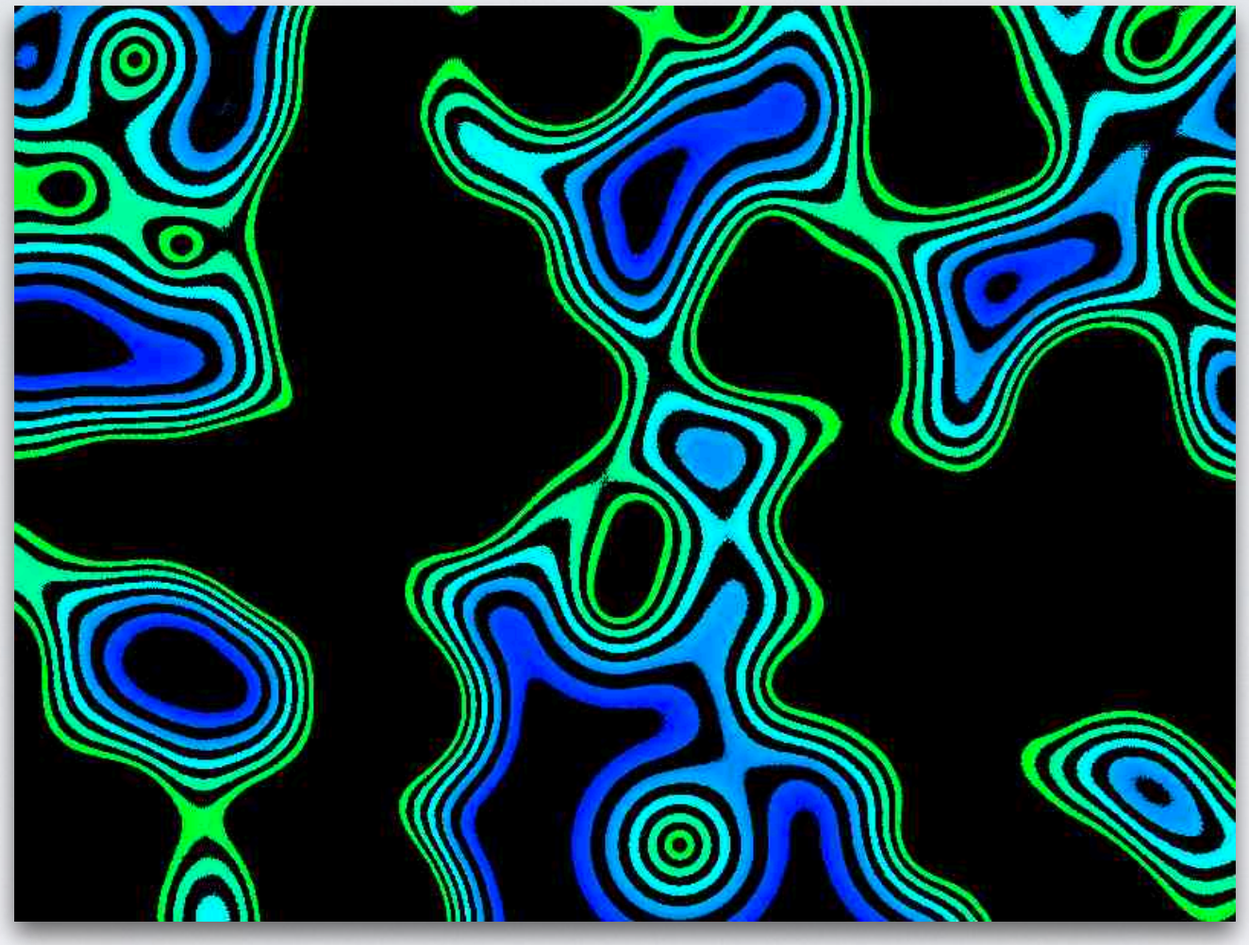
IMPLICIT SURFACES

Alternative methods for representing complex scenes in 3D

REPRESENTING SURFACES



Jeff Hube (source)

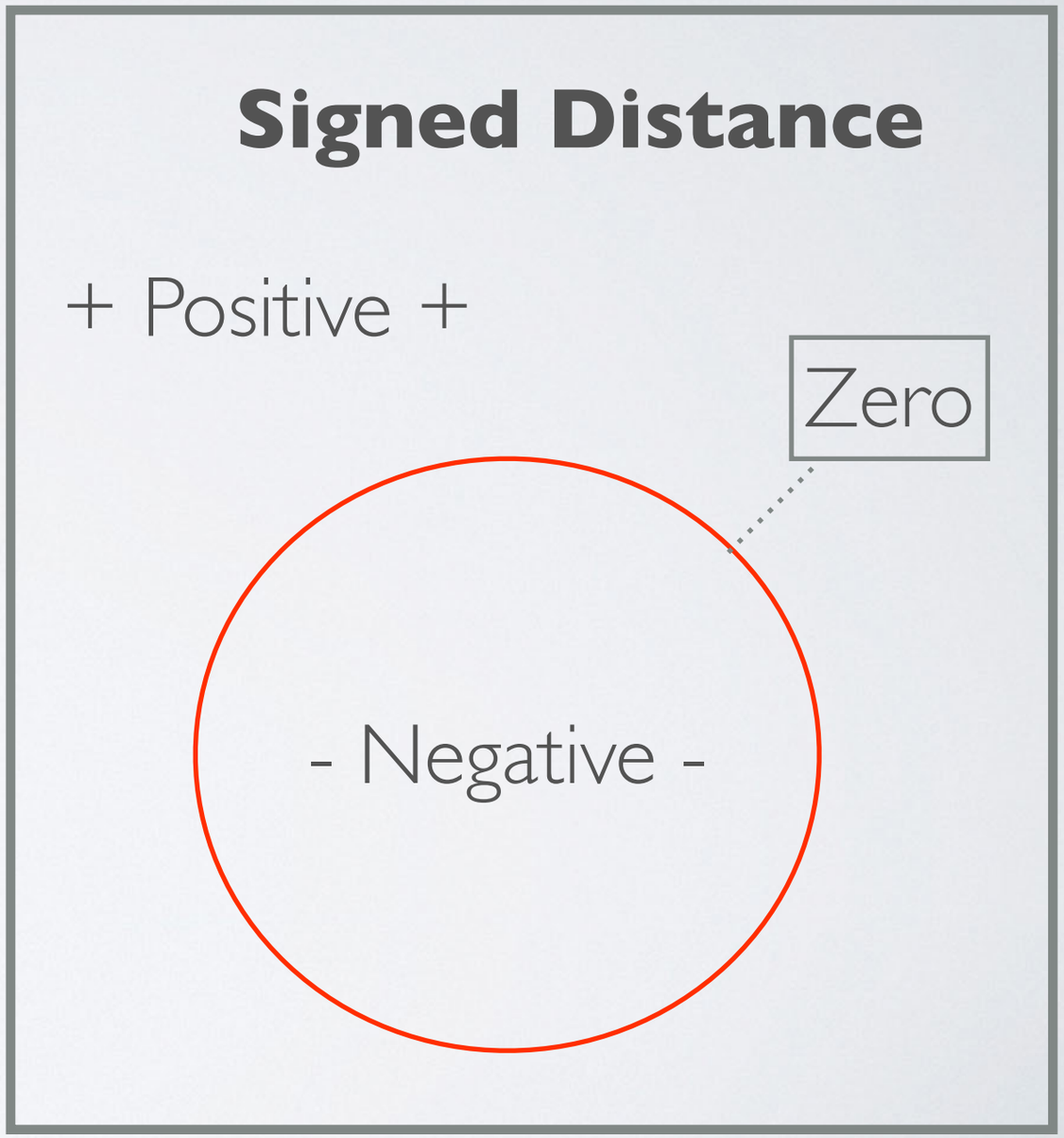
- There are multiple ways to represent surfaces
- We can manually specify the location of some vertices and interpolate between them.
- We can also come up with a function that describes the surface of an object relative to the objects center position. Demo demo - there are tons of cool examples out there!

SIGNED DISTANCE FUNCTIONS

- One way to represent a surface is functionally determining where we are relative to an object's surface
- Signed distance functions (SDFs) take an input point and return shortest distance to the surface of a shape
  - Zero means we're on the surface of the object
  - A positive number means we're outside the object
  - A negative number means we're inside the object
- Ex. sphere:
  - Compare length(vector to sphere center) to radius

$$f(\vec{p}) = ||\vec{p}|| - 1$$

```
// For a sphere centered at the origin
float sphereSDF(vec3 p ) {
  return length(p) - RADIUS;
}
```



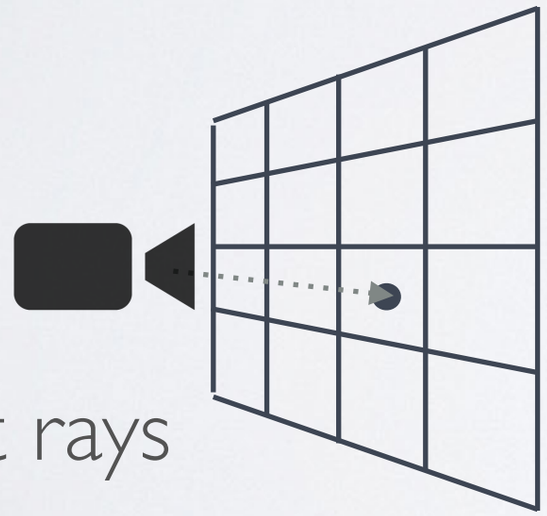
RAY MARCHING

REVIEW: RAY TRACING

- So, how do we render geometry modeled with SDFs?
- Ray marching! An algorithm very similar to ray tracing.
- **Quick ray tracing review:**
    - Cast rays through every cell of a grid (corresponding to pixels)
    - For each ray, test for intersection with each piece of geometry
        - Transform ray into model space
        - Plug ray equation into geometry equation, solve for t (intersection distance)

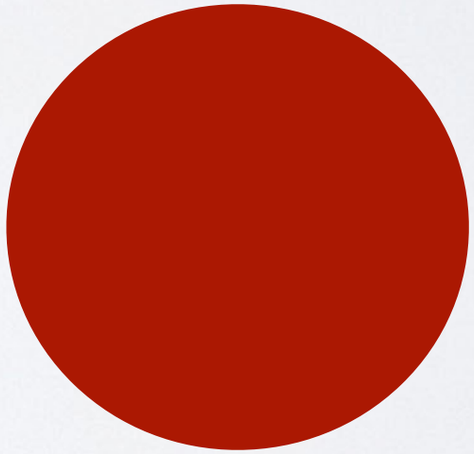
RAY MARCHING

- With SDFs, we can't just conveniently plug in our ray equation. But, we can essentially guess and check!
- However, we know zero or negative distance from object means we're on/inside a surface — an intersection!
- **The ray marching algorithm:**
    - Cast rays through every cell of a grid (corresponding to pixels)
    - For each ray, using some small t value, compute a point **p = ray_origin + ray_direction * t**
        - Plug point into scene SDF. If distance is zero or negative, intersection!
        - If no intersection, take another small step forward along the ray, repeat till some max distance



cast rays
from camera

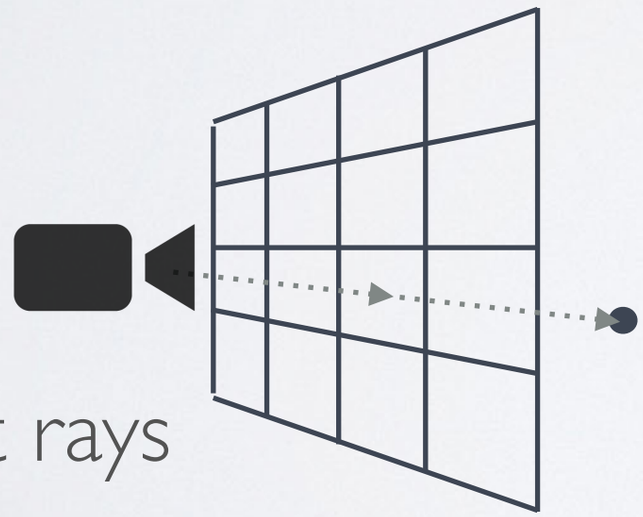
Intersection?!



No. Distance is positive. Try again.

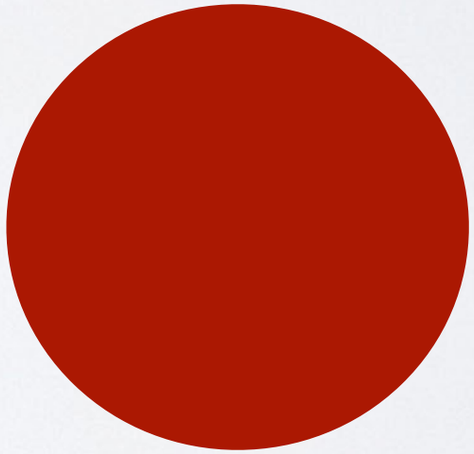
RAY MARCHING

- With SDFs, we can't just conveniently plug in our ray equation. But, we can essentially guess and check!
- However, we know zero or negative distance from object means we're on/inside a surface — an intersection!
- **The ray marching algorithm:**
    - Cast rays through every cell of a grid (corresponding to pixels)
    - For each ray, using some small t value, compute a point **p = ray_origin + ray_direction * t**
        - Plug point into scene SDF. If distance is zero or negative, intersection!
        - If no intersection, take another small step forward along the ray, repeat till some max distance



cast rays
from camera

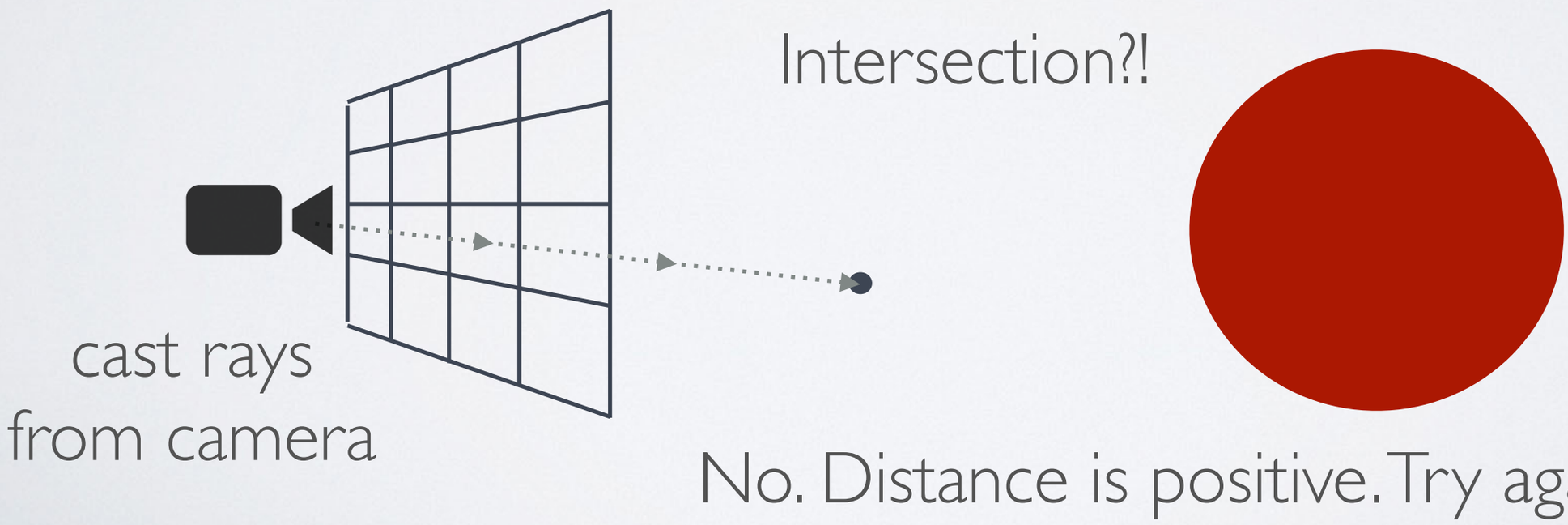
Intersection?!



No. Distance is positive. Try again.

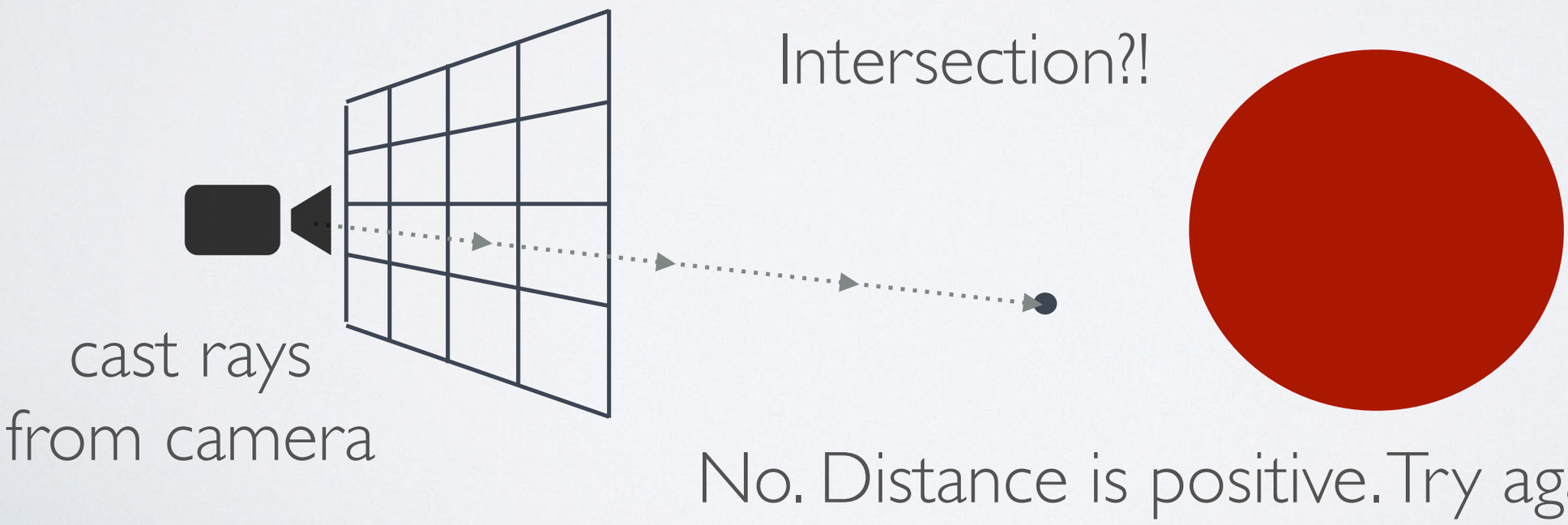
RAY MARCHING

- With SDFs, we can't just conveniently plug in our ray equation. But, we can essentially guess and check!
- However, we know zero or negative distance from object means we're on/inside a surface — an intersection!
- **The ray marching algorithm:**
    - Cast rays through every cell of a grid (corresponding to pixels)
    - For each ray, using some small t value, compute a point **p = ray_origin + ray_direction * t**
        - Plug point into scene SDF. If distance is zero or negative, intersection!
        - If no intersection, take another small step forward along the ray, repeat till some max distance



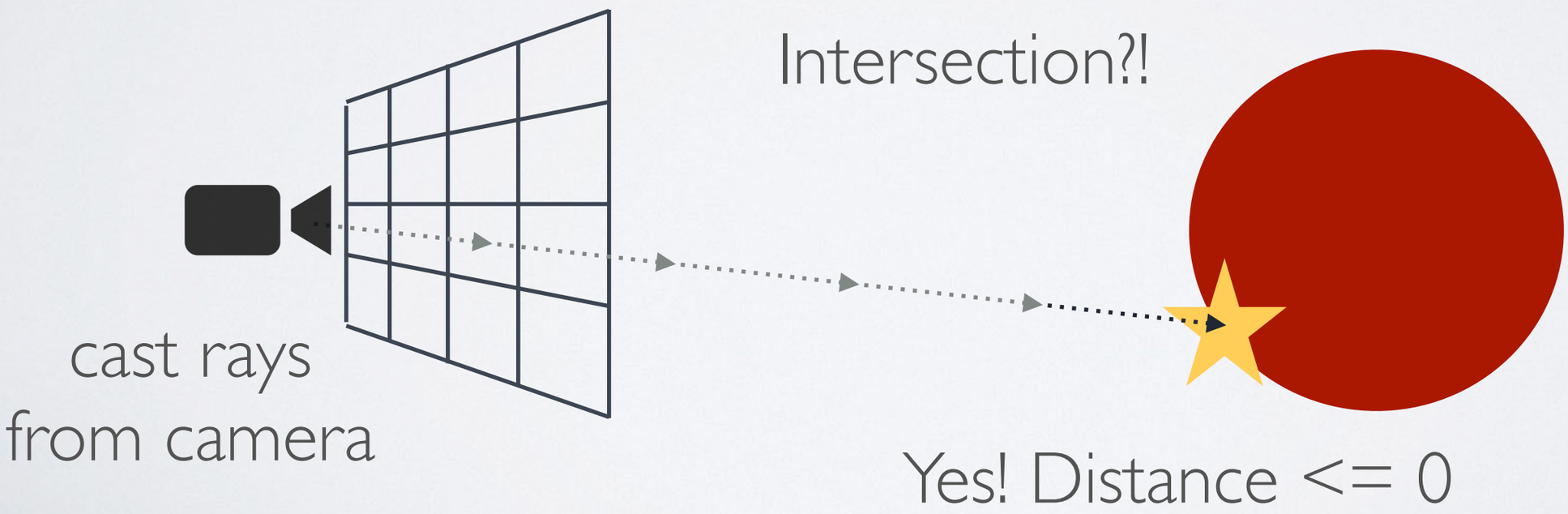
RAY MARCHING

- With SDFs, we can't just conveniently plug in our ray equation. But, we can essentially guess and check!
- However, we know zero or negative distance from object means we're on/inside a surface — an intersection!
- **The ray marching algorithm:**
    - Cast rays through every cell of a grid (corresponding to pixels)
    - For each ray, using some small t value, compute a point **p = ray_origin + ray_direction * t**
        - Plug point into scene SDF. If distance is zero or negative, intersection!
        - If no intersection, take another small step forward along the ray, repeat till some max distance.



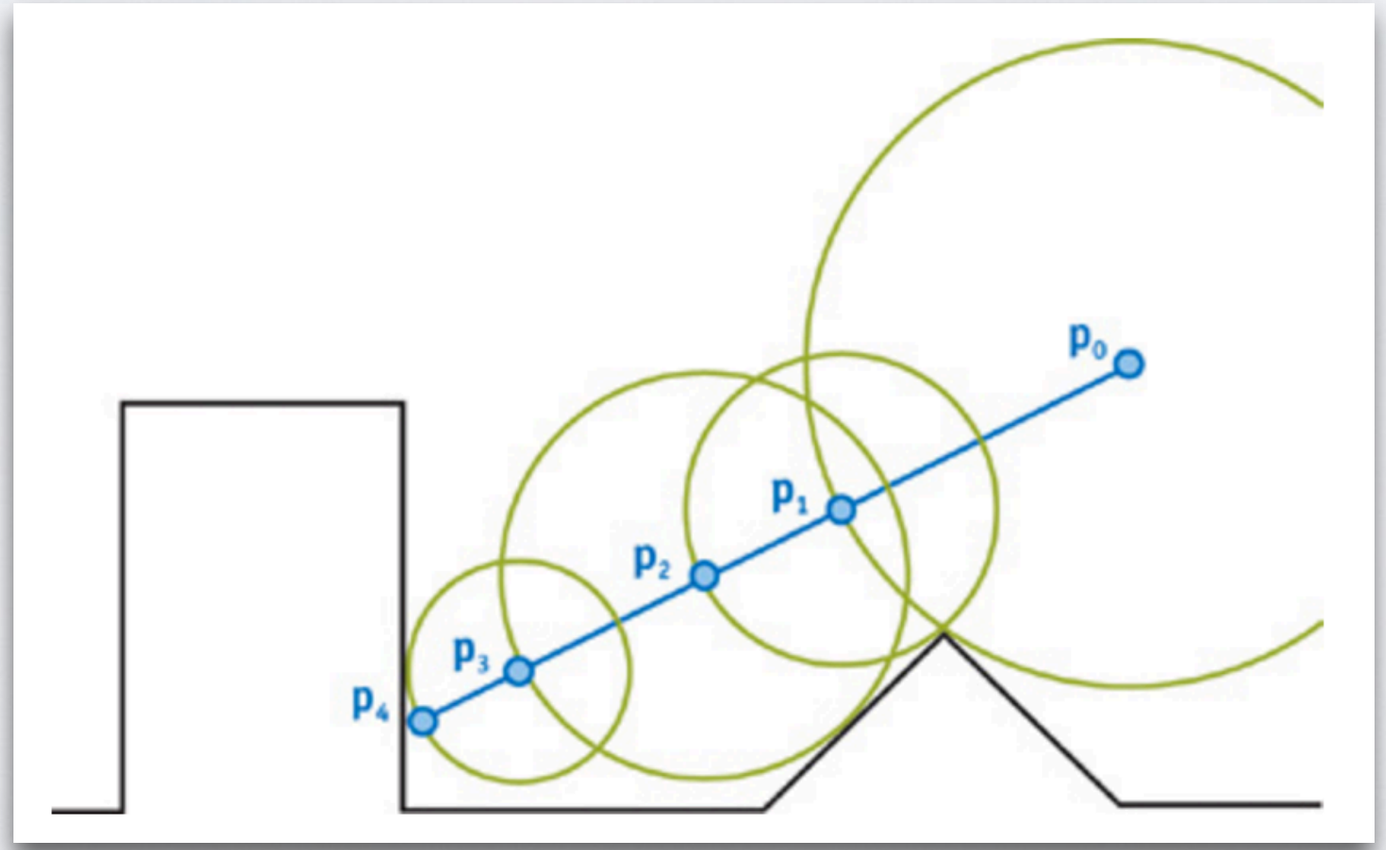
RAY MARCHING

- With SDFs, we can't just conveniently plug in our ray equation. But, we can essentially guess and check!
- However, we know zero or negative distance from object means we're on/inside a surface — an intersection!
- **The ray marching algorithm:**
    - Cast rays through every cell of a grid (corresponding to pixels)
    - For each ray, using some small t value, compute a point **p = ray_origin + ray_direction * t**
        - Plug point into scene SDF. If distance is zero or negative, intersection!
        - If no intersection, take another small step forward along the ray, repeat till some max distance.



SPHERE TRACING

- We could always just step by the same small value
- But, we can do better! Save some checks by making step size dynamic
- Since at every step, we're getting a distance value, rather than take a fixed step size, step the maximum "safe" distance, ie current distance from scene SDF.



GPU gems (source)

BASIC IMPLEMENTATION

In glsl, here's the heart of the ray marching algorithm

```
float depth = start;
for (int i = 0; i < MAX_MARCHING_STEPS; i++) {
    float dist = sceneSDF(eye + depth * viewRayDirection);
    if (dist < EPSILON) {
        // We're inside the scene surface!
        return depth;
    }
    // Move along the view ray
    depth += dist;

    if (depth >= end) {
        // Gone too far; give up
        return end;
    }
}
return end;
```

Jamie Wong (source)

APPROXIMATE NORMALS

- The intersection point is great, but for most shading, we also need normals.
- What do we have to work with? Well, an SDF returns a range of values spanning negative to positive. Zero for a surface point.
- Idea: For a surface point, the direction that will move your SDF value from negative to positive fastest is the vector orthogonal to the surface. In other words, the normal!
- This is a concept from calculus: the **gradient**
- The gradient points in the direction of the greatest rate of increase of the function, and its magnitude is the slope of the graph in that direction

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

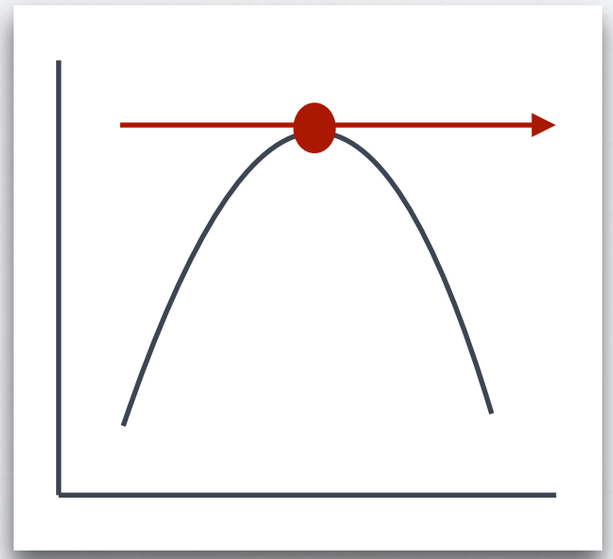
$$\nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

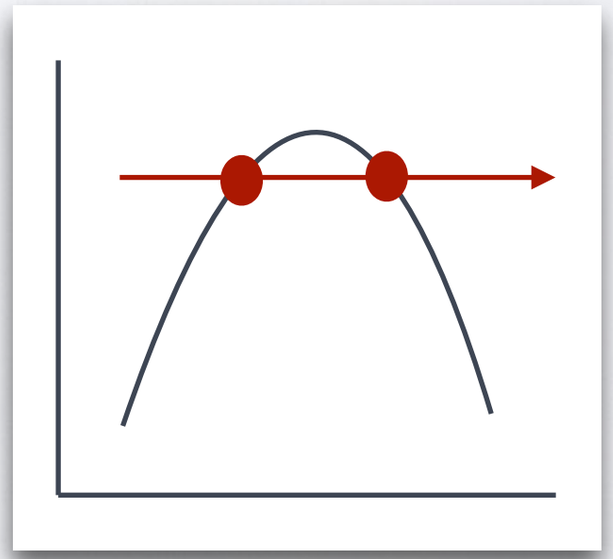
Jamie Wong (source)

APPROXIMATE NORMALS

- Fortunately, we don't even have to calculate the gradient, we can approximate!
- To get the derivative value at some point p, we can sample neighboring points and take their difference to approximate slope.
- Do for 3 dimensions, each component of our output vector 3, to get the slope in each dimension.
- So for some 1D SDF….

$$f'(p) \approx f(p+\varepsilon) - f(p-\varepsilon)$$





APPROXIMATE NORMALS

Implementation looks like this. Remember to normalize!!!!

$$\vec{n} = \begin{bmatrix} f(x+\varepsilon, y, z) - f(x-\varepsilon, y, z) \\ f(x, y+\varepsilon, z) - f(x, y-\varepsilon, z) \\ f(x, y, z+\varepsilon) - f(x, y, z-\varepsilon) \end{bmatrix}$$

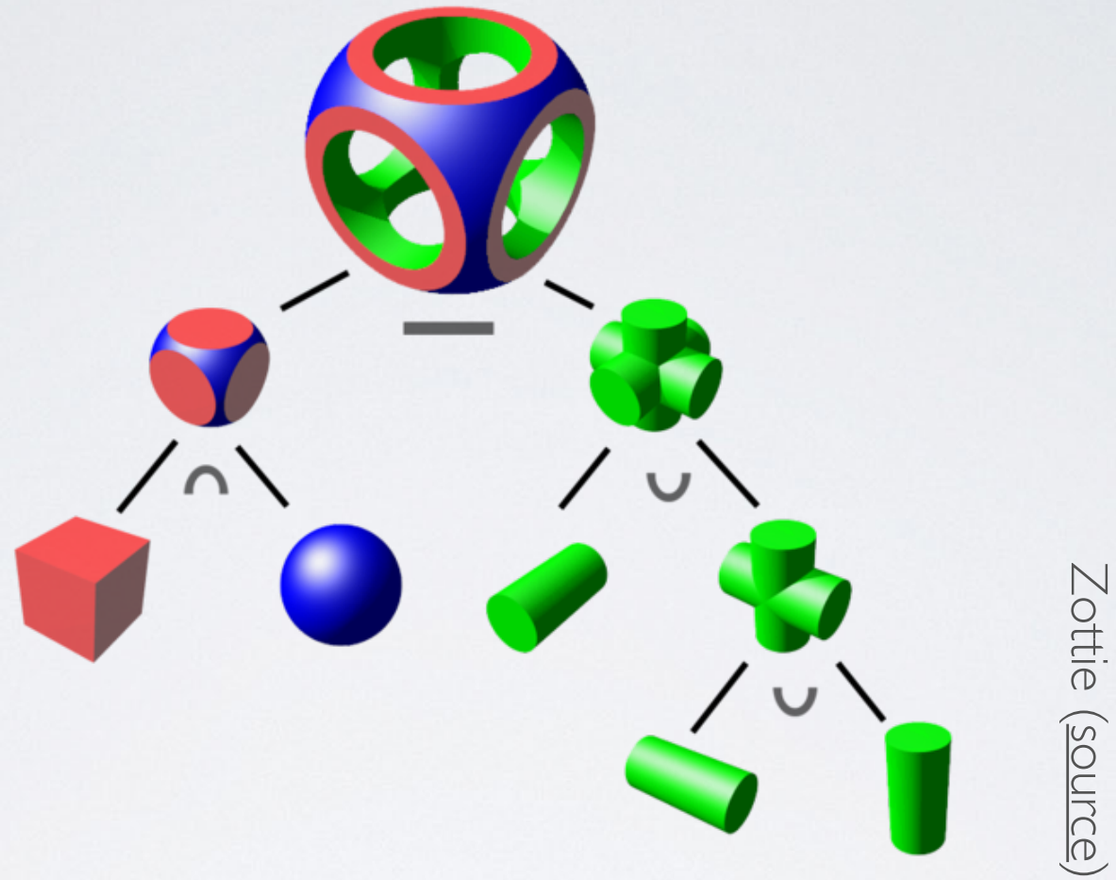
```
/**
 * Using the gradient of the SDF, estimate the normal on the surface at point p.
 */
vec3 estimateNormal(vec3 p) {
    return normalize(vec3(
        sceneSDF(vec3(p.x + EPSILON, p.y, p.z)) - sceneSDF(vec3(p.x - EPSILON, p.y, p.z)
        sceneSDF(vec3(p.x, p.y + EPSILON, p.z)) - sceneSDF(vec3(p.x, p.y - EPSILON, p.z)
        sceneSDF(vec3(p.x, p.y, p.z  + EPSILON)) - sceneSDF(vec3(p.x, p.y, p.z - EPSILON
    ));
```

Jamie Wong (source)

SIGNED DISTANCE FUNCTIONS

as usual, many credits to IQ

MODELING WITH SDFs



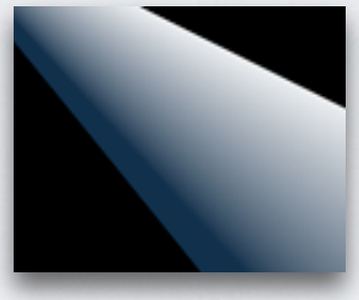
Zottie (source)

- Given a basic vocabulary of SDF shapes and the ability to combine them, you can make a wide variety of shapes.
- Constructive Solid Geometry: use boolean operations to make complex shapes.

BASIC SDFs

Cylinder - signed - exact

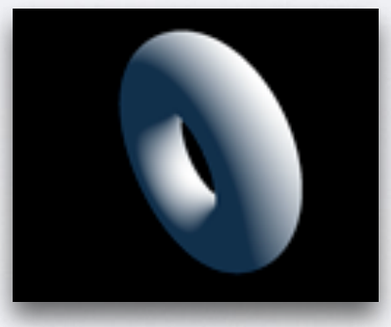
```
float sdCylinder( vec3 p, vec3 c )
{
  return length(p.xz-c.xy)-c.z;
}
```



- SDFs are a clever set of geometry problems.
- Basic SDFs assume geometry is positioned at the origin.
- Many primitive shapes to choose from!
- See here for more.

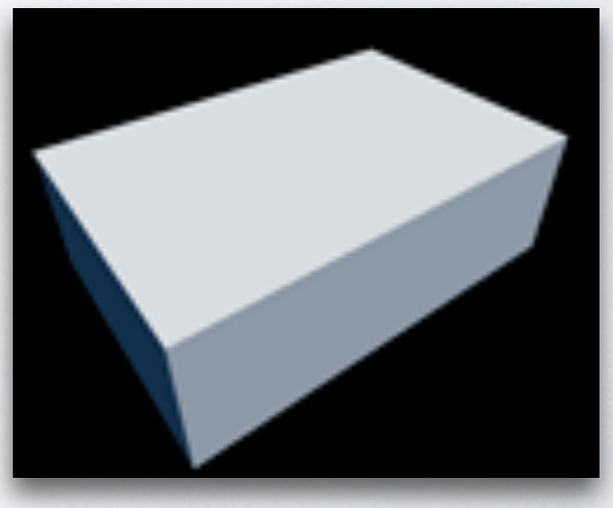
Torus - signed - exact

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



Box - signed - exact

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

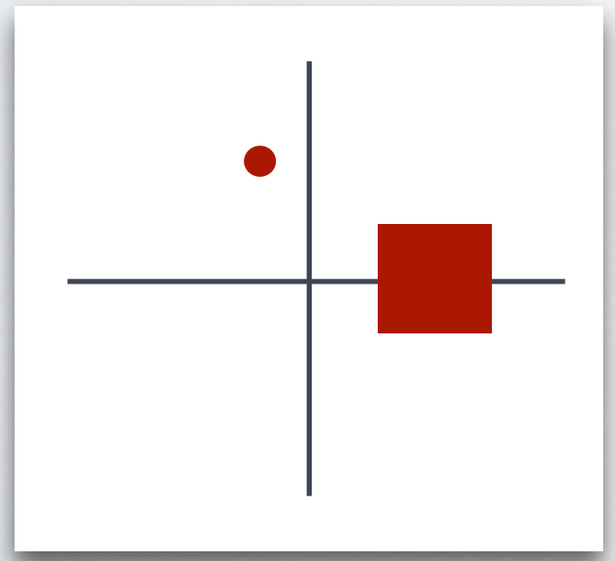


IQ (source)

TRANSFORMING SDFs

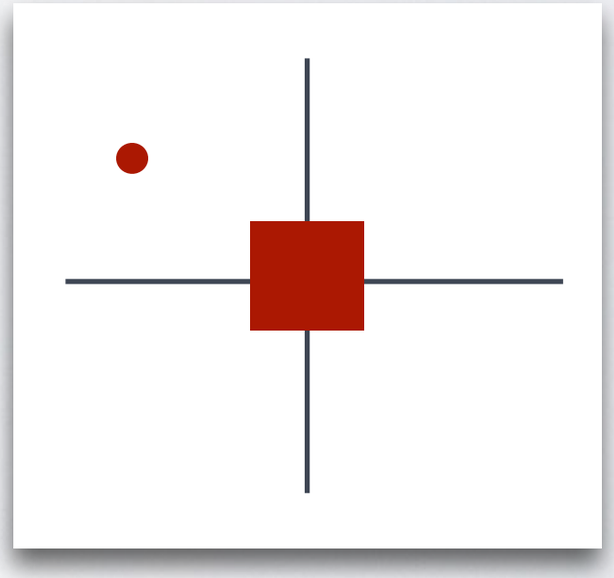
- For SDF rotation or translation, we can simply apply the inverse transformation to our point
- This works for the same reason we can apply the inverse model matrix to a ray in ray tracing — we're transforming our point into a space where the geometry is untransformed.
- Think of it like this: it's simplest to operate on platonic geometry. We make transformed geometry untransformed by multiplying with the inverse transform. But to keep our point operation equivalent, we have to transform the point by the same amount — the inverse transform!

Geometry transformed by M & original point



Equivalent, for SDFs

Point transformed by inverse M & untransformed geometry



TRANSFORMING SDFs

Repetition

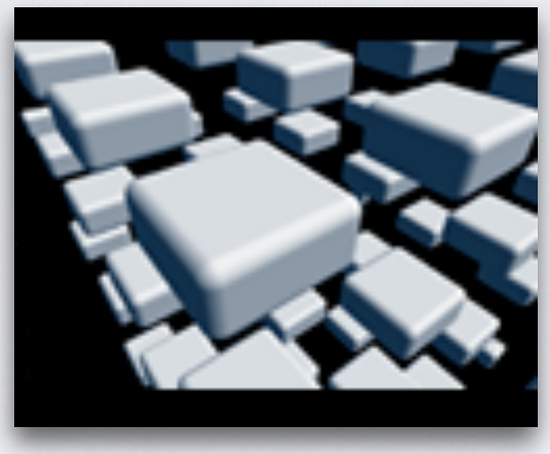
```
float opRep( vec3 p, vec3 c )
{
    vec3 q = mod(p,c)-0.5*c;
    return primitve( q );
}
```

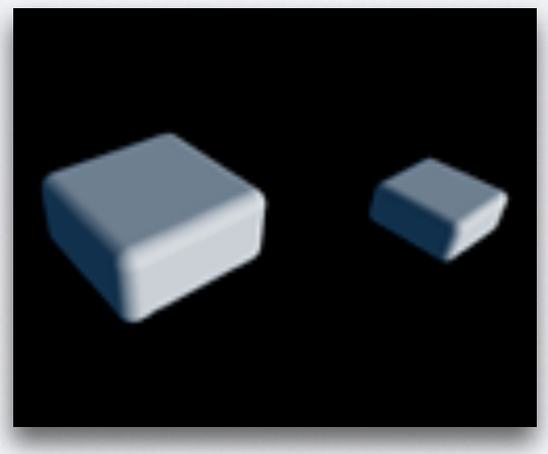
Rotation/Translation

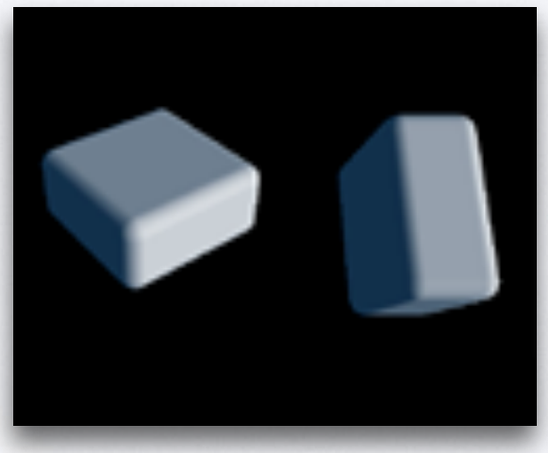
```
vec3 opTx( vec3 p, mat4 m )
{
    vec3 q = invert(m)*p;
    return primitive(q);
}
```

Scale

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





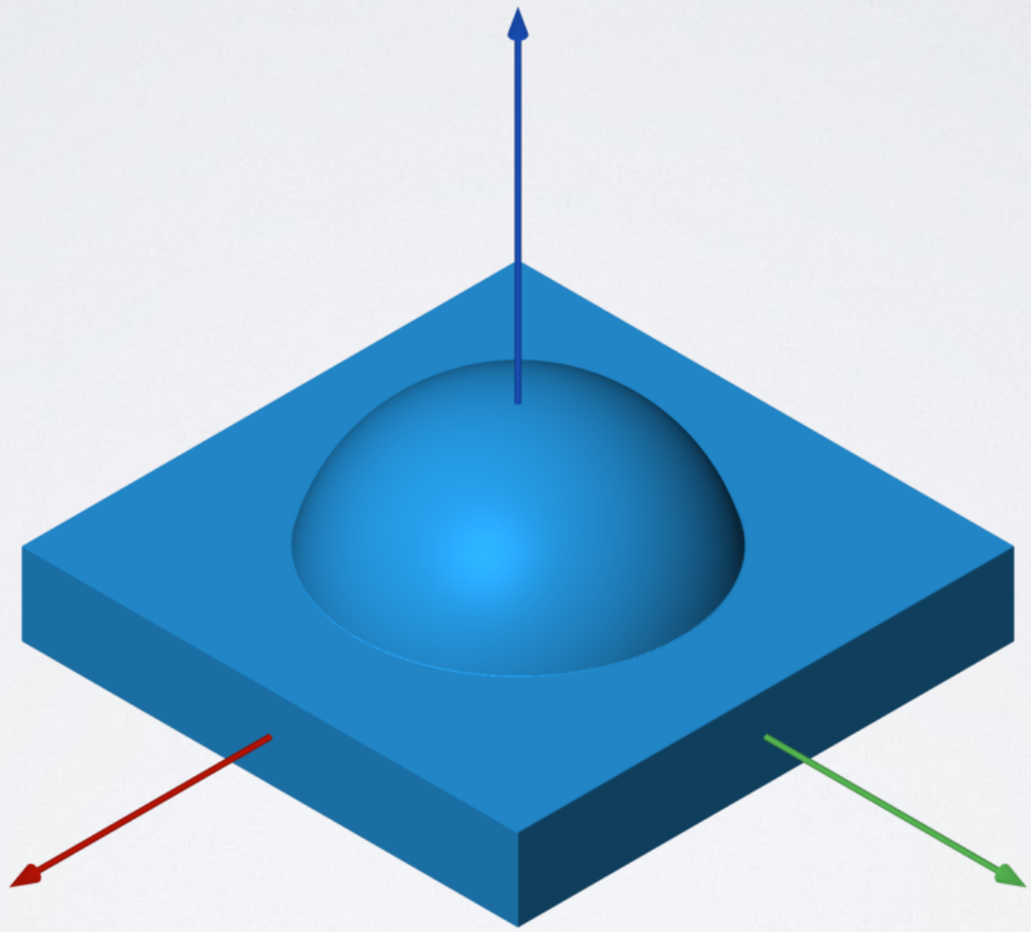


- Use a mod function to always reduce distance to nearest primitive, thus repeating.
- Transform primitives by actually transforming point with the inverse transform
- Scaling is like other transforms, except it doesn't preserve distance, so the return value must be scaled.
- Primitive = any SDF

IQ (source)

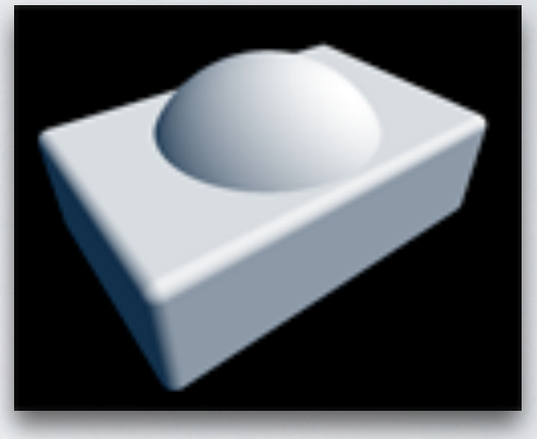
CONSTRUCTIVE SOLID GEOMETRY

- How to combine models represented with SDFs?



CONSTRUCTIVE SOLID GEOMETRY

- How to combine models represented with SDFs?
- Well, this just means you have two SDFs to consider.
- However, as with most rendering problems, we only care about the nearest intersection point, because geometry behind is occluded.



IQ (source)

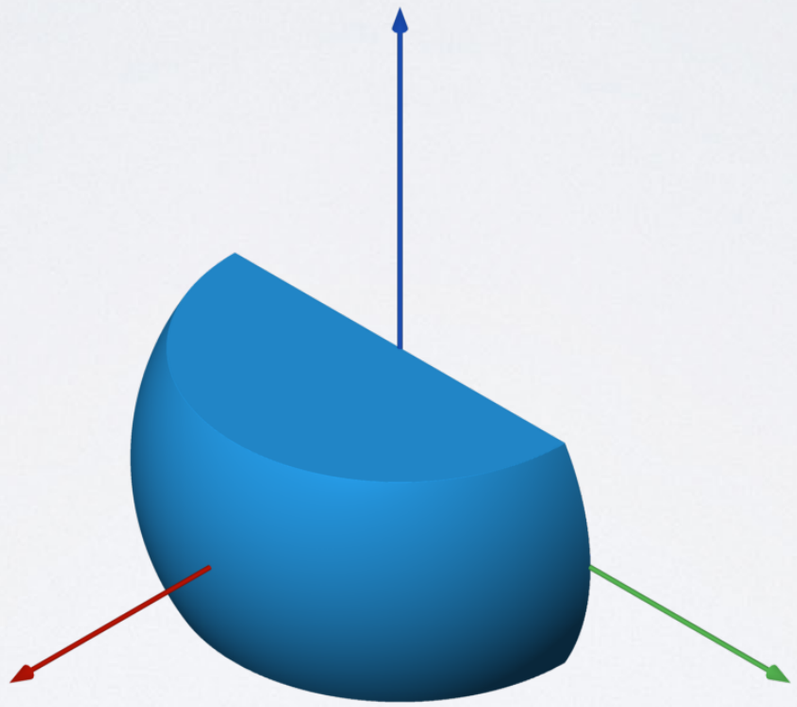
- The solution is simple, compute each SDF results, then just return the nearest distance of two SDFs!

Union

```
float unionSDF(float distance1, float distance2) {
  return min(distance1, distance 2);
}
```

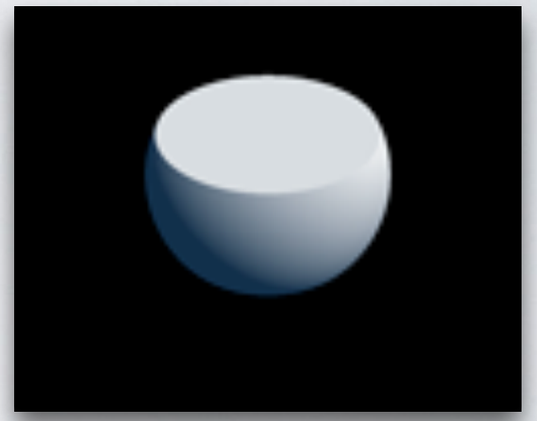
CONSTRUCTIVE SOLID GEOMETRY

- How about taking the intersection of two objects?



CONSTRUCTIVE SOLID GEOMETRY

- How about the intersection of two objects?
- Well, you only want to consider a surface if its within both of the objects
- So we can ignore the first intersection, and keep going until we hit the second object — if there's only one intersection, we're only intersecting with one object!



IQ (source)

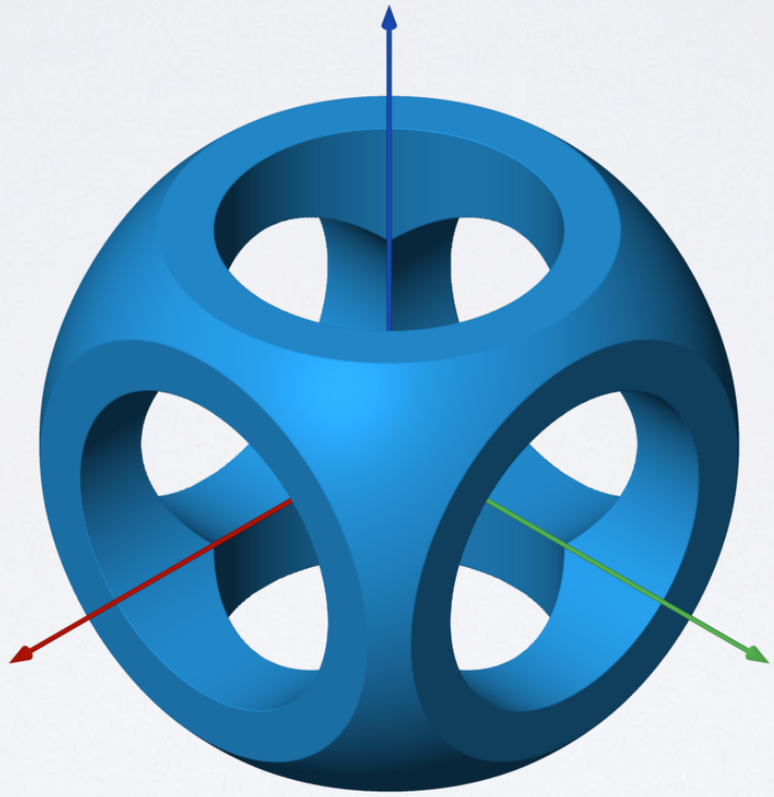
- This operation is taking the max of the two distances.

Intersection

```
float intersectionSDF(float distance1, float distance2) {
  return max(distance1, distance2);
}
```

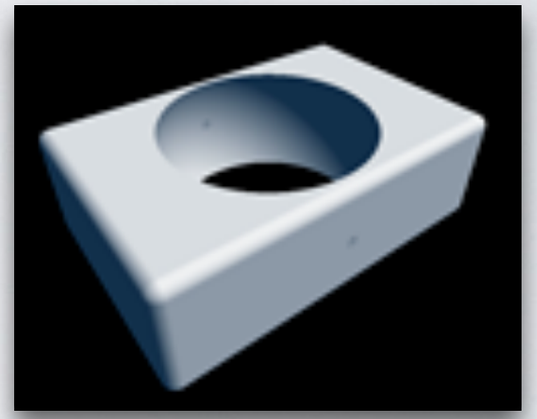
CONSTRUCTIVE SOLID GEOMETRY

- How about subtracting one object from another?



CONSTRUCTIVE SOLID GEOMETRY

- How to model a shape that is one shape minus another? Say A - B.
- Well, this is equivalent to taking the intersection of A and the inverse of B, meaning we can treat the outside of B like the inside, and vice versa.
- Simple to take the inverse of B, just negate!
- Since positive means outside and negative means inside, flipping the sign with flip inside/outside.



IQ (source)

Subtraction

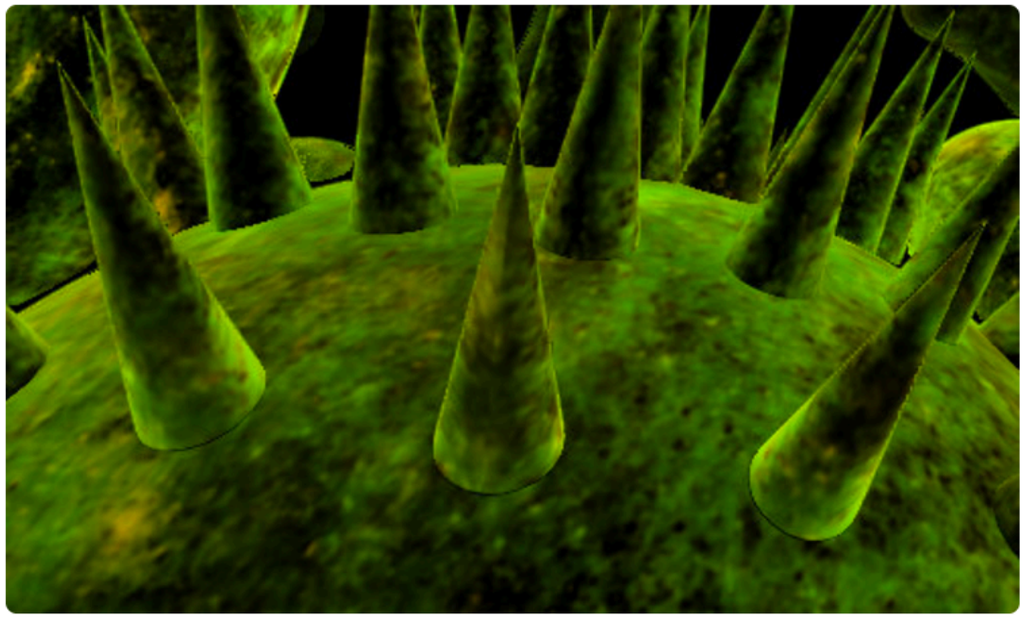
```
float subtractSDF(float distance1, float distance2) {
  return max(-distance1, distance2);
}
```

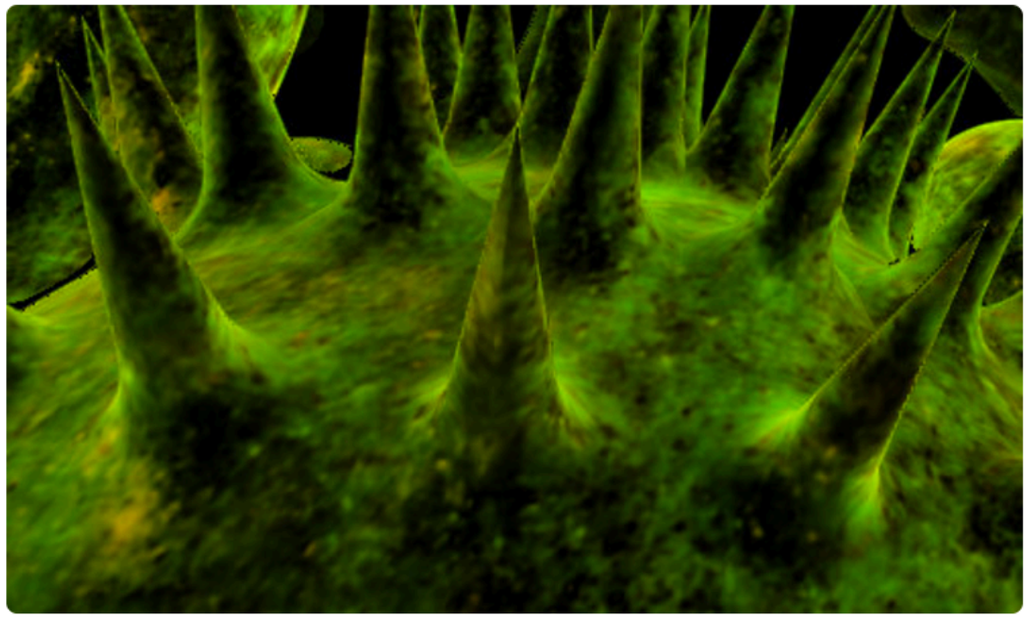
BLENDING SDFs

- Issue: our union produces a sharp discontinuity as we move from one surface to another
- Solution: use a smooth_min function instead of min. No discontinuity in derivatives!

```
// polynomial smooth min (k = 0.1);
float smin( float a, float b, float k )
{
    float h = clamp( 0.5+0.5*(b-a)/k, 0.0, 1.0 );
    return mix( b, a, h ) - k*h*(1.0-h);
}
```

IQ (source)





Vinicius Santos (source)

BONUS CONTENT

METABALLS

Clicktorelease (source)

- Common demo scene effect consisting of organic-looking balls that blend together when they're close together
- Implemented using isosurfaces, another type of implicit surface
    - Like SDFs, represent a surface as points of constant value within a volume.

METABALLS

- So with the SDFs we've been discussing, we would just check whether we're inside the surface to render.
- With metaballs, we can instead model each ball as having an influence field. We draw a surface at points where the total influence (from all the balls) are over some threshold value.
- Let's explain the 2D case (3D is the same, conceptually)
- Here's the formula for all points within a sphere of radius **r**, centered at **(x0, y0)**

$$(x - x_0)^2 + (y - y_0)^2 \leq r^2$$

- We can rearrange the function to look like this:

$$\frac{r^2}{(x - x_0)^2 + (y - y_0)^2} \geq 1$$

METABALLS

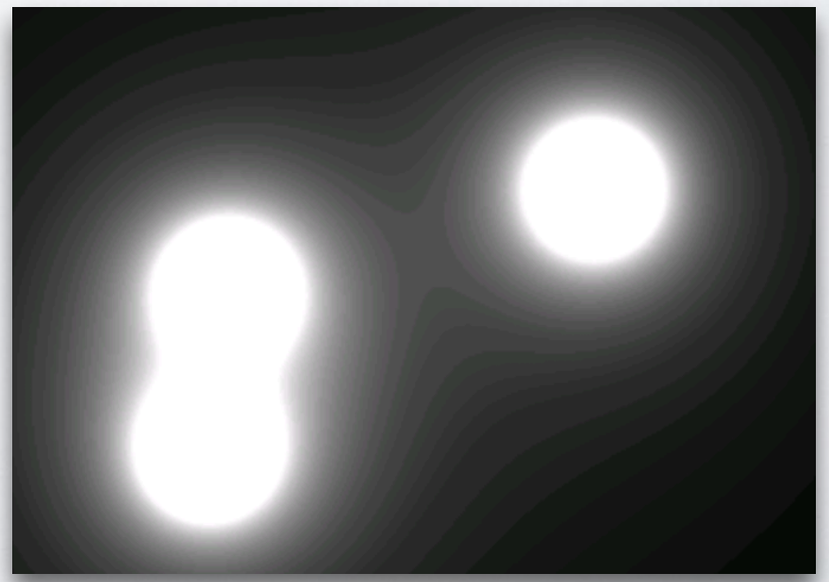
- Plugging in the appropriate values, we know that if the result of the left hand side is greater than or equal to 1, we're inside the surface. Outside the surface, we just have smaller values that falloff, decreasing slowly (visualized below as light).

- One way to model a surface between nearby balls is just to sum the influence from all of them at each point. If the total influence > 1, we're inside the metaball surface.

- Note: this is only one metaball equation. So long as we have a falloff function as distance from center increases (> 1 somewhere!), we can use it.

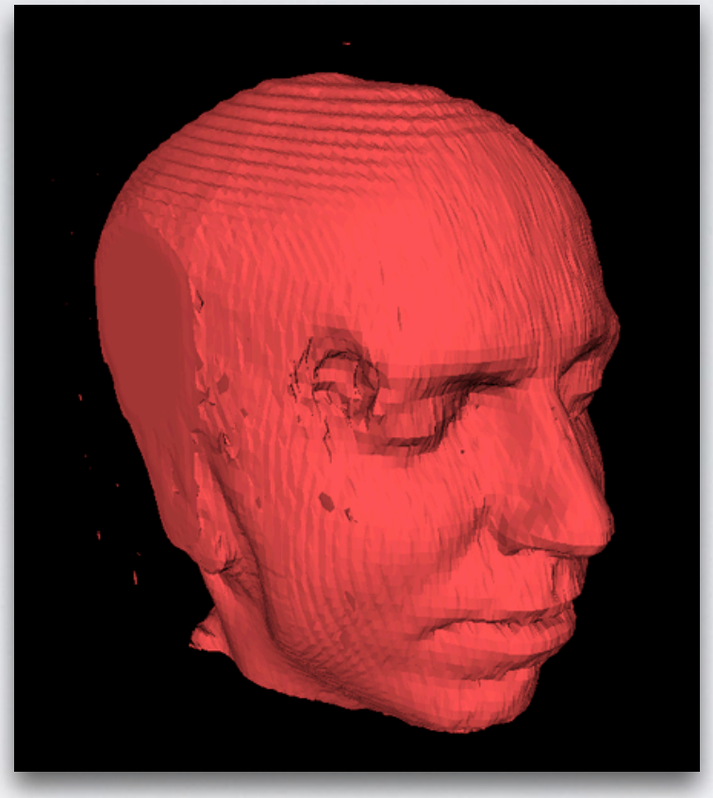
- This particular function is modeled off of formula for calculating the strength of an electrical field

$$f(x,y) = \sum_{i=0}^{n} \frac{r_i^2}{(x-x_i)^2 + (y-y_i)^2}$$



gamedev.net (source)

MARCHING CUBES

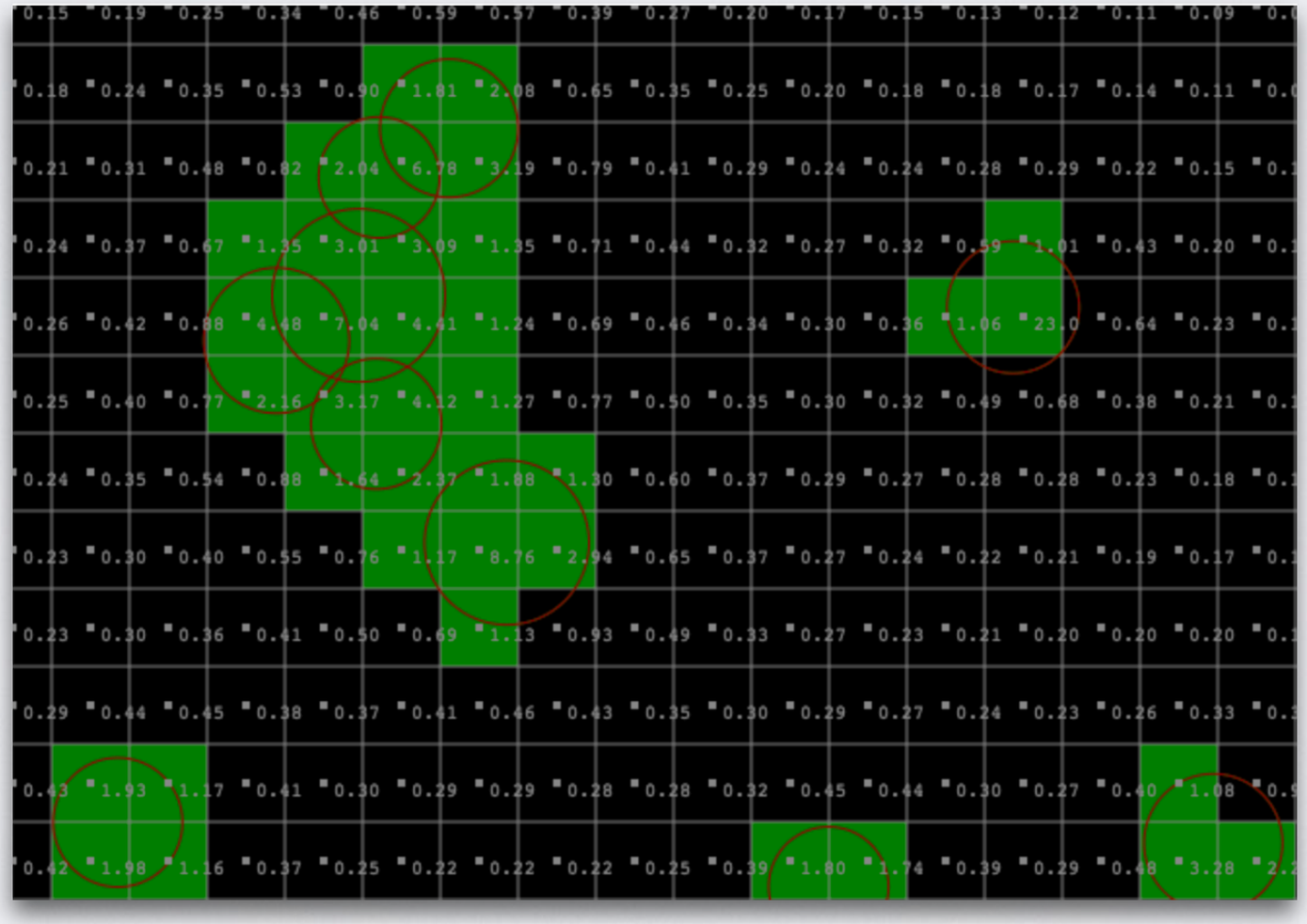


wikipedia (source)

- So how do we render these? We could use ray-marching (although its pretty slow because you can't really use the sphere-tracing optimization).
- Let's talk about another approach - **Marching cubes!**
- Supposedly the most-cited paper in graphics.
- Technique for not only rendering, but actually generating geometry mesh!

MARCHING SQUARES (2D)

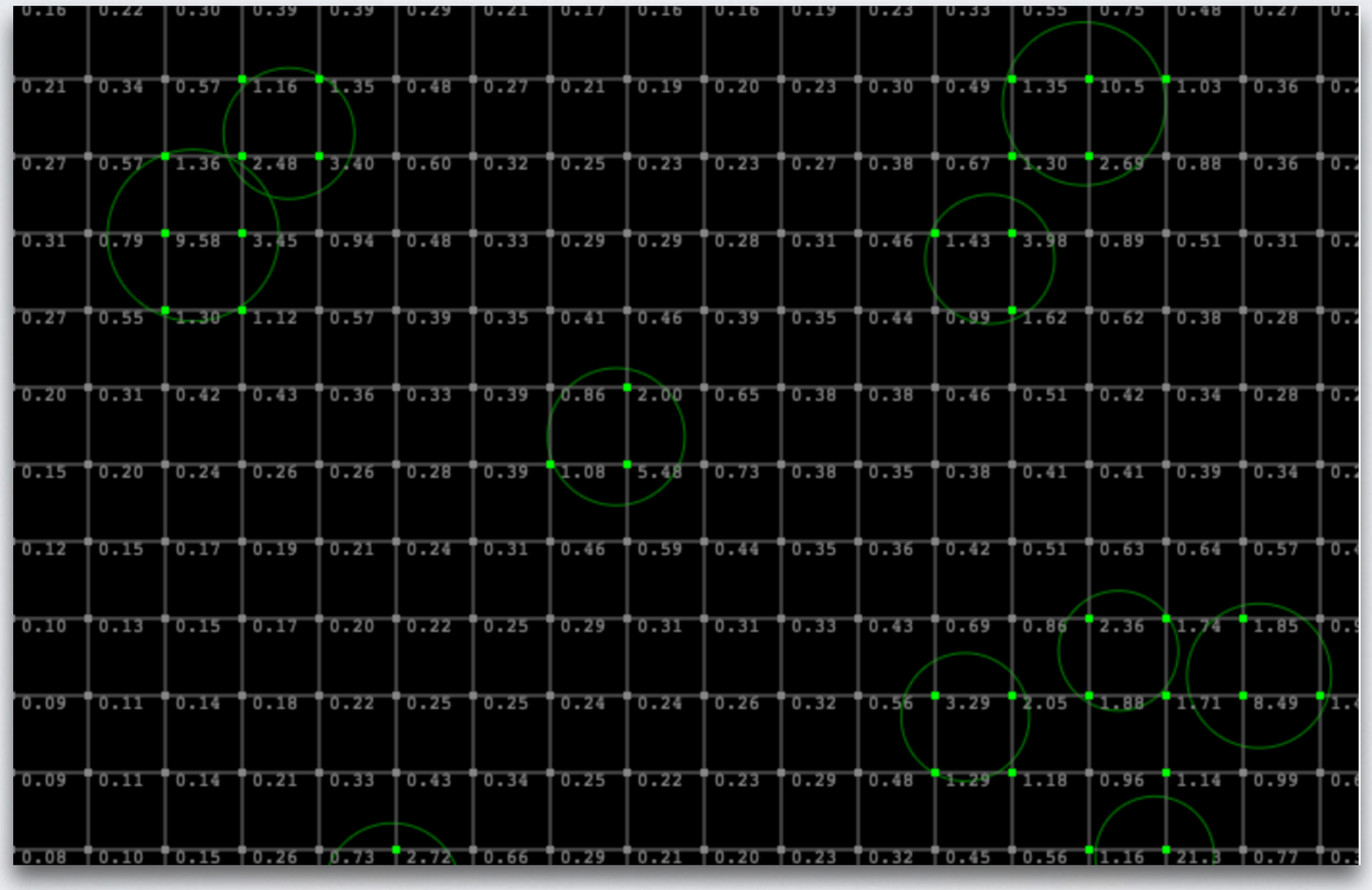
- Rather than ray-marching, we discretize our space with a uniform grid, then sample the function at the center of each cell.
- By summing up the influences with the metaball formula, we say cell is either inside or outside surface.
- But, this approach is messy because we're just categorizing cells inside or outside. Unless we make our grid super high resolution, it looks obviously blocky. Can we do better?



Jamie Wong (source)

MARCHING SQUARES

- Yes we can! Rather than sampling at the center of the cells, we can sample at the corners, and use that information to infer something about the geometry in each cell based on the corners. [below, green points are inside the metaball surface].

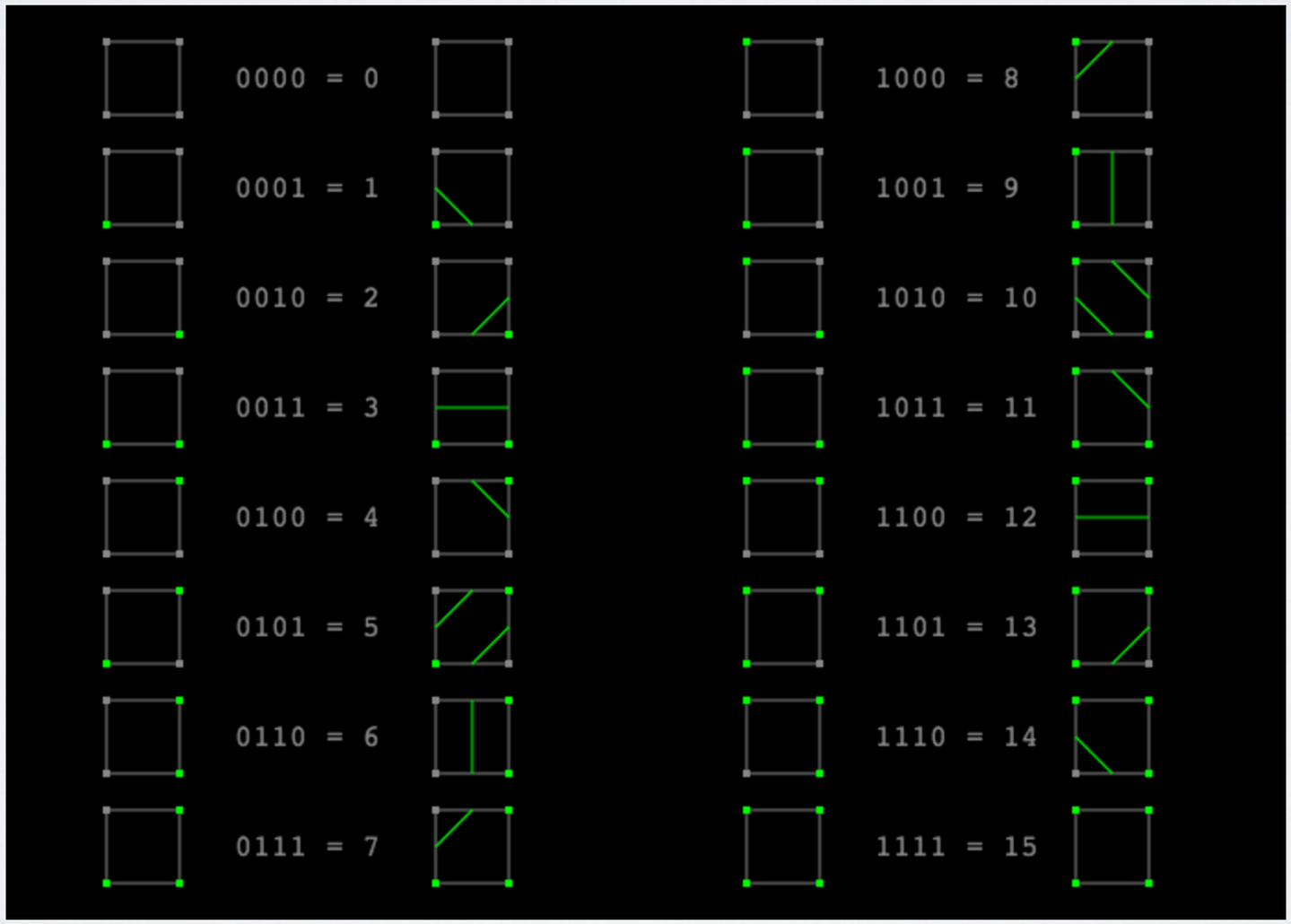


Jamie Wong (source)

MARCHING SQUARES

- If we know which corners are inside/outside, we know the surface geometry must lie somewhere between the inside corners and the outside ones.

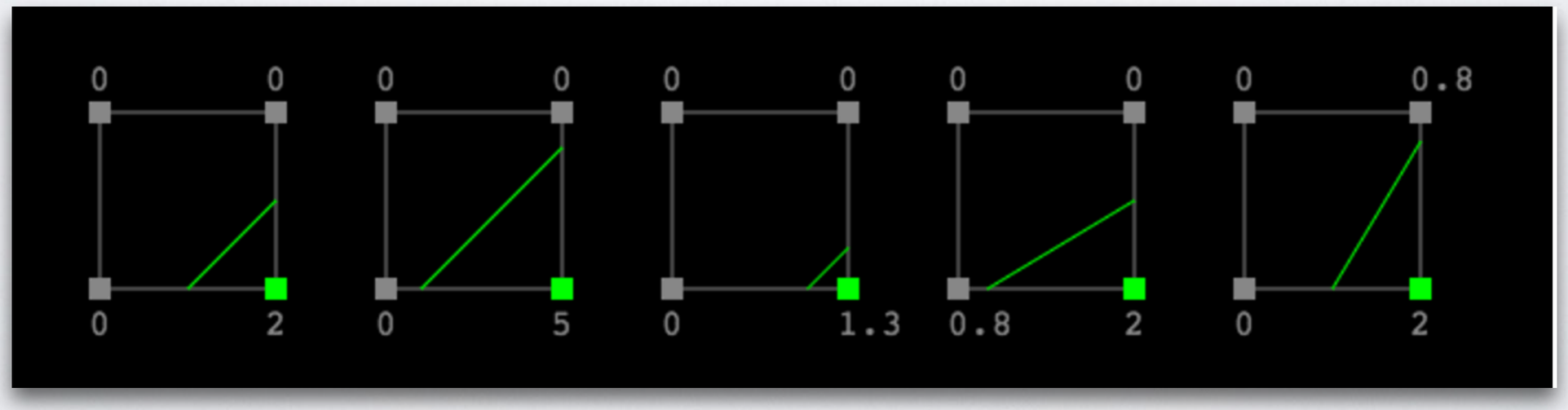
- Given four vertices, the number of possibilities is pretty manageable.



Jamie Wong (source)

MARCHING SQUARES

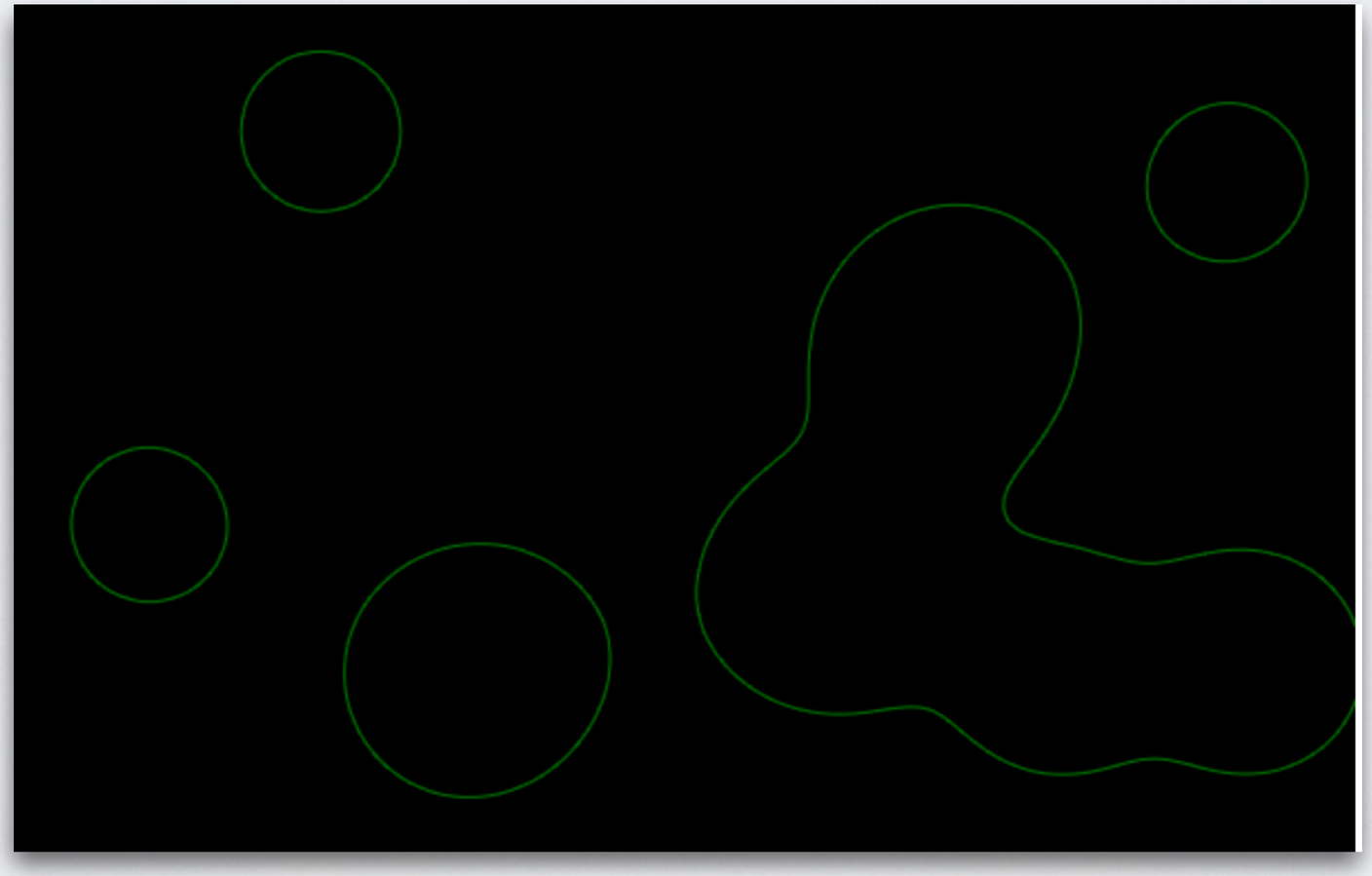
- But why settle for "somewhere"?
- We have exact isovalues at each corner of our cells. We want to find some intermediate value between them…. Smells like interpolation!
- Using interpolation, we can find exactly where the surface should be positioned (where the meatball function is one) along each edge.
- We can use this to generate the exact lines that should represent our surface.



Jamie Wong (source)

MARCHING SQUARES

- Crank up the resolution, and viola! You even have actually mesh vertices per grid cell!
- See here for animations of the whole process
- Works almost exactly the same in 3D, you just have cube cells with 8 vertices, and there are way more possible mesh configurations!



Jamie Wong (source)

IN SUMMARY

- An alternate method for representing surfaces: Signed Distance Function.

- SDFs return a point's distance from an object's surface. Zero means on the surface. Positive is outside. Negative is inside.

- Building with a set of basic SDFs for primitive geometry, we can transform and combine primitives to get many complex shapes.

- We can render SDFs with ray marching, basically stepping along a ray from the camera until we hit something or hit maximum.

REFERENCES

- Papers
    - Marching Cubes
- Helpful articles
    - IQs library of signed distances functions
    - Review of ray tracing
    - Great explanation of ray marching & SDF
    - Metaballs and marching squares
    - GPU Gems chapter on distance functions
    - Smooth min explained
    - Demo of many SDF operations