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
  - Wavelet noise
  - Worley noise
  - Etc.
- Many applications!
<table>
<thead>
<tr>
<th>Noise</th>
<th>UNIFIED NOISE</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="https://www.sidefx.com/houdini/nodes/vop/unifiednoise.html">https://www.sidefx.com/houdini/nodes/vop/unifiednoise.html</a></td>
<td></td>
</tr>
</tbody>
</table>
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): \[ X_{n+1} = (aX_n + c) \mod m \]
    - 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"
- \( c, 0 \leq c < m \) — 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
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

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

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

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

**Cubic blending function**

**Simple Hack:**
- Assume gradients are zero at vertices.
- Just blend data using Hermite functions!

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

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

<table>
<thead>
<tr>
<th>$h_{00}(t)$</th>
<th>$h_{10}(t)$</th>
<th>$h_{01}(t)$</th>
<th>$h_{11}(t)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2t^3 - 3t^2 + 1$</td>
<td>$t^3 - 2t^2 + t$</td>
<td>$-2t^3 + 3t^2$</td>
<td>$t^3 - t^2$</td>
</tr>
<tr>
<td>$(1 + 2t)(1 - t)^2$</td>
<td>$t(1 - t)^2$</td>
<td>$t^2(3 - 2t)$</td>
<td>$t^2(t - 1)$</td>
</tr>
<tr>
<td>$B_0(t) + B_1(t)$</td>
<td>$1/3 \cdot B_1(t)$</td>
<td>$B_3(t) + B_2(t)$</td>
<td>$-1/3 \cdot B_2(t)$</td>
</tr>
</tbody>
</table>
Simplest noise

Value Noise

Random Values (at vertices)

interpolate

Blended (cubic)
Blending functions

Blending Vertex Values

Graphtoy v0.3 by Inigo Quilez (thanks Rafael Couto)

f1(x) = x
f2(x) = x*x*(3.0-2.0*x)

https://graphtoy.com
Multiple octaves

**Fractal Noise (a.k.a. “turbulence”)**

\[
\text{noise}(p) + \frac{1}{2}\text{noise}(2p) + \frac{1}{4}\text{noise}(4p) + \ldots
\]

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

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

- Special case of Fractal Brownian motion (fBm) noise
- See Inigo Quilez’s [fBm tutorial](https://inigoquilez.com/articles/fbm)
Demo

Value Noise (2D)

https://www.shadertoy.com/view/Isf3WH
Value Noise (3D)

https://www.shadertoy.com/view/4sfGzS
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.: 
\[
g_x \, \Delta x + g_y \, \Delta y = g \cdot f
\]
where \( f=(\Delta x, \Delta y) \) is the vertex delta.

Blend all vertex values across cell like before.
Better noise

Gradient Noise

Given point, p
Better noise

Gradient Noise

Given point, $p$
Get containing cell: $i = \text{floor}(p)$
Better noise

Gradient Noise

Given point, \( p \)
Get containing cell: \( i = \text{floor}(p) \)
Extract \textbf{vertex offsets}, \( f = \text{fract}(p) \)
Better noise

**Gradient Noise**

Given point, $p$
Get containing cell: $i=\text{floor}(p)$
Extract vertex offsets, $f=\text{fract}(p)$
Lookup random vertex gradient, $g_i$, $i=1..4$. 

[Diagram showing a point $p$ with vertex offsets $f_i$ and random vertex gradients $g_i$.]
**Better noise**

**Gradient Noise**

Given point, \( p \)

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

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

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

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

**Gradient Noise**

Given point, \( p \)
Get containing cell: \( i=\text{floor}(p) \)
Extract **vertex offsets**, \( f=\text{fract}(p) \)
Lookup **random vertex gradient**, \( g_i \), \( i=1..4 \).
Dot vertex offset with gradient: \( (f_i \cdot g_i) \), \( i=1..4 \).
Blend vertex values across cell:

\[
\text{Noise} = \sum_{i=1..4} w_i (f_i \cdot g_i)
\]
Demo

Gradient Noise 2D

https://www.shadertoy.com/view/XdXGW8
The classic noise function

**Perlin Noise** *(SIGGRAPH 1985)*

- Famous instance of gradient noise
- 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
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 $G[n]$
- $G = G[ ( i + P[ (j + P[k]) \mod n ] ) \mod n ]$

Some example implementations:
[https://rosettacode.org/wiki/Perlin_noise#C](https://rosettacode.org/wiki/Perlin_noise#C)
\[
\text{noise}(p) + \frac{1}{2} \text{noise}(2p) + \frac{1}{4} \text{noise}(4p) + \ldots
\]
$|\text{noise}(p)| + \frac{1}{2}|\text{noise}(2p)| + \frac{1}{4}|\text{noise}(4p)| + \ldots$
\[ \sin(x + |\text{noise}(p)| + \frac{1}{2}|\text{noise}(2p)| + \frac{1}{4}|\text{noise}(4p)| + \ldots) \]
colorMap(y + F(x,y,z))
Further reading

**Noise Tricks**

See “Texture generation using random noise” by Lode

- [https://lodev.org/cgtutor/randomnoise.html](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
Improved Perlin Noise

Smotherer interpolants & better gradients.
**Improved Perlin Noise**

**Smother interpolant:** Cubic ➔ Quintic blend function:

\[ x^2(3.0-2.0^x) \quad \rightarrow \quad x^2x^3(x^6-15.) + 10. \]

**GraphToy Plot**

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

**Explanatory article:**
Improved Perlin Noise

Smotherer interpolants & better gradients.

Figure 2a: High-frequency Noise, with old gradient distributions
Figure 2b: High-frequency Noise, with new gradient distributions
public final class ImprovedNoise {
  static double 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),
    v = fade(y),
    w = fade(z);
    int A = p[X+Y], BA = p[A+Z], BB = p[B+Z];  // HASH COORDINATES OF
    int B = p[X+Y], BA = p[A+Z], BB = p[B+Z];  // THE 8 CUBE CORNERS,
    return lerp(lerp(lerp(u, lerpx, grad[AA]), x, y, z), // AND ADD
               lerpx, grad[BA]), x, y, z),  // BLENDED
           lerpx, grad[AB]), x, y, z),  // RESULTS
               grad[BB]), x, y, z)); // FROM 8
    return 4 - (lerpelru(lerpelru(lerpelru(lerpelru(lerpelru(lerpelru(x, y, z), 1), 1), 1), 1), 1), 1); // CORNERS
  }

  static double double fade(double t) { return t * t * t * (t * (t * 6 - 15) + 10); }
  static double double lerpx(double x, double a, double b) { return a + t * (b - a); }
  static double 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<12 ? h<14 : x ;
    return ((hash & 16) == 0 ? u : u - u) + ((hash & 8) == 0 ? v : -v);
  }


  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

https://www.shadertoy.com/view/XsI3DI
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.

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.
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

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


Archipelagos/Islands/Glacier.png
Terraria
Breakout Discussion

Terrain Generation in Terraria

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

Noisy Sketches

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

Figure 2: A hit and a miss.

https://www.shadertoy.com/view/IsIXD8
Part I:

Implicit Geometry
**Distance Fields**

**Distance Fields:** Encode distance from a **point**, \( x \), to an object, \( O \):
\[
d(x) = \min ||x-y||_2 \quad \text{over all } y \text{ on } O.
\]

**Signed Distance Fields:** Sign indicates if inside/outside object:
\[
\begin{align*}
    d(x) &> 0 : \text{Outside object} \\
    d(x) &= 0 : \text{On object} \\
    d(x) &< 0 : \text{Inside object}
\end{align*}
\]

**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](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:** \( I(x) = \text{step}(0, -d(x)) \)
- 1 inside, \( d \leq 0 \)
- 0 outside, \( d > 0 \)

**Smooth-step indicator function:** \( \text{smoothstep}(-\epsilon, \epsilon, -d) \)

http://www.iquilezles.org/apps/graphtov/?f1(x)=\text{step}(0,0.2-x)&f2(x)=\text{smoothstep}(-0.1,0.1,0.2-x)
**Color Mixing**

"On top":

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

Additive blending:

\[
\text{color}(x) = \sum_i I_i(x) \text{color}_i
\]

Subtracting blending:

\[
\text{color}(x) = \text{white} - \sum_i I_i(x) \text{color}_i
\]
Recall: Intro to GLSL Demo #20

2D Indicator Functions

Smooth-step indicator function, \( l(r) \): \( \text{disk}(r, \text{center}, \text{radius}) \)

```cpp
// 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
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)),
        length(max(vec3(q.x,p.y,q.z),0.0)),
        length(max(vec3(q.x,q.y,p.z),0.0)));
}

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

Youtube Tutorial on formula derivation: https://www.youtube.com/watch?v=62-pRvZs55c

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
Ray Tracing

Image

Camera

Light Source

View Ray

Shadow Ray

Scene Object

Shooting rays through camera pixels
Simplified notation

**Shooting rays through camera pixels**

Ray origin (eye): \(ro\)

Ray direction (unit vector):
\[
rd = \text{normalize}(\text{pix}-ro)
\]

Ray equation:
\[
r(t) = ro + rd * t, \quad t \geq 0
\]

Goal:
Find \(t^*\) at ray-surface intersection;
First \(t^*\) where \(sdf(r(t^*)) \approx 0\)

Intersection point: \(p = r(t^*)\)
Ray Marching through SDFs

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](https://www.iquilezles.org/www/articles/normalsSDF/normalsSDF.htm)
Surface Reflectance

incident light

diffuse reflection

specular reflection

Simplest reflectance model

**Diffuse Reflectance (Lambertian)**

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

\[
C_{\text{out}} = (\mathbf{L} \cdot \mathbf{N}) \; C_{\text{diffuse}} \; \cdot \; I_{\text{incident}}
\]

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
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.

```c
float opUnion( float d1, float d2 ) { return 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, 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 on this site. You can code here: https://www.shadertoy.com/view/53BW2

```c
float opSmoothUnion( float d1, float d2, float k ) {
    float h = clamp(0.5 + 0.5*(d2-d1)/k, 0.0, 1.0);
    return mix(d1, d2, 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(d1, d2, 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 an exact Euclidean metric to them, which is an important property since SDFs with unaltered Euclidean metric allow for faster ray marching.

**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 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/MJDR

```c
float elongate( in float3 p, in vec3 v, in vec3 h )
{
    vec3 q = p - clamp( p, -h, h );
    return primitive( q );
}
```

The reason I provide two implementations is the following. For 2D elongations, the first function works perfectly and gives exact exterior and interior distances. However, the first implementation produces a small set of very distances inside the volume by 3D 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 above 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 cube in the image below. If you happen to be interested in preserving the overall volume of the shape, most of the time 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/PS3E6S

```c
float roundb( in float3 p, in float rad )
{
    return primitive( p ) - rad
}
```

**Onion - exact**

For creating 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/MJ4D

```c
float onionb( in float3 p, in float rad )
{
    return primitive( p )
}
```
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 translations don’t compress nor dilate space, all we need to do is simply 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 4x4 matrix for example, or as a quaternion and a vector), and that it does not contain any scaling factors in it.

```c
vec3 op2t3( in vec3 p, in transform t, in sdfs primitive )
{
    return primitive invert(t)(p);
}
```

Scale - exact

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

```c
float opscale( in vec3 p, in float s, in sdfs primitive )
{
    return primitive(p/s);
}
```

Symmetry - bound and exact

Symmetry is useful since many things around us are symmetric; from humans, animals, vehicles, instruments, furniture, … Sometimes, 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.

```c
float op2sym3( in vec3 p, in sdfs primitive )
{
    p.x = abs(p.x);
    return primitive(p);
}
```
```c
float op2sym4( in vec3 p, in sdfs primitive )
{
    p.x = abs(p.x);
    return primitive(p);
}
```
Modifying SDFs

Primitive Deformations and Distortions

- Displacement
- Twist
- Bend

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-accumulative 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 (dilation, rounding, erosion, union) or light bounded function (intersection, subtraction) and then only apply as seal if a distortion or displacement is 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 \( \sin(2\pi x) \cdot \sin(2\pi y) \cdot \sin(2\pi z) \) as displacement pattern, but you can of course use anything you might imagine.

```cpp
float optDisplaced( in edf3d primitive, in vec3 p )
{
  float dl = primitive(p);
  float dl = displacement(p);
  return dl+dl;
}
```

Twist

```cpp
float optTwist( in edf3d primitive, in vec3 p )
{
  const float k = 10.0f; // or some other amount
  float c = cos(k*p.y);
  float s = sin(k*p.y);
  mat3 m = mat3(c,-s,0);
  vec3 q = vec3(m*vec3(p.x,p.y,p.z));
  return primitive(q);
}
```

Bend

```cpp
float optBending( in edf3d primitive, in vec3 p )
{
  const float k = 10.0f; // or some other amount
  float c = cos(k*p.x);
  float s = sin(k*p.x);
  mat3 m = mat3(c,-s,0);
  vec3 q = vec3(m*vec3(p.x,p.y,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

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

(c) Landing gear 1024x1024x1024 signed distance field
(d) Detailed view of landing gear distance field

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


https://www.openvdb.org/
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.

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**
- **WedNov04**

- **MonNov09**: MMO Games
  - Build a simple multiplayer game using socket.io in OpenProcessing.
- **WedNov11**
- **MonNov16**
- **WedNov18**: Due + Demo on WedNov18
Part IV:

Fin-i’-shing the Jack-o’-lantern
Ray Marching Together
Live Coding (Part II)

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)

https://www.shadertoy.com/view/wdKyD3
Ray Marching Together

Live Coding (Final)

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