Stanford CS 248

Interactive Computer Graphics

Noise

Noise

A virtual landscape generated using Perlin noise

Computer Graphics

Procedural Noise

- **Why?**
  - Can't model everything: Use noise to fill in the details using digital synthesis
- **Noise is a class of continuously varying random functions.**
  - Random numbers → Random functions
  - Provides randomness with continuity, e.g., in space, time or other parameter spaces
- Varying degrees of detail & roughness possible.
  - Important for synthesizing/faking fine-scale details
- Varying degrees of smoothness possible.
  - Can be important if taking derivatives, e.g., for shading
- Computer implementations requirements:
  - Fast
  - Low memory
  - Random access
  - Parallel friendly

Overview

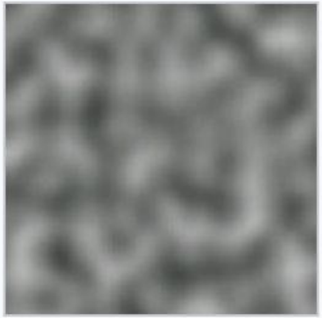
Noise

- What is noise?
- Random numbers
- Value noise
- Fractal noise
- Gradient noise
  - Perlin noise (SIGGRAPH 1985)
  - Improved Perlin noise (SIGGRAPH 2002)
  - Simplex noise
- Noise tricks (mapping, animating, etc.)
- Voronoi noise (Worley, ...)
- Things to try (terrain, worldgen, clouds, animation, etc.)

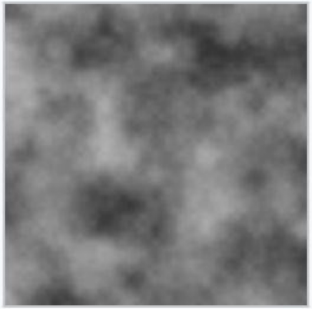
https://en.wikipedia.org/wiki/Perlin_noise

Noise

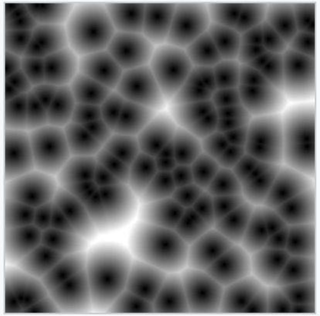
- Random numbers → Random functions
- General purpose image/shape synthesis
- Examples:
  - Perlin noise
    - Perlin, Ken (July 1985). "An Image Synthesizer". *SIGGRAPH Comput. Graph*. **19** (97–8930): 287–296.
  - Wavelet noise
  - Worley noise
  - Etc.
- Many applications!



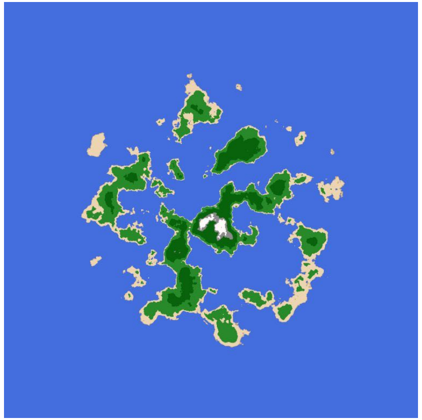
Two-dimensional slice through 3D Perlin noise at z=0



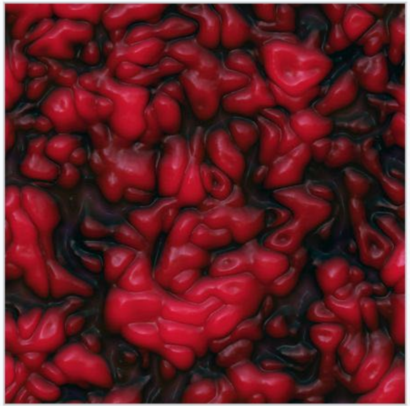
Perlin noise rescaled and added into itself to create fractal noise.



Example picture generated with Worley noise's basic algorithm. Tweaking of seed points and colors would be necessary to make this look like stone.

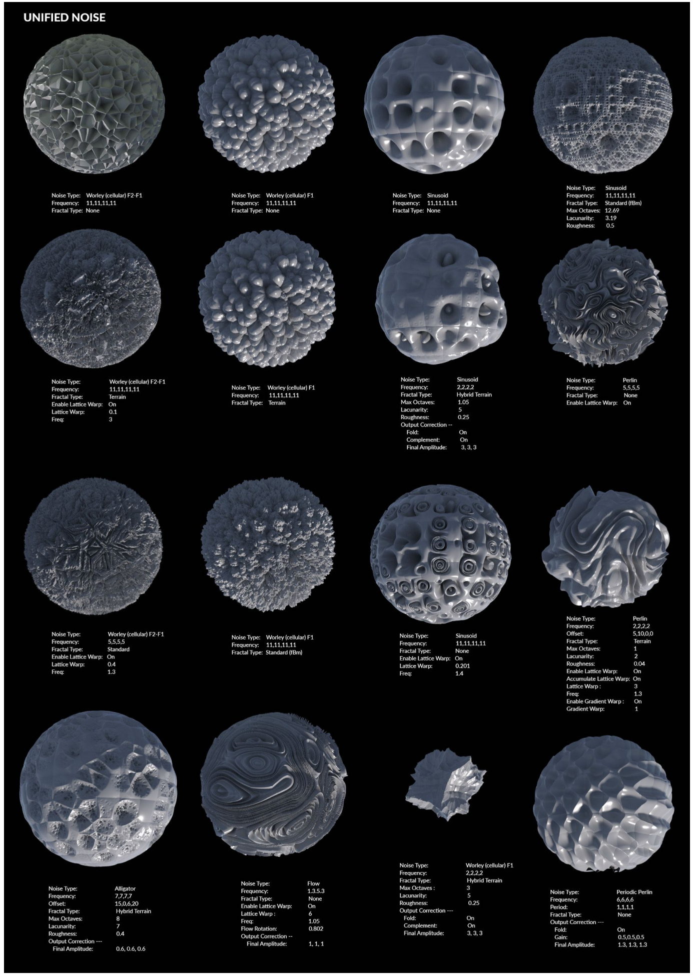


http://yvanscher.com/2017-11-07_Playing-with-Perlin-Noise--Generating-Realistic-Archipelagos-b59f004d8401.html



An organic surface generated with Perlin noise

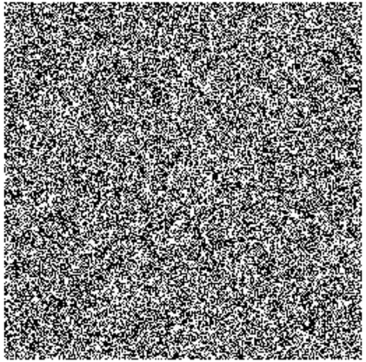
Noise



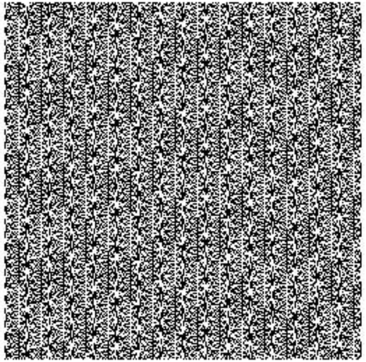
https://www.sidefx.com/docs/houdini/nodes/vop/unifiednoise.html

In the beginning there were just...

Random Numbers



RANDOM.ORG



PHP rand() on Microsoft Windows

- **Random number generators (RNGs)**
- **True random number generators (TRNGs)**
  - Use physical source of randomness
  - E.g., atmospheric noise: http://random.org
- **Pseudo-random number generators (PRNGs)**
  - Fast computer implementation
  - Deterministic
  - Sufficiently random for many applications
  - **Simple:** Linear congruential generator (LCG): → → → → → → → → → → →
    - https://en.wikipedia.org/wiki/Linear_congruential_generator
  - **Better:** Mersenne Twister,
    - https://en.wikipedia.org/wiki/Mersenne_Twister

The generator is defined by recurrence relation:

$$X_{n+1} = (aX_n + c) \mod m$$

where X is the sequence of pseudorandom values, and

$m,\ 0 < m$ — the "modulus"

$a,\ 0 < a < m$ — the "multiplier"

$c,\ 0 \le c < m$ — the "increment"

$X_0,\ 0 \le X_0 < m$ — the "seed" or "start value"

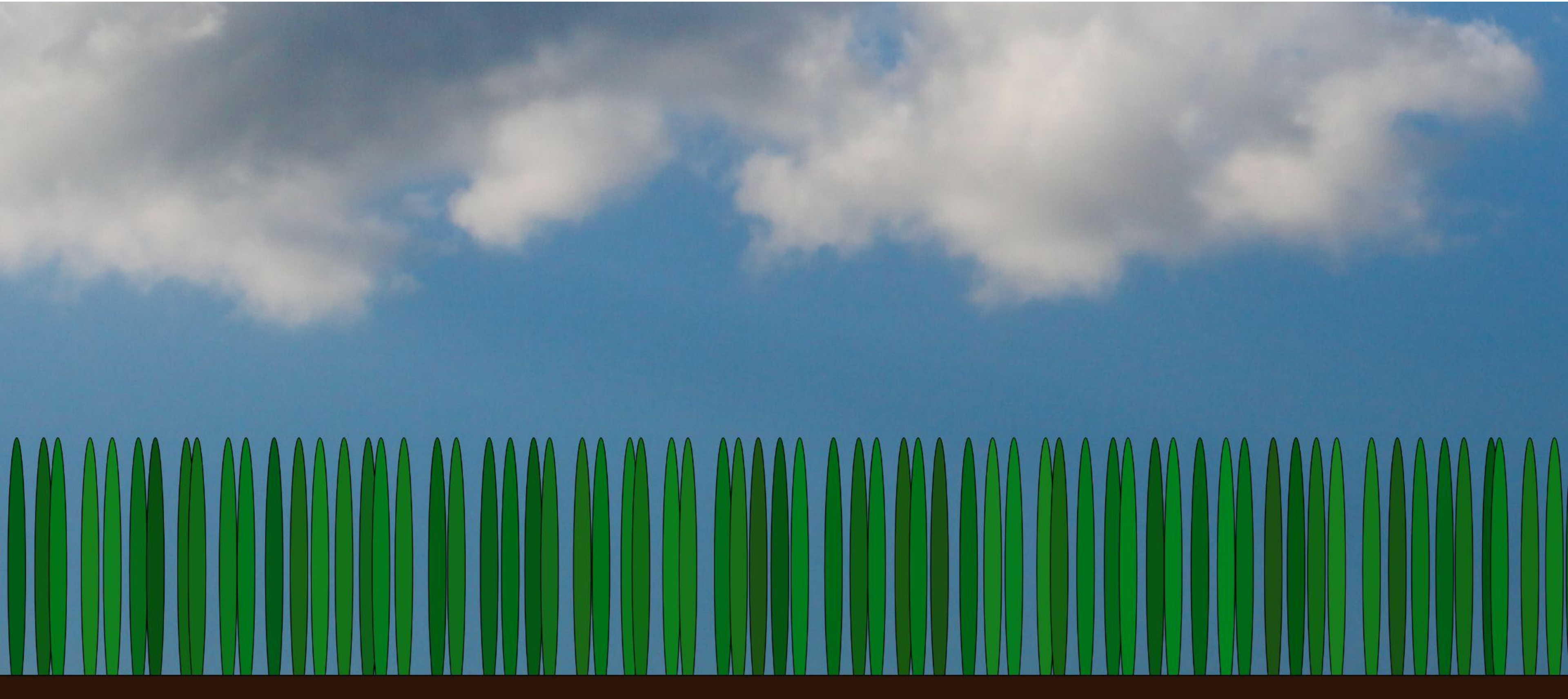
Relaxation

Grass blowing in the wind

https://www.youtube.com/watch?v=yEn8_X7Ei3A

Motivating example

Grass blades rotated using random()

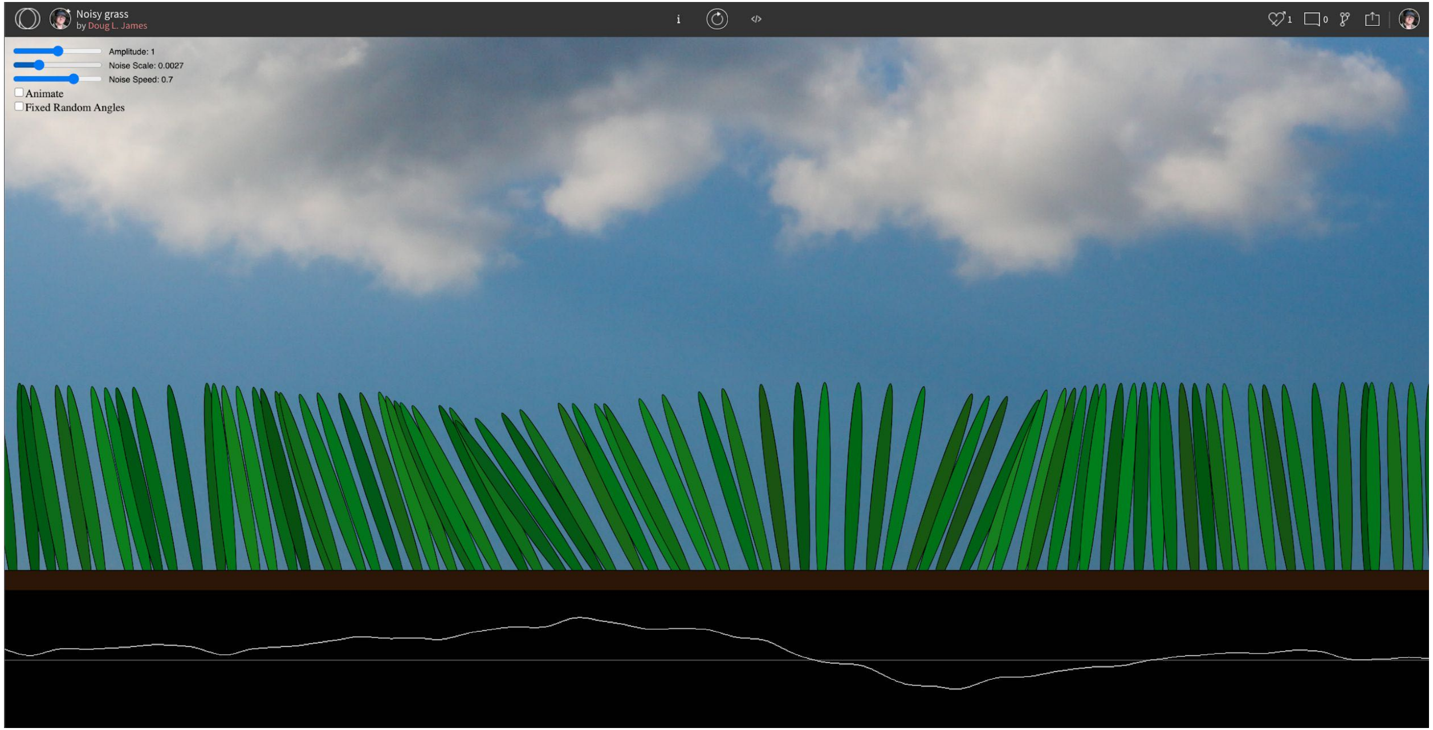


Motivating example

Grass blades rotated using random() :/

Motivating example

"Noisy Grass"



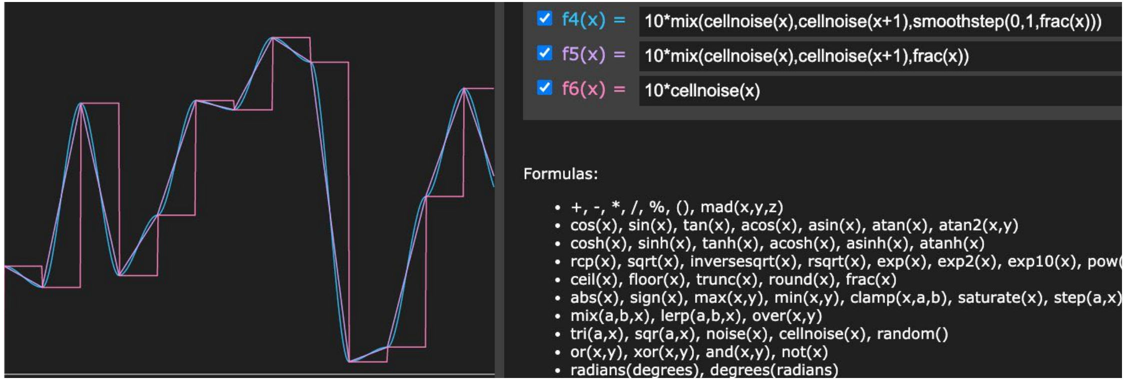
https://www.openprocessing.org/sketch/990261

Types of Noise

Simplest noise

Value Noise

- Simple noise model
- Blend random grid values



http://www.iquilezles.org/apps/graphtoy/?f4(x)=10*mix(cellnoise(x),%20cellnoise(x+1),smoothstep(0,1,frac(x)))&f5(x)=10*mix(cellnoise(x),%20cellnoise(x+1),frac(x))&f6(x)=10*cellnoise(x)

http://en.wikipedia.org/wiki/Value_noise

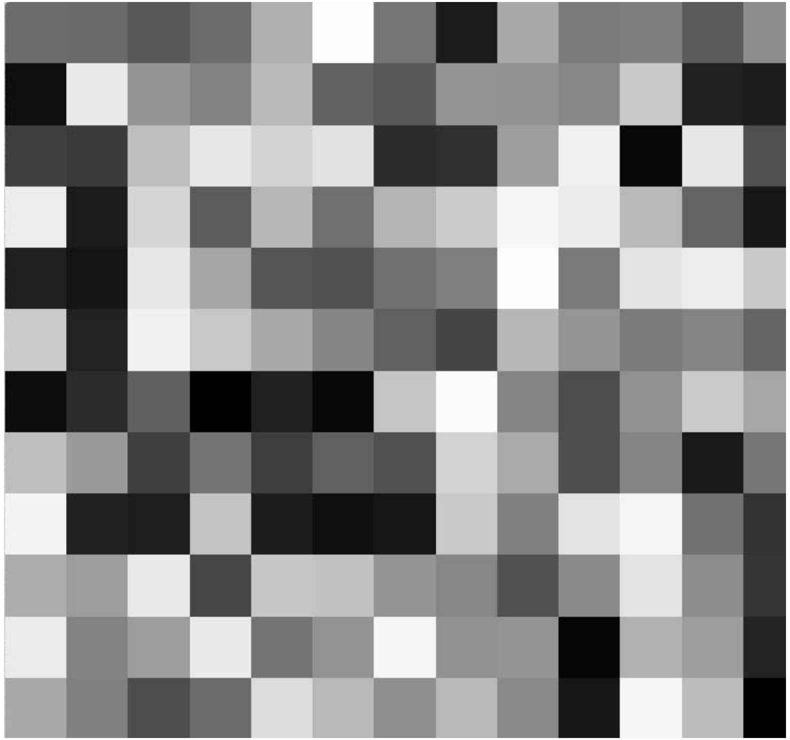
Simplest noise

Value Noise

- Simple noise model
- Blend random grid values
- Pros:
  - Very fast
  - Very low memory
    - Values can be generated on the fly
- Cons:
  - Grid artifacts are more apparent
- Sometimes good enough

Simplest noise

Value Noise



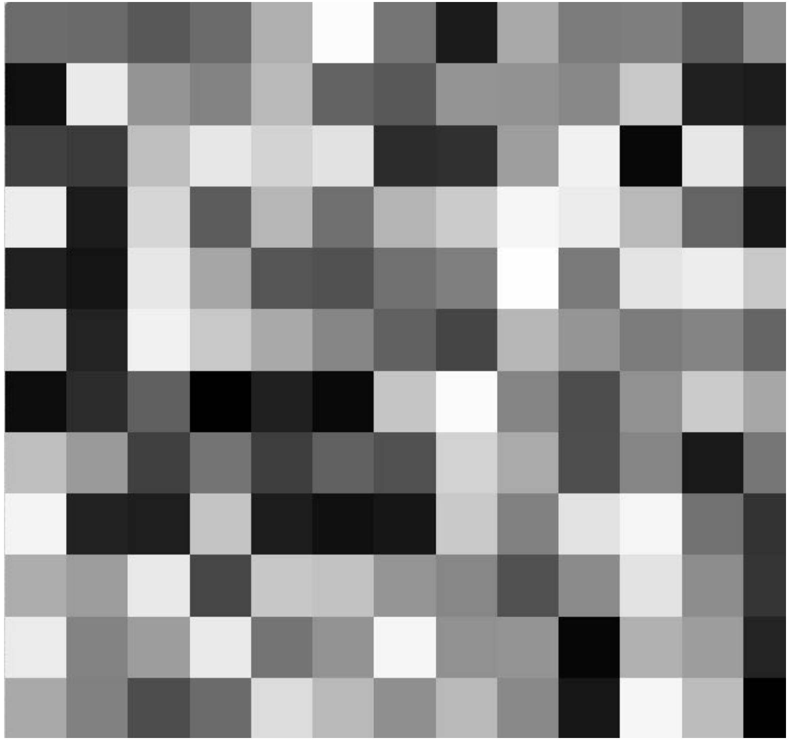
Random Values (at vertices)

interpolate →

Blended (linear)

Simplest noise

Value Noise



Random Values (at vertices)

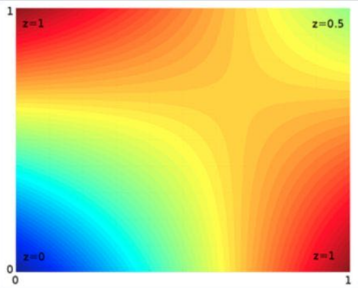
interpolate →

Blended (cubic)

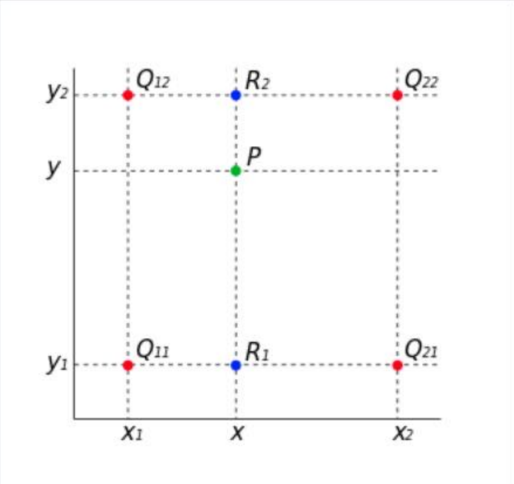
https://en.wikipedia.org/wiki/Bilinear_interpolation

Blending functions

Bilinear Interpolation



Example of bilinear interpolation on the unit square with the *z* values 0, 1, 1 and 0.5 as indicated. Interpolated values in between represented by color.



The four red dots show the data points and the green dot is the point at which we want to interpolate.

Unit square [edit]

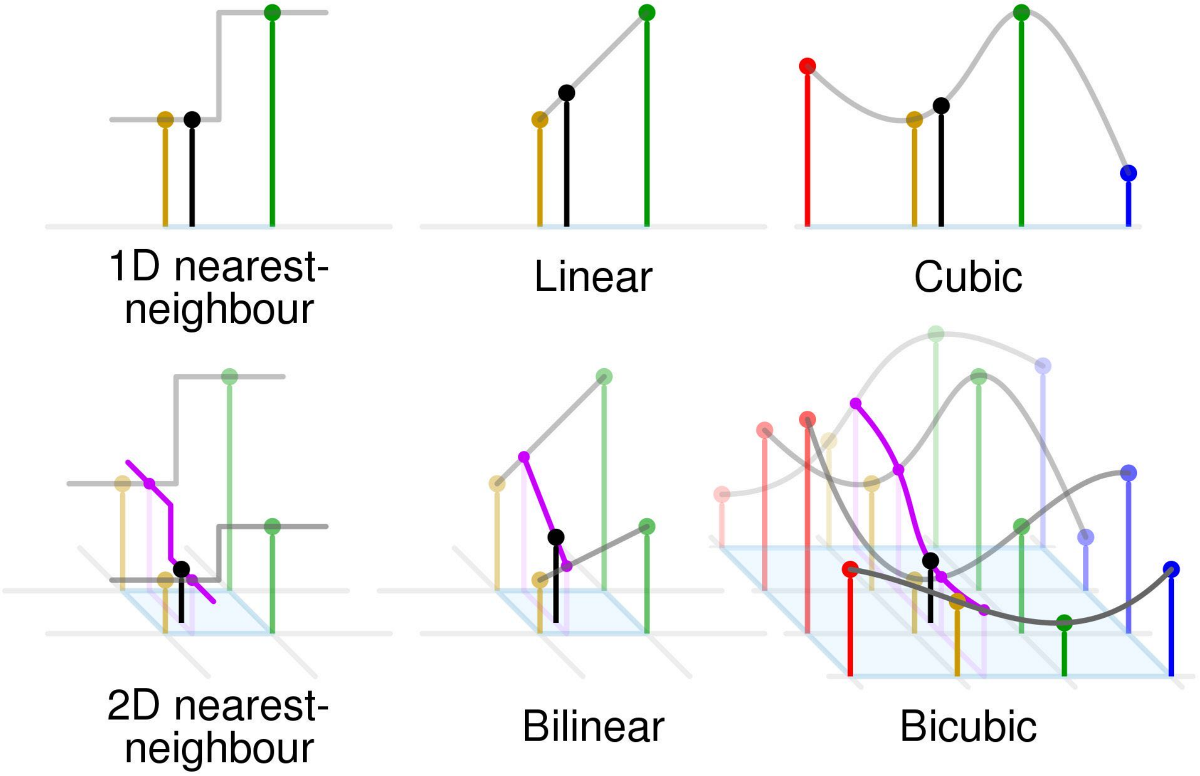
If we choose a coordinate system in which the four points where *f* is known are (0, 0), (1, 0), (0, 1), and (1, 1), then the interpolation formula simplifies to

$$f(x,y) \approx f(0,0)(1-x)(1-y) + f(1,0)x(1-y) + f(0,1)(1-x)y + f(1,1)xy,$$

https://en.wikipedia.org/wiki/Bicubic_interpolation

Blending functions

Bicubic Interpolation



Issues with bicubic interpolation:

Need 4^2 values

- Expensive
- 4^3 values for 3D tricubic

Or 4 values + 12 gradients

- Don’t have gradient info

https://en.wikipedia.org/wiki/Cubic_Hermite_spline

Blending functions

Cubic blending function

Simple Hack:

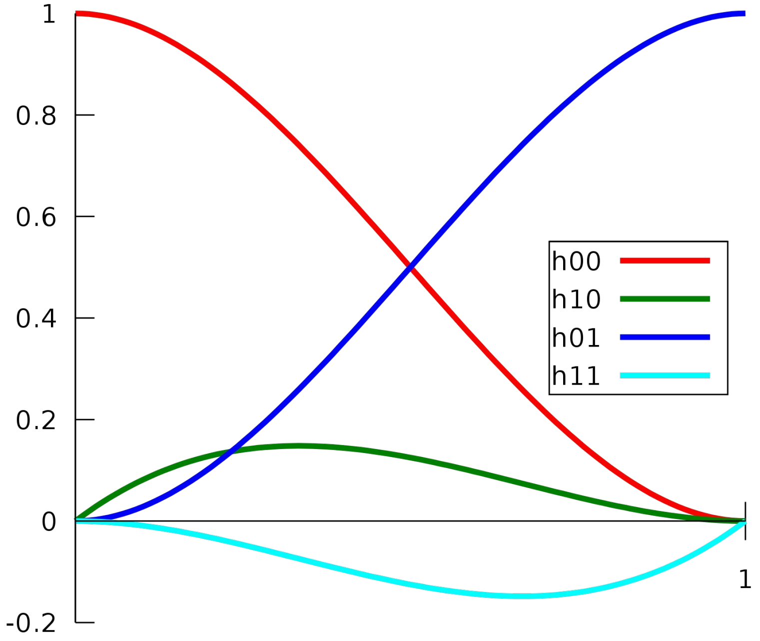
- Assume gradients are zero at vertices.
- Just blend data using Hermite functions(!)

Example: 1D Hermite interpolation set $\boldsymbol{m}_0=\boldsymbol{m}_1=\mathbf{0}$

$$\boldsymbol{p}(t) = h_{00}(t)\boldsymbol{p}_0 + \cancel{h_{10}(t)\boldsymbol{m}_0} + h_{01}(t)\boldsymbol{p}_1 + \cancel{h_{11}(t)\boldsymbol{m}_1}$$

	expanded	factorized	Bernstein
$h_{00}(t)$	$2t^3 - 3t^2 + 1$	$(1+2t)(1-t)^2$	$B_0(t) + B_1(t)$
$h_{10}(t)$	$t^3 - 2t^2 + t$	$t(1-t)^2$	$\frac{1}{3} \cdot B_1(t)$
$h_{01}(t)$	$-2t^3 + 3t^2$	$t^2(3-2t)$	$B_3(t) + B_2(t)$
$h_{11}(t)$	$t^3 - t^2$	$t^2(t-1)$	$-\frac{1}{3} \cdot B_2(t)$

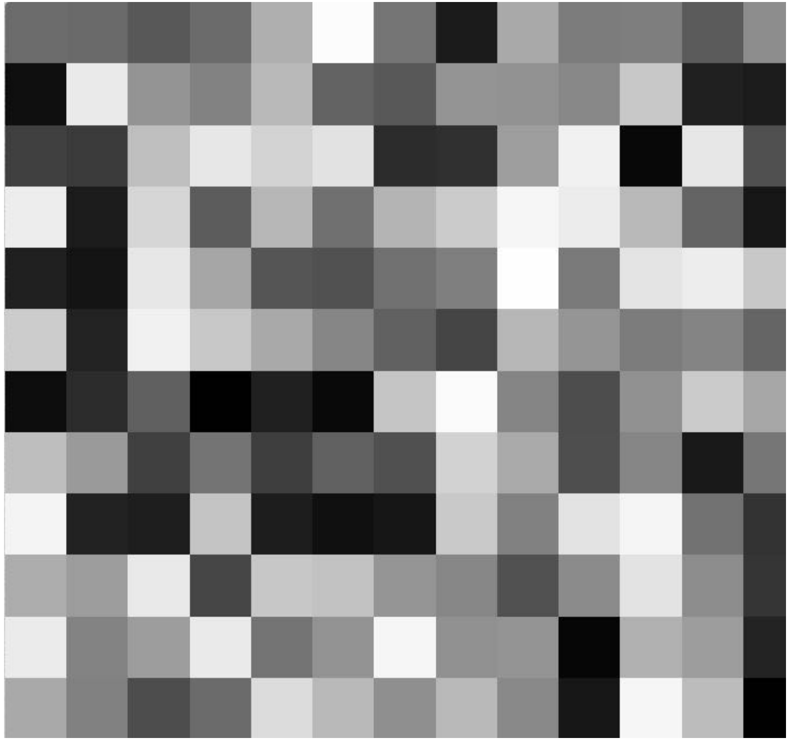
a.k.a. "smoothstep" function



https://en.wikipedia.org/wiki/Cubic_Hermite_spline#/media/File:HermiteBasis.svg

Simplest noise

Value Noise



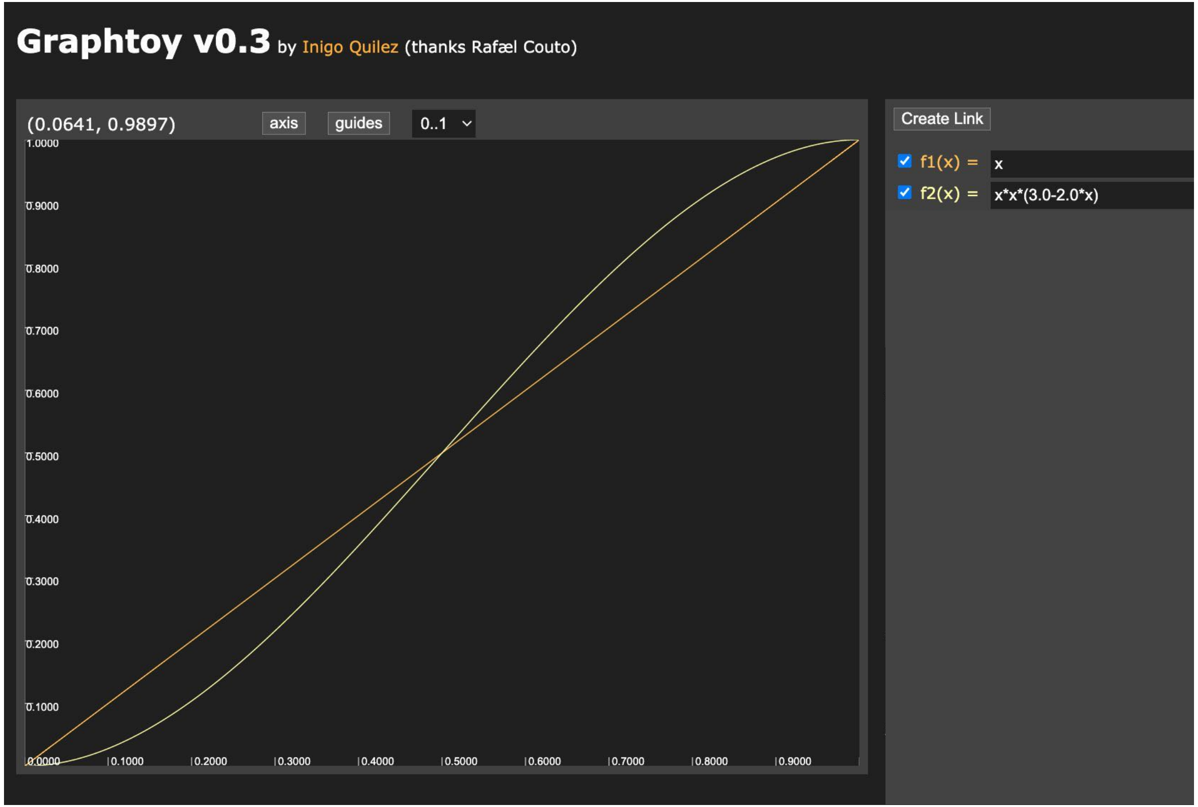
Random Values (at vertices)

interpolate

Blended (cubic)

Blending functions

Blending Vertex Values



https://graphtoy.com

Multiple octaves

Fractal Noise (a.k.a. “turbulence”)

COARSE → FINE

$$\text{noise}(\mathbf{p}) + \frac{1}{2}\text{noise}(2\mathbf{p}) + \frac{1}{4}\text{noise}(4\mathbf{p}) + \ldots$$

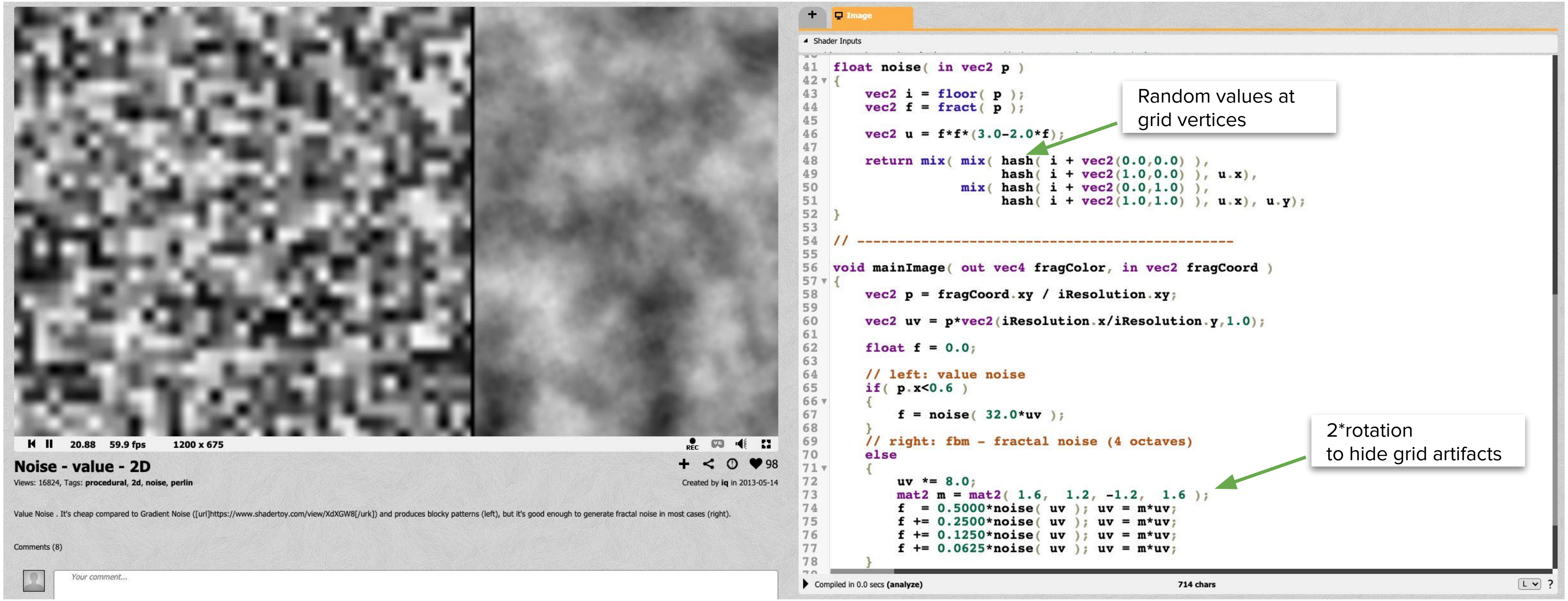
$$= \sum_{\ell=0}^{L} \frac{1}{2^\ell}\text{noise}(2^\ell\,\mathbf{p})$$

“Fall-off” factor
(so-called “yellow noise”)

- Special case of Fractal Brownian motion (**fBm**) noise
- https://en.wikipedia.org/wiki/Fractional_Brownian_motion
- See **Inigo Quilez**’s **fBm tutorial**

Demo

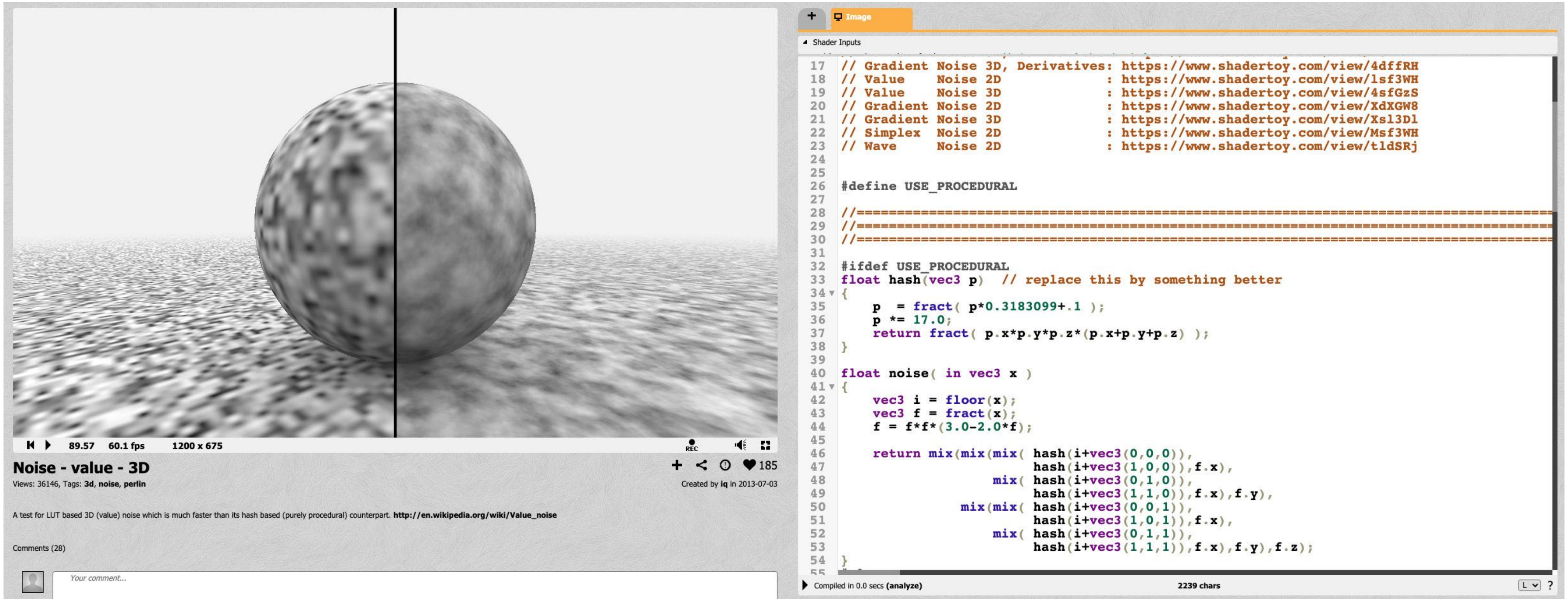
Value Noise (2D)



https://www.shadertoy.com/view/lsf3WH

Demo

Value Noise (3D)



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

https://en.wikipedia.org/wiki/Gradient_noise

Better noise

Gradient Noise

Different approach:

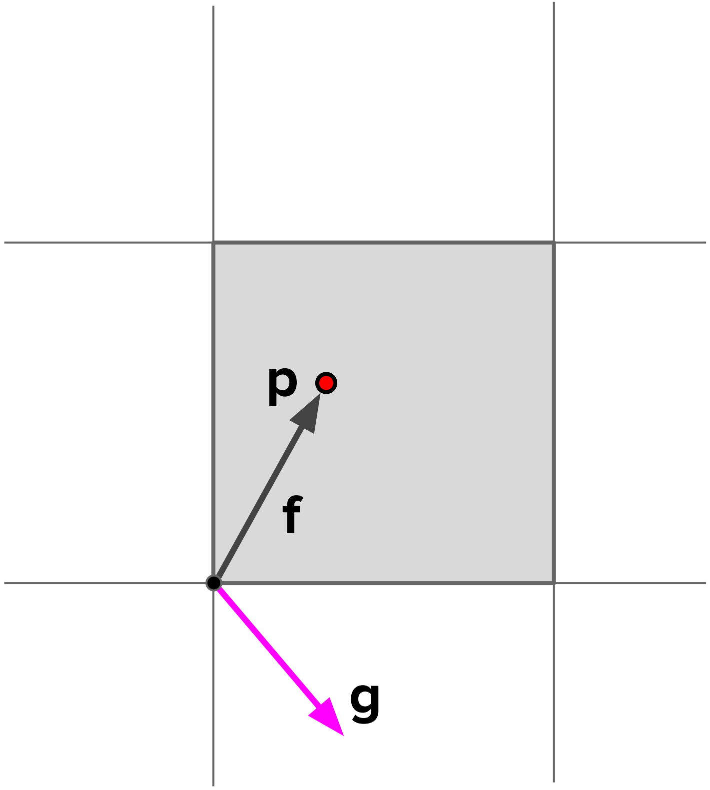
Zero vertex values but ***random gradients g****=(g_x,g_y).*

Each vertex's gradient noise uses linear approx.:

$$g_x\, dx + g_y\, dy = \mathbf{g} \cdot \mathbf{f}$$

where **f**=(dx,dy) is the vertex delta.

Blend all vertex values across cell like before.

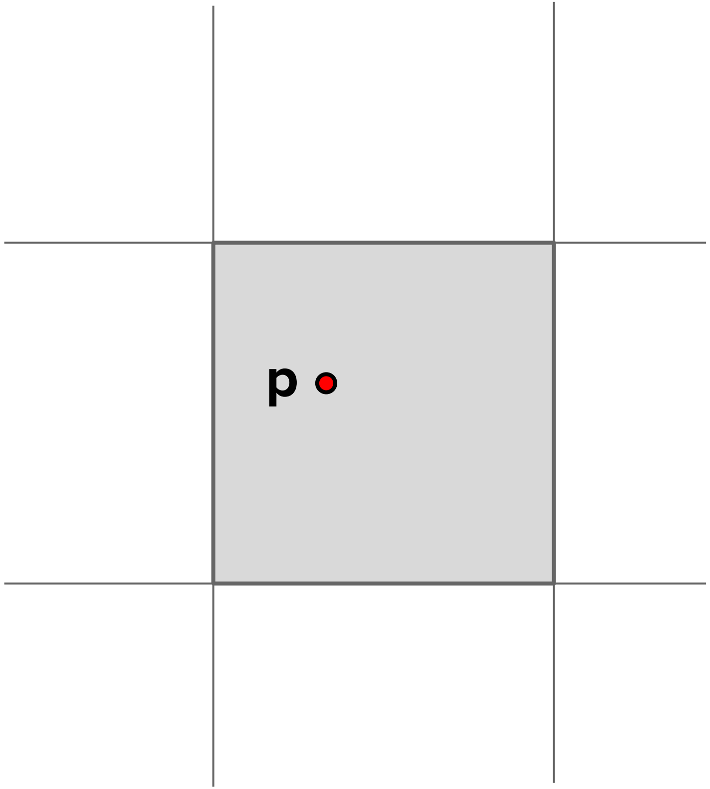


https://en.wikipedia.org/wiki/Gradient_noise

Better noise

Gradient Noise

Given point, **p**



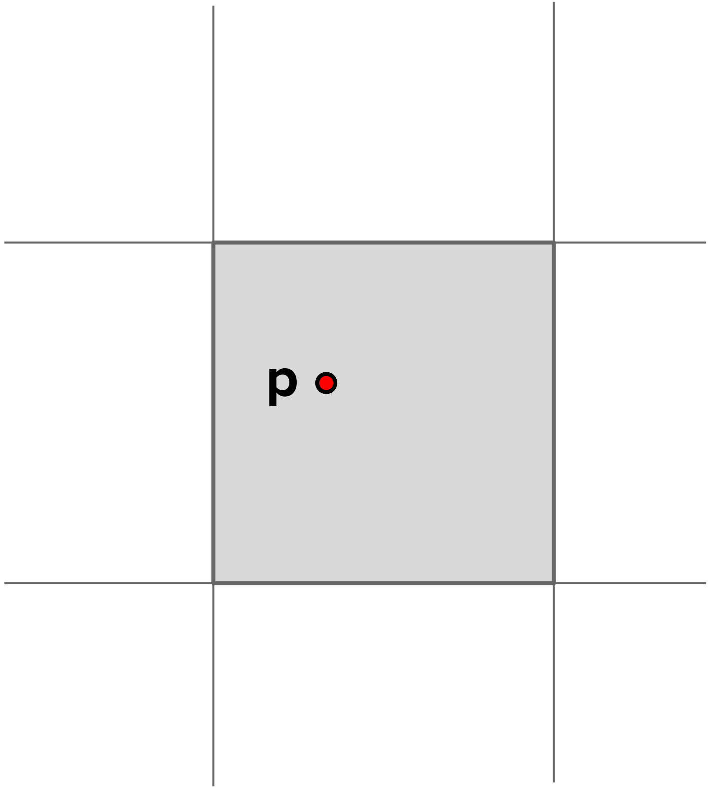
https://en.wikipedia.org/wiki/Gradient_noise

Better noise

Gradient Noise

Given point, **p**

Get containing cell: **i=floor(p)**



https://en.wikipedia.org/wiki/Gradient_noise

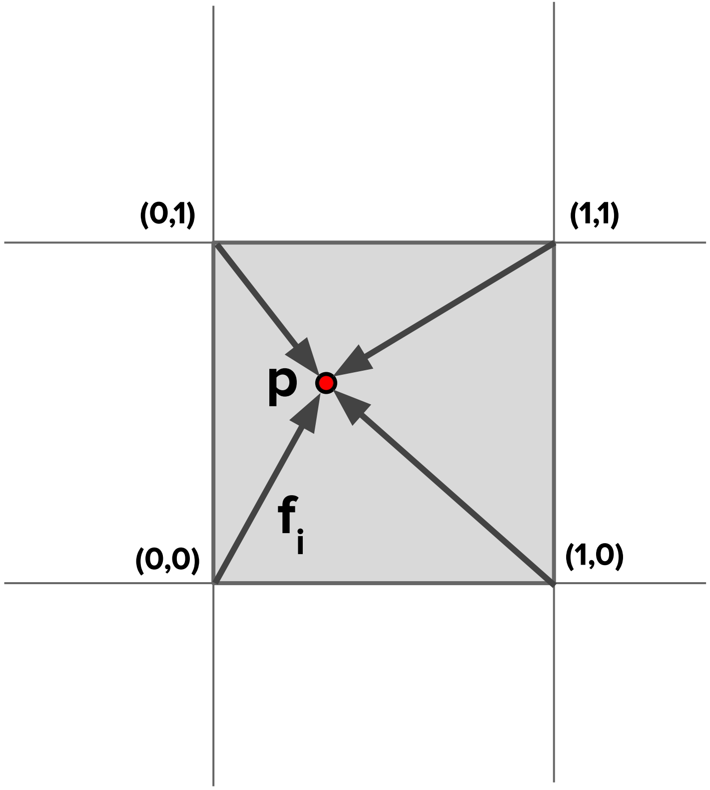
Better noise

Gradient Noise

Given point, **p**

Get containing cell: **i=floor(p)**

Extract **vertex offsets, f=fract(p)**



https://en.wikipedia.org/wiki/Gradient_noise

Better noise

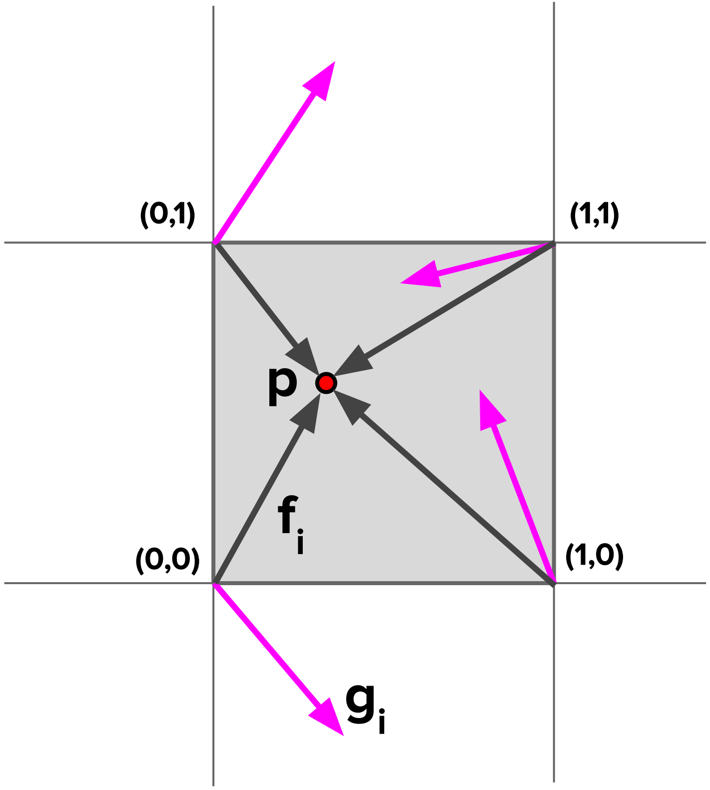
Gradient Noise

Given point, **p**

Get containing cell: **i=floor(p)**

Extract **vertex offsets, f=fract(p)**

Lookup **random vertex gradient, $\mathbf{g_i}$**, i=1..4.



https://en.wikipedia.org/wiki/Gradient_noise

Better noise

Gradient Noise

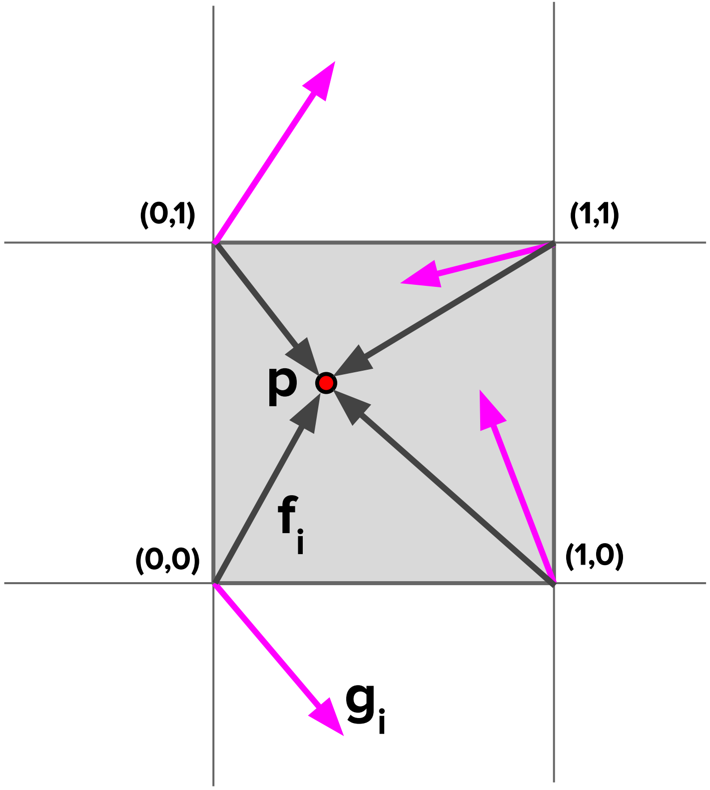
Given point, **p**

Get containing cell: **i=floor(p)**

Extract **vertex offsets, f=fract(p)**

Lookup **random vertex gradient, g_i**, i=1..4.

Dot vertex offset with gradient: ($f_i \cdot g_i$), i=1..4.



https://en.wikipedia.org/wiki/Gradient_noise

Better noise

Gradient Noise

Given point, **p**

Get containing cell: **i=floor(p)**

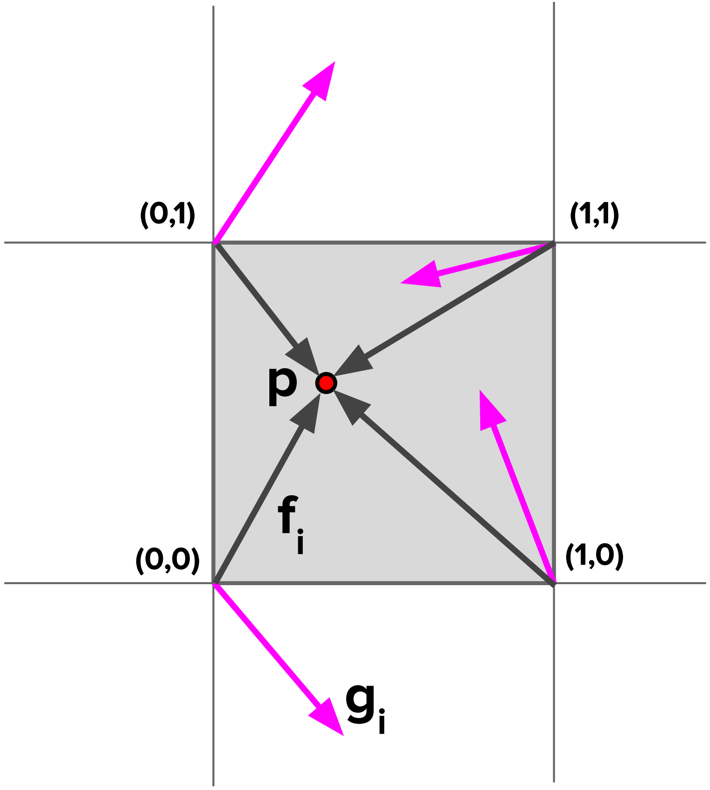
Extract **vertex offsets, f=fract(p)**

Lookup **random vertex gradient, g_i**, i=1..4.

Dot vertex offset with gradient: $(\mathbf{f}_i \bullet \mathbf{g}_i)$, i=1..4.

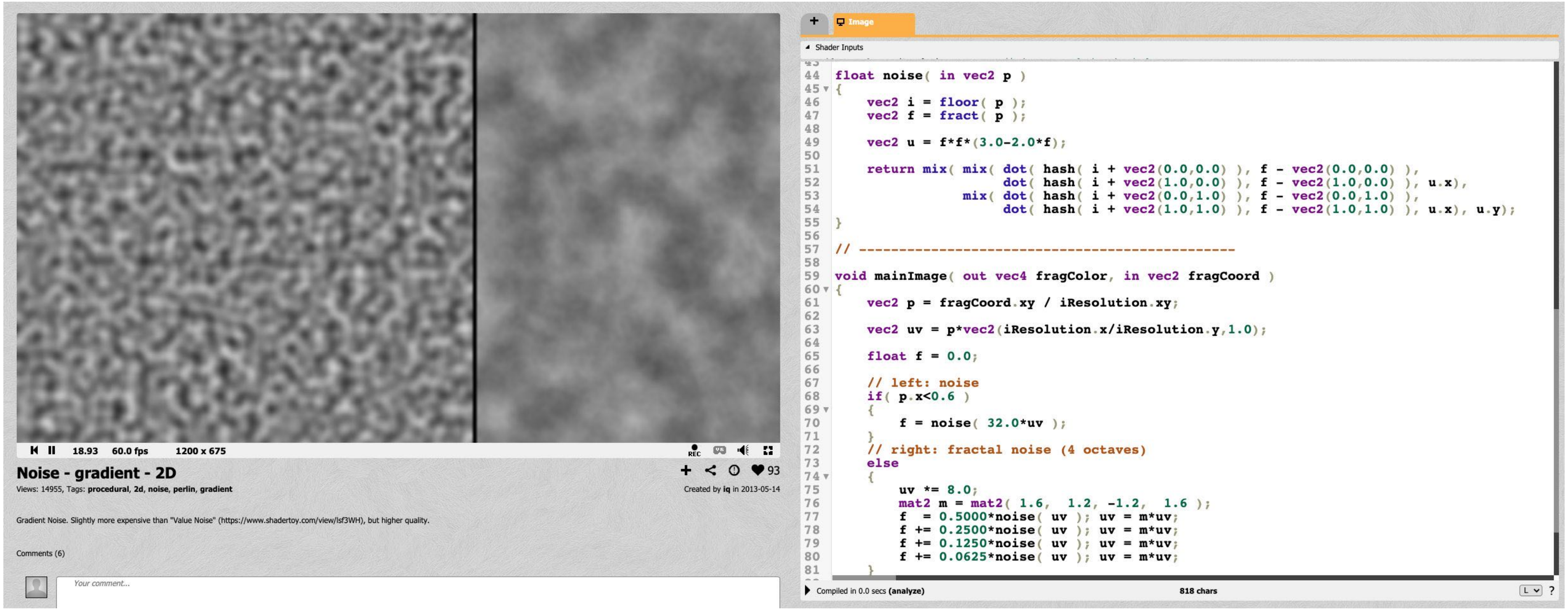
Blend vertex values across cell:

$$\text{Noise} = \Sigma_{i=1..4}\, w_i\ (\mathbf{f}_i \bullet \mathbf{g}_i)$$



Demo

Gradient Noise 2D

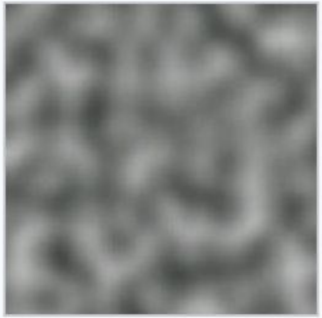


https://www.shadertoy.com/view/XdXGW8

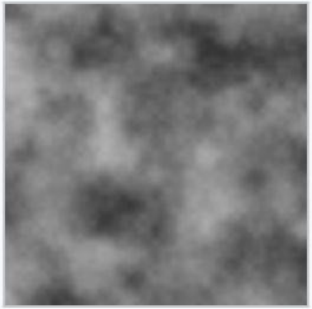
The classic noise function

Perlin Noise (SIGGRAPH 1985)

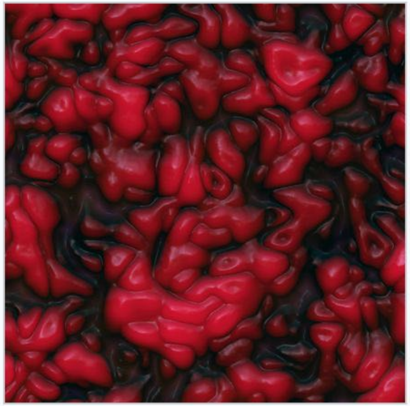
- Famous instance of gradient noise
  - Perlin, Ken (July 1985). "An Image Synthesizer". SIGGRAPH Comput. Graph. 19 (97–8930): 287–296.
- Cubic blending
- Precomputed gradient table
- Permutation map stored
  - Enables fast pseudo-random gradient lookup



Two-dimensional slice through 3D Perlin noise at z=0



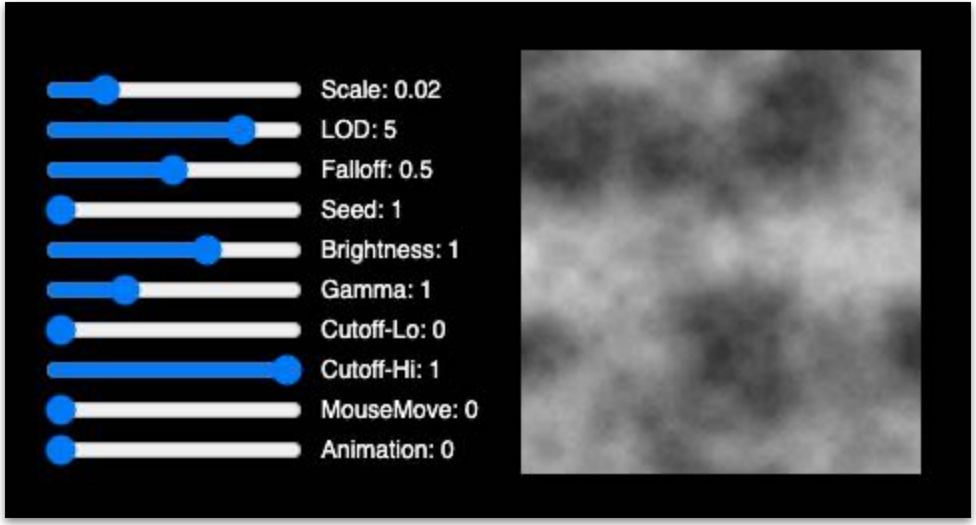
Perlin noise rescaled and added into itself to create fractal noise.



An organic surface generated with Perlin noise https://p5js.org/reference/#/p5/noise

OpenProcessing Demo

Perlin noise() function



https://www.openprocessing.org/sketch/978945

Relevant links

Ken's Webpage

Noise and Turbulence

In 1997 I received a Technical Achievement Award from the Academy of Motion Picture Arts and Sciences for work I had done on procedural texture. For example, the NYU Torch on the right is made entirely from procedural textures (except for the text along the bottom). The flame, background, and metal and marble handle are not actually 3D models - they are all entirely faked with textures. A hi-res image of a marble vase I made using this technique can be found here. A bunch of other texture images I created can be found here.

I created an on-line tutorial about noise, which you can view here.

I then improved it, and wrote a paper about that. You can see code and examples of the improved version here.

You can play with designing noise-based textures yourself with a really nice interactive Java Applet created by Justin Legakis. Also, the interactive fractal planet demo on my home page is made using these techniques.

It seems that my techniques found their way into the various software packages, such as Autodesk *Maya, SoftImage, 3D Studio Max, Dynamation, RenderMan,* etc., that folks use to make the effects for feature films, which is way cool. Movies look better now, and I guess that makes me a good American.

https://mrl.nyu.edu/~perlin/doc/oscar.html

From Ken's noise talk

MAKING NOISE

Ken Perlin

https://web.archive.org/web/20160306054434/http://www.noisemachine.com/talk1/20.html

From Ken's noise talk

Algorithm

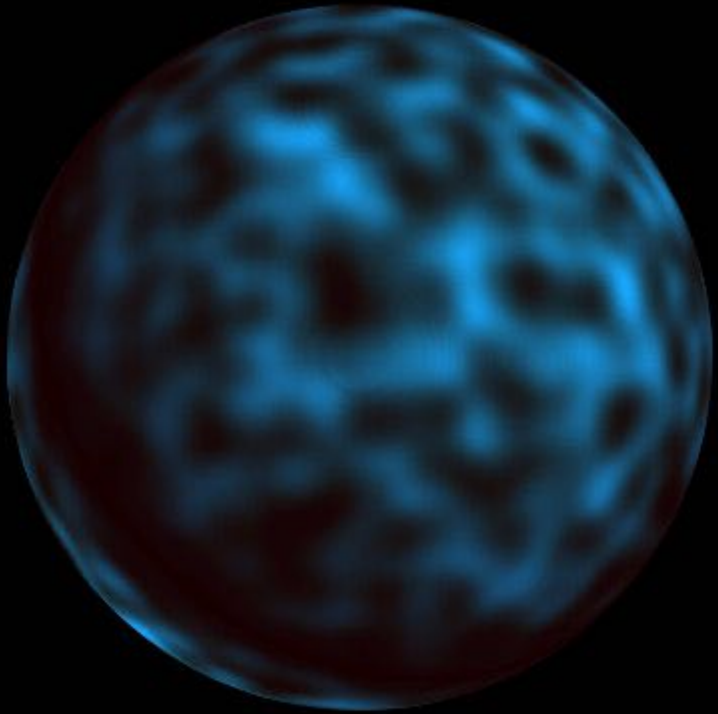
1. Given an input point
2. For each of its neighboring grid points:
   - Pick a "pseudo-random" gradient vector
   - Compute linear function (dot product)
3. Take weighted sum, using ease curves

From Ken's noise talk

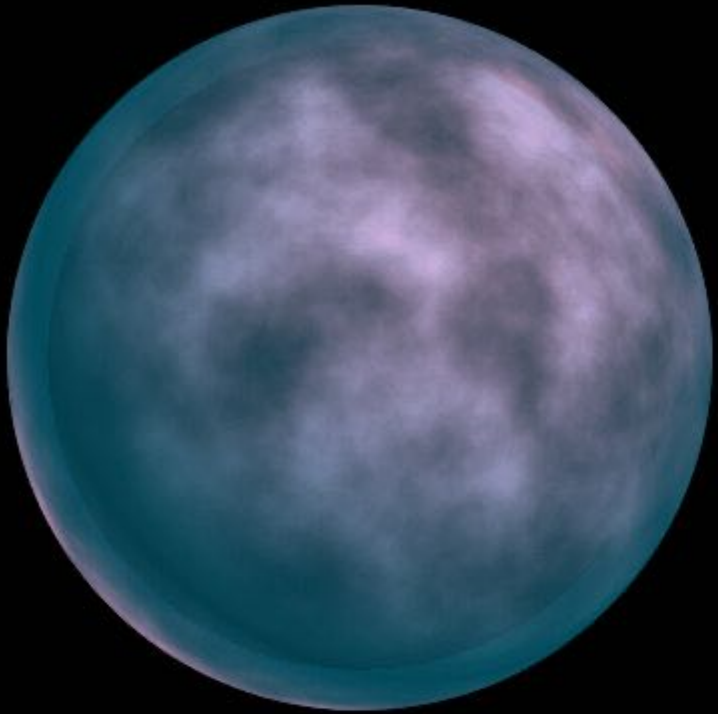
Computing the pseudo-random gradient:

- Precompute table of permutations P[n]
- Precompute table of gradients G[n]
- $G = G[\, (\, i + P[\, (j + P[k])_{\text{mod } n} \,]\,)_{\text{mod } n} \,]$

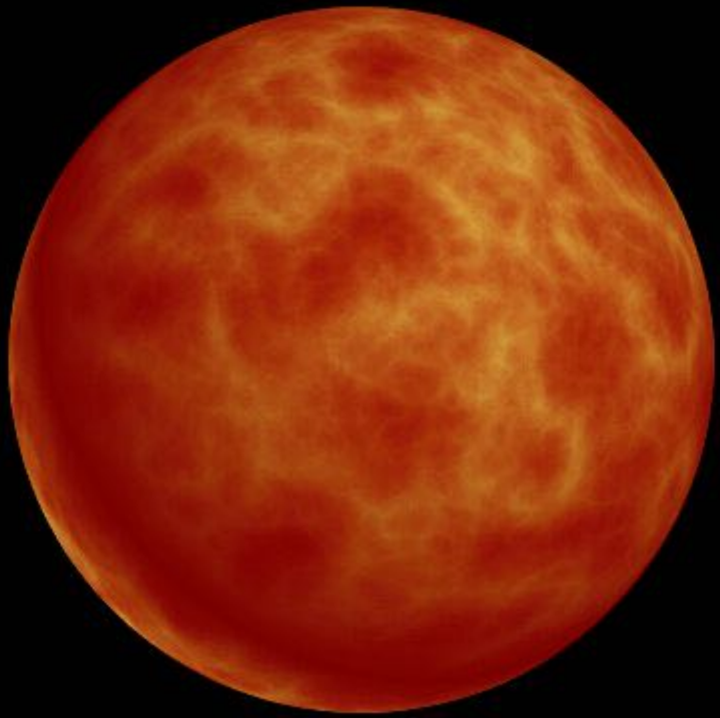
Some example implementations:
https://rosettacode.org/wiki/Perlin_noise#C



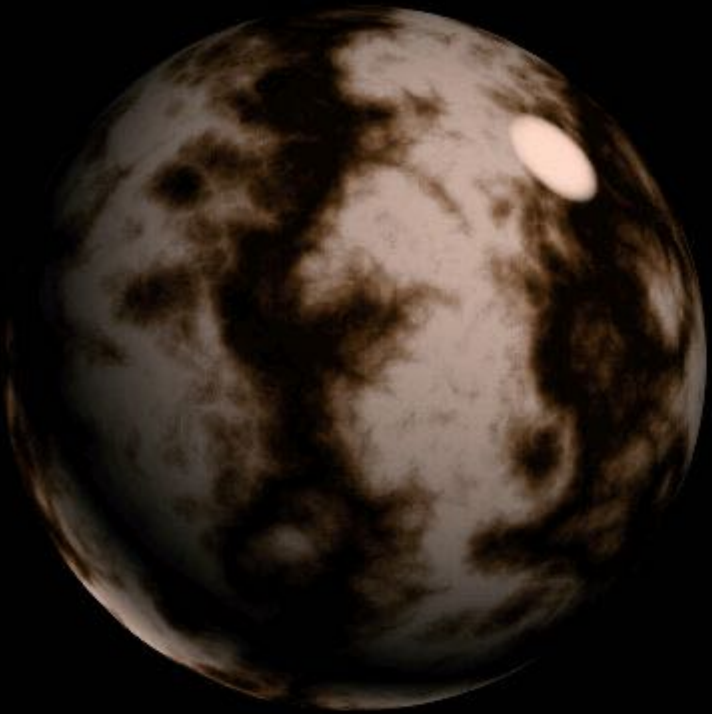
$$\mathrm{noise}(\mathbf{p})$$



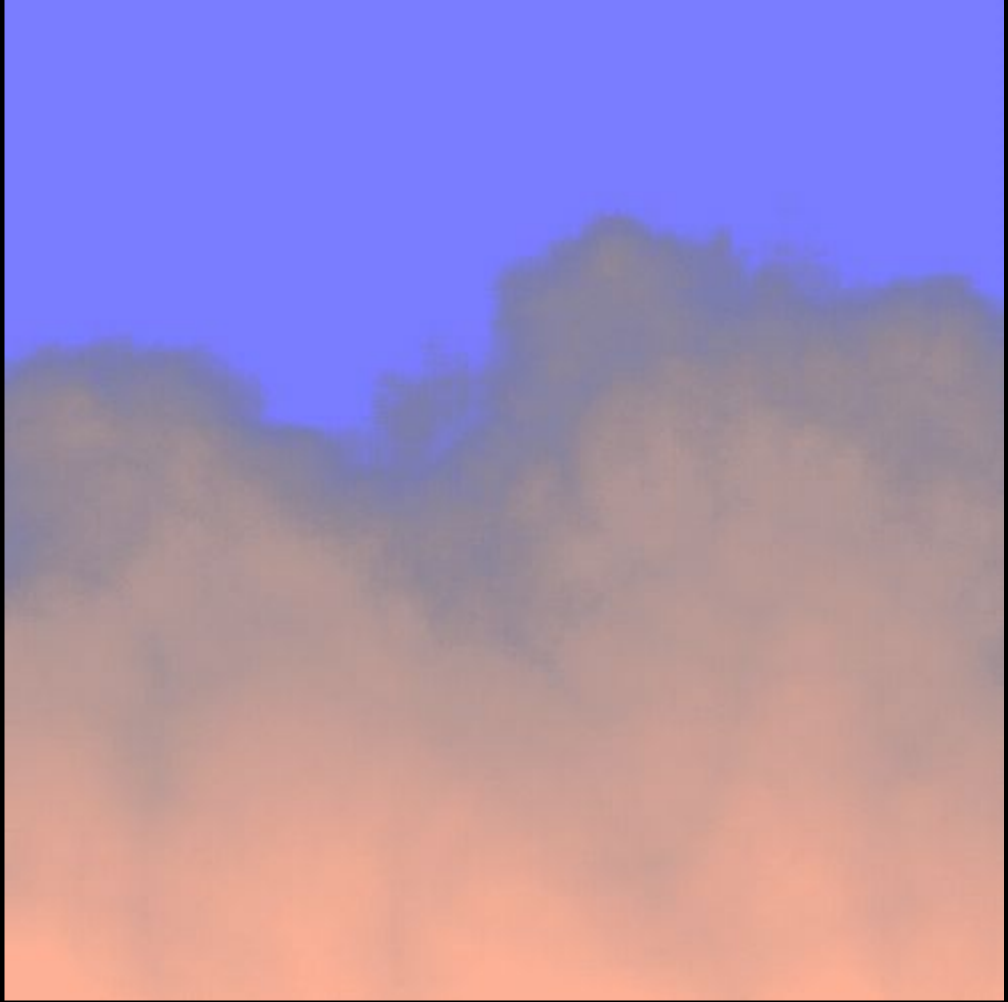
$$\text{noise}(\mathbf{p}) + \frac{1}{2}\text{noise}(2\mathbf{p}) + \frac{1}{4}\text{noise}(4\mathbf{p}) + \ldots$$



$$|\mathrm{noise}(\mathbf{p})| + \frac{1}{2}|\mathrm{noise}(2\mathbf{p})| + \frac{1}{4}|\mathrm{noise}(4\mathbf{p})| + \ldots$$



$$\sin(x + |\text{noise}(\mathbf{p})| + \frac{1}{2}|\text{noise}(2\mathbf{p})| + \frac{1}{4}|\text{noise}(4\mathbf{p})| + \ldots)$$



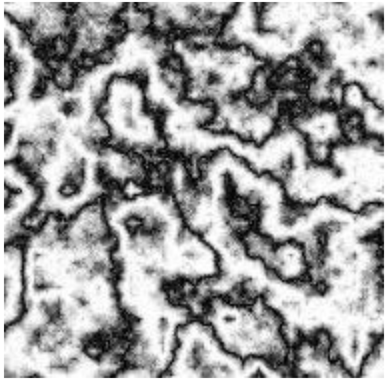
colorMap(y + F(x,y,z))

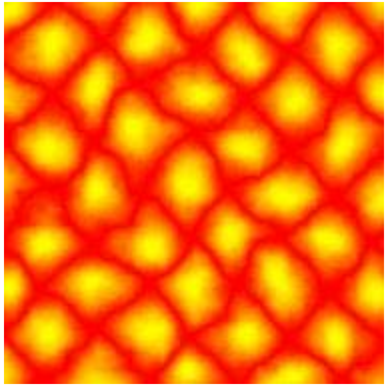
Further reading

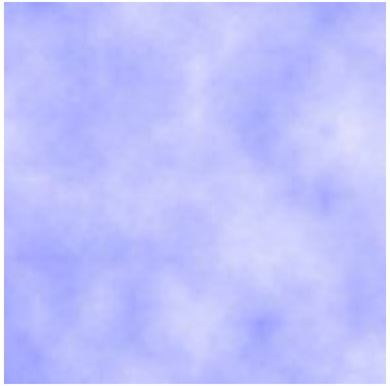
Noise Tricks

See “Texture generation using random noise” by Lode

- https://lodev.org/cgtutor/randomnoise.html





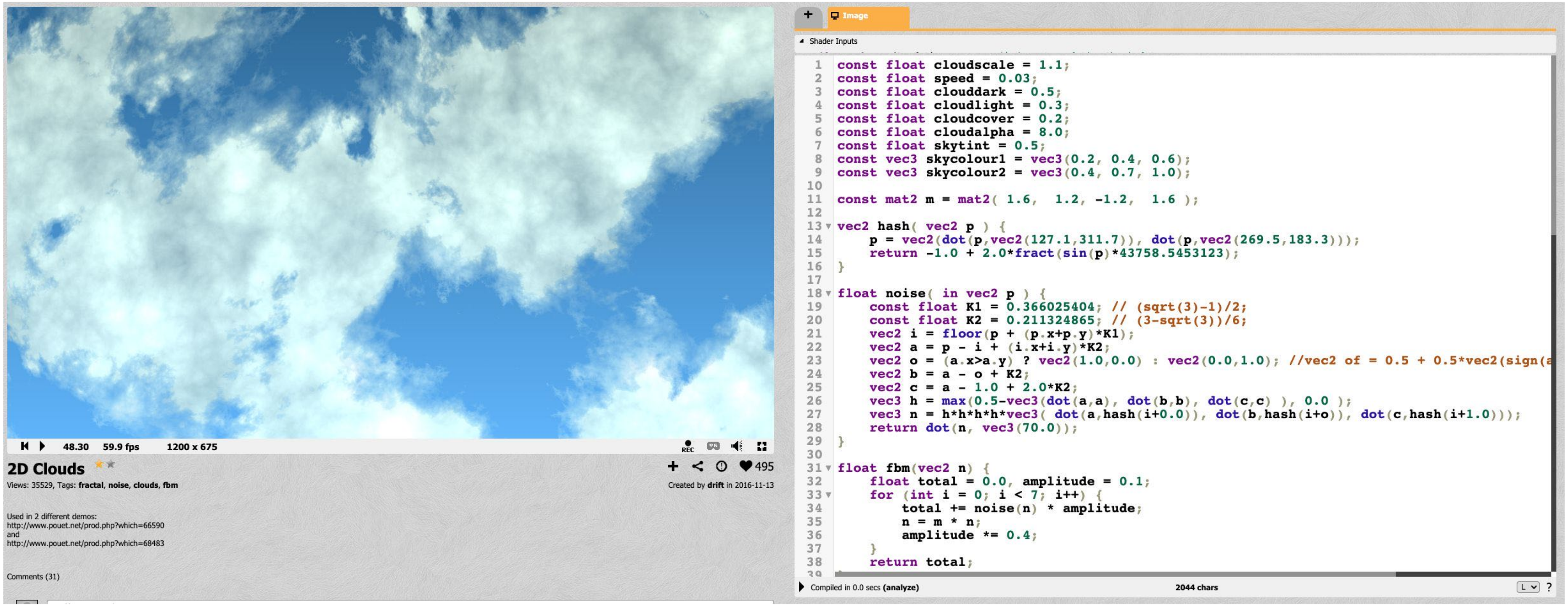


Shadertoy demo I made

Simple Cloud

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

Shadertoy Demo
"2D Clouds"



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

SIGGRAPH 2002

Improved Perlin Noise

Smoother interpolants & better gradients.

Figure 1a: Noise-displaced superquadric with old interpolants

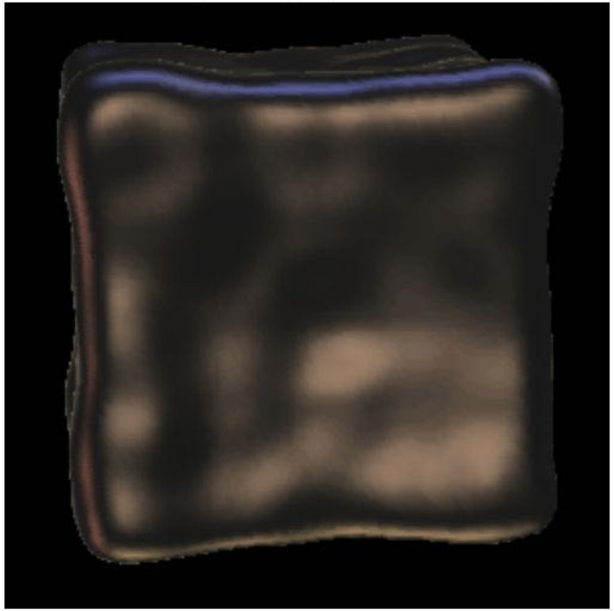


Figure 1b: Noise-displaced superquadric with new interpolants

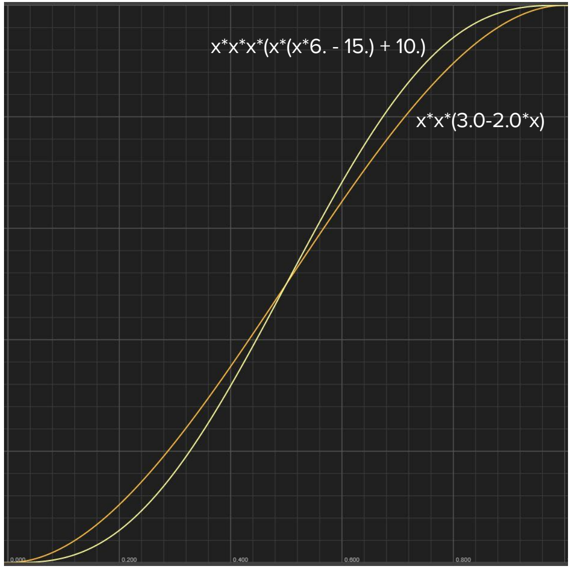
SIGGRAPH 2002

Improved Perlin Noise

Smoother interpolant: Cubic ➔ Quintic blend function:

x*x*(3.0-2.0*x) ➔ x*x*x*(x*(x*6. - 15.) + 10.)

[Graphtoy Plot]



Better gradients: Avoids gradient precomputation & storage, and has less bias.

Explanatory article:
https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/perlin-noise-part-2/improved-perlin-noise

SIGGRAPH 2002

Improved Perlin Noise

Smoother interpolants & better gradients.

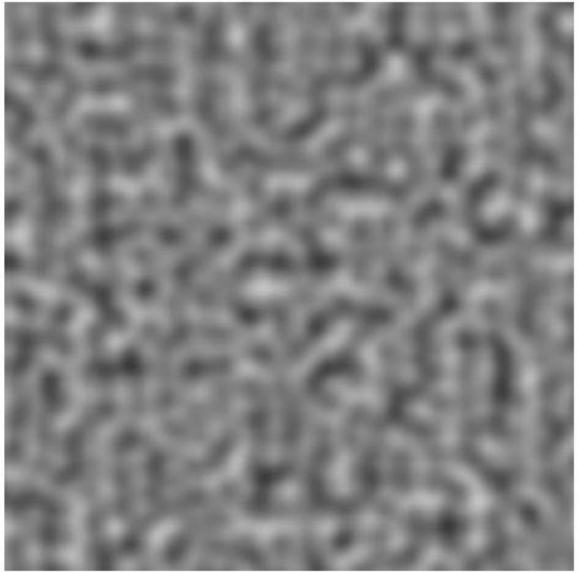


Figure 2a: High-frequency Noise, with old gradient distributions

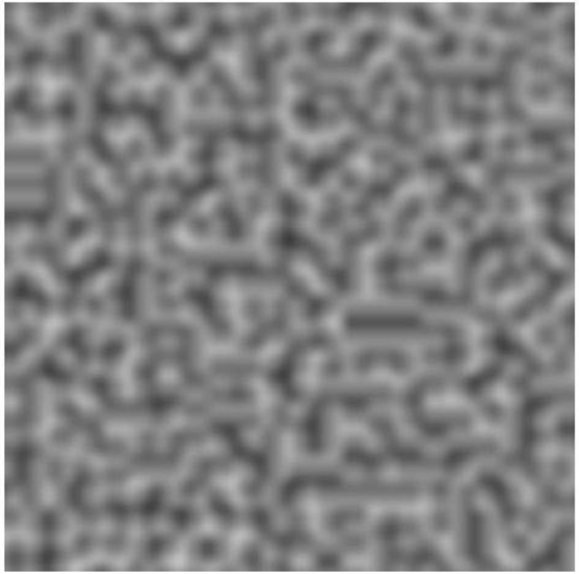


Figure 2b: High-frequency Noise, with new gradient distributions

SIGGRAPH 2002

Improved Perlin Noise

```java
// 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,
   77,146,158,231,83,111,229,122,60,211,133,230,220,105,92,41,55,46,245,40,244,
   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,
   129,22,39,253, 19,98,108,110,79,113,224,232,178,185, 112,104,218,246,97,228,
   251,34,242,193,238,210,144,12,191,179,162,241, 81,51,145,235,249,14,239,107,
   49,192,214, 31,181,199,106,157,184, 84,204,176,115,121,50,45,127, 4,150,254,
   138,236,205,93,222,114,67,29,24,72,243,141,128,195,78,66,215,61,156,180
   };
   static { for (int i=0; i < 256 ; i++) p[256+i] = p[i] = permutation[i]; }
}
```

https://en.wikipedia.org/wiki/Simplex_noise

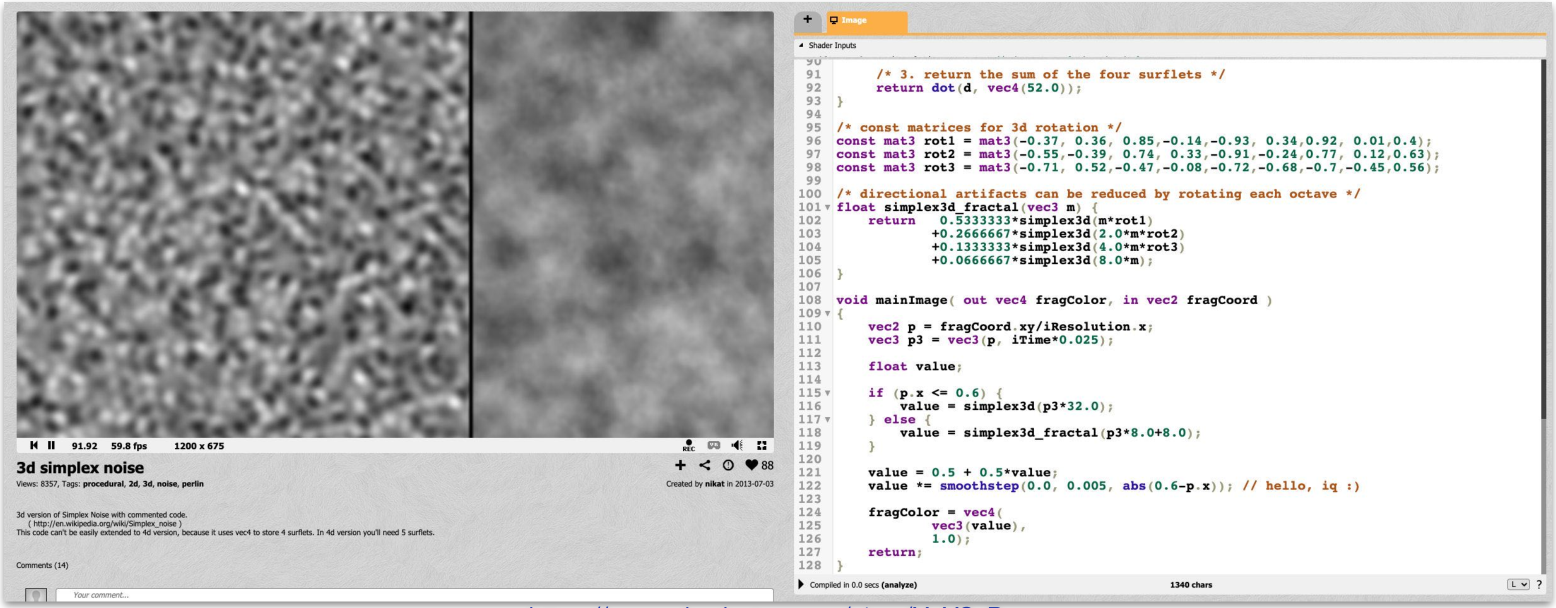
A Perlin noise extension

Simplex Noise (Perlin 1999)

Interpolation on tetrahedra instead of cubes.

Cheaper in higher dimensions: Only 4 vertex neighbors instead of 2^D in D-dim hypercube.

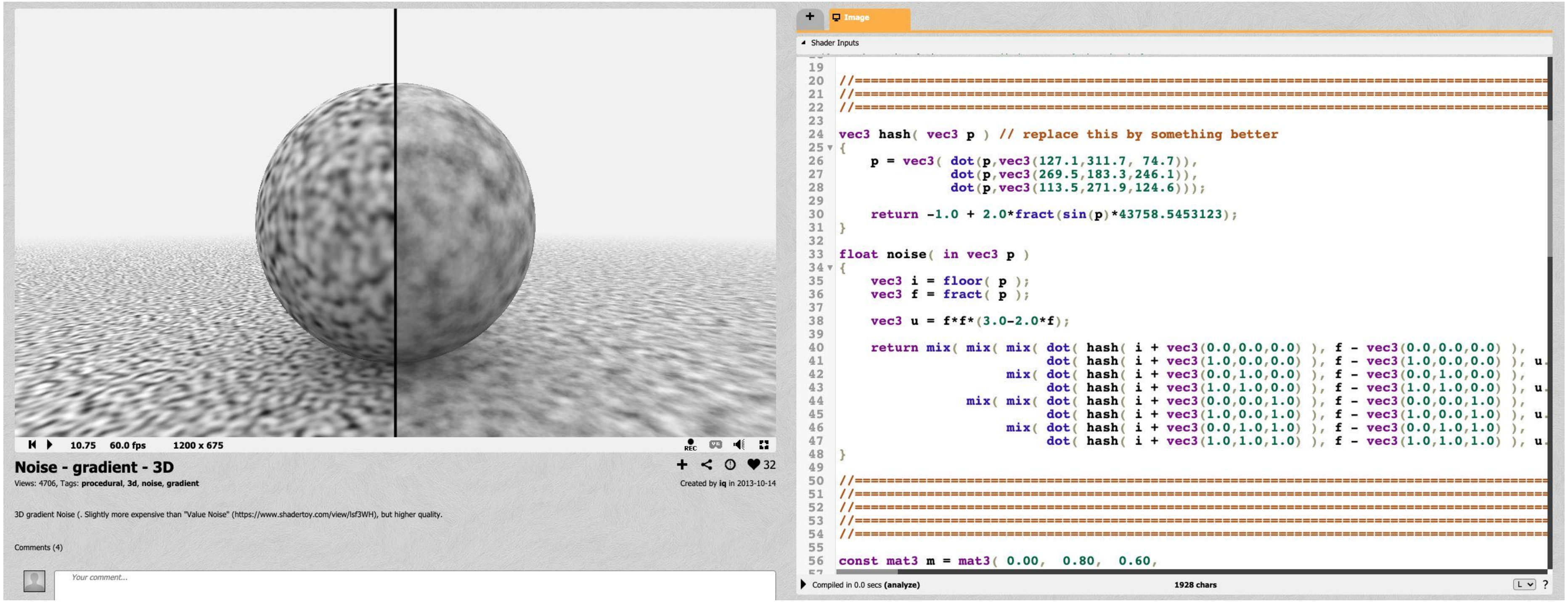
Fewer directional artifacts. Good for animation!



https://www.shadertoy.com/view/XsX3zB

Detail vs Aliasing

Problems with Gradient Noise



https://www.shadertoy.com/view/Xsl3Dl

http://graphics.pixar.com/library/WaveletNoise/

Pixar

Wavelet Noise (SIGGRAPH 2005)

Wavelet Noise

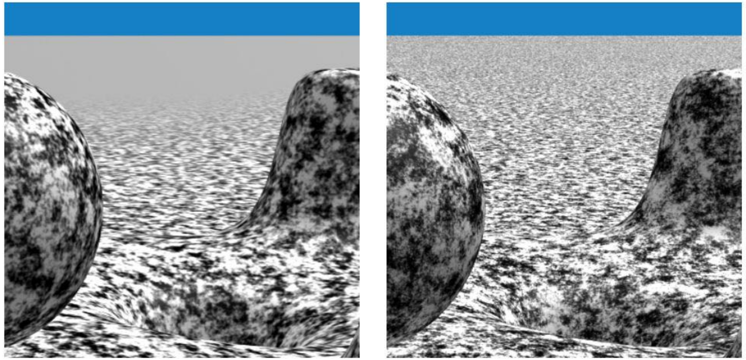
Robert L. Cook Tony DeRose
Pixar Animation Studios

Abstract

Noise functions are an essential building block for writing procedural shaders in 3D computer graphics. The original noise function introduced by Ken Perlin is still the most popular because it is simple and fast, and many spectacular images have been made with it. Nevertheless, it is prone to problems with aliasing and detail loss. In this paper we analyze these problems and show that they are particularly severe when 3D noise is used to texture a 2D surface. We use the theory of wavelets to create a new class of simple and fast noise functions that avoid these problems.

CR Categories: I.3.3 [Picture/Image generation]: Antialiasing— [I.3.7]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture

Keywords: Multiresolution analysis, noise, procedural textures, rendering, shading, texture synthesis, texturing, wavelets.



(a) (b)

Figure 1: *A comparison between images created using (a) Perlin noise and (b) wavelet noise. Image (a) represents best practices use of Perlin noise at Pixar to achieve the optimal tradeoff between detail and aliasing; notice how much detail is missing at high spatial frequencies in the far distance.*

http://graphics.pixar.com/library/WaveletNoise/

Pixar

Wavelet Noise

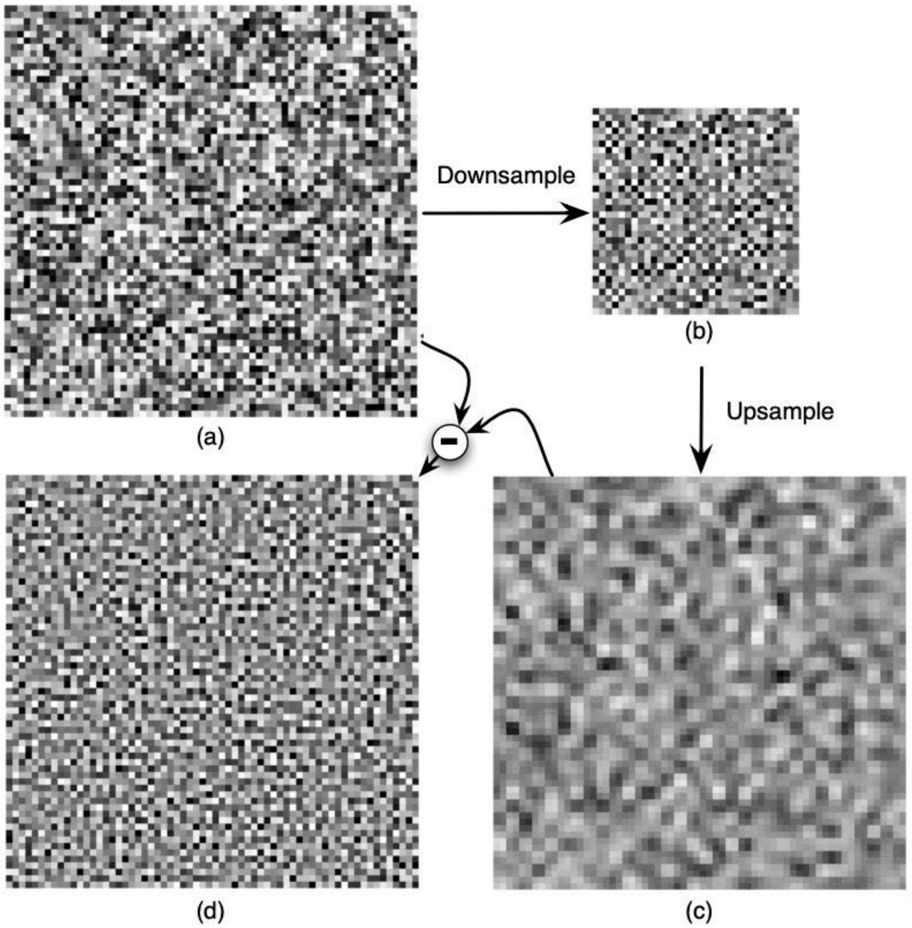


Figure 2: *(a). Image R of random noise, (b) Half-size image* $R^{\downarrow}$, *(c) Half-resolution image* $R^{\downarrow\uparrow}$, *(d) Noise band image* $N = R - R^{\downarrow\uparrow}$.

http://graphics.pixar.com/library/WaveletNoise/

Pixar
Wavelet Noise

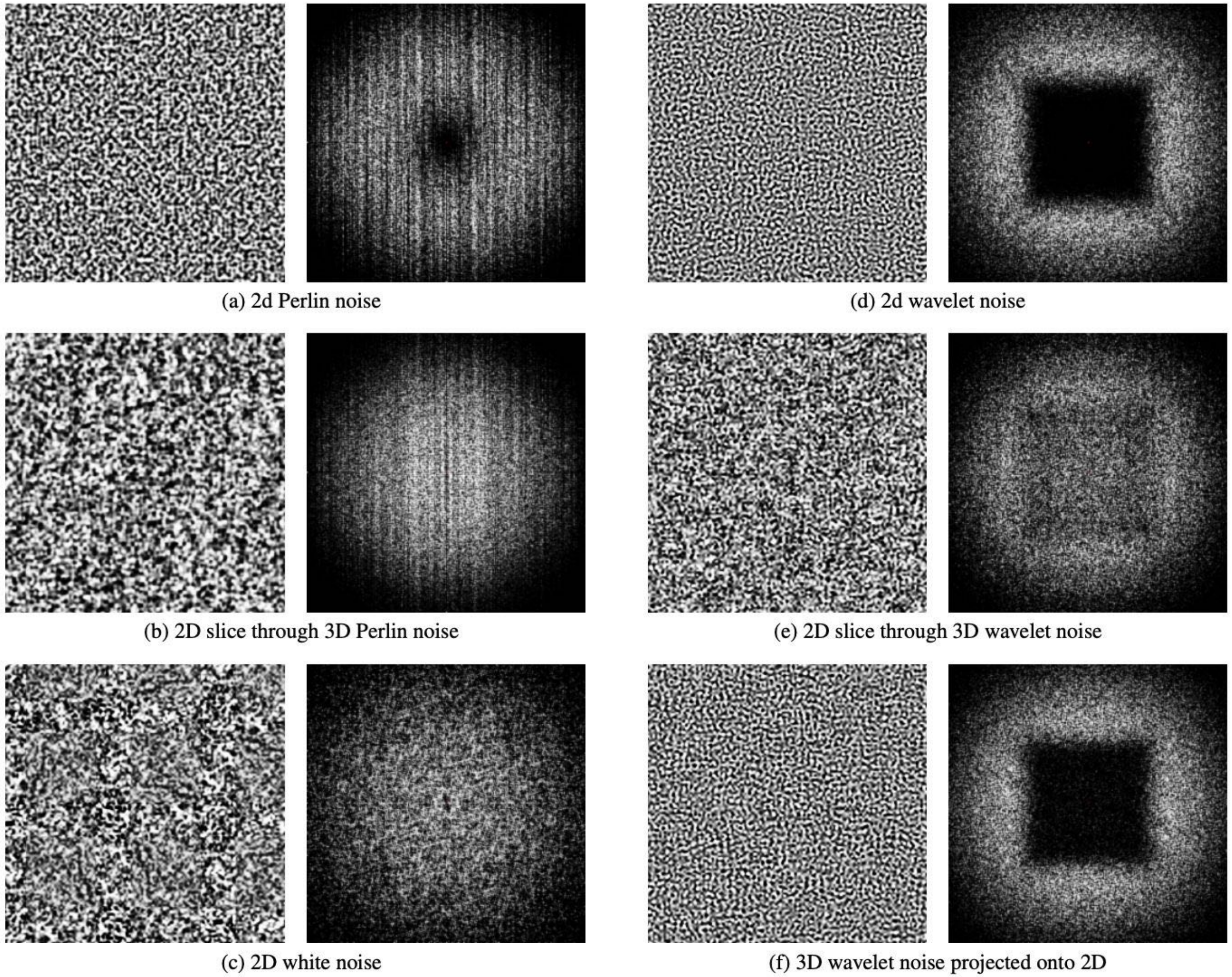


Figure 8: *Noise patterns (left) with their Fourier transforms (right). For Perlin noise, we use the RenderMan implementation of [Perlin 2002].*

http://graphics.pixar.com/library/WaveletNoise/

Pixar

Wavelet Noise - Rap

If you want to score some noise
Ken Perlin is your man.
Got the best funky noise
Anywhere in the land.

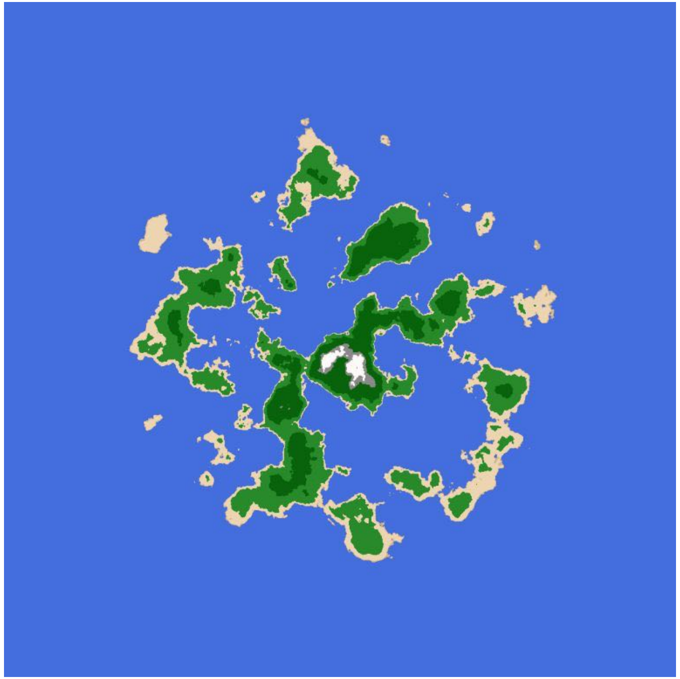
But its bands t really banded,
Which has caused a lot of grief.
But now those days are over
'Cause the wavelets bring relief.

Oh the wavelets they be simple,
And the wavelets they be quick,
And the wavelets they be better
'Cause the wavelet bands be slick.

Fun applications

Terrain Generation

Minecraft terrain

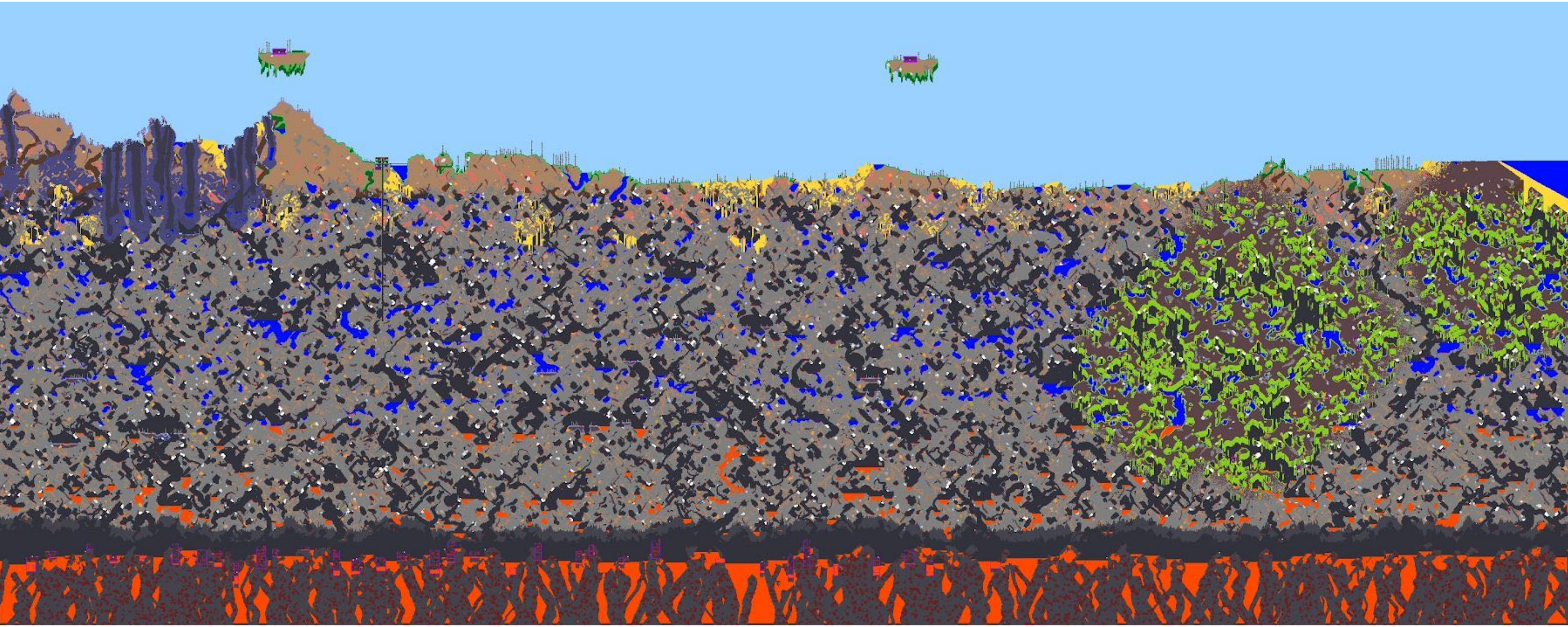


http://yvanscher.com/2017-11-07_Playing-with-Perlin-Noise--Generating-Realistic-Archipelagos-b59f004d8401.html

Terraria

Breakout Discussion

Terrain Generation in Terraria

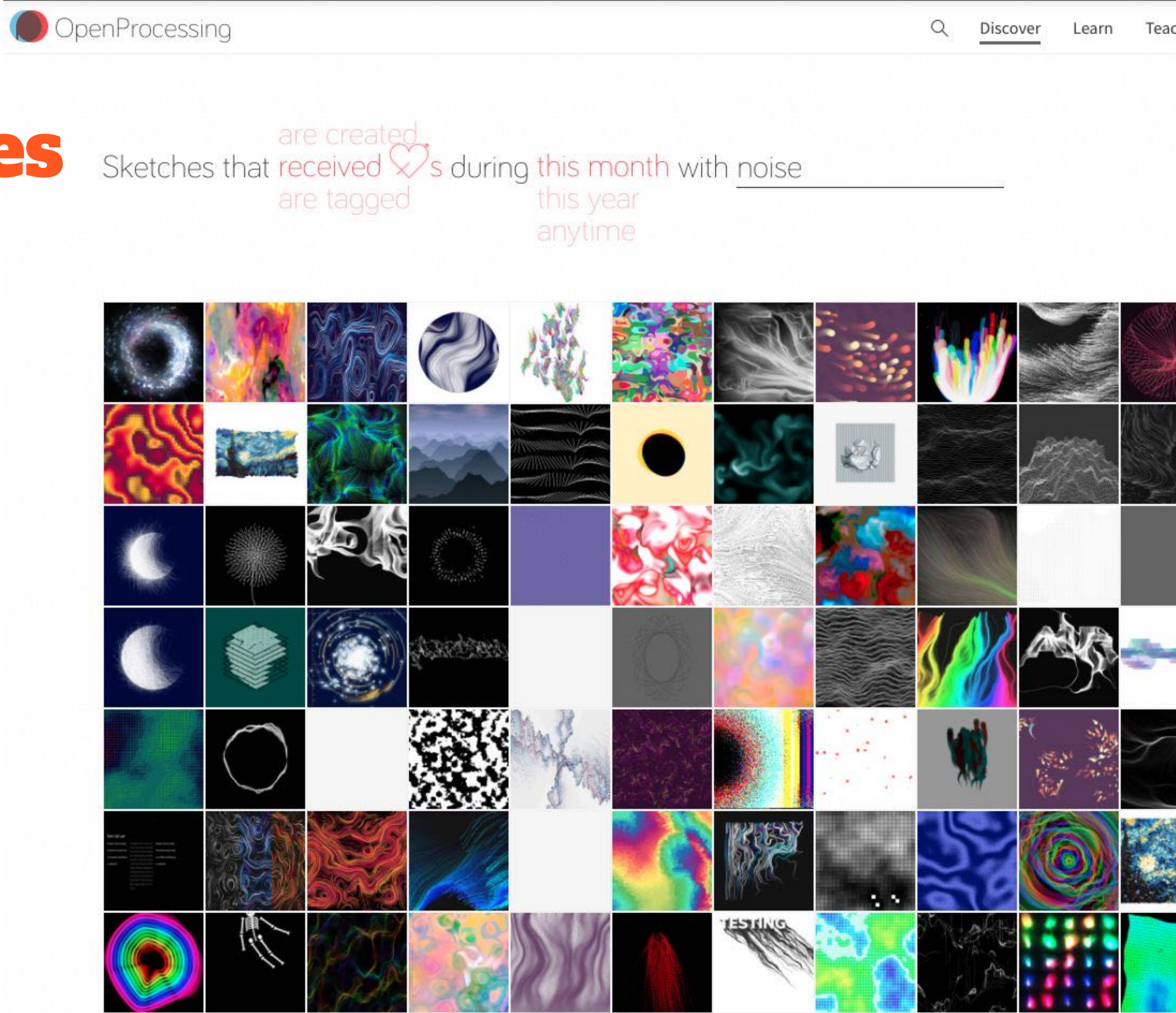


Terraria

https://terraria.fandom.com/wiki/Underground_Jungle

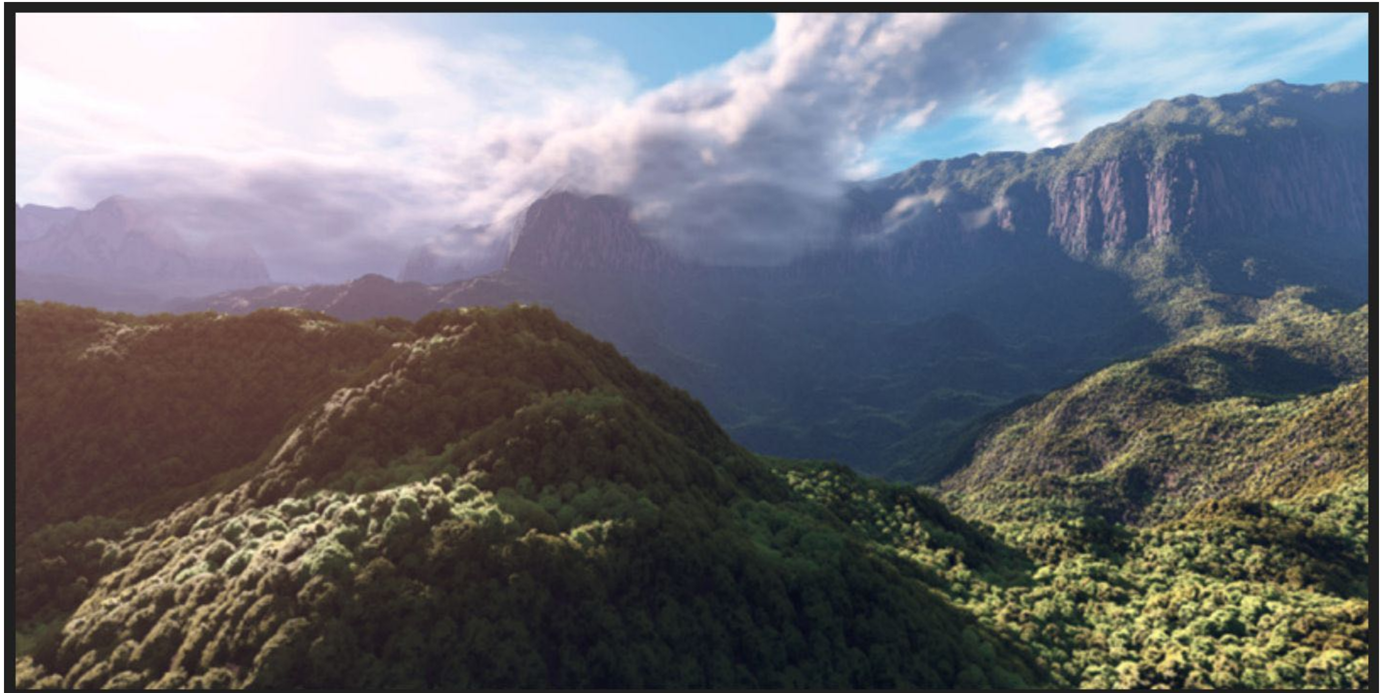
OpenProcessing

Noisy Sketches



Shadertoy

“Rainforest” by iq



fBM() was used to generate the terrain, the clouds, the tree distribution, their color variations, and the canopy details. "Rainforest", 2016: https://www.shadertoy.com/view/4ttSWf

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

Stanford CS 248

Interactive Computer Graphics

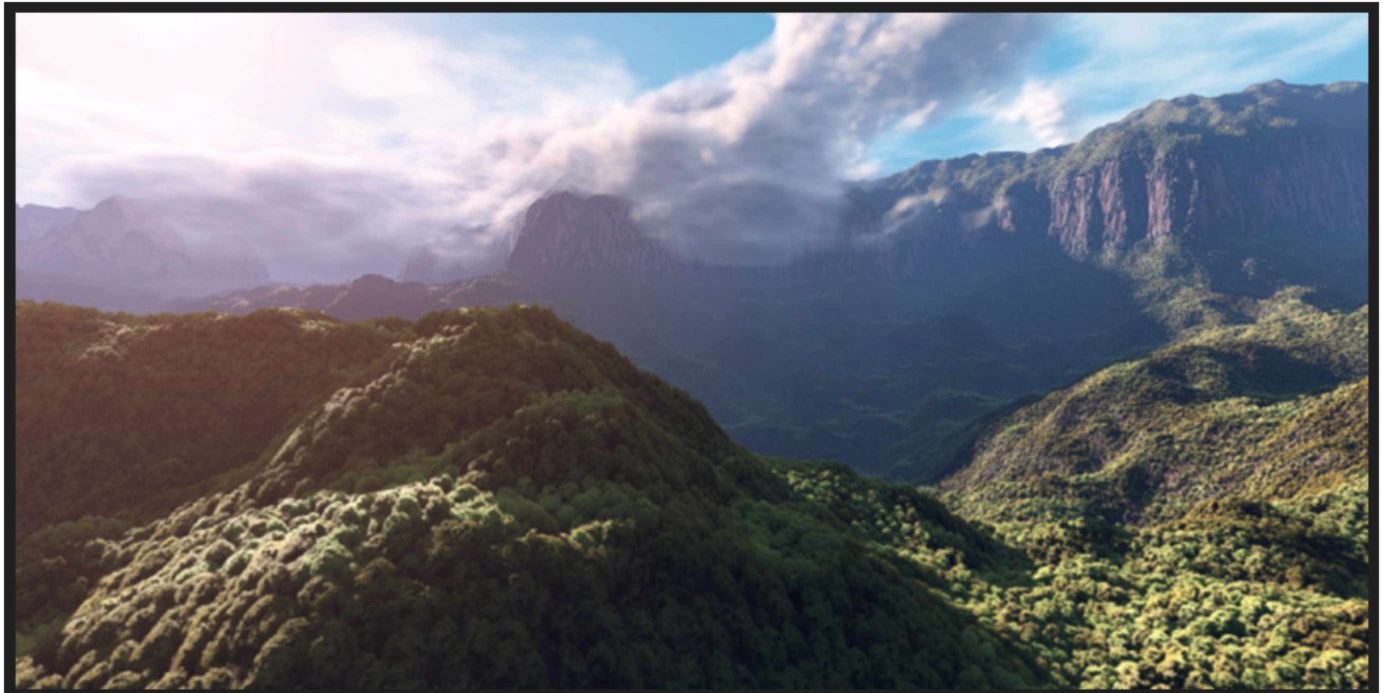
Ray Marching & Implicit Geometry

Wow, so much bonus material.

Ray Marching & Implicit Geometry

Shadertoy

“Rainforest” by iq (Inigo Quilez)

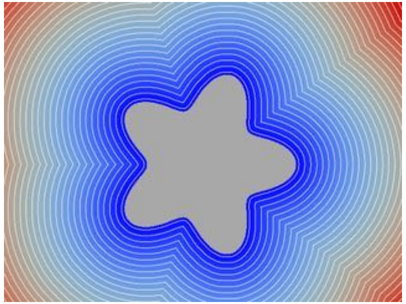


fBM() was used to generate the terrain, the clouds, the tree distribution, their color variations, and the canopy details. "Rainforest", 2016: https://www.shadertoy.com/view/4ttSWf

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

Ray-marching Implicit Functions

- Two great ideas in one:
  - Ray marching
  - Implicit modeling
- [“Sphere Tracing”] of signed distance fields (SDFs) [Hart 1995]
  - Common tool for rendering SDFs on ShaderToy



Distance field for a star shape

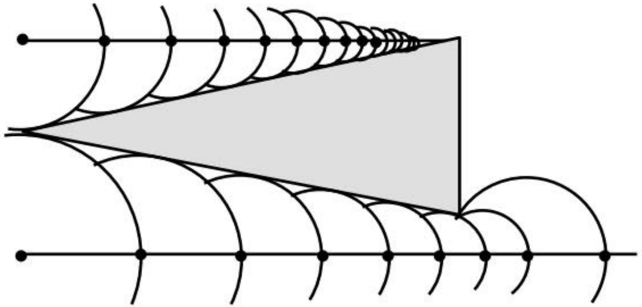
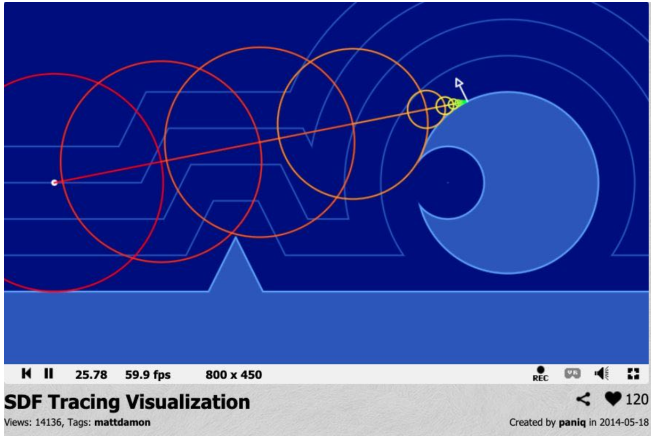


Figure 2: A hit and a miss.



https://www.shadertoy.com/view/lslXD8

Part I:

Implicit Geometry

Distance Fields

Distance Fields: Encode distance from a **point**, x, to an object, *O*:

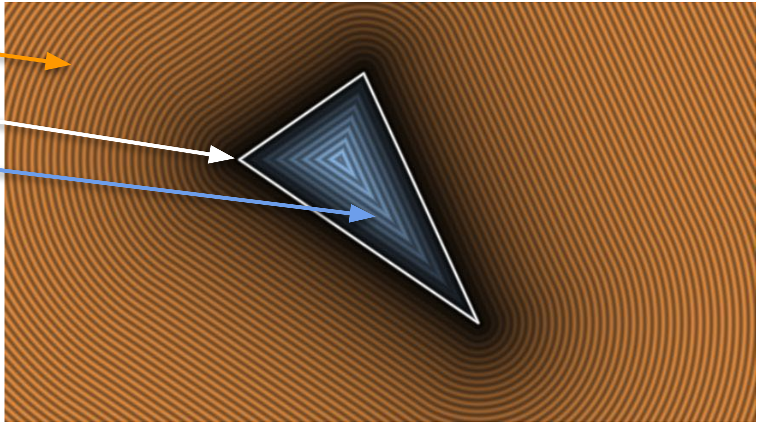
$d(x) = \min \|x-y\|_2$ over all y on *O*.

Signed Distance Fields: Sign indicates if inside/outside object:

$d(x) > 0$: Outside object

$d(x) = 0$: On object

$d(x) < 0$: Inside object



Gradient of d(x) is direction to closest surface.

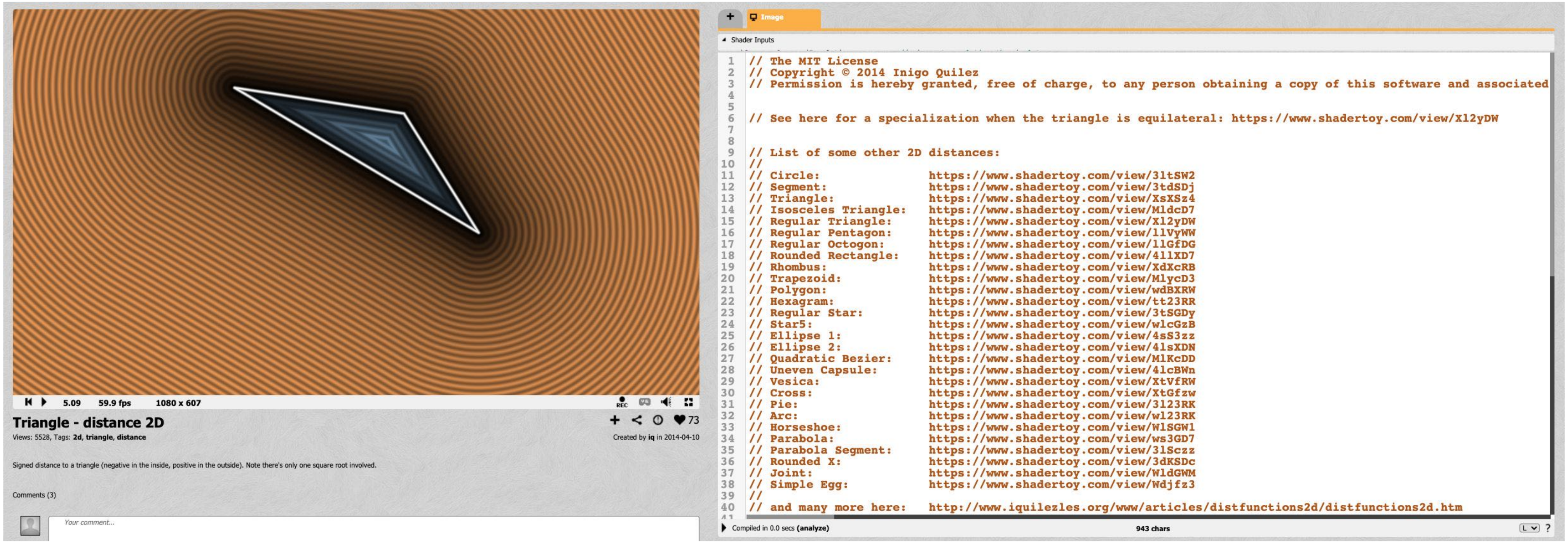
Useful for surface normals (near d=0 isosurface)

Analytical formulas exist only for simple shapes,

E.g., https://www.shadertoy.com/view/XsXSz4

Recall

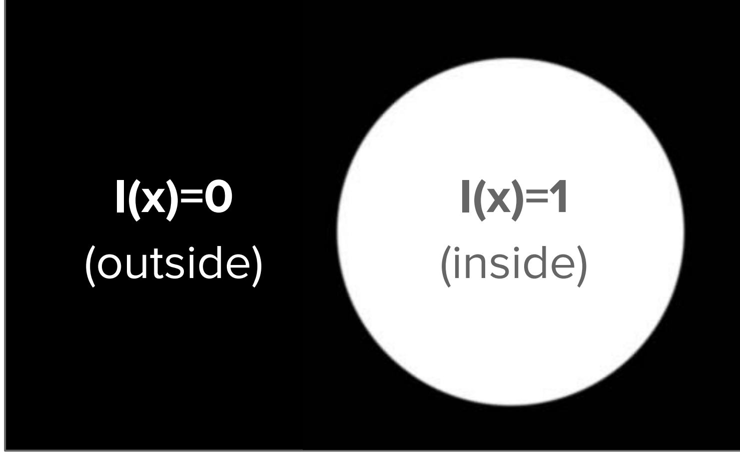
2D Distance Fields



https://www.shadertoy.com/view/XsXSz4

Where is the object?

Indicator Functions, I(x)

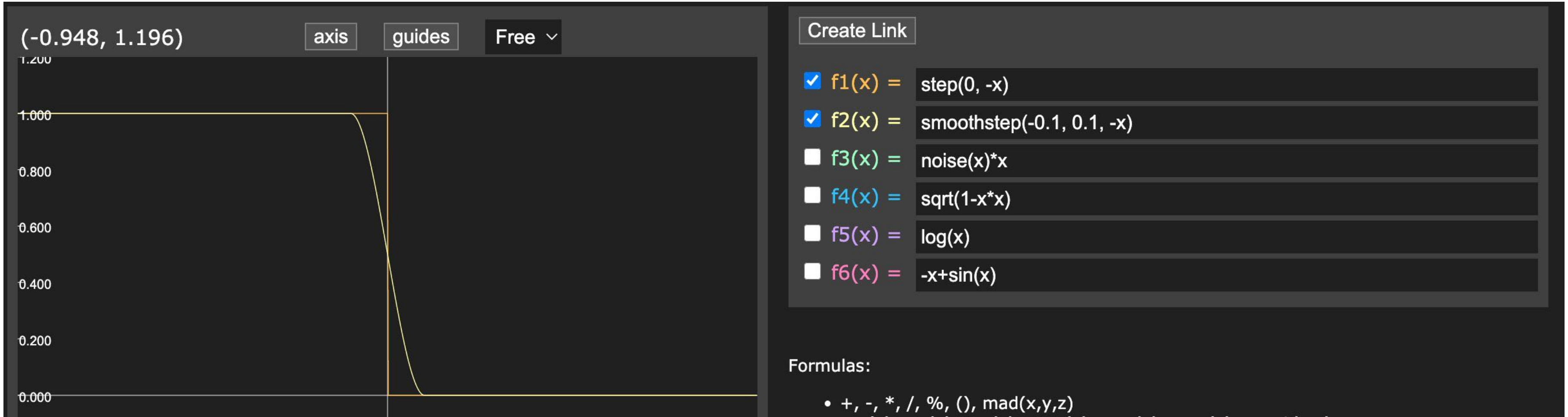


Step indicator function: I(x) = `step(0,-d(x))`

- 1 inside, d<=0
- 0 outside, d>0

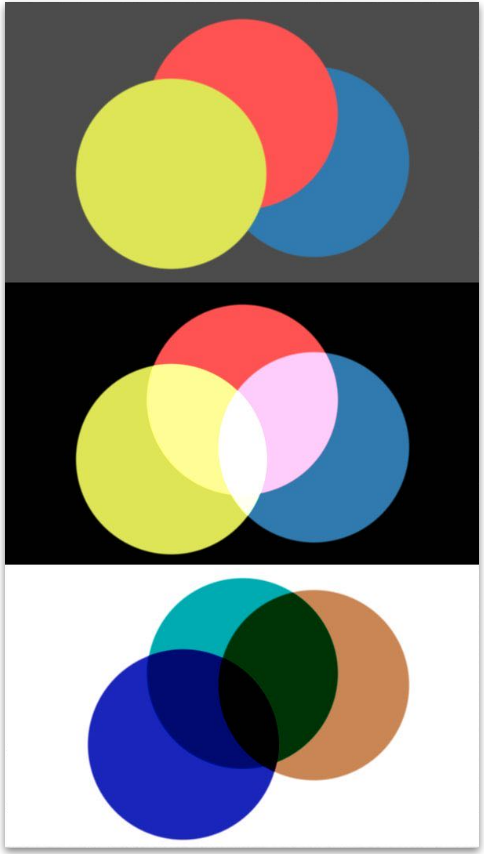
Smooth-step indicator function: `smoothstep(-eps,eps, -d)`

http://www.iquilezles.org/apps/graphtoy/?f1(x)=step(0,%20-x)&f2(x)=smoothstep(-0.1,%200.1,%20-x)



Indicator function application

Color Mixing



"On top":

$$\text{color}(x) \leftarrow (1 - I_i(x))\,\text{color}(x) + I_i(x)\ \text{color}_i$$

Additive blending:

$$\text{color}(\mathbf{x}) = \sum_i I_i(\mathbf{x})\ \text{color}_i$$

Subtracting blending:

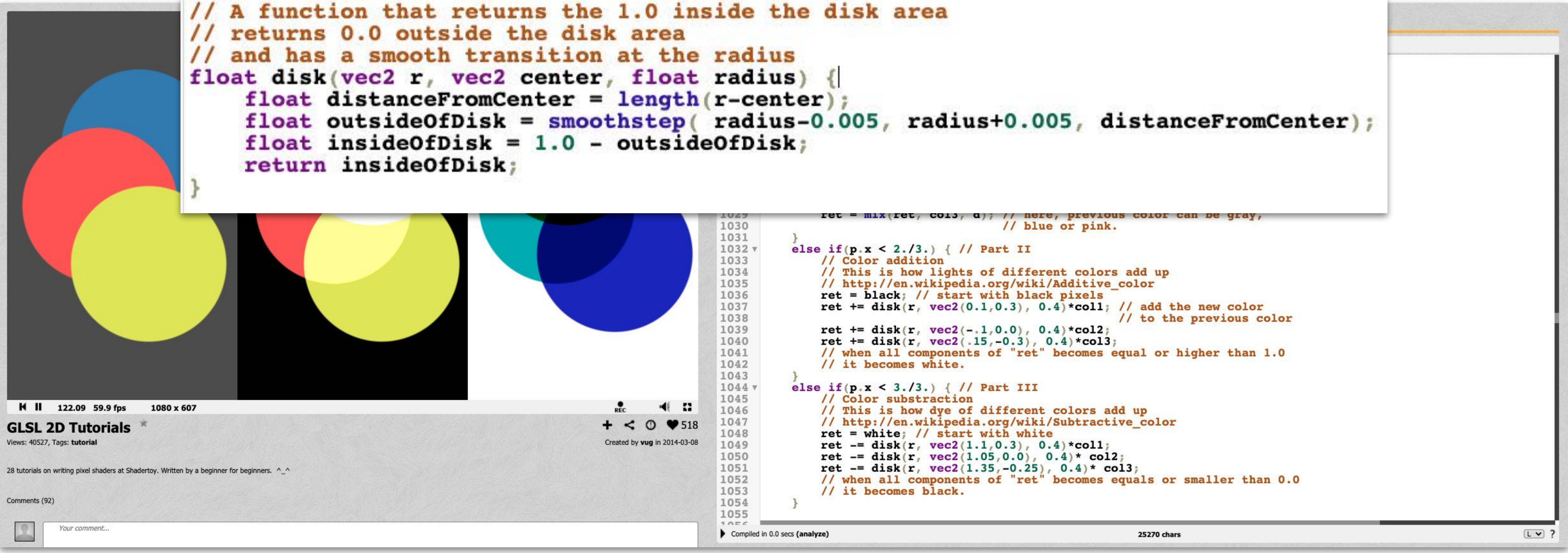
$$\text{color}(\mathbf{x}) = \text{white} - \sum_i I_i(\mathbf{x})\ \text{color}_i$$

Recall: Intro to GLSL Demo #20

2D Indicator Functions

Smooth-step indicator function, I(r): `disk(r, center, radius)`

```
// A function that returns the 1.0 inside the disk area
// returns 0.0 outside the disk area
// and has a smooth transition at the radius
float disk(vec2 r, vec2 center, float radius) {
    float distanceFromCenter = length(r-center);
    float outsideOfDisk = smoothstep( radius-0.005, radius+0.005, distanceFromCenter);
    float insideOfDisk = 1.0 - outsideOfDisk;
    return insideOfDisk;
}
```



https://www.shadertoy.com/view/Md23DV

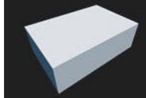
Analytical SDFs

3D Distance Field Primitives

Sphere - exact

```
float sdSphere( vec3 p, float s )
{
  return length(p)-s;
}
```

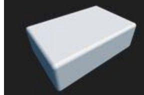
Box - exact



Youtube Tutorial on formula derivation: https://www.youtube.com/watch?v=62-pRVZuS5c

```
float sdBox( vec3 p, vec3 b )
{
  vec3 q = abs(p) - b;
  return length(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0);
}
```

Round Box - exact



```
float sdRoundBox( vec3 p, vec3 b, float r )
{
  vec3 q = abs(p) - b;
  return length(max(q,0.0)) + min(max(q.x,max(q.y,q.z)),0.0) - r;
}
```

Bounding Box - exact

```
float sdBoundingBox( vec3 p, vec3 b, float e )
{
       p = abs(p  )-b;
  vec3 q = abs(p+e)-e;
  return min(min(
      length(max(vec3(p.x,q.y,q.z),0.0))+min(max(p.x,max(q.y,q.z)),0.0),
      length(max(vec3(q.x,p.y,q.z),0.0))+min(max(q.x,max(p.y,q.z)),0.0)),
      length(max(vec3(q.x,q.y,p.z),0.0))+min(max(q.x,max(q.y,p.z)),0.0));
}
```

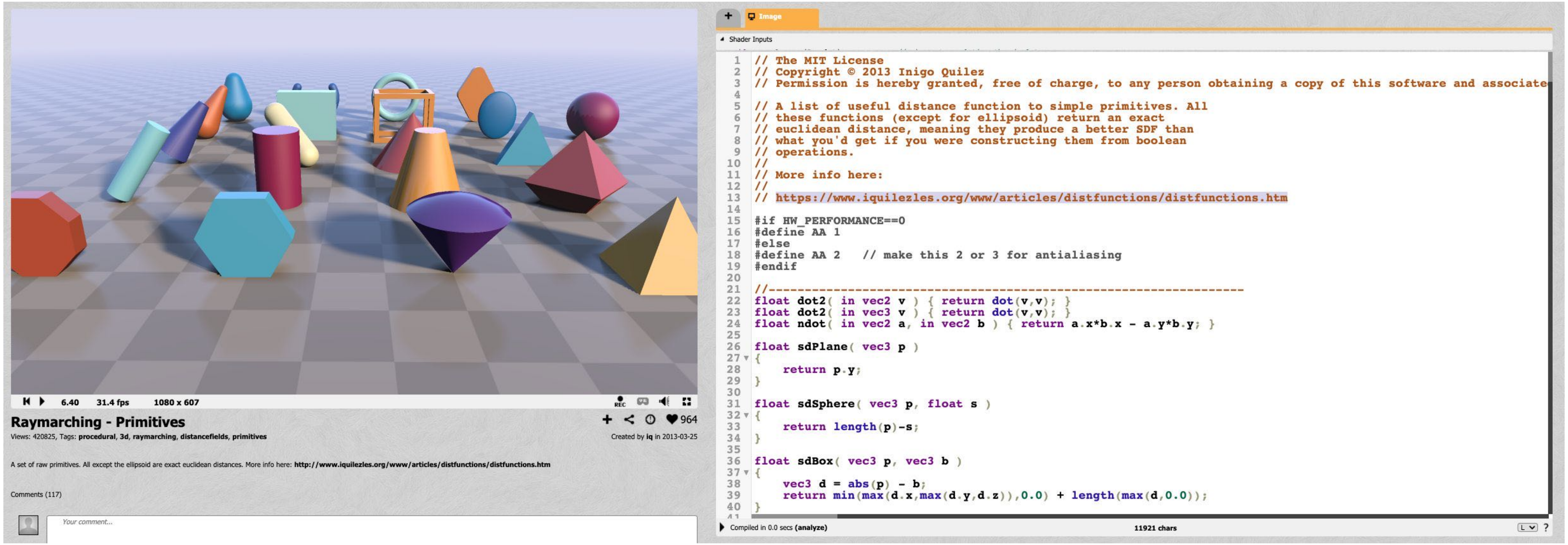
Torus - exact

```
float sdTorus( vec3 p, vec2 t )
{
  vec2 q = vec2(length(p.xz)-t.x,p.y);
  return length(q)-t.y;
}
```

https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

Ray Marching Demo

3D Distance Field Primitives



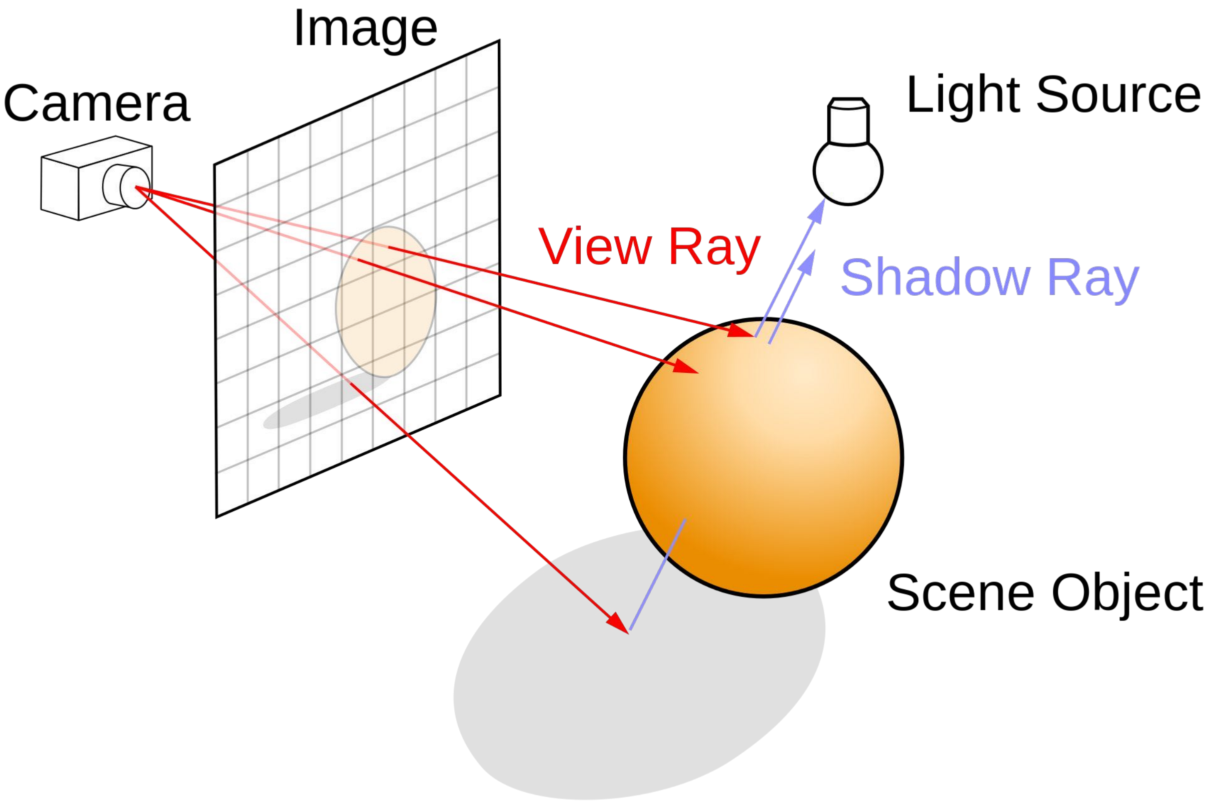
https://www.shadertoy.com/view/Xds3zN

Part II:

Ray Marching

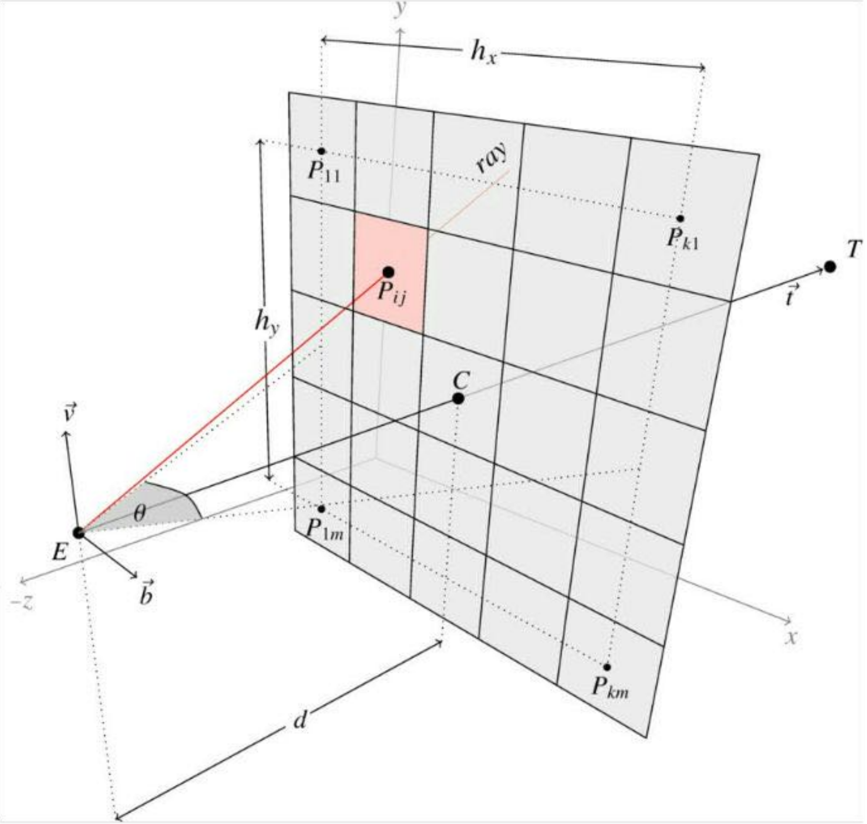
https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Ray Tracing



https://en.wikipedia.org/wiki/Ray_tracing_(graphics)

Shooting rays through camera pixels



Simplified notation

Shooting rays through camera pixels

Ray origin (eye): `ro`

Ray direction (unit vector):

`rd = normalize(pix-ro)`

Ray equation:

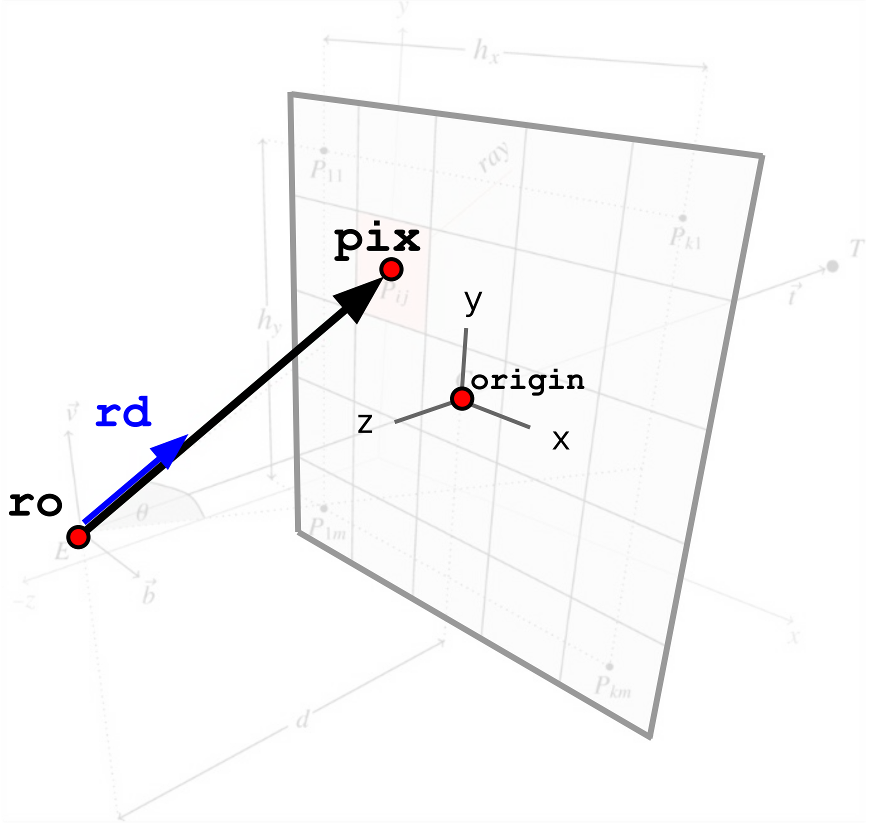
`r(t)= ro + rd*t, t≥0`

Goal:

Find t^* at ray-surface intersection;

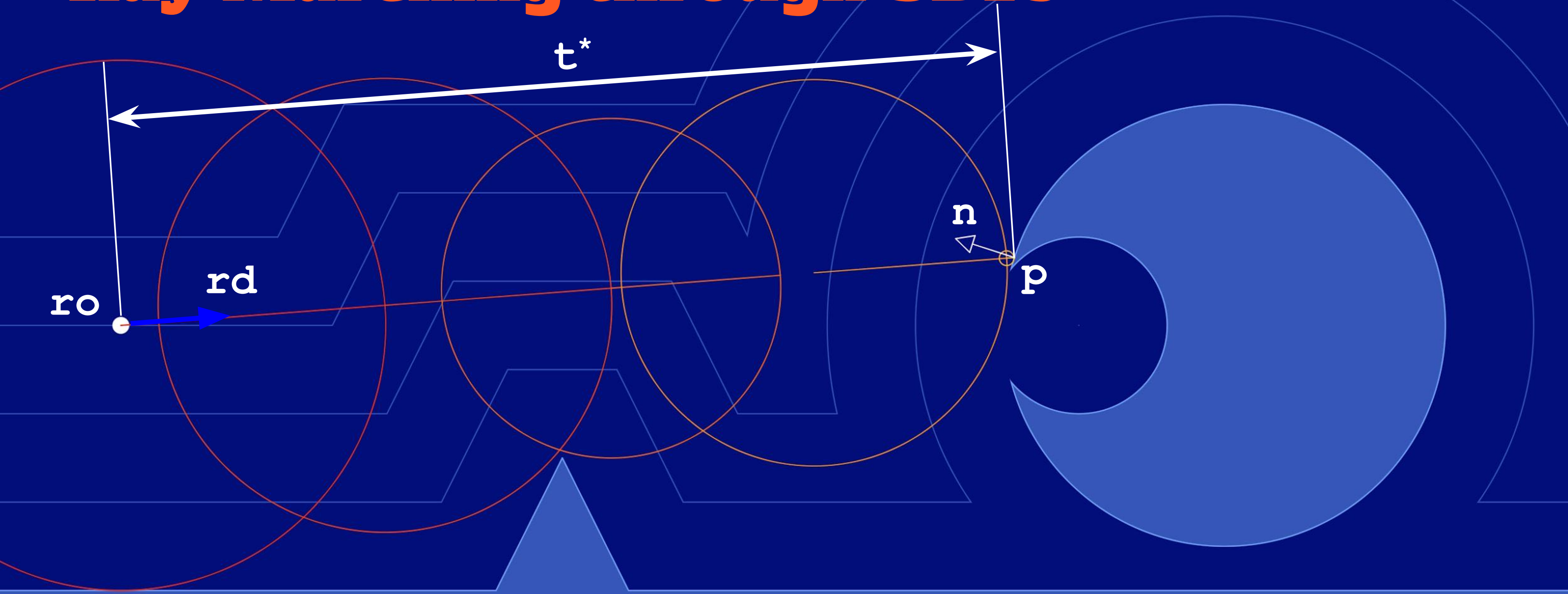
First t^* where `sdf(r(t*))≈0`

Intersection point: `p = r(t*)`



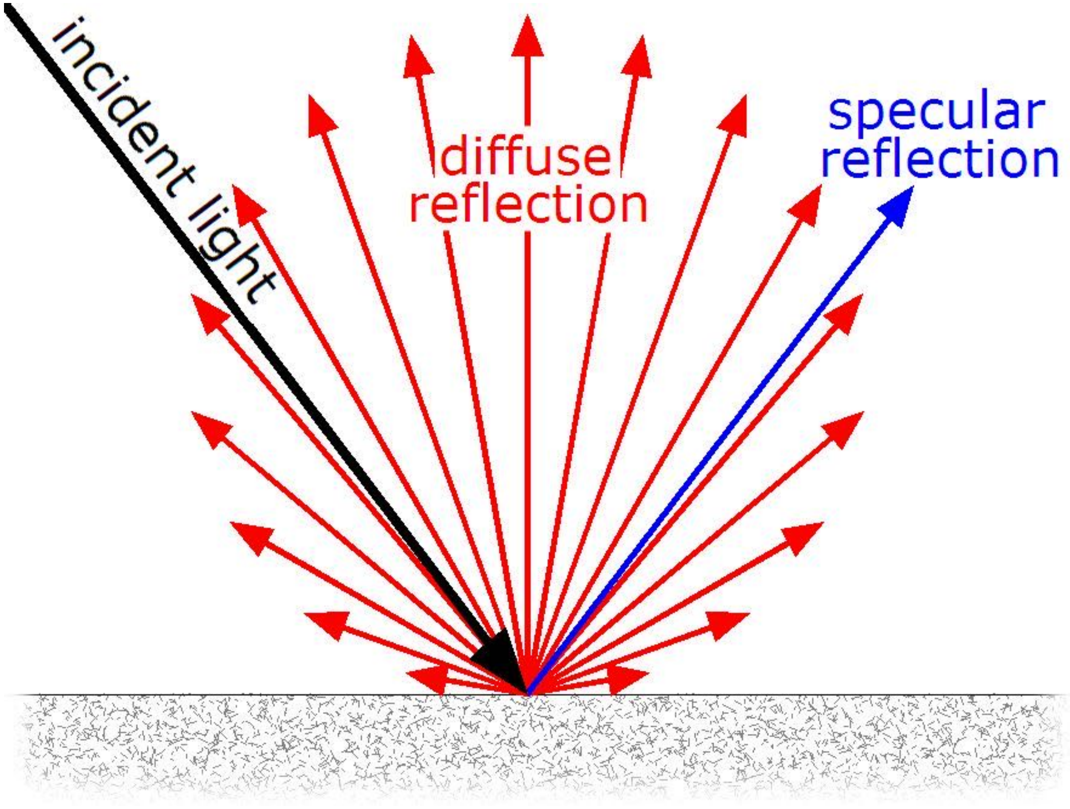
https://www.shadertoy.com/view/lslXD8

Ray Marching through SDFs



Compute normal **n** to SDF, f(**p**), using numerical approximation to $\nabla f(\mathbf{p})$.
See https://www.iquilezles.org/www/articles/normalsSDF/normalsSDF.htm

Surface Reflectance



https://en.wikipedia.org/wiki/Diffuse_reflection#/media/File:Lambert2.gif https://en.wikipedia.org/wiki/Lambertian_reflectance

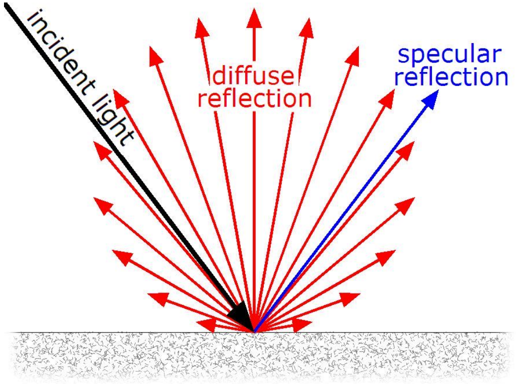
Simplest reflectance model

Diffuse Reflectance (Lambertian)

Models reflectance of a “matte” surface.

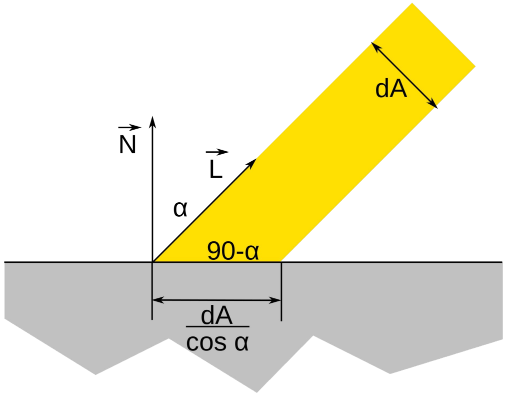
Appears the same from all view angles.

$$C_{out} = (\mathbf{L} \cdot \mathbf{N}) \; C_{diffuse} * I_{incident}$$



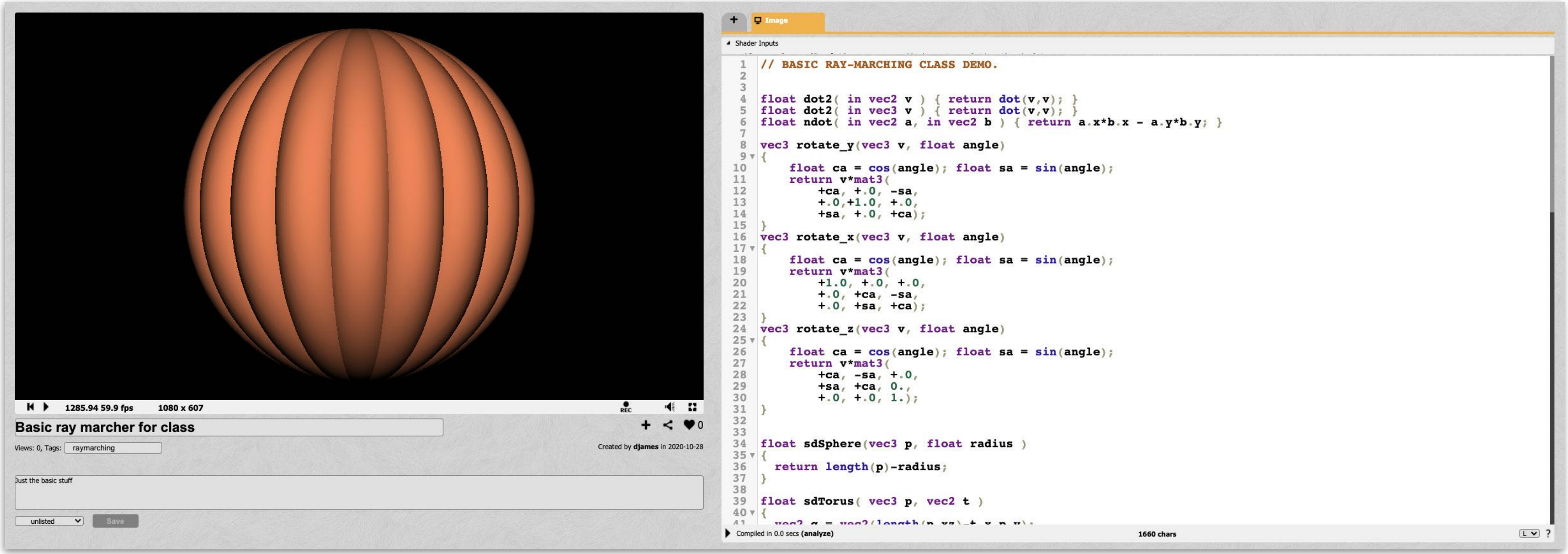
Cosine dependence on light direction,

$$\mathbf{L} \cdot \mathbf{N} = \cos \alpha$$



Ray Marching Together

Live Coding



https://www.shadertoy.com/view/wdKyD3

Part III:

More on Distance Fields

https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

Combining SDFs

Primitive Combinations

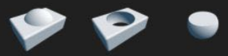
- Union
- Subtraction
- Intersection
- +Smooth versions

Primitive combinations

Sometimes you cannot simply elongate, round or onion a primitive, and you need to combine, carve or intersect basic primitives. Given the SDFs d1 and d2 of two primitives, you can use the following operators to combine together.

Union, Subtraction, Intersection - exact/bound, bound, bound

These are the most basic combinations of pairs of primitives you can do. They correspond to the basic boolean operations. **Please note** that only the Union of two SDFs returns a true SDF, not the Subtraction or Intersection. To make it more subtle, this is only true in the exterior of the SDF (where distances are positive) and not in the interior. You can learn more about this and how to work around it in the article "Interior Distances". Also note that opSubtraction() is not commutative and depending on the order of the operand it will produce different results.



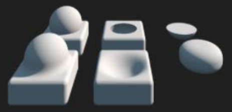
```
float opUnion( float d1, float d2 ) {  min(d1,d2); }

float opSubtraction( float d1, float d2 ) { return max(-d1,d2); }

float opIntersection( float d1, float d2 ) { return max(d1,d2); }
```

Smooth Union, Subtraction and Intersection - bound, bound, bound

Blending primitives is a really powerful tool - it allows to construct complex and organic shapes without the geometrical semas that normal boolean operations produce. There are many flavors of such operations, but the basic ones try to replace the min() and max() functions used in the opUnion, opSubstraction and opIntersection above with smooth versions. They all accept an extra parameter called **k** that defines the size of the smooth transition between the two primitives. It is given in actual distance units. You can find more details in the smooth minimum article article in this same site. You can code here: https://www.shadertoy.com/view/lt3BW2



```
float opSmoothUnion( float d1, float d2, float k ) {
    float h = clamp( 0.5 + 0.5*(d2-d1)/k, 0.0, 1.0 );
    return mix( d2, d1, h ) - k*h*(1.0-h); }

float opSmoothSubtraction( float d1, float d2, float k ) {
    float h = clamp( 0.5 - 0.5*(d2+d1)/k, 0.0, 1.0 );
    return mix( d2, -d1, h ) + k*h*(1.0-h); }

float opSmoothIntersection( float d1, float d2, float k ) {
    float h = clamp( 0.5 - 0.5*(d2-d1)/k, 0.0, 1.0 );
    return mix( d2, d1, h ) + k*h*(1.0-h); }
```

https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

Modifying SDFs

Primitive Alterations

- Elongation
- Rounding
- Onion
- Revolution and extrusion from 2D
- Change of Metric - bound

Primitive alterations

Once we have the basic primitives, it's possible to apply some simple operations that change their shape while still retaining exact an euclidean metric to them, which is an important property since SDFs with undistorted euclidean metric allow for faster ray marchine.

Elongation - exact

Elongating is a useful way to construct new shapes. It basically splits a primitive in two (four or eight), moves the pieces apart and and connects them. It is a perfect distance preserving operation, it does not introduce any artifacts in the SDF. Some of the basic primitives above use this technique. For example, the Capsule is an elongated Sphere along an axis really. You can find code here: https://www.shadertoy.com/view/Ml3fWj

```
float opElongate( in sdf3d primitive, in vec3 p, in vec3 h )
{
    vec3 q = p - clamp( p, -h, h );
    return primitive( q );
}

float opElongate( in sdf3d primitive, in vec3 p, in vec3 h )
{
    vec3 q = abs(p)-h;
    return primitive( max(q,0.0) ) + min(max(q.x,max(q.y,q.z)),0.0);
}
```

The reason I provide to implementations is the following. For 1D elongations, the first function works perfectly and gives exact exterior and interior distances. However, the first implementation produces a small core of zero distances inside the volume for 2D and 3D elongations. Depending on your application that might be a problem. One way to create exact interior distances all the way to the very elongated core of the volume, is the following, which is in languages like GLSL that don't have function pointers or lambdas need to be implemented a bit differently (check the code linked about in Shadertoy to see one example).

Rounding - exact

Rounding a shape is as simple as subtracting some distance (jumping to a different isosurface). The rounded box above is an example, but you can apply it to cones, hexagons or any other shape like the cone in the image below. If you happen to be interested in preserving the overall volume of the shape, most of the times it's pretty easy to shrink the source primitive by the same amount we are rounding it by. You can find code here: https://www.shadertoy.com/view/Mt3BDj

```
float opRound( in sdf3d primitive, float rad )
{
    return primitive(p) - rad
}
```

Onion - exact

For carving interiors or giving thickness to primitives, without performing expensive boolean operations (see below) and without distorting the distance field into a bound, one can use "onioning". You can use it multiple times to create concentric layers in your SDF. You can find code here: https://www.shadertoy.com/view/MlcBDj

https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

Modifying SDFs

Primitive Transformations

- Rotation/Translation
- Scale
- Symmetry
- Infinite Repetition
- Finite Repetition

Positioning

Placing primitives in different locations and orientations in space is a fundamental operation in designing SDFs. While rotations, uniform scaling and translations are exact operations, non-uniform scaling distorts the euclidean spaces and can only be bound. Therefore I do not include it here.

Rotation/Translation - exact

Since rotations and translation don't compress nor dilate space, all we need to do is simply to transform the point being sampled with the inverse of the transformation used to place an object in the scene. This code below assumes that transform encodes only a rotation and a translation (as a 3x4 matrix for example, or as a quaternion and a vector), and that it does not contain any scaling factors in it.

```
vec3 opTx( in vec3 p, in transform t, in sdf3d primitive )
{
    return primitive( invert(t)*p );
}
```

Scale - exact

Scaling an obect is slightly more tricky since that compresses/dilates spaces, so we have to take that into account on the resulting distance estimation. Still, it's not difficult to perform:

```
float opScale( in vec3 p, in float s, in sdf3d primitive )
{
    return primitive(p/s)*s;
}
```

Symmetry - bound and exact

Symmetry is useful, since many things around us are symmetric, from humans, animals, vehicles, instruments, furniture, ... Oftentimes, one can take shortcuts and only model half or a quarter of the desired shape, and get it duplicated automatically by using the absolute value of the domain coordinates before evaluation. For example, in the image below, there's a single object evaluation instead of two. This is a great savings in performance. You have to be aware however that the resulting SDF might not be an exact SDF but a bound, if the object you are mirroring crosses the mirroring plane.

```
float opSymX( in vec3 p, in sdf3d primitive )
{
    p.x = abs(p.x);
    return primitive(p);
}

float opSymXZ( in vec3 p, in sdf3d primitive )
{
    p.xz = abs(p.xz);
    return primitive(p);
}
```

https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

Modifying SDFs

Primitive Deformations and Distortions

- Displacement
- Twist
- Bend

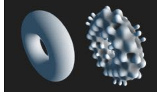
Deformations and distortions

Deformations and distortions allow to enhance the shape of primitives or even fuse different primitives together. The operations usually distort the distance field and make it non euclidean anymore, so one must be careful when raymarching them, you will probably need to decrease your step size, if you are using a raymarcher to sample this. In principle one can compute the factor by which the step size needs to be reduced (inversely proportional to the compression of the space, which is given by the Jacobian of the deformation function). But even with dual numbers or automatic differentiation, it's usually just easier to find the constant by hand for a given primitive.

I'd say that while it is tempting to use a distortion or displacement to achieve a given shape, and I often use them myself of course, it is sometimes better to get as close to the desired shape with actual exact euclidean primitive operations (elongation, rounding, onioning, union) or tight bounded functions (intersection, subtraction) and then only apply as small of a distortion or displacement as possible. That way the field stays as close as possible to an actual distance field, and the raymarcher will be faster.

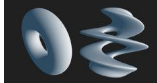
Displacement

The displacement example below is using sin(20*p.x)*sin(20*p.y)*sin(20*p.z) as displacement pattern, but you can of course use anything you might imagine.



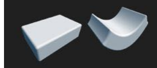
```
float opDisplace( in sdf3d primitive, in vec3 p )
{
    float d1 = primitive(p);
    float d2 = displacement(p);
    return d1+d2;
}
```

Twist



```
float opTwist( in sdf3d primitive, in vec3 p )
{
    const float k = 10.0; // or some other amount
    float c = cos(k*p.y);
    float s = sin(k*p.y);
    mat2  m = mat2(c,-s,s,c);
    vec3  q = vec3(m*p.xz,p.y);
    return primitive(q);
}
```

Bend



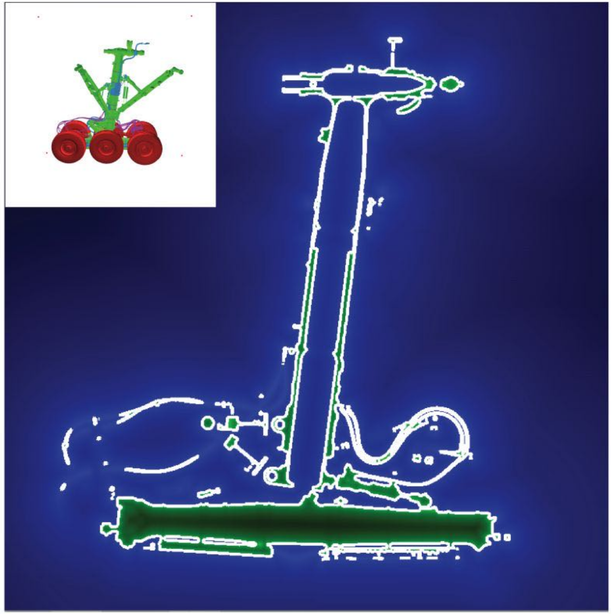
```
float opCheapBend( in sdf3d primitive, in vec3 p )
{
    const float k = 10.0; // or some other amount
    float c = cos(k*p.x);
    float s = sin(k*p.x);
    mat2  m = mat2(c,-s,s,c);
    vec3  q = vec3(m*p.xy,p.z);
    return primitive(q);
}
```

What if you don't know the analytical formula for the SDF?

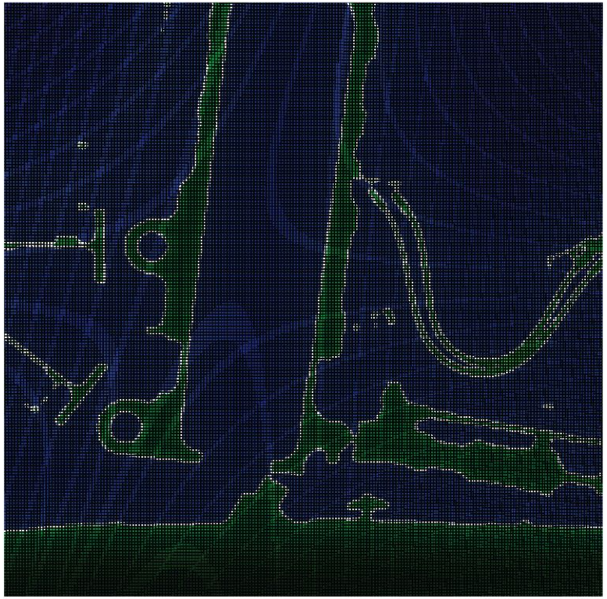
Numerical SDFs: Sampled on Grids

Distance Fields for Complex Shapes

Interpolate distances previously sampled on a regular (or adaptive) grid



(c) Landing gear 1024x1024x1024 signed distance field



(d) Detailed view of landing gear distance field

http://barbic.usc.edu/signedDistanceField/

Pros:

- Fast d(x) evaluation
- Complex shapes
- Parallel precompute

Cons:

- High memory overhead
  - Adaptive helps
- Precomputation
- Rigid geometry
  - Can't deform
- Only point-object distance

Industry standard

OpenVDB

Museth, K. 2013. **VDB: High-resolution sparse volumes with dynamic topology.** ACM Trans. Graph. 32, 3, Article 27 (June 2013) 22 pages. DOI: http://dx.doi.org/10.1145/2487228.2487235

http://www.museth.org/Ken/Publications_files/Museth_TOG13.pdf

https://www.openvdb.org/

Fig. 1. Top: Shot from the animated feature *Puss in Boots*, showing high-resolution animated clouds generated using VDB [Miller et al. 2012]. Left: The clouds are initially modelled as polygonal surfaces, then scan-converted into narrow-band level sets, after which procedural noise is applied to create the puffy volumetric look. Right: The final animated sparse volumes typically have bounding voxel resolutions of $15,000 \times 900 \times 500$ and are rendered using a proprietary renderer that exploits VDB's hierarchical tree structure. Images are courtesy of *DreamWorks Animation*.

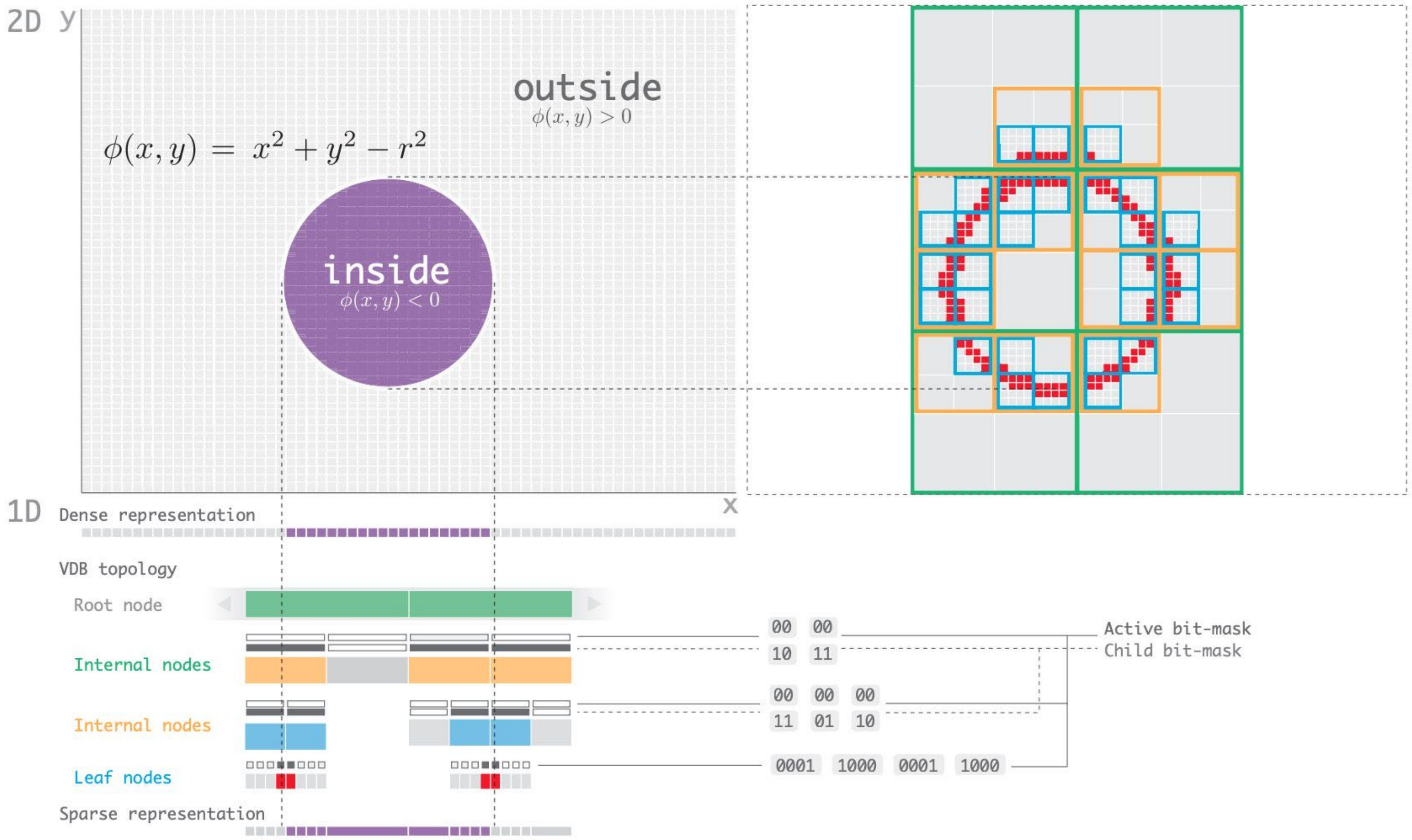


Fig. 3. Illustration of a narrow-band level set of a circle represented in, respectively, a 1D and 2D VDB. Top Left: The implicit signed distance, that is, level set, of a circle is discretized on a uniform dense grid. Bottom: Tree structure of a 1D VDB representing a single y-row of the narrow-band level set. Top Right: Illustration of the adaptive grid corresponding to a VDB representation of the 2D narrow-band level set. The tree structure of the 2D VDB is too big to be shown. Voxels correspond to the smallest squares, and tiles to the larger squares. The small branching factors at each level of the tree are chosen to avoid visual cluttering; in practice they are typically much larger.

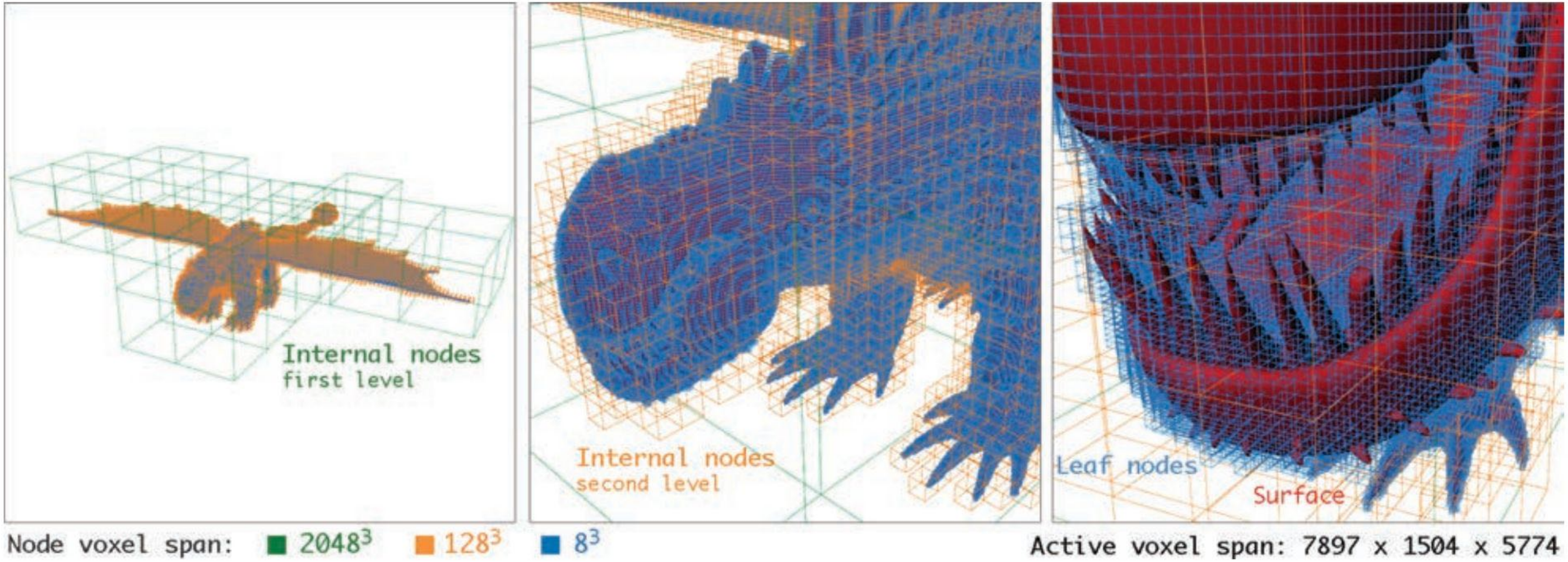


Fig. 4. High-resolution VDB created by converting polygonal model from *How To Train Your Dragon* to a narrow-band level set. The bounding resolution of the 228 million active voxels is $7897 \times 1504 \times 5774$ and the memory footprint of the VDB is 1GB, versus the $\frac{1}{4}$ TB for a corresponding dense volume. This VDB is configured with `LeafNodes` (blue) of size 8^3 and two levels of `InternalNodes` (green/orange) of size 16^3. The index extents of the various nodes are shown as colored wireframes, and a polygonal mesh representation of the zero level set is shaded red. Images are courtesy of *DreamWorks Animation*.

GPU-accelerated version of OpenVDB

nanoVDB

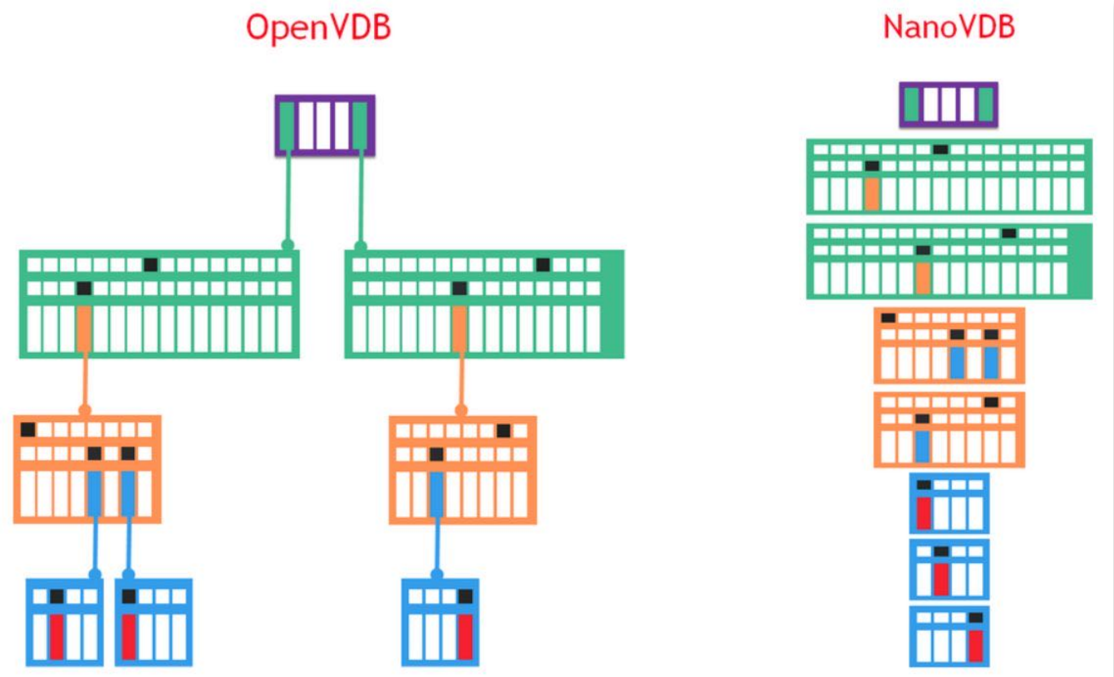
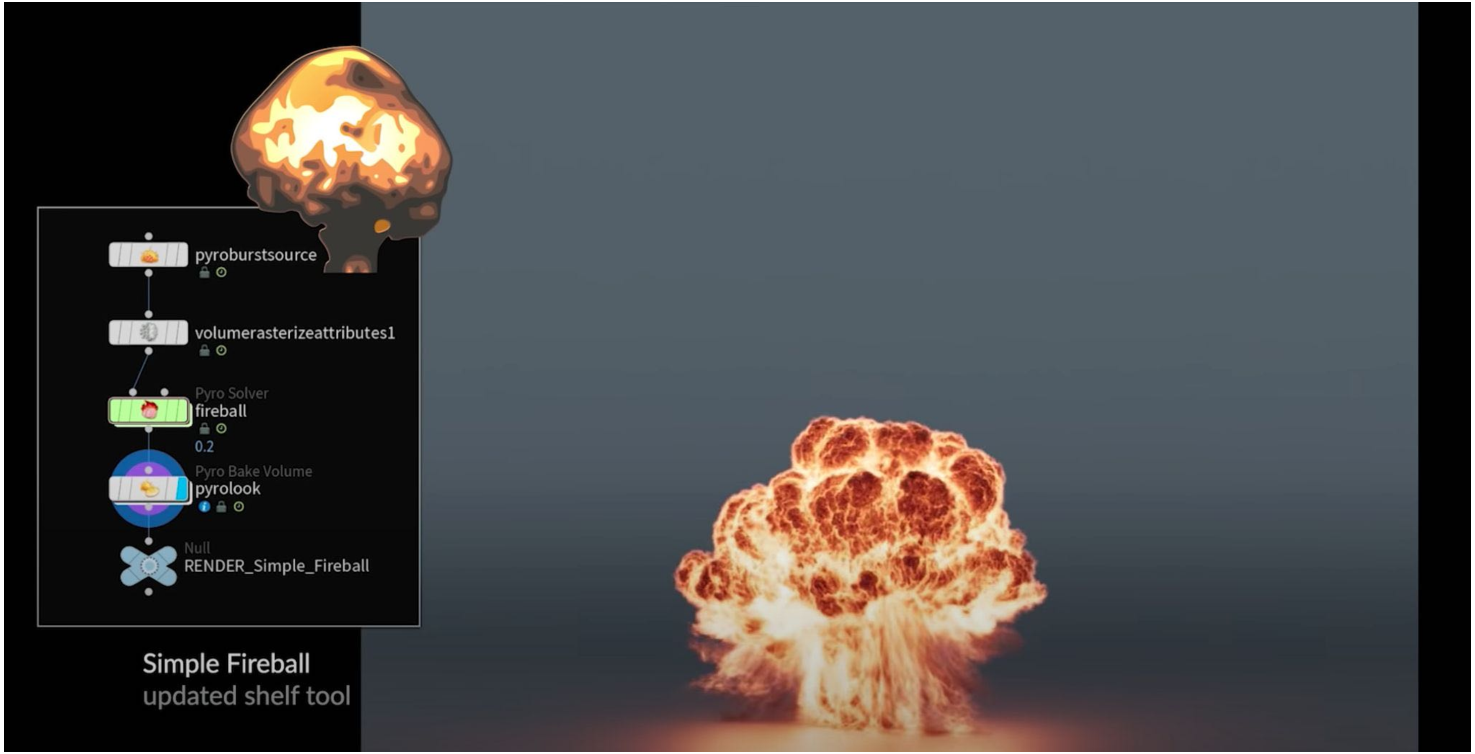


Figure 1. Illustration of the OpenVDB and NanoVDB data structures.

https://developer.nvidia.com/blog/accelerating-openvdb-on-gpus-with-nanovdb/

Example

Houdini - Uses VDB w/ acceleration



From “Houdini 18.5 Keynote”

Endgame

Two sketches/demos remaining

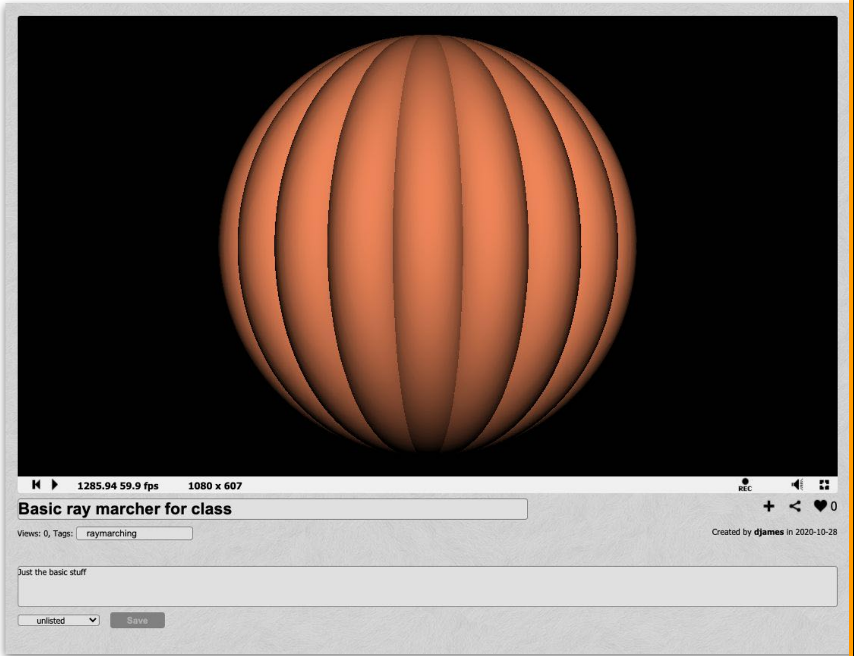
MonOct26	Ray Marching & Implicit Geometry	Explore real-time hardware rendering using implicit geometry.
WedOct28		
MonNov02		Due + Demo on MonNov09
WedNov04		
MonNov09	**MMO Games**	Build a simple multiplayer game using socket.io in OpenProcessing.
WedNov11		
MonNov16		Due + Demo on WedNov18
WedNov18		

Part IV:

Fin-i’-shing the Jack-o’-lantern

Ray Marching Together

Live Coding (Part II)



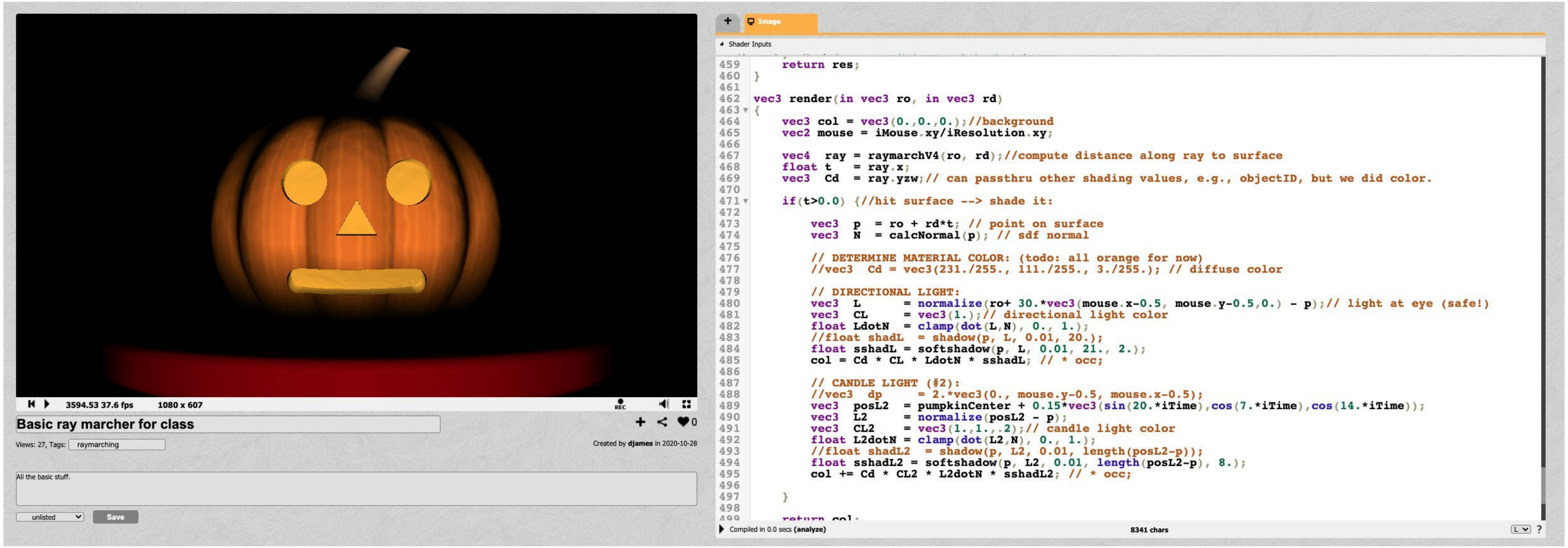
https://www.shadertoy.com/view/wdKyD3

Jack-o'-lantern Wishlist

- **Natural Shape**
  - Smoother blends (smin)
  - Bumpy (add noise)
  - Add stem (bumpy cone + bend + noise)
  - Less spherical & perfect (deform)
- **Carve it!**
  - Hollow out (onion or subtract)
  - Create/cut top (subtract onion cone)
  - Nose (subtract prism)
  - Eyes (subtract opSymX shape)
  - Mouth (subtract mouth shape)
- **Render with Spooky Lighting**
  - Flickering candle flame (point light)
  - Moonlight (directional light)
  - Shadows (shadow & softshadow) [link]
  - Ground (noisy plane or heightfield)
  - Background (ambient lighting)
  - Antialiasing (supersample)

Ray Marching Together

Live Coding (Final)



https://www.shadertoy.com/view/wdKyD3

Homework/Sketch

Ray-Marching SDFs

Modify the pumpkin renderer

https://www.shadertoy.com/view/wdKyD3

to render your own scene.

Note: You do not need to write any ray marching code of your own.

Modify the mapV4(p) function to compute a distance field and color of your scene.

Modify the lighting (light positions, etc.) as needed.

Again, the primary resource for modeling things with distance fields is iq's page:

https://iquilezles.org/www/articles/distfunctions/distfunctions.htm