Game Graphics & Real-time Rendering

CMPM 163, W2018

Prof. Angus Forbes (instructor)
angus@ucsc.edu

Lucas Ferreira (TA)
lferreira@ucsc.edu

creativecoding.soe.ucsc.edu/courses/cmpm163
github.com/CreativeCodingLab

Last week

- Homework #2 introduced – Due Feb. 18th at 12noon

- Voronoi cells
- Drawing 2D shapes using signed distance functions (SDFs)
- Drawing nice looking text using SDFs
- https://creativecoding.soe.ucsc.edu/courses/cmpm163/code/week5_codeExamples.zip

This week

- Calculating normals from a dynamic heightmap
- High-level overview of coordinate systems
- Composing with noise
- Raymarching 3D objects
- Shading 3D SDFs
- Morphing between SDFs
- Links to tutorials + Code examples in slides

Future class

- Visit from Manu Mathew Thomas, researcher at Intel Corp.
    - Will talk about Global Illumination and a project connecting Deep Learning with rendering 3D graphics.
- Visit from Larry Cuba, pioneering computational artist
    - Created some of the very first artworks using computer graphics (lives in Santa Cruz!)

Coordinate systems

- World coordinates
- Object coordinates
- Light coordinates
- Eye / Camera coordinates

A coordinate system defines the objects in the scene according to a particular origin (0,0,0).

You can "transform" from one coordinate system to another using matrix multiplication. You get to decide which coordinate system to work with.

Coordinate systems

A matrix usually encodes a *translation*, and also can encode a scale or a rotation. The translation operation requires an “affine” transformation.

The rotation and scaling operations can act on 3D vectors, ie by multiplying a 3D vector by a 3D matrix that encodes a translation. The output of this operation is a new 3D vector.

A 4x4 matrix can additionally encode a translation operation, where the last column indicates the translation. This 4x4 matrix outputs position in “homogenous coordinates” – which is a topic for another day.

Coordinate systems

But the idea is that you can concatenate all of your operations into a single 4x4 matrix, and then multiply it by a particular point.

The 4^{th} element in the vector indicates if you are considering your vector a Point (ie with position information) or a Vector (which only has orientation and magnitude).

Coordinate systems

A more complicated matrix is the Projection Matrix, which transforms your points into a 2D space defined by the camera values: fovy, aspect ratio, and near and far plane.

After this operation, the 3rd element will contain the "depth" of the point as a value 0.0 to 1.0, in terms of where it sits in relation to the near and far plane.

Coordinate systems

If the value is < 0.0 or > 1.0, it will *not* be rendered (ie, not passed to the fragment shader).

If DEPTH TESTING is turned on, then WebGL will be smart enough to ignore any points that are blocked by other points.

(Which is why we needed to turn off depth testing in our transparent point sprite example.)

Camera coordinates

For Phong shading, we need all of our values in Camera Coordinates, as the relation of the position+orientation of the camera affects specular highlighting.

For geometry that is displaced dynamically in the vertex shader, eg by a noise function or a height map, we will need to recalculate the normals if we want to utilize a lighting method that requires normals (ie, pretty much all lighting models)

How?

Updating normals in a shader

For most meshes, each point is connected to 6 different triangles.

- We can calculate the normal by taking the cross product of one of the attached triangles.

- A only slightly more sophisticated version calculates the normal for all 6 of these triangles and takes the average.

- The normal is not a point, but a vector, and and an affine operation could skew our vectors, so to be safe, we use only the inner part of the matrices to put our normal in camera coordinates (or set the 4th value to 0).

Transformation matrix

http://csclab.murraystate.edu/~bob.pilgrim/515/redbook.pdf

Appendix F. Calculating Normal Vectors

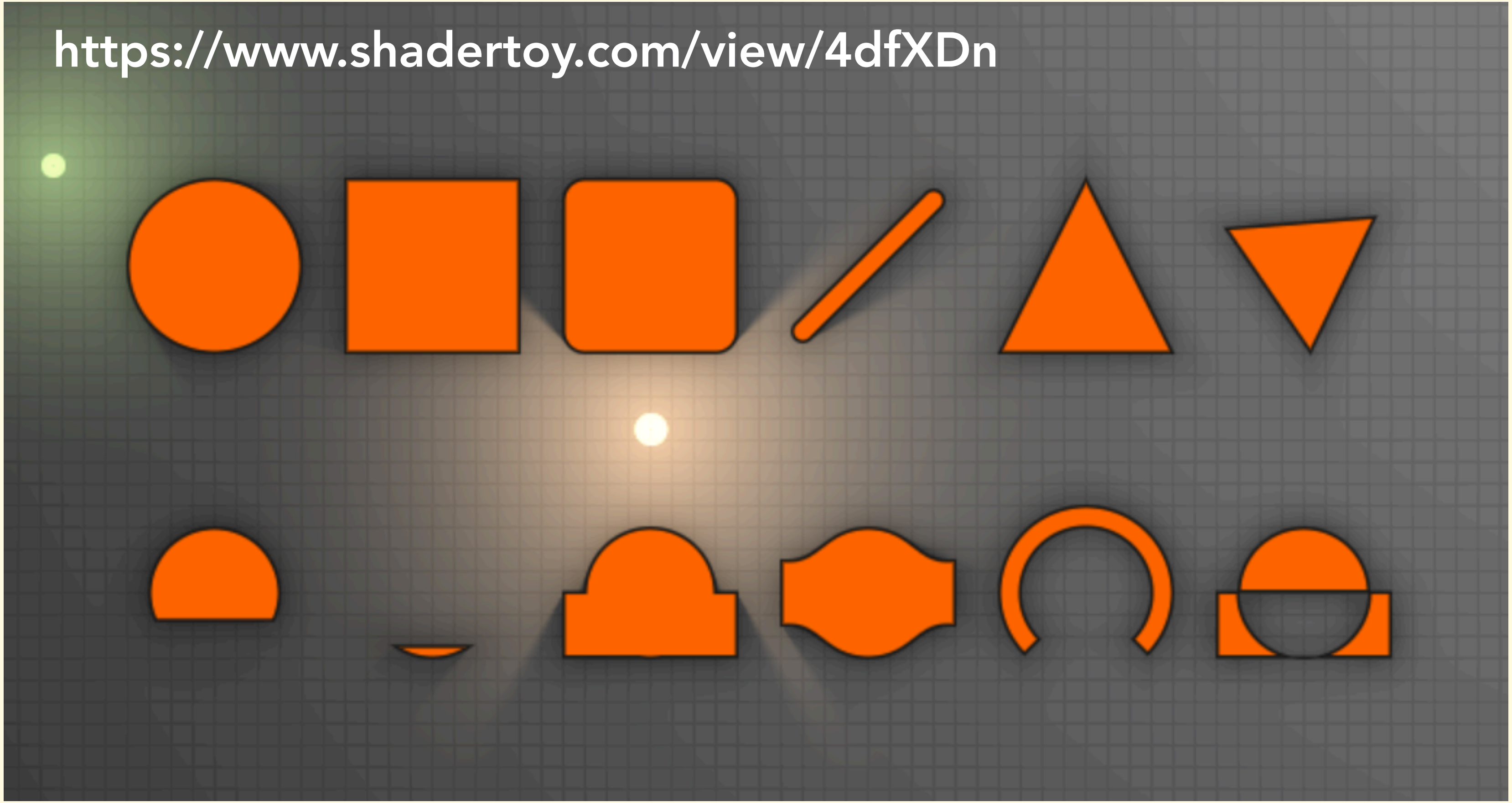
Appendix G. Homogeneous Coordinates and Transformation Matrices

https://en.wikipedia.org/wiki/Transformation_matrix

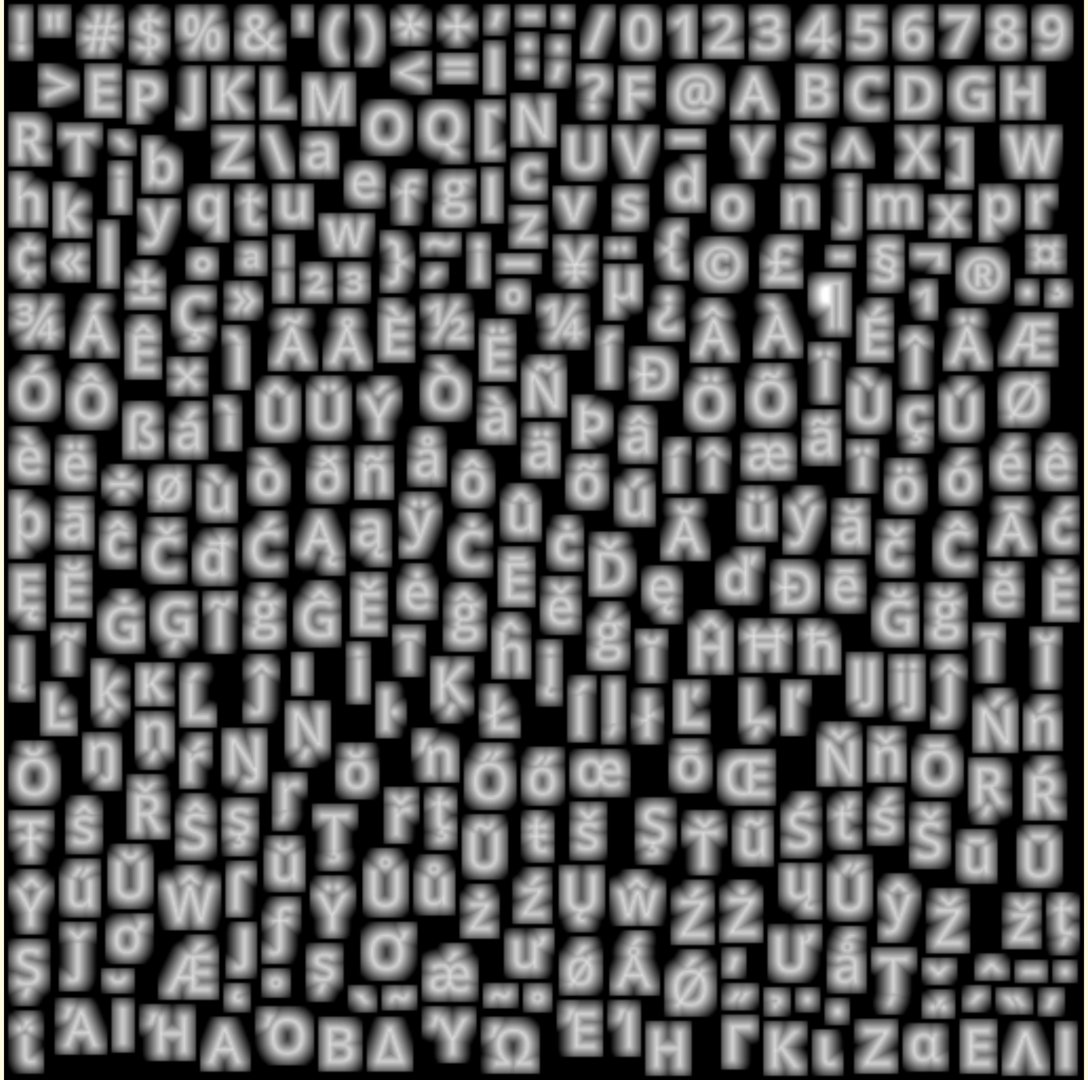
https://en.wikipedia.org/wiki/Affine_transformation

https://en.wikipedia.org/wiki/Homogeneous_coordinates

2D Signed distance functions (SDF)



SDF atlas



SDF for rendering text

Overview:

https://blog.mapbox.com/drawing-text-with-signed-distance-fields-in-mapbox-gl-b0933af6f817

Demo:

https://mapbox.s3.amazonaws.com/kkaefer/sdf/index.html

Source:

view-source:https://mapbox.s3.amazonaws.com/kkaefer/sdf/index.html

Raymarching 3D SDFs

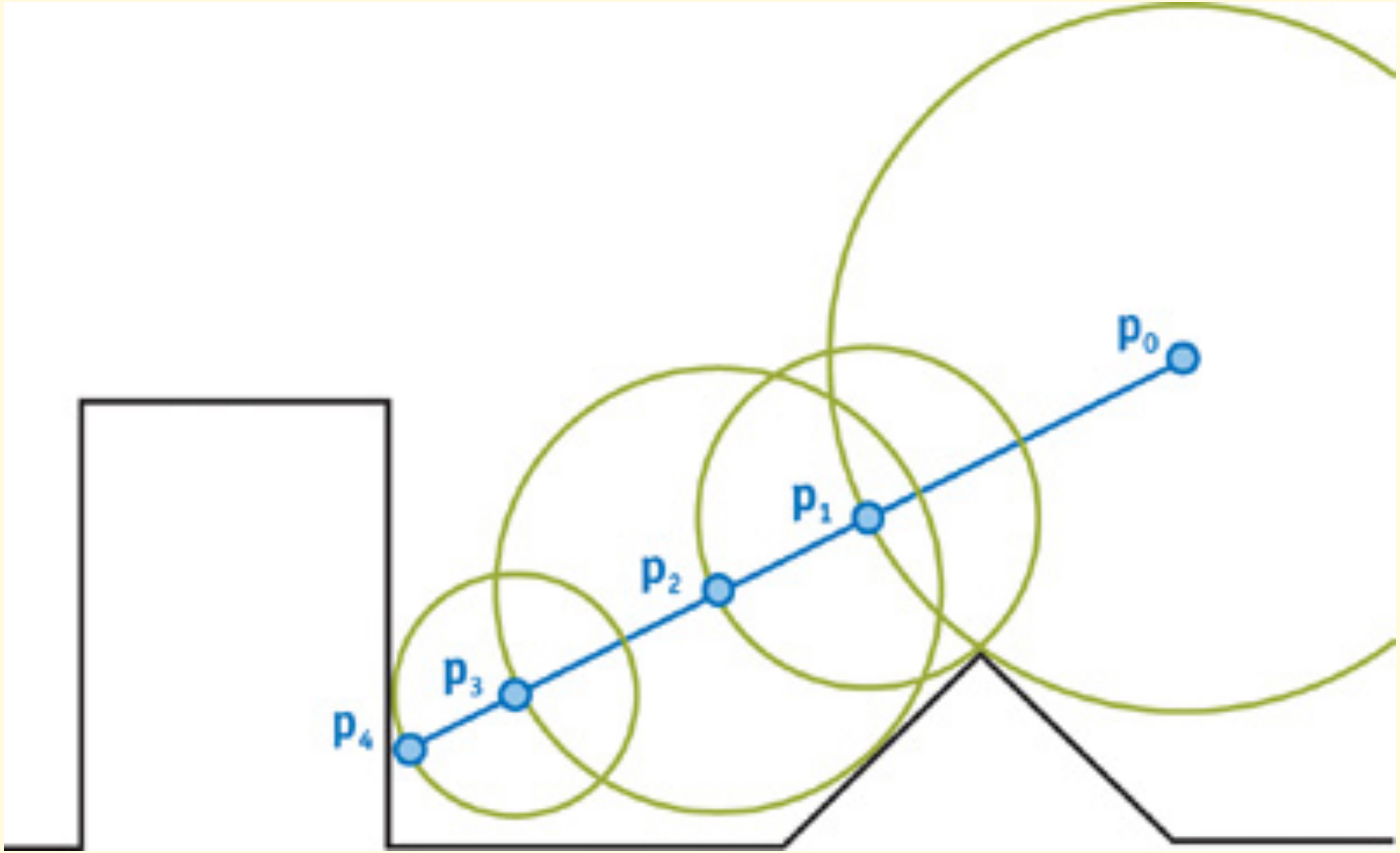
Using 3D SDFs is quite similar to 2D SDFs, except that we need to "march" through the scene to see where the ray emitted from the camera intersects with an objects defined by the SDF.

We want to find the precise point where the ray hits the boundary of the object.

- Points inside object have a negative distance
- Points outside have a positive distance
- Points on the boundary are zero

Sphere tracing with 3 Objects (pyramid, cube, plane)

https://developer.nvidia.com/gpugems/GPUGems2/gpugems2_chapter08.html



Raymarching 3D SDFs

```
vec3 rayDirection(float fieldOfView, vec2 size, vec2 fragCoord) {
    vec2 xy = fragCoord - size / 2.0;
    float z = size.y / tan(radians(fieldOfView) / 2.0);
    return normalize(vec3(xy, -z));
}
```

- very similar to how the view frustum is calculated and encoding in the projection matrix

Raymarching 3D SDFs

```
float shortestDistanceToSurface(vec3 eye, vec3 marchingDirection, float start, float end) {
    float depth = start;  //i.e. start = the position of the near plane, usually a bit in front
of the camera
    for (int i = 0; i < MAX_MARCHING_STEPS; i++) {
        float dist = sceneSDF(eye + depth * marchingDirection);
        if (dist < EPSILON) { //unlikely that value will probably be exactly zero
            return depth; //We have hit the boundary of a 3D SDF
        }
        depth += dist;
        if (depth >= end) {
            return end; //We have traversed the entire scene without ever hitting anything
        }
    }
    return end; //Should never get here, but want to limit number of steps
}
```

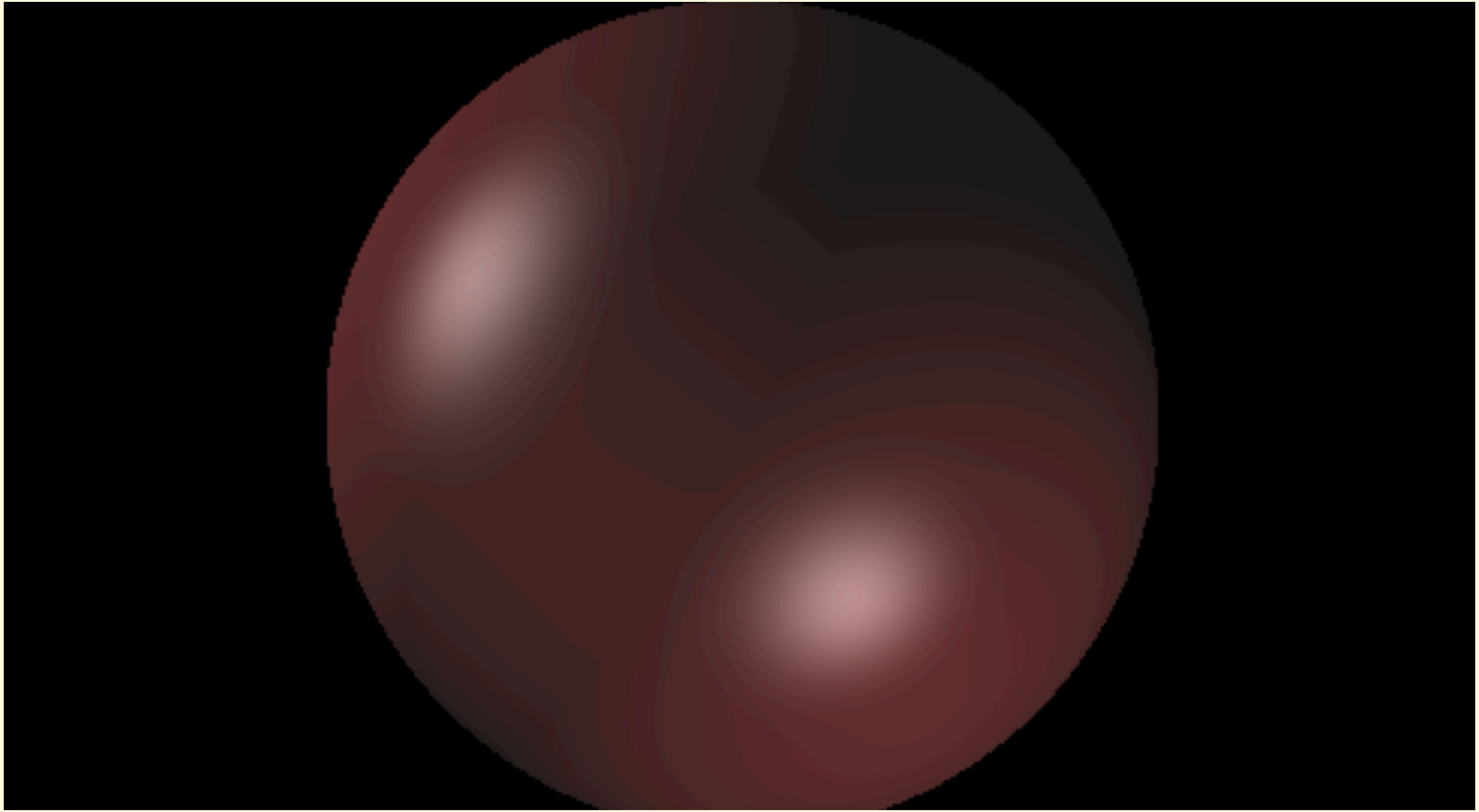
Raymarching 3D SDFs

Find the gradient for the intersecting point by querying the distance of nearby points, some epsilon (E) distance away. Normalize to unit length, and can use as an approximation with which to calculating lighting.

```
vec3 estimateNormal(vec3 p) {
    return normalize(
        vec3(
            sceneSDF(vec3(p.x + E, p.y, p.z)) - sceneSDF(vec3(p.x - E, p.y, p.z)),
            sceneSDF(vec3(p.x, p.y + E, p.z)) - sceneSDF(vec3(p.x, p.y - E, p.z)),
            sceneSDF(vec3(p.x, p.y, p.z + E)) - sceneSDF(vec3(p.x, p.y, p.z - E))
        )
    );
}
```

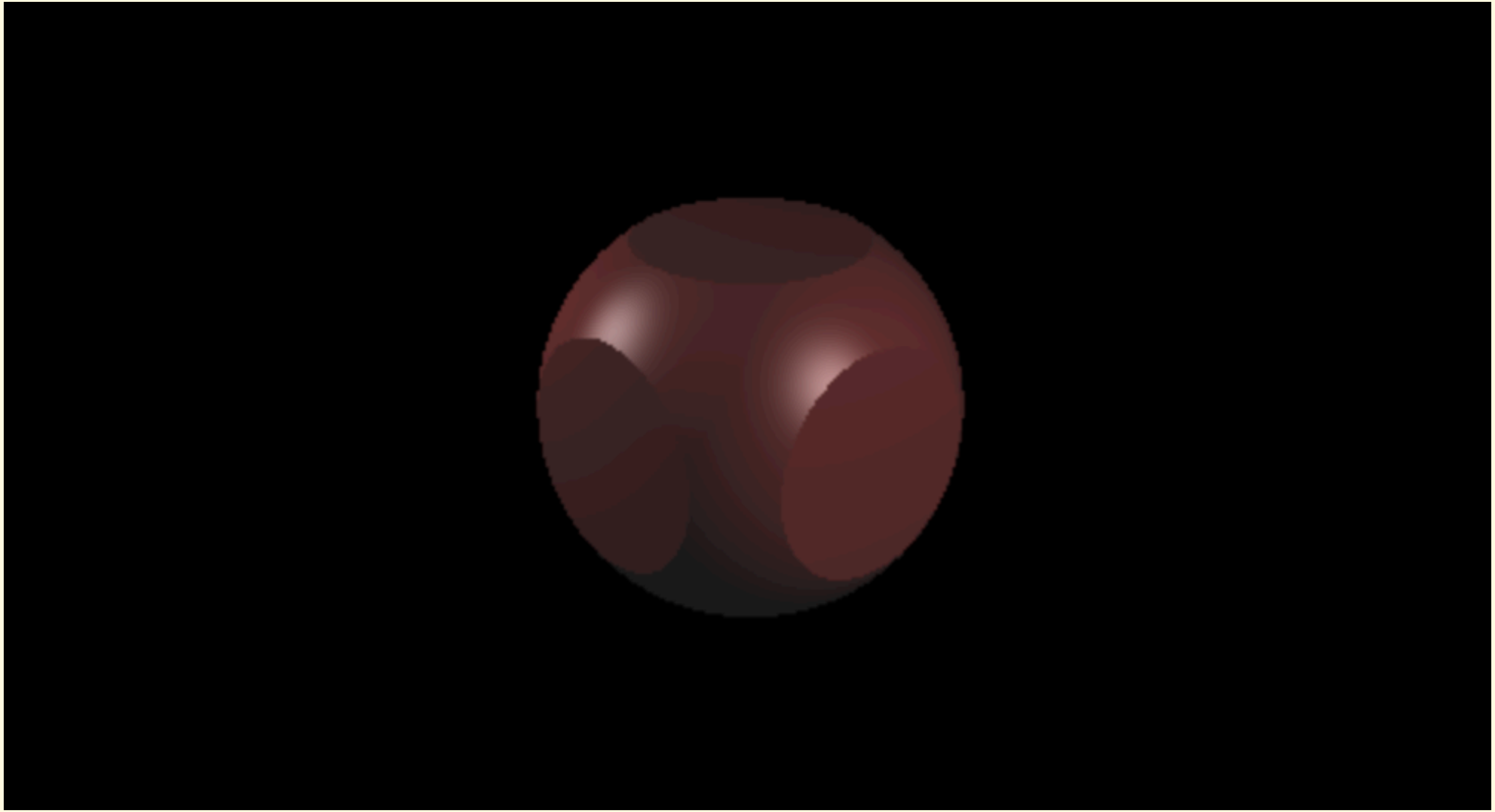
Raymarching 3D SDFs

https://www.shadertoy.com/view/lt33z7



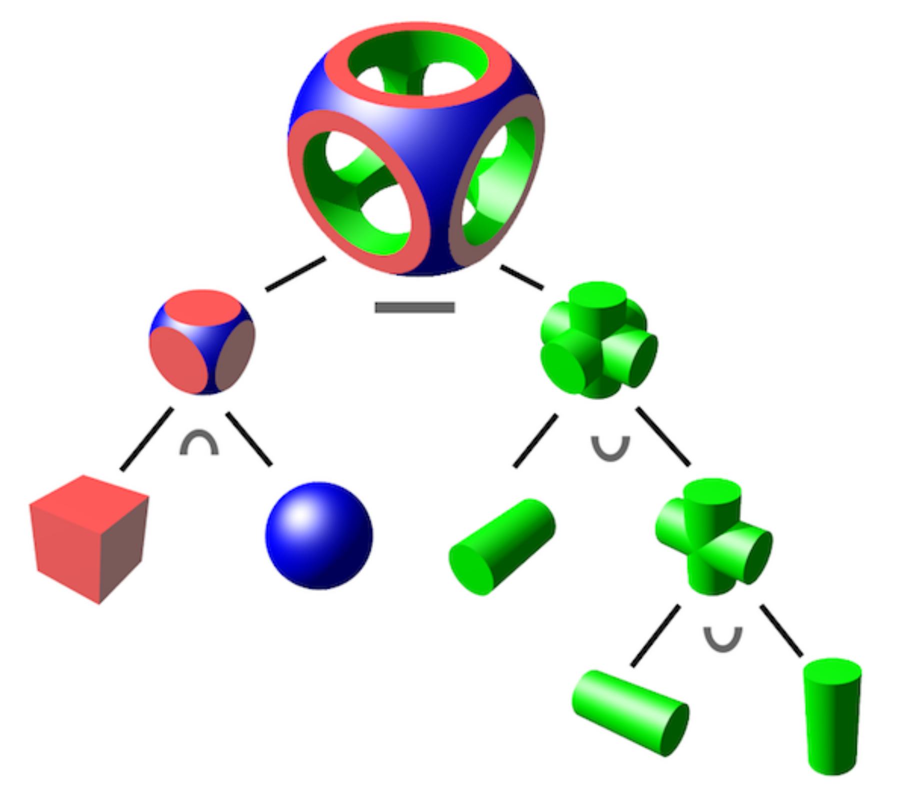
Raymarching 3D SDFs

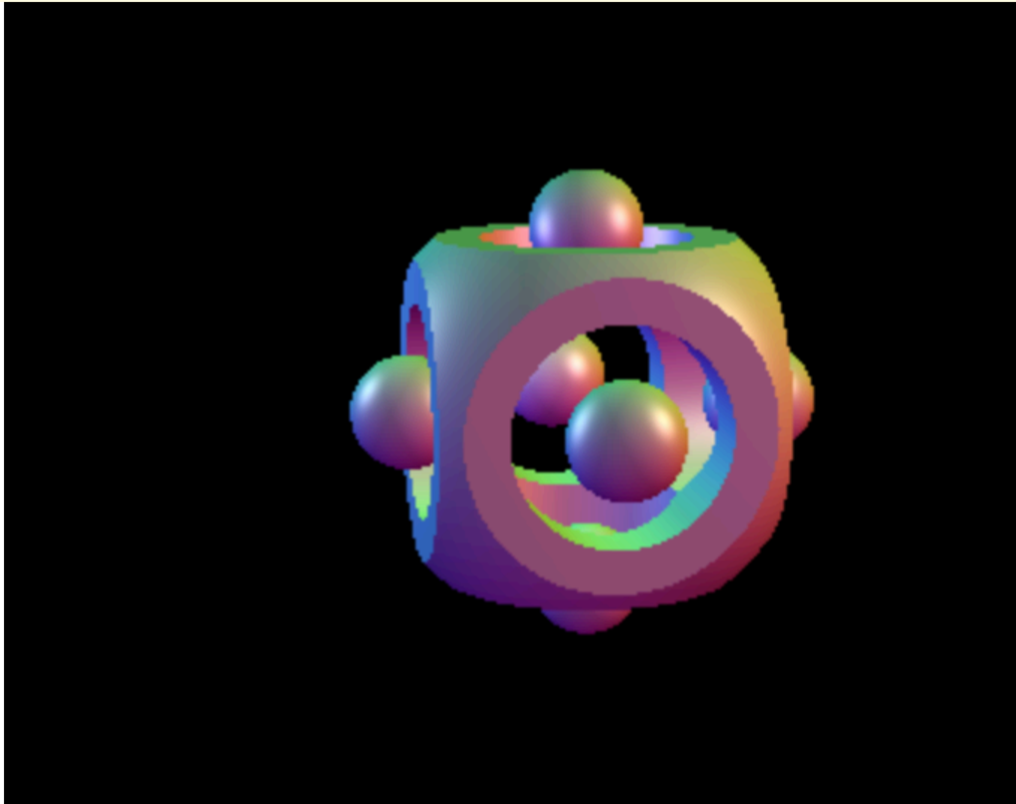
https://www.shadertoy.com/view/MttGz7



Constructive solid geometry, or CSG

- https://www.shadertoy.com/results?query=Ray+Marching
- http://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/





3D Signed distance functions (SDF)

In class exercise

Modify the ShaderToy code:

- change color + position + timing of lights
- add additional lights
- add two spheres to the scene
- add noise to the shapes???

- play around with CSG

Next week

- Casting shadows!
- Various shader effects

- Homework #3 will be assigned on Tuesday (will focus on raymarching, porting examples to Three.js)

- Start thinking about final project

Game Graphics & Real-time Rendering

CMPM 163, W2018

Prof. Angus Forbes (instructor)
angus@ucsc.edu

Lucas Ferreira (TA)
lferreira@ucsc.edu

creativecoding.soe.ucsc.edu/courses/cmpm163
github.com/CreativeCodingLab