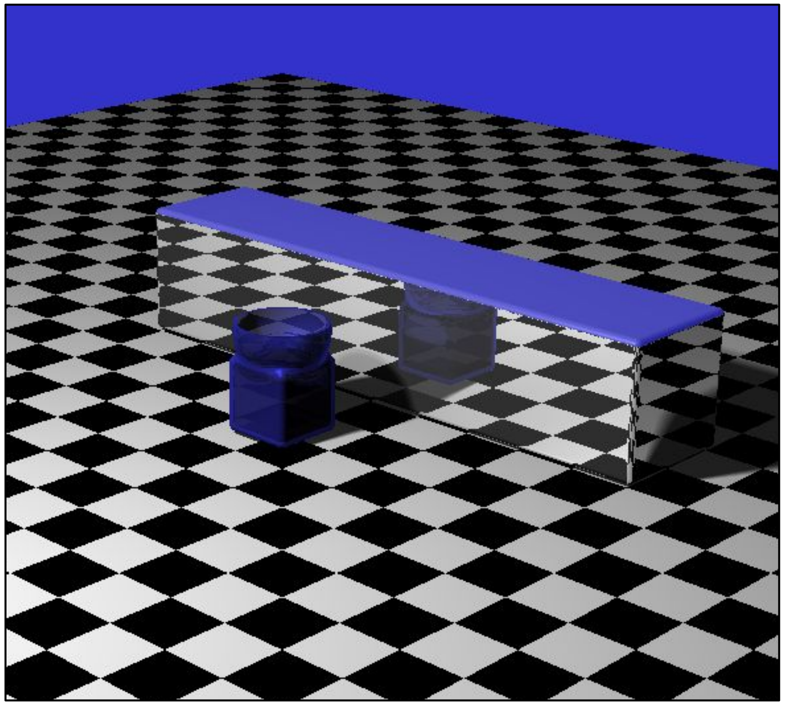
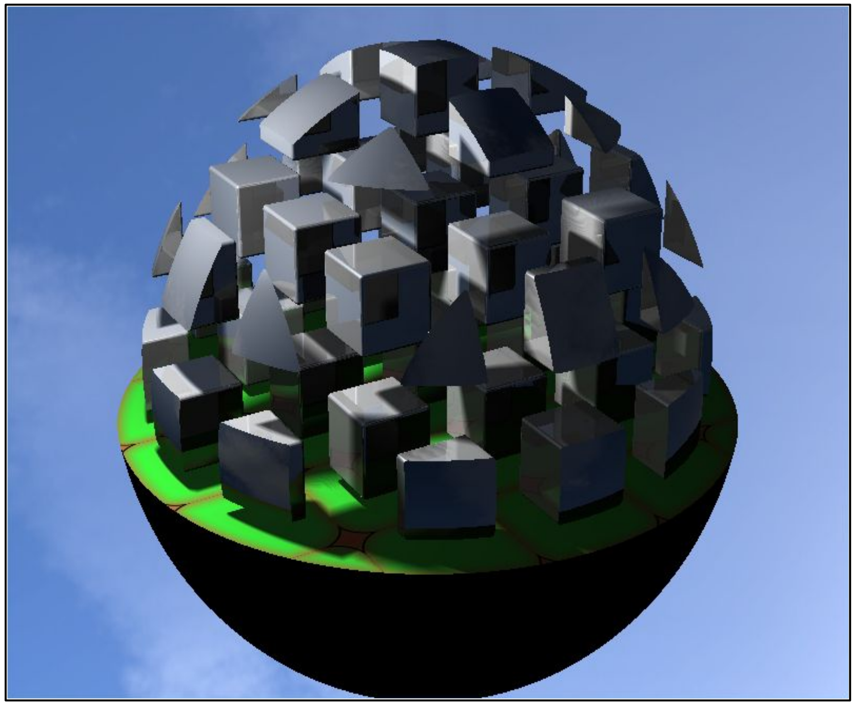
Advanced Graphics





Ray Marching and Advanced Scenes

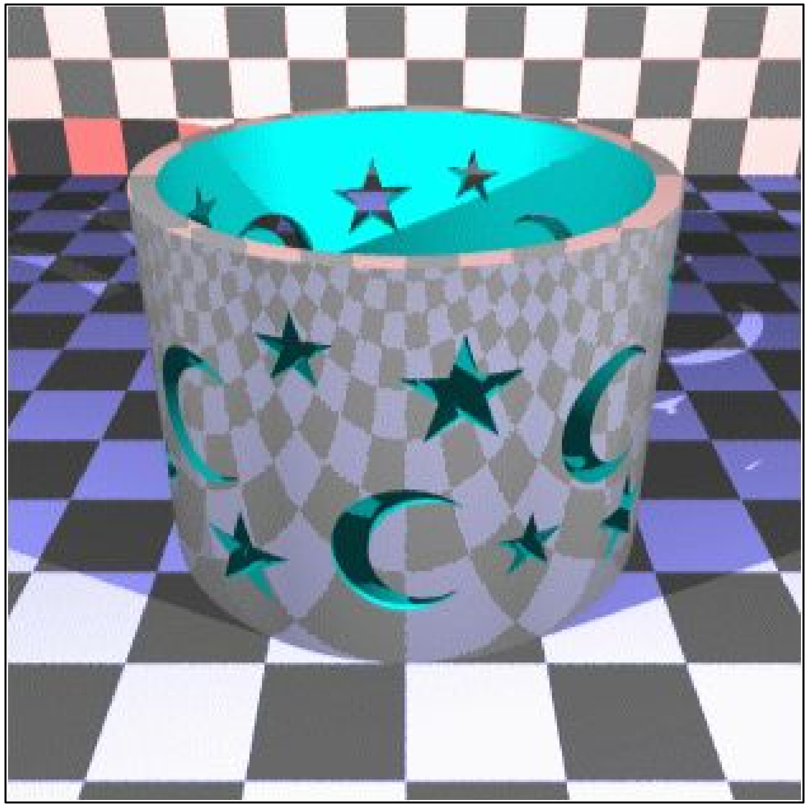
Alex Benton, University of Cambridge – A.Benton@damtp.cam.ac.uk

Supported in part by Google UK, Ltd

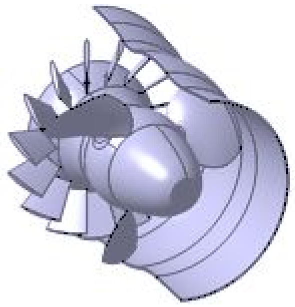
Constructive Solid Geometry

Constructive Solid Geometry (CSG) builds complicated forms out of simple primitives.

These primitives are combined with basic boolean operations: add, subtract, intersect.


CSG figure by Neil Dodgson

Constructive Solid Geometry



CSG models are easy to ray-trace but difficult to polygonalize

- Issues include choosing polygon boundaries at edges; converting adequately from pure smooth primitives to discrete (flat) faces; handling 'infinitely thin' sheet surfaces; and others.
- This is an ongoing research topic.

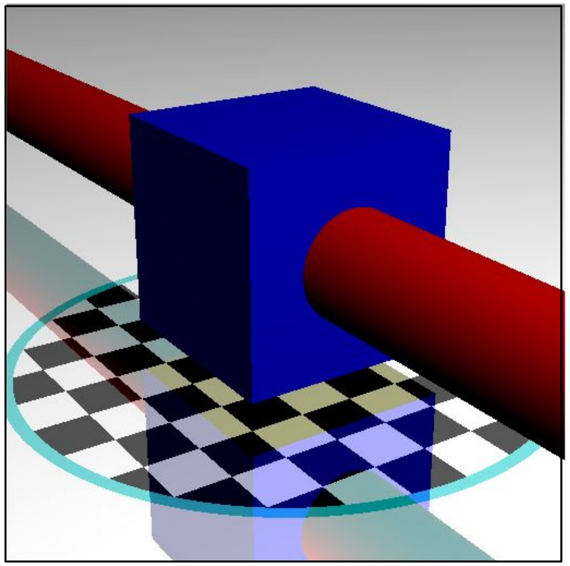
CSG models are well-suited to machine milling, automated manufacture, etc

- Great for 3D printers!

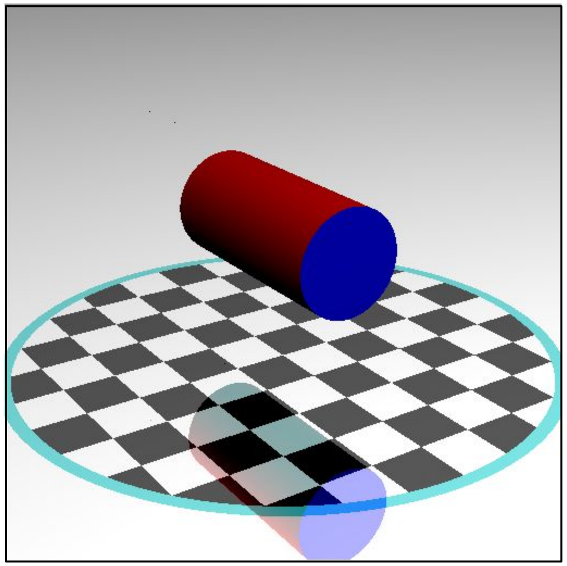
Constructive Solid Geometry

Three operations:

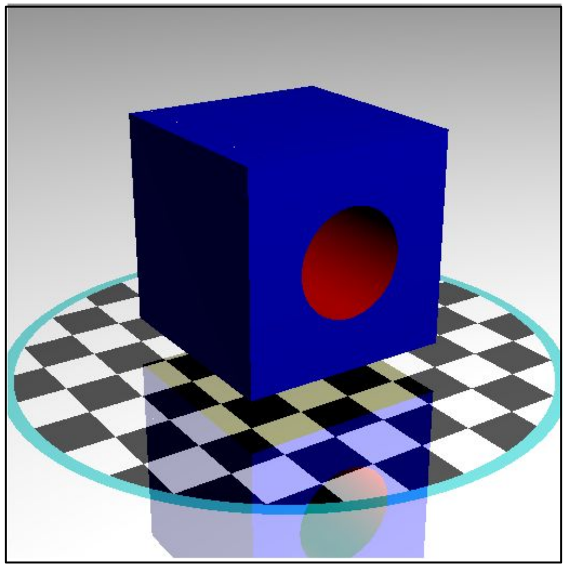
1. *Union*



2. *Intersection*



3. *Difference*



Constructive Solid Geometry

CSG surfaces can be described by a binary tree, where each leaf node is a primitive and each non-leaf node is a boolean operation.

(What would the *not* of a surface look like?)

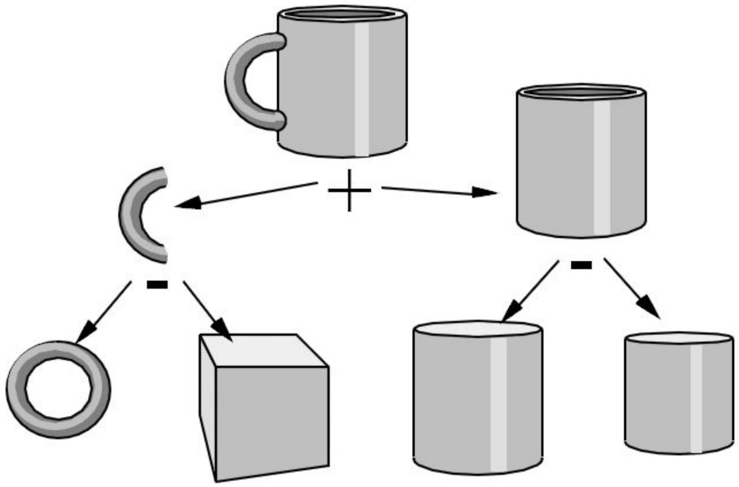
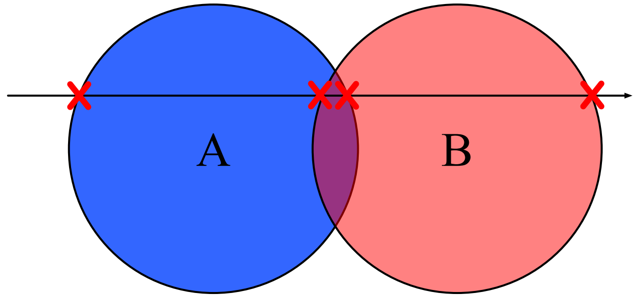


Figure from Wyvill (1995) part two, p. 4

Ray-tracing CSG models

For each node of the binary tree:



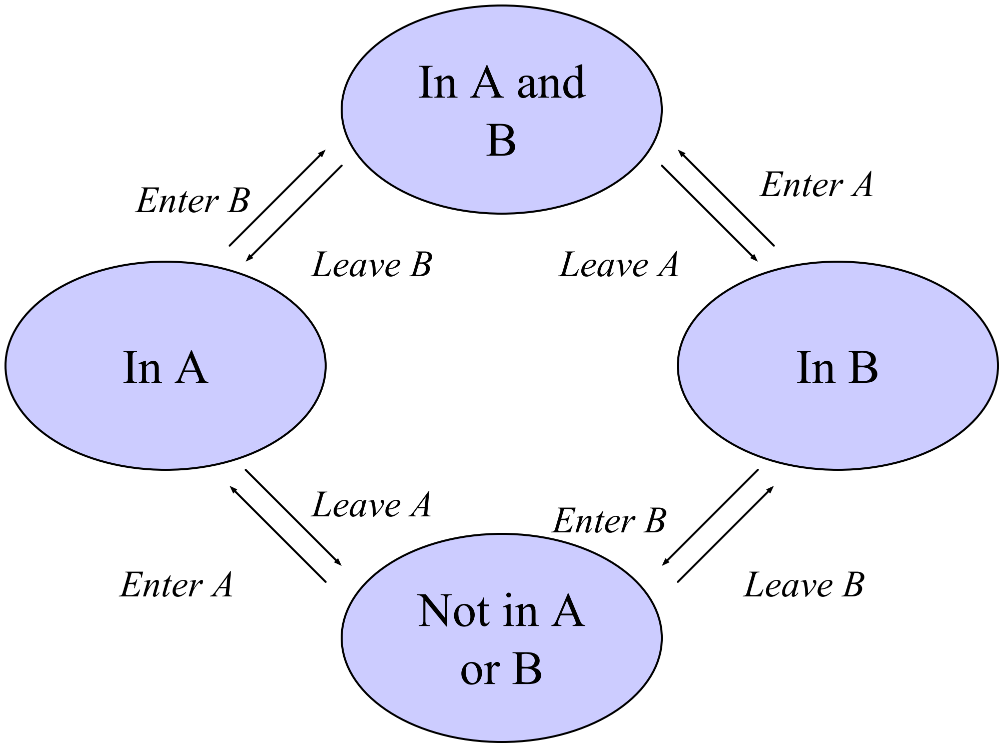
- Fire ray *r* at *A* and *B*.
- List in *t*-order all points where *r* enters of leaves *A* or *B*.
    - You can think of each intersection as a quad of booleans-- (*wasInA*, *isInA, wasInB, isInB*)
- Discard from the list all intersections which don't matter to the current boolean operation.
- Pass the list up to the parent node and recurse.

Ray-tracing CSG models

Each boolean operation can be modeled as a state machine.

For each operation, retain those intersections that transition into or out of the critical state(s).

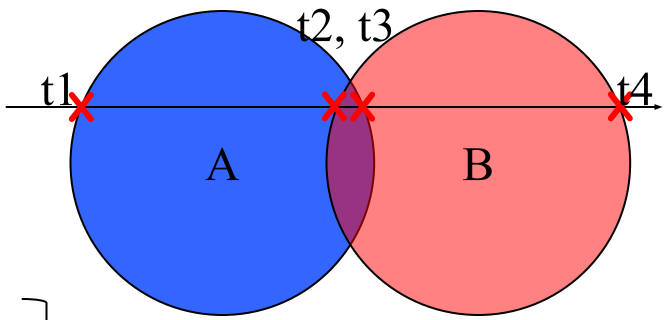
- Union: `{In A | In B | In A and B}`
- Intersection: `{In A and B}`
- Difference: `{In A}`



Ray-tracing CSG models

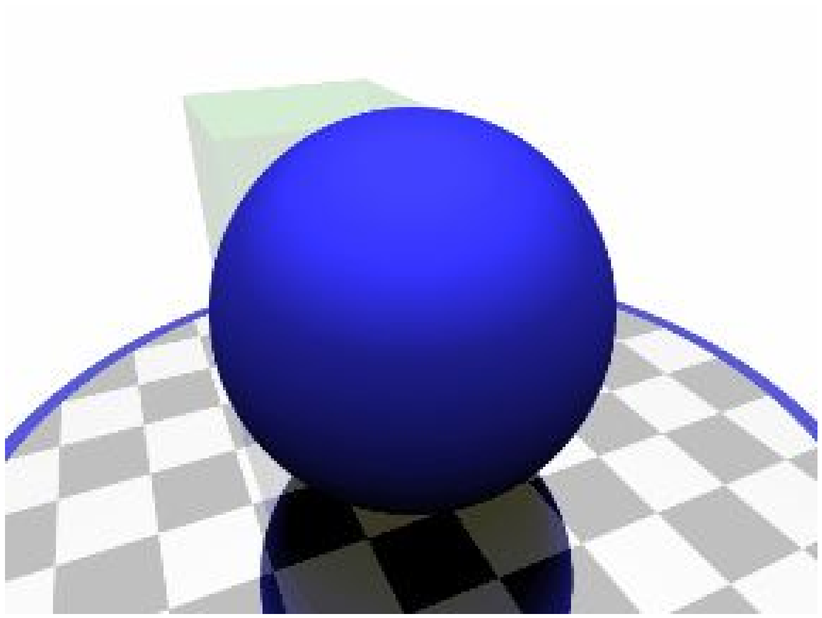
Example: Difference (A-B)

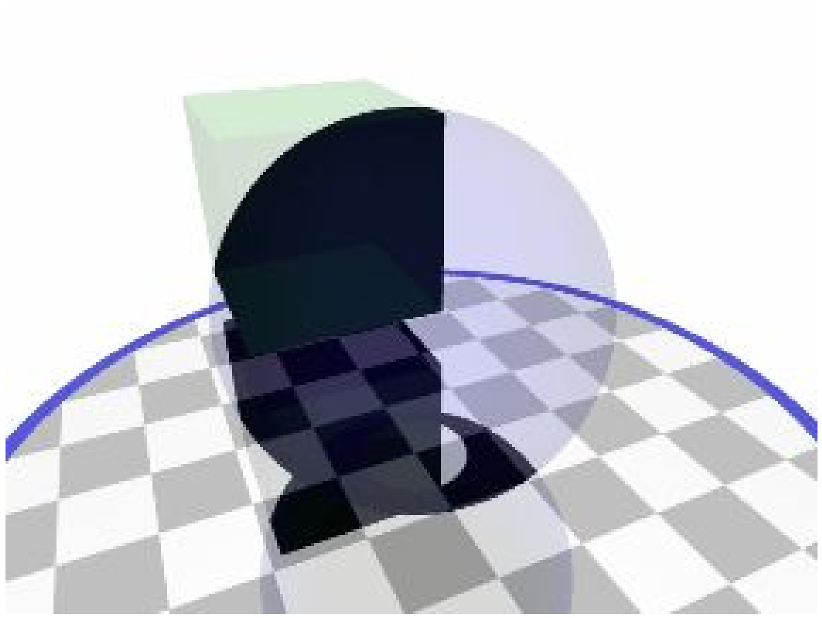
A-B	Was In A	Is In A	Was In B	Is In B
t1	No	Yes	No	No
t2	Yes	Yes	No	Yes
t3	Yes	No	Yes	Yes
t4	No	No	Yes	No



```
difference =
((wasInA != isInA) &&
 (!isInB)&&(!wasInB))
||
((wasInB != isInB) &&
 (wasInA || isInA))
```

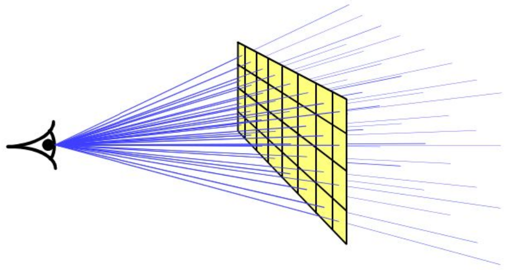
CSG in action


Difference

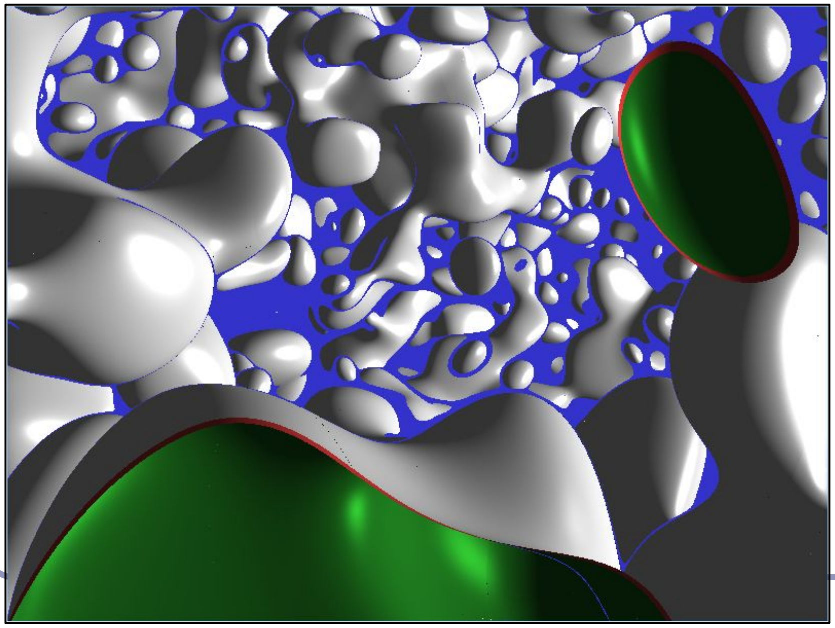

Intersection

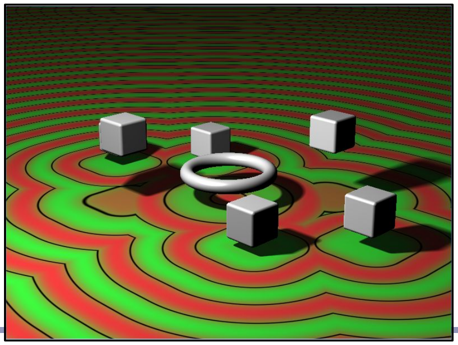
GPU Ray-tracing

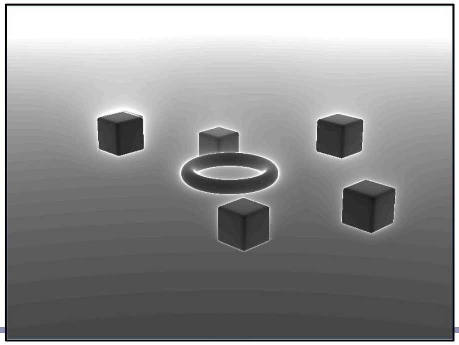
Ray tracing 101: “Choose the color of the pixel by firing a ray through and seeing what it hits.”



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

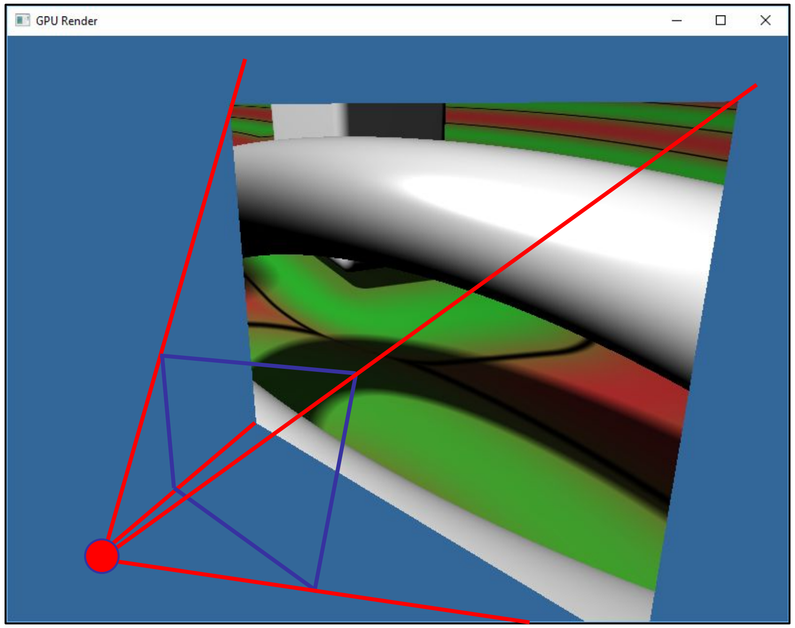






GPU Ray-tracing

1. Use a minimal fragment shader (no transforms)
2. Set up OpenGL with minimal geometry, a single quad
3. Bind a `vec2` to each vertex specifying ‘texture’ coordinates
4. Implement raytracing in GLSL per pixel:
   a. For each pixel, compute the ray from the eye through the pixel, using the interpolated texture coordinate to identify the pixel
   b. Run the ray tracing algorithm for every ray



GPU Ray-tracing

```
// Window dimensions
uniform vec2 iResolution;
// Camera position
uniform vec3 iRayOrigin;
// Camera facing direction
uniform vec3 iRayDir;
// Camera up direction
uniform vec3 iRayUp;
// Distance to viewing plane
uniform float iPlaneDist;

// 'Texture' coordinate of each
// vertex, interpolated across
// fragments
in vec2 texCoord;
```

```
vec3 getRayDir(
    vec3 camDir,
    vec3 camUp,
    vec2 texCoord) {
  vec3 xAxis = normalize(
      cross(camDir, camUp));
  vec2 p = 2.0 * texCoord - 1.0;
  p.x *= iResolution.x
      / iResolution.y;
  return normalize(
      p.x * xAxis
      + p.y * camUp
      + iPlaneDist * camDir);
}
```

GPU Ray-tracing

```
Hit traceSphere(vec3 rayorig, vec3 raydir, vec3 pos, float radius) {
  float OdotD = dot(rayorig - pos, raydir);
  float OdotO = dot(rayorig - pos, rayorig - pos);
  float base = OdotD * OdotD - OdotO + radius * radius;

  if (base >= 0) {
    float root = sqrt(base);
    float t1 = -OdotD + root;
    float t2 = -OdotD - root;
    if (t1 >= 0 || t2 >= 0) {
      float t = (t1 < t2 && t1 >= 0) ? t1 : t2;
      vec3 pt = rayorig + raydir * t;
      vec3 normal = normalize(pt - pos);
      return Hit(pt, normal, t);
    }
  }
  return Hit(vec3(0), vec3(0), -1);
}
```

GPU Ray-tracing

One key limitation of some GLSL platforms (specifically GLSL ES, for mobile devices and WebGL) is that **GLSL may not support recursion**. That makes recursing to find reflected / refracted /transparency colors difficult.

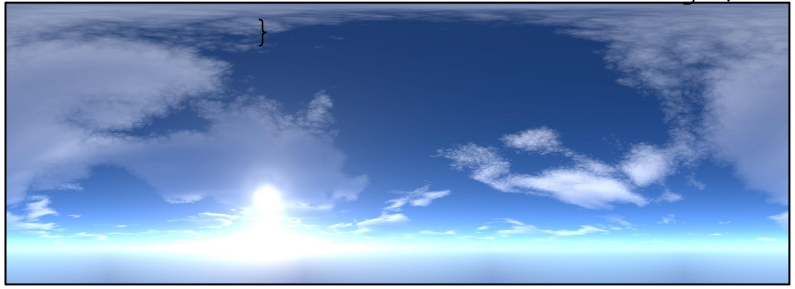
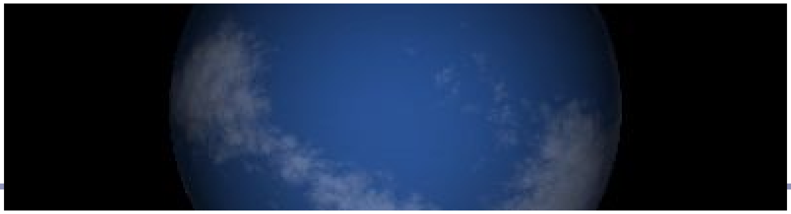
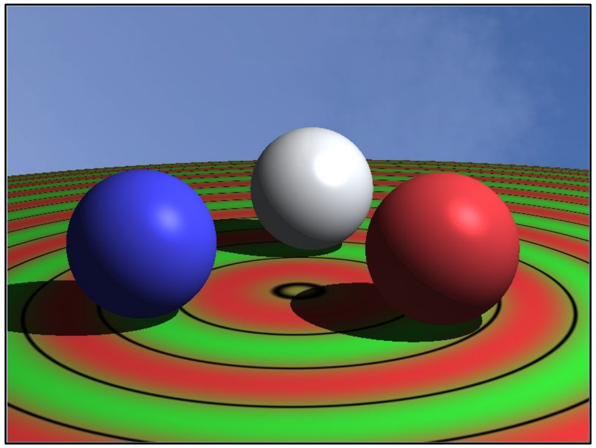
We can work around this by treating the illumination equation as a weighted polynomial, where the weight of each blended contribution is computed before the contribution itself.

```
struct TBD {
  vec3 src;
  vec3 dir;
  float weight;
};
```

```
vec3 renderScene(vec3 rayorig, vec3 raydir) {
  TBD tbd[10];
  int numTbd = 0;
  vec3 cumulativeColor = vec3(0);

  tbd[numTbd++] = TBD(rayorig, raydir, 1.0);
  for (int i = 0; i < 10 && numTbd > 0; i++) {
    color = // fire ray, compute local color
    cumulativeColor += tbd[i].weight * color;
    tbd[numTbd++] = // reflection ray
    tbd[numTbd++] = // refraction ray
    ...
```

Textured skies

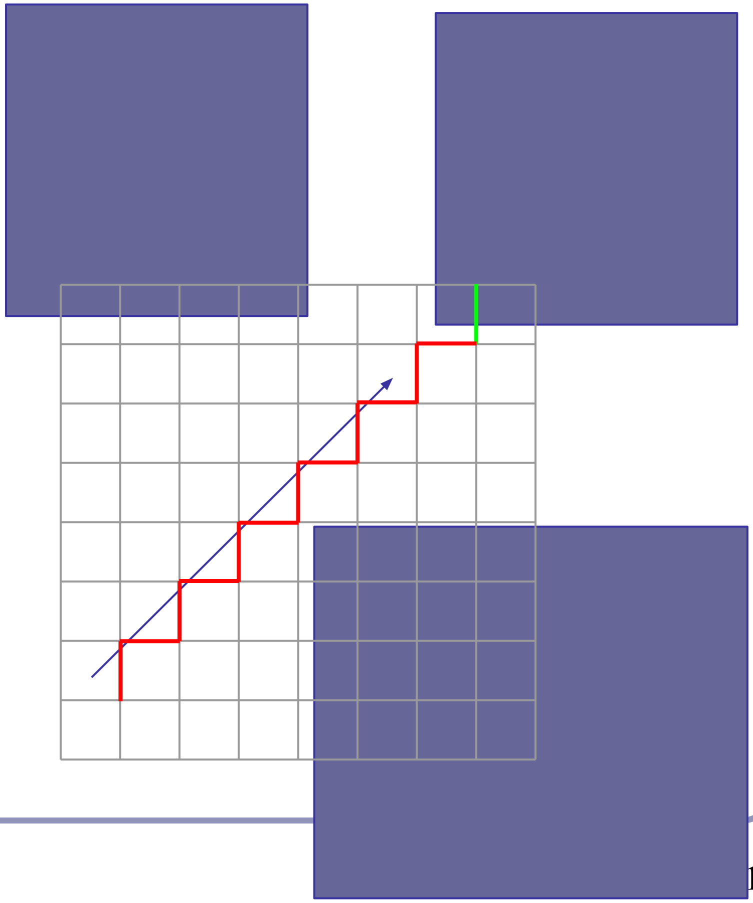
```
#define PI 3.14159
uniform sampler2D texture;

vec3 getBackground(vec3 dir) {
  float u = 0.5 + atan(dir.z, -dir.x) / (2 *
PI);
  float v = 0.5 - asin(dir.y) / PI;
  vec4 texColor = texture2D(texture, vec2(u,
v));
  return texColor.rgb;
}
```

An alternative to raytracing: *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.

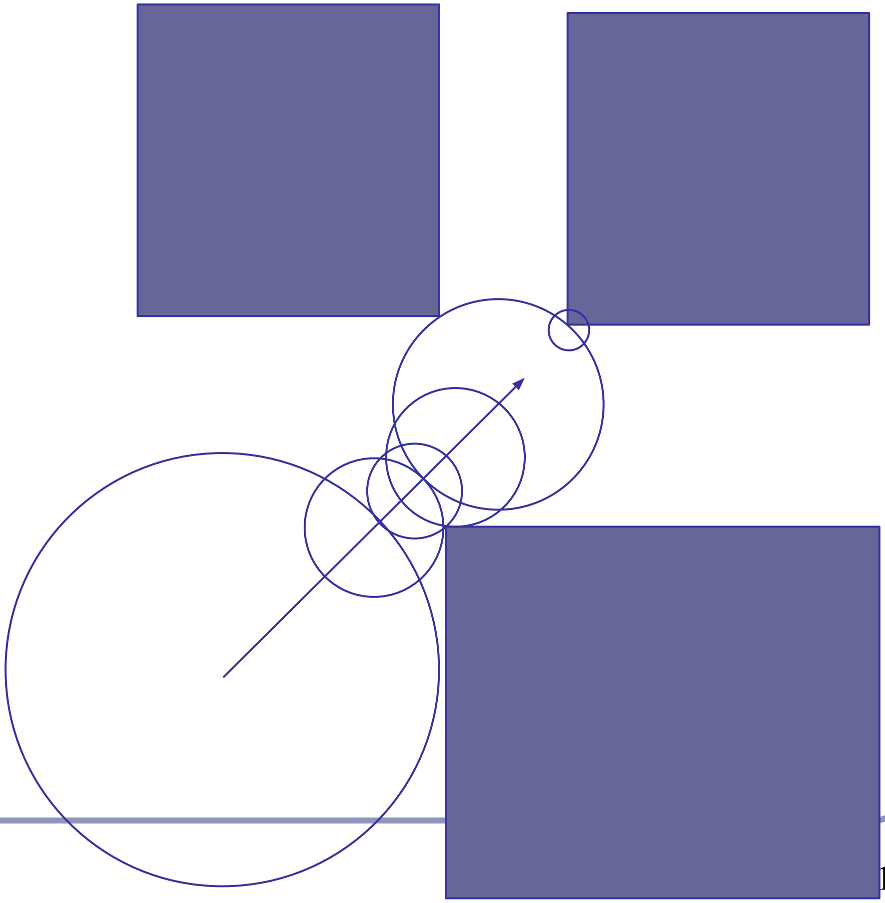
- Also sometimes called ray casting
- Scene objects only need to answer, *"has this ray hit you? y/n"*
- Great solution for data like height fields
- Unfortunately…
  - often involves many steps
  - too large a step size can lead to lost intersections (step over the object)
  - an `if()` test in the heart of a `for()` loop is very hard for the GPU to optimize



GPU Ray-marching: 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 functions

The theory is simple: the SDF computes the minimum possible distance to the surface.

The sphere, for instance, is the distance from p to the center of the sphere, less 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, vec3 c) {
  vec3 d = abs(p) - c;
  return min(max(d.x,
      max(d.y, d.z)), 0.0)
      + length(max(d, 0.0));
}

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

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

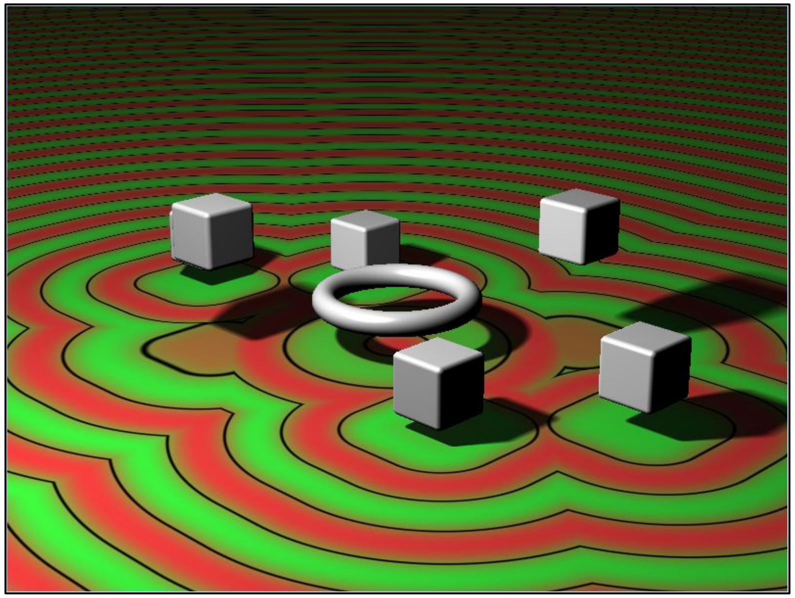
http://iquilezles.org/www/articles/distfunctions/distfunctions.htm

Raymarching signed distance fields

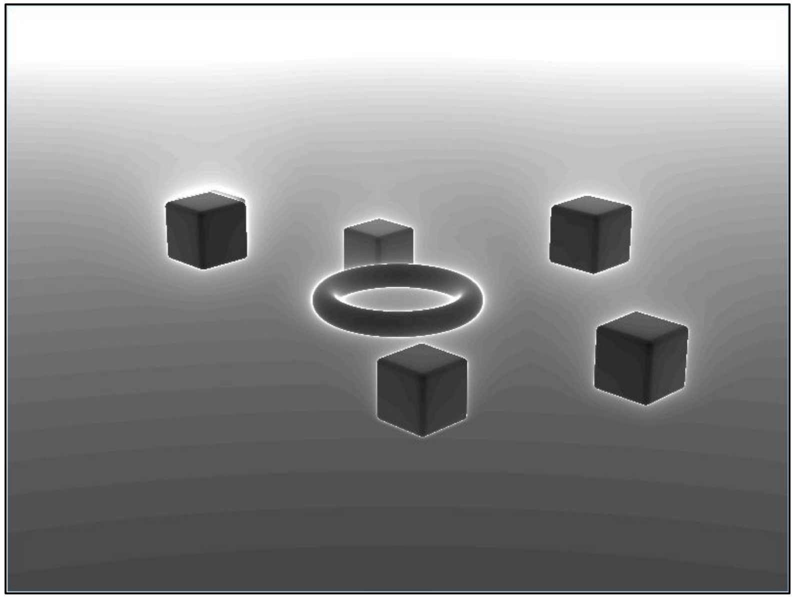
```
vec3 raymarch(vec3 pos, vec3 raydir) {
  int step = 0;
  float d = getSdf(pos);

  while (abs(d) > 0.001 && step < 50) {
    pos = pos + raydir * d;
    d = getSdf(pos);  // Return sphere(pos) or any other
    step++;
  }

  return
      (step < 50) ? illuminate(pos, rayorig) : background;
}
```

Visualizing step count

Final image



Distance field



Brighter = more steps, up to 50

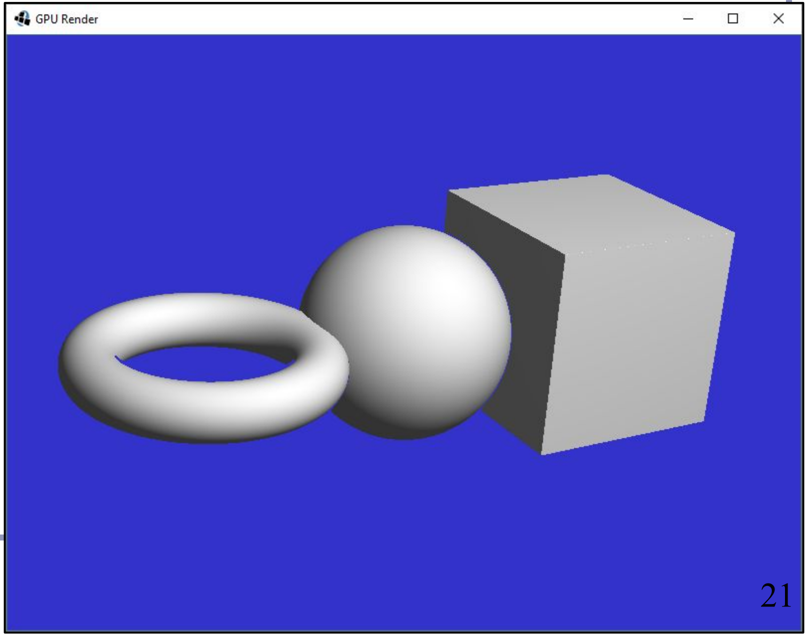
Find the normal to an SDF

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.

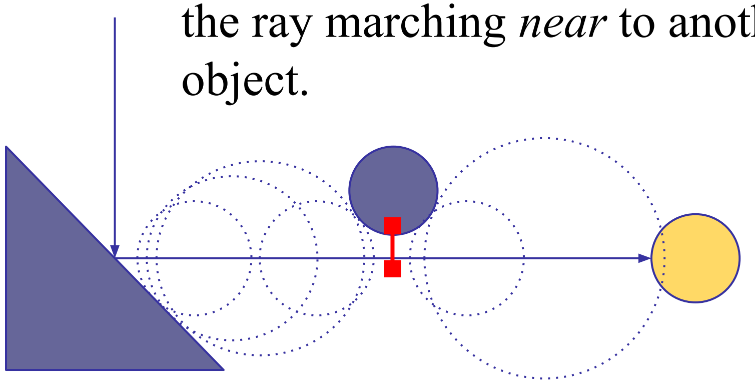
Thus the local gradient (the normal) can be approximated from the distance function.

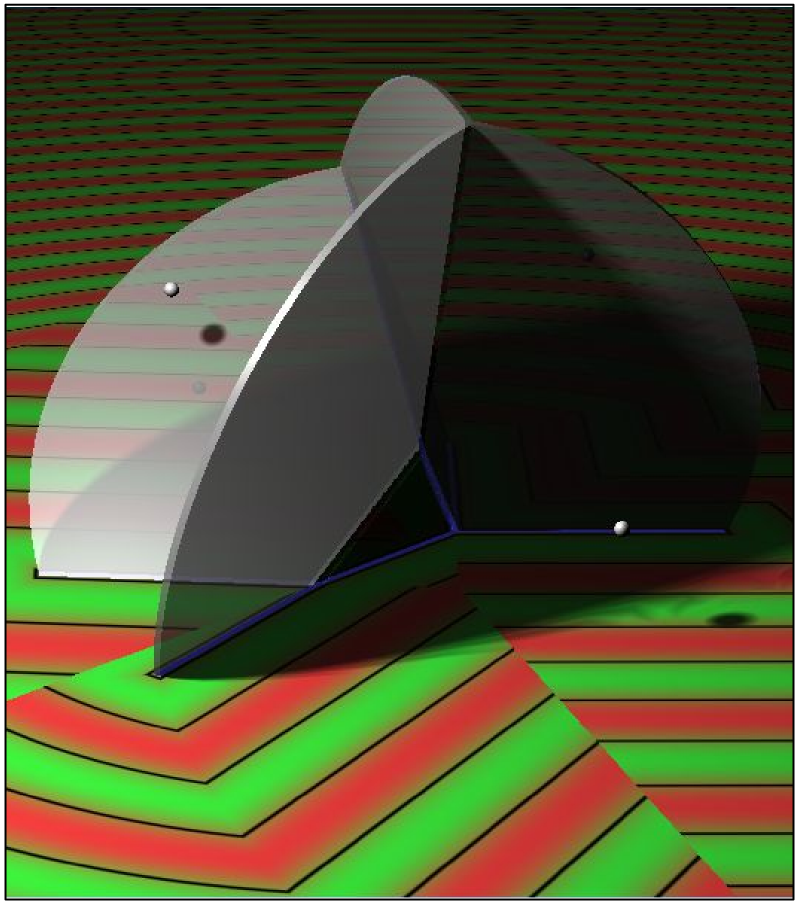


SDF shadows

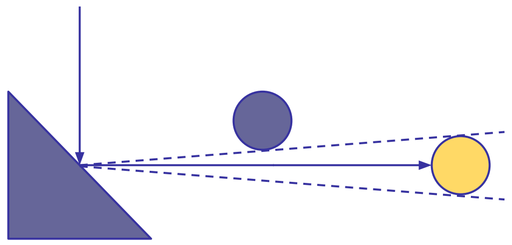
Ray-marched shadows are straightforward: march a ray towards each light source, illuminate if the SDF ever drops too close to zero.

Unlike ray-tracing, soft shadows are almost free with SDFs: attenuate illumination by a linear function of the ray marching *near* to another object.



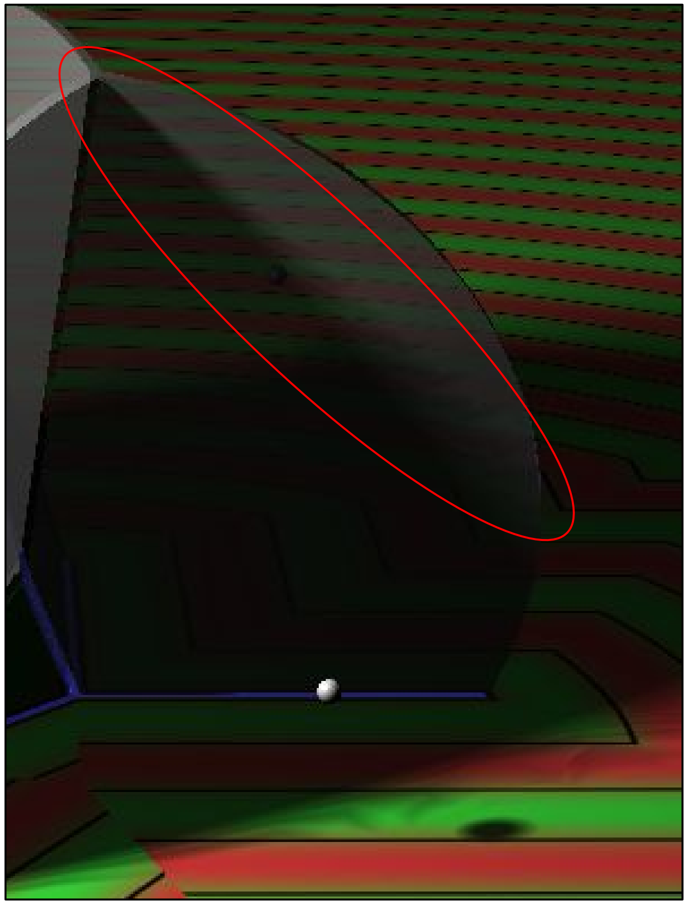


Soft SDF shadows



```
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;
}
```

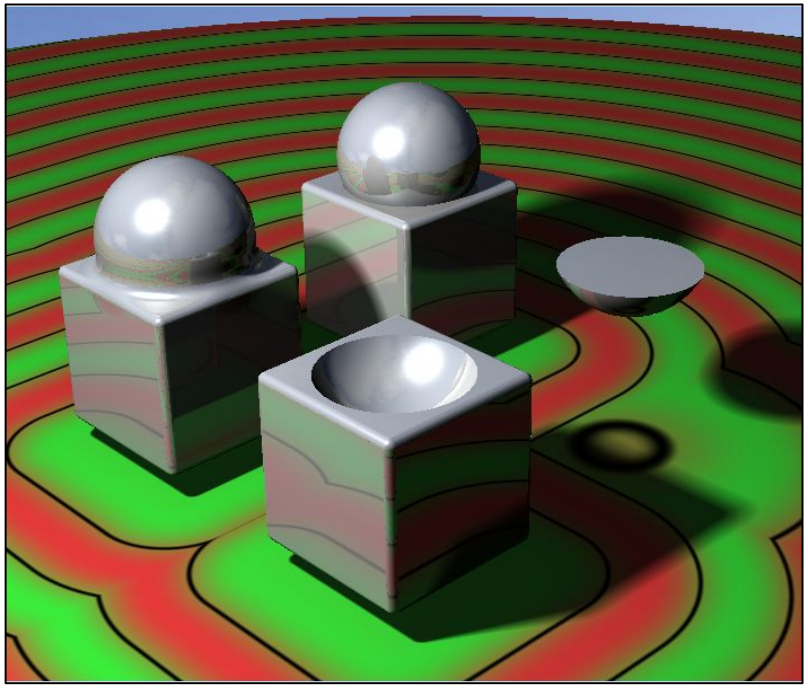
By dividing *d* by *t*, we attenuate the strength of the shadow as its source is further from the illuminated point.



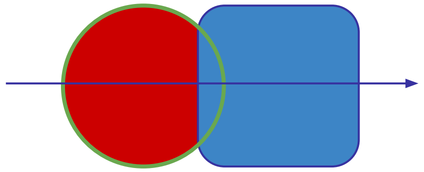
Combining SDFs

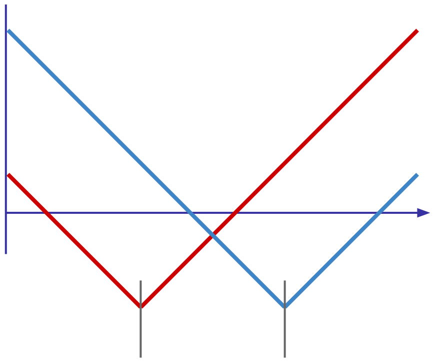
We combine SDF models by choosing which is closer to the sampled point.

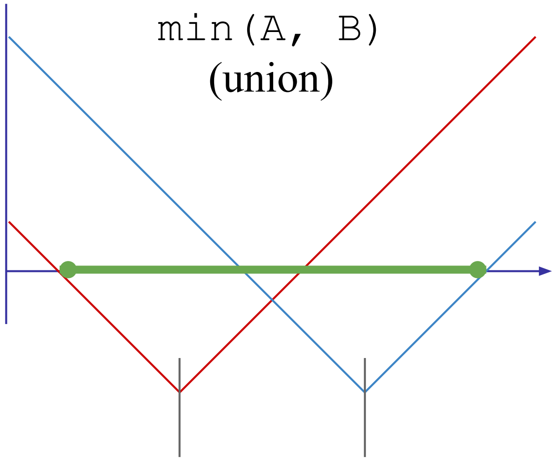
- Take the **union** of two SDFs by taking the `min()` of their functions.
- Take the **intersection** of two SDFs by taking the `max()` of their functions.
- The `max()` of function A and the negative of function B will return the **difference** of A - B.

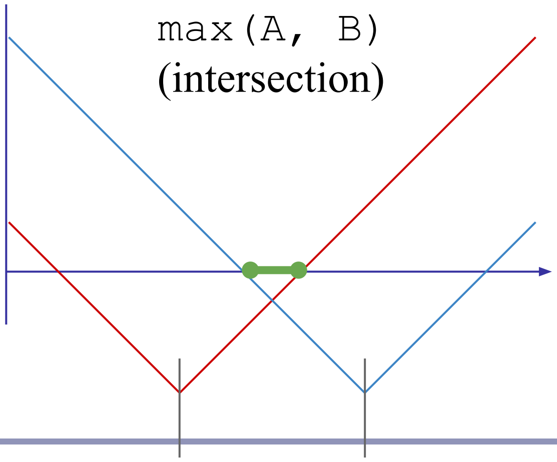


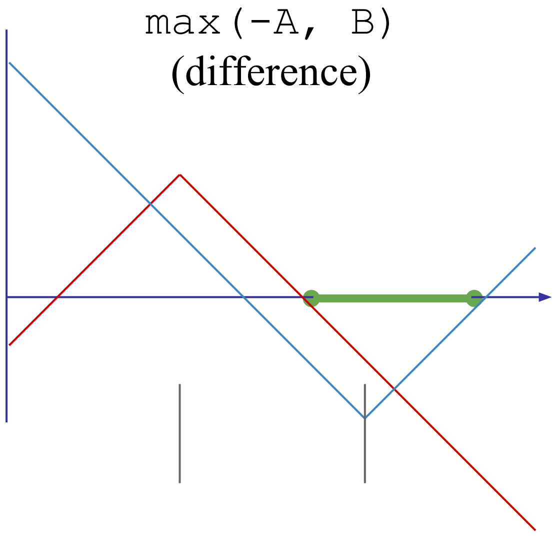
Combining SDFs





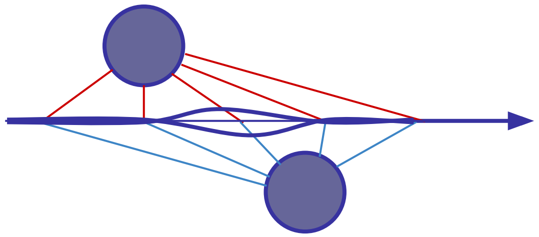


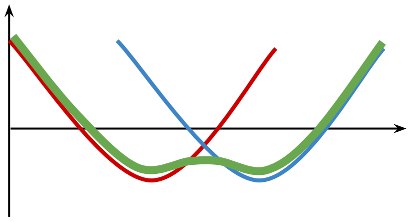


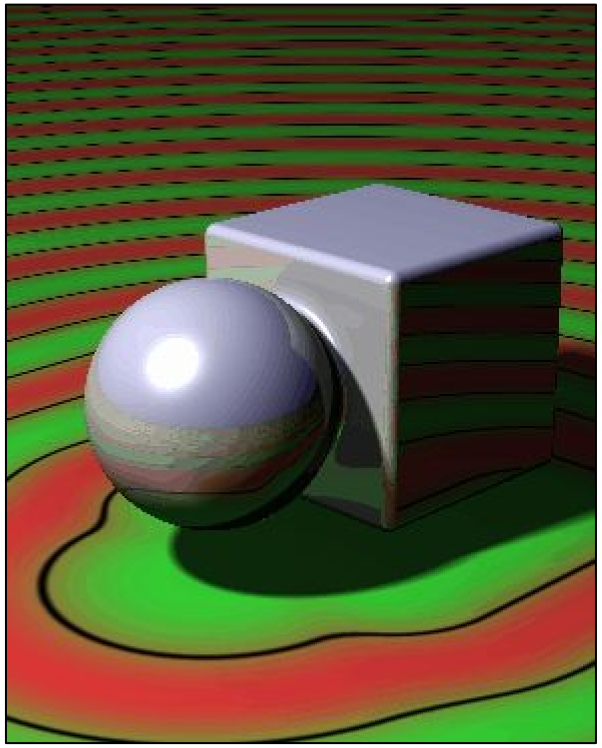


Blending SDFs

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 blend(float a, float b, float k) {
  a = pow(a, k);
  b = pow(b, k);
  return pow((a * b) / (a + b), 1.0 / k);
}
```

Transforming SDF geometry

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

Ex:

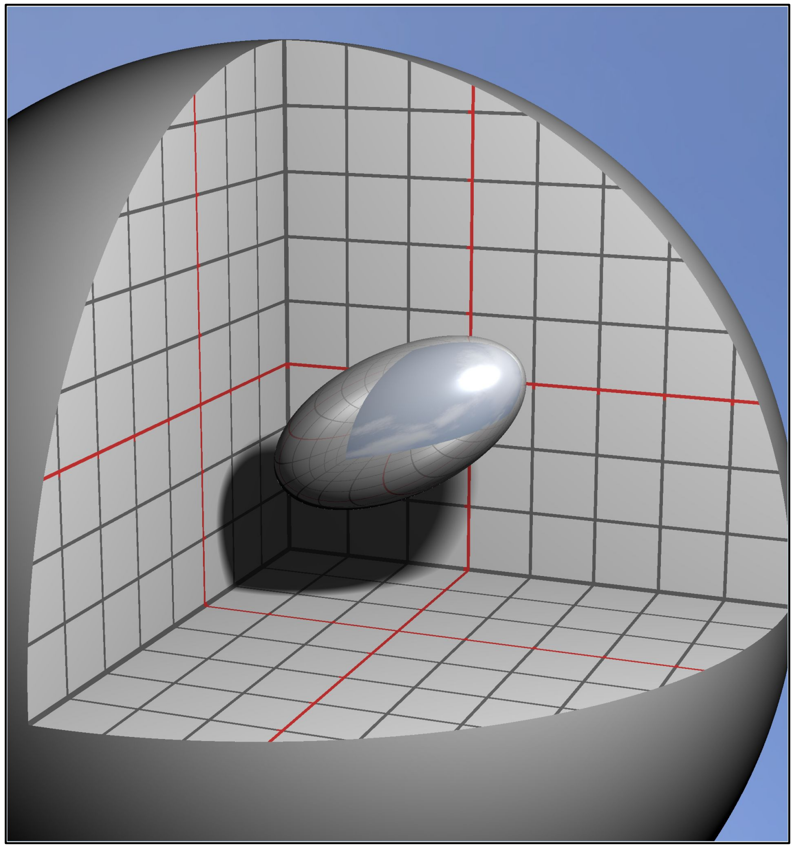
```
float sphere(vec3 pt, float radius) {
  return length(pt) - radius;
}

float f(vec3 pt) {
  return sphere(pt - vec3(0, 3, 0));
}
```

This renders a sphere centered at `(0, 3, 0)`.

More prosaically, assemble your local-to-world transform as usual, but apply its inverse to the pt within your distance function.

Transforming SDF geometry

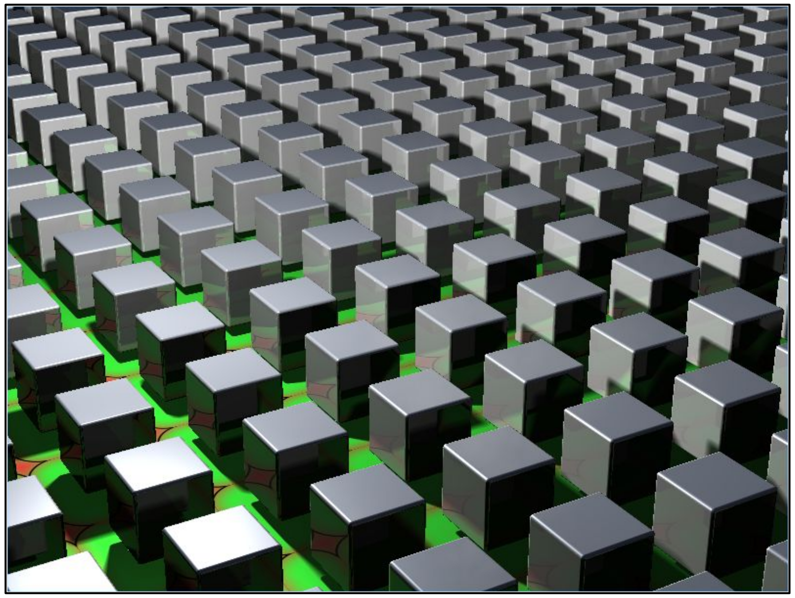
```
float fScene(vec3 pt) {

  // 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);
}
```



Repeating SDF geometry

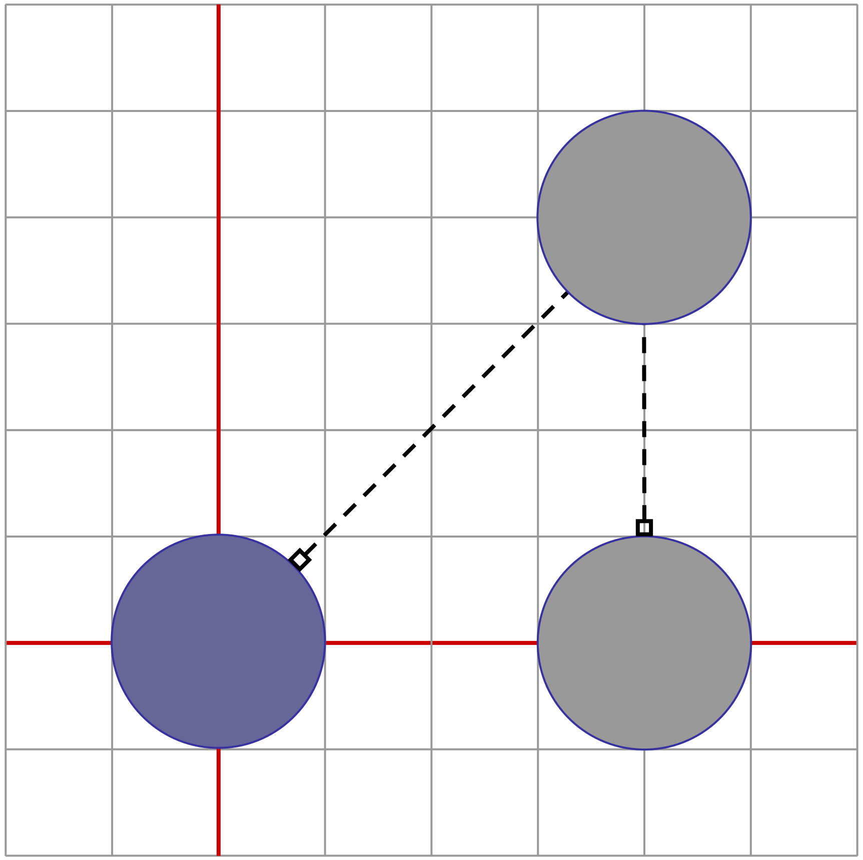
If we take the modulus of a point's position along one or more axes before computing its signed distance, then we segment space into infinite parallel regions of repeated distance. Space near the origin 'repeats'.

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));
}
```

Repeating SDF geometry



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

-
  ```
  sdSphere(4, 4)
    = √(4*4+4*4) - 1
    = ~4.5
  ```

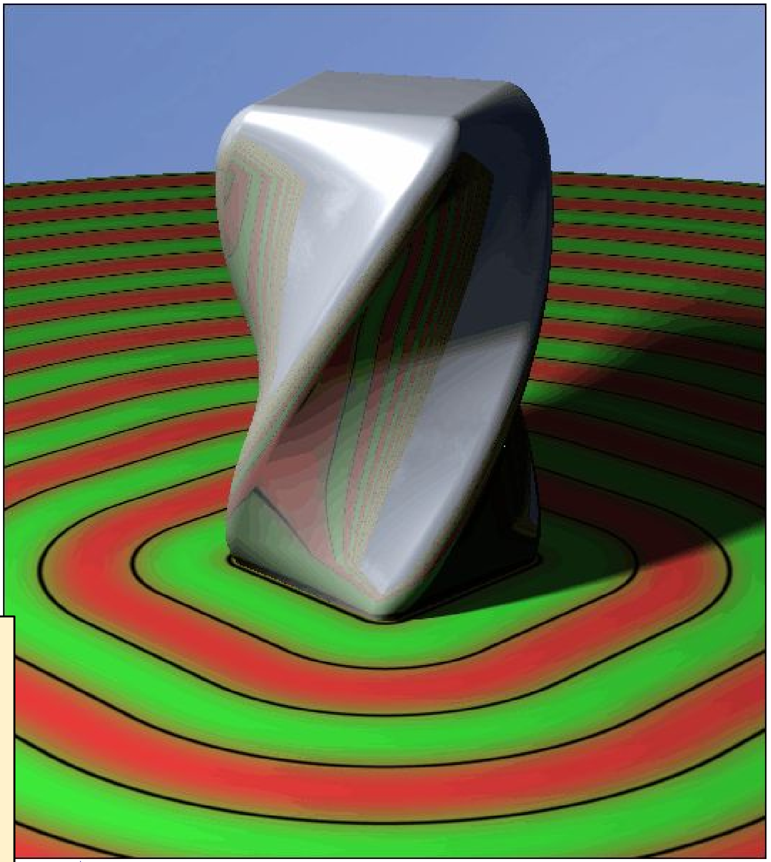
-
  ```
  sdSphere(
        ((4 + 2) % 4) - 2, 4)
    = √(0*0+4*4) - 1
    = 3
  ```

-
  ```
  sdSphere(
        ((4 + 2) % 4) - 2,
        ((4 + 2) % 4) - 2)
    = √(0*0+0*0) - 1
    = -1 // Inside surface
  ```

Transforming SDF geometry

The previous example modified 'all of space' with the same transform, so its distance functions retain their local linearity.

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



```
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));
}
```

Recommended reading

Seminal papers:

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

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