

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.)

Noise

- Random numbers → Random functions
- General purpose image/shape synthesis

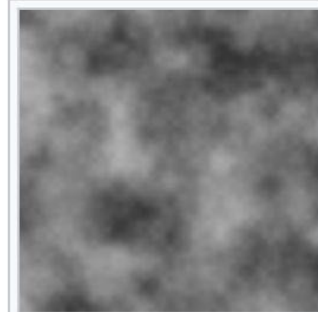
- Examples:

- Perlin noise
 - Perlin, Ken (July 1985). "An Image Synthesizer". *SIGGRAPH Comput. Graph.* **19** (97–8930): 287–296.
- Wavelet noise
- Worley noise
- Etc.

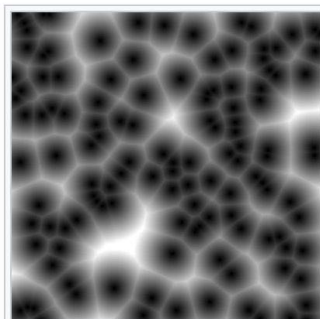
- Many applications!



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.

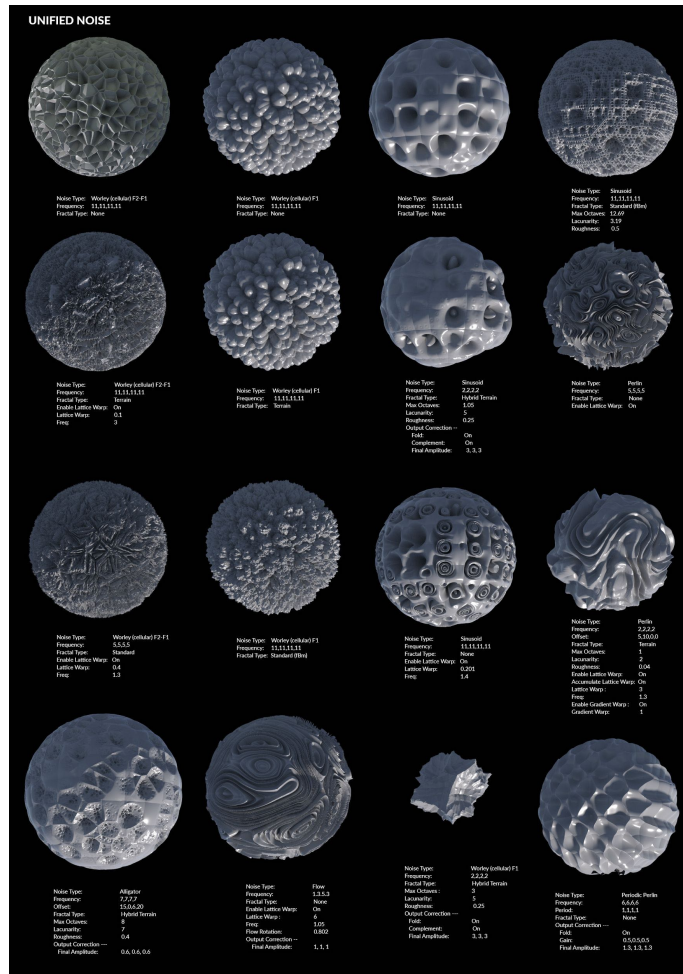


http://vianscher.com/2017-11-07_Playing-with-Perlin-Noise-Generating-Realistic-Archipelagos-358f00468401.html



An organic surface generated with Perlin noise

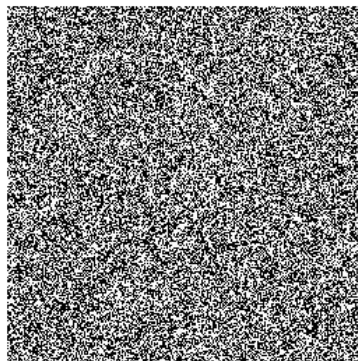
Noise



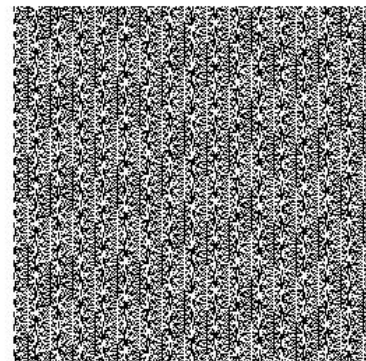
In the beginning there were just...

Random Numbers

- 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



RANDOM.ORG



PHP rand() on Microsoft Windows

The generator is defined by **recurrence relation**:

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

where X is the **sequence** of pseudorandom values, and

m , $0 < m$ – the "modulus"

a , $0 < a < m$ – the "multiplier"

c , $0 \leq c < m$ – the "increment"

X_0 , $0 \leq X_0 < m$ – the "seed" or "start value"

Relaxation

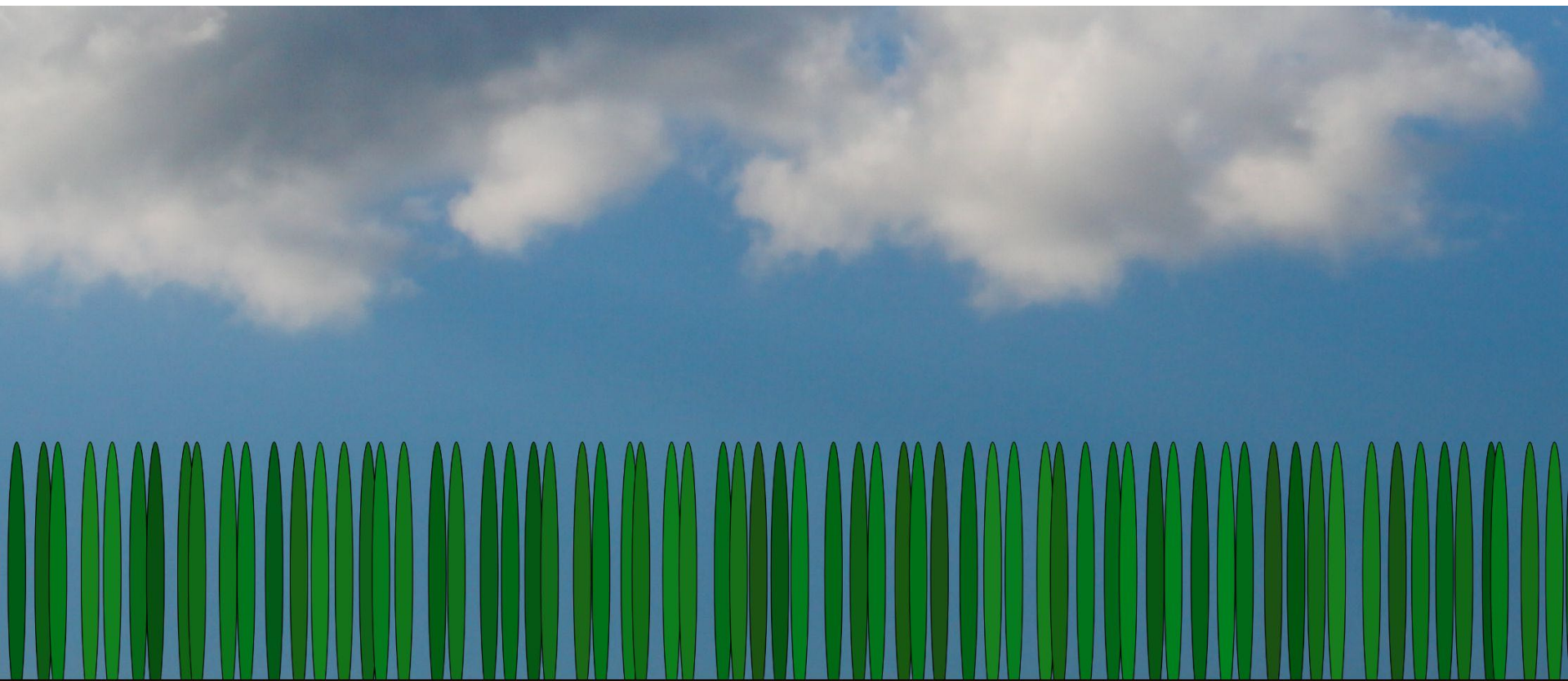
Grass blowing in the wind



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

Motivating example

Grass blades rotated using random()



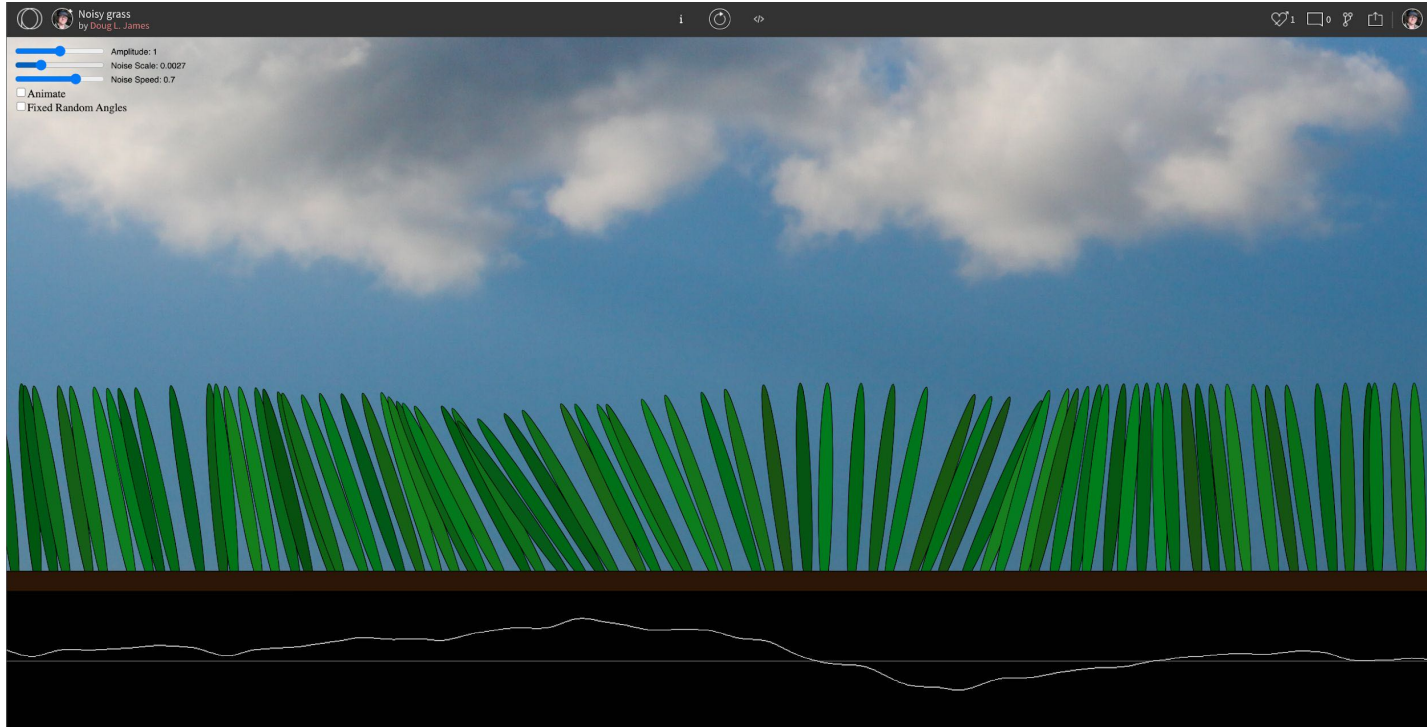
Motivating example

Grass blades rotated using random() :/



Motivating example

“Noisy Grass”



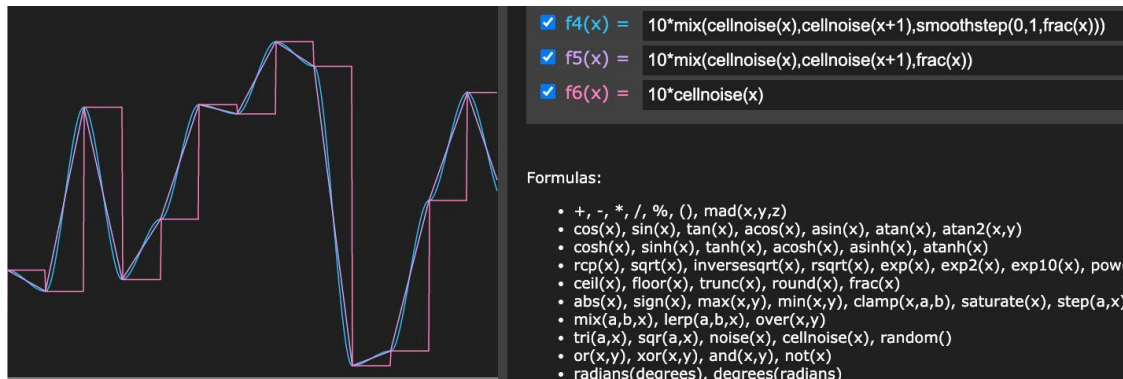
<https://www.openprocessing.org/sketch/990261>

Types of Noise

Simplest noise

Value Noise

- Simple noise model
- Blend random grid values



[http://www.iquliezles.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://www.iquliezles.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))

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)

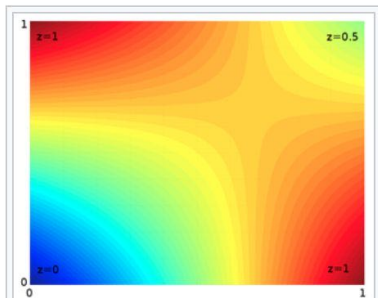
interpolate
→



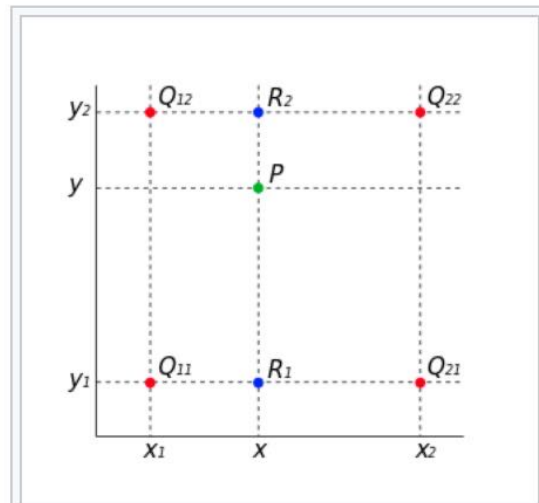
Blended (cubic)

Blending functions

Bilinear Interpolation



Example of bilinear interpolation on the unit square with the z values 0, 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.

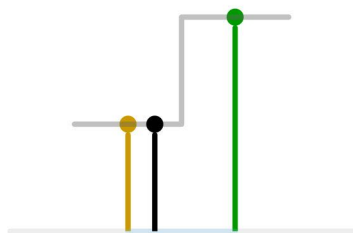
Unit square [\[edit \]](#)

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

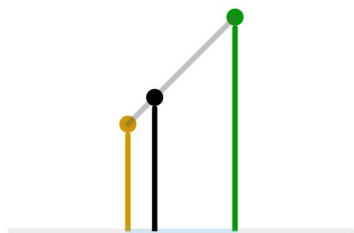
$$f(x, y) \approx f(0, 0)(1 - x)(1 - y) + f(1, 0)x(1 - y) + f(0, 1)(1 - x)y + f(1, 1)xy,$$

Blending functions

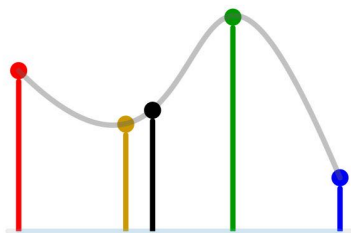
Bicubic Interpolation



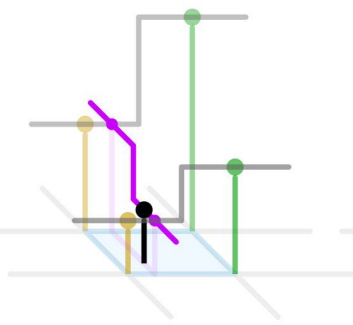
1D nearest-neighbour



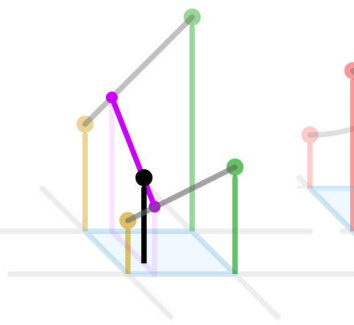
Linear



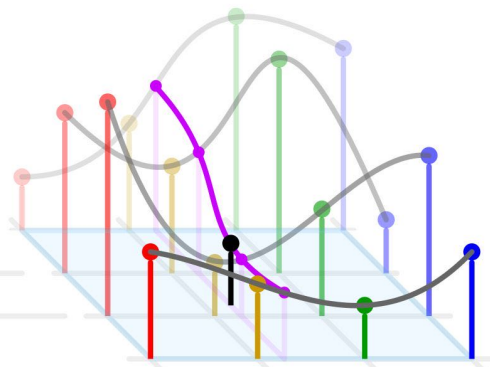
Cubic



2D nearest-neighbour



Bilinear



Bicubic

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

Blending functions

Cubic blending function

Simple Hack:

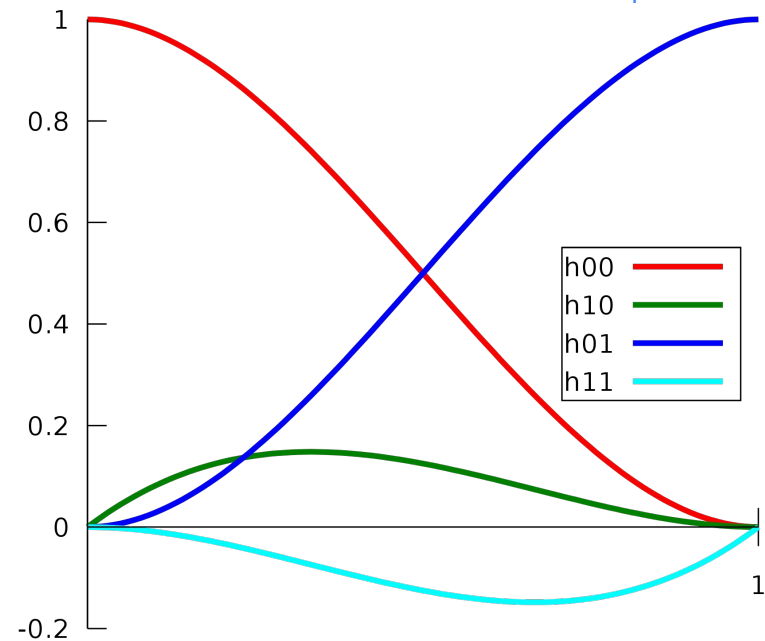
- Assume gradients are zero at vertices.
- Just blend data using Hermite functions(!)

Example: 1D Hermite interpolation set $m_0=m_1=0$

$$p(t) = h_{00}(t)p_0 + \cancel{h_{10}(t)m_0} + h_{01}(t)p_1 + \cancel{h_{11}(t)m_1}$$

	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



Simplest noise

Value Noise



Random Values (at vertices)

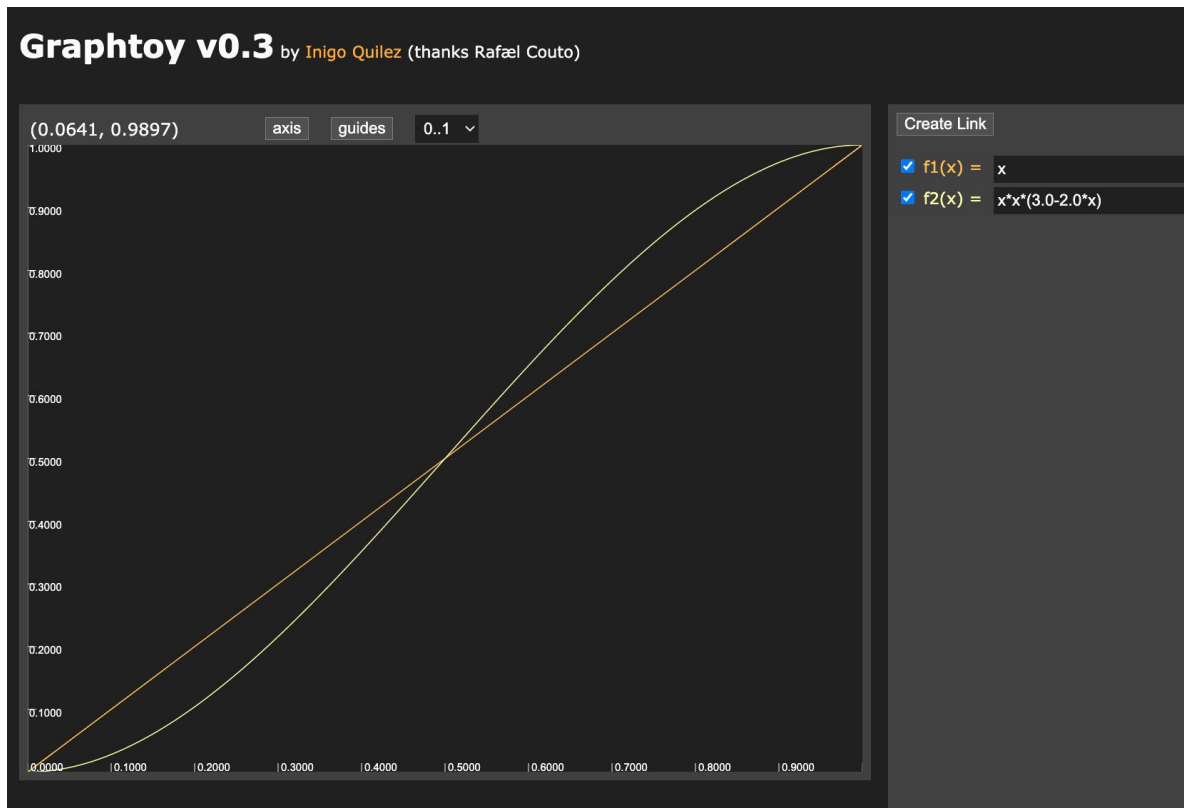
interpolate
→



Blended (cubic)

Blending functions

Blending Vertex Values




Multiple octaves

Fractal Noise (a.k.a. “turbulence”)

COARSE

FINE


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

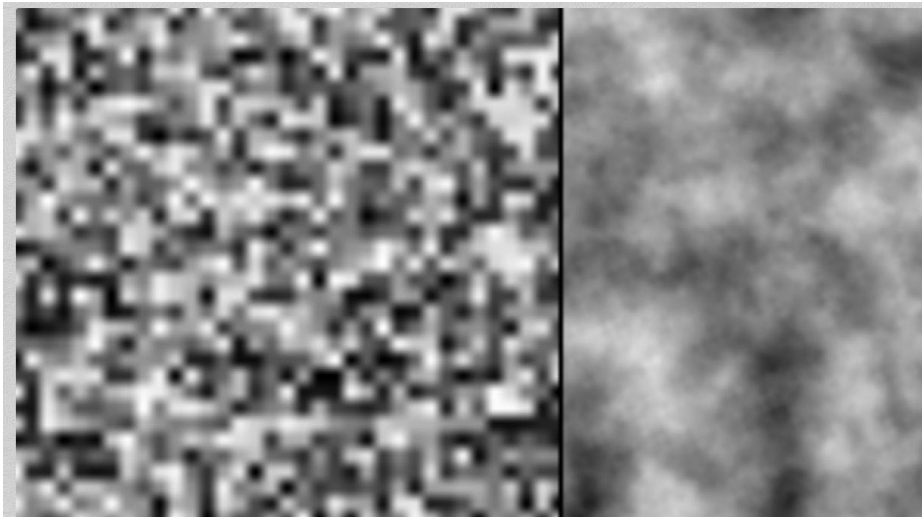
$$= \sum_{\ell=0}^L \frac{1}{2^{\ell}} \text{noise}(2^{\ell} \mathbf{p})$$

“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)



20.88 59.9 fps 1200 x 675

Noise - value - 2D

Views: 16824, Tags: procedural, 2d, noise, perlin

Value Noise - It's cheap compared to Gradient Noise ([url]https://www.shadertoy.com/view/XdXGWB[url]) and produces blocky patterns (left), but it's good enough to generate fractal noise in most cases (right).

Comments (8)



Your comment...

```
41 float noise( in vec2 p )
42 {
43     vec2 i = floor( p );
44     vec2 f = fract( p );
45
46     vec2 u = f*f*(3.0-2.0*f);
47
48     return mix( mix( hash( i + vec2(0.0,0.0) ),
49                   hash( i + vec2(1.0,0.0) ), u.x ),
50              mix( hash( i + vec2(0.0,1.0) ),
51                  hash( i + vec2(1.0,1.0) ), u.x ), u.y );
52 }
53
54 // -----
55
56 void mainImage( out vec4 fragColor, in vec2 fragCoord )
57 {
58     vec2 p = fragCoord.xy / iResolution.xy;
59     vec2 uv = p*vec2(iResolution.x/iResolution.y,1.0);
60
61     float f = 0.0;
62
63     // left: value noise
64     if( p.x<0.6 )
65     {
66         f = noise( 32.0*uv );
67     }
68     // right: fbm - fractal noise (4 octaves)
69     else
70     {
71         uv *= 8.0;
72         mat2 m = mat2( 1.6, 1.2, -1.2, 1.6 );
73         f = 0.5000*noise( uv ); uv = m*uv;
74         f += 0.2500*noise( uv ); uv = m*uv;
75         f += 0.1250*noise( uv ); uv = m*uv;
76         f += 0.0625*noise( uv ); uv = m*uv;
77     }
78 }
```

Random values at grid vertices

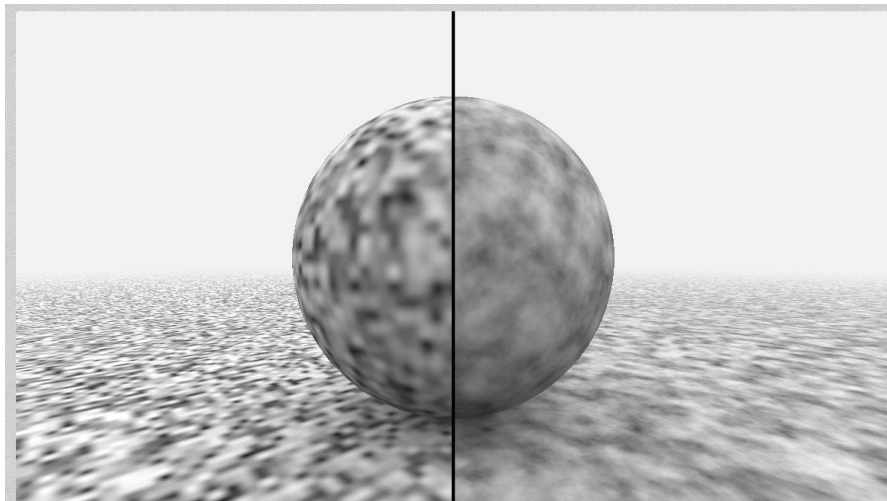
2*rotation to hide grid artifacts

Compiled in 0.0 secs (analyze) 714 chars

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

Demo

Value Noise (3D)



89.57 60.1 fps 1200 x 675

Noise - value - 3D

Views: 36146, Tags: 3d, noise, perlin

A test for LUT based 3D (Value) noise which is much faster than its hash based (purely procedural) counterpart. http://en.wikipedia.org/wiki/Value_noise

Comments (28)



Your comment...

```
+ Image
Shader Inputs
17 // Gradient Noise 3D, Derivatives: https://www.shadertoy.com/view/4dffRH
18 // Value Noise 2D : https://www.shadertoy.com/view/1sf3WH
19 // Value Noise 3D : https://www.shadertoy.com/view/4sfGzS
20 // Gradient Noise 2D : https://www.shadertoy.com/view/XdXGWB
21 // Gradient Noise 3D : https://www.shadertoy.com/view/Xsl3Dl
22 // Simplex Noise 2D : https://www.shadertoy.com/view/Msf3WH
23 // Wave Noise 2D : https://www.shadertoy.com/view/tldSRj
24
25
26 #define USE_PROCEDURAL
27
28 //=====
29 //=====
30 //=====
31
32 #ifndef USE_PROCEDURAL
33 float hash(vec3 p) // replace this by something better
34 {
35     p = fract( p*0.3183099+1 );
36     p *= 17.0;
37     return fract( p.x*p.y*p.z*(p.x+p.y+p.z) );
38 }
39
40 float noise( in vec3 x )
41 {
42     vec3 i = floor(x);
43     vec3 f = fract(x);
44     f = f*f*(3.0-2.0*f);
45
46     return mix(mix(mix( hash(i+vec3(0,0,0)),
47                    hash(i+vec3(1,0,0)), f.x),
48                mix( hash(i+vec3(0,1,0)),
49                    hash(i+vec3(1,1,0)), f.x), f.y),
50            mix(mix( hash(i+vec3(0,0,1)),
51                    hash(i+vec3(1,0,1)), f.x),
52                mix( hash(i+vec3(0,1,1)),
53                    hash(i+vec3(1,1,1)), f.x), f.y), f.z);
54 }
Compiled in 0.0 secs (analyze) 2239 chars
```

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

Better noise

Gradient Noise

Different approach:

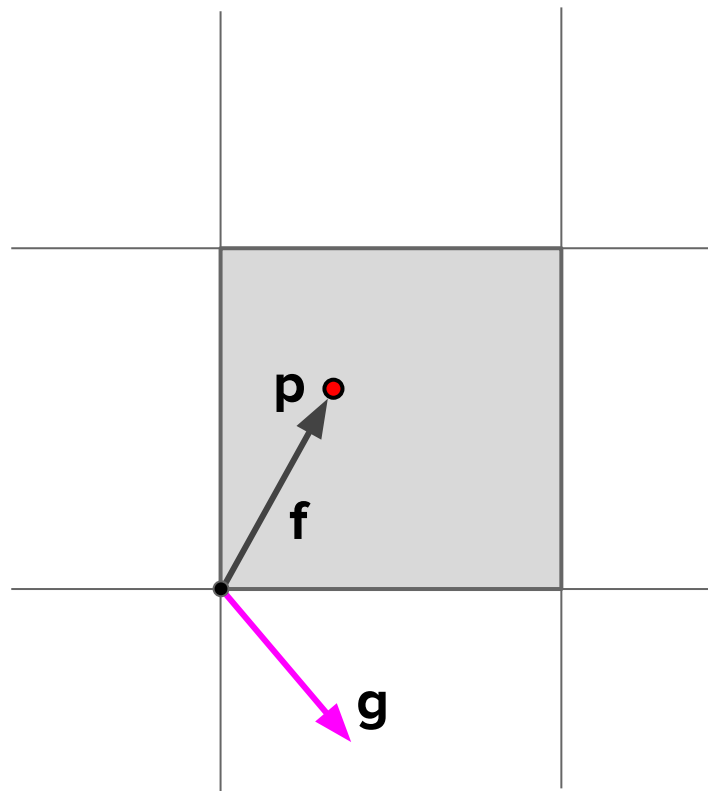
Zero vertex values but *random gradients* $\mathbf{g}=(g_x, g_y)$.

Each vertex's gradient noise uses linear approx.:

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

where $\mathbf{f}=(dx,dy)$ is the vertex delta.

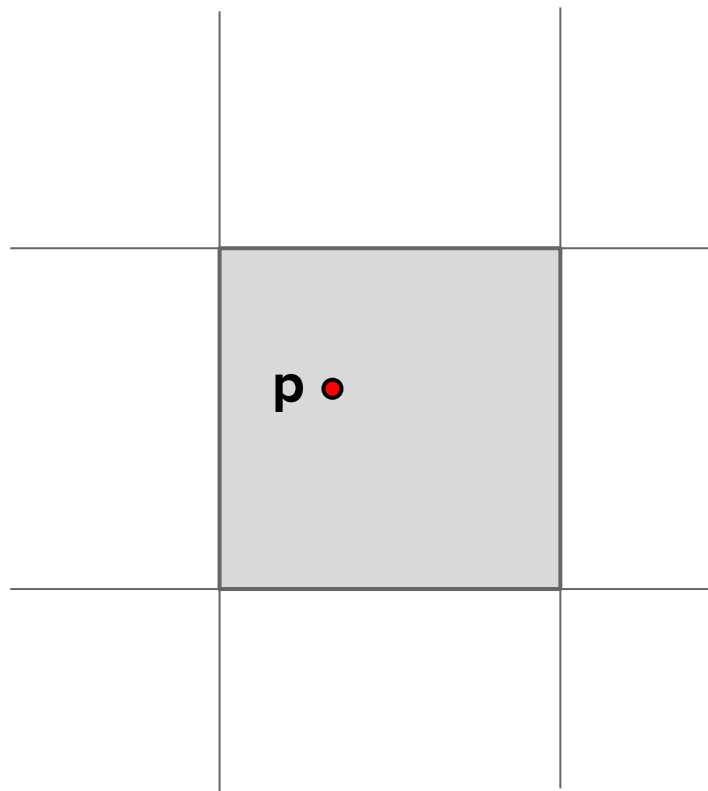
Blend all vertex values across cell like before.



Better noise

Gradient Noise

Given point, \mathbf{p}

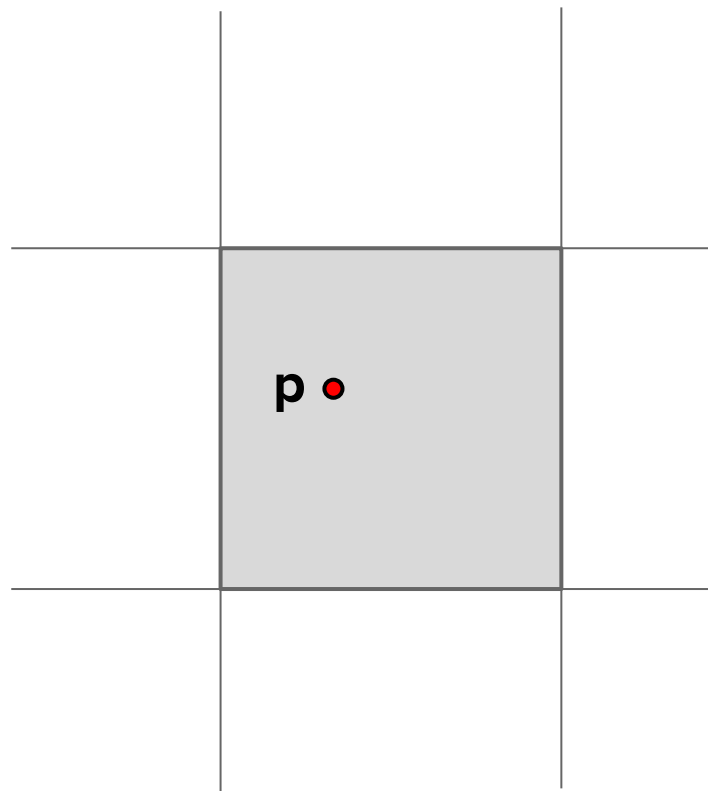


Better noise

Gradient Noise

Given point, \mathbf{p}

Get containing cell: $\mathbf{i}=\text{floor}(\mathbf{p})$



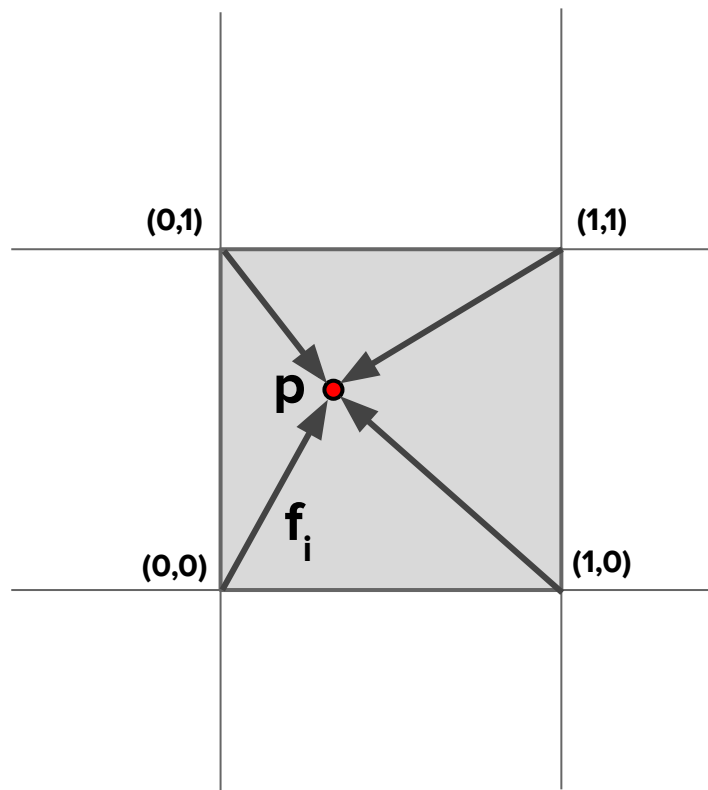
Better noise

Gradient Noise

Given point, \mathbf{p}

Get containing cell: $\mathbf{i} = \text{floor}(\mathbf{p})$

Extract **vertex offsets**, $\mathbf{f} = \text{fract}(\mathbf{p})$



Better noise

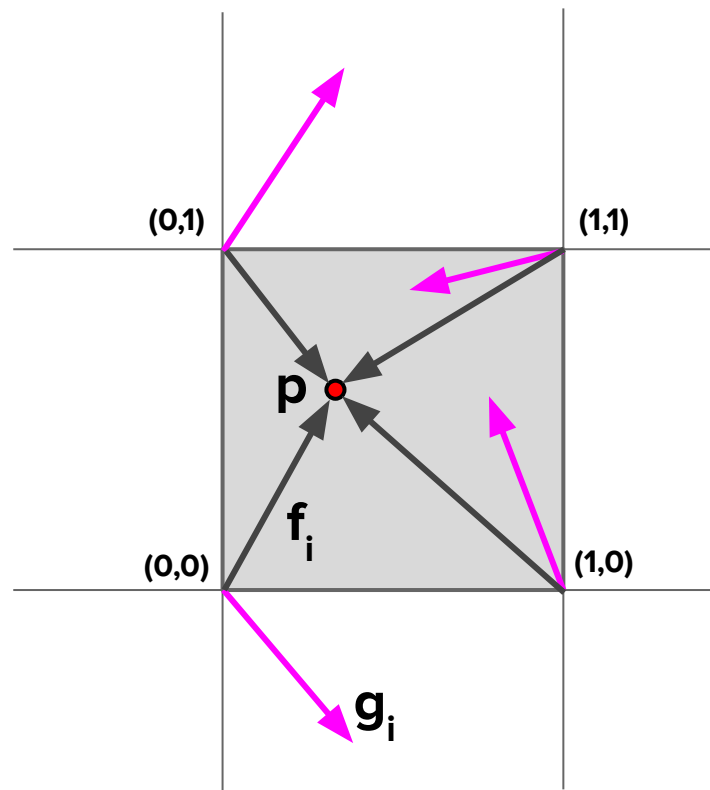
Gradient Noise

Given point, \mathbf{p}

Get containing cell: $\mathbf{i} = \text{floor}(\mathbf{p})$

Extract **vertex offsets**, $\mathbf{f} = \text{fract}(\mathbf{p})$

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



Better noise

Gradient Noise

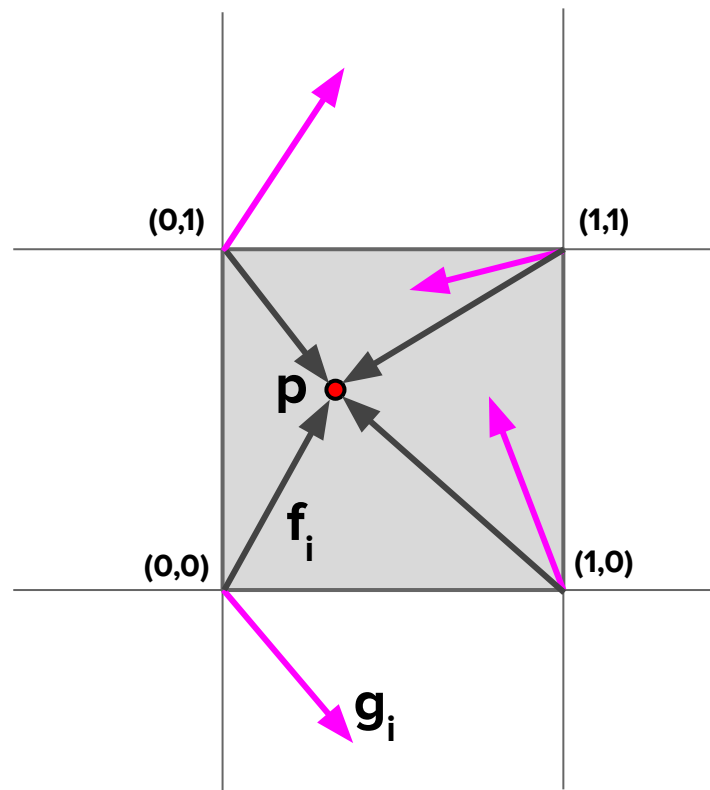
Given point, \mathbf{p}

Get containing cell: $\mathbf{i} = \text{floor}(\mathbf{p})$

Extract **vertex offsets**, $\mathbf{f} = \text{fract}(\mathbf{p})$

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

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



Better noise

Gradient Noise

Given point, \mathbf{p}

Get containing cell: $\mathbf{i} = \text{floor}(\mathbf{p})$

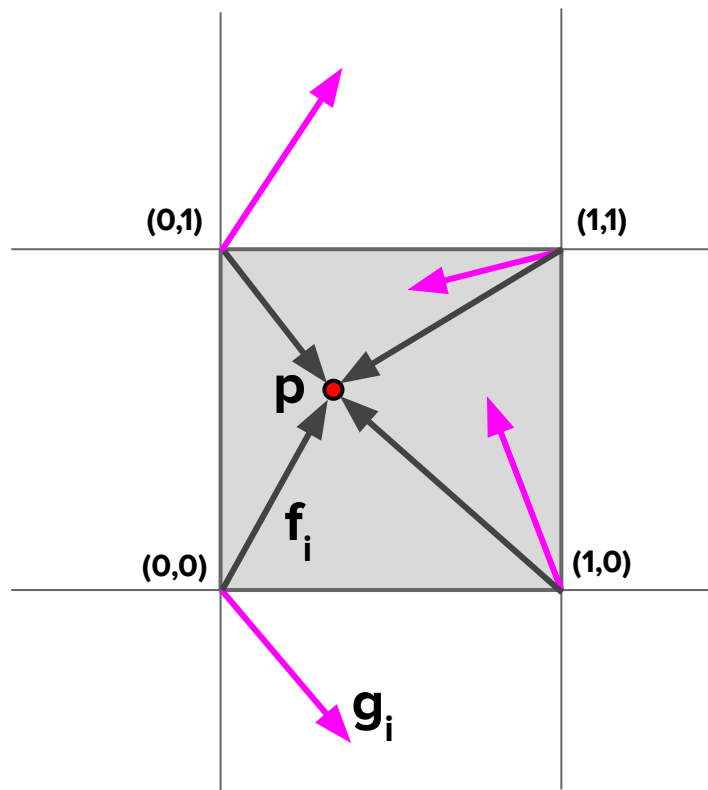
Extract **vertex offsets**, $\mathbf{f} = \text{fract}(\mathbf{p})$

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

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

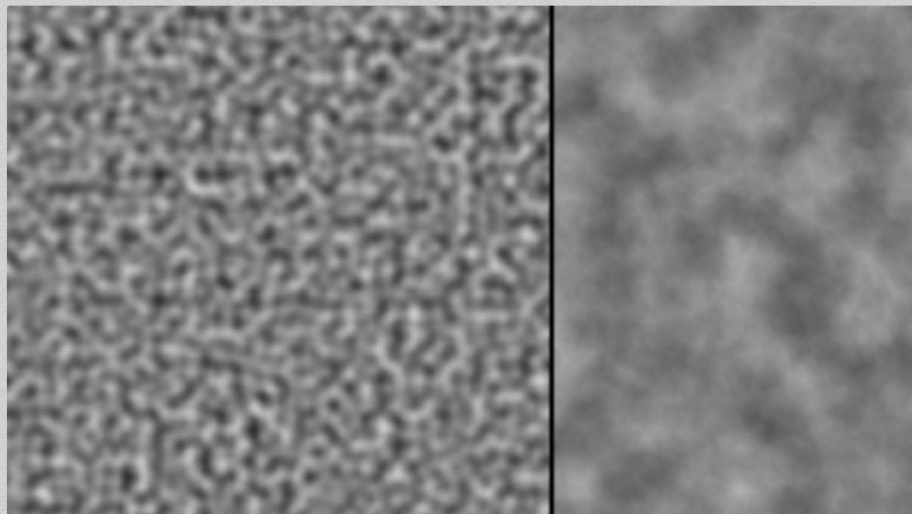
Blend vertex values across cell:

$$\text{Noise} = \sum_{i=1..4} w_i (\mathbf{f}_i \cdot \mathbf{g}_i)$$



Demo

Gradient Noise 2D



M II 18.93 60.0 fps 1200 x 675

Noise - gradient - 2D

Views: 14955, Tags: procedural, 2d, noise, perlin, gradient

Gradient Noise. Slightly more expensive than "Value Noise" (<https://www.shadertoy.com/view/tf3WH>), but higher quality.

Comments (6)



Your comment...

```
44 float noise( in vec2 p )
45 {
46     vec2 i = floor( p );
47     vec2 f = fract( p );
48
49     vec2 u = f*f*(3.0-2.0*f);
50
51     return mix( mix( dot( hash( i + vec2(0.0,0.0) ), f - vec2(0.0,0.0) ),
52                   dot( hash( i + vec2(1.0,0.0) ), f - vec2(1.0,0.0) ), u.x),
53             mix( dot( hash( i + vec2(0.0,1.0) ), f - vec2(0.0,1.0) ),
54                 dot( hash( i + vec2(1.0,1.0) ), f - vec2(1.0,1.0) ), u.x, u.y);
55 }
56
57 // -----
58
59 void mainImage( out vec4 fragColor, in vec2 fragCoord )
60 {
61     vec2 p = fragCoord.xy / iResolution.xy;
62     vec2 uv = p*vec2(iResolution.x/iResolution.y,1.0);
63     float f = 0.0;
64
65     // left: noise
66     if( p.x<0.6 )
67     {
68         f = noise( 32.0*uv );
69     }
70
71     // right: fractal noise (4 octaves)
72     else
73     {
74         uv *= 8.0;
75         mat2 m = mat2( 1.6, 1.2, -1.2, 1.6 );
76         f = 0.500*noise( uv ); uv = m*uv;
77         f += 0.250*noise( uv ); uv = m*uv;
78         f += 0.125*noise( uv ); uv = m*uv;
79         f += 0.0625*noise( uv ); uv = m*uv;
80     }
81 }
```

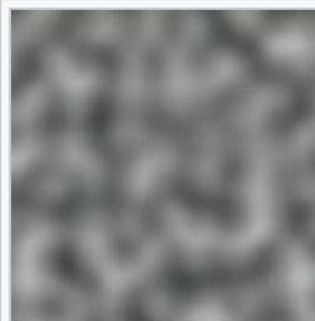
Compiled in 0.0 secs (analyze) 818 chars

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

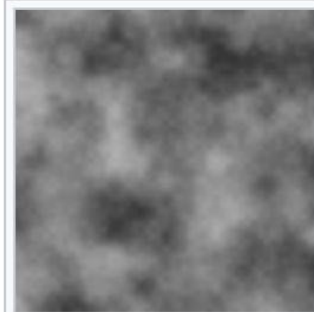
The classic noise function

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 $z=0$



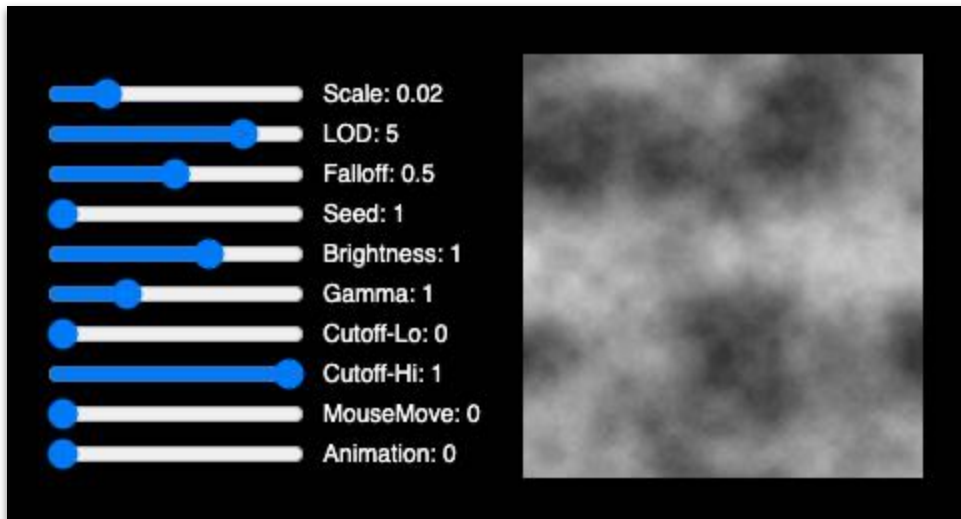
Perlin noise rescaled and added into itself to create [fractal noise](#).



An organic surface generated with Perlin noise

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



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

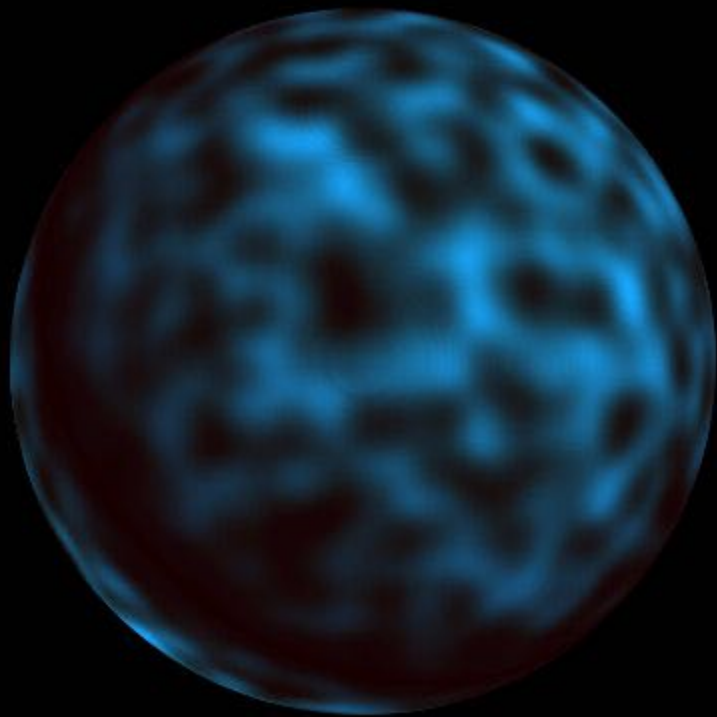
From Ken's noise talk

Computing the pseudo-random gradient:

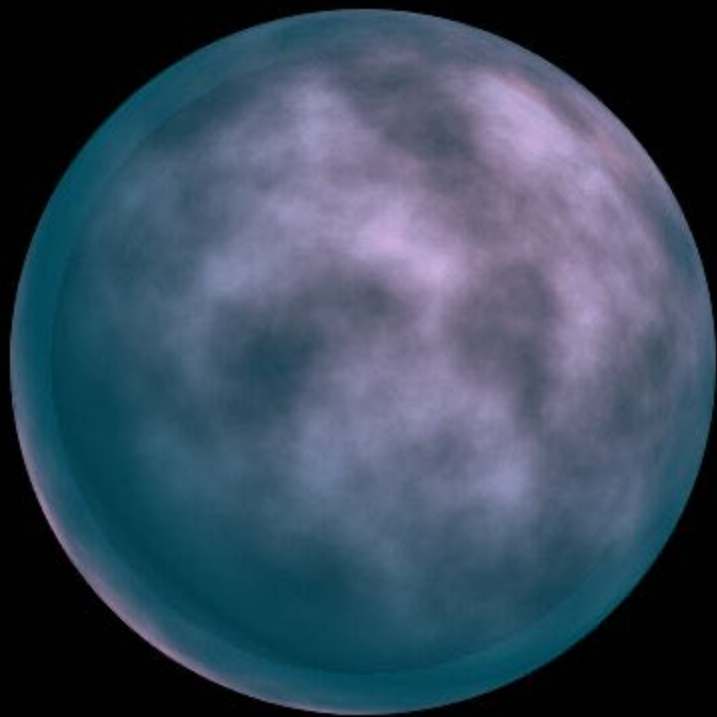
- Precompute table of permutations $P[n]$
- Precompute table of gradients $G[n]$
- $G = G[(i + P[(j + P[k]) \bmod n]) \bmod n]$

Some example implementations:

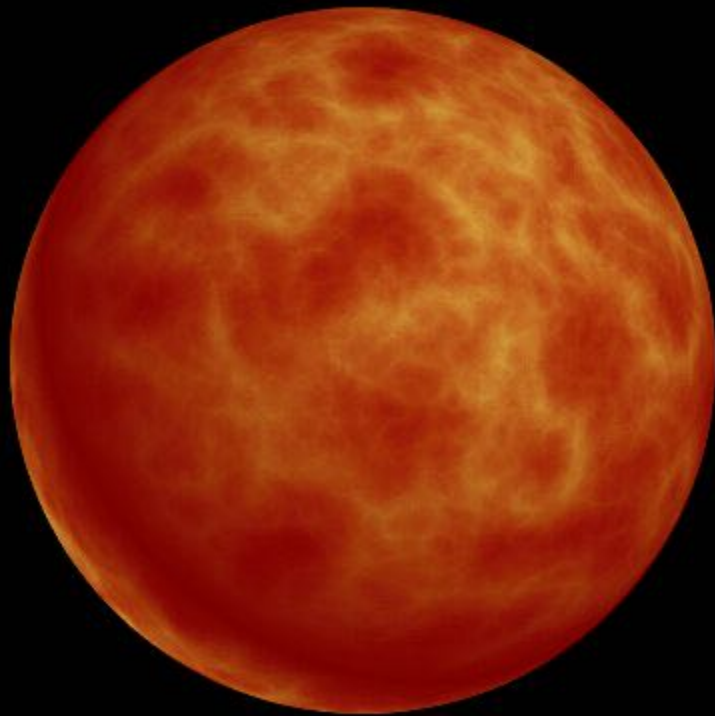
https://rosettacode.org/wiki/Perlin_noise#C



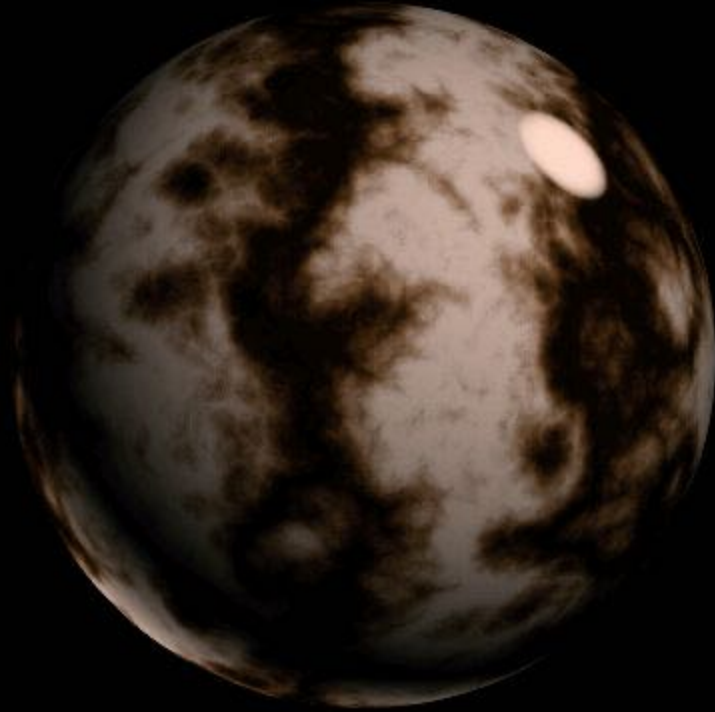
noise(**p**)



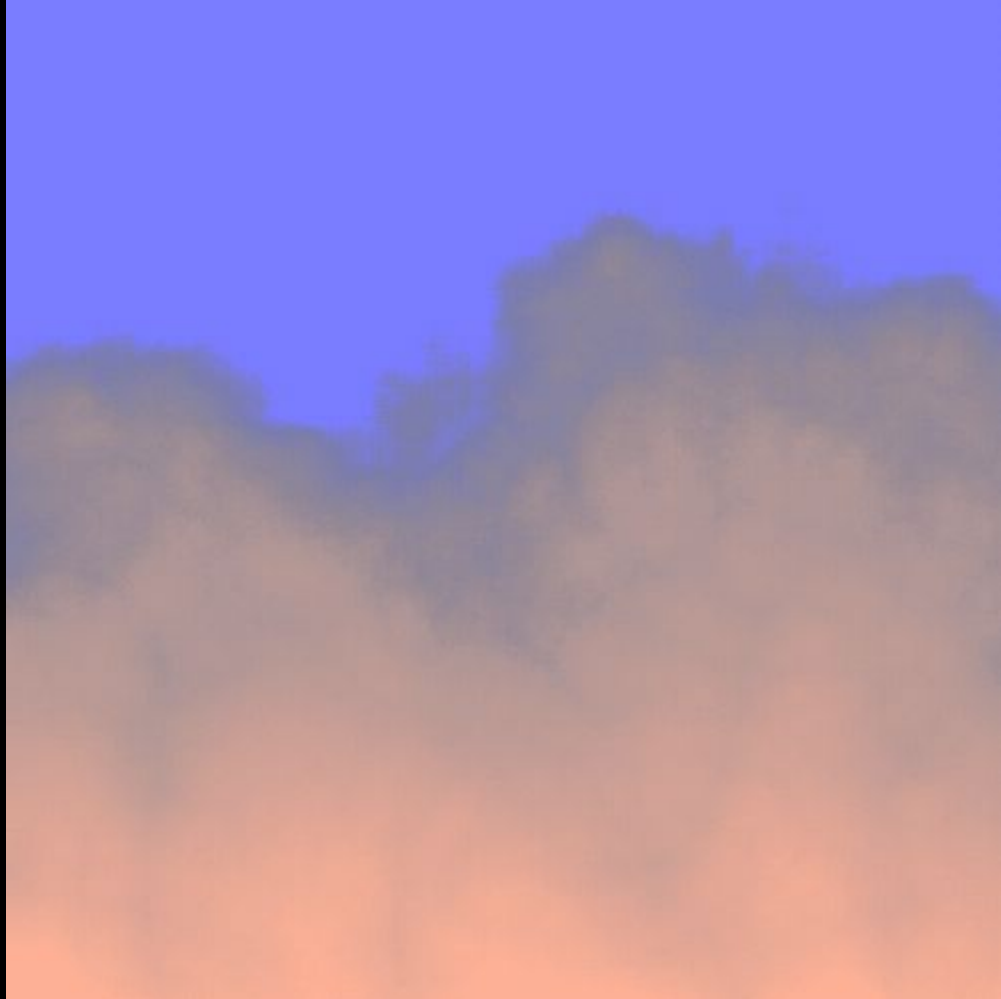
$$\text{noise}(\mathbf{p}) + \frac{1}{2}\text{noise}(2\mathbf{p}) + \frac{1}{4}\text{noise}(4\mathbf{p}) + \dots$$



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



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



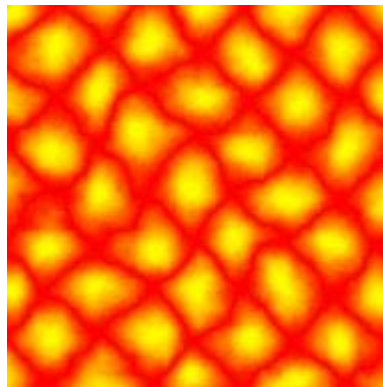
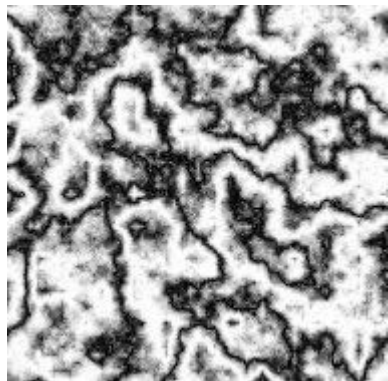
colorMap(y + F(x,y,z))

Further reading

Noise Tricks

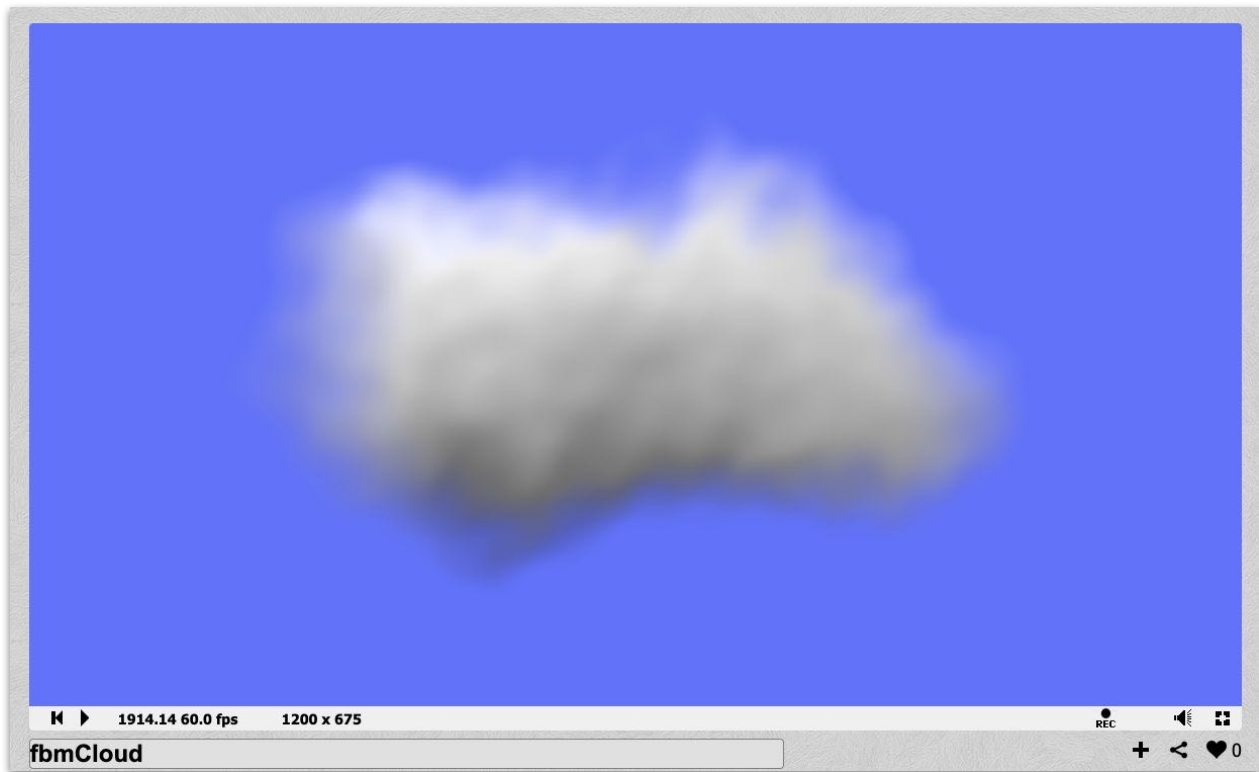
See “Texture generation using random noise” by Lode

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



Shadertoy demo I made

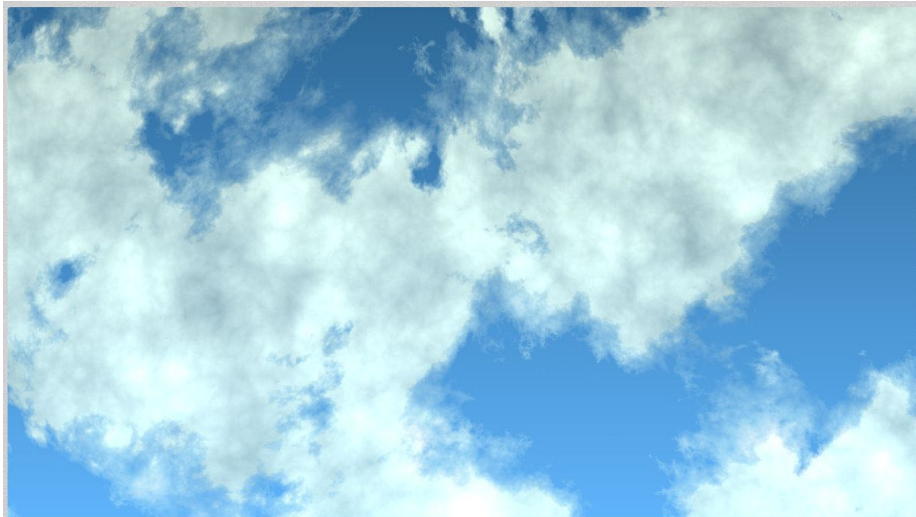
Simple Cloud



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

Shadertoy Demo

"2D Clouds"



48.30 59.9 fps 1200 x 675

2D Clouds ★★

Views: 35529, Tags: fractal, noise, clouds, fbm

Created by drift in 2016-11-13

Used in 2 different demos:
http://www.pouet.net/prod.php?which=66590
and
http://www.pouet.net/prod.php?which=68483

Comments (31)

```
1 const float cloudscale = 1.1;
2 const float speed = 0.03;
3 const float clouddark = 0.5;
4 const float clouddlight = 0.3;
5 const float cloudcover = 0.2;
6 const float clouddalpha = 8.0;
7 const float skytint = 0.5;
8 const vec3 skycolour1 = vec3(0.2, 0.4, 0.6);
9 const vec3 skycolour2 = vec3(0.4, 0.7, 1.0);
10
11 const mat2 m = mat2( 1.6, 1.2, -1.2, 1.6 );
12
13 vec2 hash( vec2 p ) {
14     p = vec2(dot(p,vec2(127.1,311.7)), dot(p,vec2(269.5,183.3)));
15     return -1.0 + 2.0*fract(sin(p)*43758.5453123);
16 }
17
18 float noise( in vec2 p ) {
19     const float K1 = 0.366025404; // (sqrt(3)-1)/2;
20     const float K2 = 0.211324865; // (3-sqrt(3))/6;
21     vec2 i = floor(p + (p.x+p.y)*K1);
22     vec2 a = p - i + (i.x+i.y)*K2;
23     vec2 o = (a.x>a.y) ? vec2(1.0,0.0) : vec2(0.0,1.0); //vec2 of = 0.5 + 0.5*vec2(sign(e
24     vec2 b = a - o + K2;
25     vec2 c = a - 1.0 + 2.0*K2;
26     vec3 h = max(0.5-vec3(dot(a,a), dot(b,b), dot(c,c)), 0.0 );
27     vec3 n = h*h*h*vec3( dot(a,hash(i+0.0)), dot(b,hash(i+0.0)), dot(c,hash(i+1.0)));
28     return dot(n, vec3(70.0));
29 }
30
31 float fbm(vec2 n) {
32     float total = 0.0, amplitude = 0.1;
33     for (int i = 0; i < 7; i++) {
34         total += noise(n) * amplitude;
35         n = m * n;
36         amplitude *= 0.4;
37     }
38     return total;
39 }
40
```

Compiled in 0.0 secs (analyze) 2044 chars

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

SIGGRAPH 2002

Improved Perlin Noise

Smother interpolants & better gradients.



Figure 1a: Noise-displaced superquadric with old interpolants

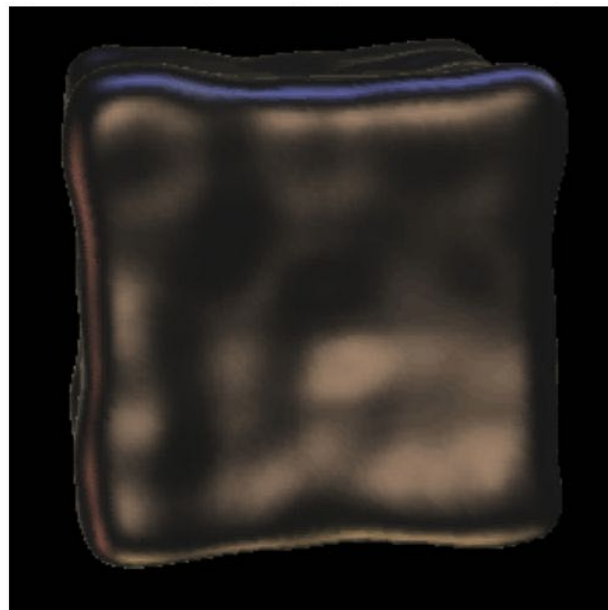


Figure 1b: Noise-displaced superquadric with new interpolants

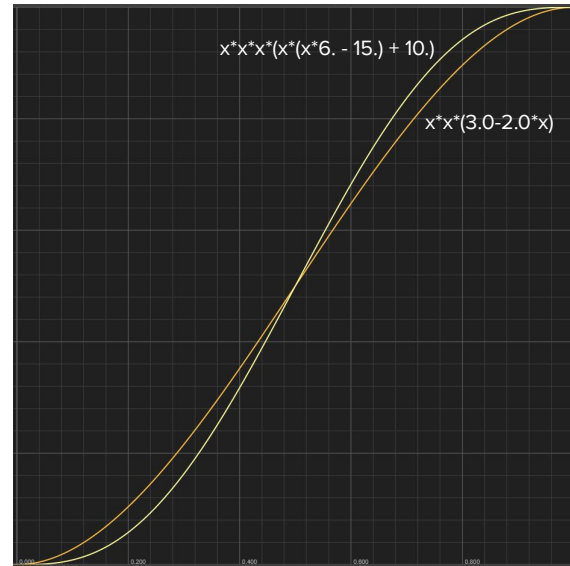
SIGGRAPH 2002

Improved Perlin Noise

Smoother interpolant: Cubic → Quintic blend function:

$$x*x*(3.0-2.0*x) \quad \rightarrow \quad x*x*x*(x*(x*6. - 15.) + 10.)$$

[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

Smother interpolants & better gradients.

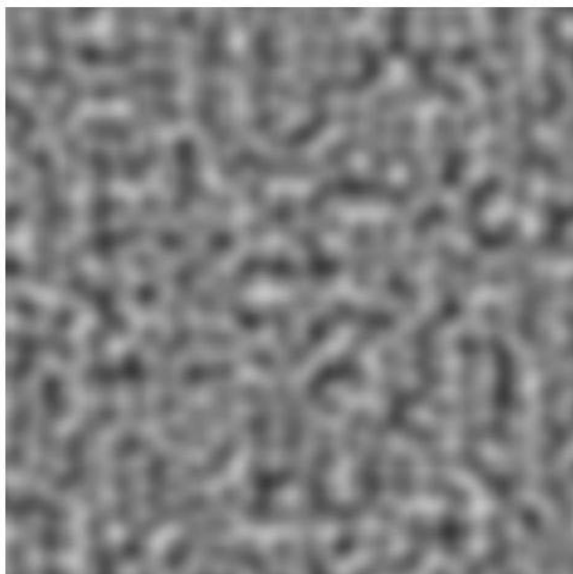


Figure 2a: High-frequency Noise, with old gradient distributions

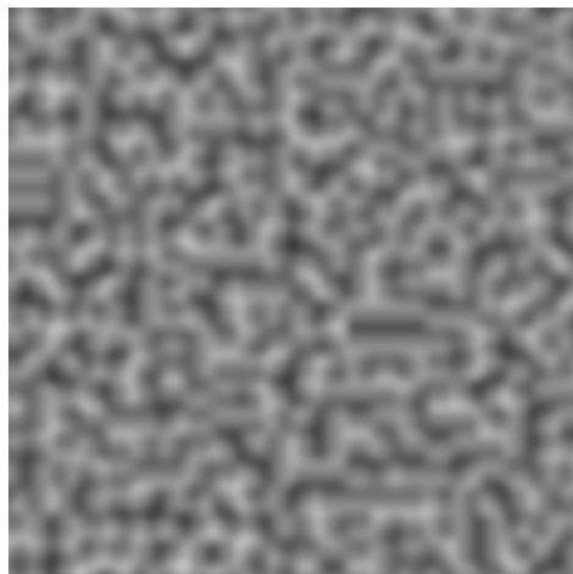


Figure 2b: High-frequency Noise, with new gradient distributions

SIGGRAPH 2002

Improved Perlin Noise

```
// JAVA 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]; }
}
```

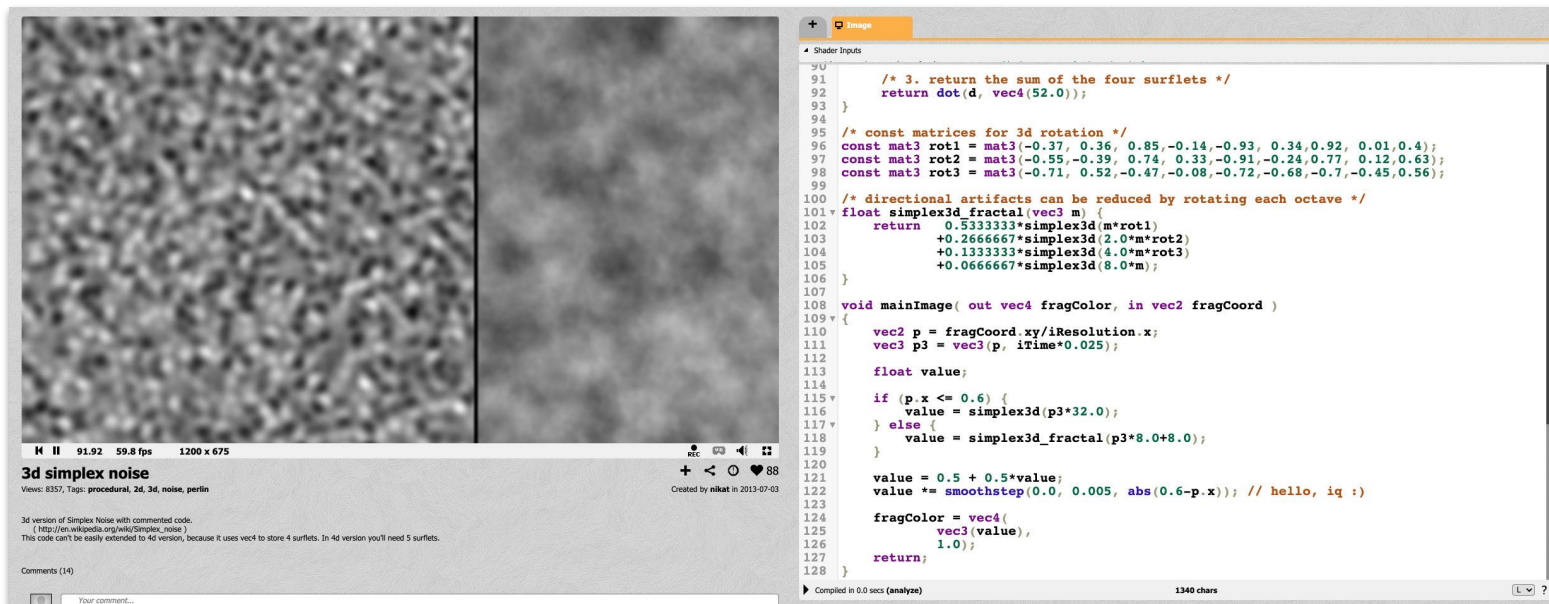
A Perlin noise extension

Simplex Noise (Perlin 1999)

Interpolation on tetrahedra instead of cubes.

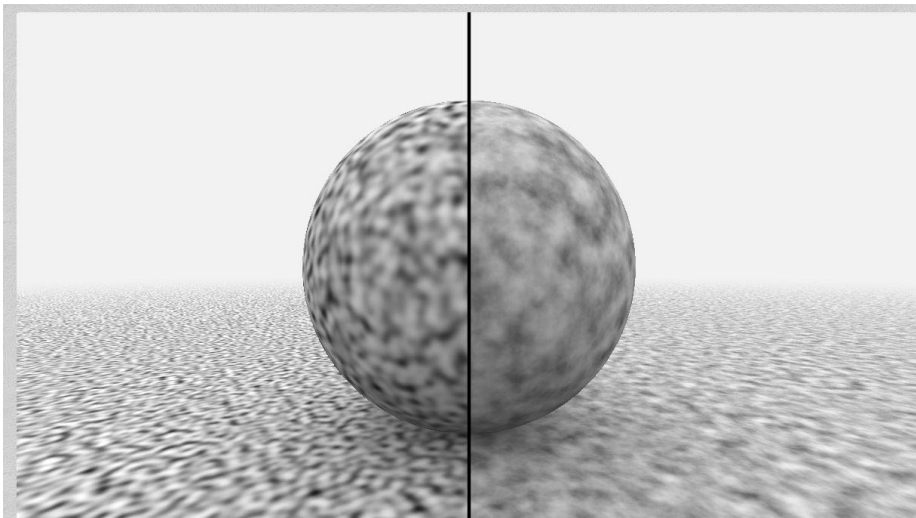
Cheaper in higher dimensions: Only 4 vertex neighbors instead of 2^D in D-dim hypercube.

Fewer directional artifacts. Good for animation!



Detail vs Aliasing

Problems with Gradient Noise



10.75 60.0 fps 1200 x 675

Noise - gradient - 3D

Views: 4706, Tags: procedural, 3d, noise, gradient

3D gradient Noise (. Slightly more expensive than "Value Noise" (<https://www.shadertoy.com/view/lf3WH4>), but higher quality.

Comments (4)



Your comment...

REC

+ < > ❤️ 32

Created by Iq in 2013-10-14

```
+ Image
Shader Inputs
19
20 //=====
21 //=====
22 //=====
23
24 vec3 hash( vec3 p ) // replace this by something better
25 {
26     p = vec3( dot(p,vec3(127.1,311.7, 74.7)),
27             dot(p,vec3(269.5,183.3,246.1)),
28             dot(p,vec3(113.5,271.9,124.6)));
29
30     return -1.0 + 2.0*fract(sin(p)*43758.5453123);
31 }
32
33 float noise( in vec3 p )
34 {
35     vec3 i = floor( p );
36     vec3 f = fract( p );
37
38     vec3 u = f*f*(3.0-2.0*f);
39
40     return mix( mix( mix( dot( hash( i + vec3(0.0,0.0,0.0) ), f - vec3(0.0,0.0,0.0) ),
41                       dot( hash( i + vec3(1.0,0.0,0.0) ), f - vec3(1.0,0.0,0.0) ), u
42                       dot( hash( i + vec3(0.0,1.0,0.0) ), f - vec3(0.0,1.0,0.0) ), u
43                       dot( hash( i + vec3(1.0,1.0,0.0) ), f - vec3(1.0,1.0,0.0) ), u
44                       mix( mix( dot( hash( i + vec3(0.0,0.0,1.0) ), f - vec3(0.0,0.0,1.0) ), u
45                       dot( hash( i + vec3(1.0,0.0,1.0) ), f - vec3(1.0,0.0,1.0) ), u
46                       dot( hash( i + vec3(0.0,1.0,1.0) ), f - vec3(0.0,1.0,1.0) ), u
47                       dot( hash( i + vec3(1.0,1.0,1.0) ), f - vec3(1.0,1.0,1.0) ), u
48                       ) ) )
49 }
50 //=====
51 //=====
52 //=====
53 //=====
54 //=====
55
56 const mat3 m = mat3( 0.00, 0.80, 0.60,
57                    0.80, 0.60, 0.00,
58                    0.60, 0.00, 0.80 );
Compiled in 0.0 secs (analyze) 1928 chars
```

<https://www.shadertoy.com/view/XsI3Dl>

Pixar

Wavelet Noise (SIGGRAPH 2005)

Wavelet Noise

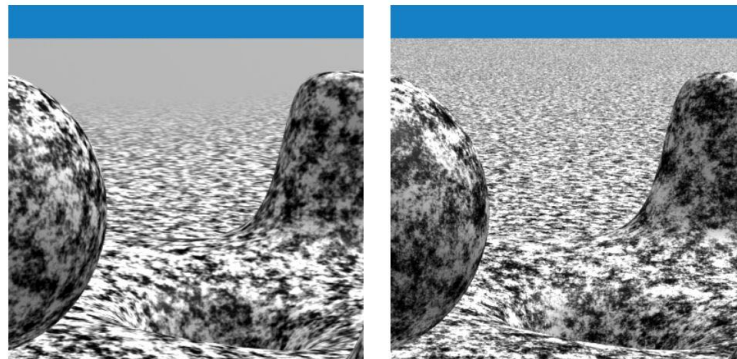
Robert L. Cook Tony DeRose
Pixar Animation Studios

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.

Pixar Wavelet Noise

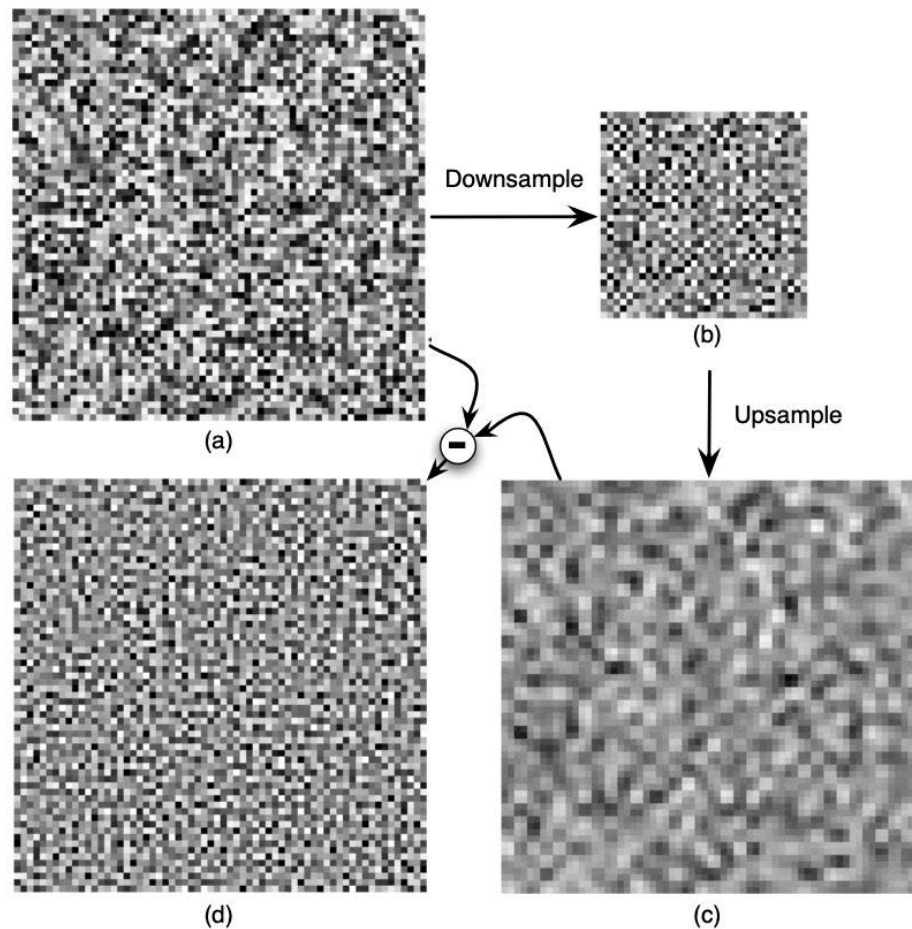
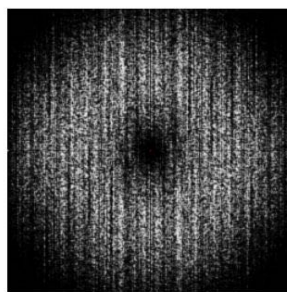
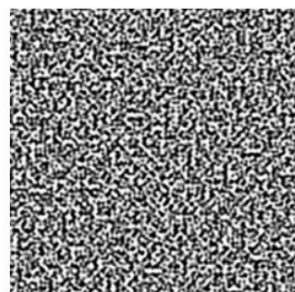
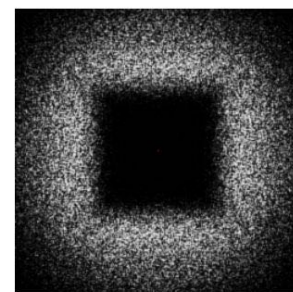
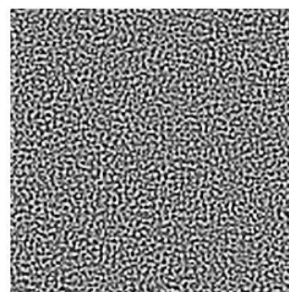


Figure 2: (a). Image R of random noise, (b) Half-size image R^\downarrow , (c) Half-resolution image $R^{\downarrow\uparrow}$, (d) Noise band image $N = R - R^{\downarrow\uparrow}$.

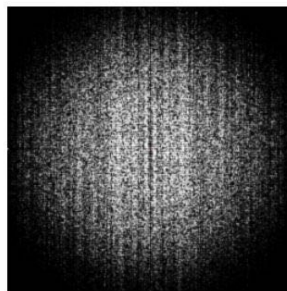
Pixar Wavelet Noise



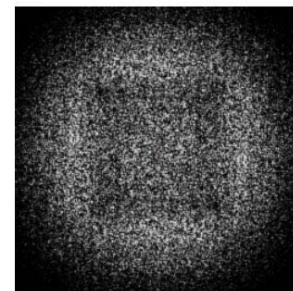
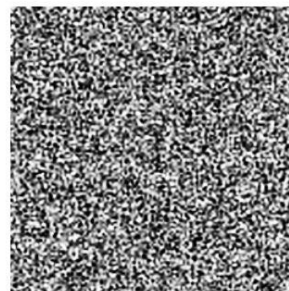
(a) 2d Perlin noise



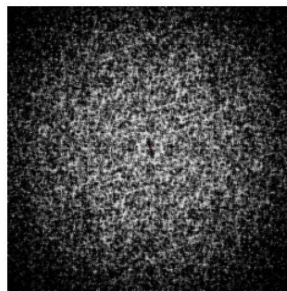
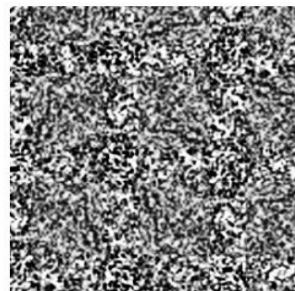
(d) 2d wavelet noise



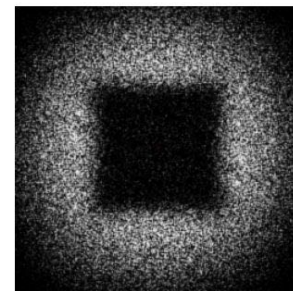
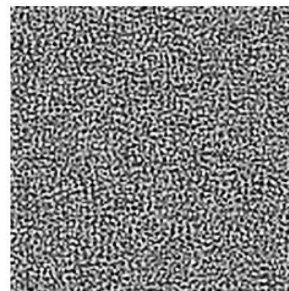
(b) 2D slice through 3D Perlin noise



(e) 2D slice through 3D wavelet noise



(c) 2D white noise



(f) 3D wavelet noise projected onto 2D

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

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



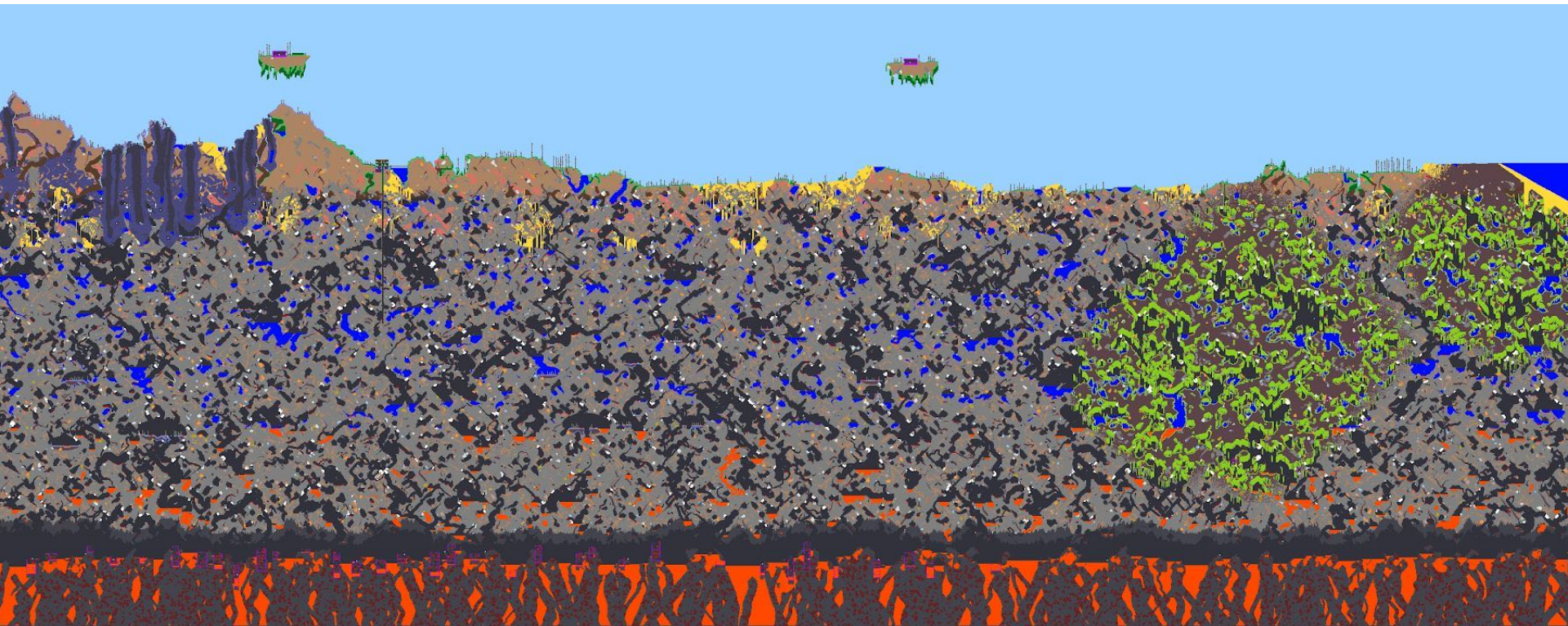
http://vanscher.com/2017-11-07_Playing-with-Perlin-Noise-Generating-Realistic-Archipelagos-b56f004d8401.html



Terraria

Breakout Discussion

Terrain Generation in Terraria



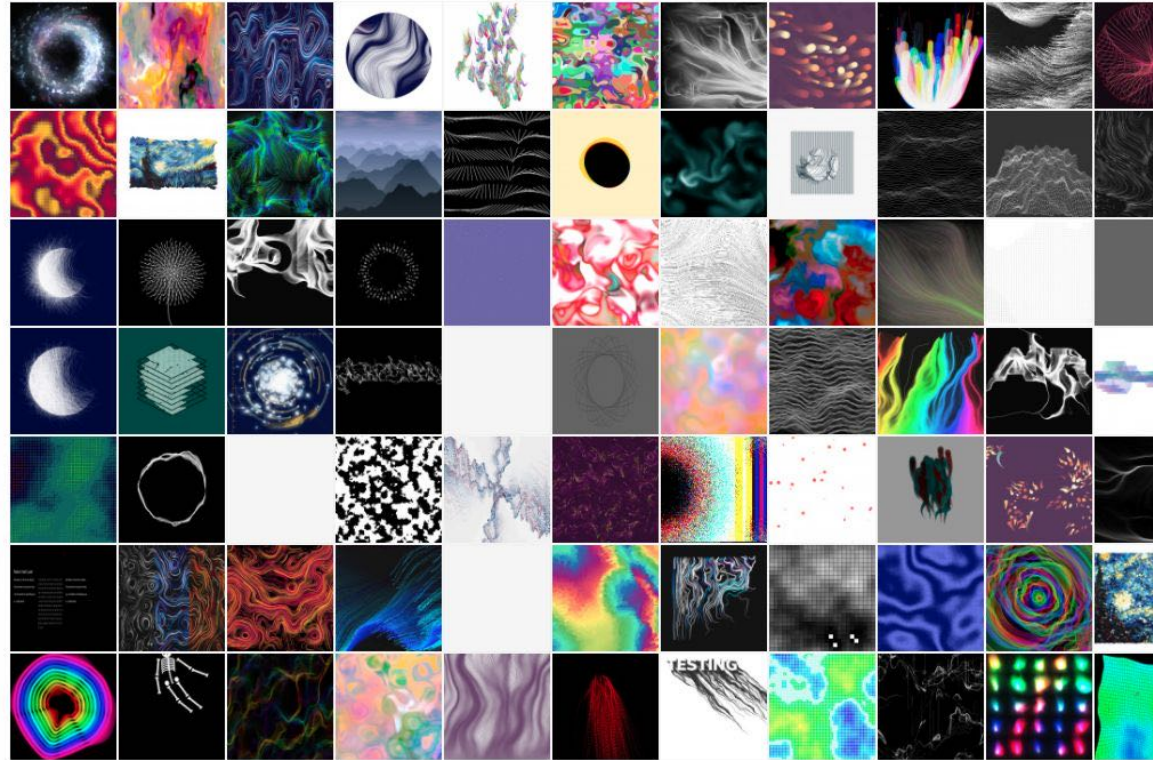
Terraria

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

OpenProcessing

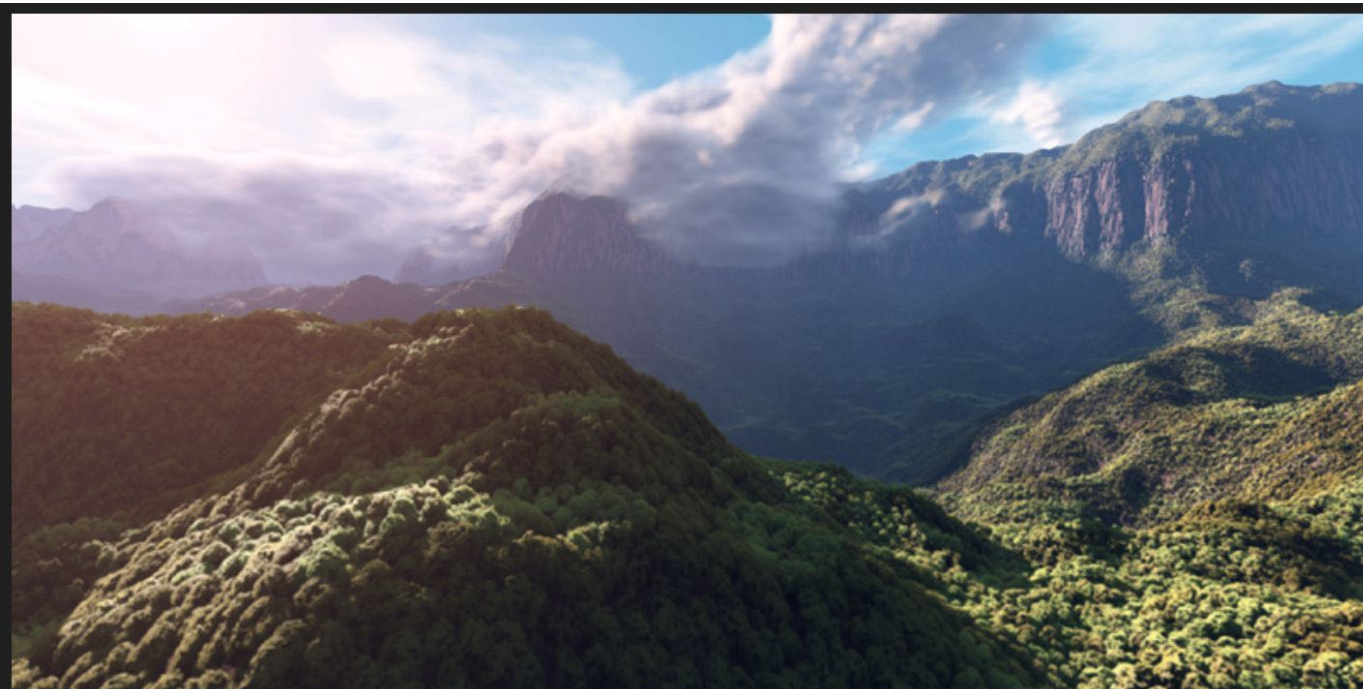
Noisy Sketches

Sketches that are created
received s during this month
are tagged this year
with noise
anytime



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

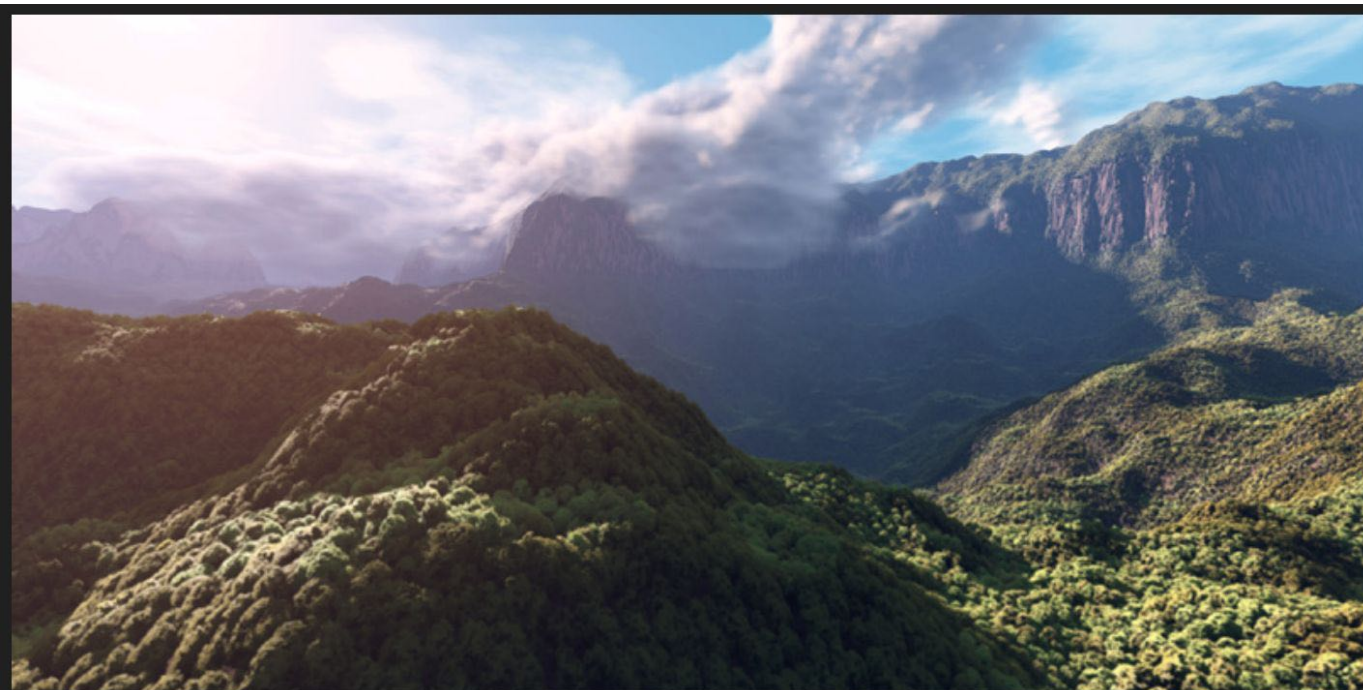


Wow, so much bonus material.

Ray Marching & Implicit Geometry

Shadertoy

“Rainforest” by iq (Inigo Quilez)

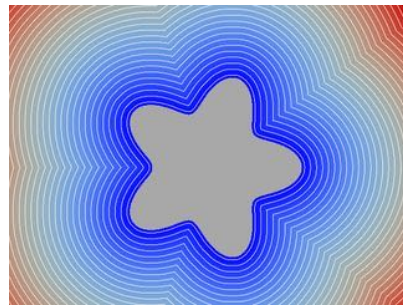


fBM() was used to generate the terrain, the clouds, the tree distribution, their color variations, and the canopy details. "Rainforest", 2016: <https://www.shadertoy.com/view/4ttSWf>

<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

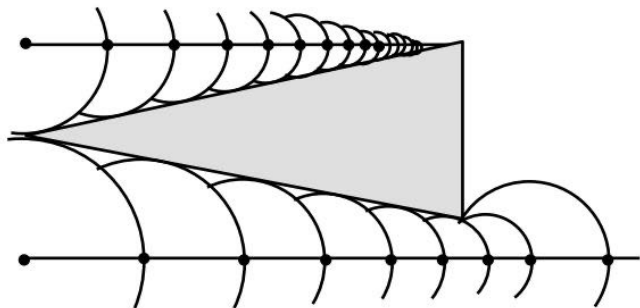
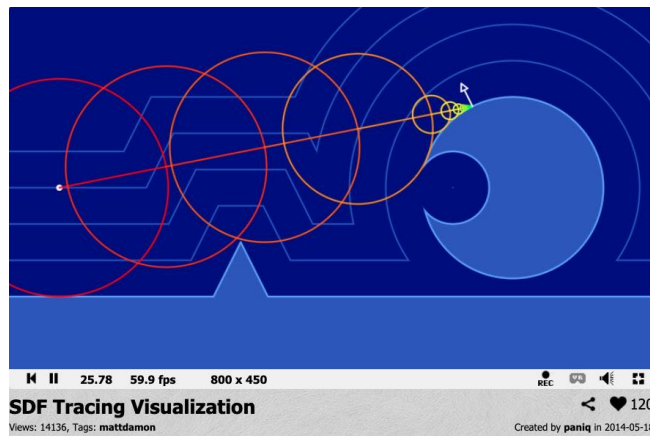


Figure 2: A hit and a miss.



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

Part I:

Implicit Geometry

Distance Fields

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

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

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

$d(x) > 0$: Outside object

$d(x) = 0$: On object

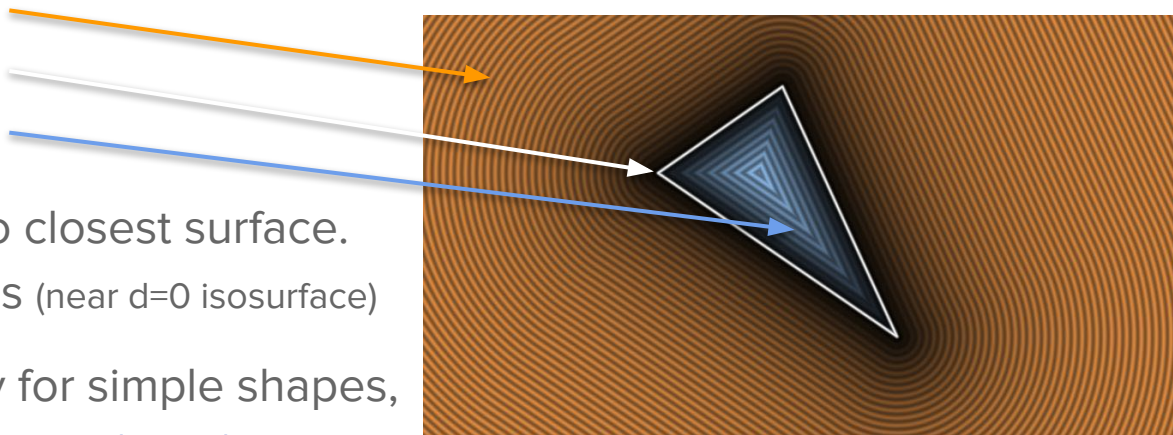
$d(x) < 0$: Inside object

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

Useful for surface normals (near $d=0$ isosurface)

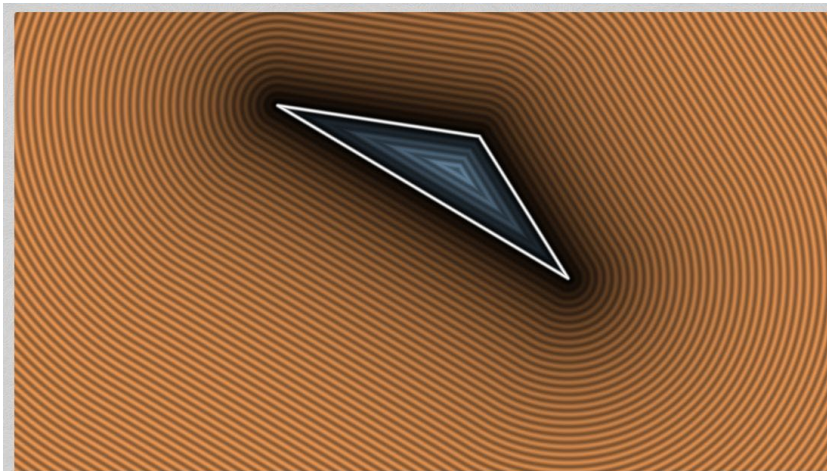
Analytical formulas exist only for simple shapes,

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



Recall

2D Distance Fields



Triangle - distance 2D
Views: 5526, Tags: 2d, triangle, distance
Created by iq in 2014-04-10

Signed distance to a triangle (negative in the inside, positive in the outside). Note there's only one square root involved.

Comments (3)

Your comment...

```
1 // The MIT License
2 // Copyright © 2014 Inigo Quilez
3 // Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated
4
5
6 // See here for a specialization when the triangle is equilateral: https://www.shadertoy.com/view/Xl2yDW
7
8
9 // List of some other 2D distances:
10 //
11 // Circle: https://www.shadertoy.com/view/3ltSW2
12 // Segment: https://www.shadertoy.com/view/3tdSDj
13 // Triangle: https://www.shadertoy.com/view/XsXSz4
14 // Isosceles Triangle: https://www.shadertoy.com/view/MldcD7
15 // Regular Triangle: https://www.shadertoy.com/view/Xl2yDW
16 // Regular Pentagon: https://www.shadertoy.com/view/1lVyWw
17 // Regular Octagon: https://www.shadertoy.com/view/1lGfDG
18 // Rounded Rectangle: https://www.shadertoy.com/view/4l1XD7
19 // Rhombus: https://www.shadertoy.com/view/XdXcRB
20 // Trapezoid: https://www.shadertoy.com/view/MlycD3
21 // Polygon: https://www.shadertoy.com/view/wdBRW
22 // Hexagram: https://www.shadertoy.com/view/t23RR
23 // Regular Star: https://www.shadertoy.com/view/3tSGDy
24 // Star5: https://www.shadertoy.com/view/wlcGzB
25 // Ellipse 1: https://www.shadertoy.com/view/4sS3zz
26 // Ellipse 2: https://www.shadertoy.com/view/4lsXDN
27 // Quadratic Bezier: https://www.shadertoy.com/view/MlKcDD
28 // Uneven Capsule: https://www.shadertoy.com/view/4lcBwn
29 // Vesica: https://www.shadertoy.com/view/XtVFRW
30 // Cross: https://www.shadertoy.com/view/XtGfzw
31 // Pie: https://www.shadertoy.com/view/3l23RK
32 // Arc: https://www.shadertoy.com/view/wl23RK
33 // Horseshoe: https://www.shadertoy.com/view/wlSGW1
34 // Parabola: https://www.shadertoy.com/view/ws3GD7
35 // Parabola Segment: https://www.shadertoy.com/view/3lScz2
36 // Rounded X: https://www.shadertoy.com/view/3KSDc
37 // Joint: https://www.shadertoy.com/view/WldGMM
38 // Simple Egg: https://www.shadertoy.com/view/Wdjfz3
39 //
40 // and many more here: http://www.iquilezles.org/www/articles/distfunctions2d/distfunctions2d.htm
41
```

Compiled in 0.0 secs (analyze) 943 chars

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

Where is the object?

Indicator Functions, $I(x)$

Step indicator function: $I(x) = \text{step}(0, -d(x))$

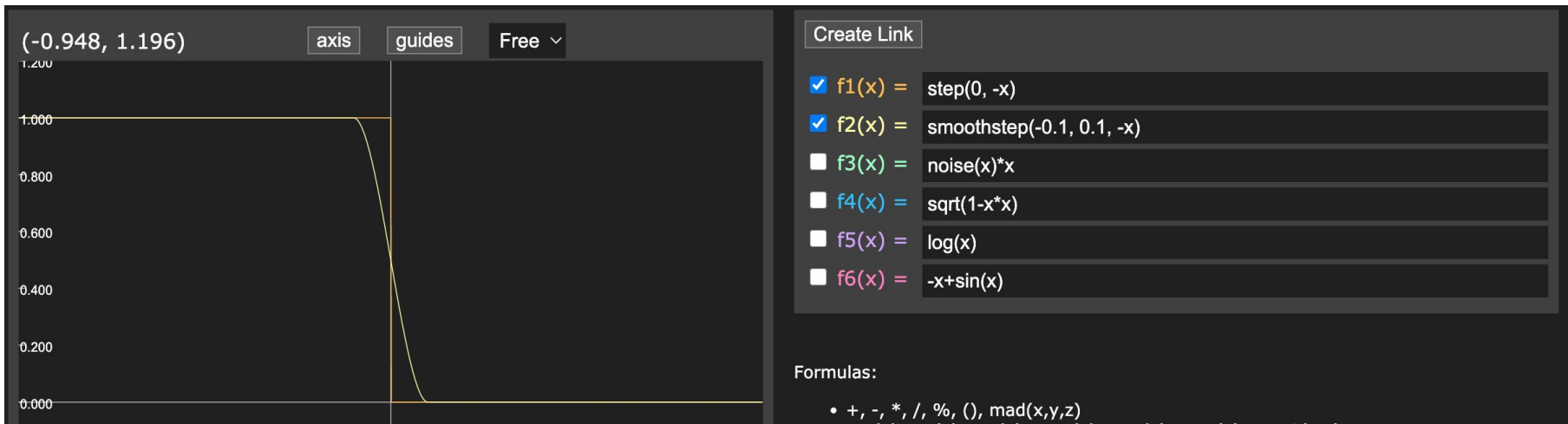
- 1 inside, $d \leq 0$
- 0 outside, $d > 0$

Smooth-step indicator function: $\text{smoothstep}(-\text{eps}, \text{eps}, -d)$

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

$I(x)=0$
(outside)

$I(x)=1$
(inside)



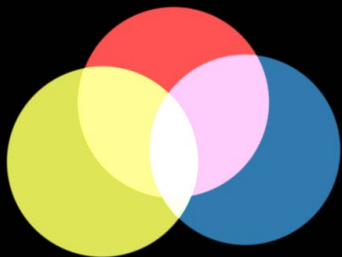
Indicator function application

Color Mixing



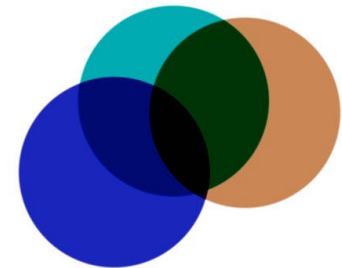
“On top”:

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



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;
}
```

```
1029     ret = max(ret, col3, 0); // here, previous color can be gray,
1030                                     // blue or pink.
1031 }
1032 v
1033 else if(p.x < 2./3.) { // Part II
1034     // Color addition
1035     // This is how lights of different colors add up
1036     // http://en.wikipedia.org/wiki/Additive_color
1037     ret = black; // start with black pixels
1038     ret += disk(r, vec2(0.1,0.3), 0.4)*col1; // add the new color
1039                                     // to the previous color
1040     ret += disk(r, vec2(-.1,0.0), 0.4)*col2;
1041     ret += disk(r, vec2(.15,-0.3), 0.4)*col3;
1042     // when all components of "ret" becomes equal or higher than 1.0
1043     // it becomes white.
1044 }
1045 v
1046 else if(p.x < 3./3.) { // Part III
1047     // Color subtraction
1048     // This is how dye of different colors add up
1049     // http://en.wikipedia.org/wiki/Subtractive_color
1050     ret = white; // start with white
1051     ret -= disk(r, vec2(1.1,0.3), 0.4)*col1;
1052     ret -= disk(r, vec2(1.05,0.0), 0.4)* col2;
1053     ret -= disk(r, vec2(1.35,-0.25), 0.4)* col3;
1054     // when all components of "ret" becomes equals or smaller than 0.0
1055     // it becomes black.
1056 }
1057 }
1058 }
1059 }
1060 }
1061 }
1062 }
1063 }
1064 }
1065 }
1066 }
1067 }
1068 }
1069 }
1070 }
1071 }
1072 }
1073 }
1074 }
1075 }
1076 }
1077 }
1078 }
1079 }
1080 }
1081 }
1082 }
1083 }
1084 }
1085 }
1086 }
1087 }
1088 }
1089 }
1090 }
1091 }
1092 }
1093 }
1094 }
1095 }
1096 }
1097 }
1098 }
1099 }
1100 }
```

Analytical SDFs

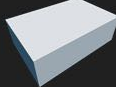
3D Distance Field Primitives

Sphere - exact



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

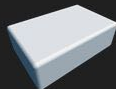
Box - exact



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));
}
```

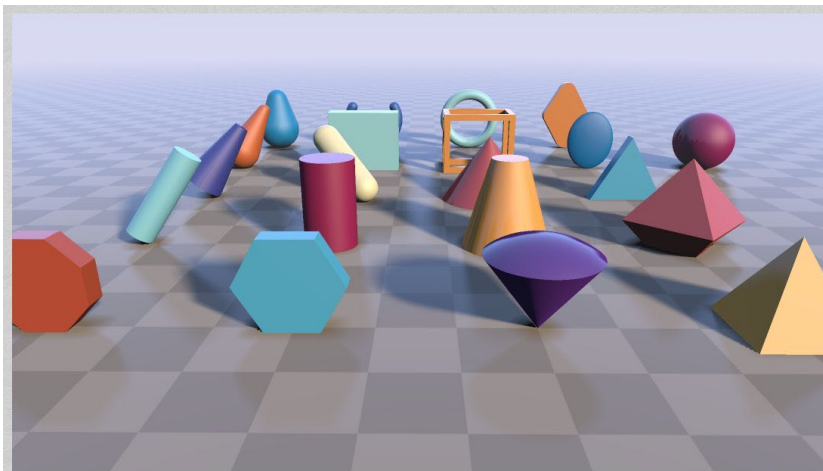
Torus - exact



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


Ray Marching Demo

3D Distance Field Primitives



6.40 31.4 fps 1080 x 607

Raymarching - Primitives

Views: 420825, Tags: procedural, 3d, raymarching, distancefields, primitives

A set of raw primitives. All except the ellipsoid are exact euclidean distances. More info here: <http://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm>

Comments (117)



Your comment...

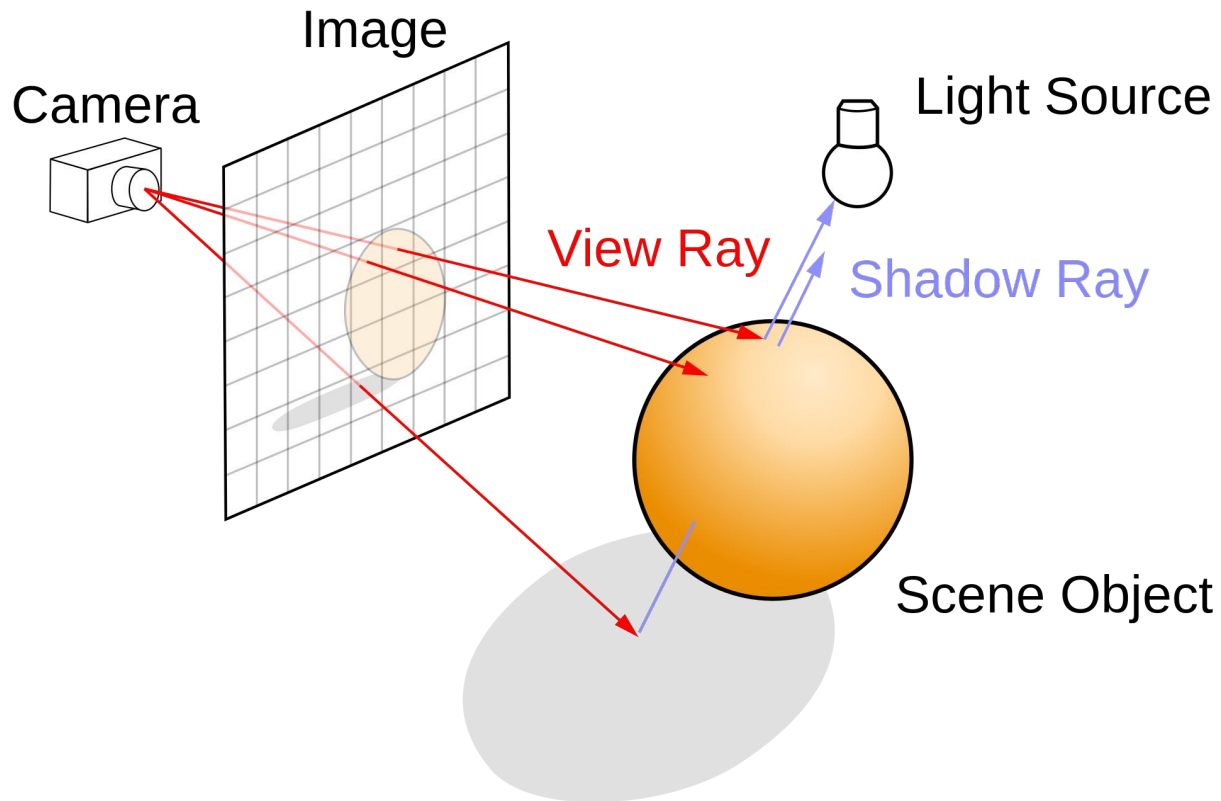
```
+ Image
# Shader Inputs
1 // The MIT License
2 // Copyright © 2013 Inigo Quilez
3 // Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associate
4
5 // A list of useful distance function to simple primitives. All
6 // these functions (except for ellipsoid) return an exact
7 // euclidean distance, meaning they produce a better SDF than
8 // what you'd get if you were constructing them from boolean
9 // operations.
10 //
11 // More info here:
12 //
13 // https://www.iquilezles.org/www/articles/distfunctions/distfunctions.htm
14
15 #if HW_PERFORMANCE==0
16 #define AA 1
17 #else
18 #define AA 2 // make this 2 or 3 for antialiasing
19 #endif
20
21 -----
22 float dot2( in vec2 v ) { return dot(v,v); }
23 float dot2( in vec3 v ) { return dot(v,v); }
24 float ndot( in vec2 a, in vec2 b ) { return a.x*b.x - a.y*b.y; }
25
26 float sdPlane( vec3 p )
27 {
28     return p.y;
29 }
30
31 float sdSphere( vec3 p, float s )
32 {
33     return length(p)-s;
34 }
35
36 float sdBox( vec3 p, vec3 b )
37 {
38     vec3 d = abs(p) - b;
39     return min(max(d.x,max(d.y,d.z)),0.0) + length(max(d,0.0));
40 }
41
42 Compiled in 0.0 secs (analyze) 11921 chars
```

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

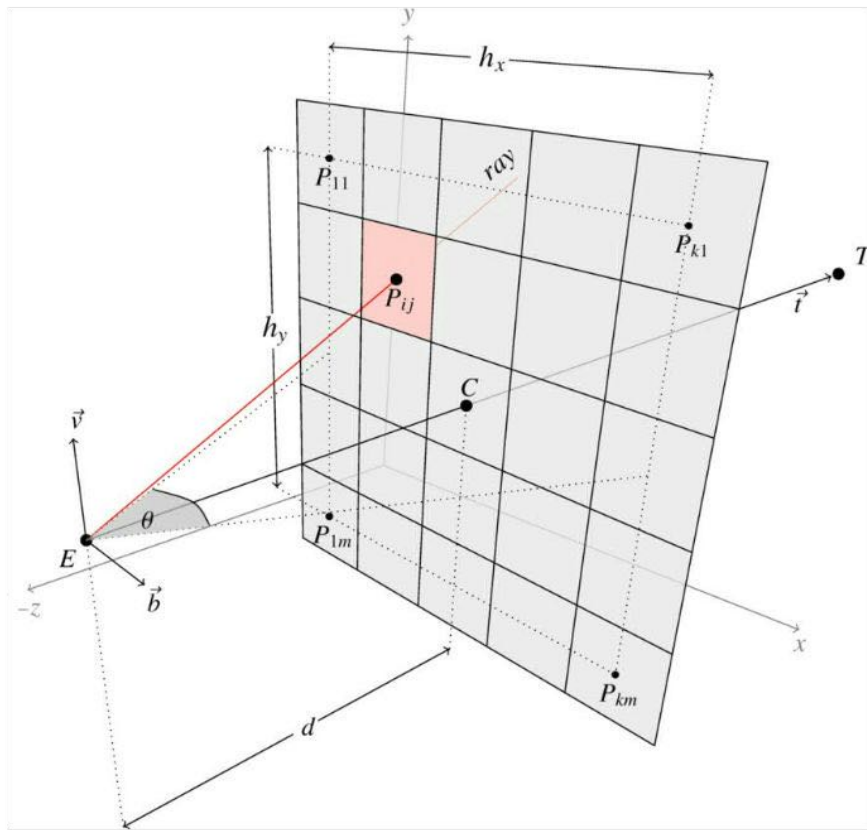
Part II:

Ray Marching

Ray Tracing



Shooting rays through camera pixels



Simplified notation

Shooting rays through camera pixels

Ray origin (eye): \mathbf{ro}

Ray direction (unit vector):

$$\mathbf{rd} = \text{normalize}(\mathbf{pix} - \mathbf{ro})$$

Ray equation:

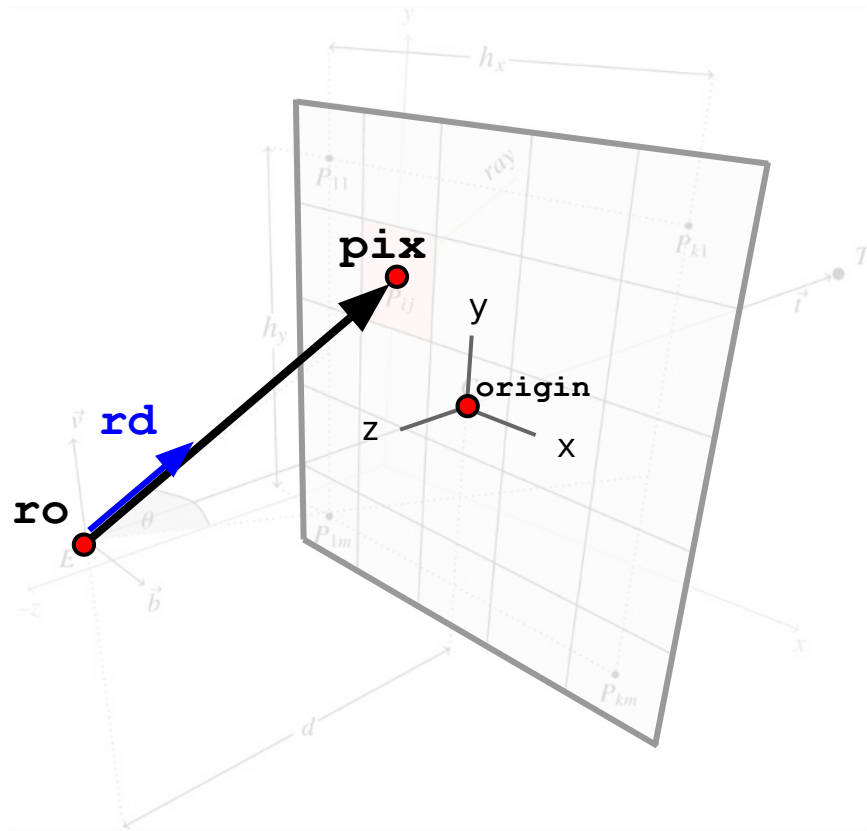
$$\mathbf{r}(t) = \mathbf{ro} + \mathbf{rd} * t, \quad t \geq 0$$

Goal:

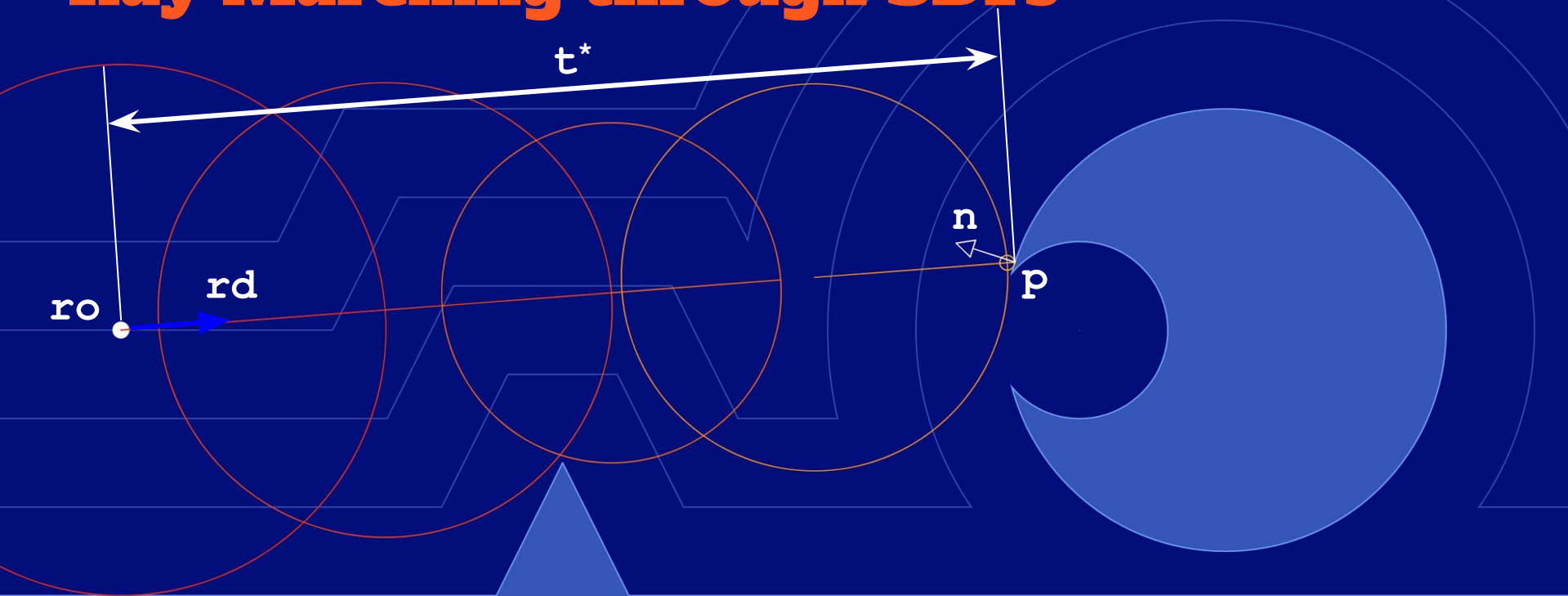
Find t^* at ray-surface intersection;

First t^* where $\text{sdf}(\mathbf{r}(t^*)) \approx 0$

Intersection point: $\mathbf{p} = \mathbf{r}(t^*)$

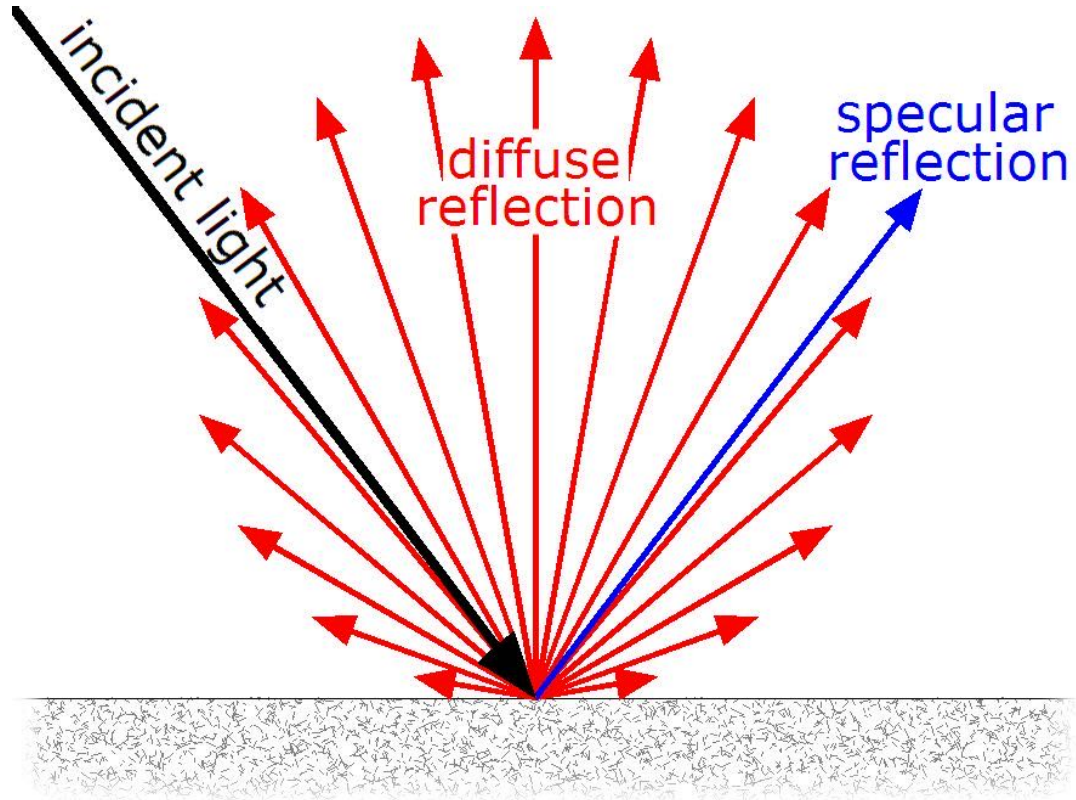


Ray Marching through SDEs



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

Surface Reflectance



Simplest reflectance model

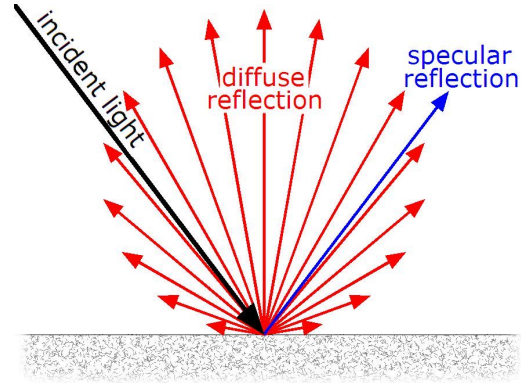
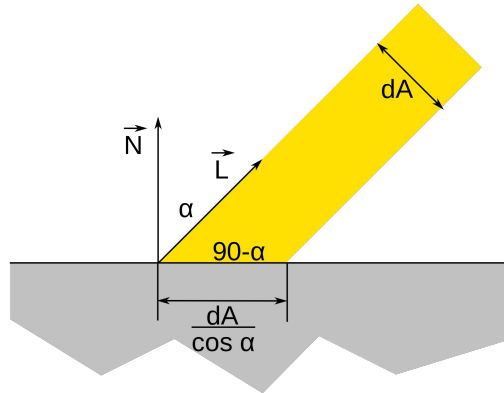
Diffuse Reflectance (Lambertian)

Models reflectance of a “matte” surface.
Appears the same from all view angles.

$$\mathbf{C}_{\text{out}} = (\mathbf{L} \cdot \mathbf{N}) \mathbf{C}_{\text{diffuse}} * \mathbf{I}_{\text{incident}}$$

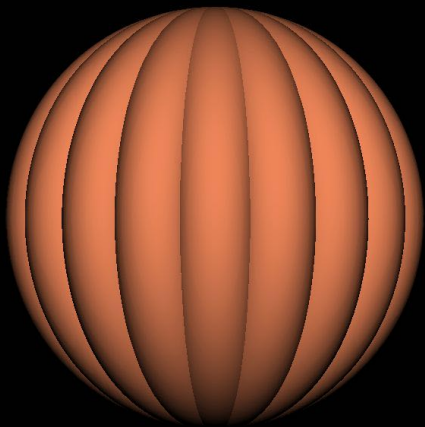
Cosine dependence on light direction,

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



Ray Marching Together

Live Coding



1285.94 59.9 fps 1080 x 607

Basic ray marcher for class

Views: 0, Tags: raymarching

Created by dJames in 2020-10-28

Just the basic stuff

unlisted

Save

```
+ Timage
4 Shader Inputs
1 // BASIC RAY-MARCHING CLASS DEMO.
2
3
4 float dot2( in vec2 v ) { return dot(v,v); }
5 float dot2( in vec3 v ) { return dot(v,v); }
6 float ndot( in vec2 a, in vec2 b ) { return a.x*b.x - a.y*b.y; }
7
8 vec3 rotate_y(vec3 v, float angle)
9 {
10     float ca = cos(angle); float sa = sin(angle);
11     return v*mat3(
12         +ca, +.0, -sa,
13         +.0,+1.0, +.0,
14         +sa, +.0, +ca);
15 }
16 vec3 rotate_x(vec3 v, float angle)
17 {
18     float ca = cos(angle); float sa = sin(angle);
19     return v*mat3(
20         +1.0, +.0, +.0,
21         +.0, +ca, -sa,
22         +.0, +sa, +ca);
23 }
24 vec3 rotate_z(vec3 v, float angle)
25 {
26     float ca = cos(angle); float sa = sin(angle);
27     return v*mat3(
28         +ca, -sa, +.0,
29         +sa, +ca, 0.,
30         +.0, +.0, 1.);
31 }
32
33
34 float sdSphere(vec3 p, float radius )
35 {
36     return length(p)-radius;
37 }
38
39 float sdTorus( vec3 p, vec2 t )
40 {
41     vec2 q = vec2(length(p)-t.x, t.y);
42     return t.z + sqrt(abs(q.x*q.x + q.y*q.y));
43 }
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025
1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2
```

Part III:

More on

Distance Fields

Combining SDFs

Primitive Combinations

- Union
- Subtraction
- Intersection
- +Smooth versions

Primitive combinations

Sometimes you cannot simply elongate, round or union a primitive, and you need to combine, carve or intersect basic primitives. Given the SDFs d_1 and d_2 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.



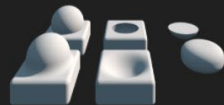
```
float opUnion( float d1, float d2 ) { min(d1,d2); }
```

```
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 seams 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`, `opSubtraction` 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](https://www.shadertoy.com/view/l3BW2) in this same site. You can code here: <https://www.shadertoy.com/view/l3BW2>



```
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); }
```


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 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 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/Ml3BDj>



```
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>

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 object 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);
}
```

Modifying SDFs

Primitive Deformations and Distortions

- Displacement
- Twist
- Bend

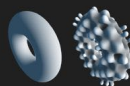
Deformations and distortions

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, orioning, 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 $\sin(20 \cdot p.x) \cdot \sin(20 \cdot p.y) \cdot \sin(20 \cdot p.z)$ as displacement pattern, but you can of course use anything you might imagine.



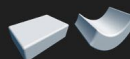
```
float opDisplace( in sdf3d primitive, in vec3 p )
{
    float d1 = primitive(p);
    float d2 = displacement(p);
    return d1+d2;
}
```

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



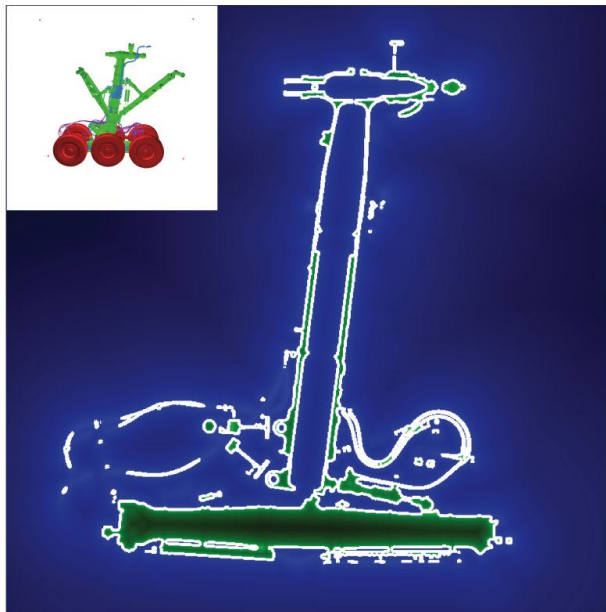
```
float opCheapBend( in sdf3d primitive, in vec3 p )
{
    const float k = 10.0; // or some other amount
    float c = cos(k*p.x);
    float s = sin(k*p.x);
    mat2 m = mat2(c,-s,s,c);
    vec3 q = vec3(m*p.xy,p.z);
    return primitive(q);
}
```

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

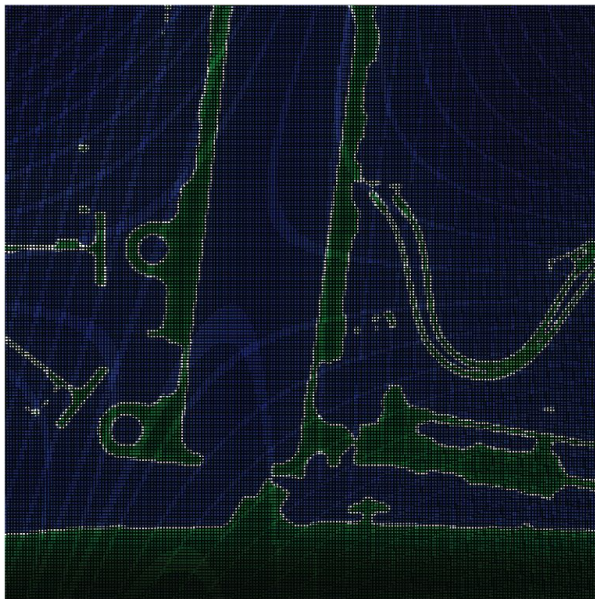
Numerical SDFs: Sampled on Grids

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

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*.

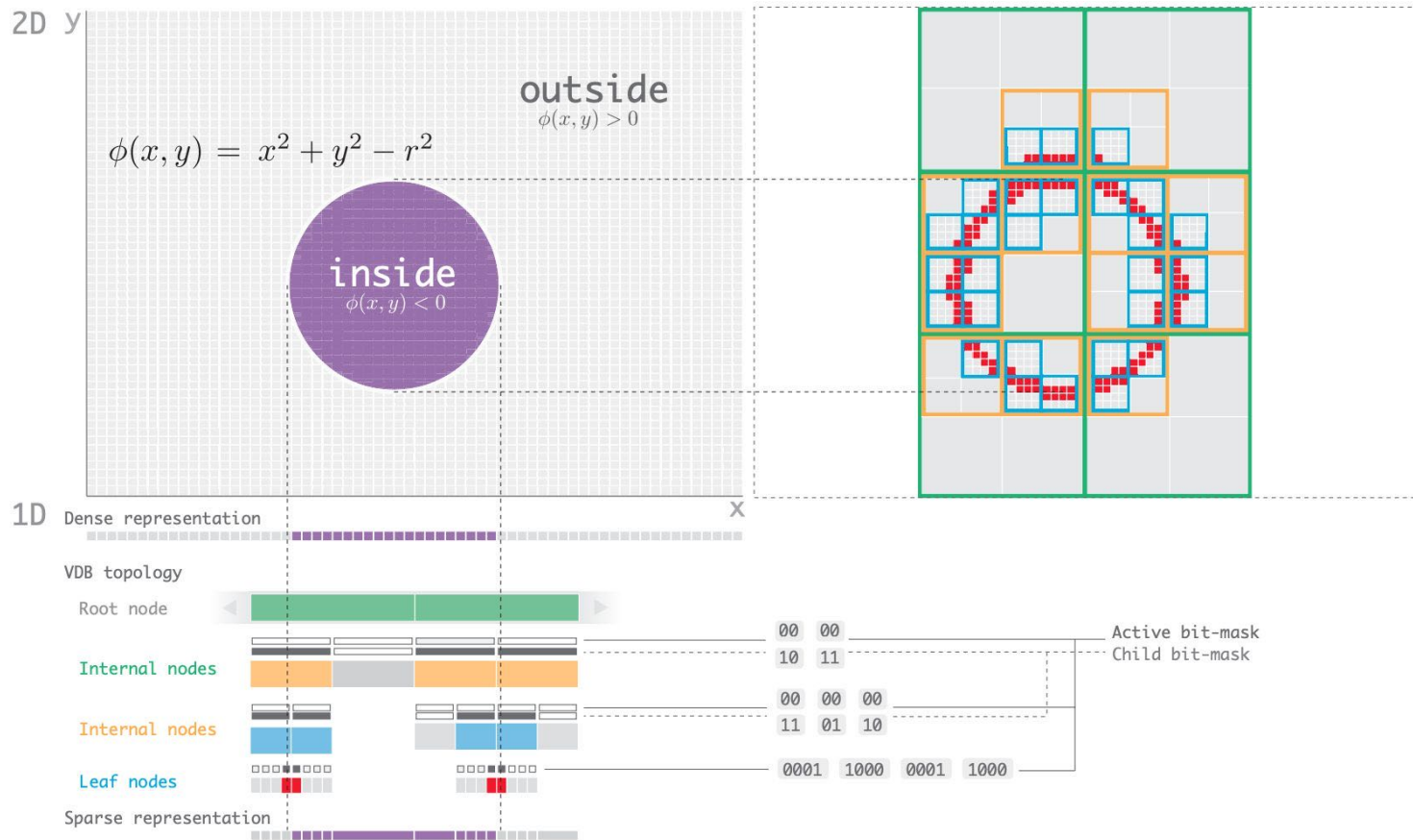


Fig. 3. Illustration of a narrow-band level set of a circle represented in, respectively, a 1D and 2D VDB. Top Left: The implicit signed distance, that is, level set, of a circle is discretized on a uniform dense grid. Bottom: Tree structure of a 1D VDB representing a single y -row of the narrow-band 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.

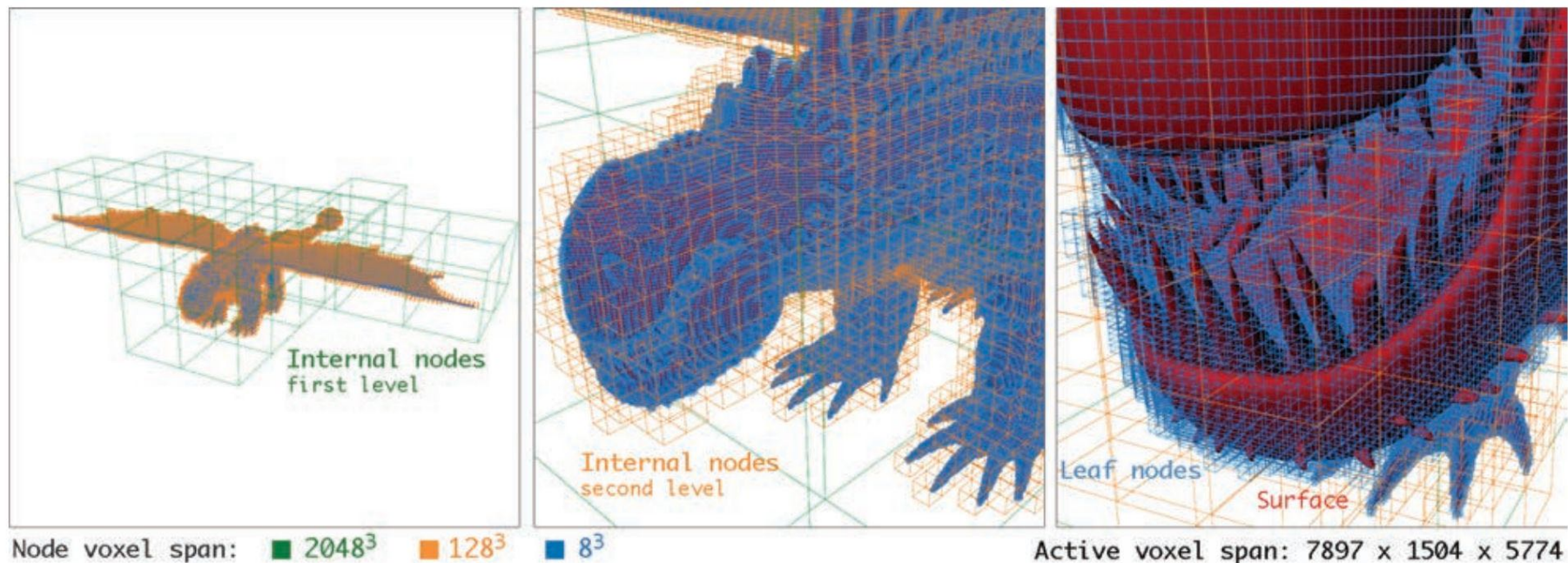
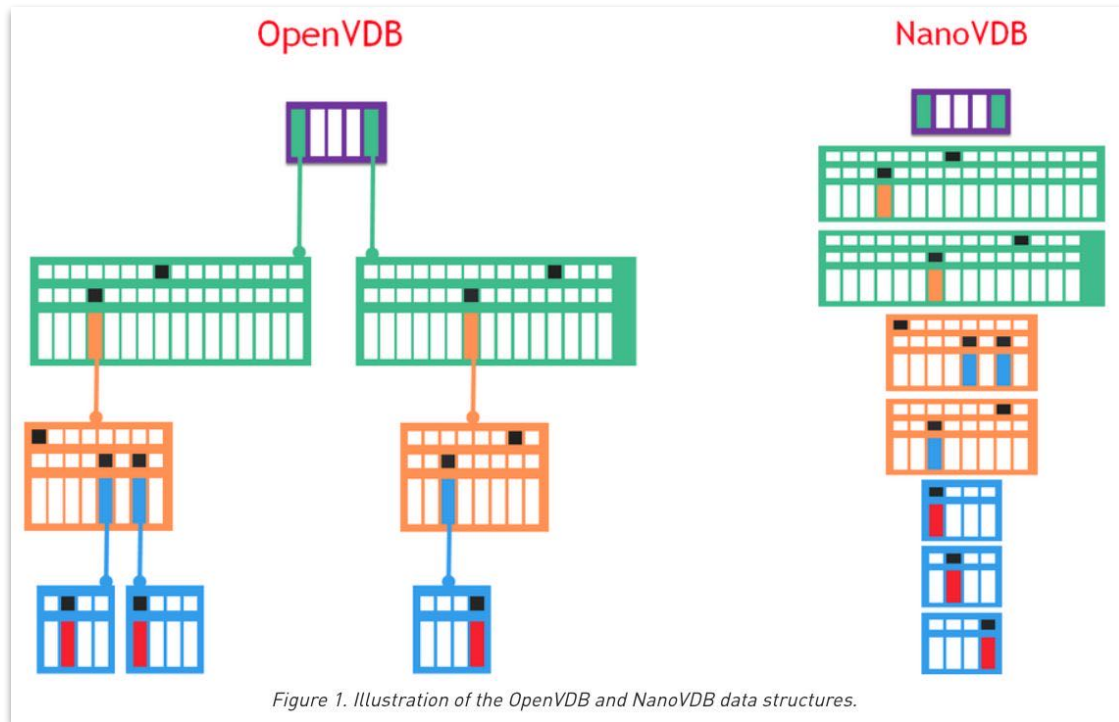


Fig. 4. High-resolution VDB created by converting polygonal model from *How To Train Your Dragon* to a narrow-band level set. The bounding resolution of the 228 million active voxels is $7897 \times 1504 \times 5774$ and the memory footprint of the VDB is 1GB, versus the $\frac{1}{4}$ TB for a corresponding 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

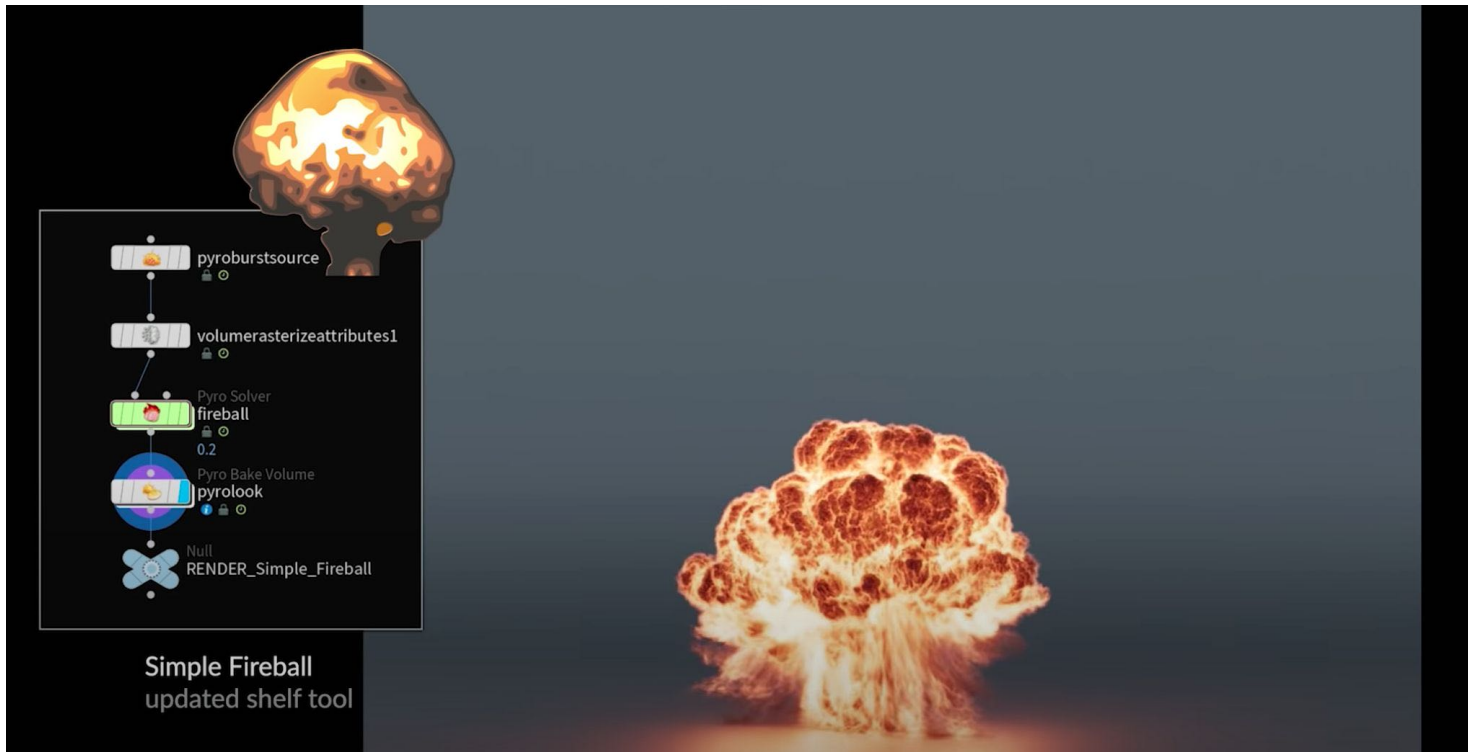
nanoVDB



<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

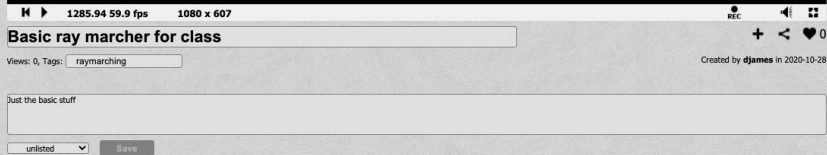
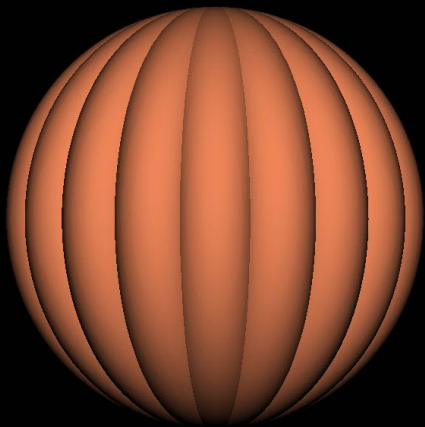
WedNov18

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)



3594.53 37.6 fps 1080 x 607

Basic ray marcher for class

Views: 27, Tags: raymarching

All the basic stuff.

unlisted Save

```
459     return res;
460 }
461
462 vec3 render(in vec3 ro, in vec3 rd)
463 {
464     vec3 col = vec3(0.,0.,0.); //background
465     vec2 mouse = iMouse.xy/iResolution.xy;
466
467     vec4 ray = raymarchV4(ro, rd); //compute distance along ray to surface
468     float t = ray.x;
469     vec3 Cd = ray.yzw; // can passthru other shading values, e.g., objectID, but we did color.
470
471     if (t > 0.0) { //hit surface --> shade it:
472
473         vec3 p = ro + rd*t; // point on surface
474         vec3 N = calcNormal(p); // sdf normal
475
476         // DETERMINE MATERIAL COLOR: (todo: all orange for now)
477         //vec3 Cd = vec3(231./255., 111./255., 3./255.); // diffuse color
478
479         // DIRECTIONAL LIGHT:
480         vec3 L = normalize(ro + 30.*vec3(mouse.x-0.5, mouse.y-0.5, 0.) - p); // light at eye (safe!)
481         vec3 CL = vec3(1.); // directional light color
482         float LdotN = clamp(dot(L,N), 0., 1.);
483         //float shadL = shadow(p, L, 0.01, 20.);
484         float sshadL = softshadow(p, L, 0.01, 21., 2.);
485         col = Cd * CL * LdotN * sshadL; // * occ;
486
487         // CANDLE LIGHT (#2):
488         //vec3 dp = 2.*vec3(0., mouse.y-0.5, mouse.x-0.5);
489         vec3 posL2 = pumpkinCenter + 0.15*vec3(sin(20.*iTime), cos(7.*iTime), cos(14.*iTime));
490         vec3 L2 = normalize(posL2 - p);
491         vec3 CL2 = vec3(1.,1.,2.); // candle light color
492         float L2dotN = clamp(dot(L2,N), 0., 1.);
493         //float shadL2 = shadow(p, L2, 0.01, length(posL2-p));
494         float sshadL2 = softshadow(p, L2, 0.01, length(posL2-p), 8.);
495         col += Cd * CL2 * L2dotN * sshadL2; // * occ;
496     }
497
498     return col;
499 }
```

Compiled in 0.0 secs (analyze)

8341 chars

LV ?

<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>