TA 9

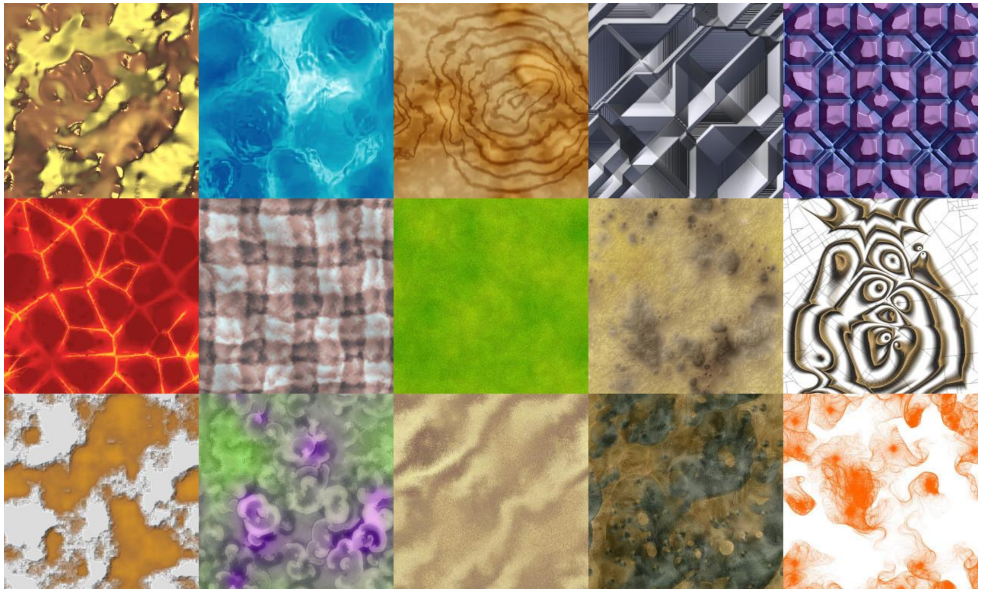
- 2D Procedural Texturing
- 3D Procedural Texturing
- Random Noise Algorithms
- EX4

Procedural Texturing

Computer Graphics 2020

Procedural Texturing

- A ***procedural texture*** is a texture created using a mathematical description (i.e. an algorithm or function) rather than directly storing image data

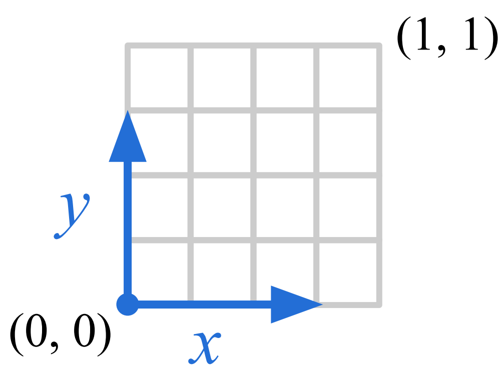


Procedural Texturing Advantages

- Easy sampling - no need for interpolation or mipmapping because every coordinate (u, v) can be directly evaluated
- Unlimited texture resolution
- Low storage cost - compute values at runtime rather than loading from memory
- Periodic functions can create seamlessly tiling textures

2D Procedural Textures

- We can think of a procedural texture as a function $f : [0,1]^2 \rightarrow [0,1]^4$ that maps 2D texture coordinates (x, y) / (u, v) to a 4D color (r, g, b, a)
- Later we will see procedural textures of higher dimensions as well



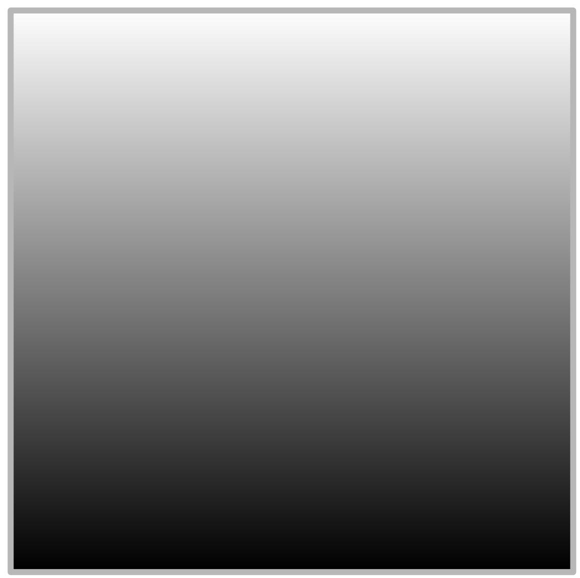
2D Procedural Textures

$$f(x, y) = x$$

*shorthand for $(x, x, x, 1)$

2D Procedural Textures

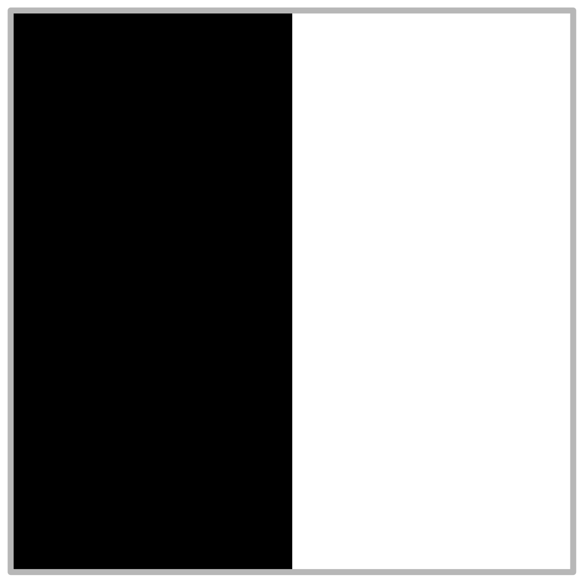
$$f(x, y) = y$$



2D Procedural Textures

$$f(x, y) = \text{step}(0.5, x)$$

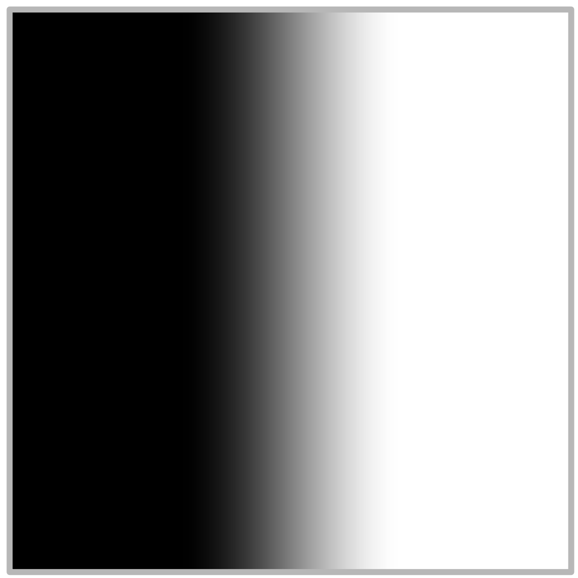
$\text{step}(a, x)$ returns 0 if $x < a$ and 1 if $a \leq x$



2D Procedural Textures

$$f(x, y) = \mathrm{smoothstep}(0.3, 0.7, x)$$

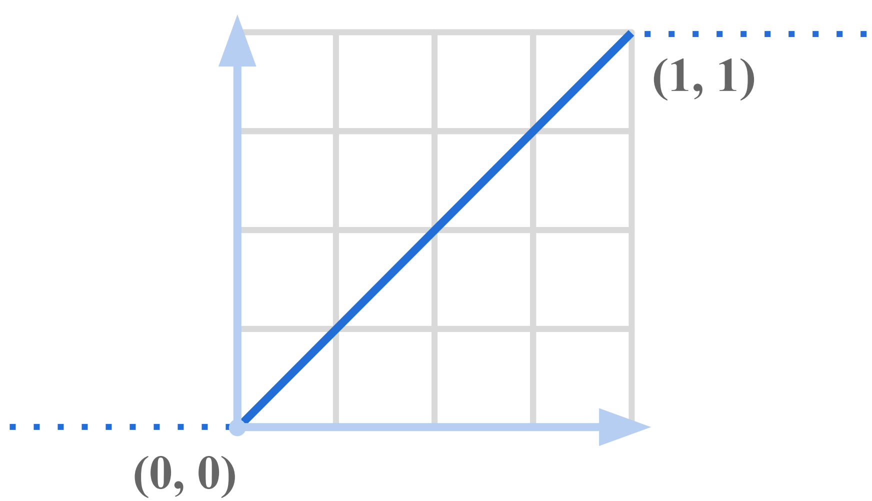
$\mathrm{smoothstep}(a, b, x)$ interpolates if $a < x < b$



Linear Interpolation

- Linear Interpolation between 0 and 1 gives a sharp transition at 0 and 1:

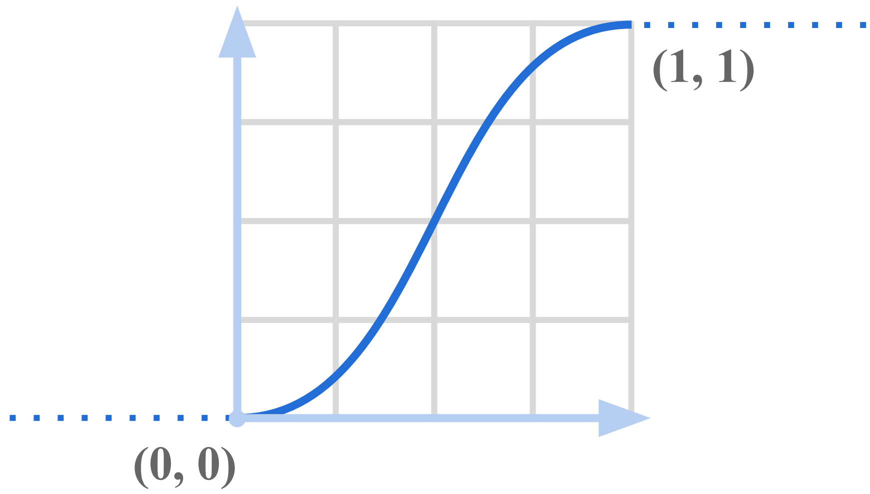
$$f(x) = x$$



Cubic Interpolation

- Using ***Cubic Interpolation*** we get a *smooth* transition at 0 and 1:

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



Smoothstep

$$\text{smoothstep}(a, b, x) = \begin{cases} 0 & x \le 0 \\ 3x^2 - 2x^3 & 0 \le x \le 1 \\ 1 & 1 \le x \end{cases}$$

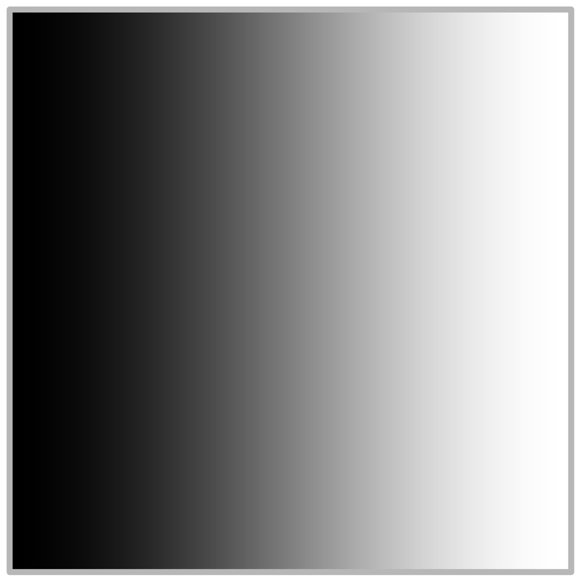
- When $0 \le x \le 1$ we get a smooth transition from 0 to 1 using a *Cubic Hermite Spline*
- This is called *Cubic Interpolation* or *Hermite Interpolation*

2D Procedural Textures

- Linear Interpolation on the x-axis

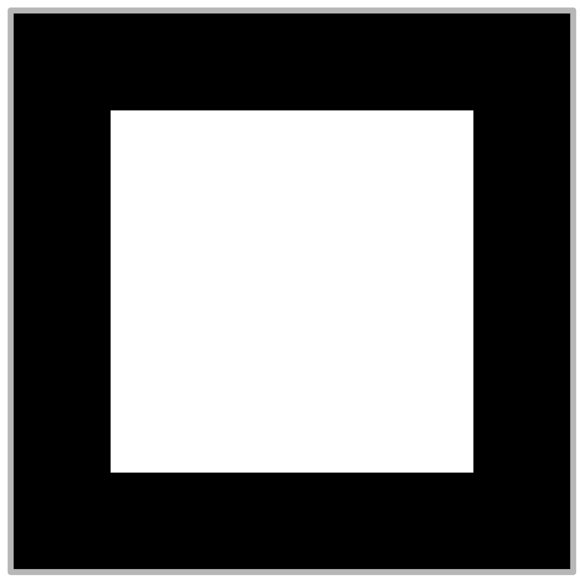
2D Procedural Textures

- Cubic Interpolation on the x-axis



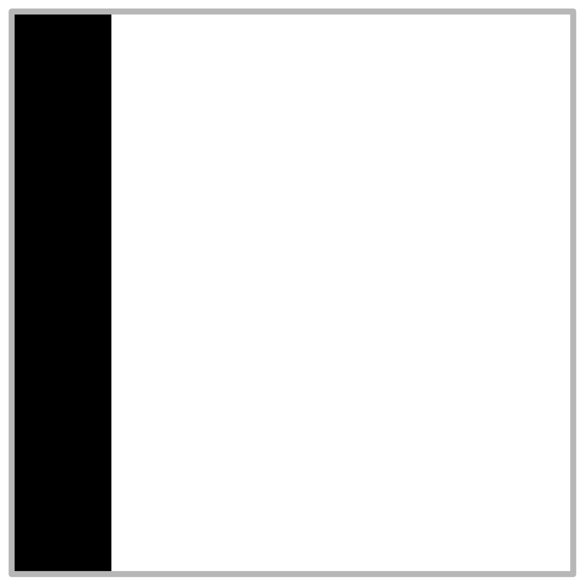
2D Procedural Textures

- How can we draw a square with a side of 0.6 in the middle of the texture?



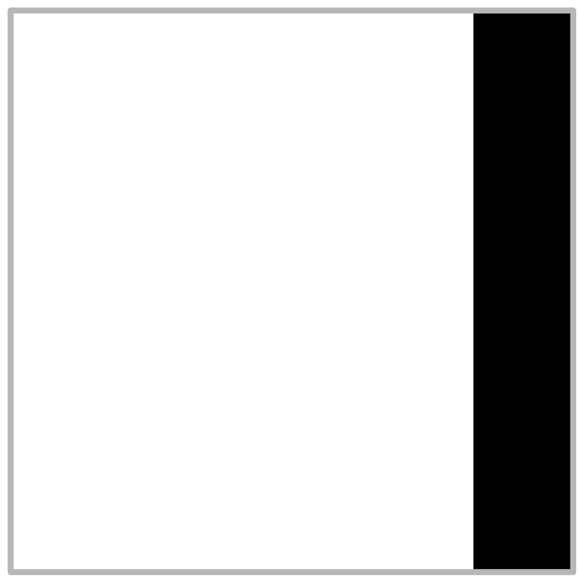
2D Procedural Textures

$$f(x, y) = \mathrm{step}(0.2, x)$$



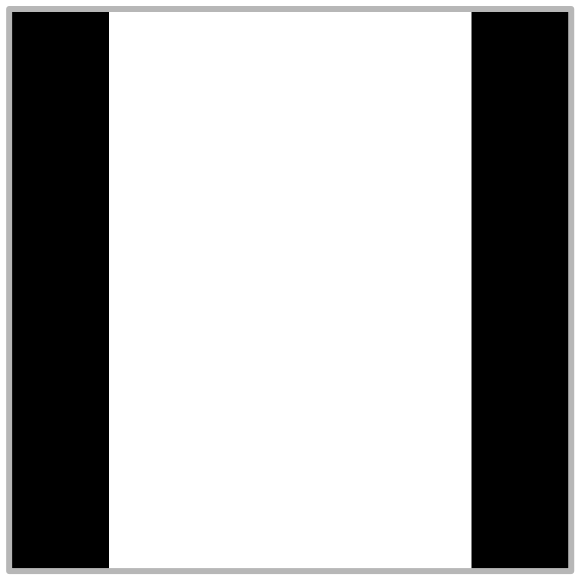
2D Procedural Textures

$$f(x, y) = \text{step}(0.2, 1 - x)$$



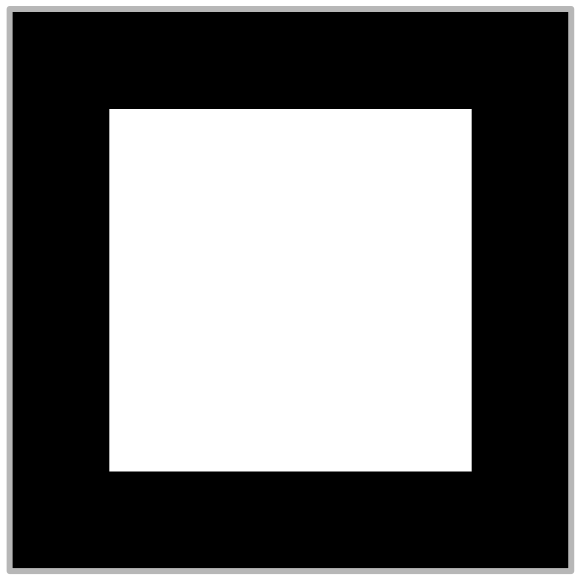
2D Procedural Textures

$$f(x, y) = \mathrm{step}(0.2, x) \cdot \mathrm{step}(0.2, 1 - x)$$



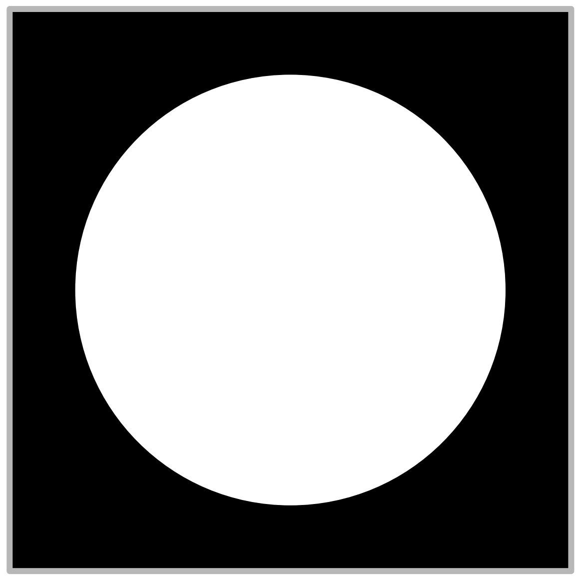
2D Procedural Textures

$$f(x, y) = \mathrm{step}(0.2, x) \cdot \mathrm{step}(0.2, 1 - x)$$
$$\cdot\ \mathrm{step}(0.2, y) \cdot \mathrm{step}(0.2, 1 - y)$$



2D Procedural Textures

- How can we draw a circle of radius 0.4 in the middle of the texture?



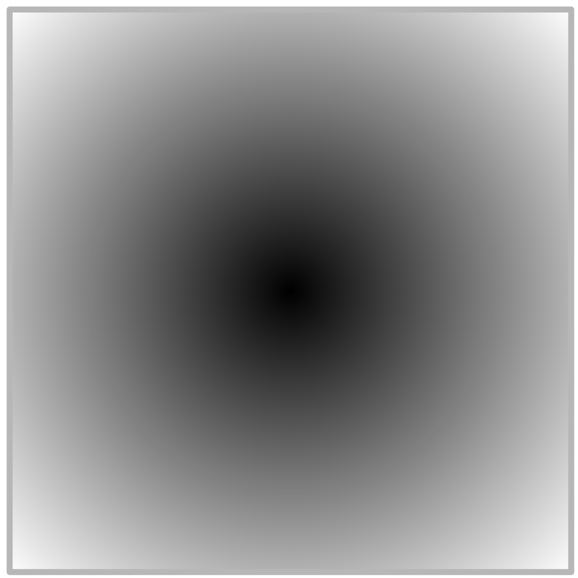
2D Procedural Textures

$$f(x, y) = \sqrt{x^2 + y^2}$$

- This is called a ***Distance Field*** or ***Distance Map***

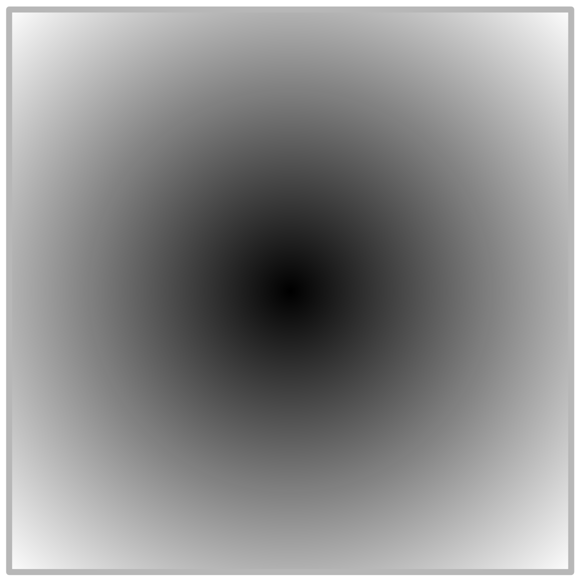
2D Procedural Textures

$$f(x, y) = \sqrt{(x-0.5)^2 + (y-0.5)^2}$$



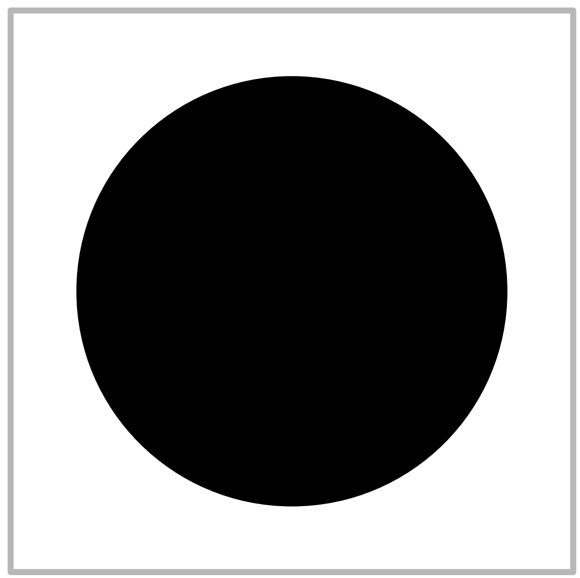
2D Procedural Textures

- How do we draw a circle of radius 0.4 using this distance map?



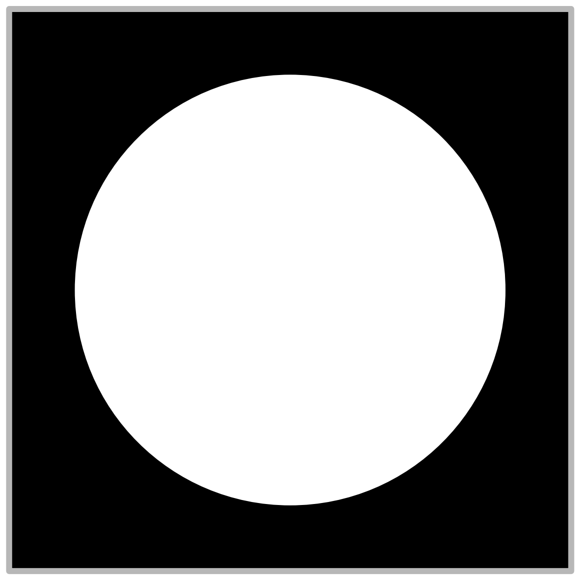
2D Procedural Textures

$$f(x, y) = \text{step}(0.4, \sqrt{(x-0.5)^2 + (y-0.5)^2})$$



2D Procedural Textures

$$f(x, y) = 1 - \mathrm{step}\left(0.4, \sqrt{(x-0.5)^2 + (y-0.5)^2}\right)$$

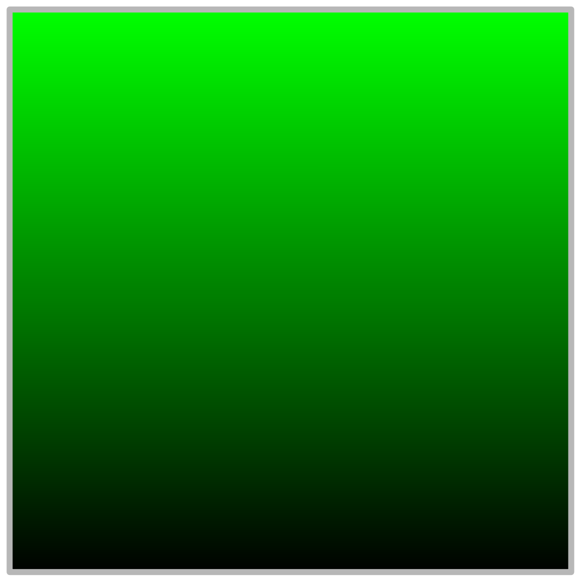


2D Procedural Textures

- Remember we can output different values to each color channel: $f(x, y) = (x, 0, 0, 1)$

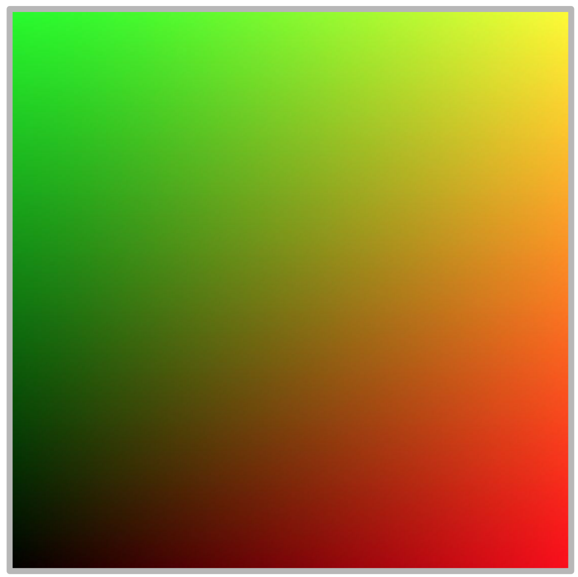
2D Procedural Textures

$$f(x, y) = (0, y, 0, 1)$$



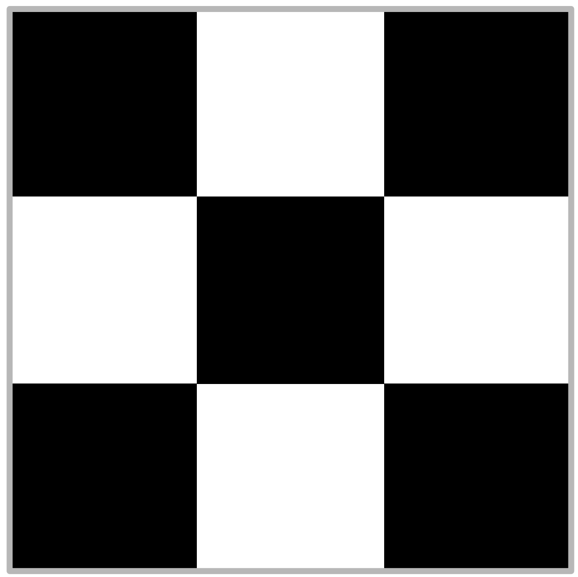
2D Procedural Textures

$$f(x, y) = (x, y, 0, 1)$$



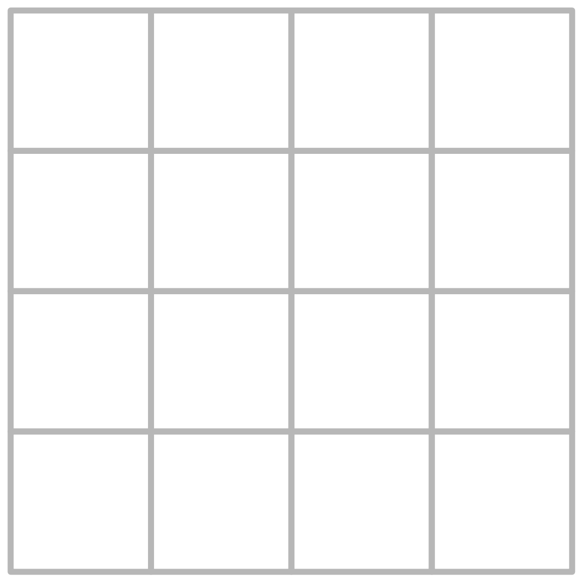
2D Procedural Textures

$$f(x, y) = \lfloor 3x \rfloor + \lfloor 3y \rfloor \bmod 2$$



2D Procedural Textures

- We may want to divide our space into a $n \times n$ grid
- For example, we may want to repeat patterns



2D Procedural Textures

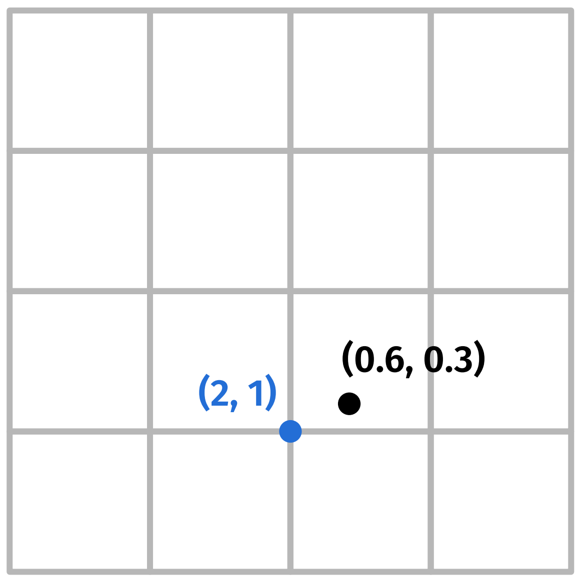
- For each coordinate (x, y) we can multiply by n and look at its integer part and fractional part:
  - $f_{\text{int}}(x, y, n) \coloneqq (\lfloor nx \rfloor, \lfloor ny \rfloor)$
  - $f_{\text{frac}}(x, y, n) \coloneqq (\text{frac}(nx), \text{frac}(ny))$

    *where $\text{frac}(a) = a - \lfloor a \rfloor$

2D Procedural Textures

$$f_{\text{int}}(x, y, n) = (\lfloor nx \rfloor, \lfloor ny \rfloor)$$

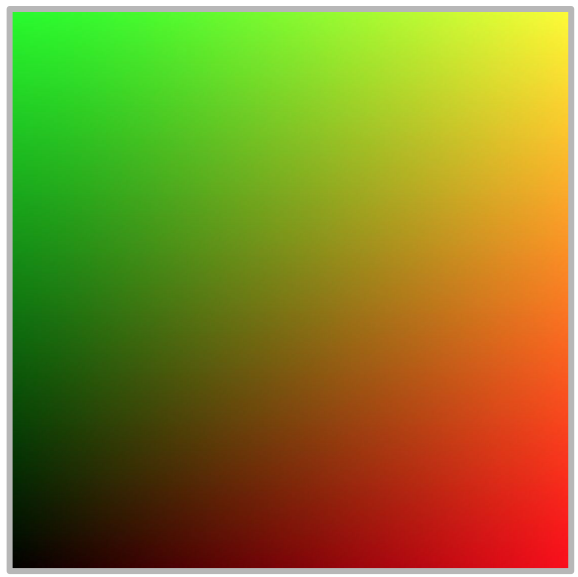
$$f_{\text{int}}(0.6, 0.3, 4) = (\lfloor 4 \cdot 0.6 \rfloor, \lfloor 4 \cdot 0.3 \rfloor) = (2, 1)$$



2D Procedural Textures

$$f(x, y) = (x, y, 0, 1)$$

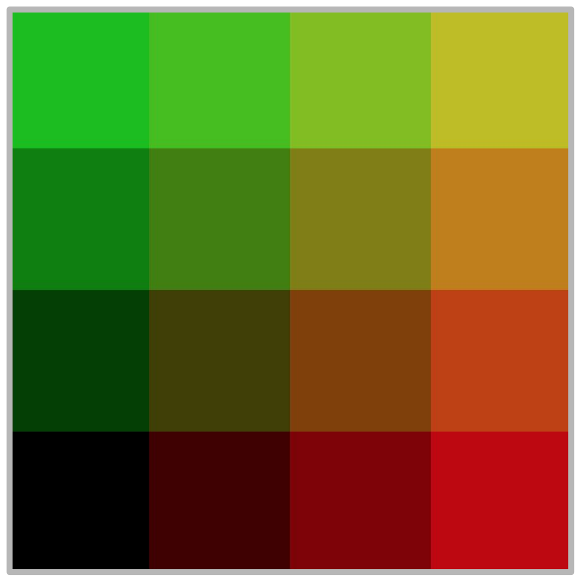
$$f(\tfrac{1}{4} f_{\text{int}}(x, y, 4)) = ?$$



2D Procedural Textures

$$f\left(\tfrac{1}{4} f_{\mathrm{int}}(x, y, 4)\right) = (\lfloor 4x \rfloor/4, \lfloor 4y \rfloor/4, 0, 1)$$

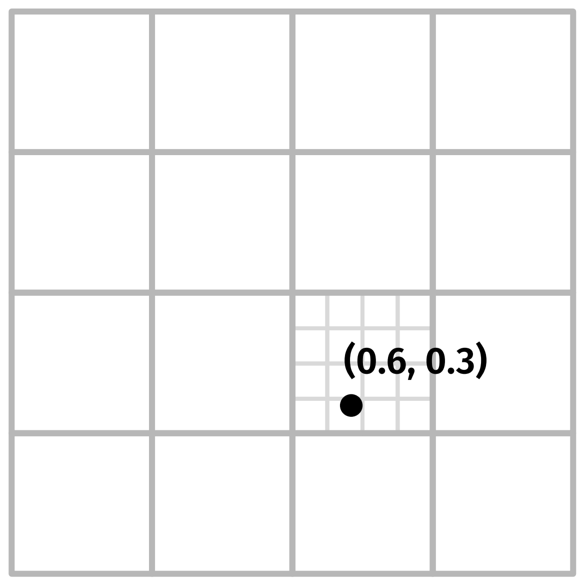
We get a discrete $n \times n$ “sampling” grid



2D Procedural Textures

$$f_{\text{frac}}(x, y, n) = (\text{frac}(nx),\ \text{frac}(ny))$$

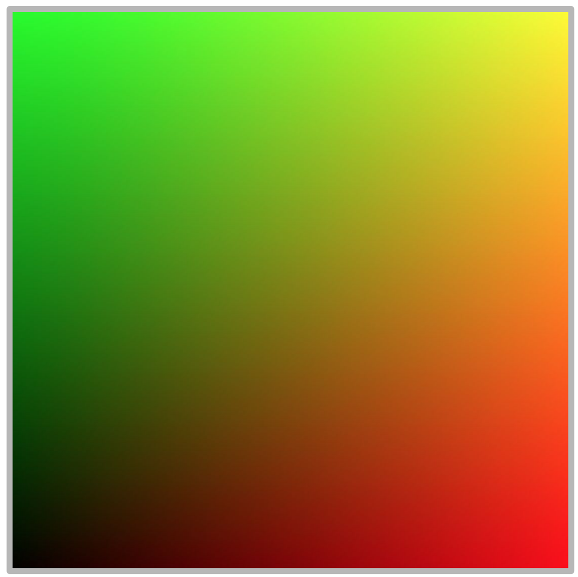
$$f_{\text{frac}}(0.6,\ 0.3,\ 4) = (\text{frac}(4 \cdot 0.6),\ \text{frac}(4 \cdot 0.3)) = (0.4,\ 0.2)$$



2D Procedural Textures

$$f(x, y) = (x, y, 0, 1)$$

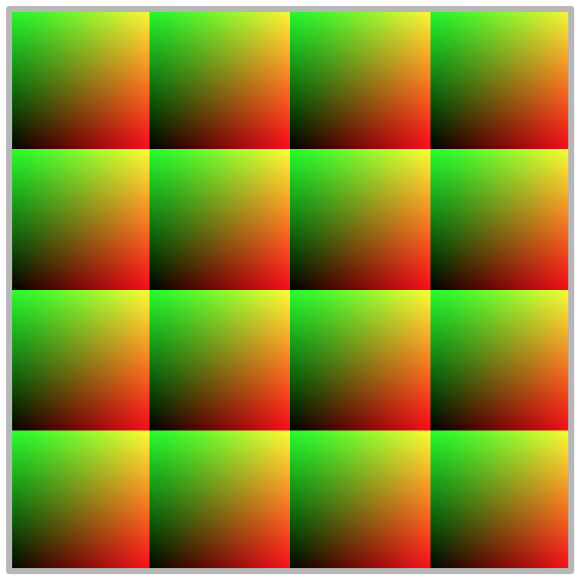
$$f(\, f_{\text{frac}}(x, y, 4)\,) = ?$$



2D Procedural Textures

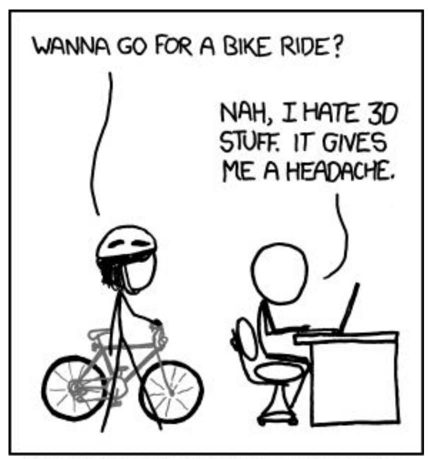
$$f(x, y) = (\mathrm{frac}(4x), \mathrm{frac}(4y), 0, 1)$$

We get $n \times n$ smaller copies of the full texture



3D Procedural Textures

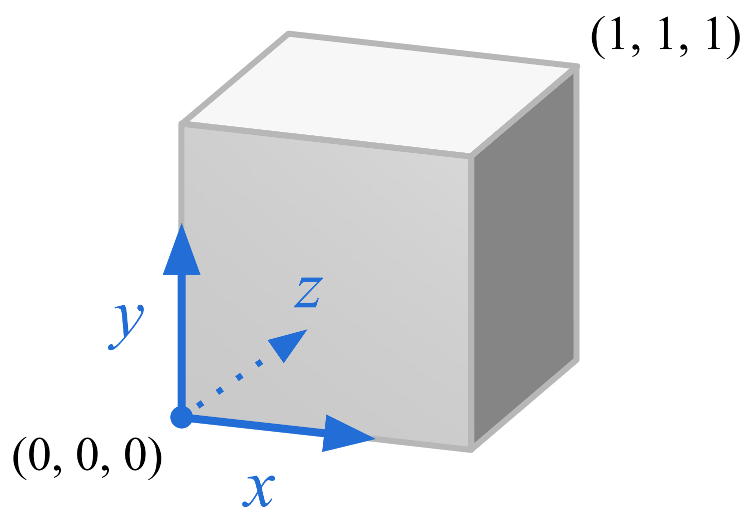
- So far we generated textures in 2D, but we can also generate 3D textures
- Using (x, y, z) object coordinates we can directly sample the 3D texture



WHEN YOU THINK ABOUT IT, THIS EXCUSE CAN GET YOU OUT OF ALMOST ANYTHING.

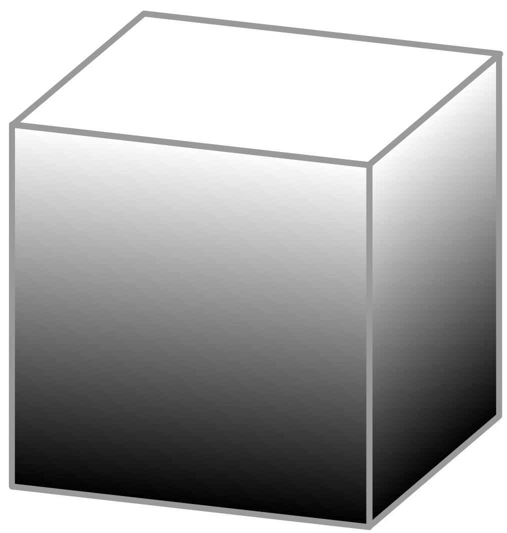
3D Procedural Textures

- We can think of a 3D procedural texture as a function $f: [0,1]^3 \to [0,1]^4$ that maps 3D coordinates (x, y, z) to a 4D color (r, g, b, a)



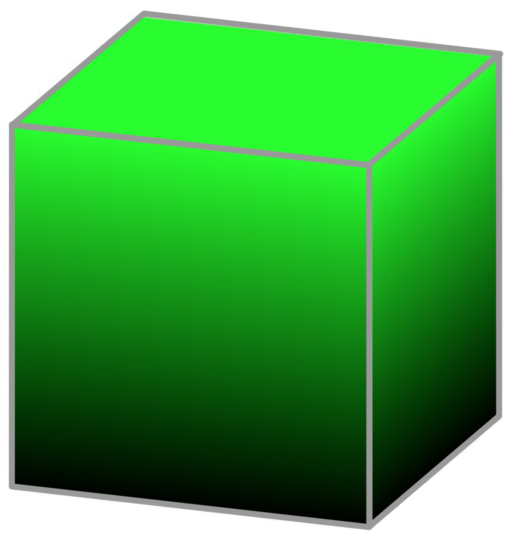
3D Procedural Textures

$$f(x, y, z) = y$$



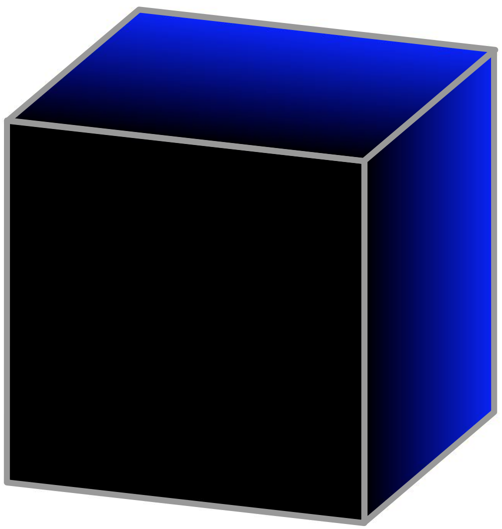
3D Procedural Textures

$$f(x, y, z) = (0, y, 0, 1)$$



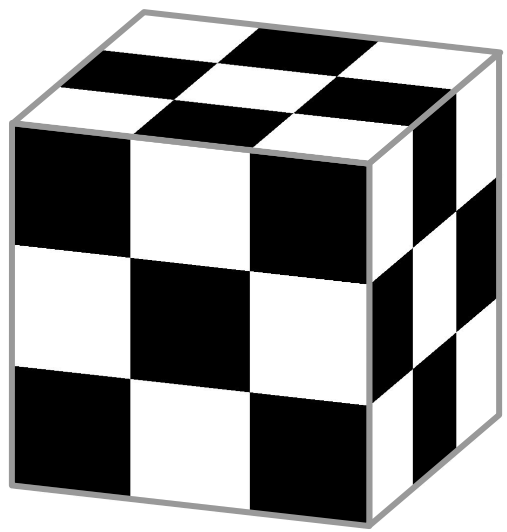
3D Procedural Textures

$$f(x, y, z) = (0, 0, z, 1)$$



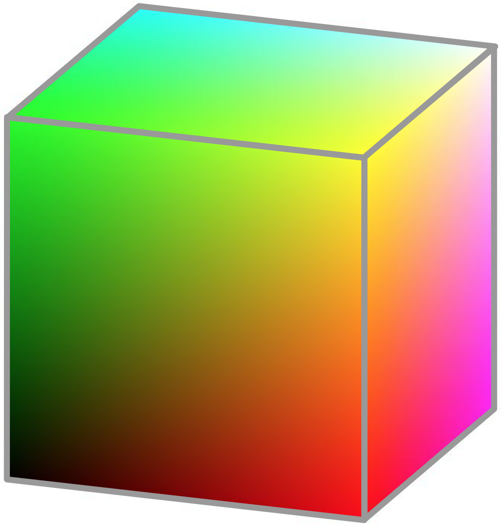
3D Procedural Textures

$$f(x, y, z) = \lfloor 3x \rfloor + \lfloor 3y \rfloor + \lfloor 3z \rfloor \bmod 2$$



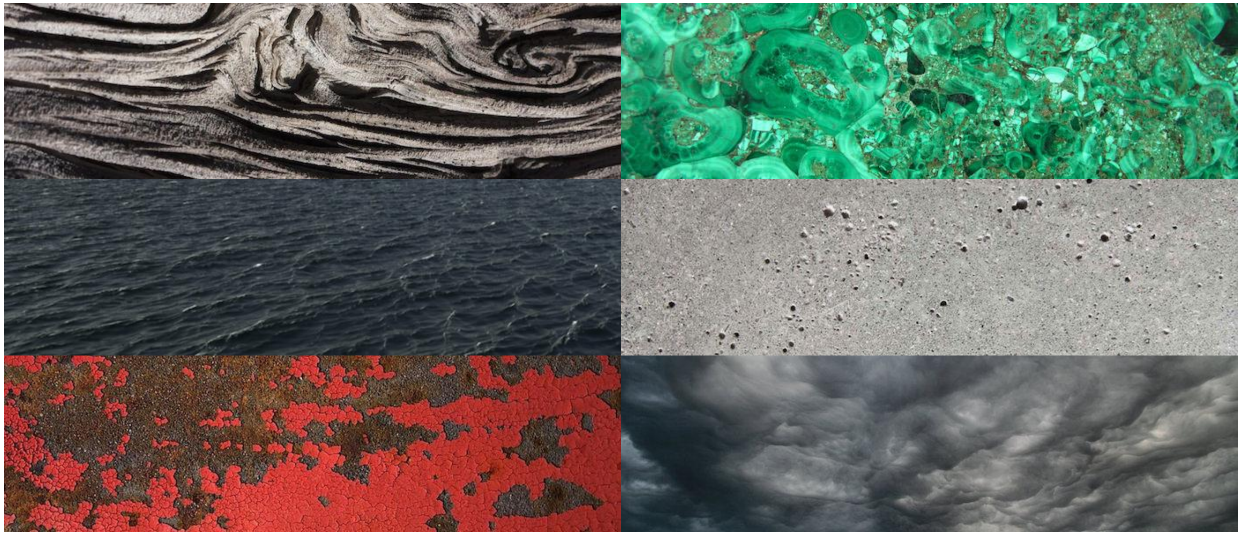
3D Procedural Textures

$$f(x, y, z) = (x, y, z, 1)$$



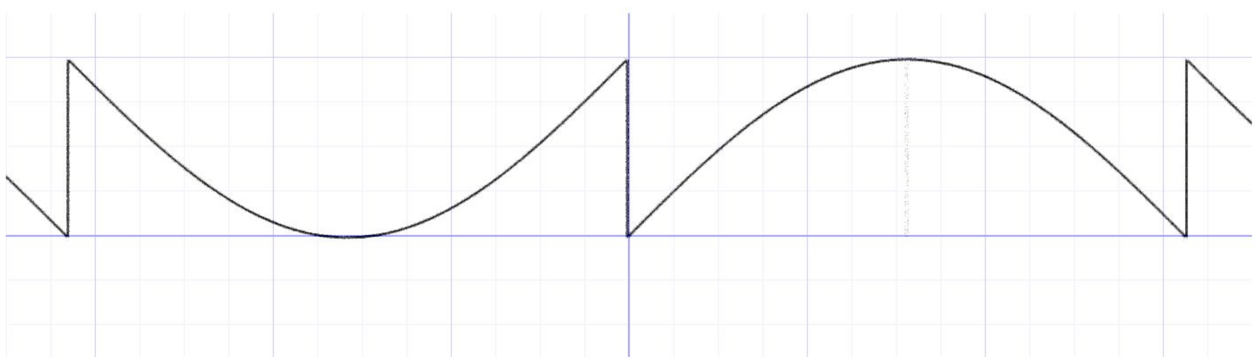
Noise

- The world is full of seemingly “random” materials
- How do we replicate this in a deterministic coding environment?

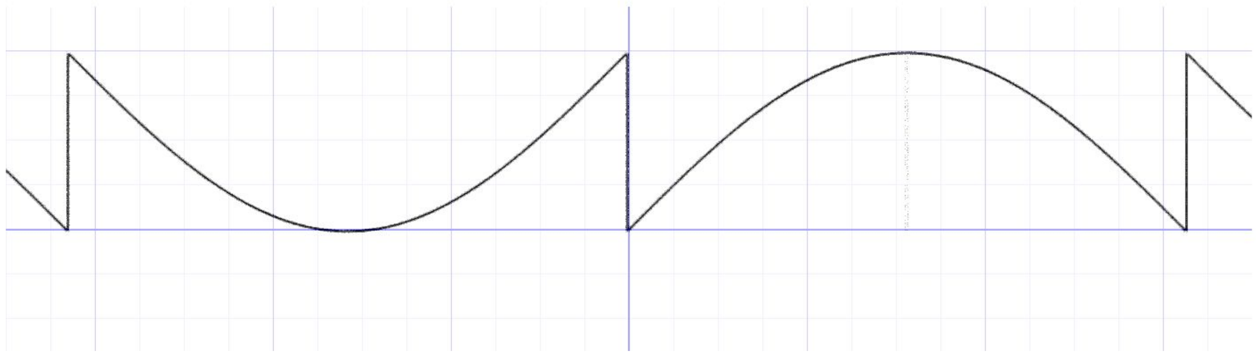


Noise

- Generating a truly random number is extremely difficult
- We can cheat - find a deterministic function that gives pseudo-random values
- For example, take a look at $\mathrm{frac}(\sin(x))$:



Noise



$$\mathrm{frac}(\alpha \sin(x))$$

- By increasing α, we “stretch” along the y axis to get more and more discontinuities
- If α is very large (i.e. 10^6), for each given x we get a pseudo-random number between 0 and 1

2D Noise

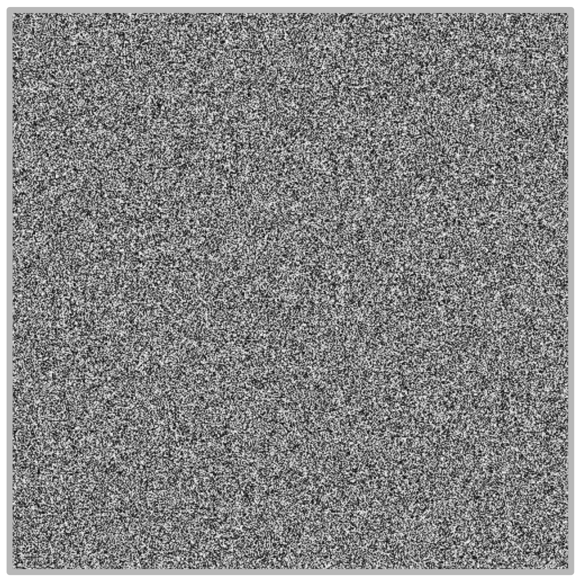
- For 2D, we need to somehow map 2D points to 1D numbers and use them to sample our pseudo-random function
- We can dot product a given (x, y) location with some positive vector $v = (v_1, v_2)$ to get a *sampling value* between 0 and 1:

$$[x, y] \cdot [v_1, v_2]$$

2D Noise

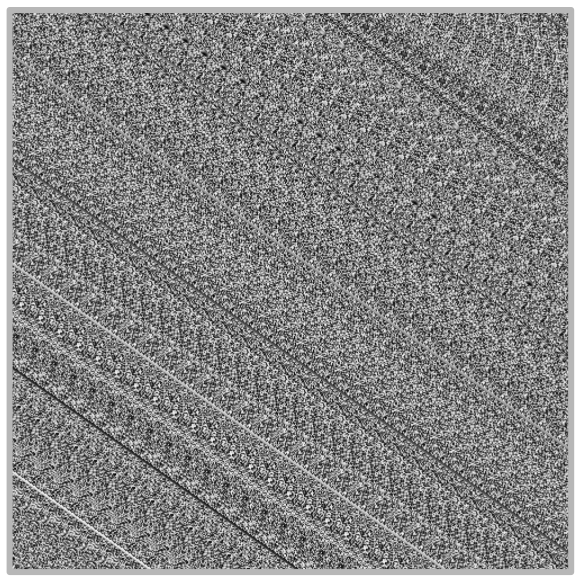
$$\mathrm{rand}(x, y) \coloneqq \mathrm{frac}(\boldsymbol{\alpha} \sin([x, y]\cdot[v_1, v_2]))$$

$$\boldsymbol{\alpha} = 43758.54 \qquad v = (12.98, 78.23)$$



2D Noise

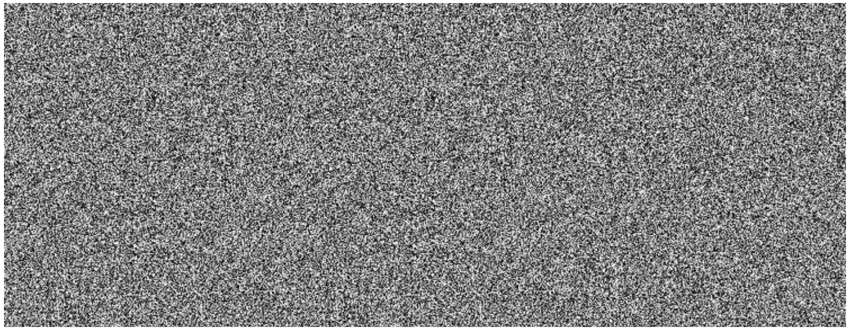
- Note that this is *pseudo*-random, for some values, say $\boldsymbol{\alpha} = 10^5 \;\; v = (12, 17)$ we get patterns:



2D Noise

- We generated a random texture, but it doesn't look so natural…

Nature



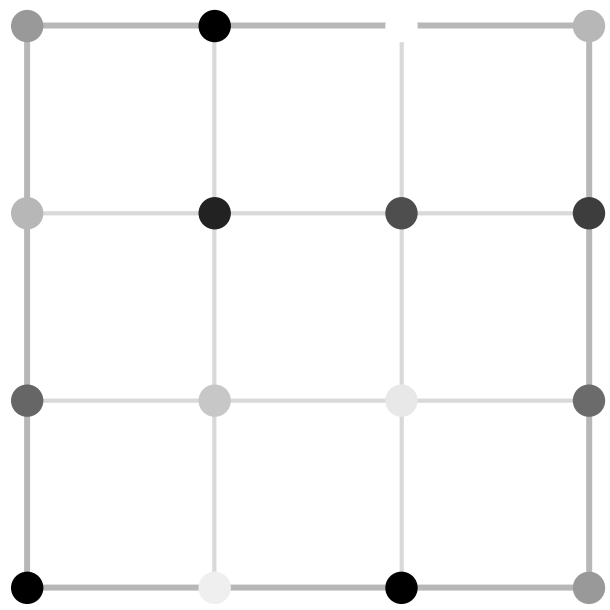
Random Noise

2D Noise

- Rather than having a unique value for each point, we want to have some correlation between nearby points in the texture
- We want to have smoother transitions between neighboring areas

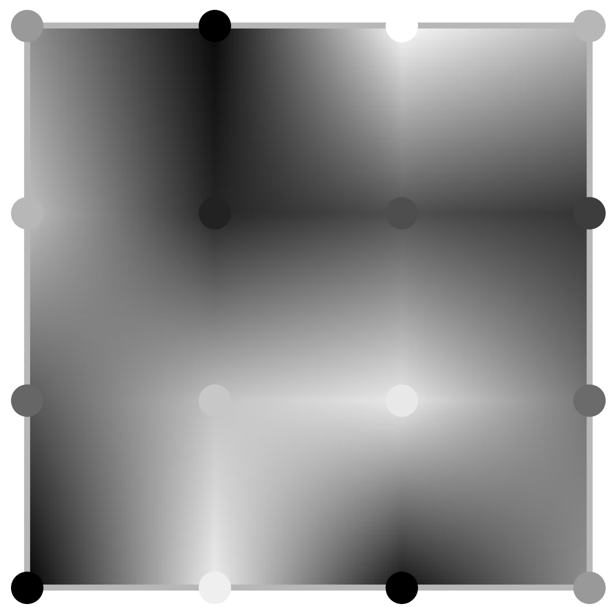
2D Value Noise

- We create a grid, then sample our pseudo-random function at each grid point



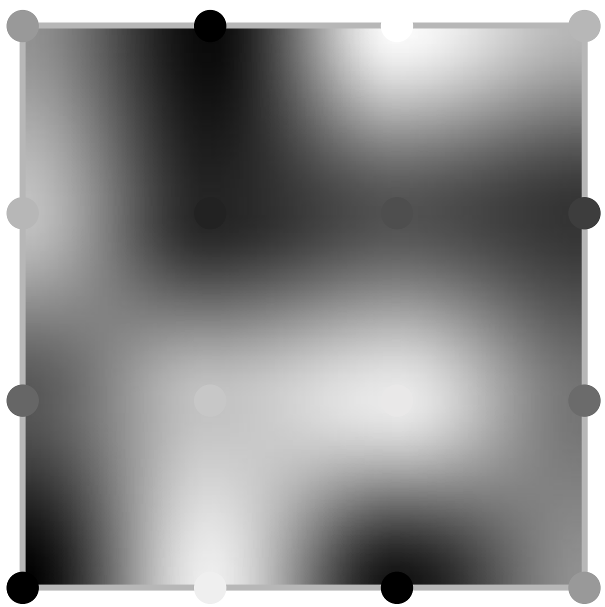
2D Value Noise

- We can then use bilinear interpolation to fill in the values between the grid points



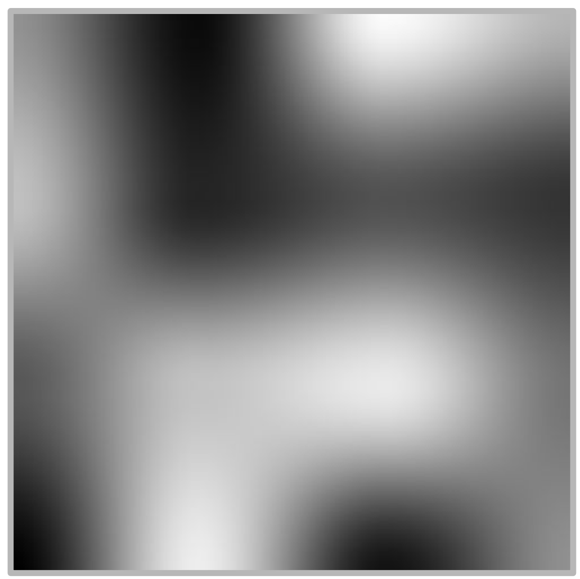
2D Value Noise

- Rather than bilinear interpolation, with *bicubic interpolation* we get a smoother result



2D Value Noise

- This is called ***Value Noise***, because we interpolate between random *values*

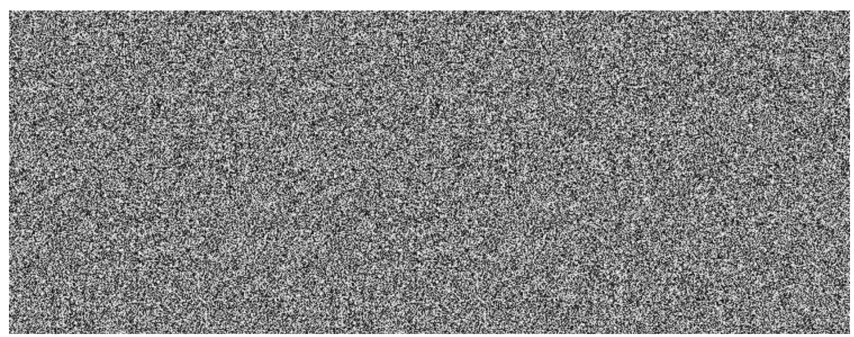


2D Value Noise

- Better than what we had, but value noise tends to look "blocky"

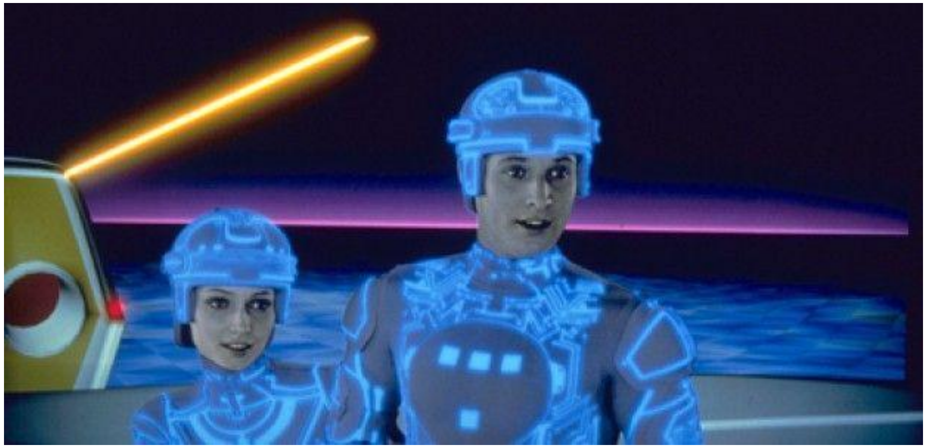
Value Noise

Random Noise



Perlin Noise

- In the early 1980s, Ken Perlin was commissioned to generate realistic textures for the movie "Tron"
- To diminish the blocky effect of value noise, Perlin developed the *Gradient Noise* algorithm



Perlin Noise

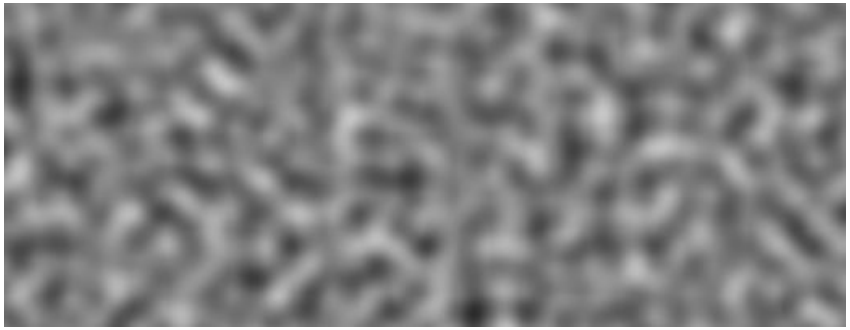
- In ***Gradient Noise*** we interpolate using gradients, rather than directly interpolating values
- The original algorithm Perlin used in his 1985 paper is now known as ***Perlin Noise***
- Perlin noise is a *type* of gradient noise

Perlin Noise

- We get a smoother, more organic-looking result when compared to value noise:

Value Noise

Perlin Noise



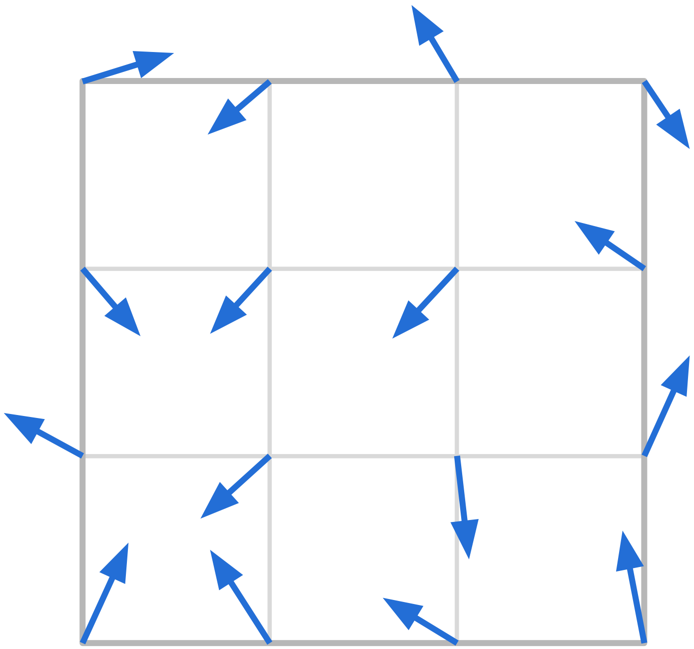
Perlin Noise

- Perlin was awarded an Oscar for the algorithm!

  The development of Perlin Noise has allowed computer graphics artists to better represent the complexity of natural phenomena in visual effects for the motion picture industry

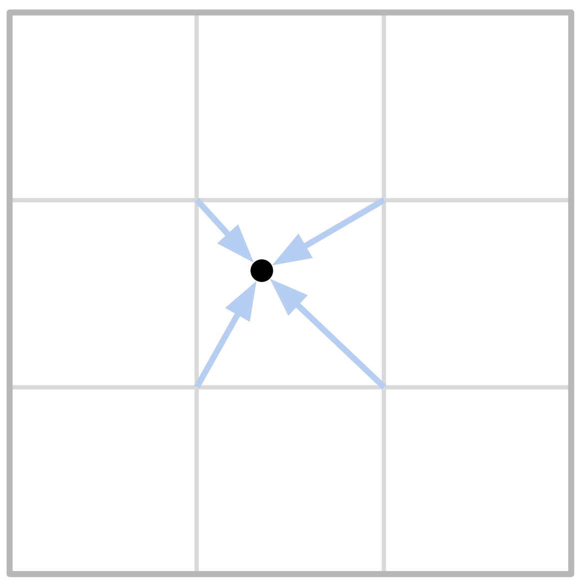
Perlin Noise

- For each point on the grid, we generate 2 pseudo-random numbers that represent a gradient vector



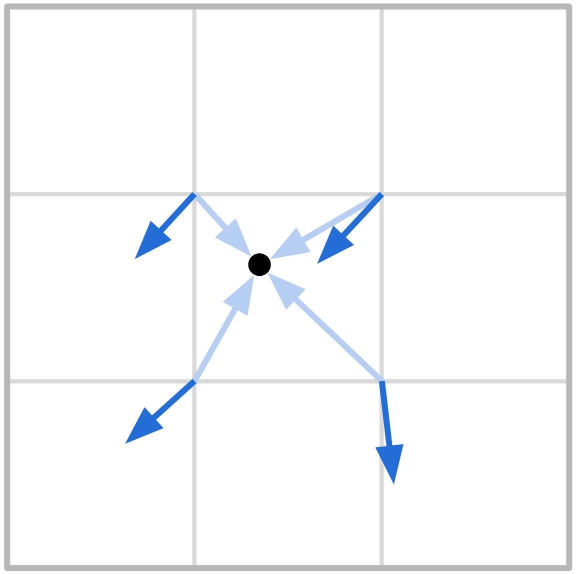
Perlin Noise

- At each texture coordinate (x, y), we calculate 4 "distance" vectors, one from every corner



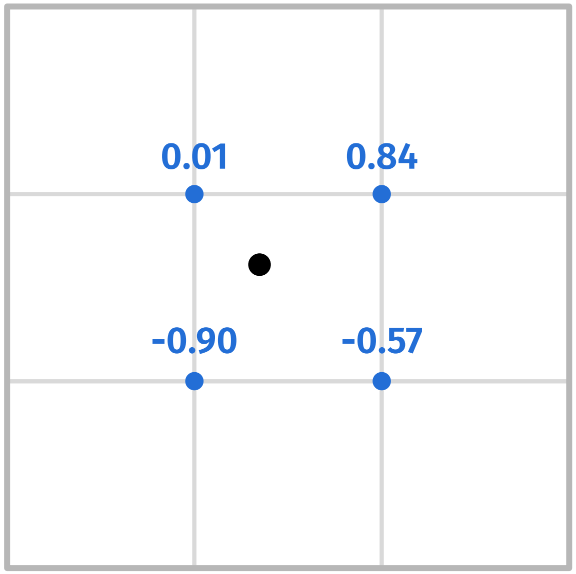
Perlin Noise

- We then calculate the dot product of each distance vector and its corresponding gradient vector



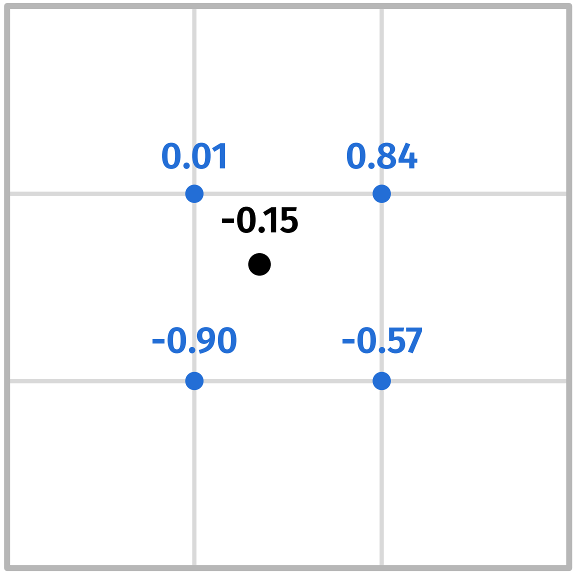
Perlin Noise

- We get 4 *influence values*



Perlin Noise

- Using bicubic interpolation, we get an interpolated value [-1, 1]

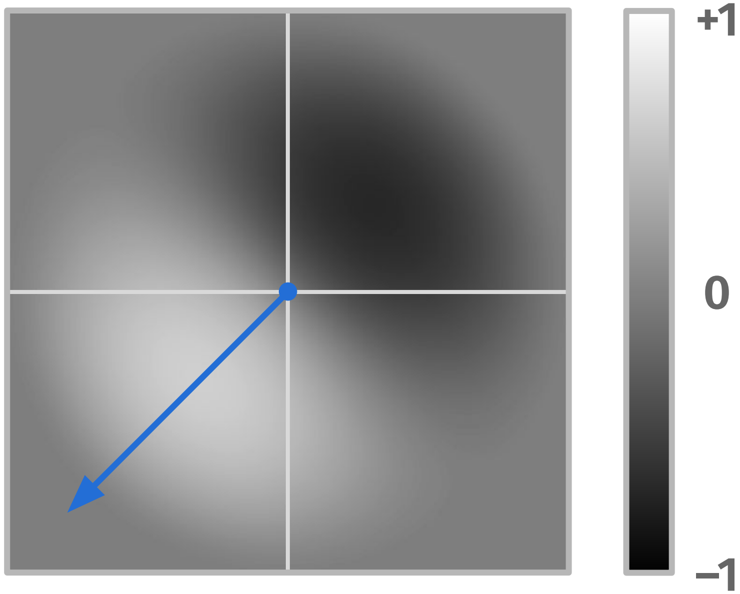


Perlin Noise

- Finally normalize to $[0, 1]$ and render

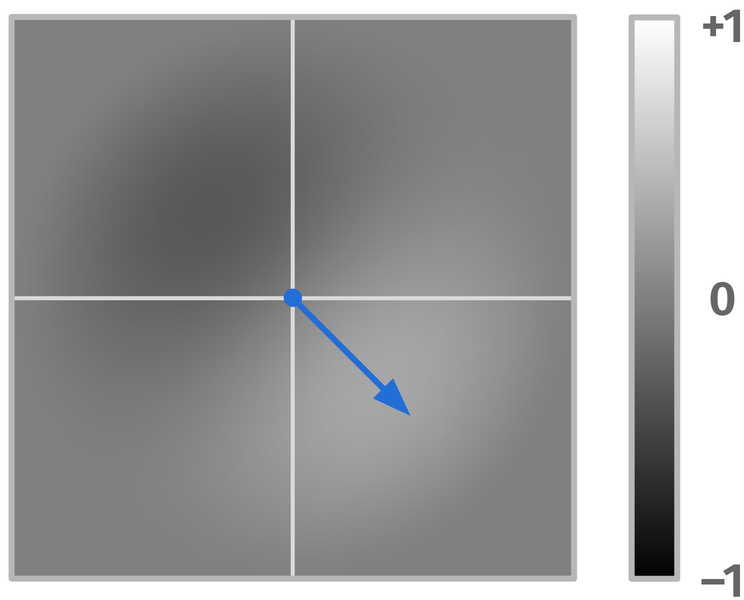
Perlin Noise

- Intuitively, the direction of each gradient vector creates a positive “area” in front of it and a negative “area” behind it



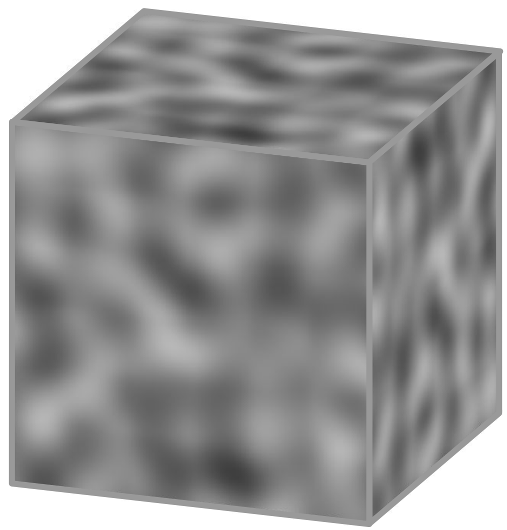
Perlin Noise

- While the magnitude of the vector controls the "intensity" or contrast between the areas



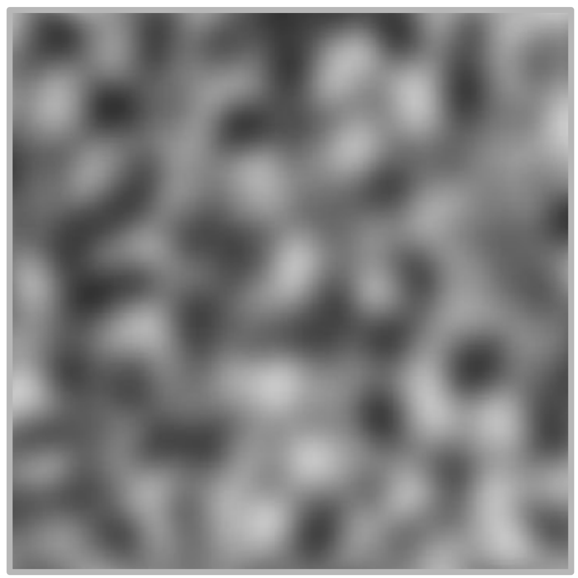
3D Perlin Noise

- We can easily generalize the algorithm to 3D by using 3D gradient vectors on a 3D lattice instead of the 2D grid



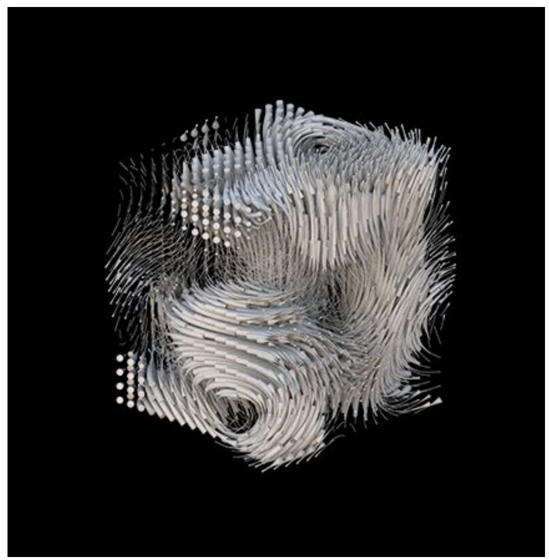
3D Perlin Noise

- By treating the third dimension as time, we can create smooth animated noise!



Higher-Dimensional Perlin Noise

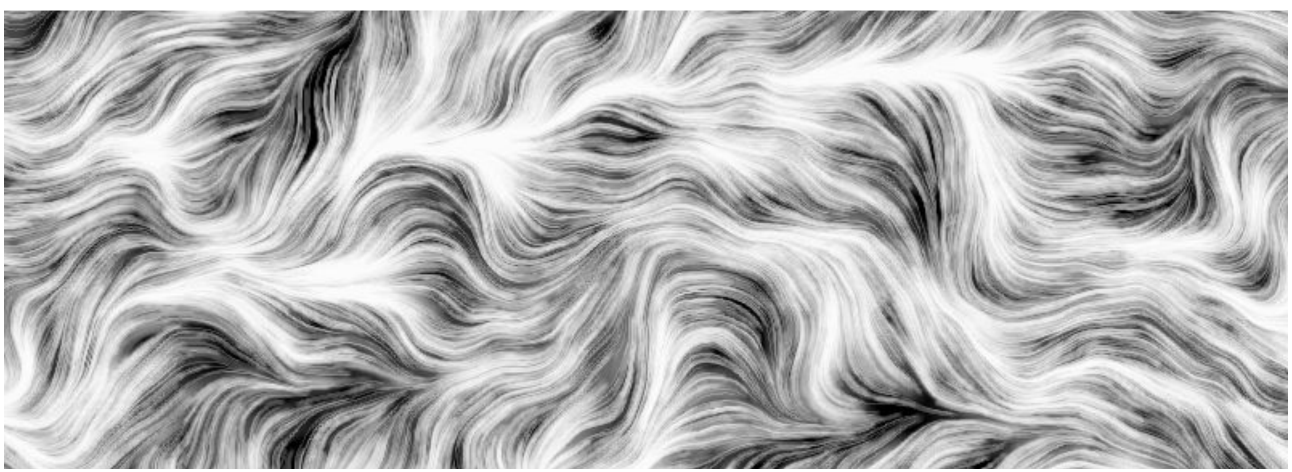
- We can also generalize the Perlin Noise algorithm to dimensions higher than 3



Visualization of 4D Perlin Noise

Perlin Noise

- With Perlin Noise we can achieve all kinds of organic-looking effects
- Perlin Noise is at the heart of of many procedural texturing and modeling algorithms



Fractal Noise

- To add complexity to the noise, we can layer different *frequencies* (scales) on top of each other
- This is called ***Fractal Noise***

Fractal Noise

$$f(x, y) = \sum_{i=1}^{n} g^{i}\,\mathrm{noise}(l^{i}x,\ l^{i}y)$$

Fractal Noise

$$f(x, y) = \sum_{i=1}^{n} g^i \,\mathrm{noise}(l^i x,\, l^i y)$$

- n is the number of layers, known as *octaves*
- g is the amplitude of the layer, known as *gain*
- l is the ratio of change in frequency between layers, known as *lacunarity*
- noise is some noise function, in our case Perlin

Perlin Noise

- Using 3D fractal noise we can create even more realistic natural-looking effects

Clouds using 3D fractal noise

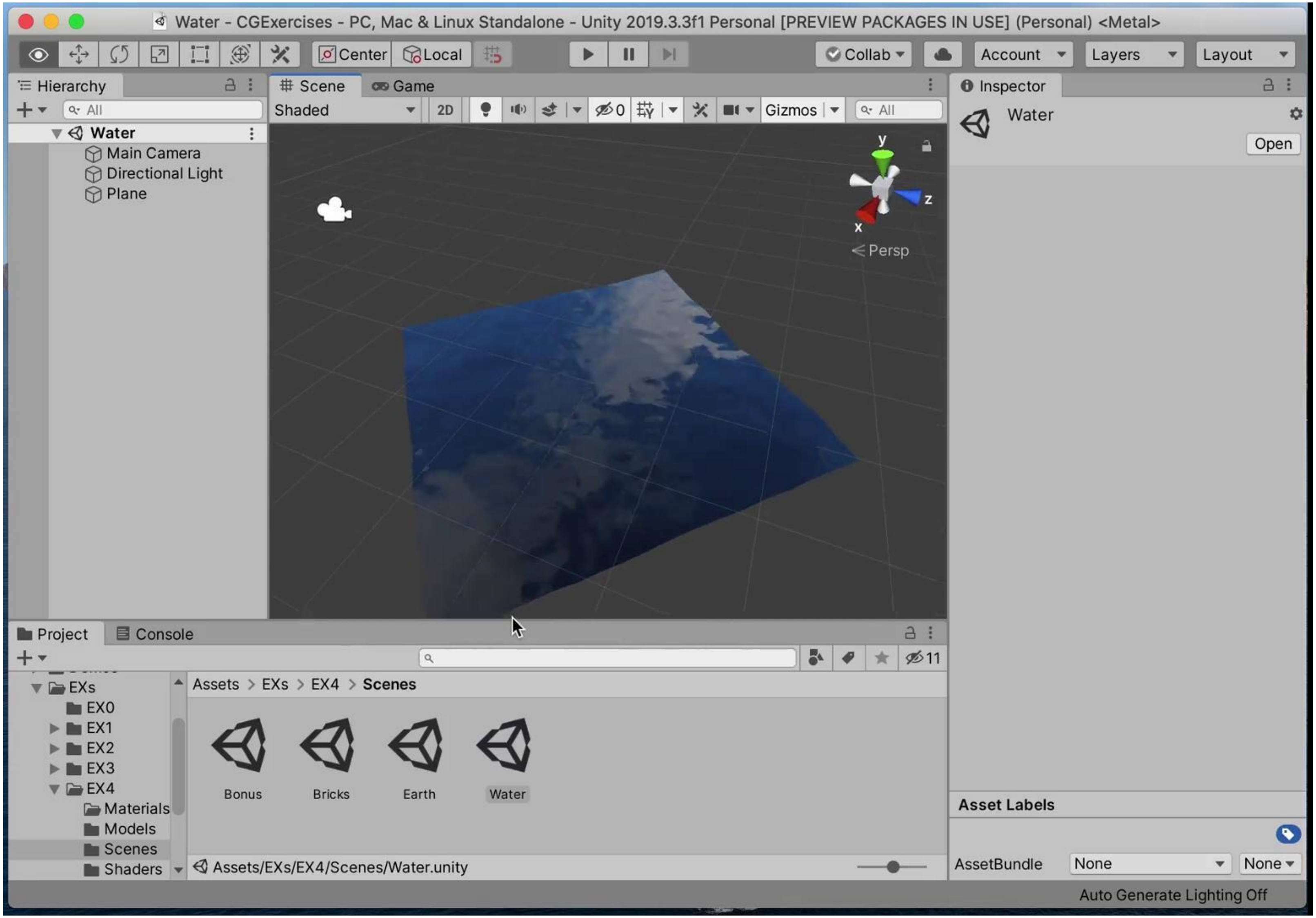
Perlin Noise

- Fractal noise is often used to generate terrain, with the large frequencies creating mountains and the smaller simulating boulders & rocks

Terrain using 2D fractal noise

EX4

- In this exercise you will create a few different materials using shaders
- You will use techniques we have seen in class such as bump mapping, environment mapping and more
- You will implement the value noise and Perlin noise algorithms
- This is a longer exercise! Start early


Water - CGExercises - PC, Mac & Linux Standalone - Unity 2019.3.3f1 Personal [PREVIEW PACKAGES IN USE] (Personal) <Metal>
Center
Local
Collab
Account
Layers
Layout
Hierarchy
Scene
Game
Inspector
Shaded
2D
Gizmos
Water
Main Camera
Directional Light
Plane
Open
Persp
Project
Console
Assets > EXs > EX4 > Scenes
EXs
EX0
EX1
EX2
EX3
EX4
Materials
Models
Scenes
Shaders
Bonus
Bricks
Earth
Water
Assets/EXs/EX4/Scenes/Water.unity
Asset Labels
AssetBundle
None
None
Auto Generate Lighting Off