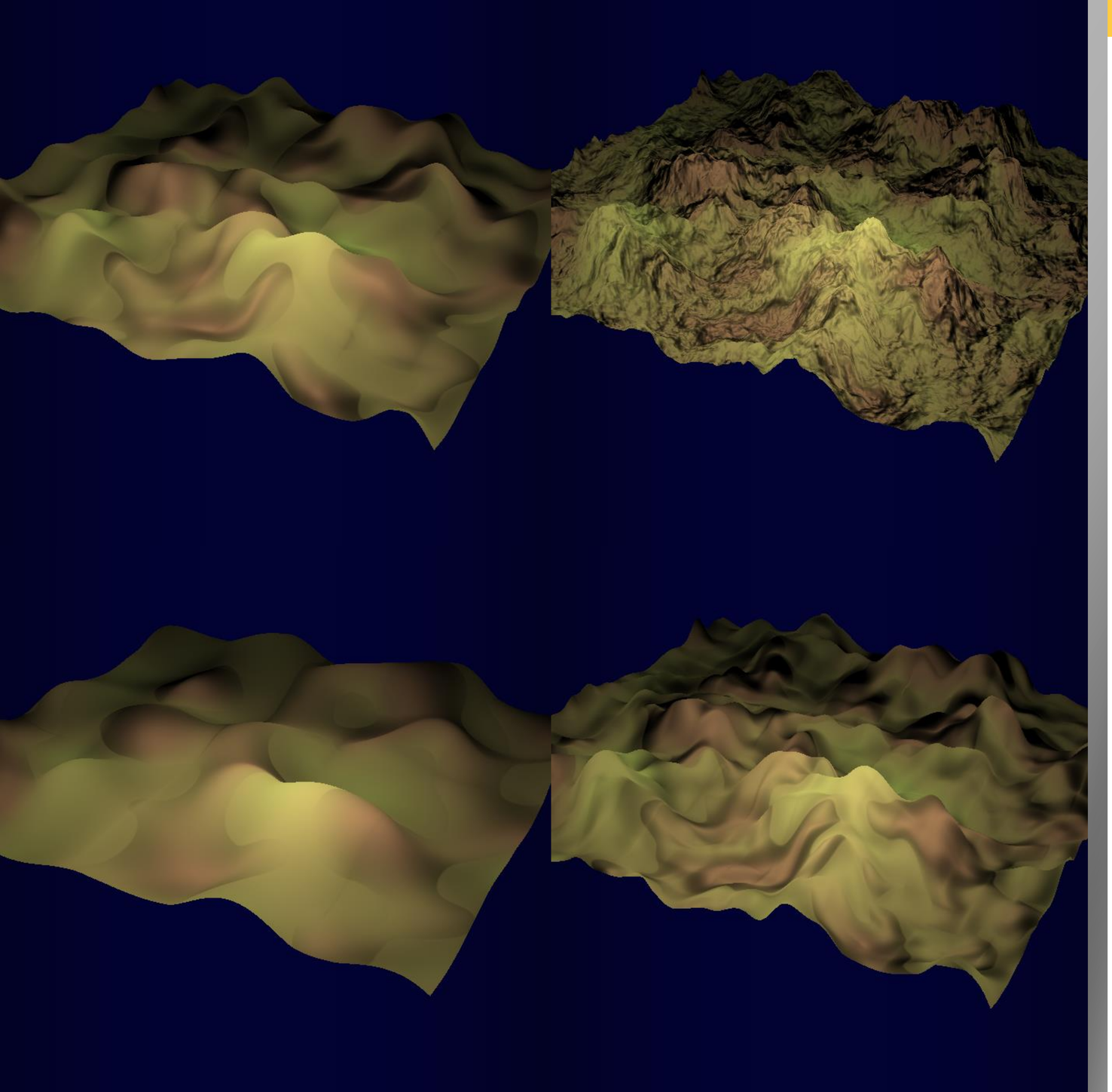
Study Plan

- Natural-looking Noise
- 1D Perlin Noise
- 2D Perlin Noise
- Terrain Generation
- Other Noise Functions

Natural-Looking Noise



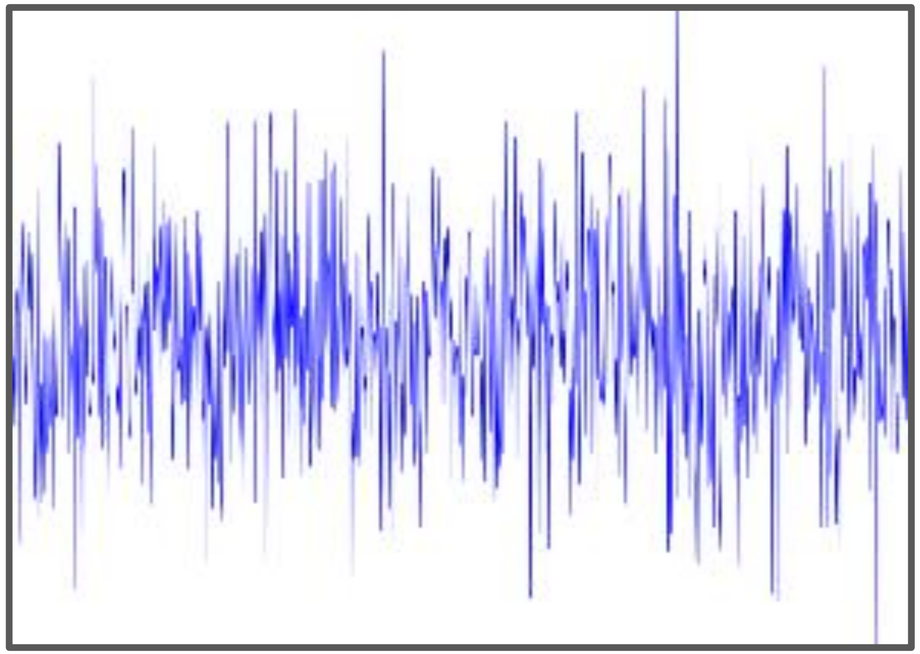
Can you describe the difference between these two noise images?

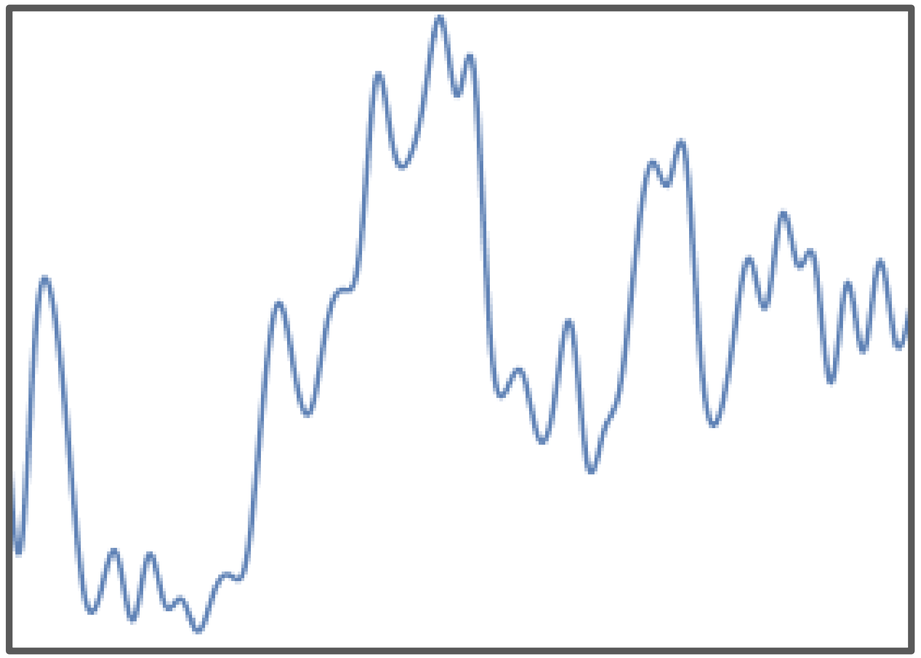


Can you tell the difference among these terrain meshes?

Which one looks more like a real mountain?

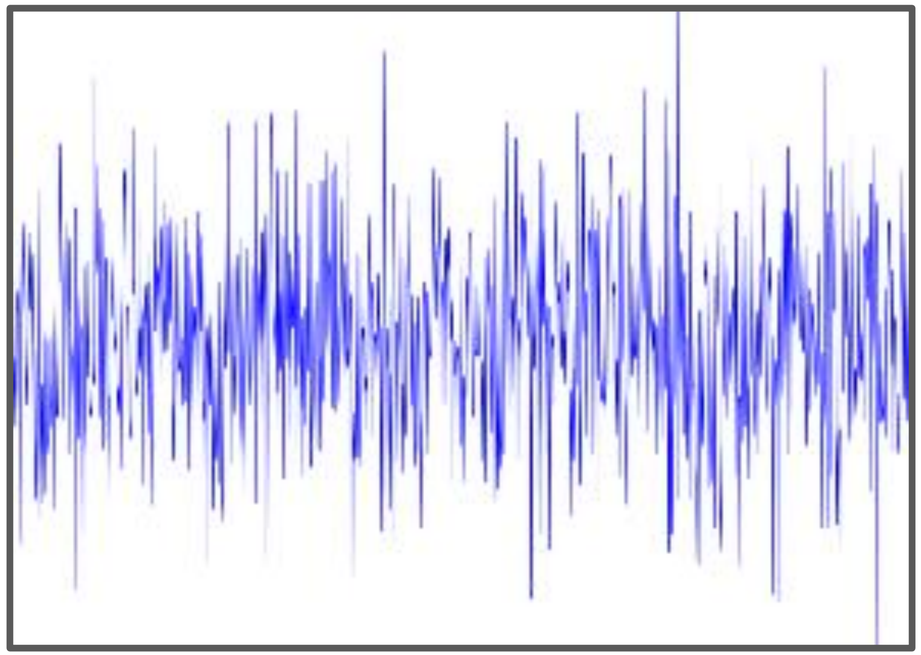
How do we describe the randomness of natural systems?



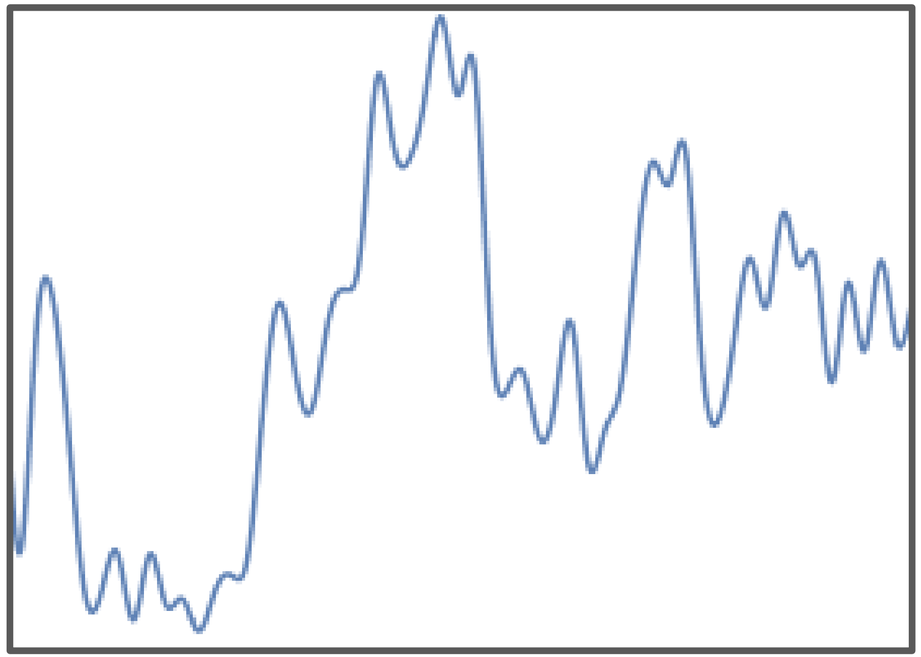


Maybe we can work on a simpler problem first:

Can we describe the difference between these two signals?



White Noise



Natural Noise

Natural-looking Noise Function

- A good noise is a noise that looks random, but **changes smoothly**
- The noise's features should have a **similar size** (band-limited)

Can we write down the mathematical function to describe a natural-looking noise?

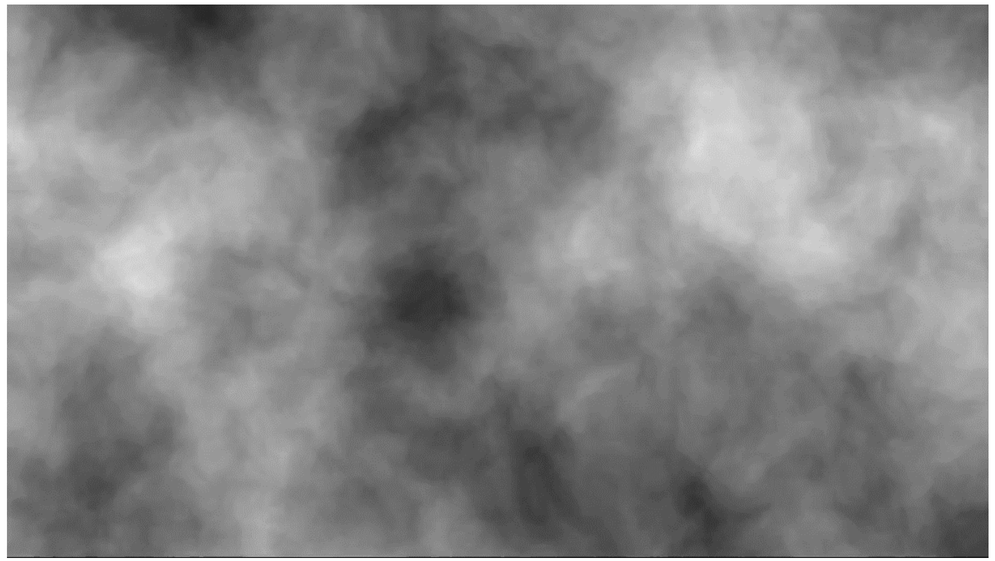
Natural Noise Functions

- Function: $\mathbf{R}^n \longrightarrow [-1, 1]$, where $n = 1,2,3,...$
- Desirable properties:
  - **No obvious repetition**
  - **Frequency band-limited**
- Fundamental building block of most procedural textures

Let's figure out how to build these natural noise functions mathematically ☺

$$f(\boldsymbol{x}) = ?$$

1D Natural Noise



2D Natural Noise

https://www.shadertoy.com/view/tddfWB

https://www.shadertoy.com/view/4lsczl

First Step: Let's create a **random** function in GLSL

- **Bad News:** There is no default random() function in GLSL. You need to implement it on your own
- **Good News:** You can implement it with one line of code
- The idea is: We want to return a "random number" with input coordinates x and y
- For the same x and y, we always return the same number
- This is called a **hash** function (or pseudorandom number, if you think of x and y as "seeds")

$$r = h(x, y)$$

Random number

Seeds

How do we design a hash function?

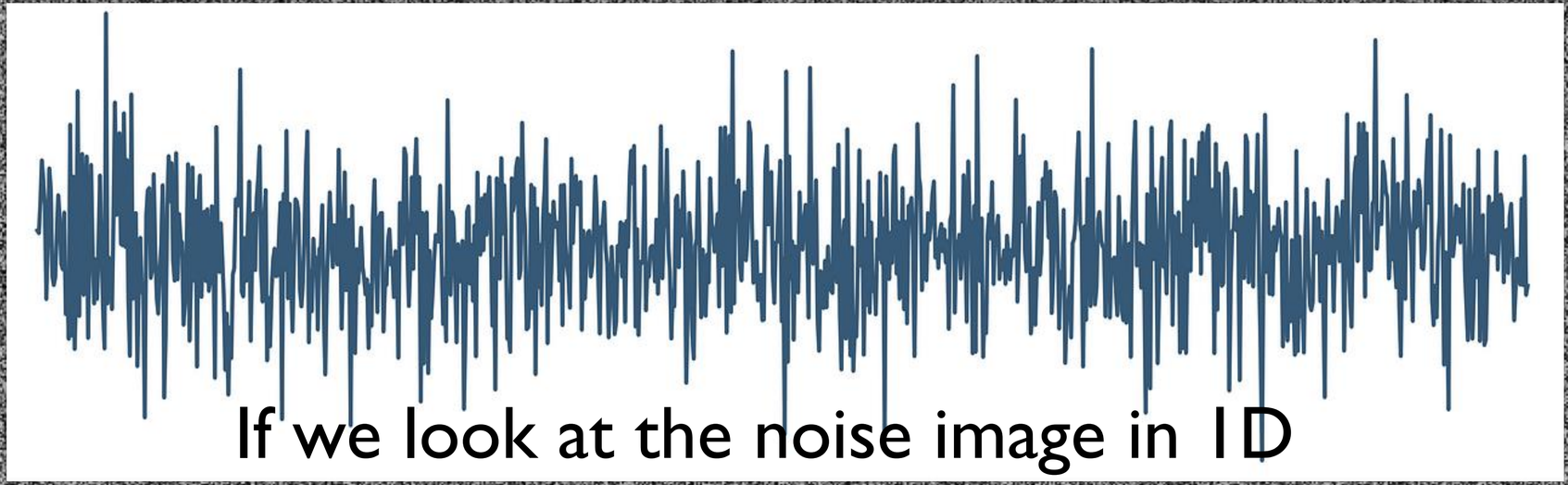
Hash Function Examples

Example:

```
float hash(vec2 v)
{
        return fract(sin(dot(v.xy ,vec2(12.9898,78.233))) * 43758.5453);
}
```

Implementation Tip:

- You may use the combination of GLSL built-in functions like sin, cos, mod, fract, dot, etc. to design your own hash (noise) function
- Search “noise” or “hash” in ShaderToy and you will find many different versions of implementation



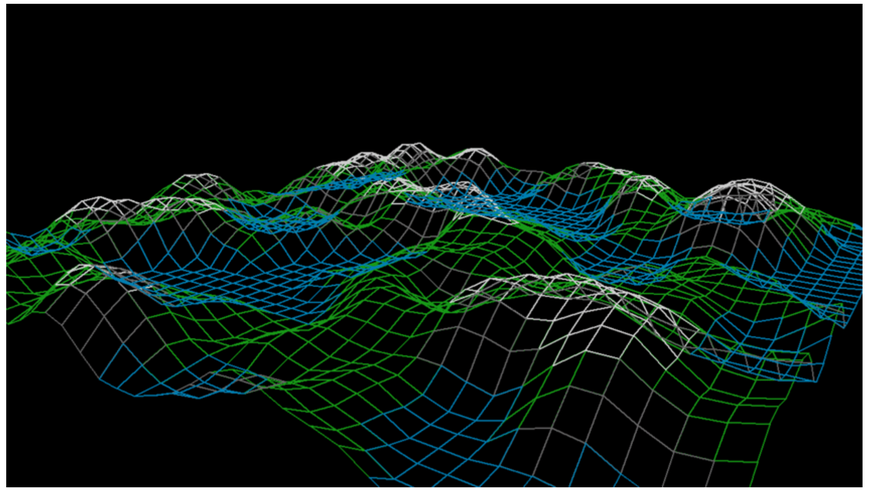
If we look at the noise image in 1D

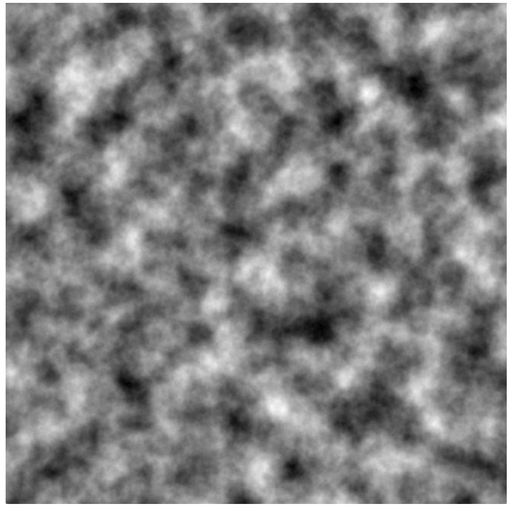
If you call the hash function for each fragment in the fragment shader:
You will get noise images that look like an untuned TV screen

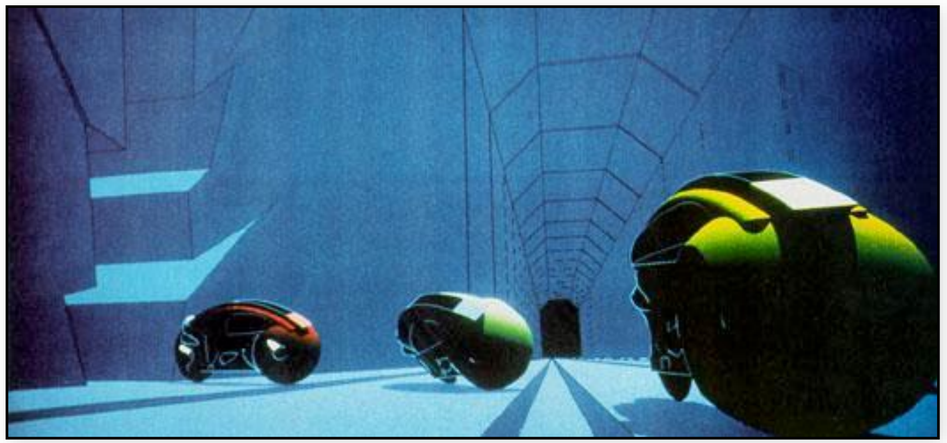
Question: How do we convert such a noise image to a natural-looking texture?

Perlin Noise

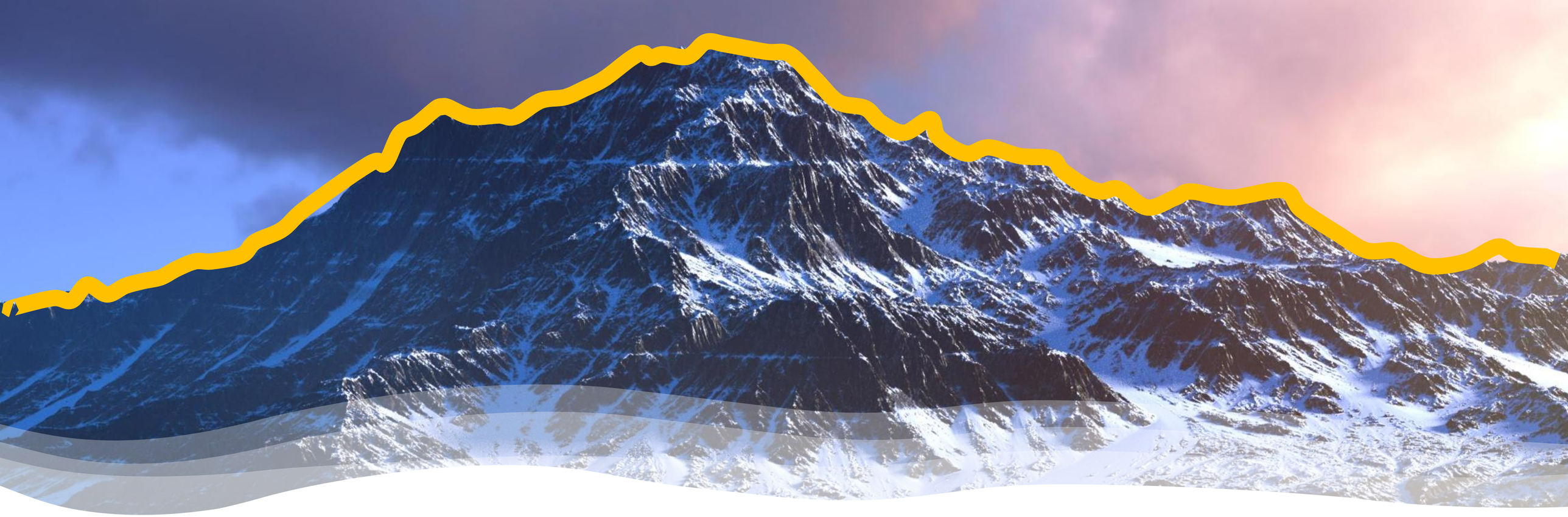
- Perlin noise, invented by Ken Perlin in 1982
  - First used in the movie Tron!
- In 1996, Ken Perlin received an Academy Award for Technical Achievement for the development of Perlin noise.







https://mrl.nyu.edu/~perlin/



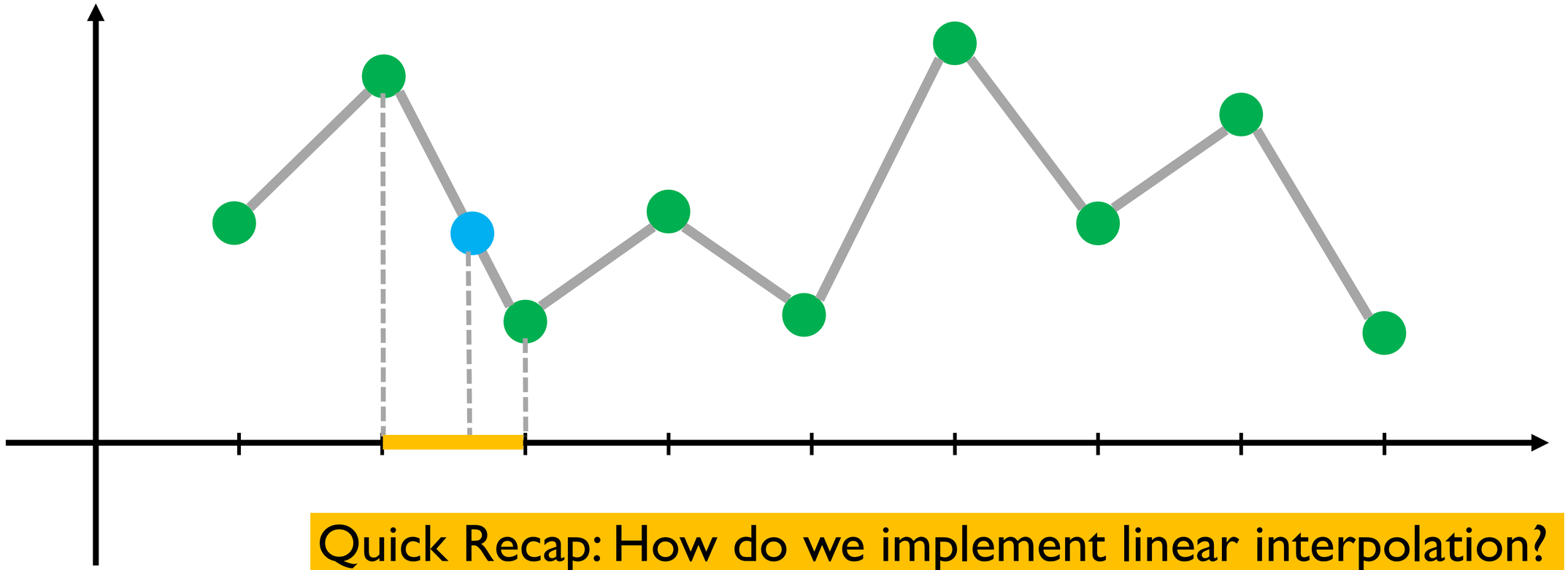
Perlin noise is one of the greatest invention in the history of computer graphics

- It serves as the mathematical foundation for all kinds of procedural model generation --- ranging from texture colors to terrain meshes

1D Perlin Noise

A Naïve Idea: Noise with Linear Interpolation

- Discretize the domain with a **uniform one-dimensional (1D) grid**
- Generate random values for each grid node using a **1D Hash() function**
- For a point, we first determine which grid cell it is in, and then interpolate its value using the neighboring grid nodes with **linear interpolation**

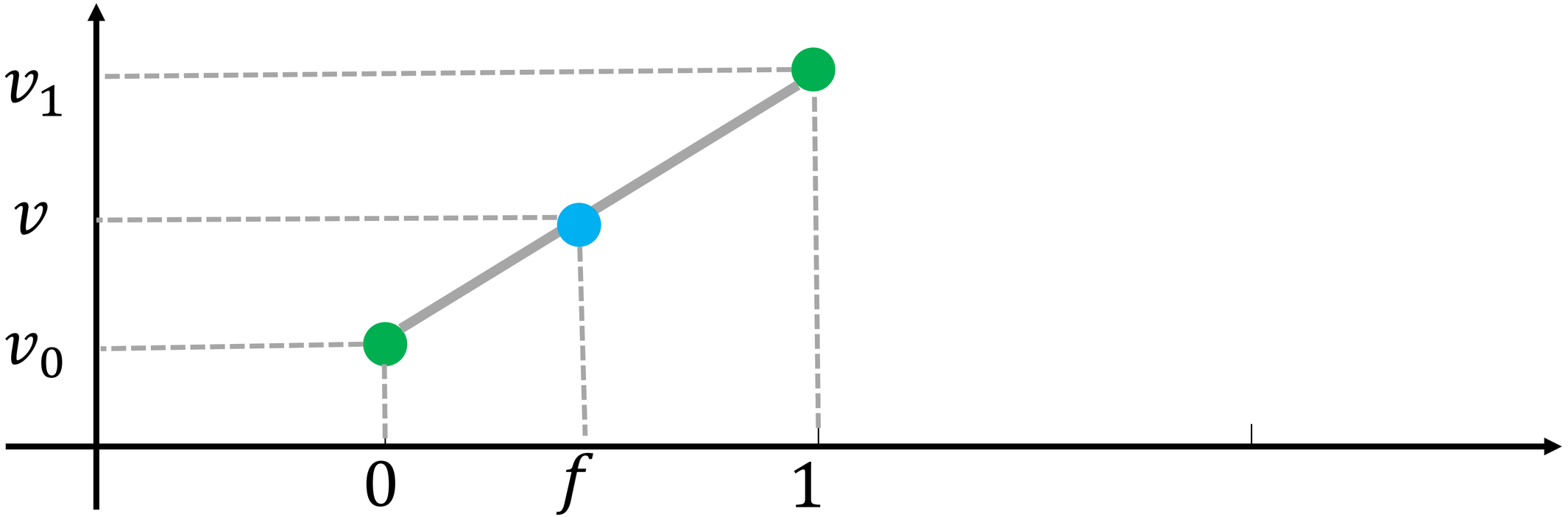


Quick Recap: How do we implement linear interpolation?

GLSL Function: **mix()**

float **mix**(float v0, float v1, float f)

- Linearly interpolate between two values
- The returned value is calculated as $\boldsymbol{v = (1 - f) * v_0 + f * v_1}$



Pseudocode for 1D **Linearly Interpolated** Noise

Input: x
Output: noise

```
int i = floor(x);
float f = fract(x);
float v0=hash(i);
float v1=hash(i+1);
float noise = (1-f)*v0+f*v1;
return noise;
```

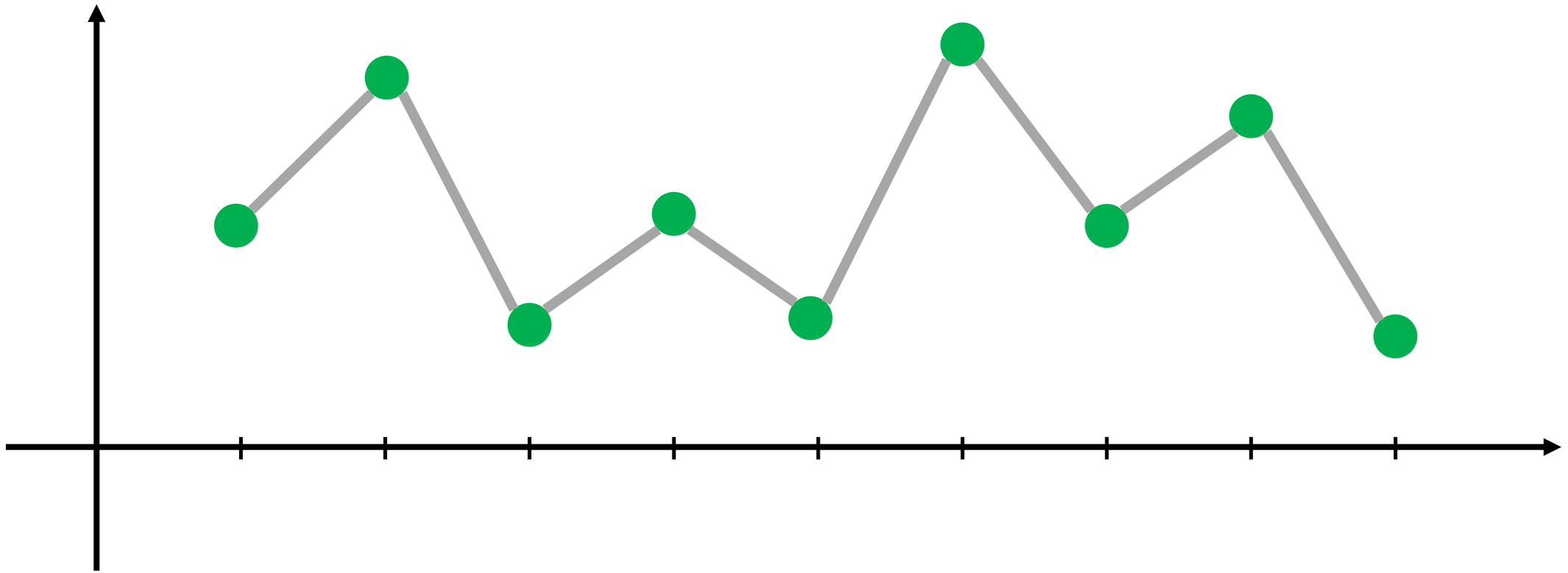
GLSL built-in **floor():**
- find the nearest integer less than or equal to the parameter
- e.g., floor(3.5) will return 3

GLSL built-in **fract():**
- compute the fractional part of the argument
- e.g., fract(1.52) will return 0.52

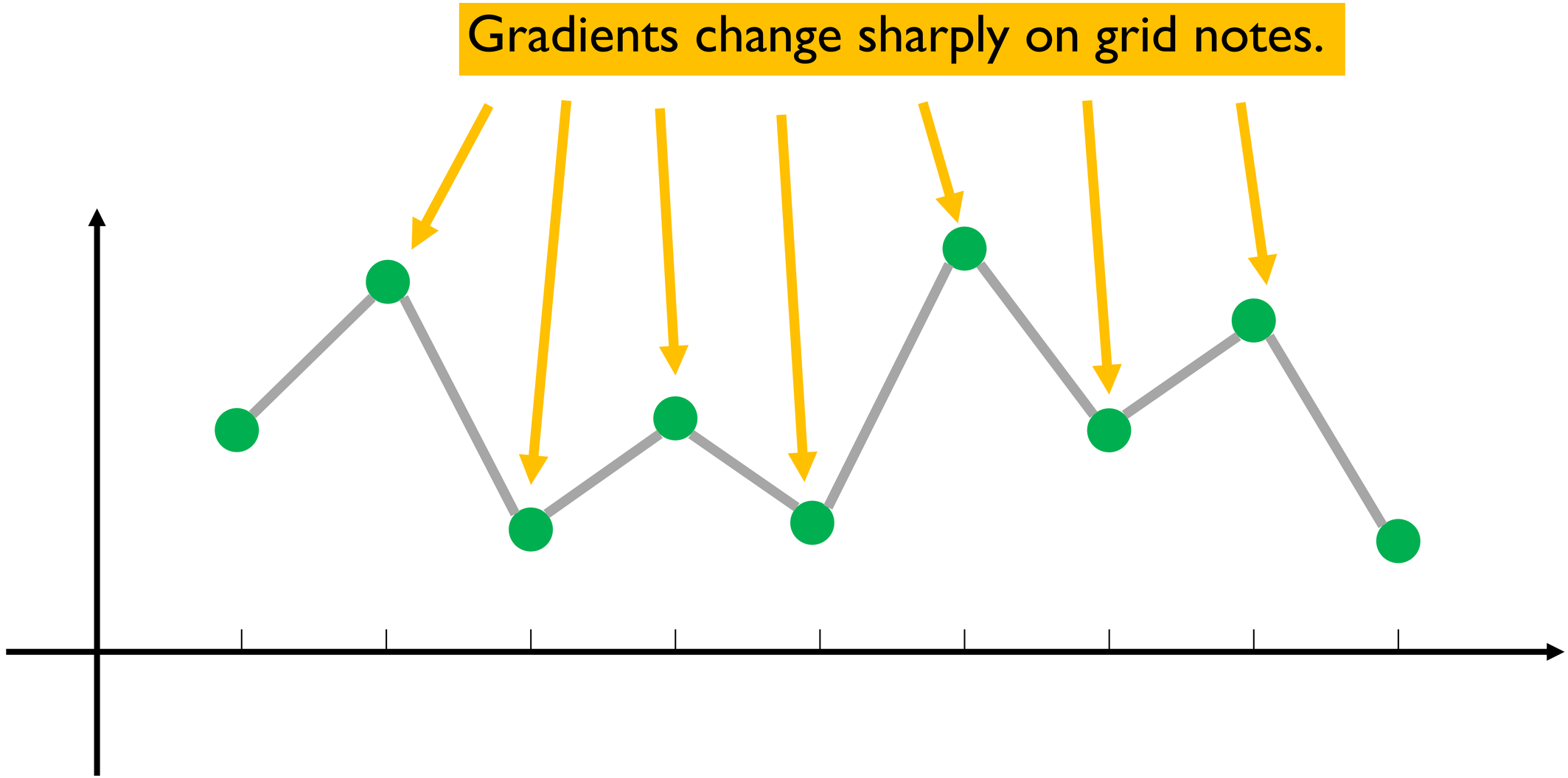
Or, we can call **mix()** in GLSL:
noise = mix(v0, v1, s);

Problems with Linear Interpolation

- **Problem I:** The noise is not smooth
- **Problem II:** The noise is not band-limited
- **Problem III:** The noise does not have multi-scale features



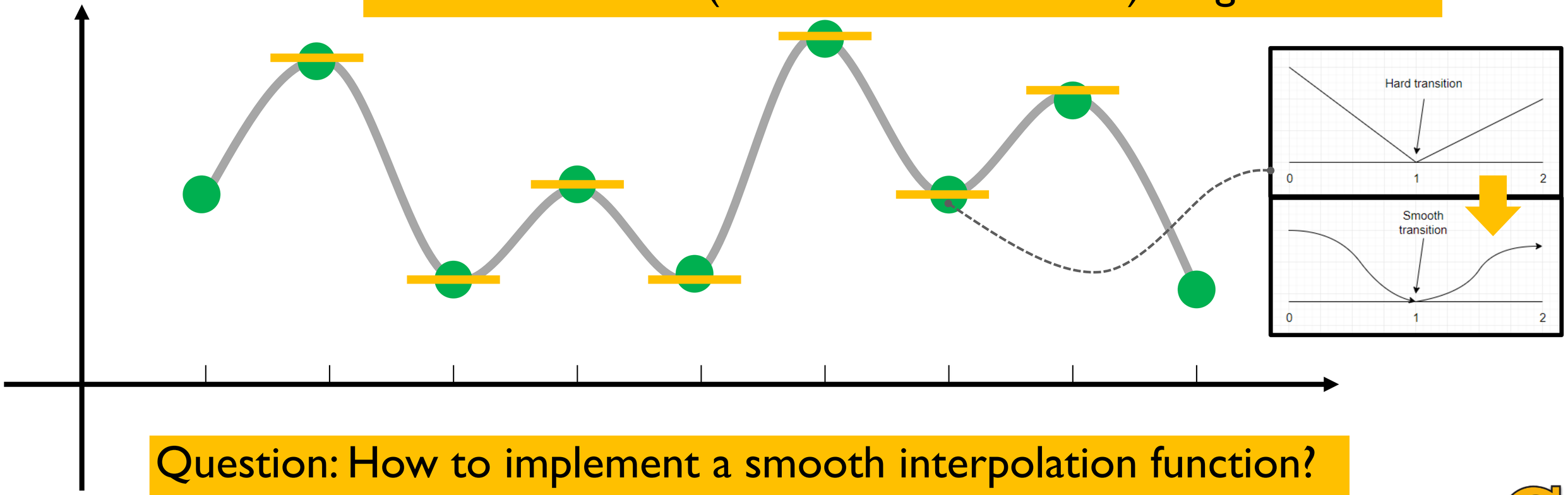
Problem 1: Values are not smooth



Solution: Smooth Interpolation

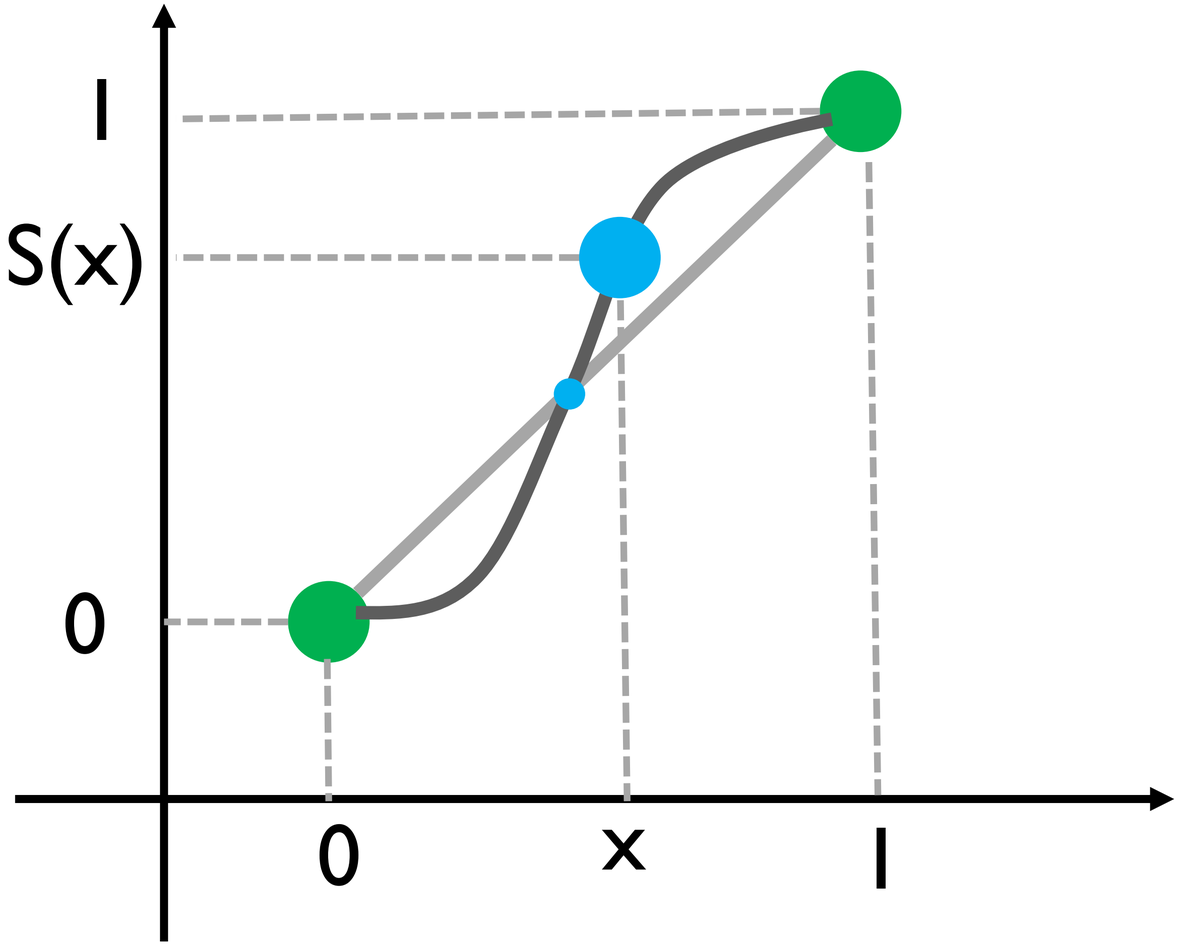
- Interpolate value from neighboring nodes using **smooth interpolation** instead of linear interpolation

Gradients are flat (and of course smooth) on grid nodes.



Question: How to implement a smooth interpolation function?

Mathematical Definition of a Smooth Interpolation



$$S(x) = 3x^2 - 2x^3 \qquad x \in [0,1]$$

Intuition: S(x) will make an x that is close to 0 (or 1) even closer to 0 (or 1)

The derivative of S(x) on x=0 and x=1 are both 0

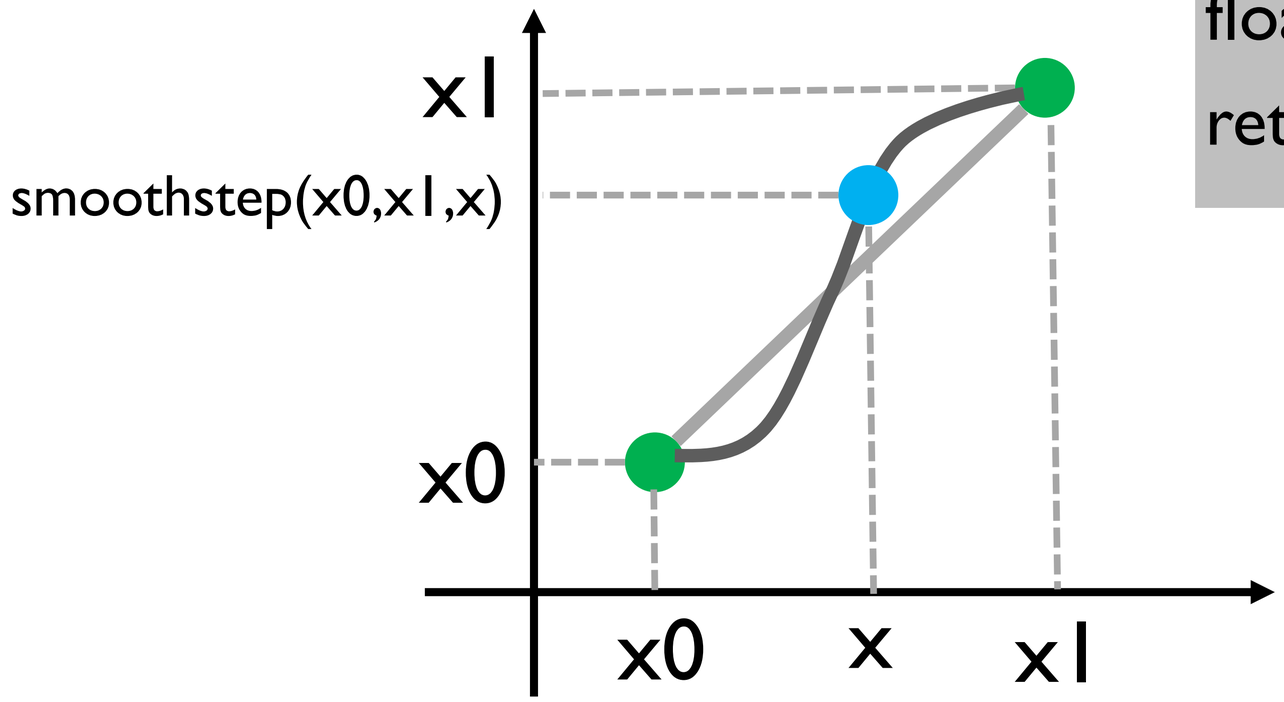
Such smooth function is also called *Hermite interpolation*, *fade function*, *ease curve*, or *smooth step* (in GLSL)

Implementation Tip:
You may either implement it with one line of code or call smoothstep in GLSL

GLSL Function: **smoothstep()**

float **smoothstep**(float x0, float x1, float x)

- Create a smooth transition between x0 and x1
- The returned value is calculated as follows:




```
float t = clamp( (x-x0)/(x1-x0),0.0,1.0);
return t*t*(3.0-2.0*t);
```

We typically use it to calculate a smoothed interpolation weight as:

float s=smoothstep(0.0,1.0,t);

by setting x0=0.0 and x1=1.0, and then use s in mix function as:

float v=mix(v0,v1,s);

Pseudocode for 1D **Smoothstep** Noise

Input: x
Output: noise

```
int i = floor(x);
float f = fract(x);
float s = f*f*(3.0-2.0*f);
float v0=hash(i);
float v1=hash(i+1);
float noise = (1-s)*v0+s*v1;
return noise;
```

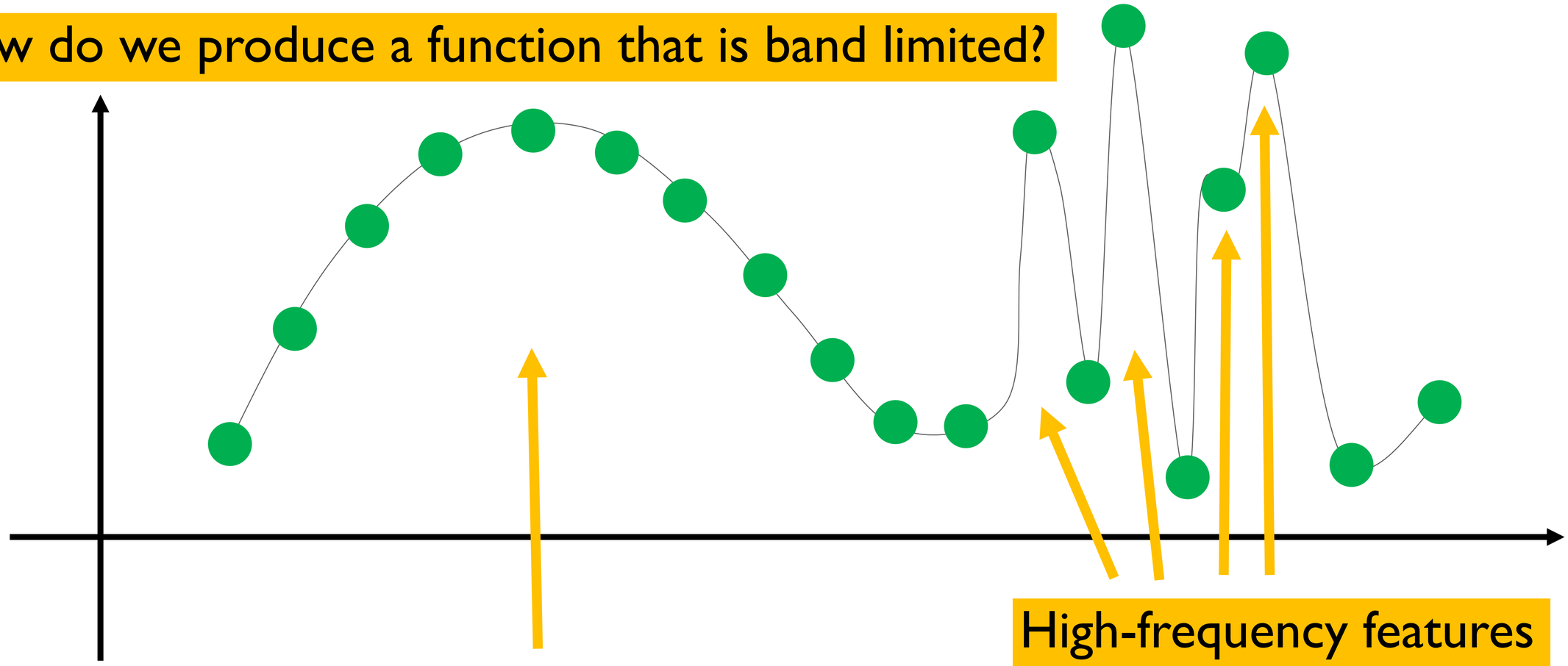
Or, we can call **smoothstep** in GLSL:
float s = smoothstep(0.0,1.0,f);

Or, we can call **mix** in GLSL:
noise = mix(v0, v1, s);

Problem II: Frequencies are not band-limited

- We might have features exhibiting dramatically different frequencies in different local regions --- some very slow slopes, and some very steep cliffs!

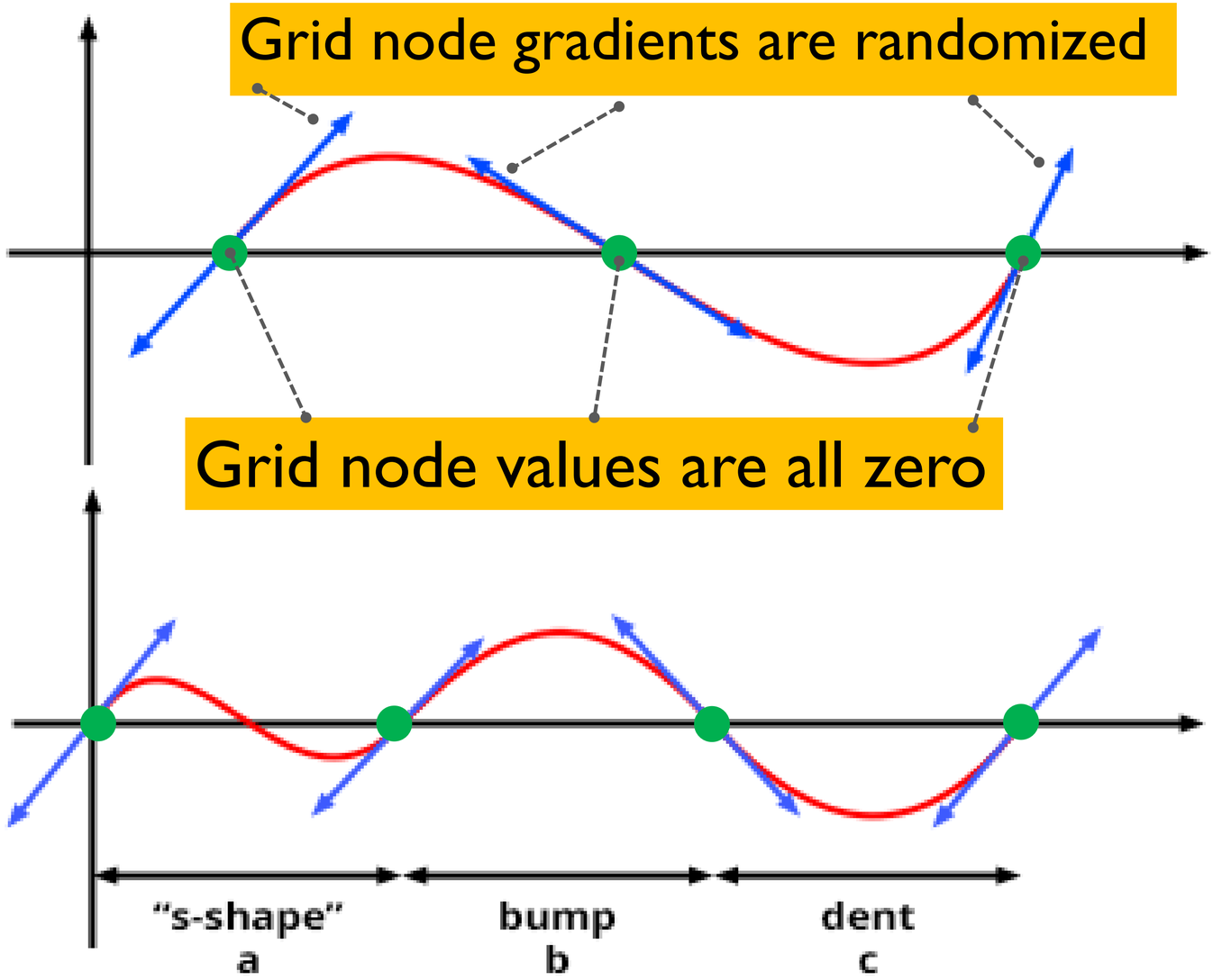
How do we produce a function that is band limited?



Low-frequency features

High-frequency features

Solution: Use Gradients in Interpolation



Key Idea: We (1) randomize the **gradient** instead of the value of a grid node, and (2) assume node values are always **zero**, and then (3) **interpolate** the function using both gradients and values on grid nodes.

All features have more or less the same size, either a "S" or a bump/dent.

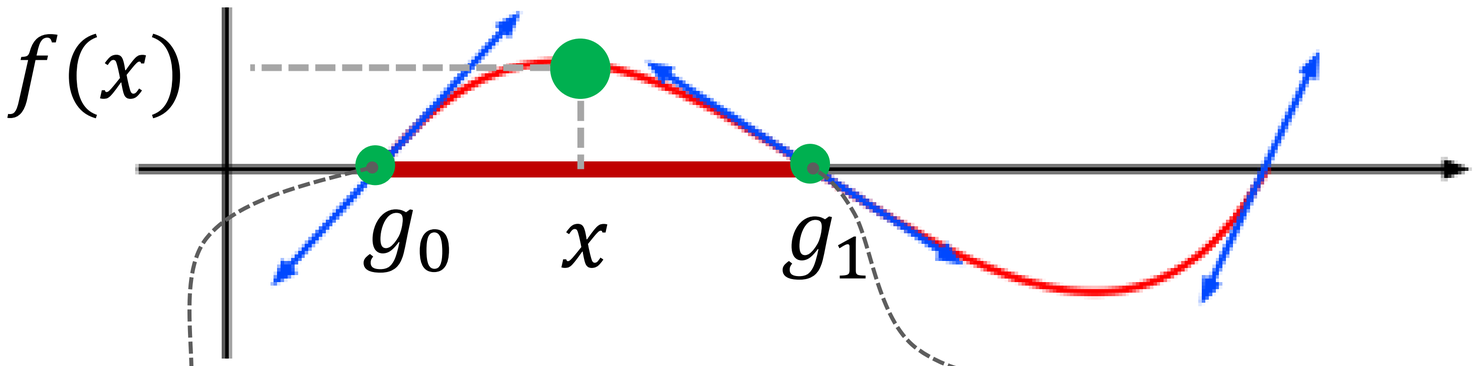
http://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds%20/perlin-noise-part-2?url=procedural-generation-virtual-worlds%20/perlin-noise-part-2

Mathematical Definition of Gradient Interpolation

- We can **design** a function satisfying both value constraints and gradient constraints on the grid nodes:

$$f(x) = \big(1 - s(x)\big)xg_0 + s(x)\,(x-1)g_1$$

with $s(x) = 3x^2 - 2x^3,\ x \in [0,1]$



$f(0) = 0$

$f'(0) = g_0$

$f(1) = 0$

$f'(1) = g_1$

Pseudocode for 1D **Gradient** Noise

Input: x
Output: noise

```
int i = floor(x);
float f = fract(x);
float s = f*f*(3.0-2.0*f);
float g0=hash(i);
float g1=hash(i+1);
float noise = (1-s)*f*g0+s*(f-1)*g1;
return noise;
```

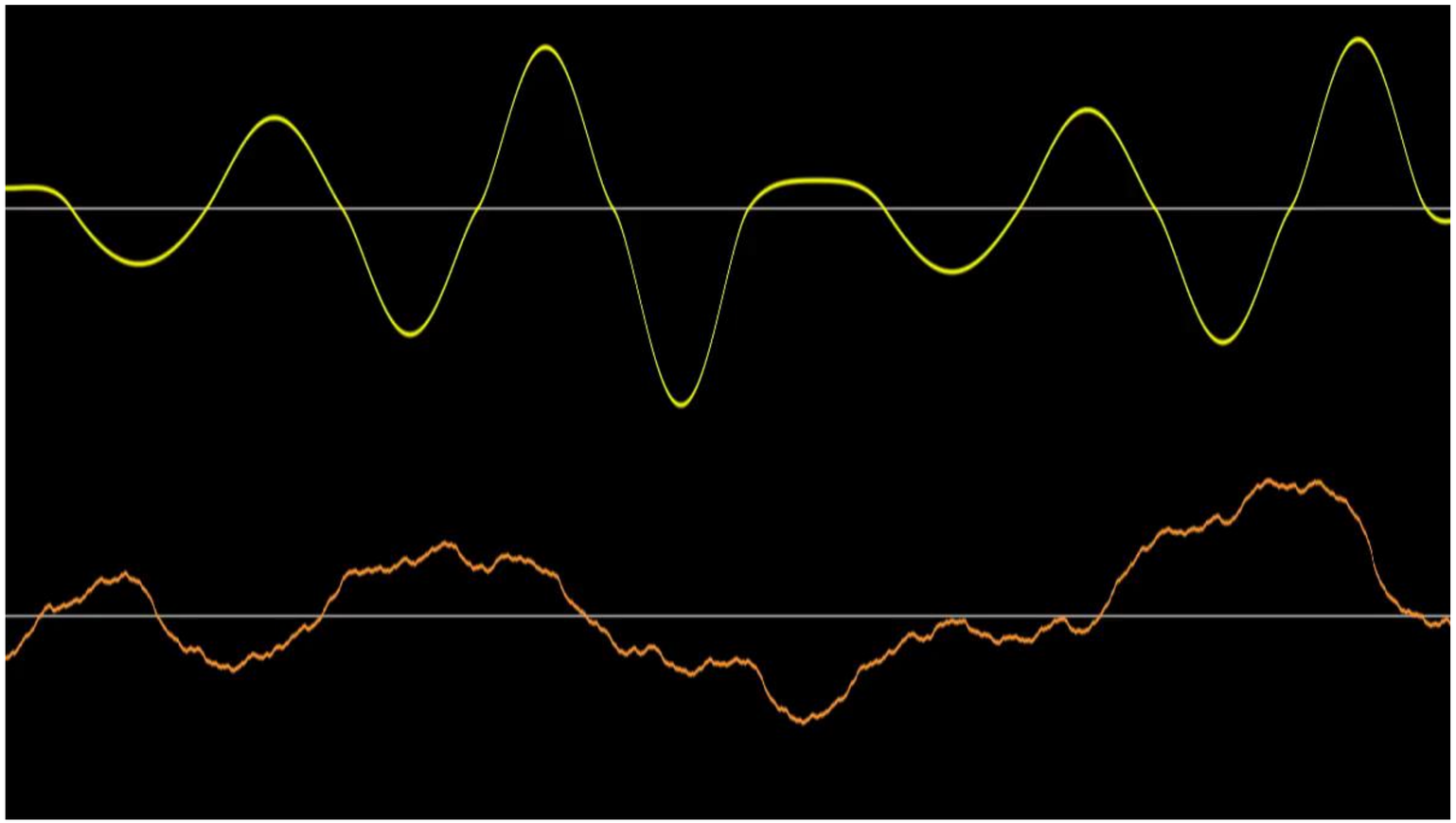
This version is the 1D Perlin noise with a single frequency.

Or, we call **smoothstep()** in GLSL:
float s = smoothstep(0.0,1.0,f);

Or, we call **mix()** in GLSL:
noise = mix(f*g0, (f-1)*g1, s);

Problem III: Synthesizing Multiple Frequencies

• Real-world textures/shapes/terrains are not always smooth: they exhibit features by mixing different frequencies together

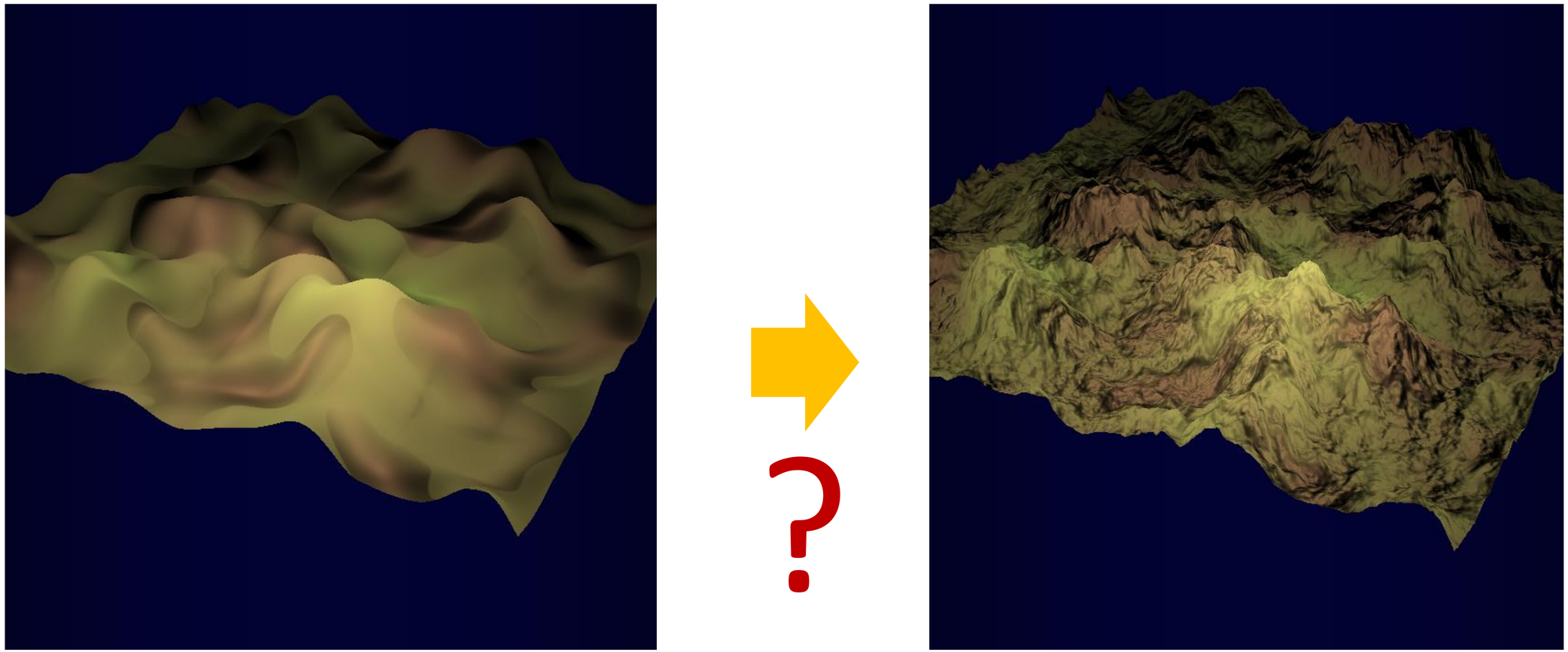


Noise with a single frequency

Noise with multiple frequencies mixed

https://www.shadertoy.com/view/3sd3Rs

Intuition: Multi-scale Features



Solution: Octave Synthesis

- Representing a complex function $f_s(x)$ by a sum of weighted contributions from a scaled function $f(x)$:

Key Idea:
double the frequency,
half the amplitude.

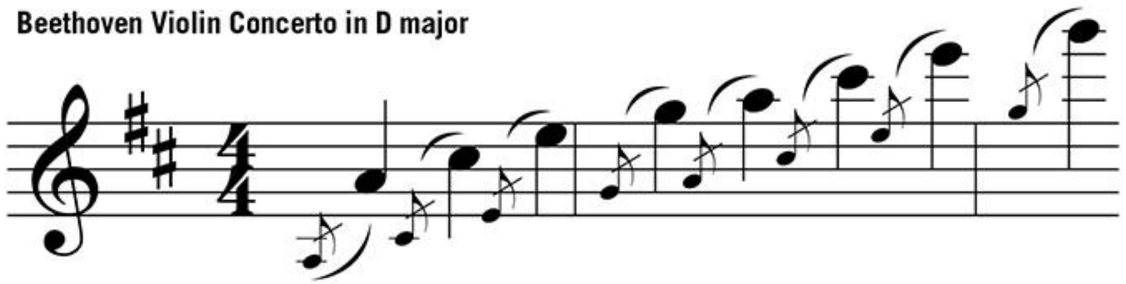
$$f_s(x) = \sum_i \omega_i f(s_i x)$$

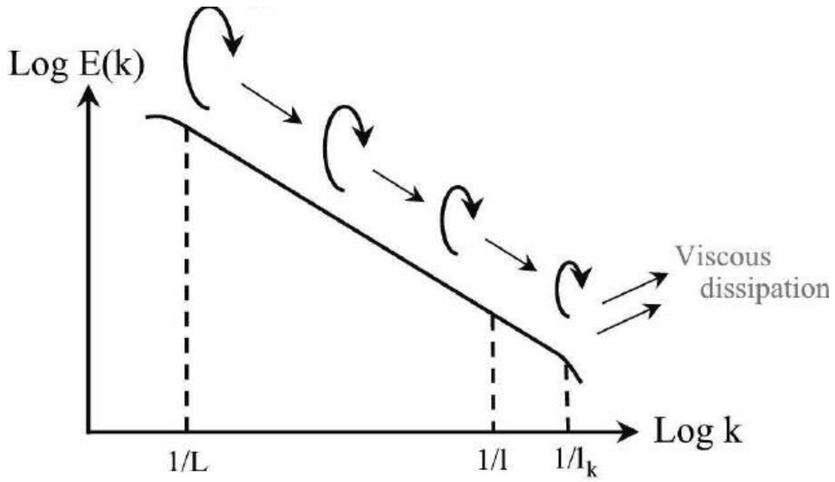
with $\omega_i = \frac{1}{2^i}, s_i = 2^i$

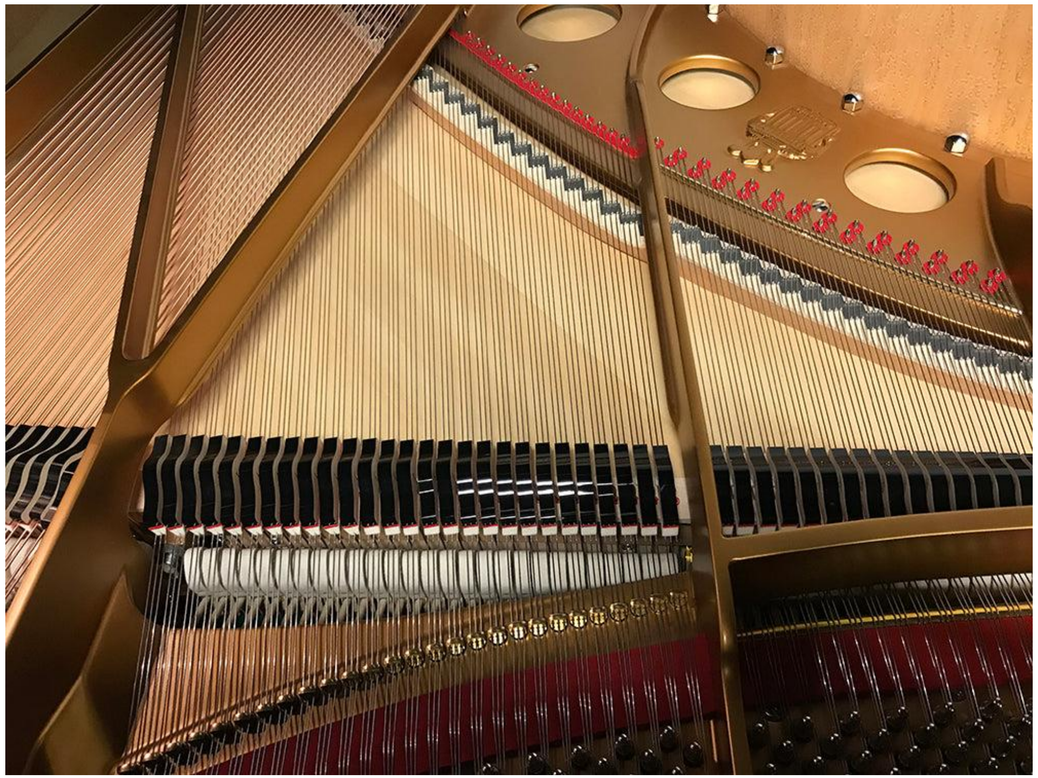
Each term in the summation is called an "**octave**"

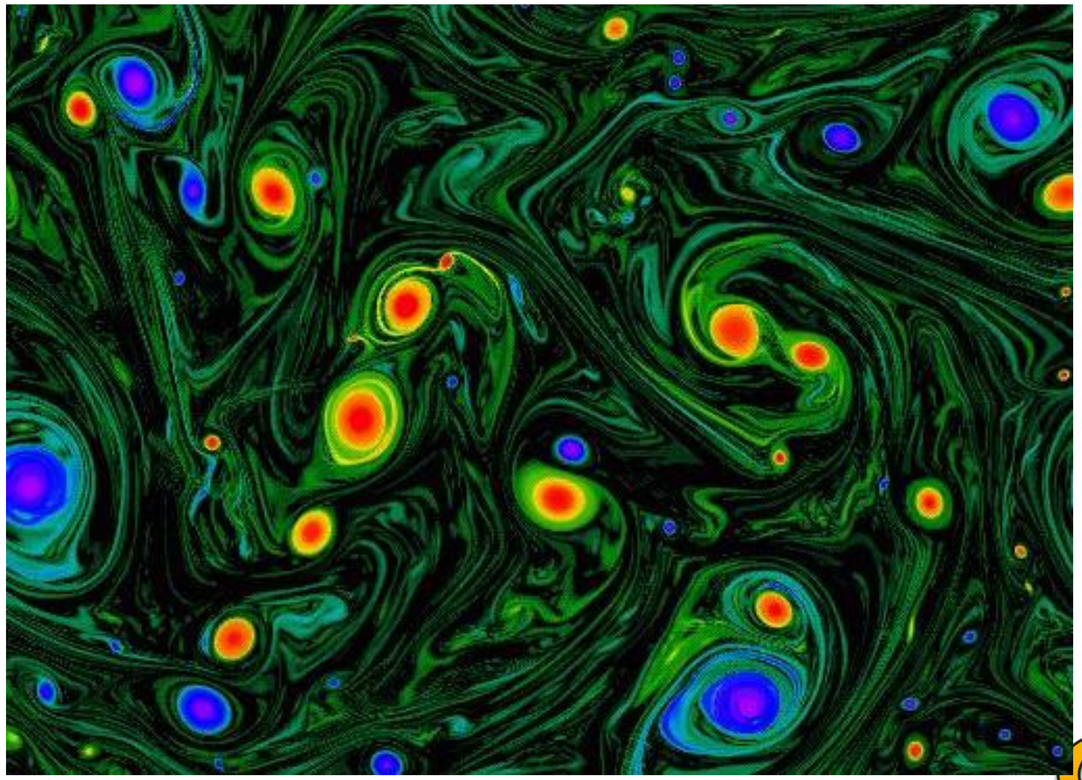
E.g., $f_s(x) = f(x) + \frac{1}{2} f(2x) + \frac{1}{4} f(4x) + \frac{1}{8} f(8x) + \cdots$

Octave

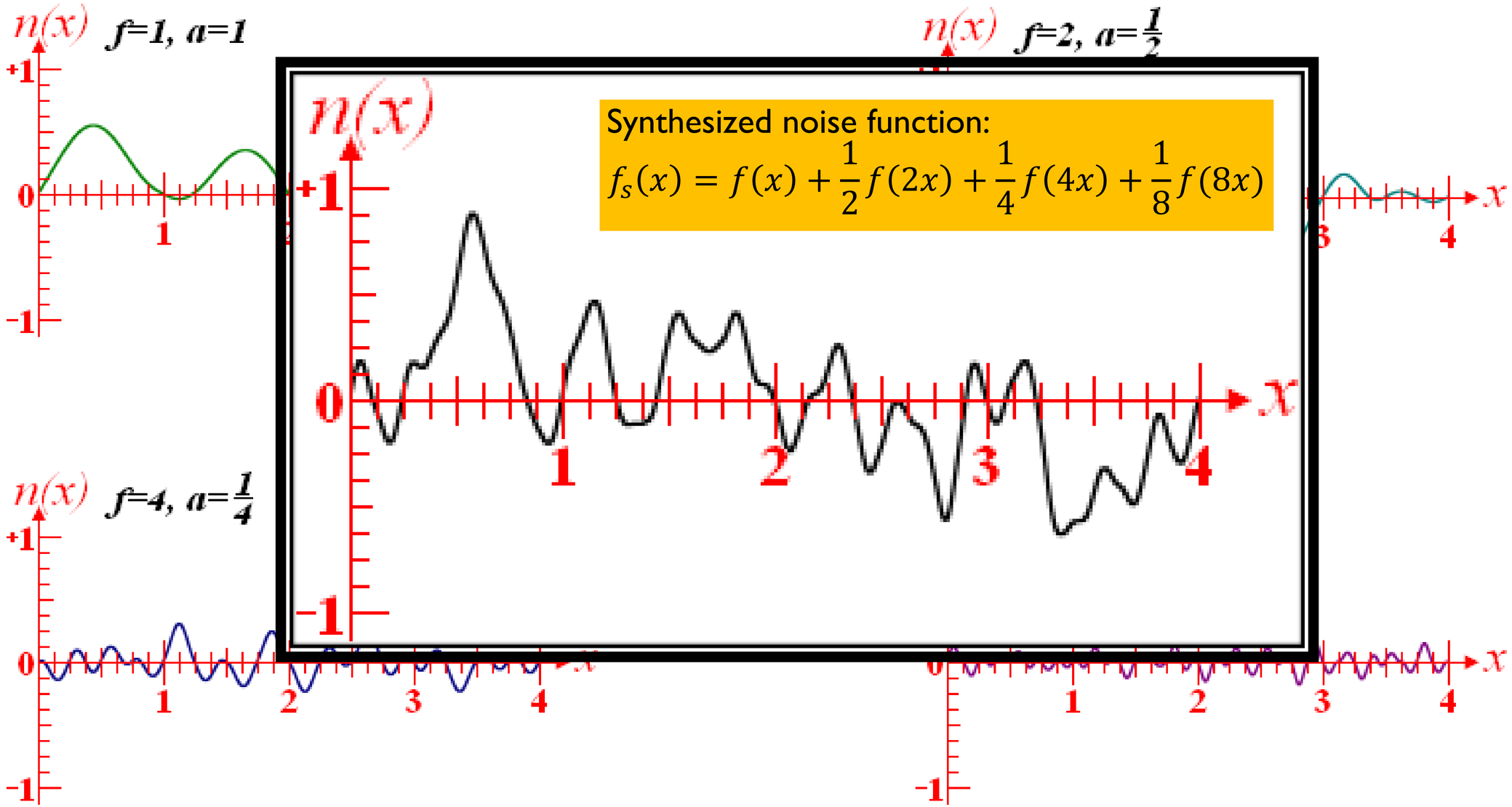

Beethoven Violin Concerto in D major


Log E(k)
Viscous
dissipation
Log k
1/L
1/l
1/l_k





Example: Noise Synthesis with Four Octaves



Pseudocode for 1D **Octave** Noise

Input: x
Output: noise

```
float noise=0.0;
for i = 0 to n-1:
        noise += 2^(-i) * perlin(2^i * x);
return noise;
```

Here **perlin()** is a pre-implemented 1D gradient noise function

That's it! We have developed the 1D Perlin noise!

- If you put together these ideas: **grid discretization, hash function, smooth step**, **gradient interpolation**, and **octave synthesis**, we will get the full model of a **1D** Perlin noise function:

$$S(x) = 3x^2 - 2x^3$$

Smooth Step

Hash Functions

$$f(x) = \big(1 - s(x)\big)x g_0 + s(x)\,(x-1)g_1$$

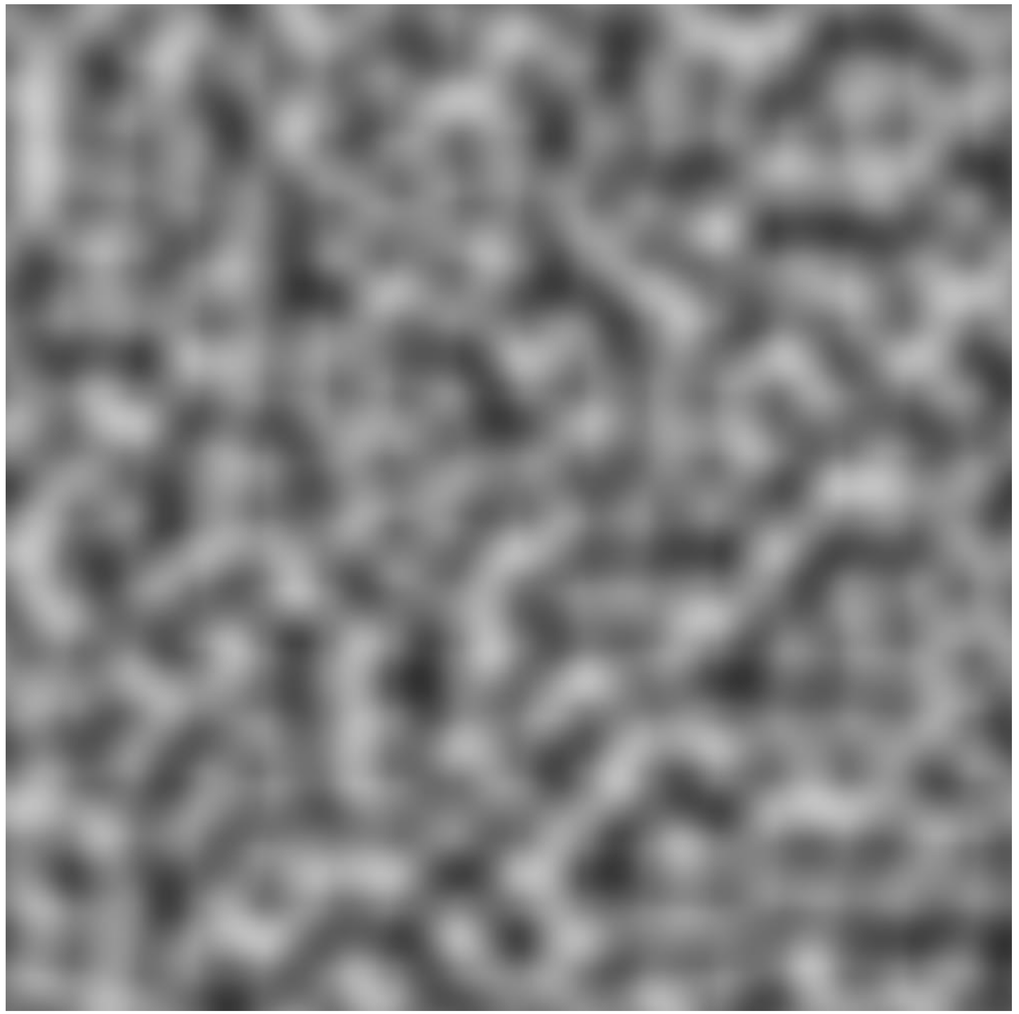
Gradient Interpolation

$$f_s(x) = f(x) + \frac{1}{2}f(2x) + \frac{1}{4}f(4x) + \frac{1}{8}f(8x) + \cdots$$

Octave Synthesis

How to go from 1D to 2D?

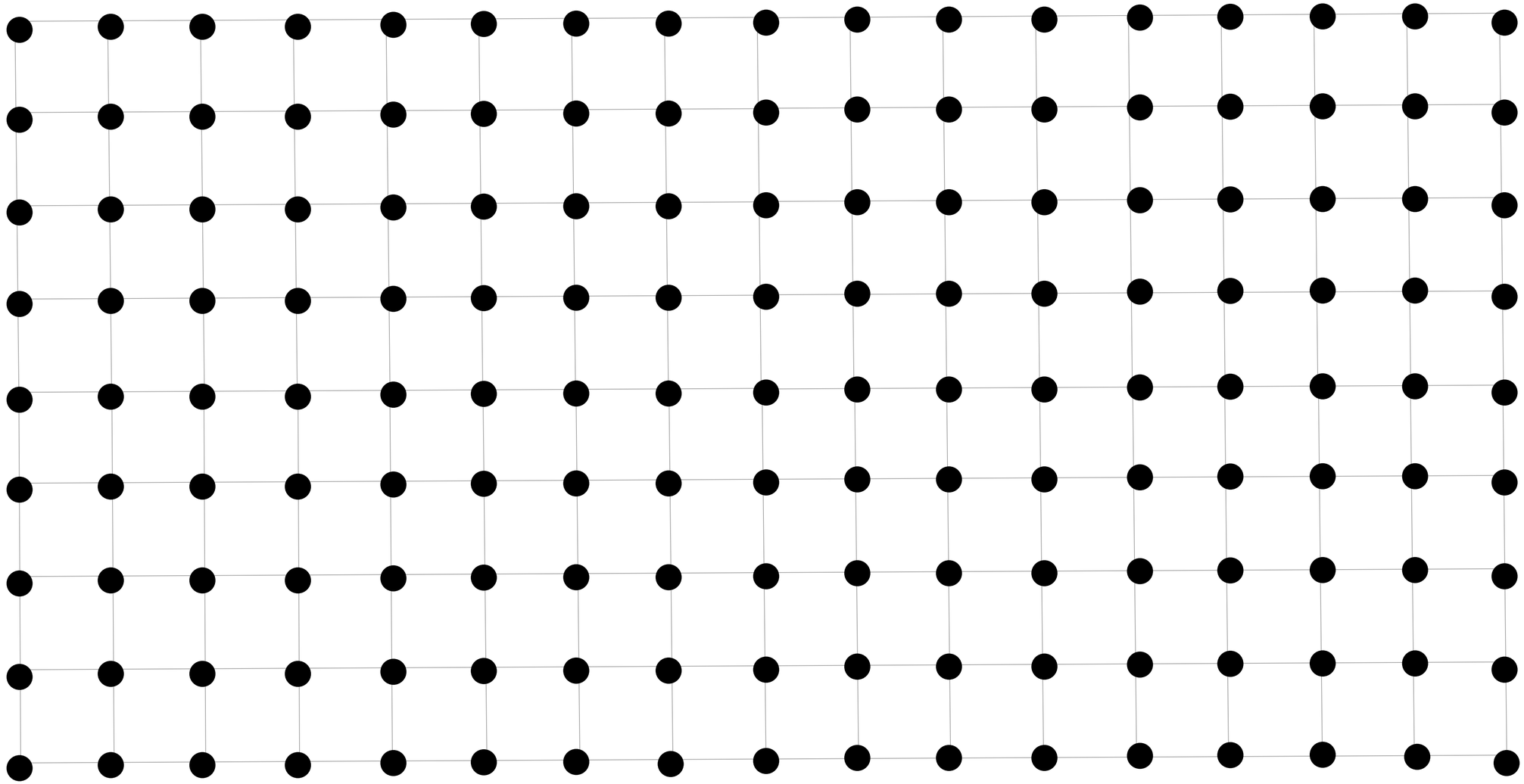
2D Perlin Noise



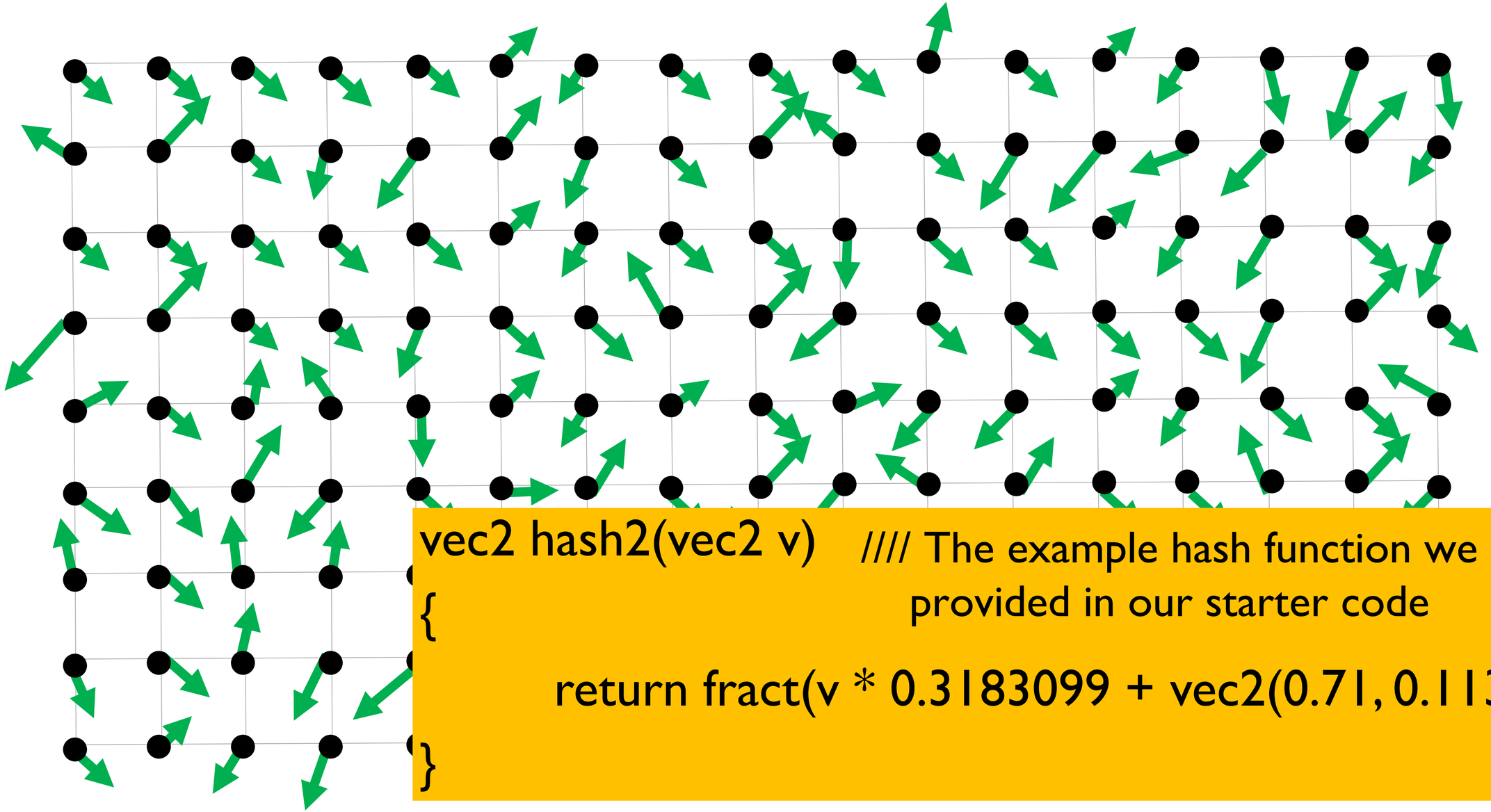
We can extend the concept of the 1D Perlin noise to calculate the 2D Perlin noise

1. Discretize the 2D domain with a 2D grid
2. Generate a randomized gradient vector for each grid nodes using a 2D hash function
3. Interpolate gradient vectors within each 2D grid cell with the smooth step function
4. Octave synthesis on 2D grids

Step 1: Discretize the domain with a grid

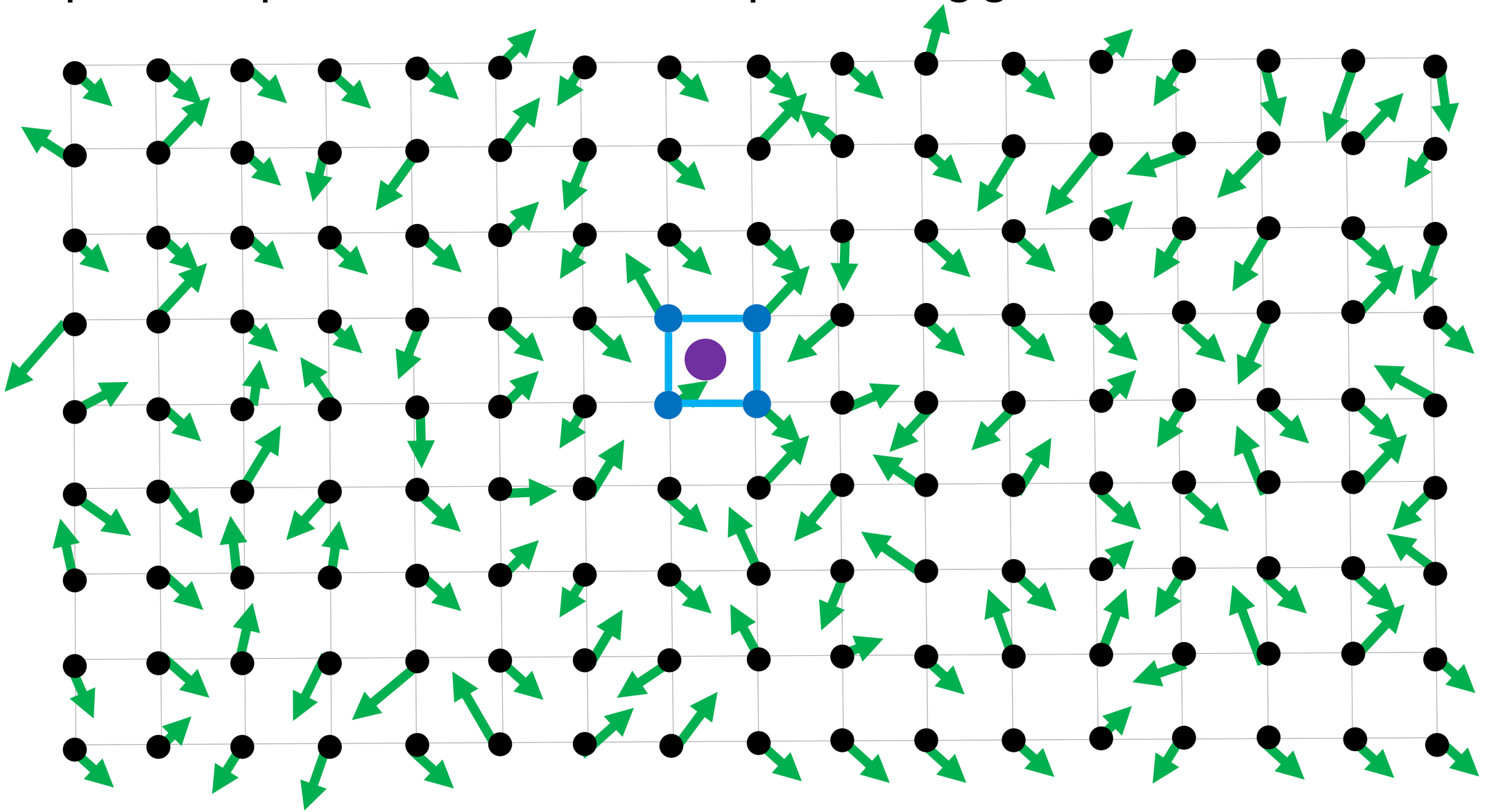


Step II: Generate **random gradient vectors** on grid nodes with hash functions

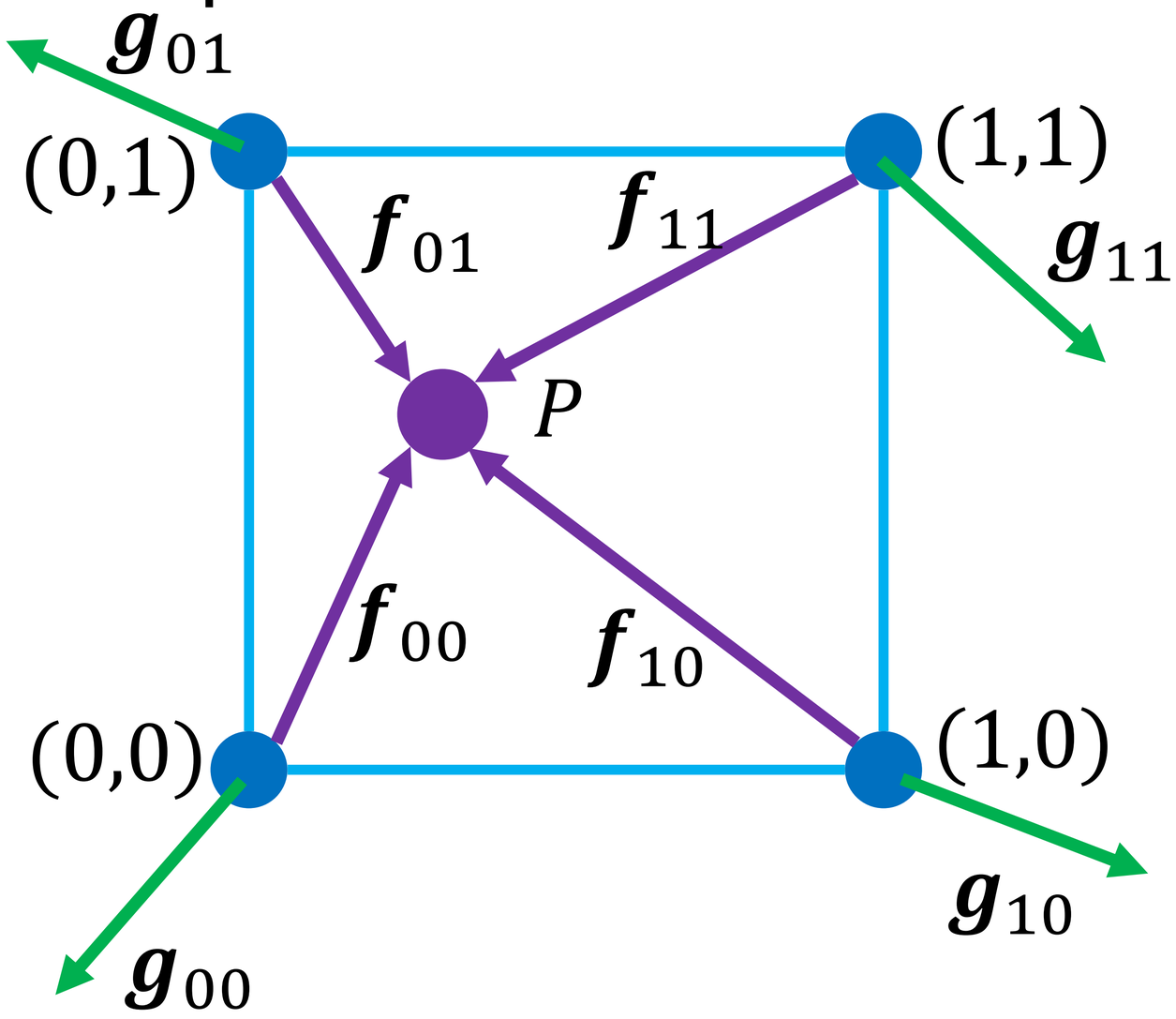


```
vec2 hash2(vec2 v)    //// The example hash function we
{                                provided in our starter code

        return fract(v * 0.3183099 + vec2(0.71, 0.113));

}
```

Step III: Interpolate noise value for a point using grid nodes



Step III.1: Calculate the **offset vector** for each grid node



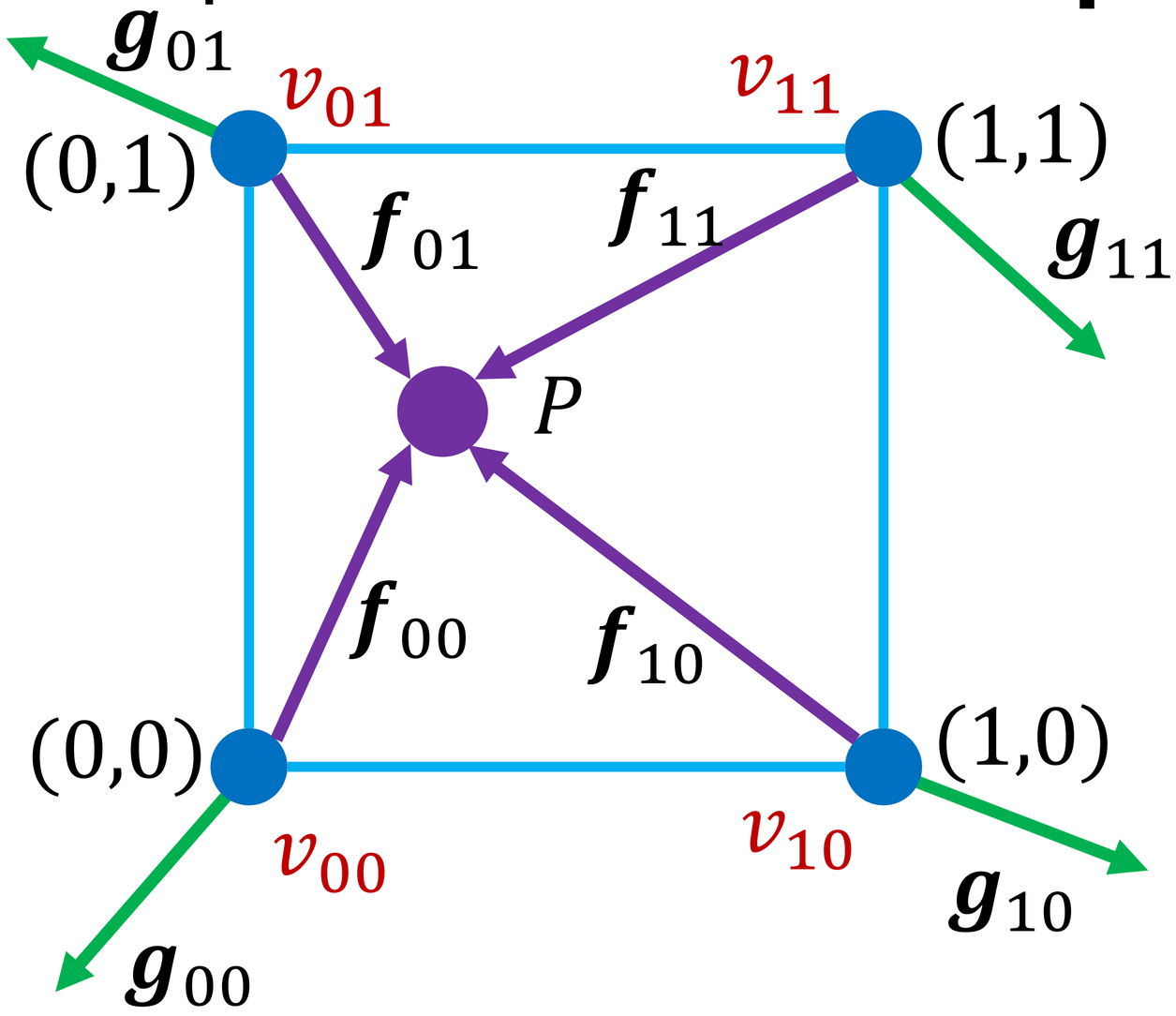
$$f_{00} = P - (0,0)$$

$$f_{10} = P - (1,0)$$

$$f_{01} = P - (0,1)$$

$$f_{11} = P - (1,1)$$

Step III.2: Calculate the **dot product** for each grid node



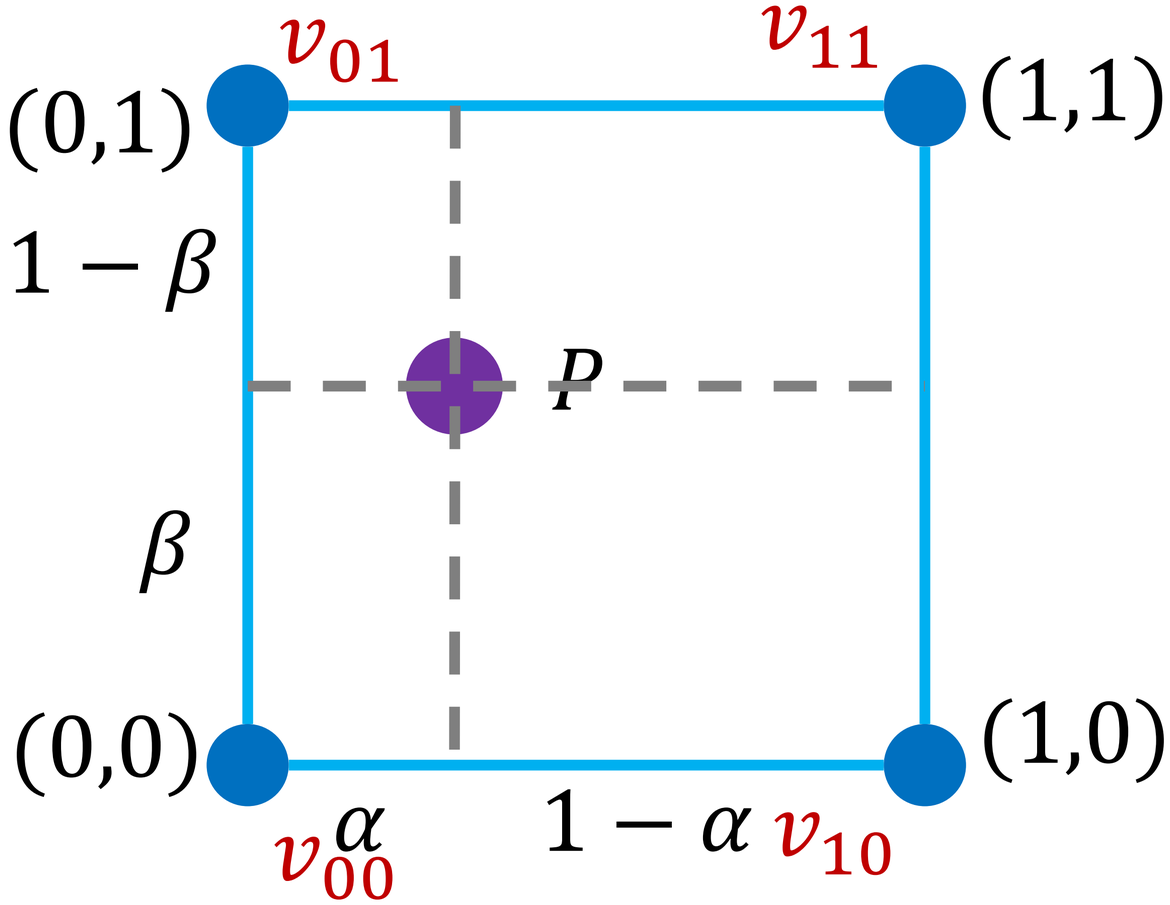
$$v_{00} = \boldsymbol{f}_{00} \cdot \boldsymbol{g}_{00}$$

$$v_{10} = \boldsymbol{f}_{10} \cdot \boldsymbol{g}_{10}$$

$$v_{01} = \boldsymbol{f}_{01} \cdot \boldsymbol{g}_{01}$$

$$v_{11} = \boldsymbol{f}_{11} \cdot \boldsymbol{g}_{11}$$

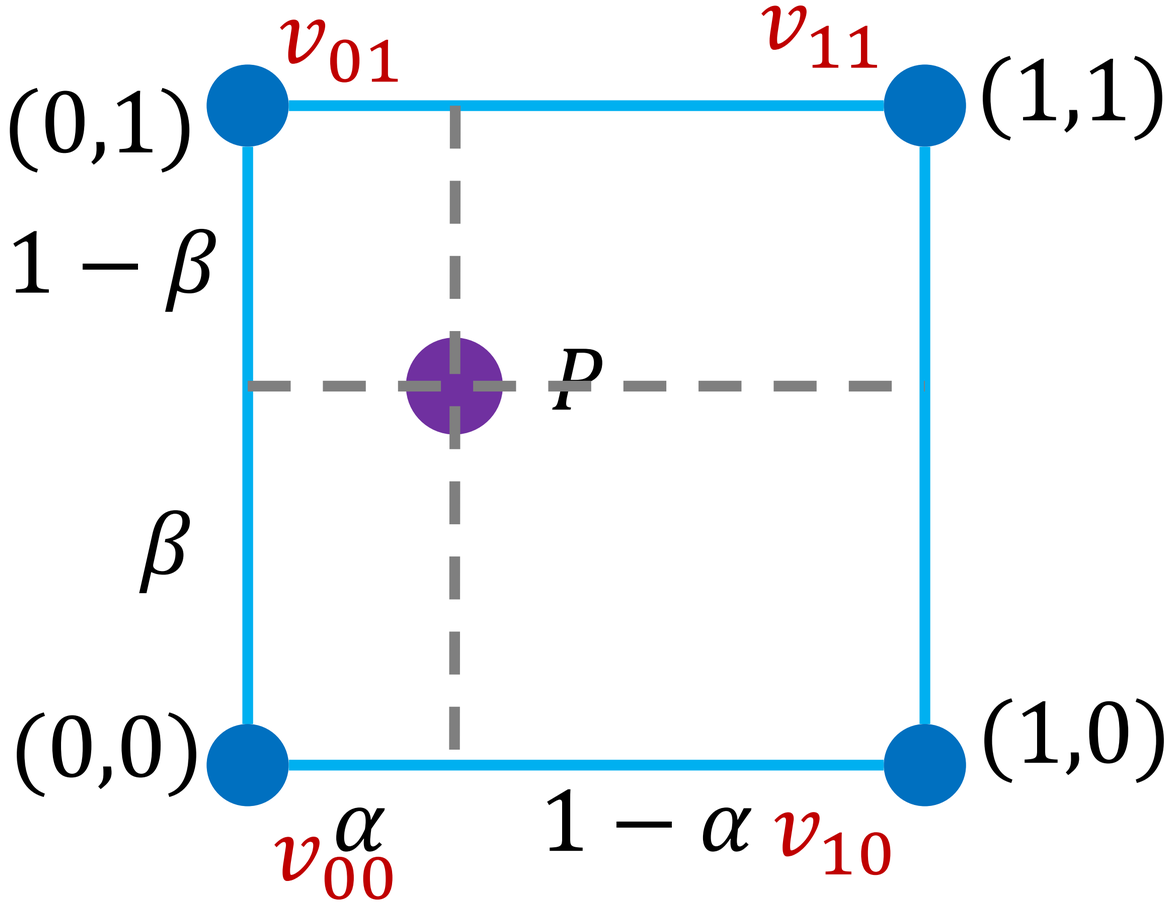
Step III.3: Use bi-linear interpolation to interpolate node values



$$
\begin{aligned}
&bilinear\_intp(P) \\
&= (1-\alpha)(1-\beta)v_{11} \\
&+ \alpha(1-\beta)v_{21} \\
&+ (1-\alpha)\beta v_{12} + \alpha\beta v_{22}
\end{aligned}
$$

One missing point here: We need to use $S(\alpha), S(\beta)$ instead of α and β to enable a smoothed interpolation!

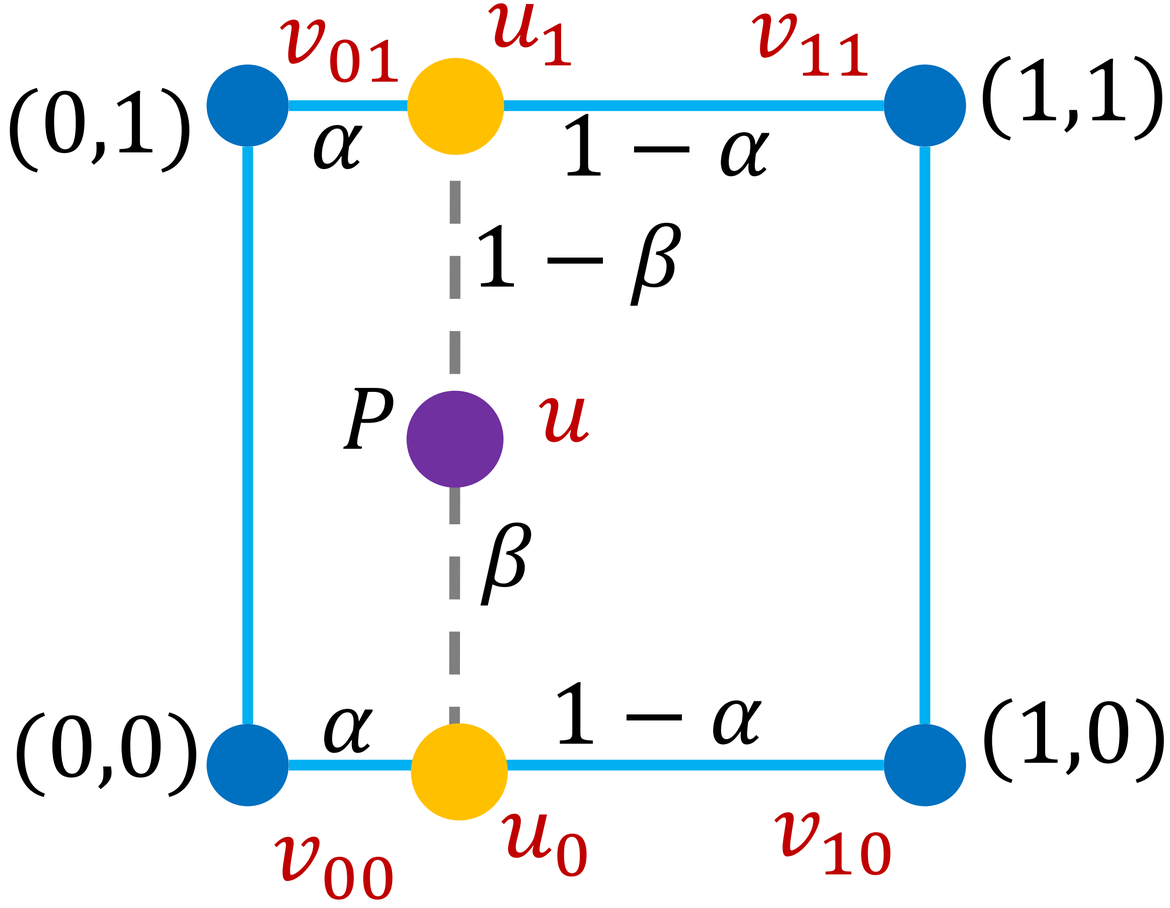
Step III.3: Use **bi-smoothed** interpolation to calculate interpolation



$$
\begin{aligned}
&smoothed\_intp(P) \\
&= \big(1 - S(\alpha)\big)\big(1 - S(\beta)\big)v_{11} \\
&+ \alpha\big(1 - S(\beta)\big)v_{21} \\
&+ \big(1 - S(\alpha)\big)\beta v_{12} \\
&+ S(\alpha)S(\beta)v_{22}
\end{aligned}
$$

with $S(x) = 3x^2 - 2x^3, x \in [0,1]$

For **implementation**, this interpolation can be realized with three linear interpolations



$$u_0 = (1 - S(\alpha))v_{00} + S(\alpha)v_{10}$$

$$u_1 = (1 - S(\alpha))v_{01} + S(\alpha)v_{11}$$

$$u = (1 - S(\beta))u_0 + S(\beta)u_1$$

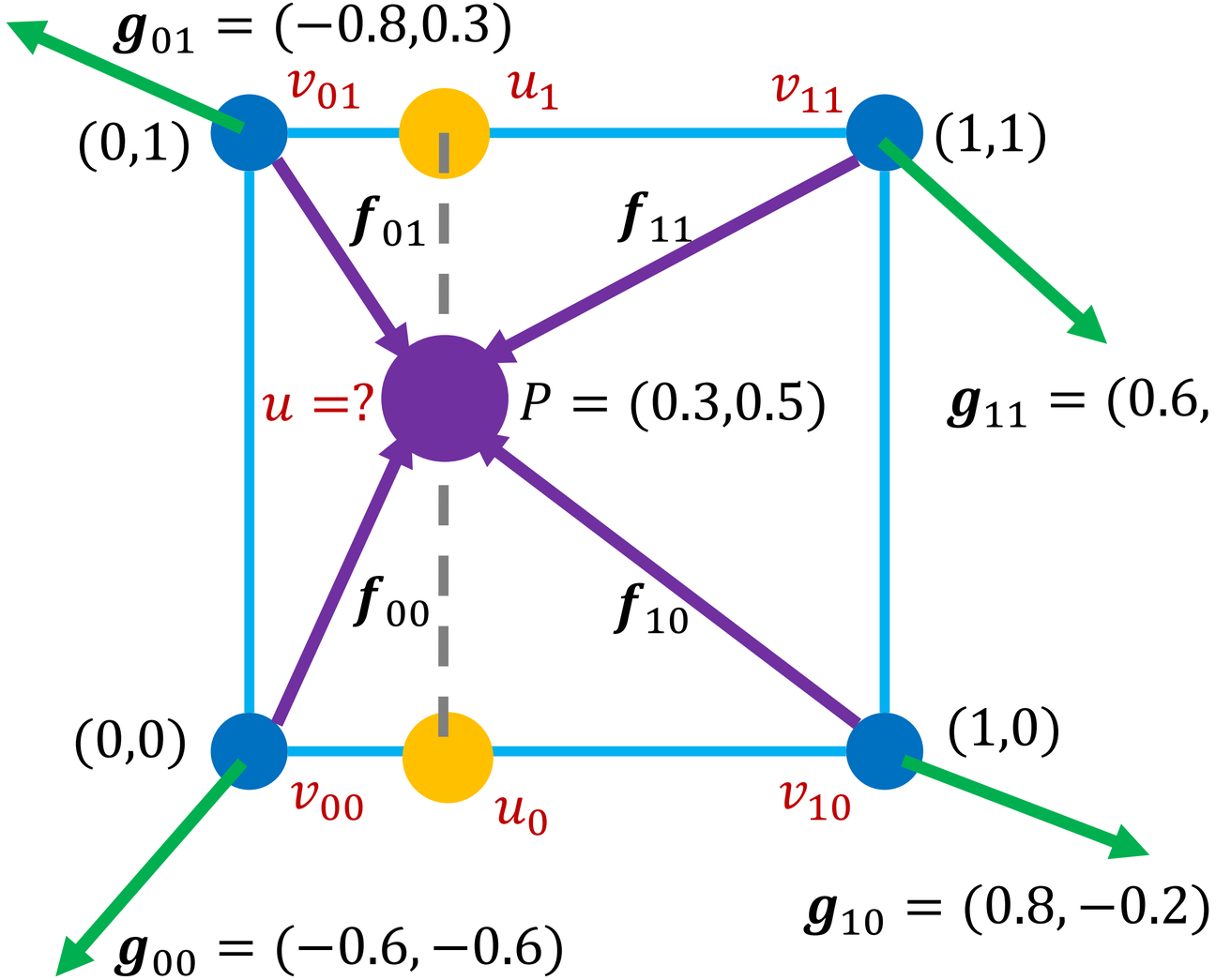
Or, with GLSL implementation:

```
u0=mix(v00,v10,S(alpha))
u1=mix(v01,v11,S(alpha))
u=mix(u0,u1,S(beta))
```

Or, even simpler:

```
u=mix(mix(v00,v10,S(alpha)), mix(v01,v11,S(alpha)),S(beta))
```

A Numerical Example: Calculate Perlin Noise for Point P



$$\boldsymbol{f}_{00} = \boldsymbol{P} - (0,0) = (0.3, 0.5)$$

$$\boldsymbol{f}_{10} = \boldsymbol{P} - (1,0) = (-0.7, 0.5)$$

$$\boldsymbol{f}_{01} = \boldsymbol{P} - (0,1) = (0.3, -0.5)$$

$$\boldsymbol{f}_{11} = \boldsymbol{P} - (1,1) = (-0.7, -0.5)$$

$$v_{00} = \boldsymbol{f}_{00} \cdot \boldsymbol{g}_{00} = -0.48$$

$$v_{10} = \boldsymbol{f}_{10} \cdot \boldsymbol{g}_{10} = -0.66$$

$$v_{01} = \boldsymbol{f}_{01} \cdot \boldsymbol{g}_{01} = -0.39$$

$$v_{11} = \boldsymbol{f}_{11} \cdot \boldsymbol{g}_{11} = -0.17$$

$$u_0 = (1 - S(0.3))v_{00} + S(0.3)v_{10} = -0.52$$

$$u_1 = (1 - S(0.3))v_{01} + S(0.3)v_{11} = -0.34$$

$$u = (1 - S(0.5))u_0 + S(0.5)u_1 = -0.43$$

Pseudocode for 2D Perlin Noise with a Single Octave

Input: **x**
Output: noise

```
vec2 i = floor(x);
vec2 f = fract(x);
vec2 s = f*f*(3.0-2.0*f);
vec2 g00 = hash2(i);
vec2 g10 = hash2(i+vec2(1,0));
vec2 g01 = hash2(i+vec2(0,1));
vec2 g11 = hash2(i+vec2(1,1));
vec2 f00 = f-vec2(0,0);
vec2 f10 = f-vec2(1,0);
vec2 f01 = f-vec2(0,1);
vec2 f11 = f-vec2(1,1);
float noise=(calculate with g00,g10,g01,g11, f00, f10, f01, f11, and s)
return noise;
```

This step can be implemented with three **mix()** function calls

Step IV: Octave Synthesis

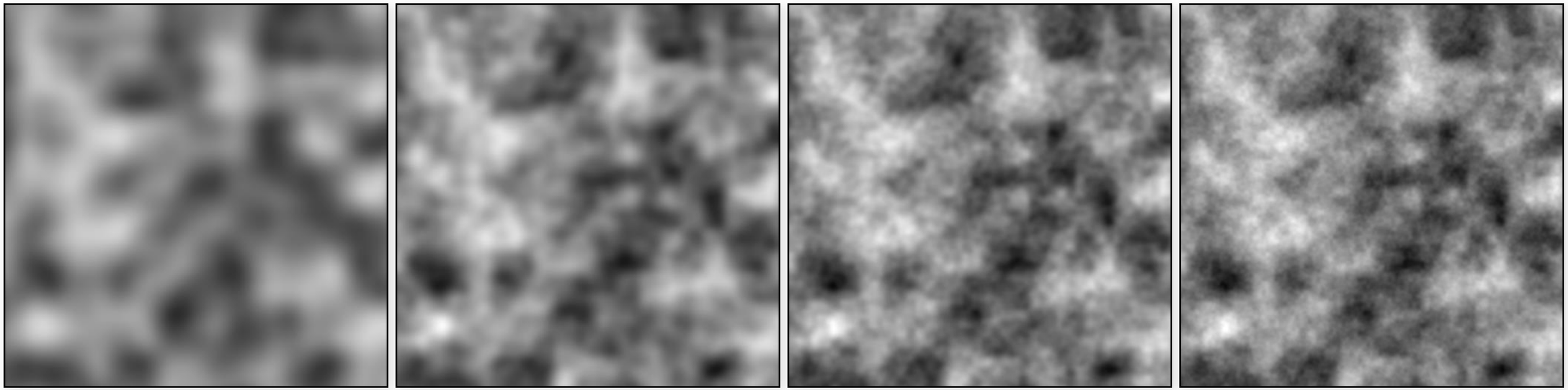
Input: x, n //// x – position, n – number of octaves

Output: noise

```
float noise=0.0;
for i = 0 to n-1:
        noise += 2^(-i) * perlin(2^i * x);
return noise;
```

Here **perlin()** is the 2D Perlin noise function implemented previously

Octave Synthesis on a Texture Image



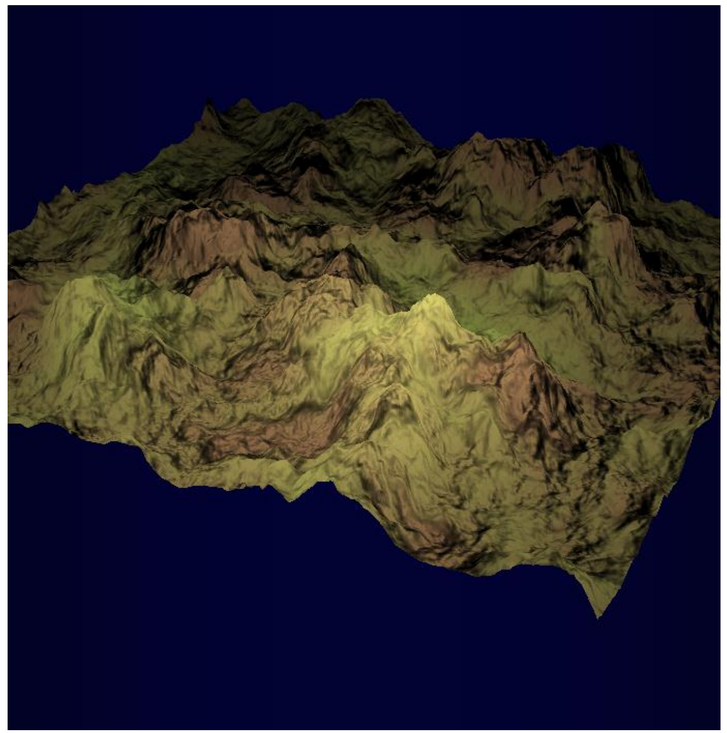
1 octaves 2 octaves 3 octaves 4 octaves

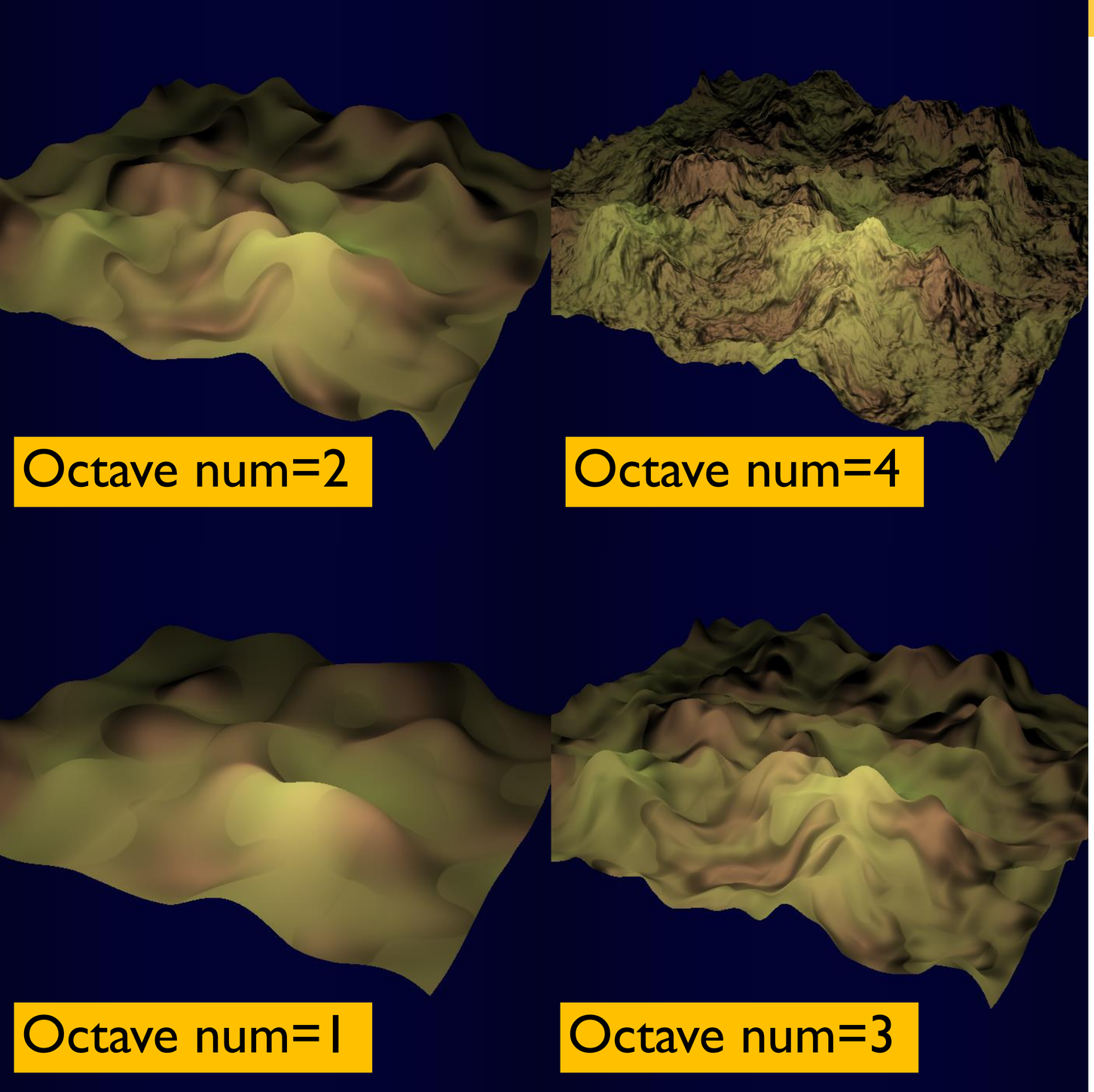
This technique is also called "fractal Brownian Motion" (*fBM*), or simply "fractal noise"

Key Takeaways for Perlin Noise

1. **Grid Discretization:** Generate a random gradient vector on each grid node using Hash() function
2. **Random Gradients:** For each grid node, dot product between gradient vectors and offset vectors
3. **Smooth Interpolation:** For an arbitrary point with a grid cell, interpolate the dot products on the four grid nodes using smooth step function and return it as the noise value
4. **Octave Synthesis:** Synthesize the noise values on different octaves

Terrain Generation

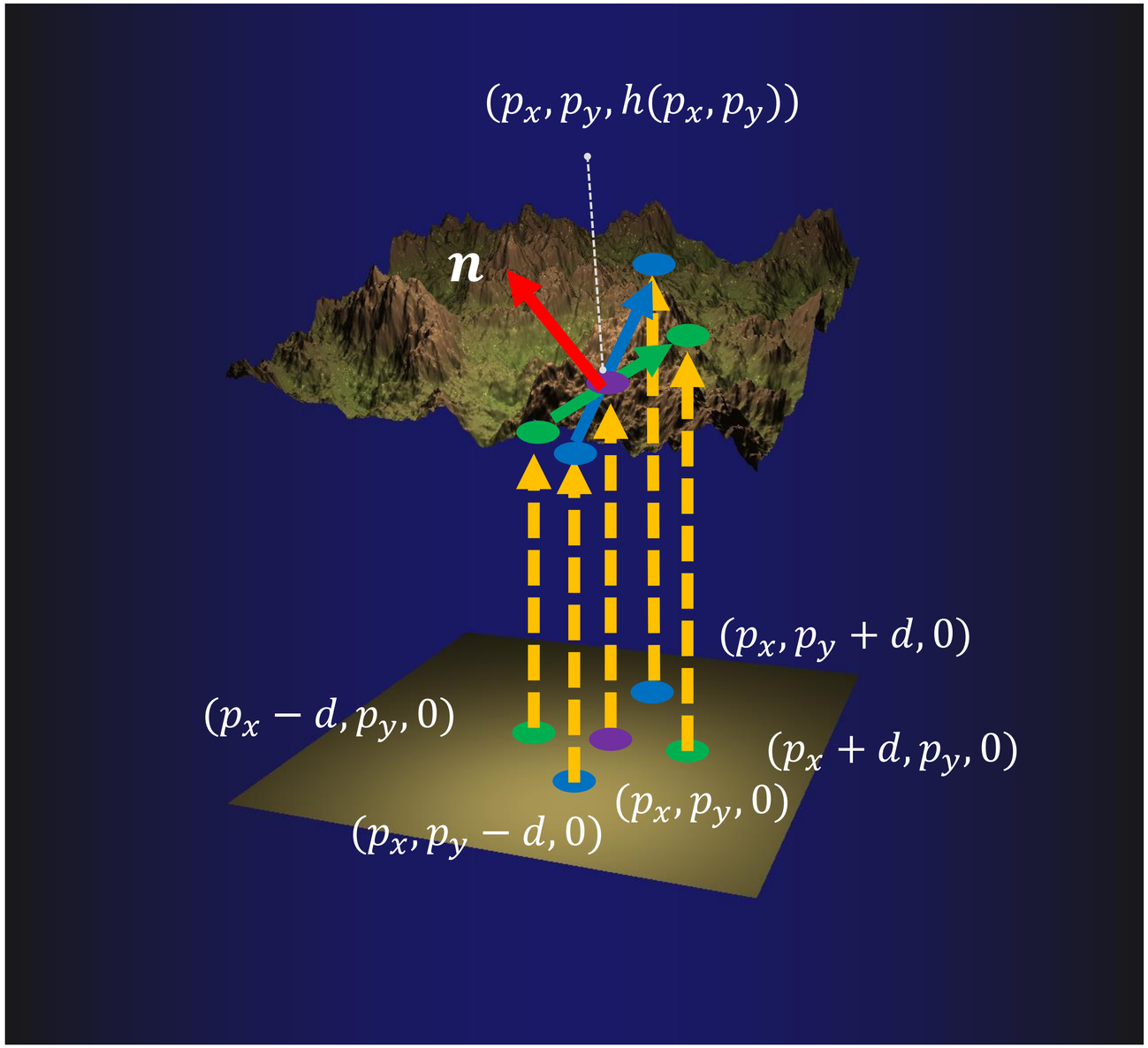




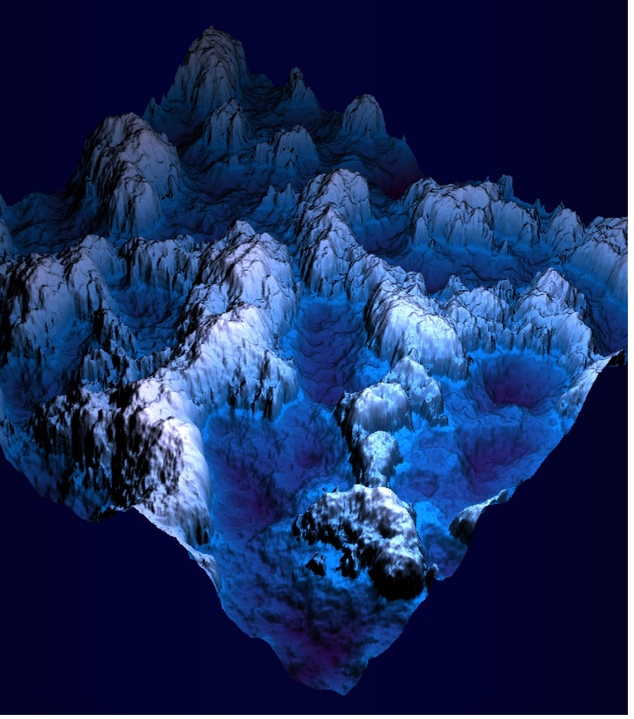
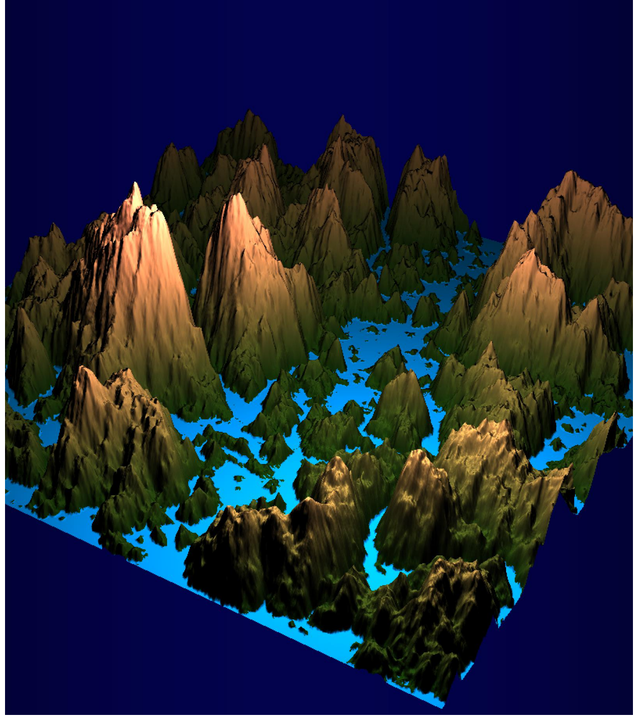
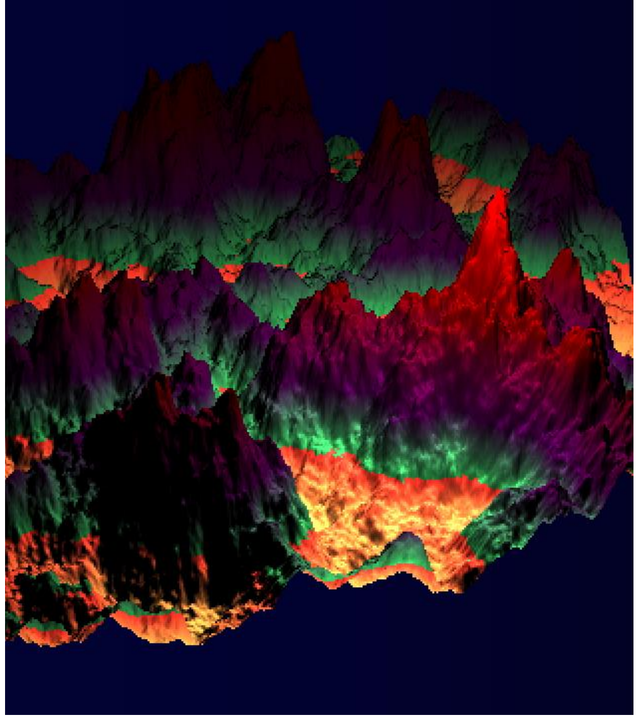
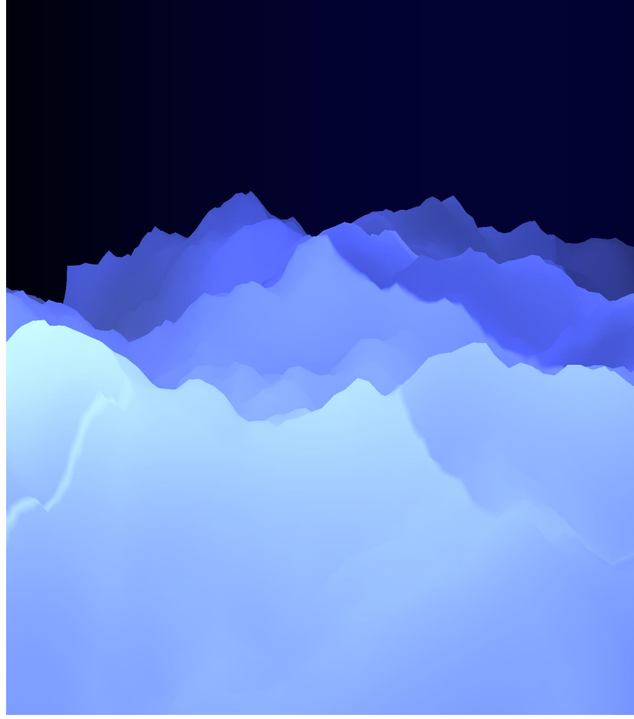
Use Perlin Noise to Generate Terrain in GLSL shaders

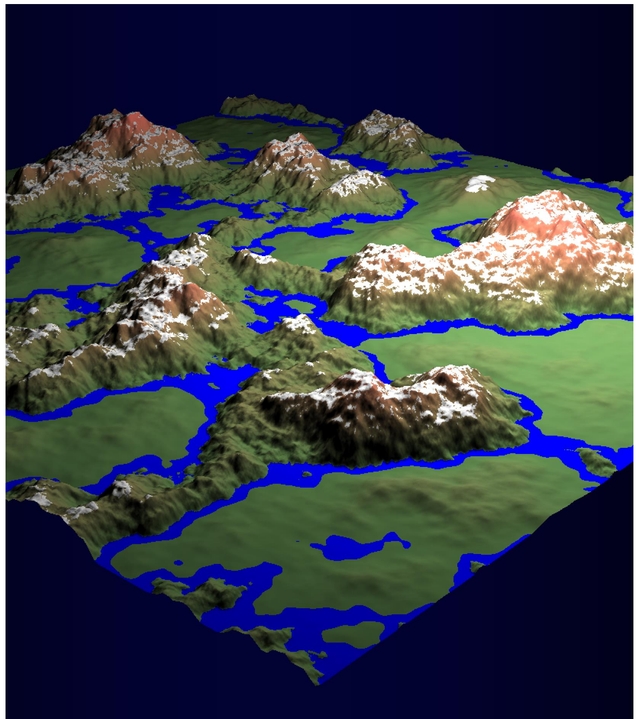
- We can use Perlin noise to in both vertex shader and fragment shader:
- In vertex shader, we use it to calculate vertex height
- In fragment shader, we use it to calculate fragment color

Calculate Terrain's Normals and Colors



- We start from a flat triangle mesh with zero height for each vertex
- [Vertex shader – Height]: For each vertex of the mesh, we calculate its **height** using a shader function, and update vertex position
- [Fragment shader – Normal]: We calculate the **normal** of a point by looking at the height of its neighboring points and calculate its normal (red) as the **cross product** of the two axis vectors on the mesh surface (blue and green)
- [Fragment shader – Emission]: We calculate an **emissive color** of each fragment based on its height (we use this color for the color of snow, trees, water, and soil!)
- The final color of a fragment is calculated as the multiplication of the lighting color and the emissive color



Mountains and Rivers

Depict the grandeur in your heart with Perlin noise functions

Additional Reading Materials for Perlin Noise

Ken Perlin, Making Noise:
https://web.archive.org/web/20071011035810/http://noisemachine.com/talk1/

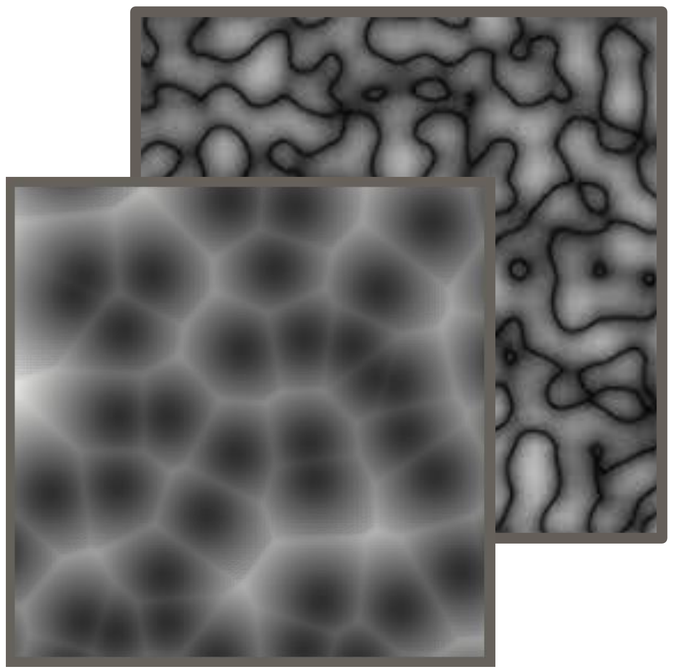
The book of shaders, noise: https://thebookofshaders.com/11/

Fragment shader code: https://thebookofshaders.com/edit.php#11/2d-gnoise.frag

The Perlin noise math FAQ:

https://mzucker.github.io/html/perlin-noise-math-faq.html

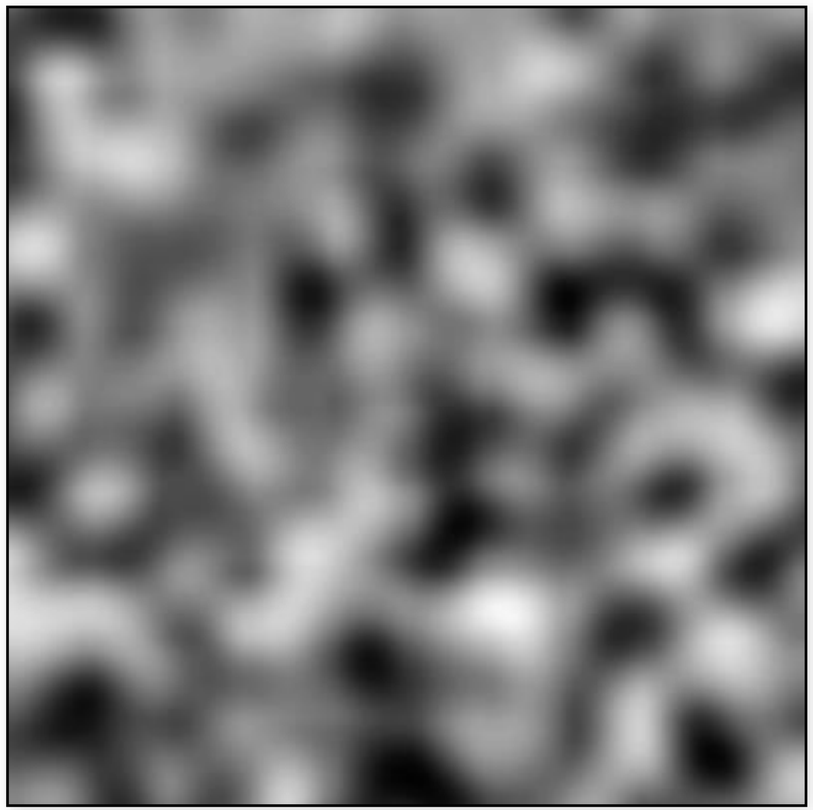
Other Noise Functions



Perlin Noise with Absolute Value

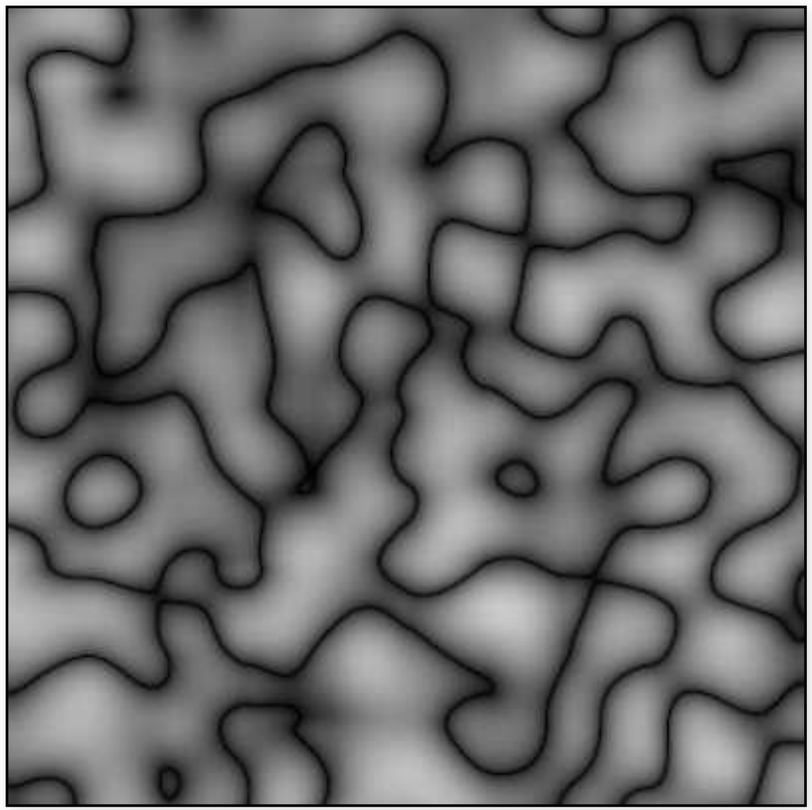
- Typically signed by default, ~in [-1,1] with a mean of 0

offset/scale to put into [0,1] range



`(noise(p)+1)/2`

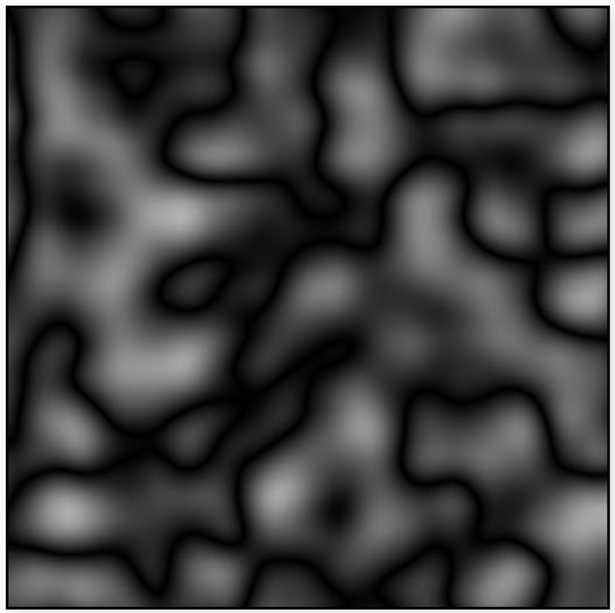
take absolute value



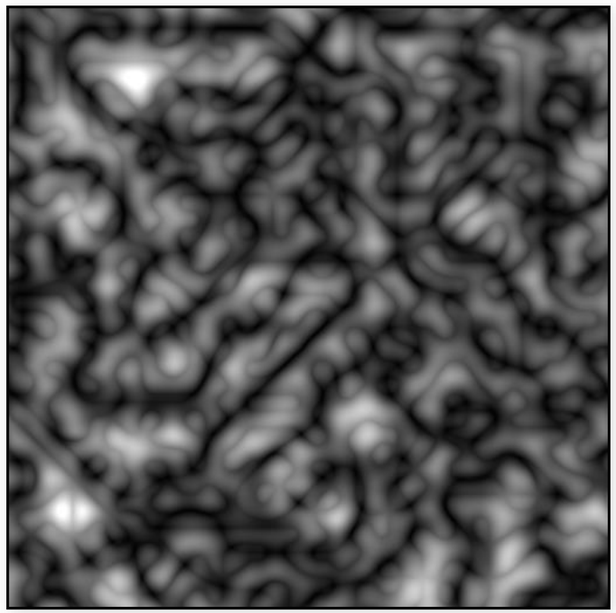
`|noise(p)|`

Turbulence

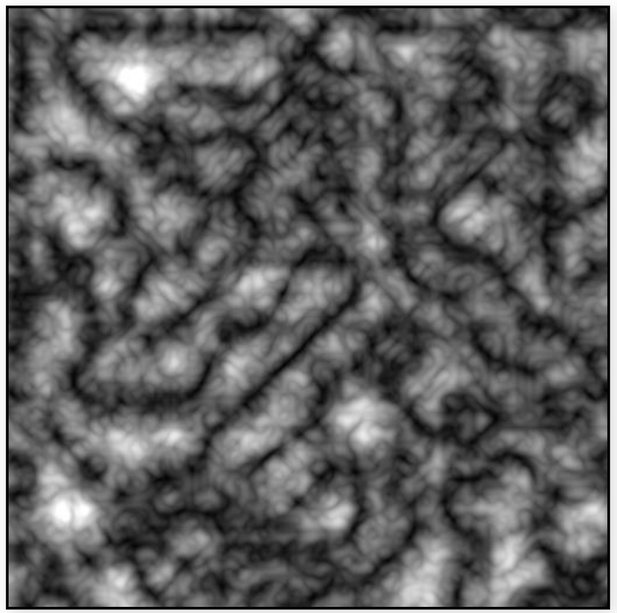
- Same as fBm, but sum **absolute value** of noise function



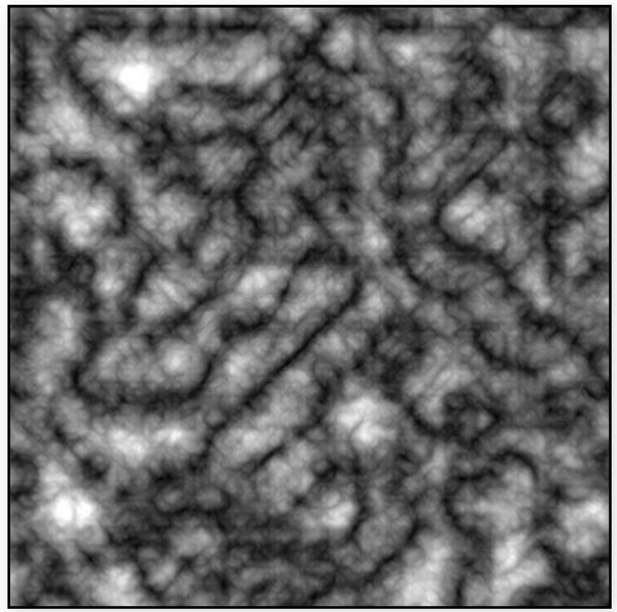
1 octaves



2 octaves



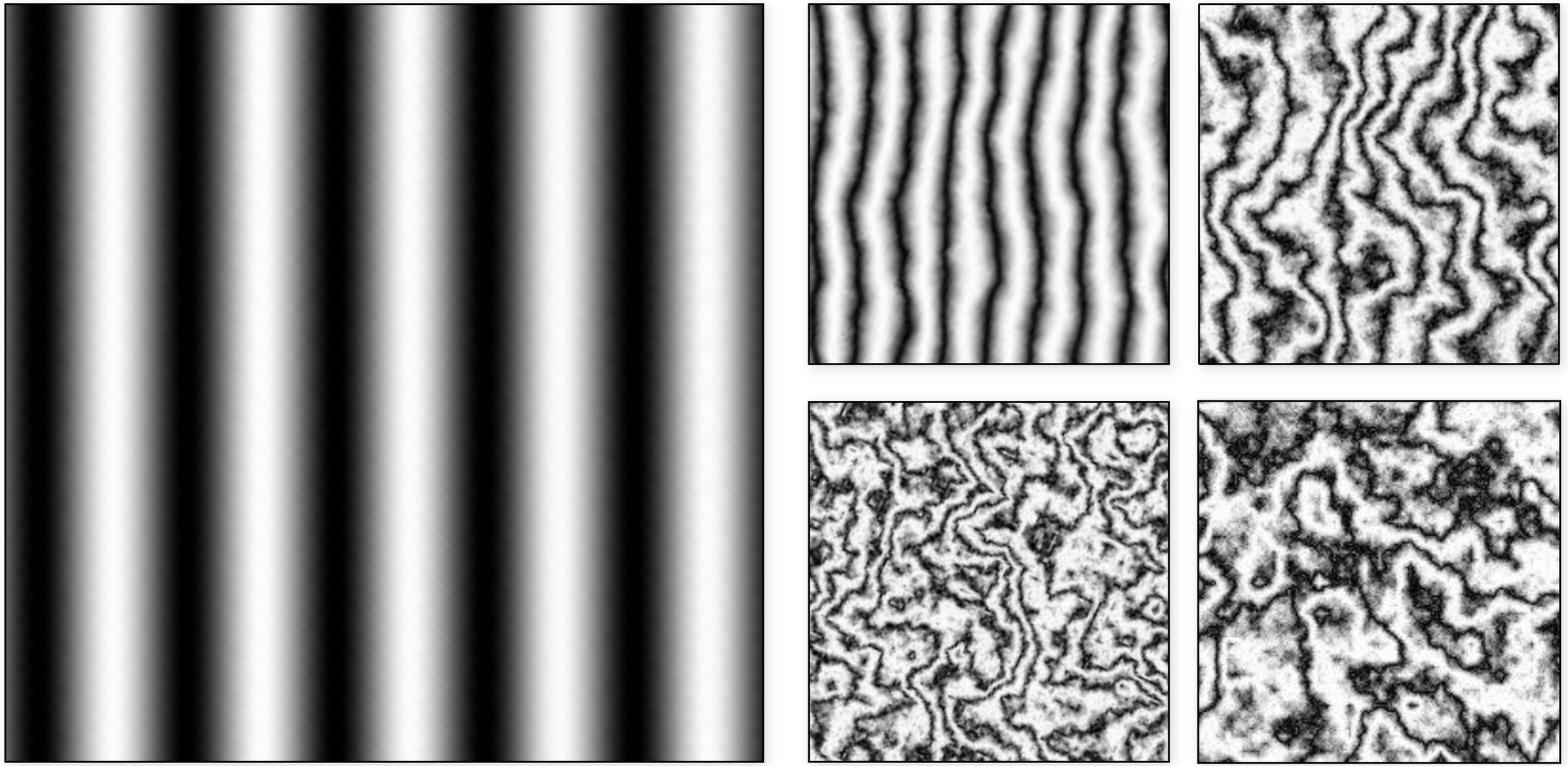
3 octaves



4 octaves

Marble

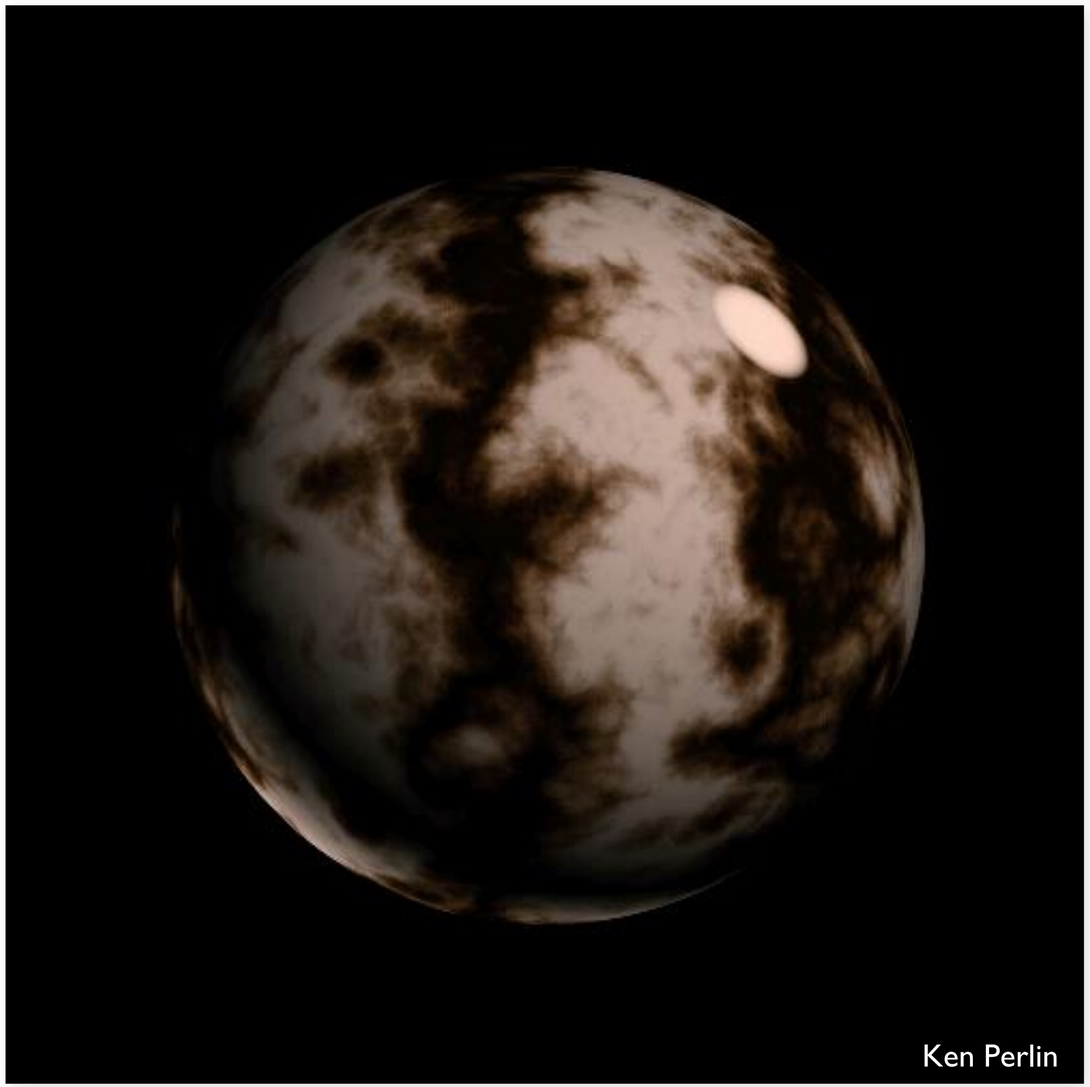
$$(1 + \sin(k_1 \mathbf{p}_x + \text{turbulence}(k_2 \mathbf{p}))/w)/2$$



from http://lodev.org/cgtutor/randomnoise.html

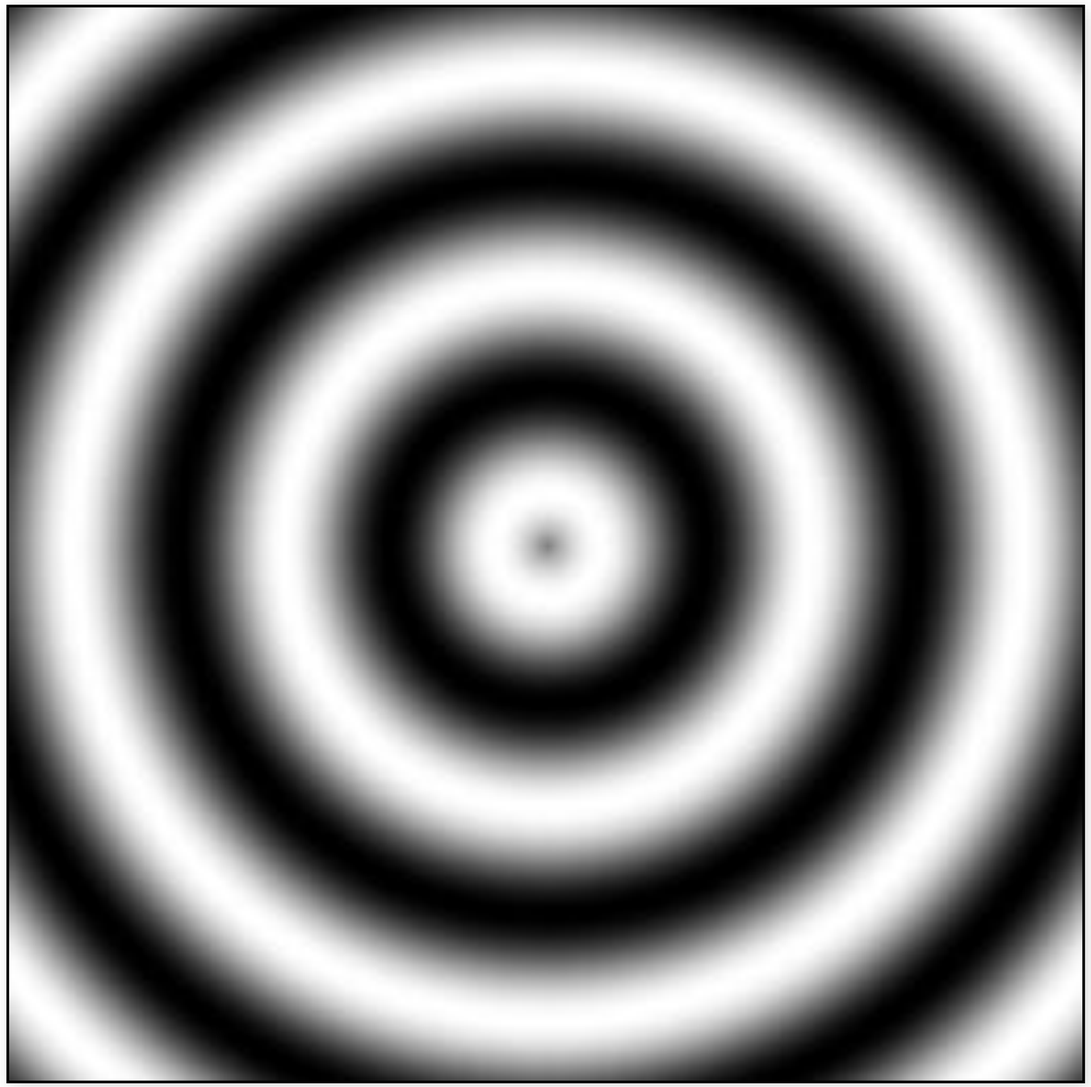
Marble

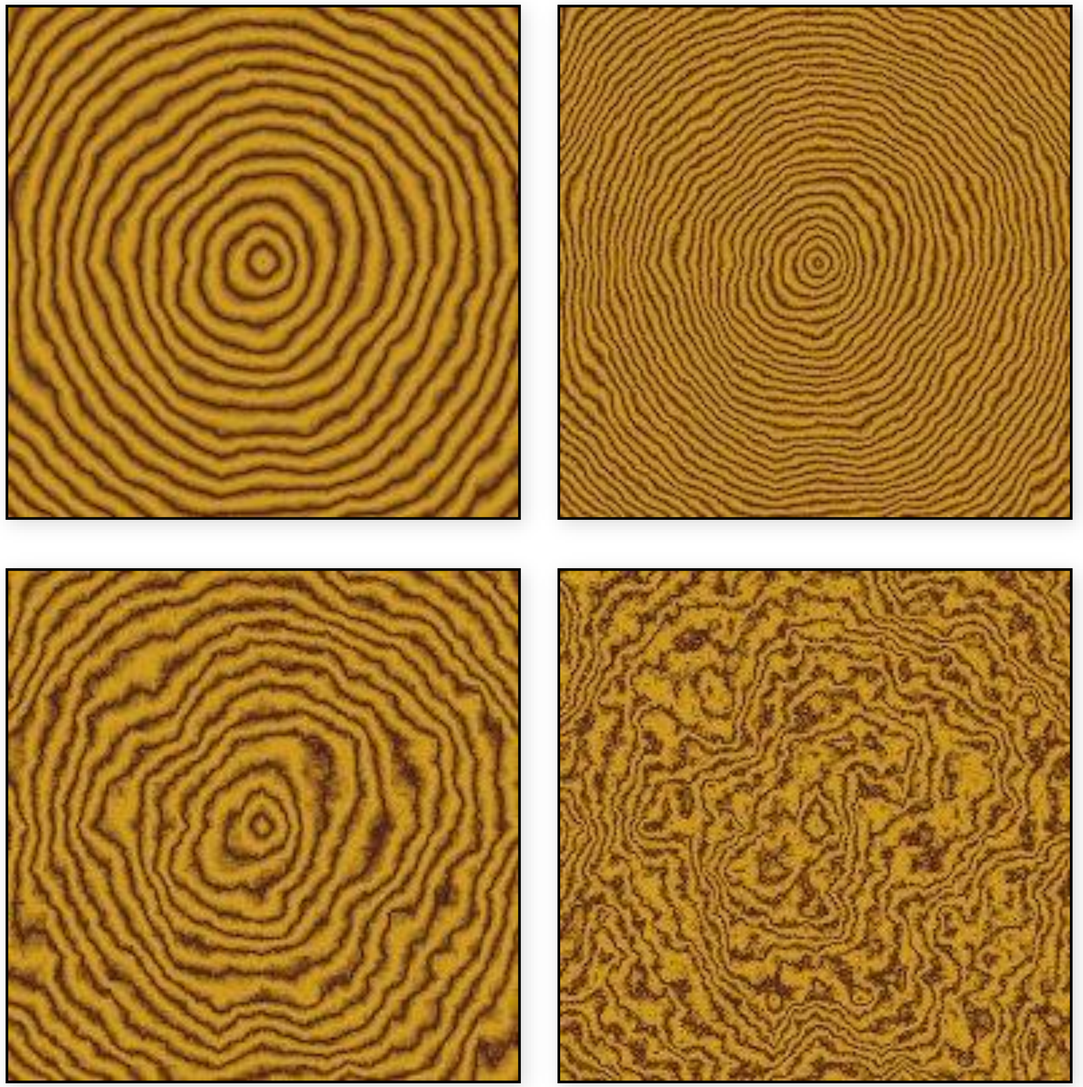
$$(1 + \sin(k_1\,\mathbf{p}_x + \text{turbulence}(k_2\,\mathbf{p}))/w)/2$$



Wood

$$(1 + \sin(\mathrm{sqrt}(\mathbf{p}_x^2 + \mathbf{p}_y^2) + \mathrm{fBm}(\mathbf{p})))/2$$

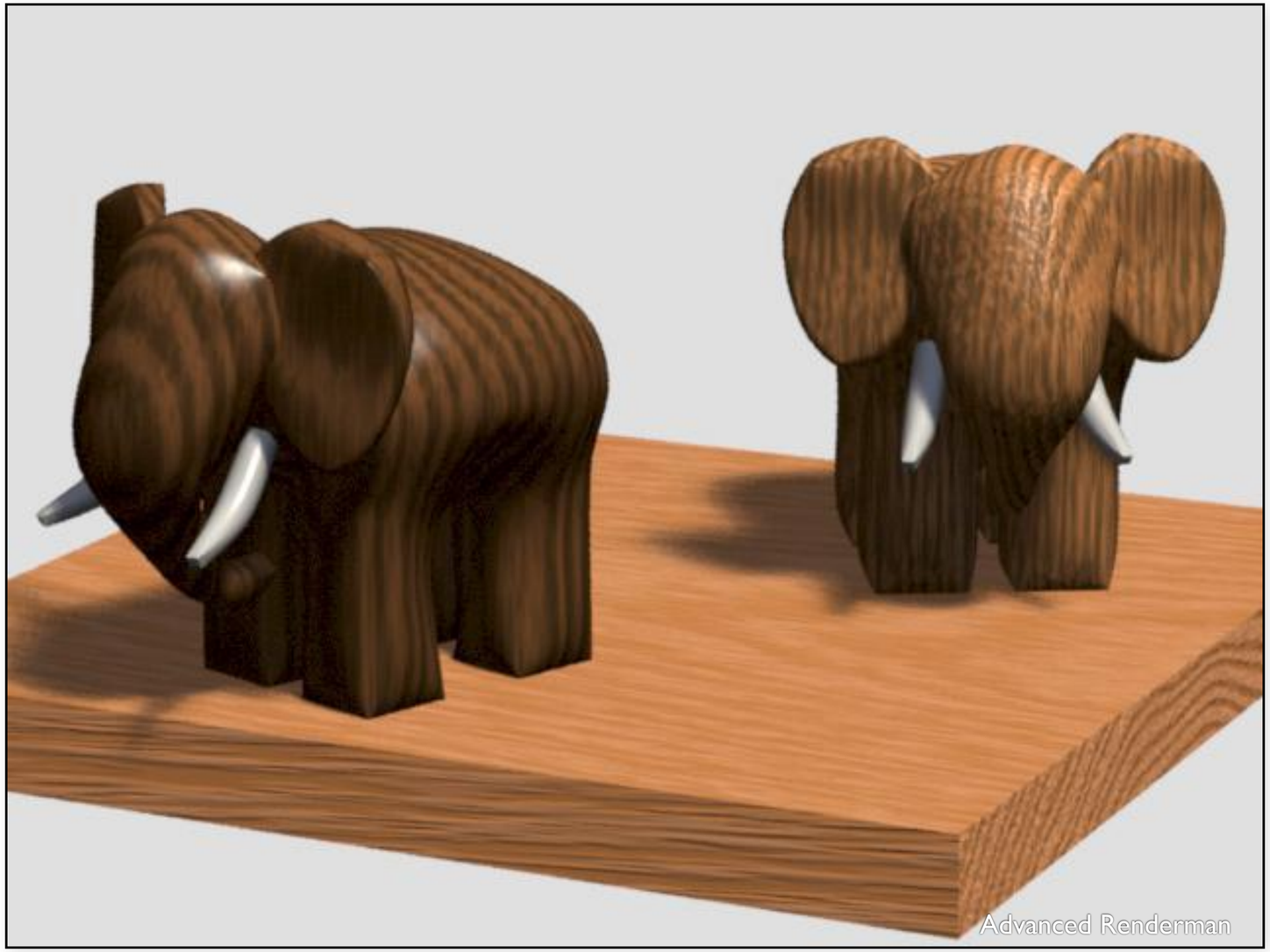




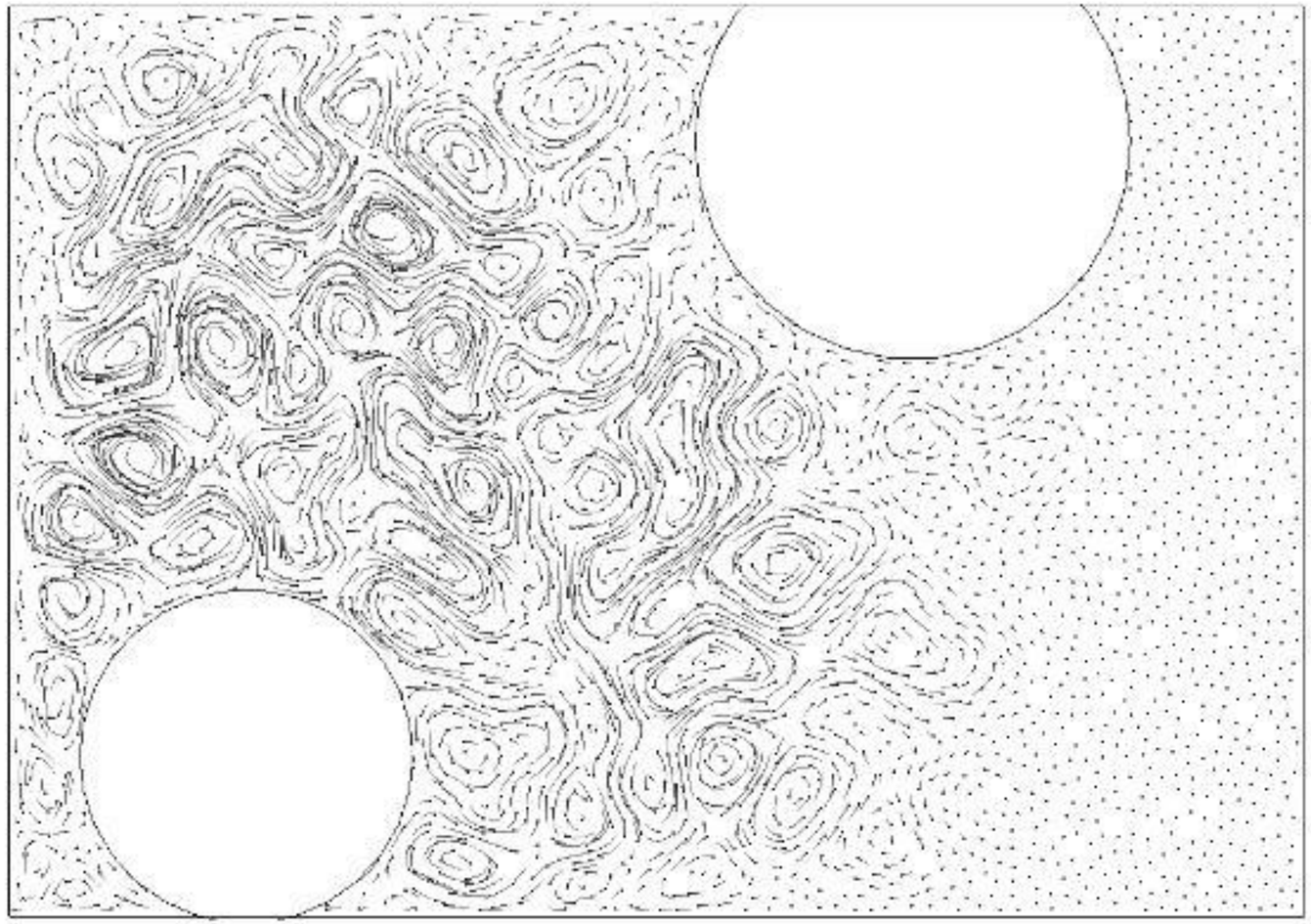
from http://lodev.org/cgtutor/randomnoise.html

Wood

$$(1 + \sin(\mathrm{sqrt}(\mathbf{p}_x^2 + \mathbf{p}_y^2) + \mathrm{fBm}(\mathbf{p})))/2$$



Curl Noise for Animation



- Curl noise for procedural fluid flow, R. Bridson, J. Hourihan, and M. Nordenstam, Proc. ACM SIGGRAPH 2007.

Worley Noise

- "Cellular texture" function

  Introduced in 1996 by Steve Worley

- Randomly distribute "feature points" in space

  $f_n(x)$ = distance to n^{th} closest point to x