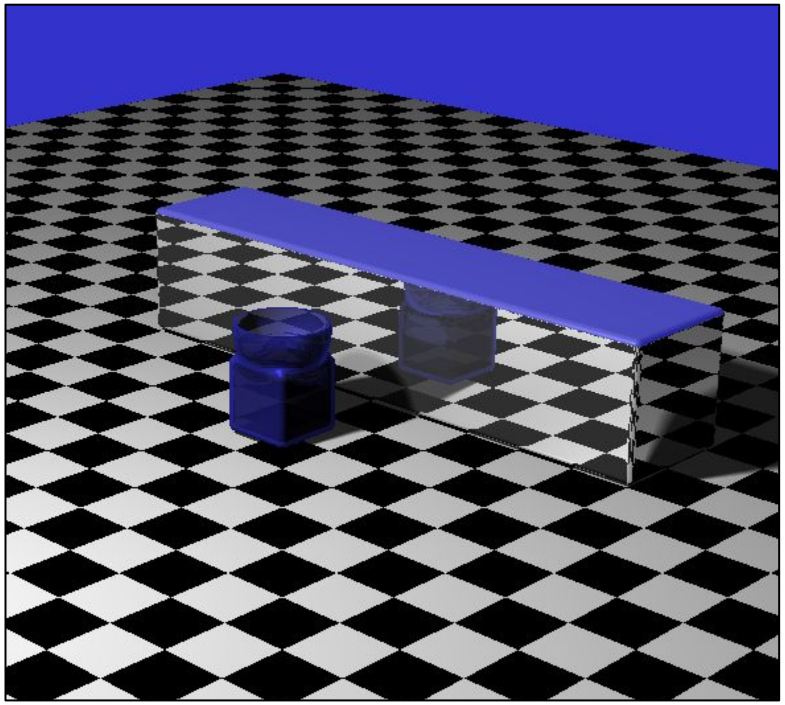
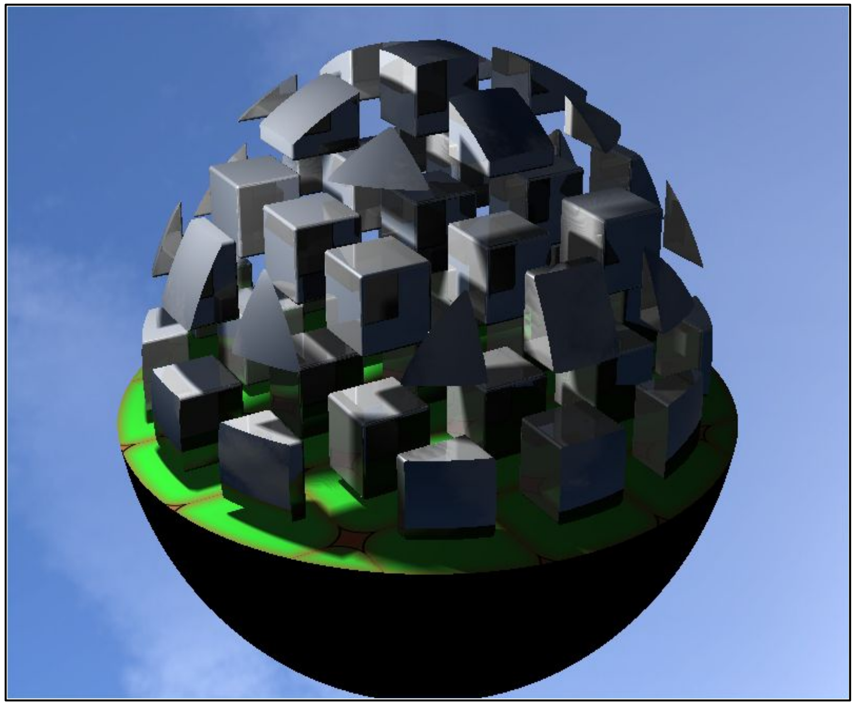
Further Graphics





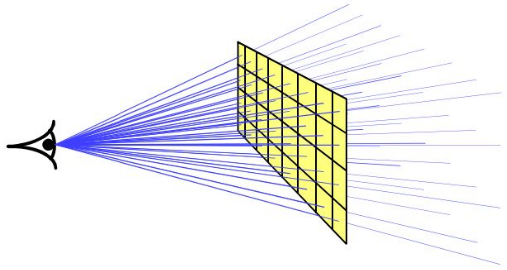
Ray Marching and Signed Distance Fields

Alex Benton, University of Cambridge – alex@bentonian.com

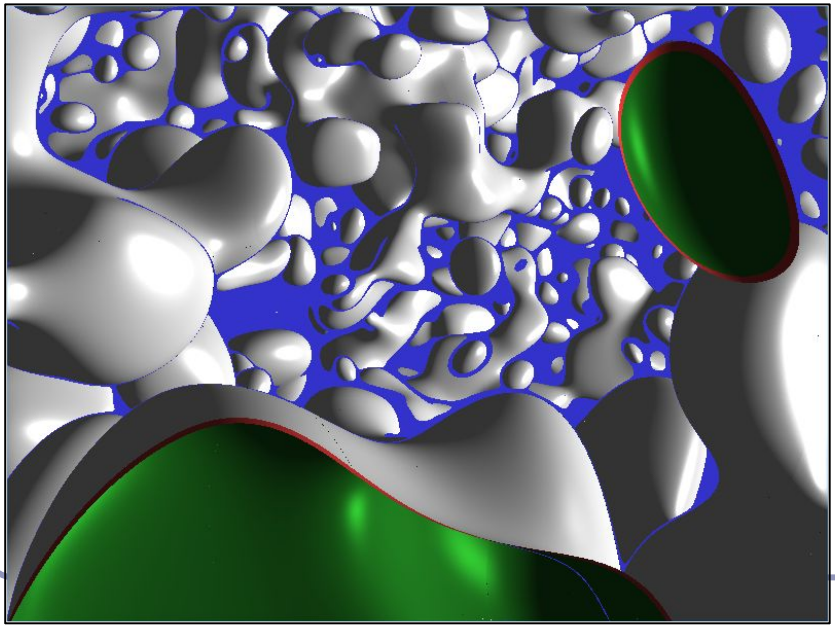
Supported in part by Google UK, Ltd

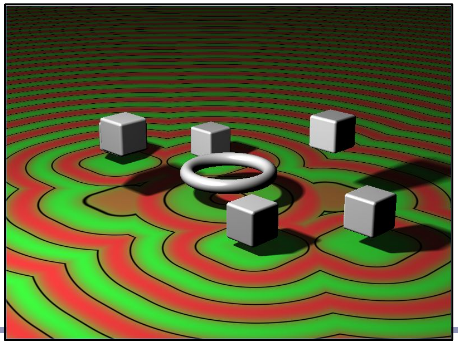
GPU Ray-tracing

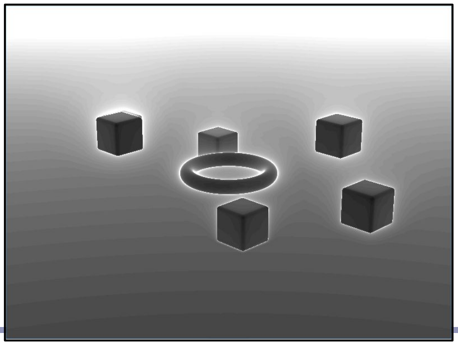
Ray tracing 101: “Choose the color of the pixel by firing a ray through and seeing what it hits.”



Ray tracing 102: “Let the pixel make up its own mind.”

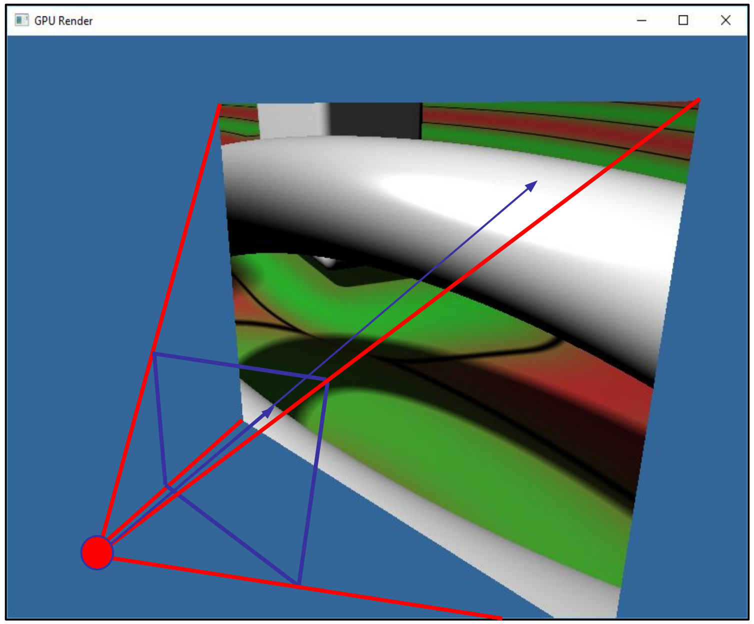




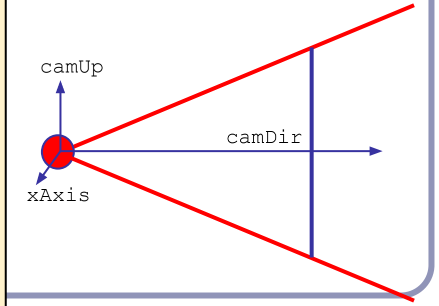


GPU Ray-tracing

1. Set up OpenGL with minimal geometry, a single quad, flat to the quad
2. Vertex shader: minimal (no transforms, just pass through)
3. Fragment shader:
   a. Compute the ray from the eye through the pixel
   b. Intersect ray against scene
   c. Shade pixel to color of hit object, with illumination, reflections, transparency...



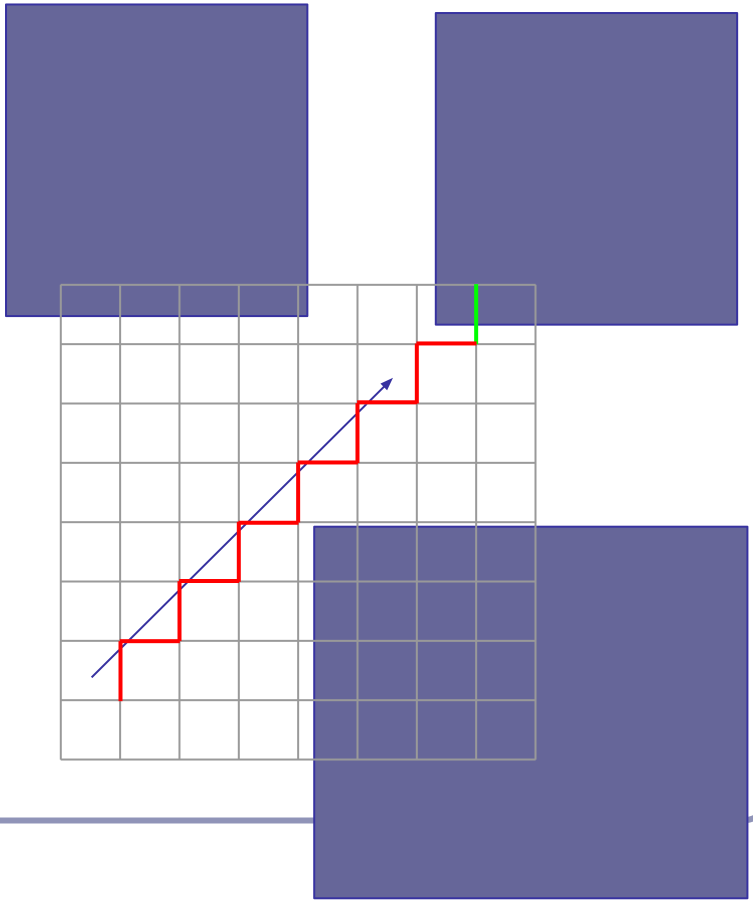
```
// Find ray from camera through pixel
// (Camera screen depth set to 5)
vec3 getRayDir(vec3 camDir, vec3 camUp, vec2 pixel) {
  vec3 xAxis = normalize(cross(camDir, camUp));
  vec2 p = 2.0 * pixel - 1.0;
  return normalize(p.x * xAxis + p.y * camUp + 5 * camDir);
}
```



Ray-marching

An alternative to classic ray-tracing is *ray-marching*, in which we take a series of finite steps along the ray until we strike an object or exceed the number of permitted steps.

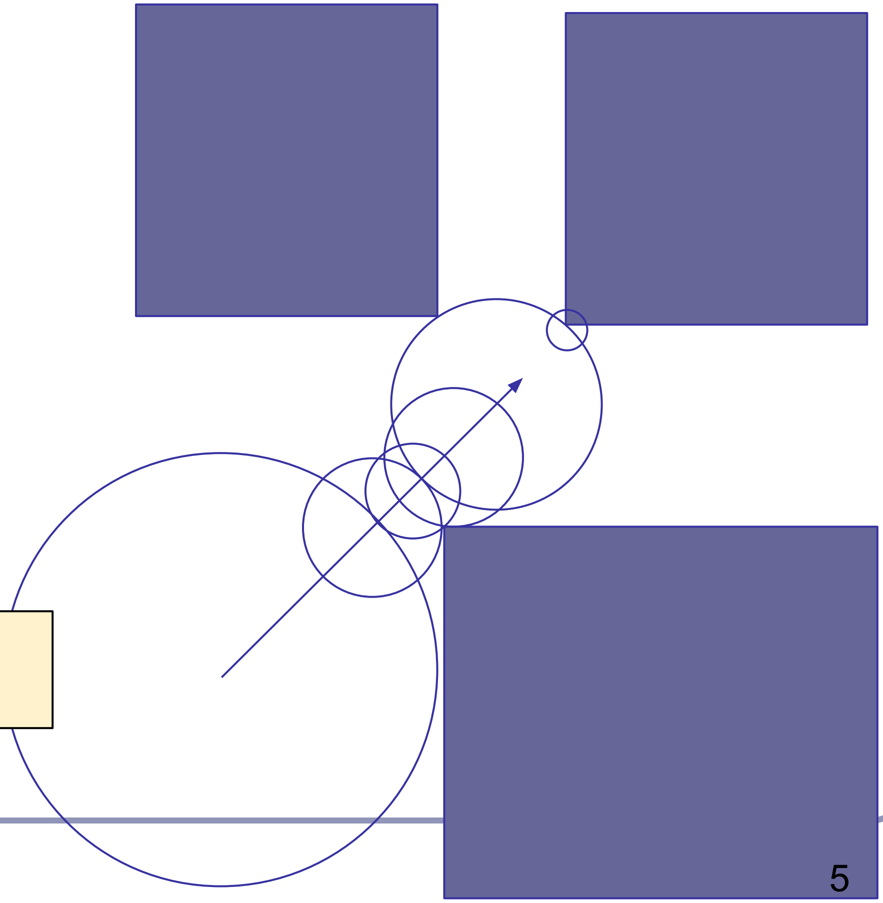
- Scene objects only need to answer, "has this ray hit you? y/n"
- Great solution for data like height fields
- Caution:
  - Too large a step size can lead to lost intersections (step over the object)
  - Too small a step size can lead to GPU churn and wasted cycles



GPU Ray-marching with Signed Distance Fields

Ray-marching can be dramatically improved, to impressive realtime GPU performance, using *signed distance fields*:

1. Fire ray into scene
2. At each step, measure distance field function: *d(p)* = [distance to nearest object in scene]
3. Advance ray along ray heading by distance *d*, because the nearest intersection can be no closer than *d*



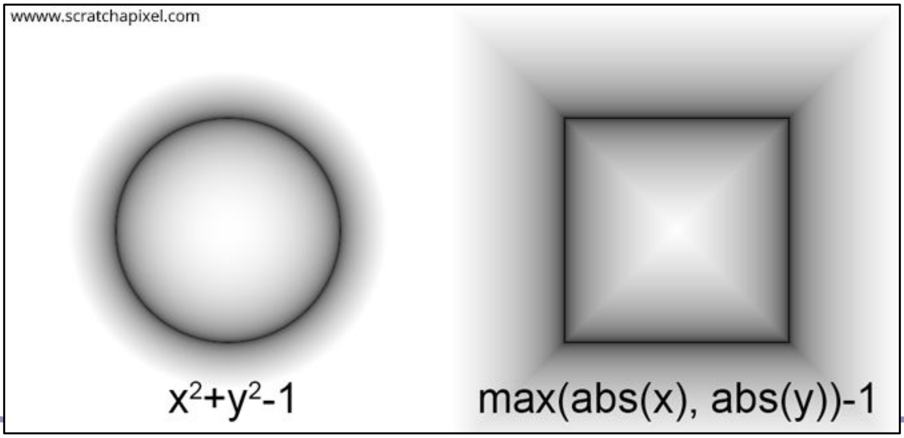
This is also sometimes called 'sphere tracing'. Early paper: http://graphics.cs.illinois.edu/sites/default/files/rtqjs.pdf

Signed distance fields

An *SDF* returns the minimum possible distance from point *p* to the surface it describes.

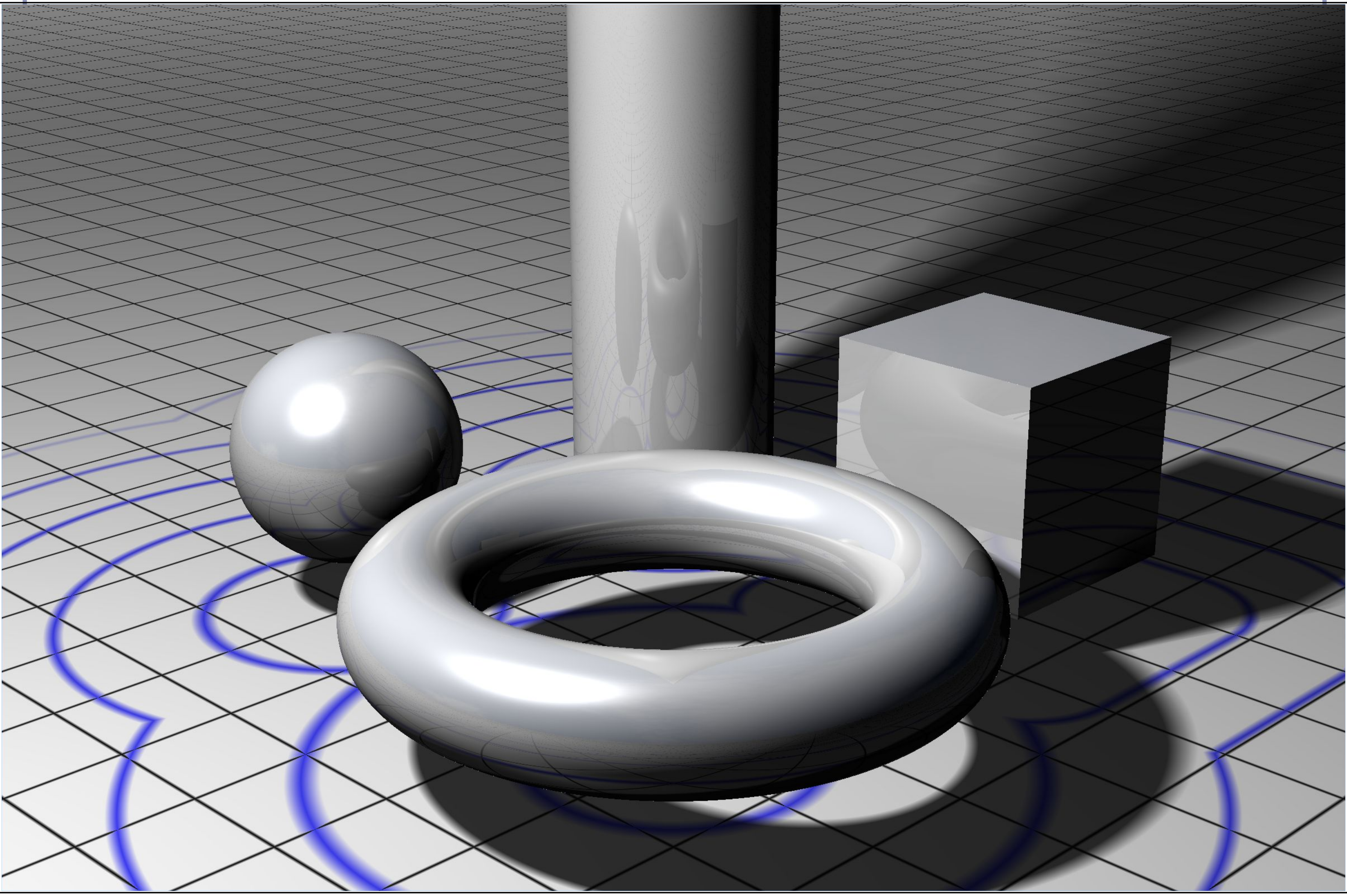
The sphere, for instance, is the distance from *p* to the center of the sphere, minus the radius.

Negative values indicate a sample inside the surface, and still express absolute distance to the surface.



```
float sphere(vec3 p, float r) {
  return length(p) - r;
}

float cube(vec3 p, float r) {
  return max(max(abs(p.x),
      abs(p.y)), abs(p.z)) - r;
}

float cylinderY(vec3 p, float r)
{
  return length(p.xz) - r;
}

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

https://www.scratchapixel.com/lessons/advanced-rendering/rendering-distance-fields



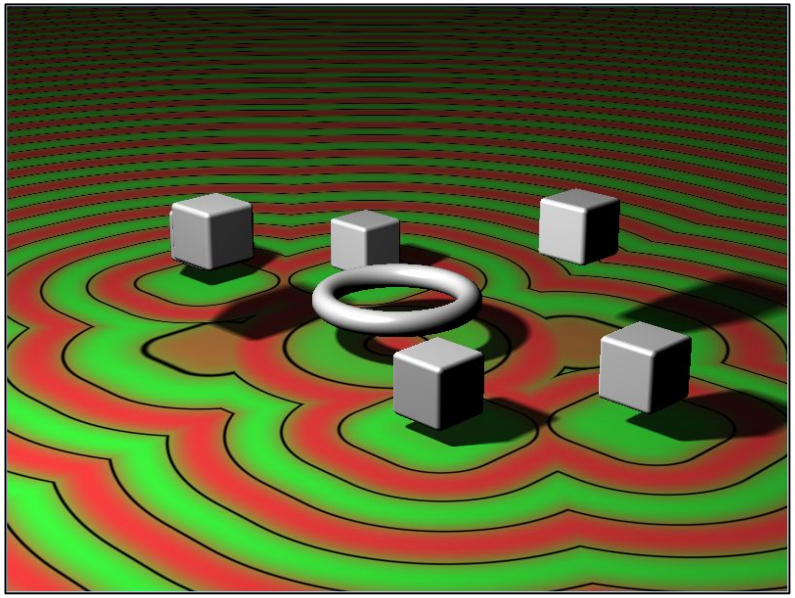
The signed distance fields raymarching algorithm in glsl

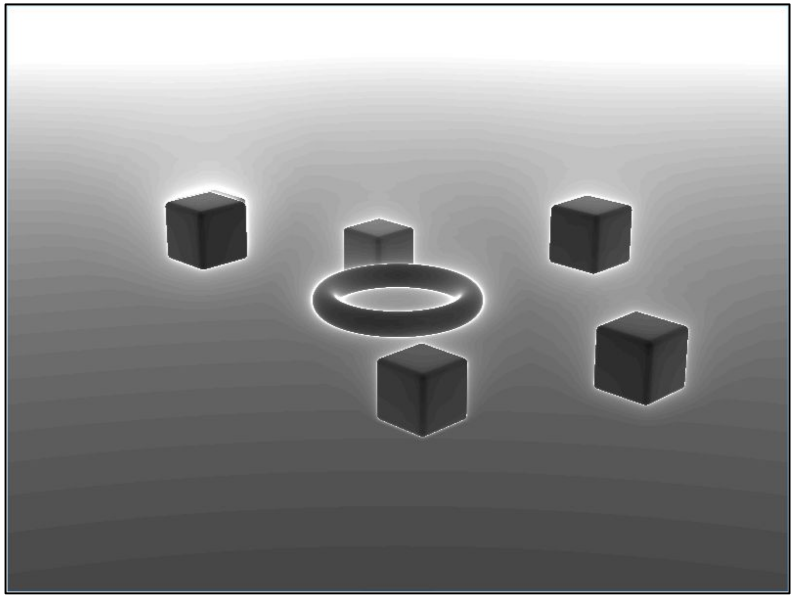
```
vec3 raymarch(vec3 pos, vec3 raydir) {
  int step = 0;
  float d = getSdf(pos);

  while (abs(d) > 0.001 && step < 50) {  // Step limit
    pos = pos + raydir * d;
    d = getSdf(pos);  // Return e.g. sphere(pos)
    step++;
  }

  return
      (step < MAX) ? illuminate(pos, rayorig) : background;
}
```

Visualizing step count

Final image

Distance field



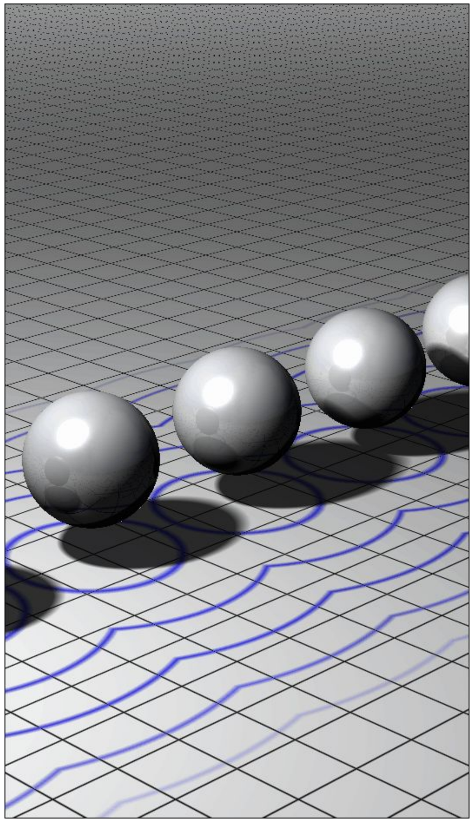


Transforming SDF geometry

To rotate, translate or scale an SDF model, apply the inverse transform to the input point within your distance function.

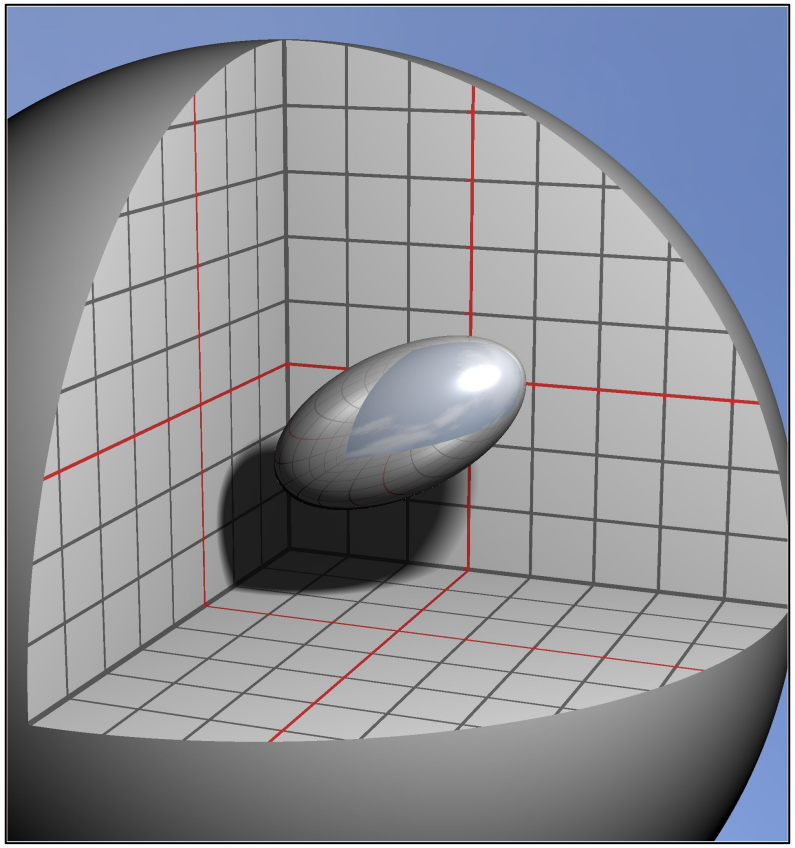
```
float sphere(vec3 pt, float radius) {
  return length(pt) - radius;
}

float f(vec3 pt) {
  return sphere(pt - vec3(0, sin(time),
0));
}
```

Ex: adding `(1,-2,3)` renders a sphere centered at `(-1,2,-3)`.



Transforming SDF geometry

```
float fScene(vec3 pt) {

  // Scale 2x along X
  mat4 S = mat4(
      vec4(2, 0, 0, 0),
      vec4(0, 1, 0, 0),
      vec4(0, 0, 1, 0),
      vec4(0, 0, 0, 1));

  // Rotation in XY
  float t = sin(time) * PI / 4;
  mat4 R = mat4(
      vec4(cos(t),  sin(t), 0, 0),
      vec4(-sin(t), cos(t), 0, 0),
      vec4(0,       0,      1, 0),
      vec4(0,       0,      0, 1));

  // Translate to (3, 3, 3)
  mat4 T = mat4(
      vec4(1, 0, 0, 3),
      vec4(0, 1, 0, 3),
      vec4(0, 0, 1, 3),
      vec4(0, 0, 0, 1));

  pt = (vec4(pt, 1) * inverse(S * R * T)).xyz;

  return sdSphere(pt, 1);
}
```



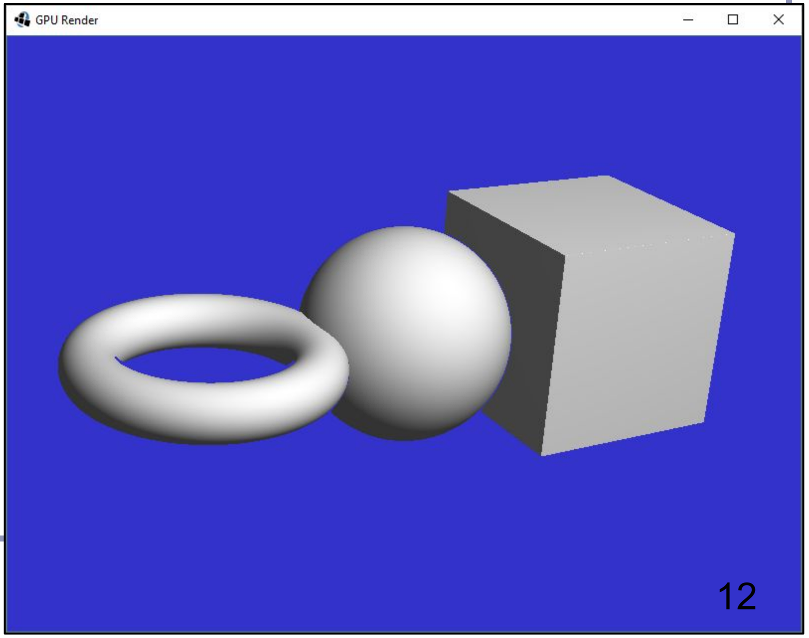
Find the normal to an SDF

Finding the normal: local gradient

```
float d = getSdf(pt);
vec3 normal = normalize(vec3(
    getSdf(vec3(pt.x + 0.0001, pt.y, pt.z)) - d,
    getSdf(vec3(pt.x, pt.y + 0.0001, pt.z)) - d,
    getSdf(vec3(pt.x, pt.y, pt.z + 0.0001)) - d));
```

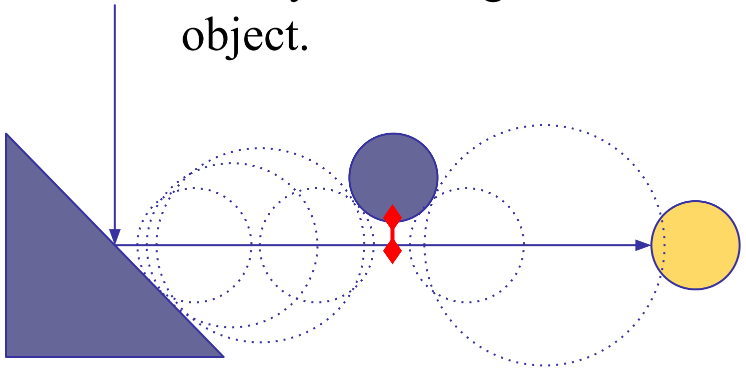
The distance function is locally linear and changes most as the sample moves directly away from the surface. At the surface, the direction of greatest change is therefore equivalent to the normal to the surface.

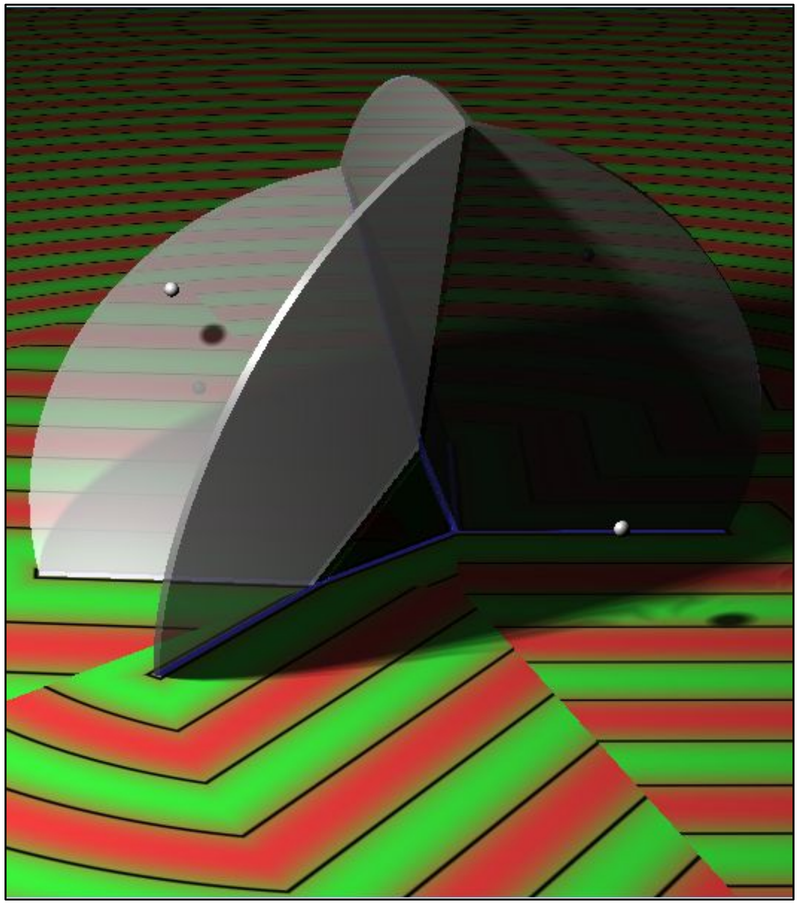
Thus the local gradient (the normal) can be approximated from the distance function.



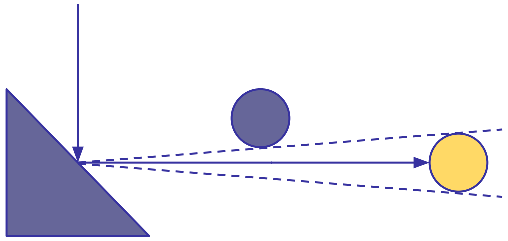
SDF shadows

- Ray-marched shadows are straightforward: march a ray towards each light source, don't illuminate if the SDF ever drops too close to zero.
- Unlike ray-tracing, soft shadows are almost free with SDFs: attenuate illumination by a linear function of the ray marching *near* to another object.



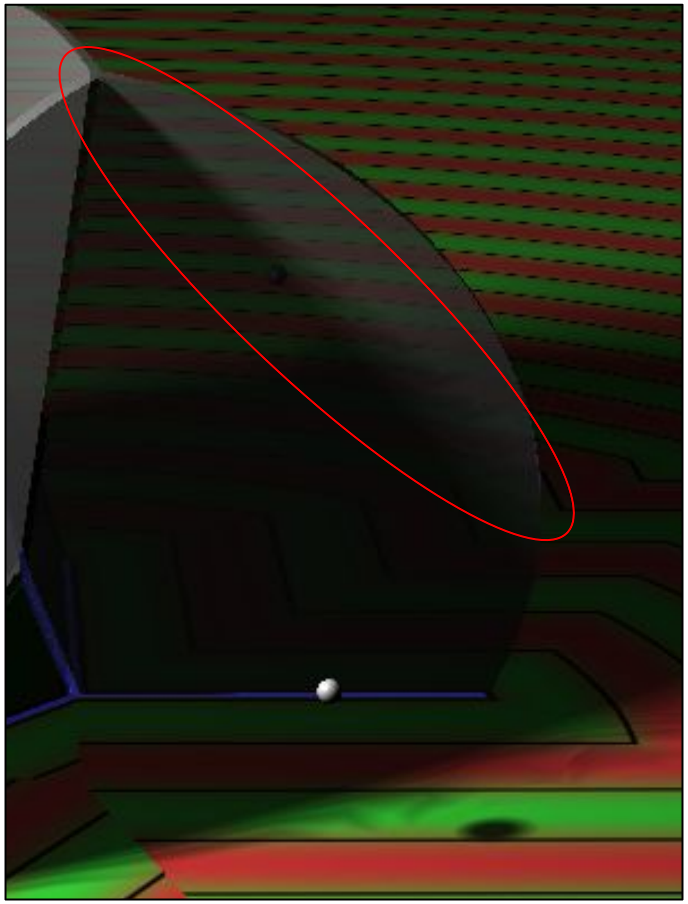


Soft SDF shadows



```
float shadow(vec3 pt) {
  vec3 lightDir = normalize(lightPos - pt);
  float kd = 1;
  int step = 0;

  for (float t = 0.1;
      t < length(lightPos - pt)
      && step < renderDepth && kd > 0.001; ) {
    float d = abs(getSDF(pt + t * lightDir));
    if (d < 0.001) {
      kd = 0;
    } else {
      kd = min(kd, 16 * d / t);
    }
    t += d;
    step++;
  }
  return kd;
}
```

By dividing *d* by *t*, we attenuate the strength of the shadow as its source is further from the illuminated point.

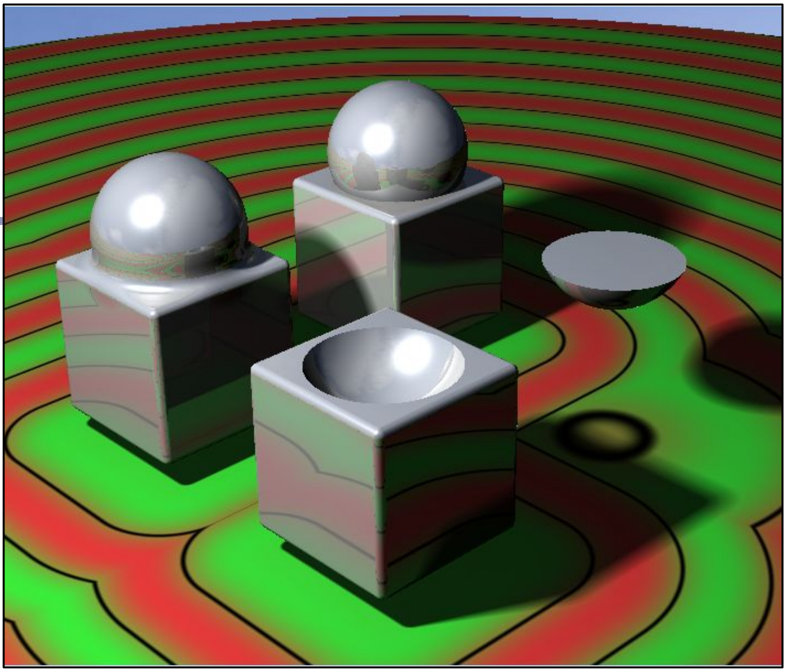


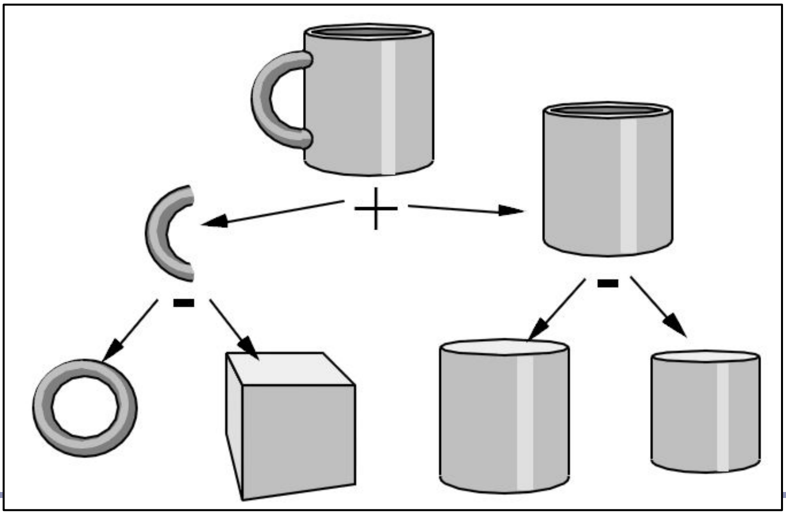
Combining SDFs

We combine SDF models by choosing which is closer to the sampled point.

- Take the **union** of two SDFs by taking the `min()` of their functions.
- Take the **intersection** of two SDFs by taking the `max()` of their functions.
- The `max()` of function A and the negative of function B will return the **difference** of A - B.

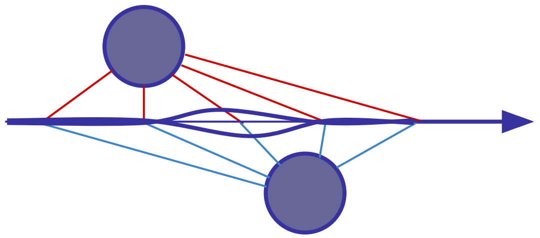
By combining these binary operations we can create functions which describe very complex primitives.

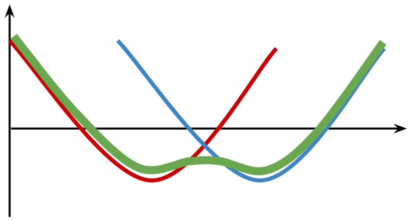


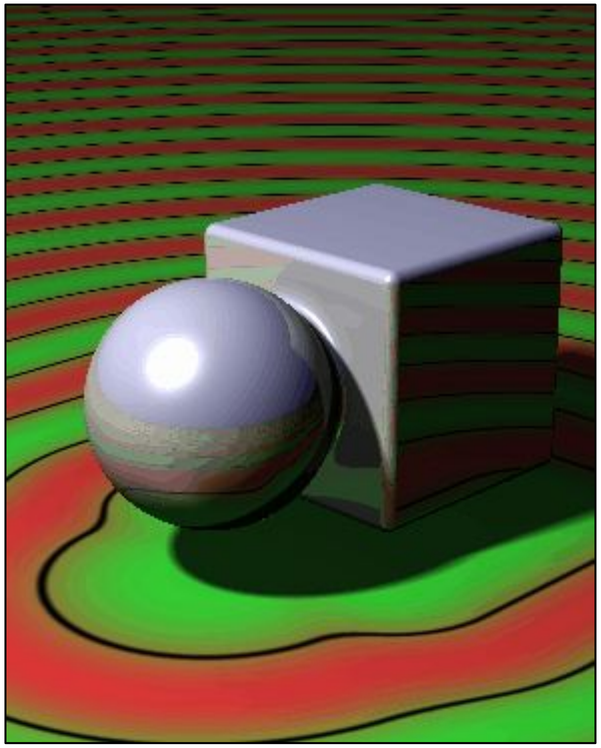


Blending SDF geometry

Taking the `min()`, `max()`, etc of two SDFs yields a sharp discontinuity. *Interpolating* the two SDFs with a smooth polynomial yields a smooth distance curve, blending the models:







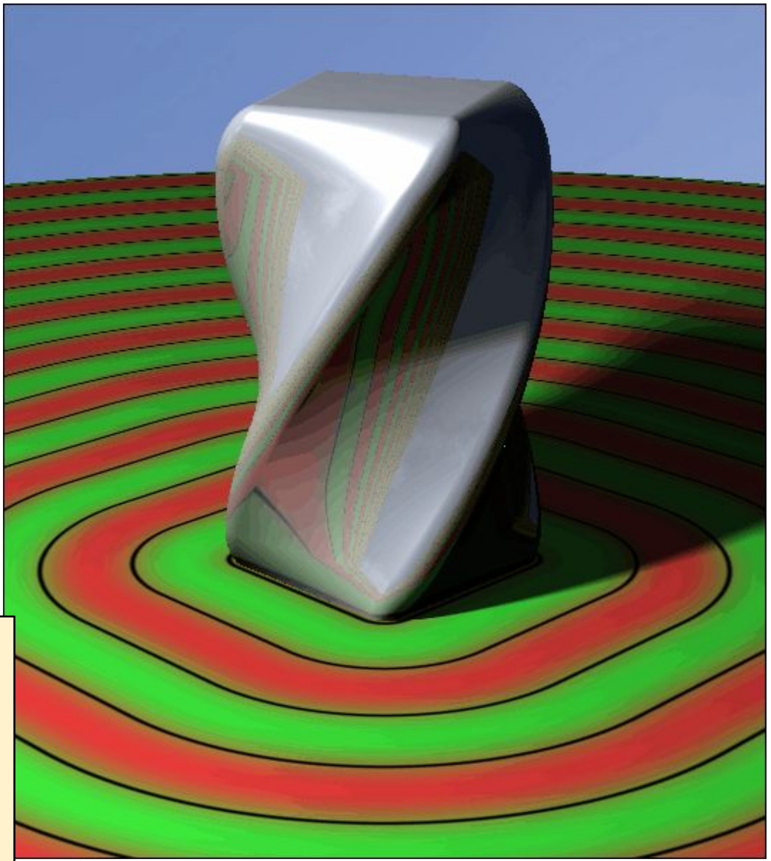
Sample blending function (Quilez)

```
float smin(float a, float b) {
  float k = 0.2;
  float h = clamp(0.5 + 0.5 * (b - a) / k, 0,
1);
  return mix(b, a, h) - k * h * (1 - h);
}
```

http://iquilezles.org/www/articles/smin/smin.htm

Twists and deformations

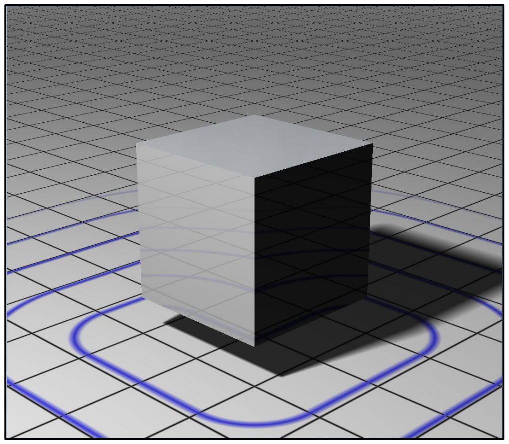
We can apply non-uniform spatial distortion, such as by choosing how much we'll modify space as a function of where we are in space.



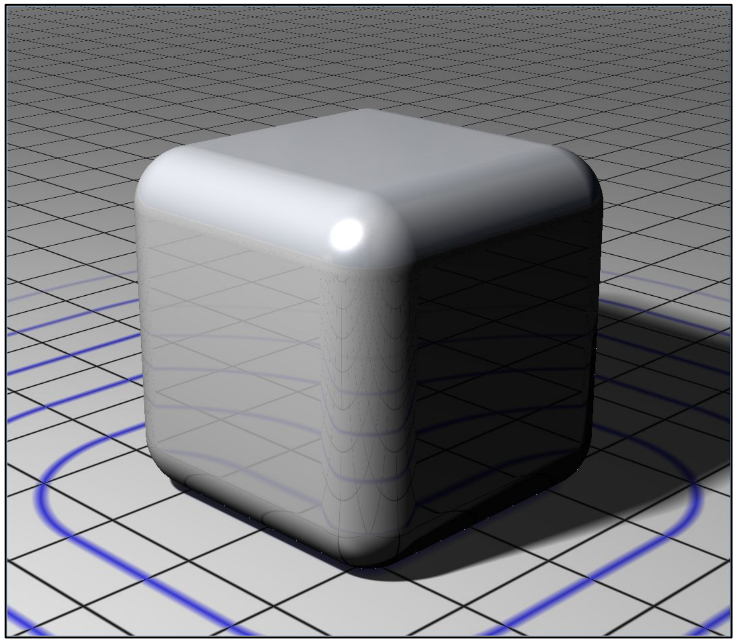
```
float fScene(vec3 pt) {
  pt.y -= 1;
  float t = (pt.y + 2.5) * sin(time);
  return sdCube(vec3(
    pt.x * cos(t) - pt.z * sin(t),
    pt.y / 2,
    pt.x * sin(t) + pt.z * cos(t)), vec3(1));
}
```

Rounded corners

An SDF function returns distance to the surface, so if you subtract a constant, you round off hard edges and corners:

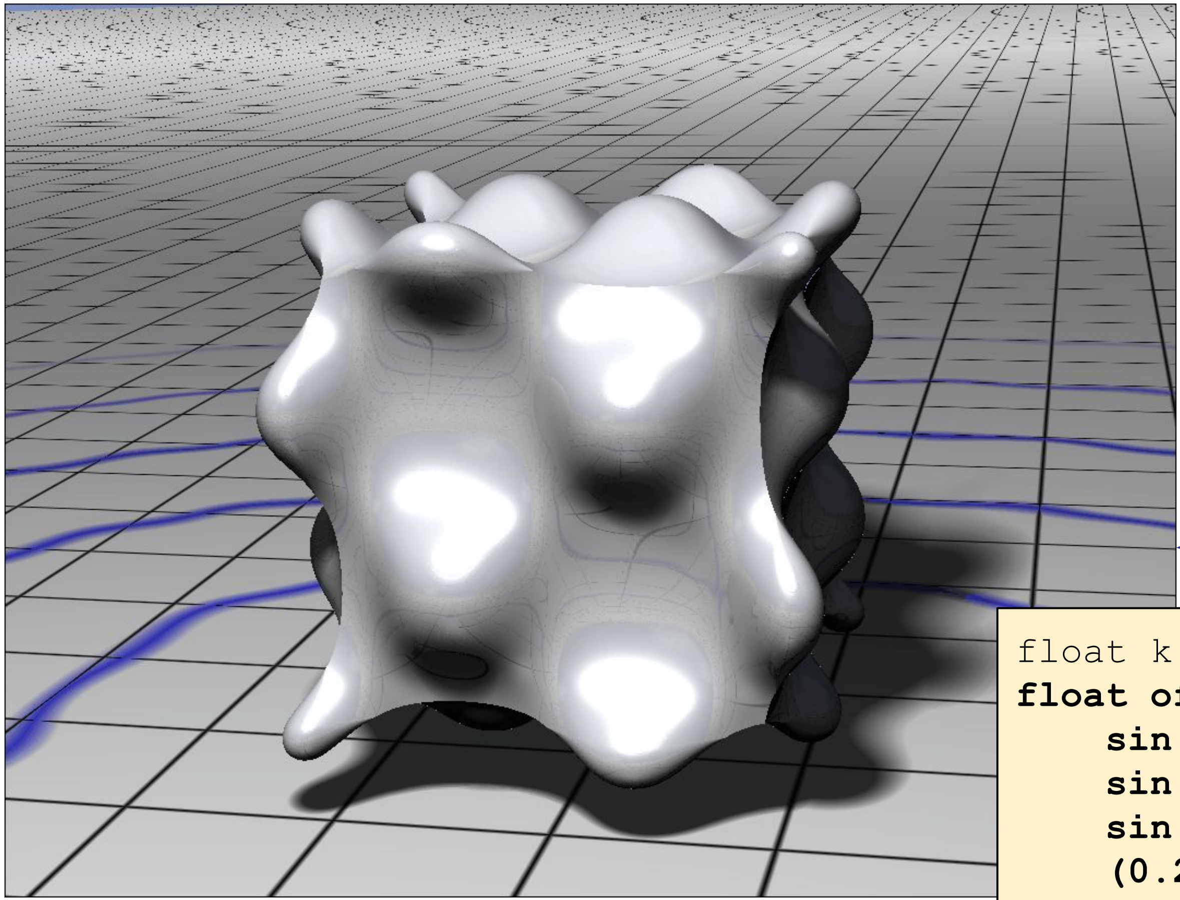


```
float cube(vec3 p) {
  vec3 d = abs(p) - vec3(1);
  return min(
      max(d.x, max(d.y, d.z)),
      0.0) + length(max(d, 0.0));
} // Exactly accurate cube SDF
```



```
dist = cube(p) - 0.5;
```

“Rounded corners” taken to extremes: Displacement maps

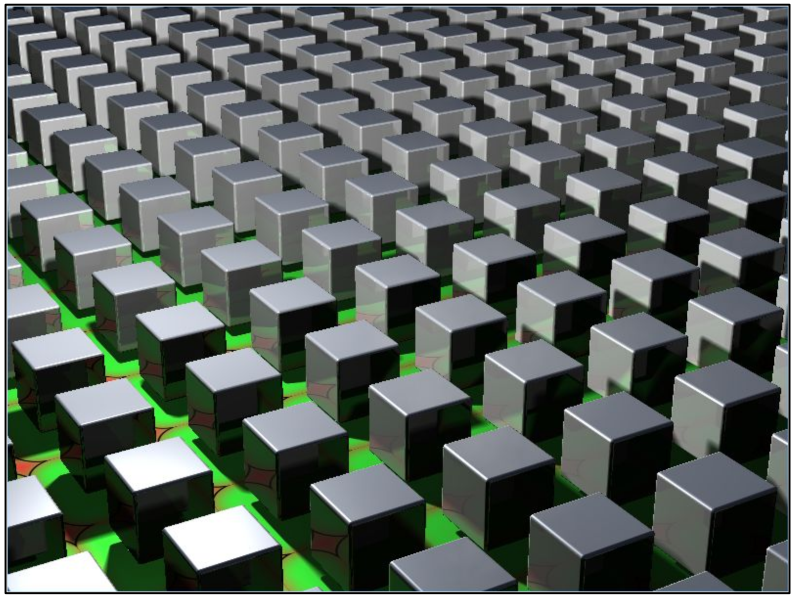


```
float k = 4;
float offset =
    sin(k * pt.x) *
    sin(k * pt.y) *
    sin(k * pt.z) *
    (0.25 + 0.25 * sin(time));
return cube(pt) + offset;
```

Repeating SDF geometry

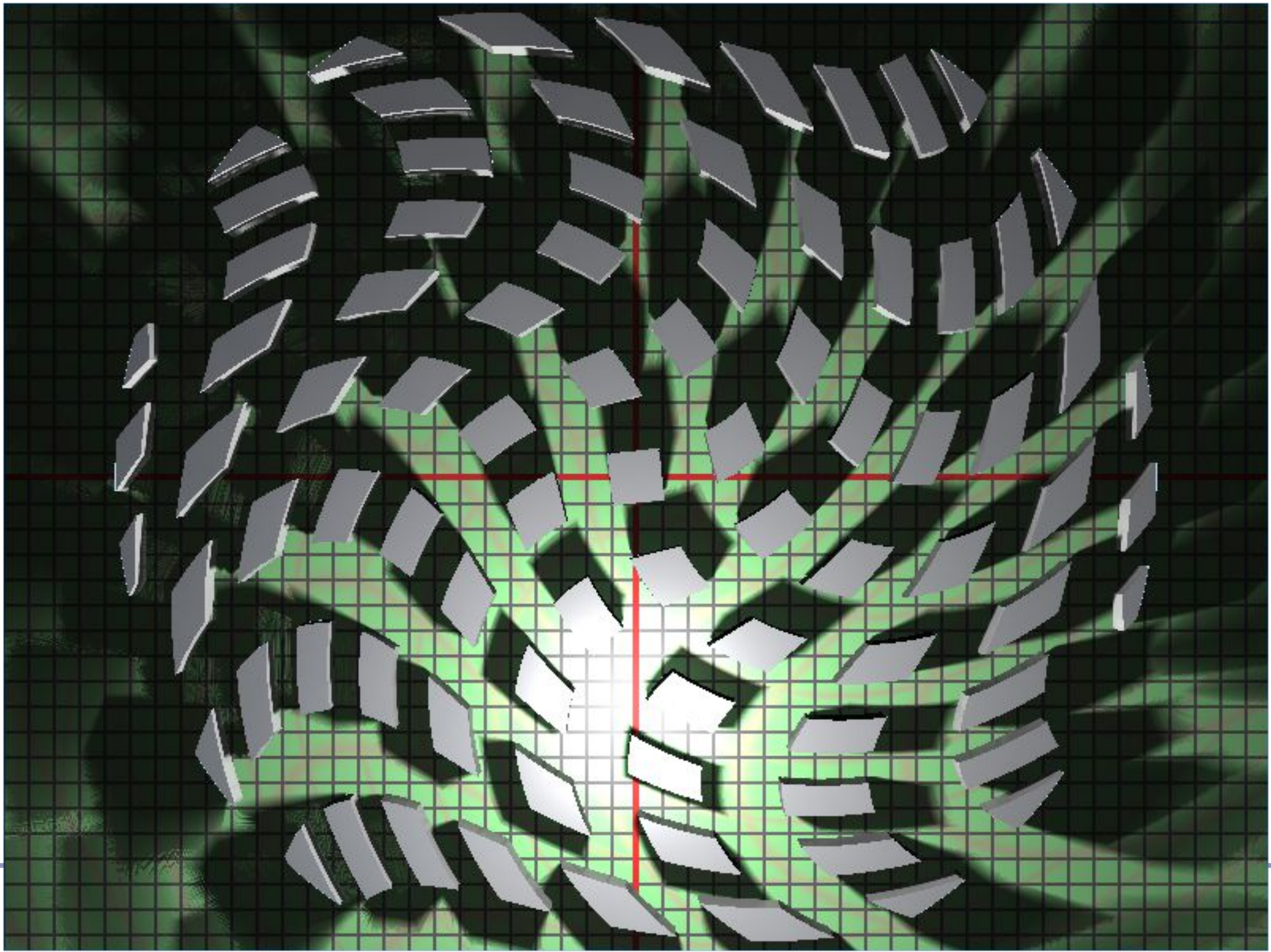
If we take the modulus of a point's position along one or more axes before computing its signed distance, then we segment space into infinite parallel regions of repeated distance. Space near the origin 'repeats'.

With SDFs we get infinite repetition of geometry for no extra cost.



```
float fScene(vec3 pt) {
  vec3 pos;
  pos = vec3(mod(pt.x + 2, 4) - 2, pt.y, mod(pt.z + 2, 4) - 2);
  return sdCube(pos, vec3(1));
}
```

SDF - Live demo



Recommended reading

Seminal papers

- John C. Hart, "Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces", http://graphics.cs.illinois.edu/papers/zeno
- John C. Hart et al., "Ray Tracing Deterministic 3-D Fractals", http://graphics.cs.illinois.edu/sites/default/files/rtqjs.pdf

Special kudos to Inigo Quilez and his amazing blog

- http://iquilezles.org/www/articles/smin/smin.htm
- http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

Other useful sources

- Johann Korndorfer, "How to Create Content with Signed Distance Functions", https://www.youtube.com/watch?v=s8nFqwOho-s
- Daniel Wright, "Dynamic Occlusion with Signed Distance Fields", http://advances.realtimerendering.com/s2015/DynamicOcclusionWithSignedDistanceFields.pdf
- 9bit Science, "Raymarching Distance Fields", http://9bitscience.blogspot.co.uk/2013/07/raymarching-distance-fields_14.html