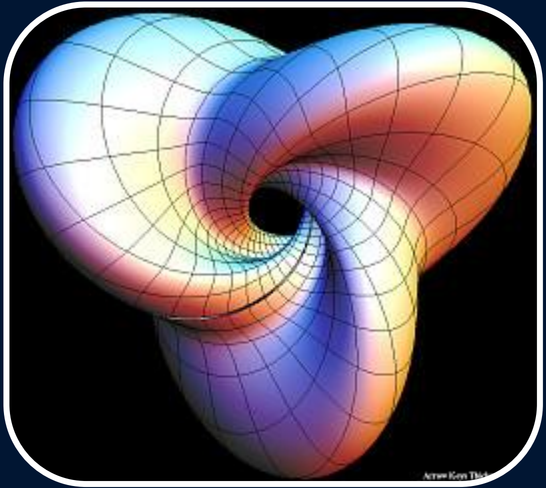
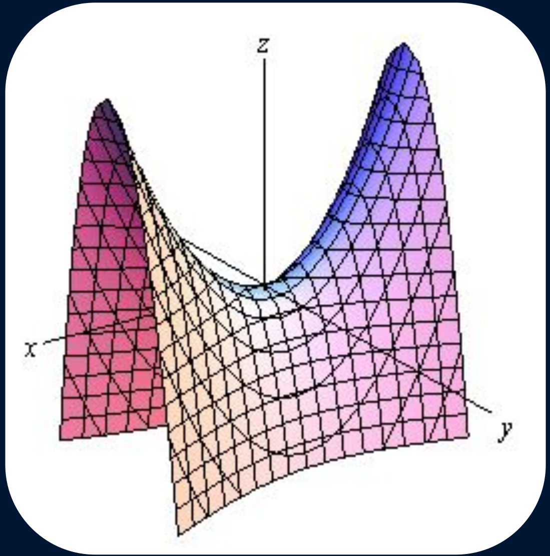
RENDERING ISOSURFACES

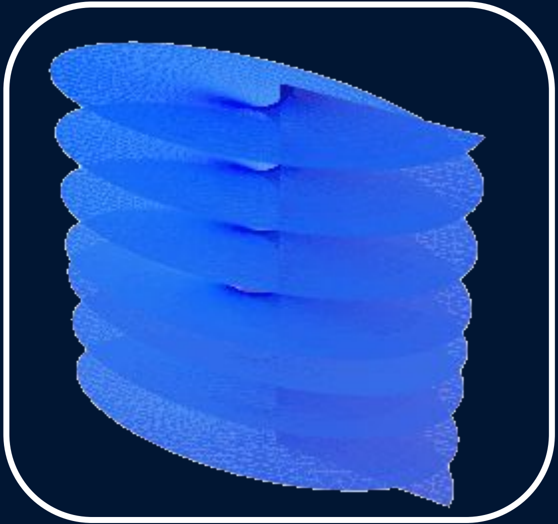
JOHN-PAUL POWERS & DMITRIY BOGUSH

WHAT IS AN ISOSURFACE?

In the simplest terms, an isosurface is a 3D... surface...







WHAT IS AN ISOSURFACE?

Some function of the form:

$$f : \mathbb{R}^3 \Rightarrow D$$

All points that give the same value form an **isosurface**.

$$d \in D, S_d = \{p : f(p) = d\}$$

For example, an SDF equal to zero.

WHY RENDER THEM?

Because they're:

Cool:

- Art
- Entertainment

Important:

- Medical Scans
- Visualizing Math

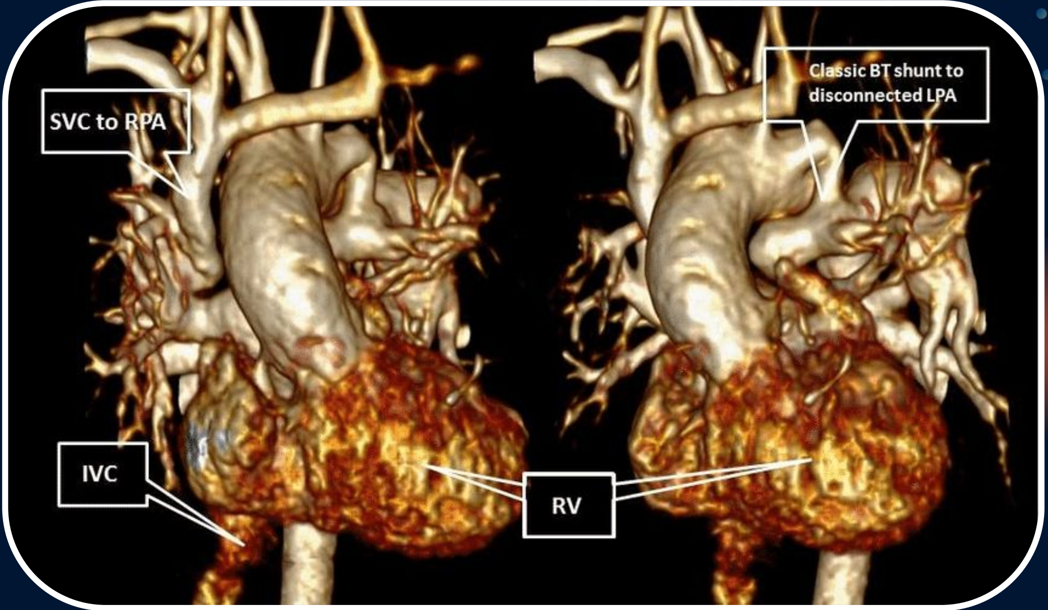
EXAMPLES

LINK

LINK

Credit: System Era, Astroneer (2019)

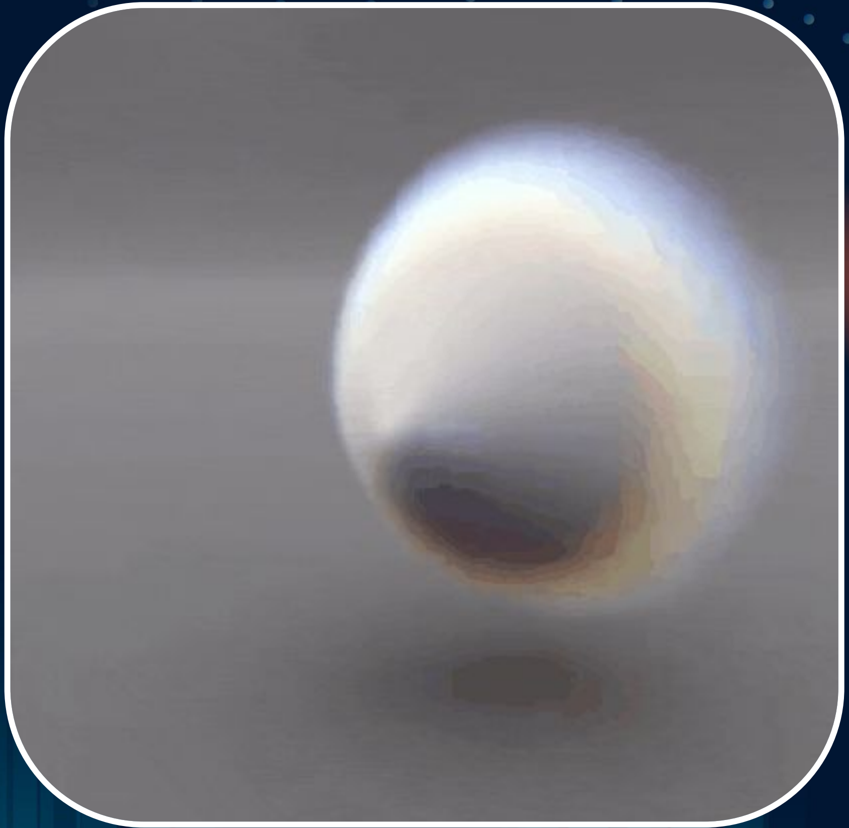
EXAMPLES



Credit: Adam Graham Stuart

EXAMPLES

Credit: *MANDELWERK* LINK

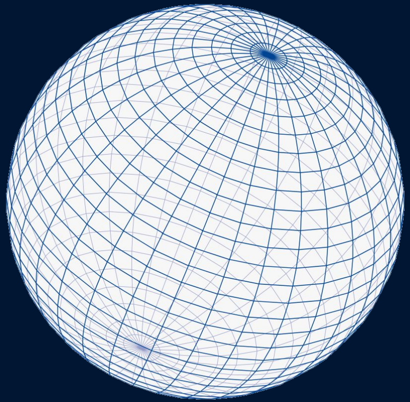


LINK

Okay, but why not raytrace?

RAYTRACING ISOSURFACES

Possible:



$$F(x, y, z) = 1 - x^2 + y^2 + z^2$$

To raytrace an isosurface, you need a **unique intersection equation**

Depending on the isosurface, this can be prohibitively hard and is not generalized.

RENDERING ISOSURFACES

Two main ways:

Ray-Marching

Polygonization

Each has pros and cons; we'll address them separately

RAY-MARCHING

SIGNED DISTANCE FUNCTION?

RAYMARCHING: MAIN IDEA

As our isosurface (which will likely be an SDF) comes from a function, this function can be used to evaluate whether a point is within the surface:

- Inside the surface (negative)
- Outside of the surface (positive)

So, generate points along the ray, then test whether or not that point is inside.

RAYMARCHING: NAIVE

Procedure:

- Increase t by a fixed amount every time.
- Once the point is inside the surface, average it with the last point, and return.

Seems easy enough - let’s try it!

RAYMARCHING: NAIVE

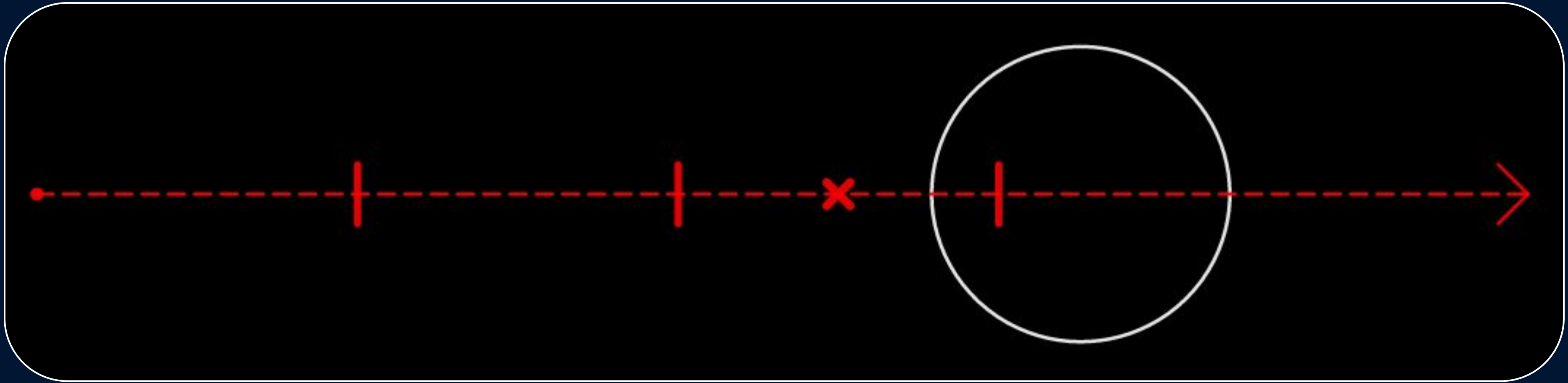
```
function march_ray(p, d, f, step, max_iterations)
    t = 0
    last_point = p
    for i = 1:max_iterations
        t += step
        point = p+t*d
        if f(point) < 0
            return (point + last_point)/2

    return nothing
```

RAYMARCHING: NAIVE

Problems:

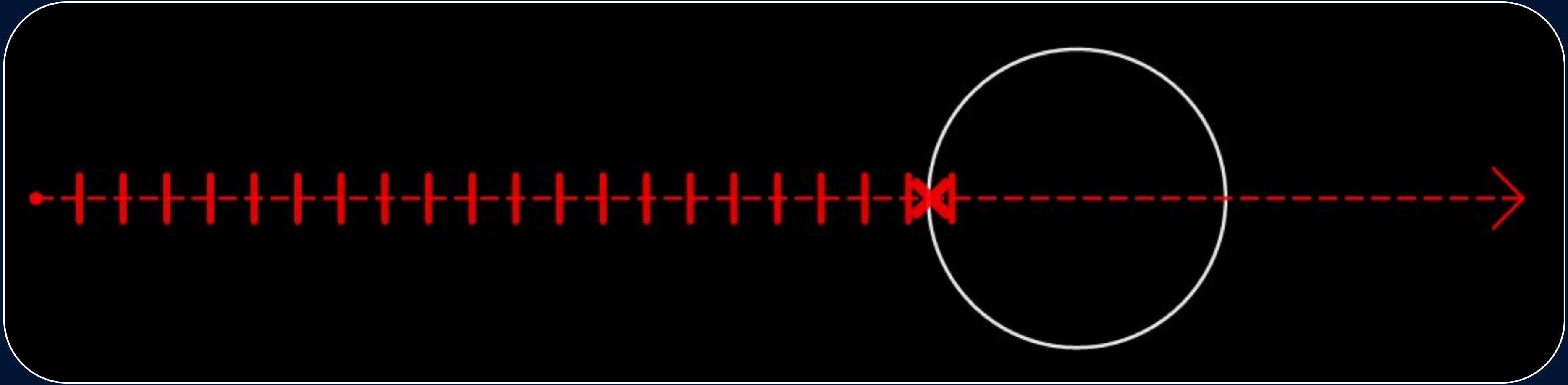
- With a large step size, the final estimate will be off.



RAYMARCHING: NAIVE

Problems:

- With a small step size, the algorithm becomes too expensive:



Solution? Intelligently select step distance.

RAYMARCHING: DISTANCE ESTIMATION

Remember what we are trying to render: a signed **distance** function.

By definition, the value of an SDF at a point is the shortest distance from the surface to said point.

So, when at a given point while ray-marching, we can use the SDF's value at the current point to calculate the **largest safe step size**.

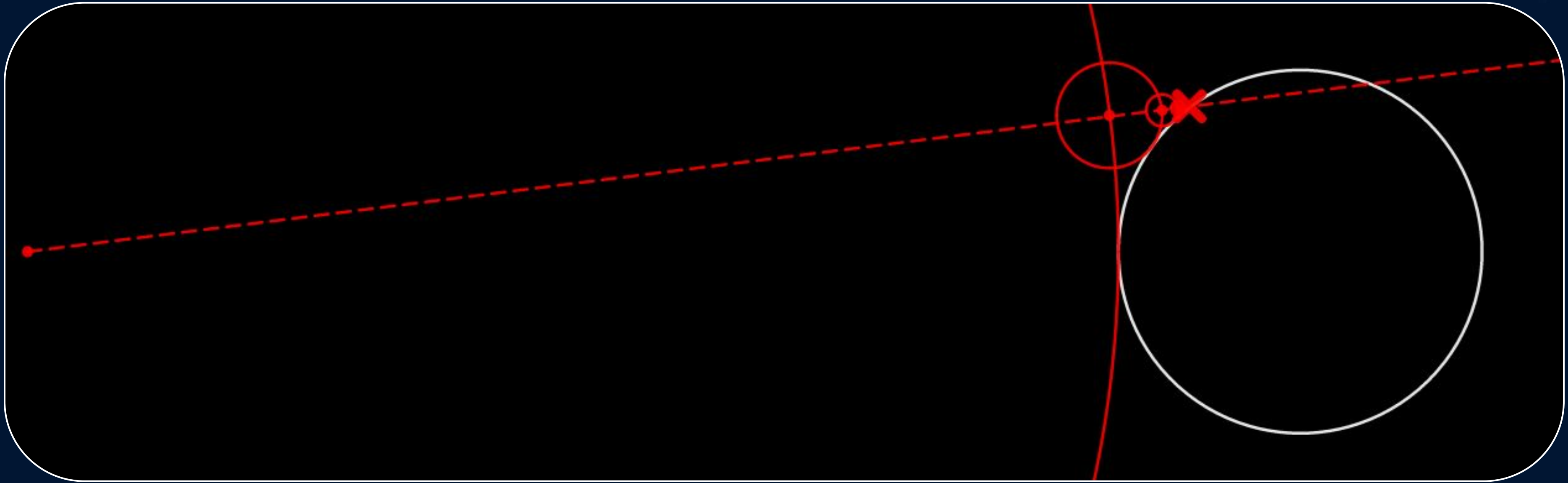
Once the step size drops below a certain threshold, we can then assume a **collision** with the surface.

RAYMARCHING: DISTANCE ESTIMATION

```
march_ray_de(p, d, f, max_iterations)
    t = 0
    point = p
    for i = 1:max_iterations
        step = f(point)
        if(step < 1e-8)
            return point;
        t += step
        point = p+t*d
    return nothing
```

RAYMARCHING: DISTANCE ESTIMATION

Works pretty well!



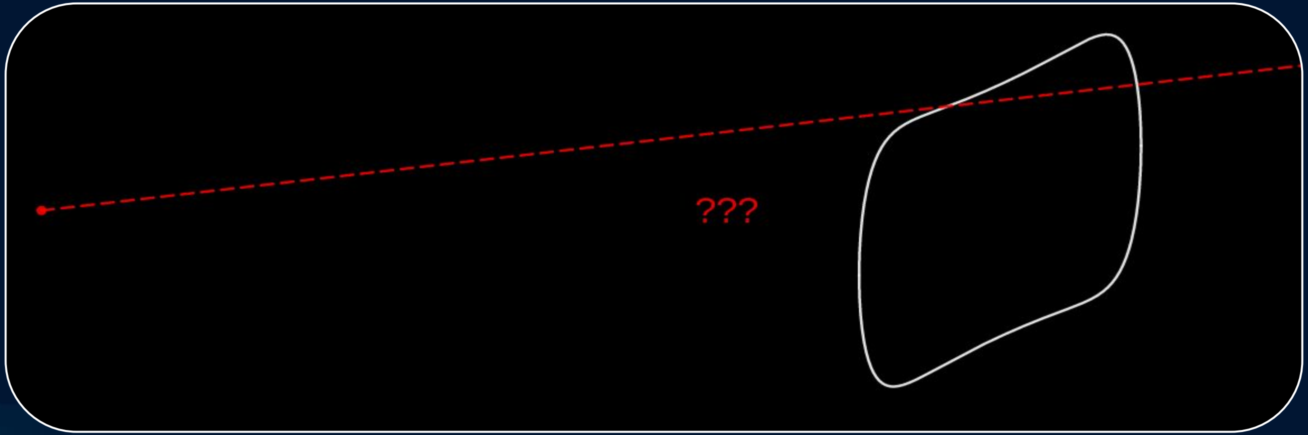
Still one problem however…

RAYMARCHING: DISTANCE ESTIMATION

Problem: We are assuming an SDF.

SDF’s are a **type** of isosurface, but not all isosurfaces are SDF’s.

Take this isosurface: $F(x, y) = x^8 + y^2 - xy - 1$



RAYMARCHING: HYBRID

Instead, we can separate the distance estimator from the isosurface, and get the best of both worlds.

We do this by finding a simple SDF that **bounds** our isosurface, then use that to coarsely estimate the distance.

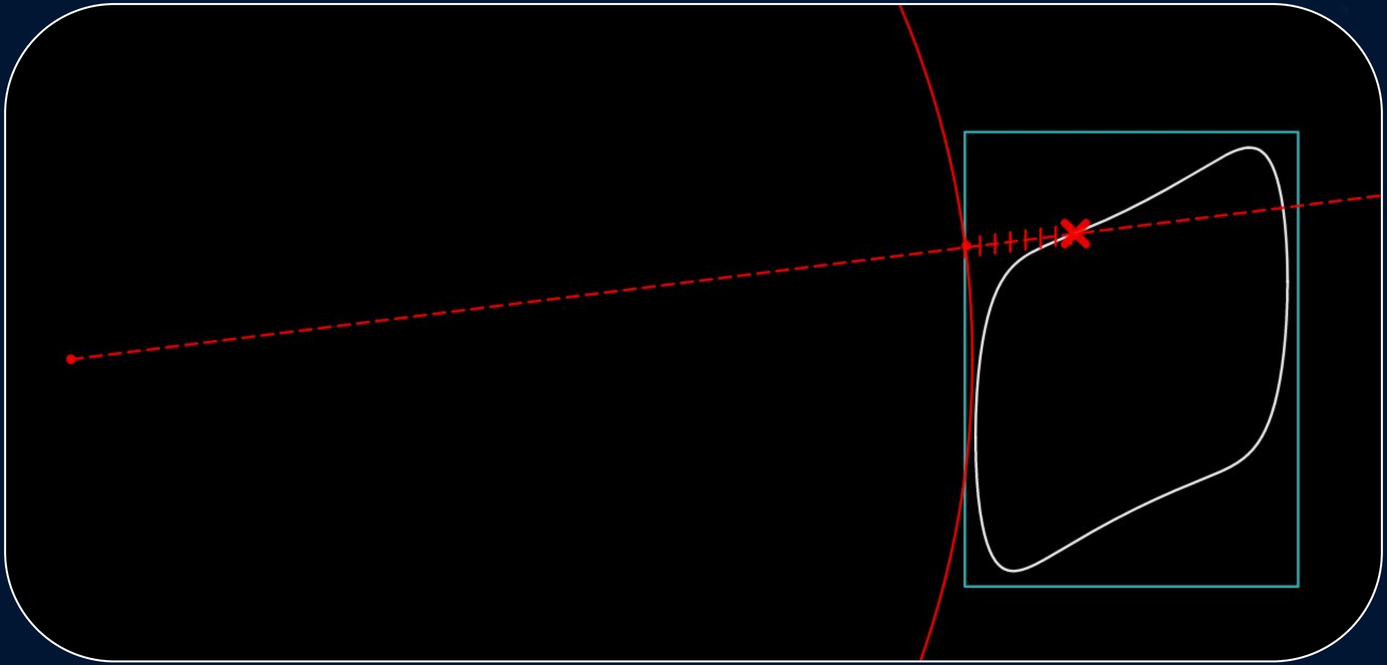
Plan:

- Use the distance estimator to get as close as possible
- Once the distance estimation drops below a certain threshold, naively march by small steps until we find a collision.

RAYMARCHING: HYBRID

```
march_ray_hybrid(p, d, f, de, max_iterations, small_step)
     t = 0
     point = p
     for i = 1:max_iterations
          step = de(point)
          if(step < small_step)
               last_point = p
               for j = 1:(max_iterations - i)
                    t += small_step
                    point = p+t*d
                    if f(point) < 0
                         return (point + last_point)/2
               break;
          t += step
          point = p+t*d
     return nothing
```

RAYMARCHING: HYBRID



RAYMARCHING

Pros:

- Can render any SDF perfectly, no matter how intricate.
- Easy to parallelize (each SDF is a primitive)
- Has solutions for non-SDF isosurfaces

Cons:

- Cannot render discrete data (e.g. Medical Scans)
- SDF’s are not very portable
- Clashes with traditional render pipelines (rasterizers)

BUT WHAT IF WE WANT TO?

ASTRONEER (LINK)

POLYGANIZATION

POLYGANIZATION: MAIN IDEA

Rendering a triangle is as portable as it is trivial.

So, instead of attempting to render the isosurface directly, we might try to represent it using a triangle mesh (polygonization), after which it can be rendered anywhere!

However, we'll need to do a little processing before our isosurface can be polygonized.

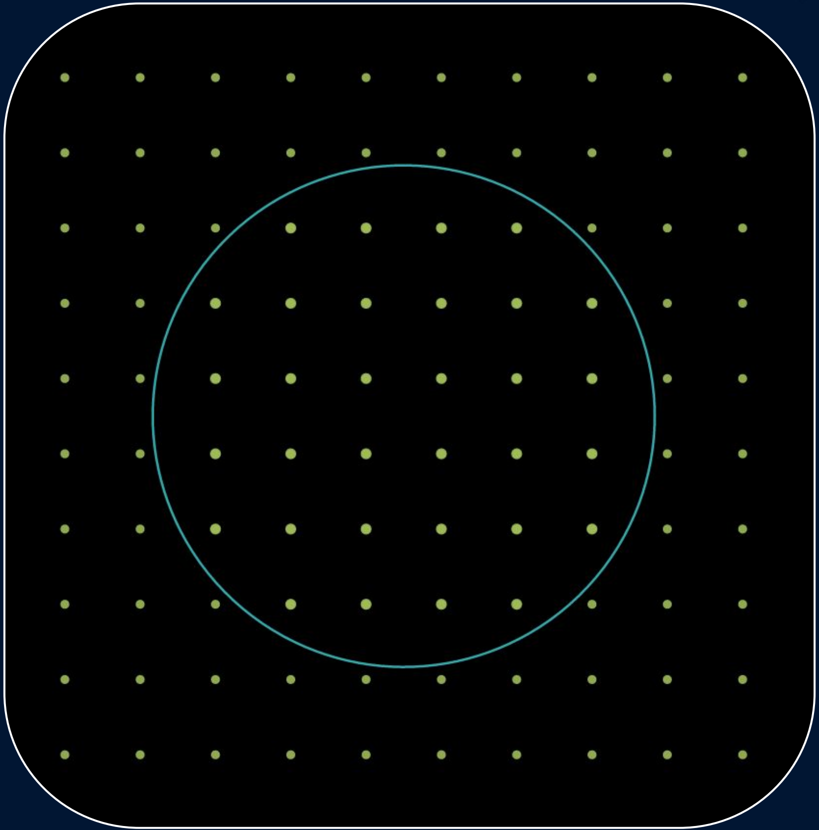
POLYGANIZATION: HERMITE DATA

As a triangle mesh is a discrete data type, we will need to discretize our isosurface.

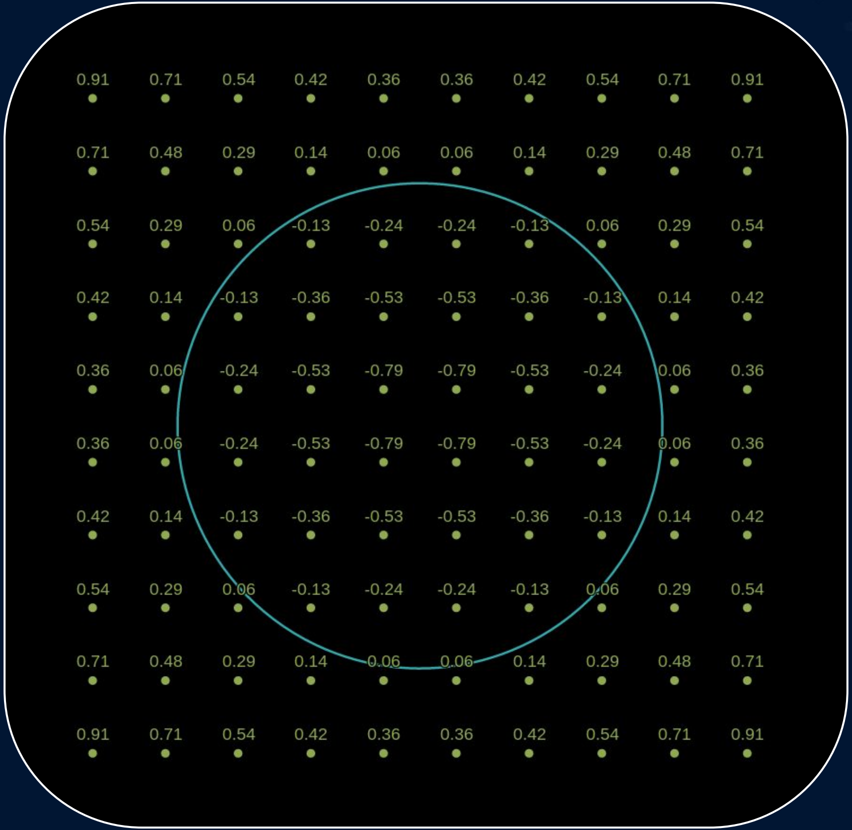
In the case where our isosurface is already discretized (such as some kind of medical scan) this step can more-or-less be skipped.

If it is not, we'll need to **sample** it, specifically on a uniform grid.

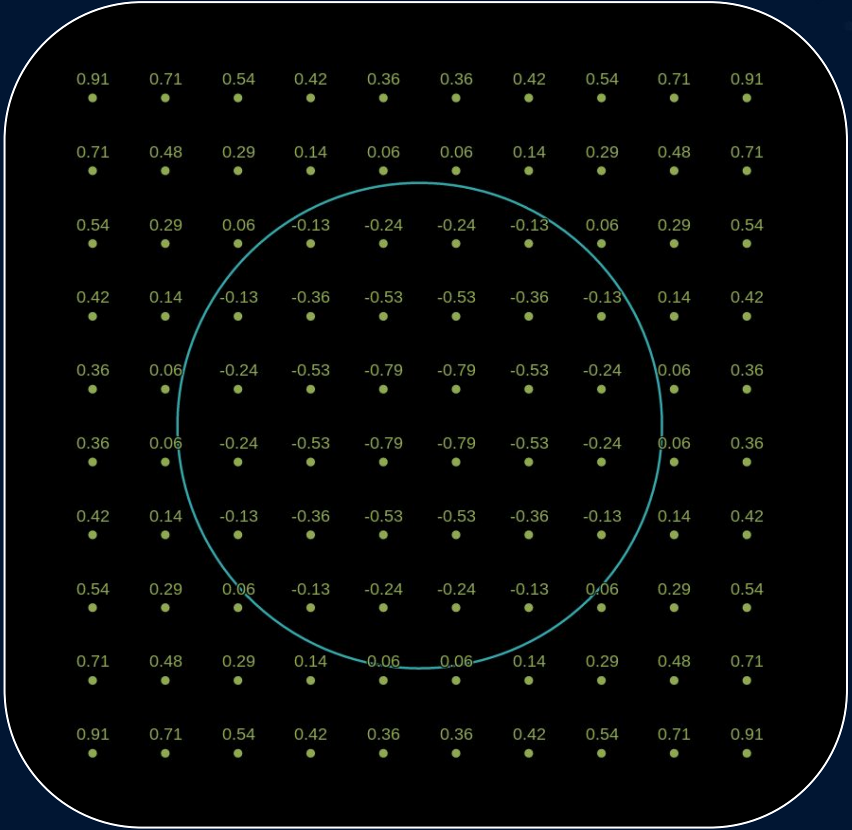
POLYGANIZATION: HERMITE DATA



POLYGANIZATION: HERMITE DATA

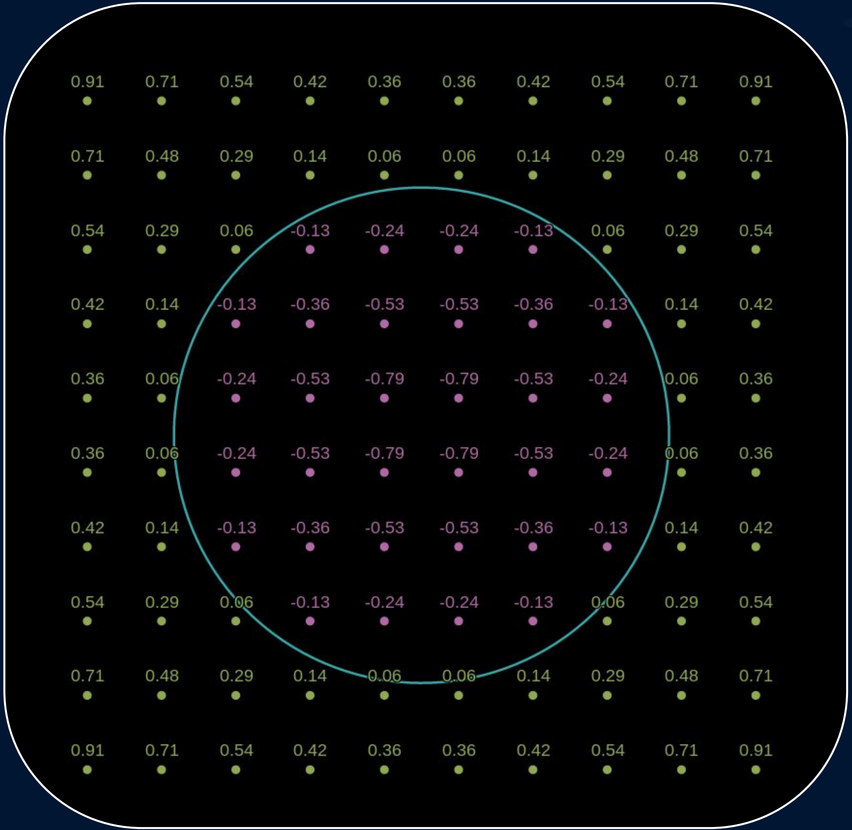


POLYGANIZATION: HERMITE DATA



Q: Anyone see a pattern?

POLYGANIZATION: HERMITE DATA



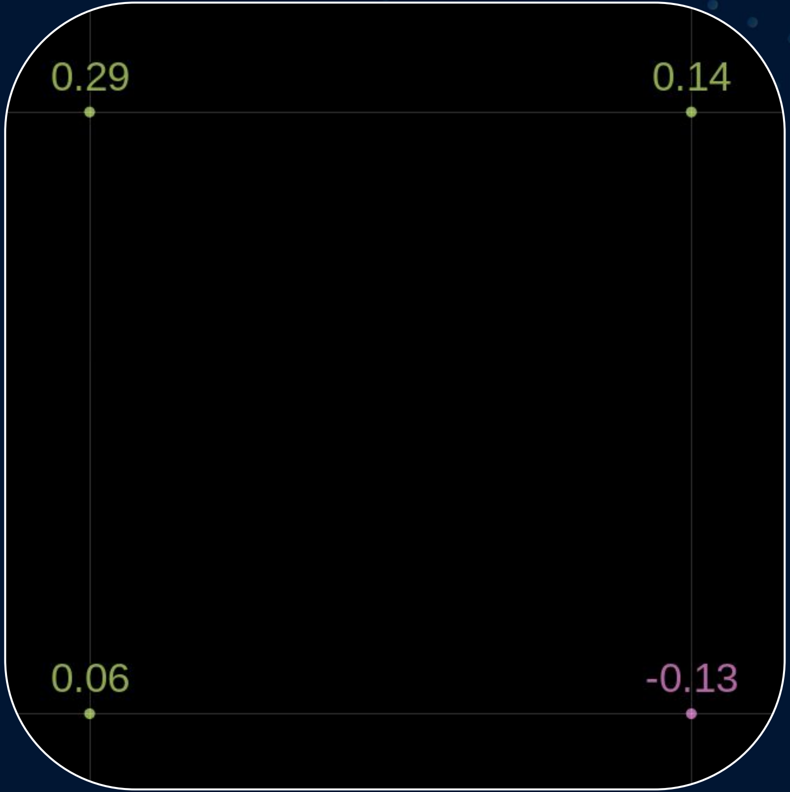
POLYGANIZATION: MARCHING CUBES

The first, and one of the simplest isosurface polygonization algorithms.

Published by William Lorenson and Harvey Cline in 1987.

POLYGANIZATION: MARCHING CUBES

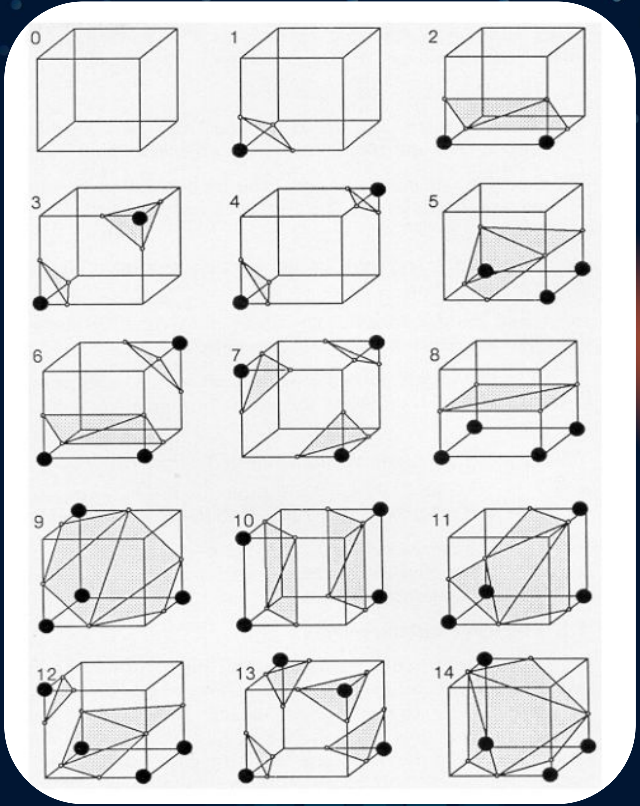
<u>Step 1:</u>
Consider each cell (square in 2D, cube in 3D) and the values at its corners.



POLYGANIZATION: MARCHING CUBES

<u>Step 2:</u>
There are a finite number of ways these corners can be arranged (inside/outside the isosurface), and as such, there is a **unique polygonization for each**.

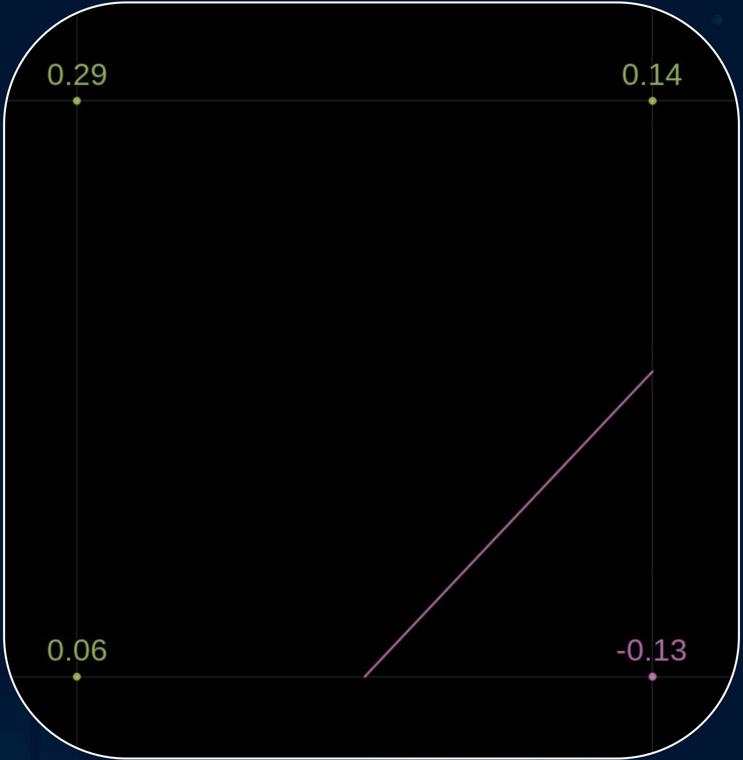
Simply polygonize each cell using these unique cases (often with a triangulation table).



Marching Cubes Cases

POLYGANIZATION: MARCHING CUBES

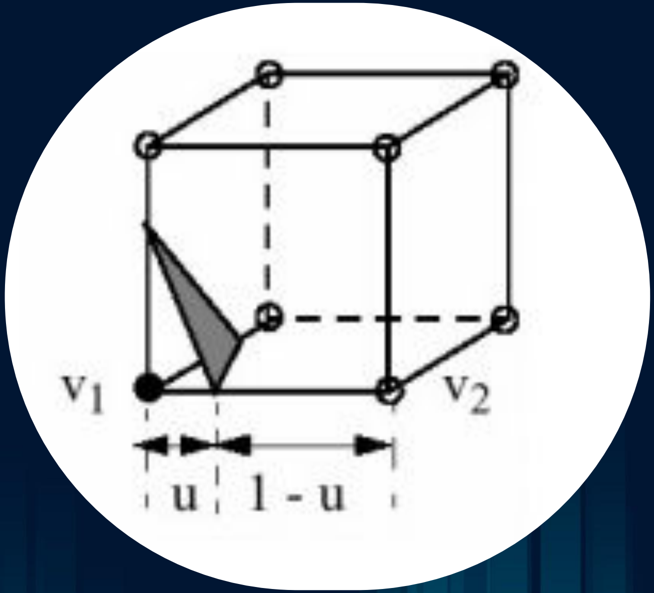
Step 2:



POLYGANIZATION: MARCHING CUBES

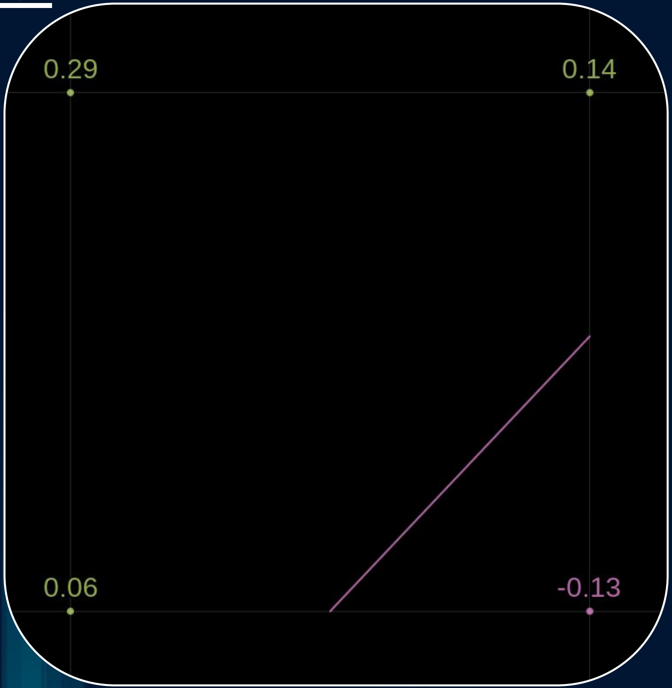
Step 3:

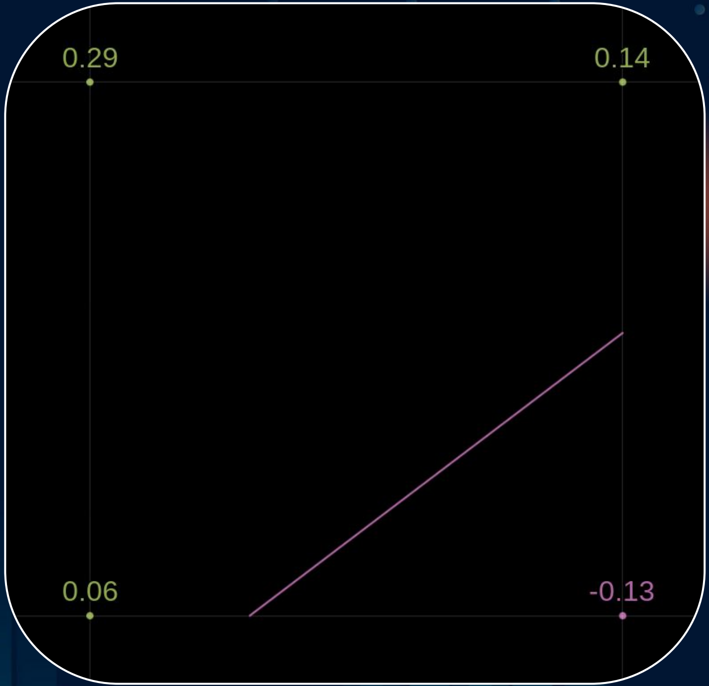
Estimate where to place vertices (within the triangulation case) by linearly interpolating between the hermite data at each end point.



POLYGANIZATION: MARCHING CUBES

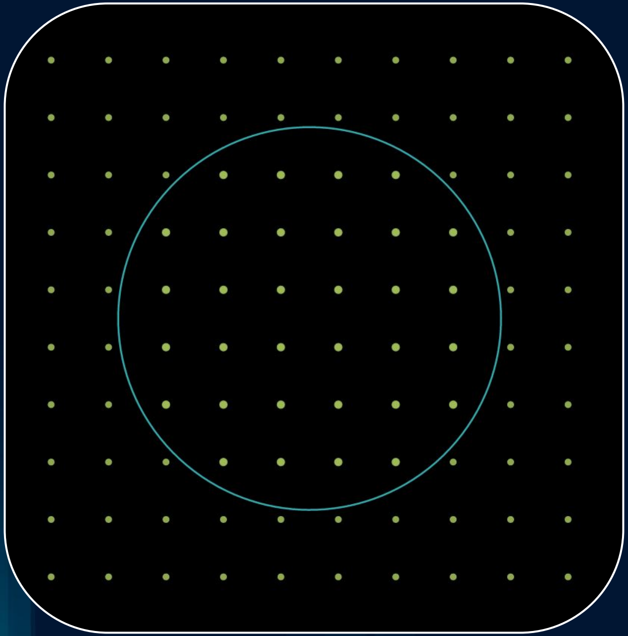
Step 3:

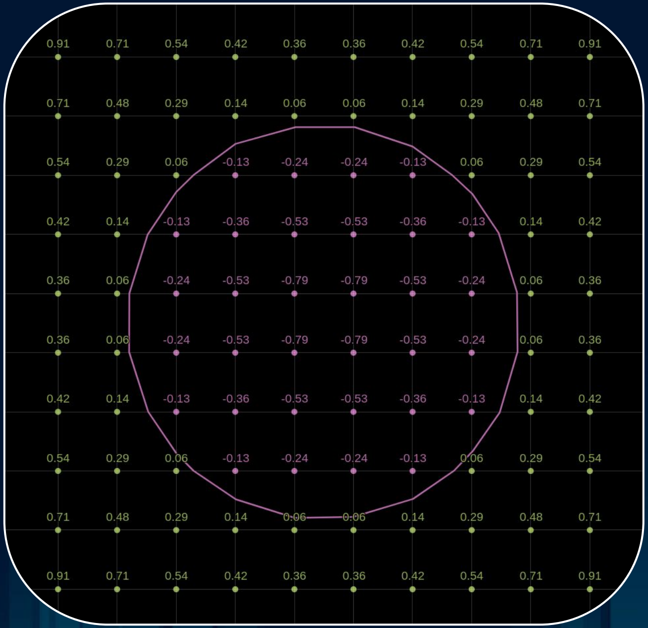




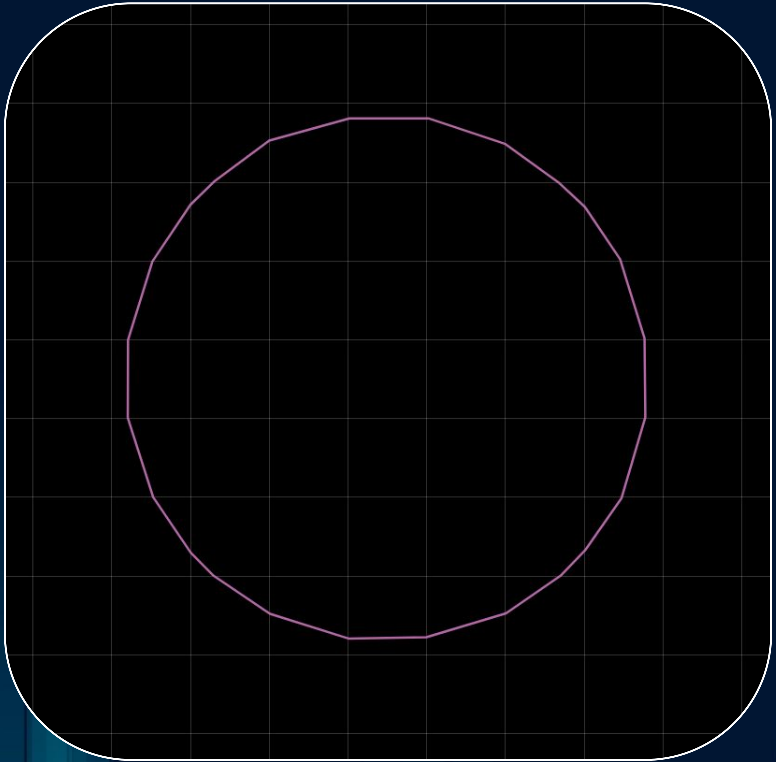
POLYGANIZATION: MARCHING CUBES

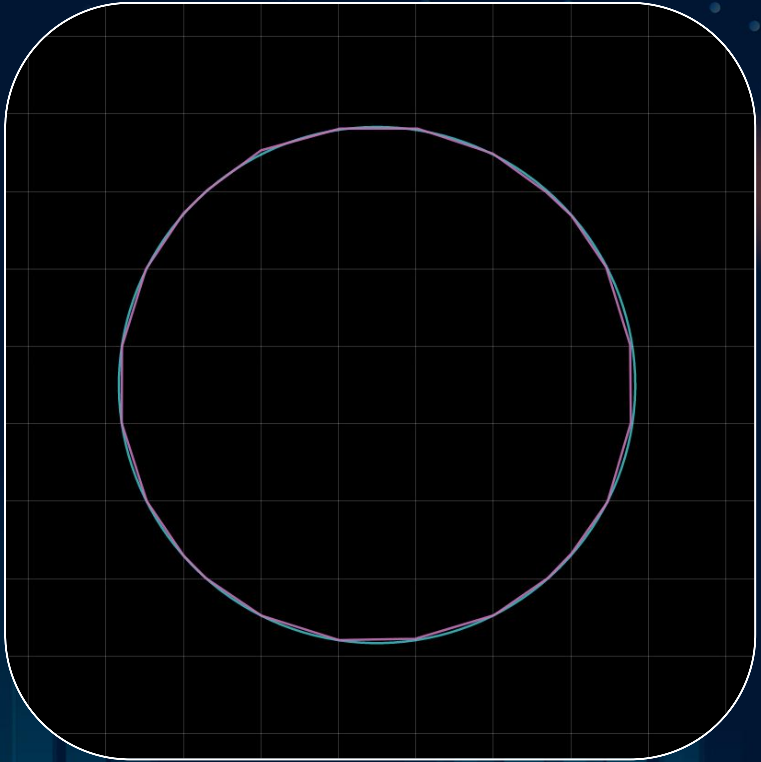
Once the steps have been completed for every cell, the polygonization is complete!



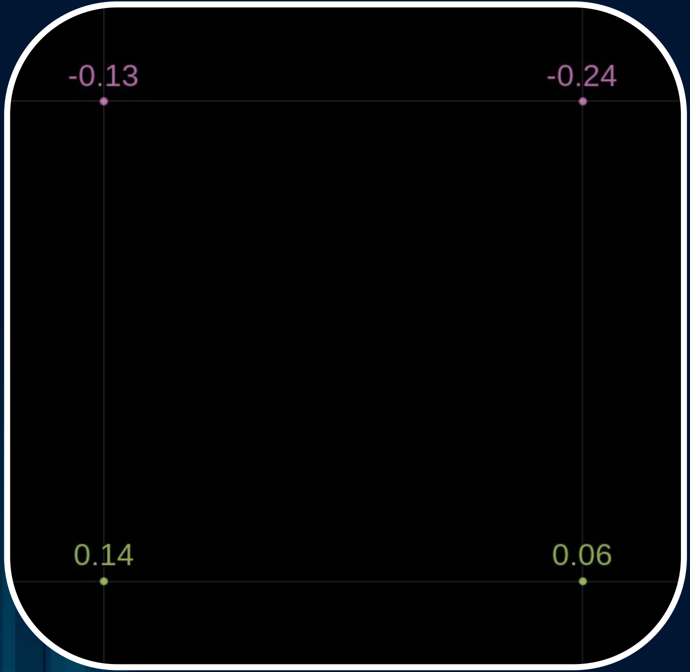


POLYGANIZATION: MARCHING CUBES



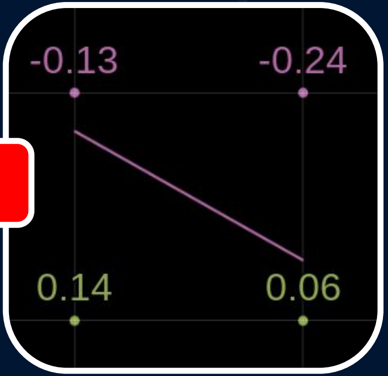


POLYGANIZATION: MARCHING CUBES



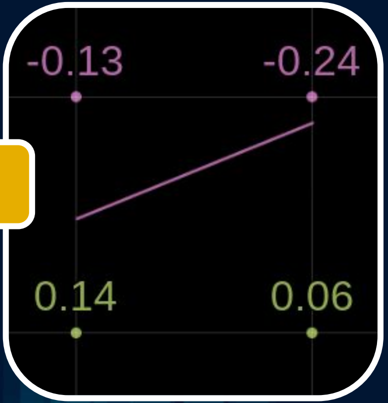
How would marching cubes polygonize this grid cell?

A

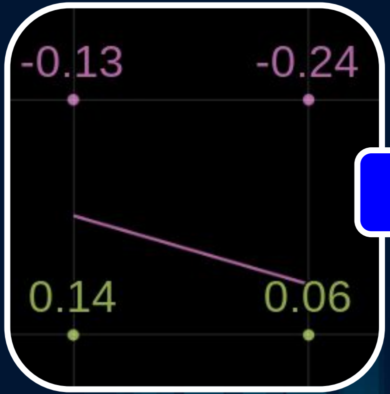


B

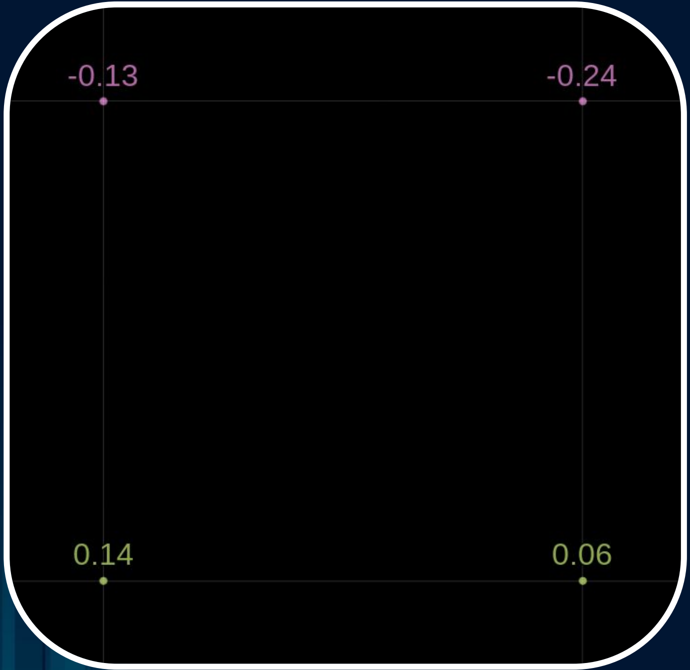
C



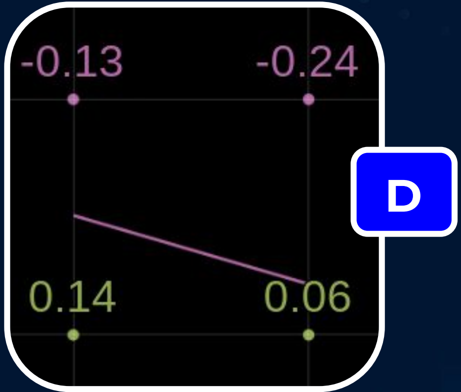
D



POLYGANIZATION: MARCHING CUBES



How would marching cubes polygonize this grid cell?



POLYGANIZATION: MARCHING CUBES

Pros:

- Simple and practical.
- Easy to parallelize (each cube is independent)

Cons:

- Somewhat inaccurate: sharp features are lost.
- Many of the cubes can be empty leading to unnecessary flops.

As always, we can do better!

POLYGONIZATION: DUAL CONTOURING

(Mostly) state-of-the-art algorithm for isosurface polygonization algorithm.

Published by Tao Ju, Frank Losasso, Scott Schaefer, Joe Warren in 2002.

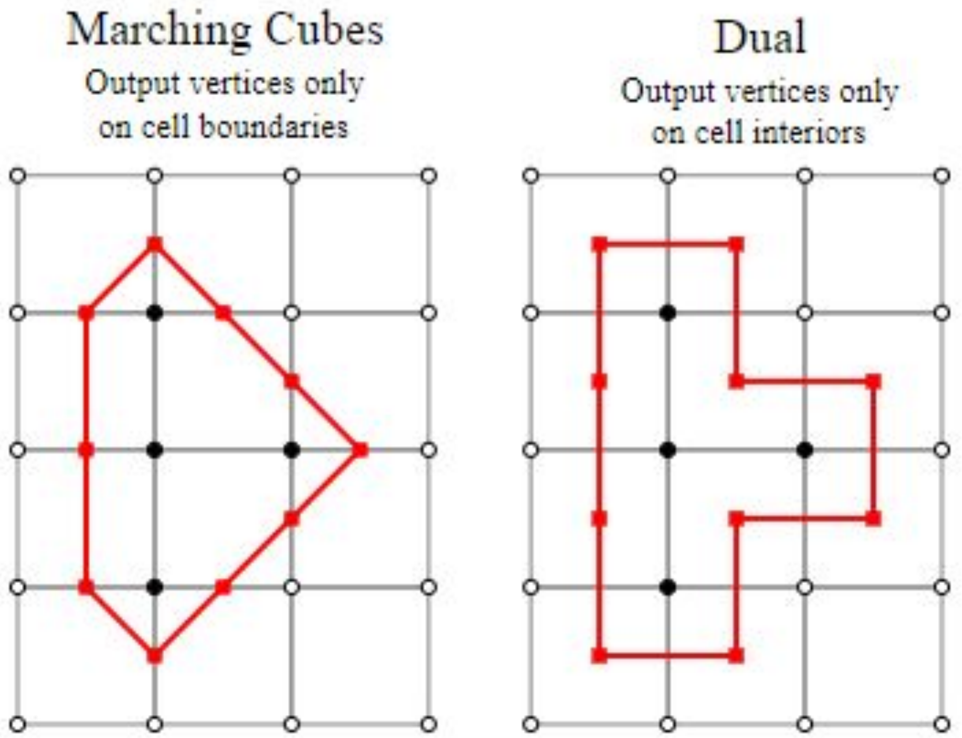
POLYGONIZATION: DUAL CONTOURING

As its name suggests, this is a **dual** algorithm, specifically to marching cubes.

While marching cubes chooses vertices along cell boundaries, dual contouring chooses vertices for the mesh **within the cell**.

As choosing vertices within cells gives **more freedom** as opposed to choosing them along cell boundaries, this leads to better preservation of sharp features, and overall improved mesh quality.

POLYGONIZATION: DUAL CONTOURING



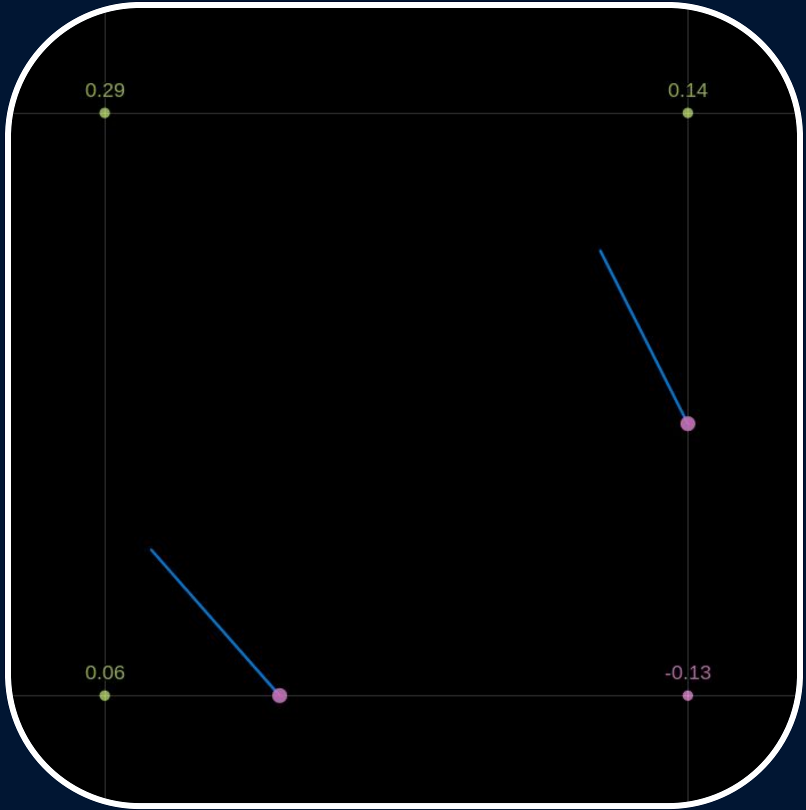
POLYGONIZATION: DUAL CONTOURING

Step 1: Take a cell, and along each edge that exhibits a sign change, calculate the normal.

The normal can easily by using the limit definition of the derivative (gradient) to see how the value of a point within the isosurface changes when it’s components (x, y, z) are perturbed by small values.

POLYGONIZATION: DUAL CONTOURING

Step 1:



POLYGONIZATION: DUAL CONTOURING

Step 2: Find the point within the cell that ‘agrees most’ with the normals.

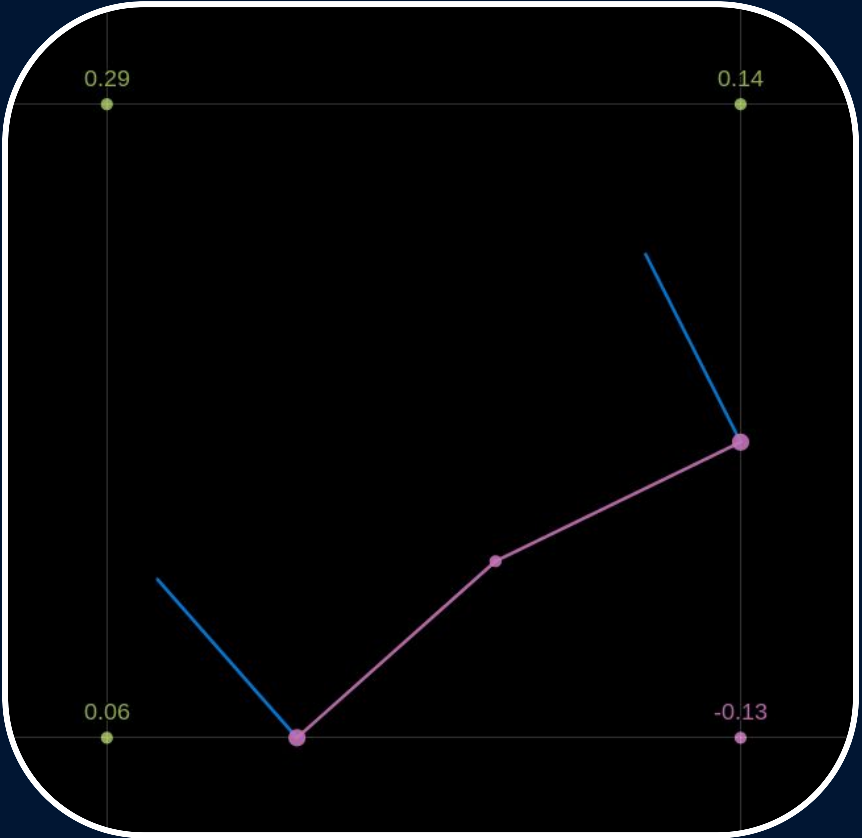
This is usually done by solving the least-squares function:

$$E[x] = \sum_i (n_i \cdot (x - p_i))^2$$

This is quite a bit more complicated than it seems.

POLYGONIZATION: DUAL CONTOURING

Step 2:



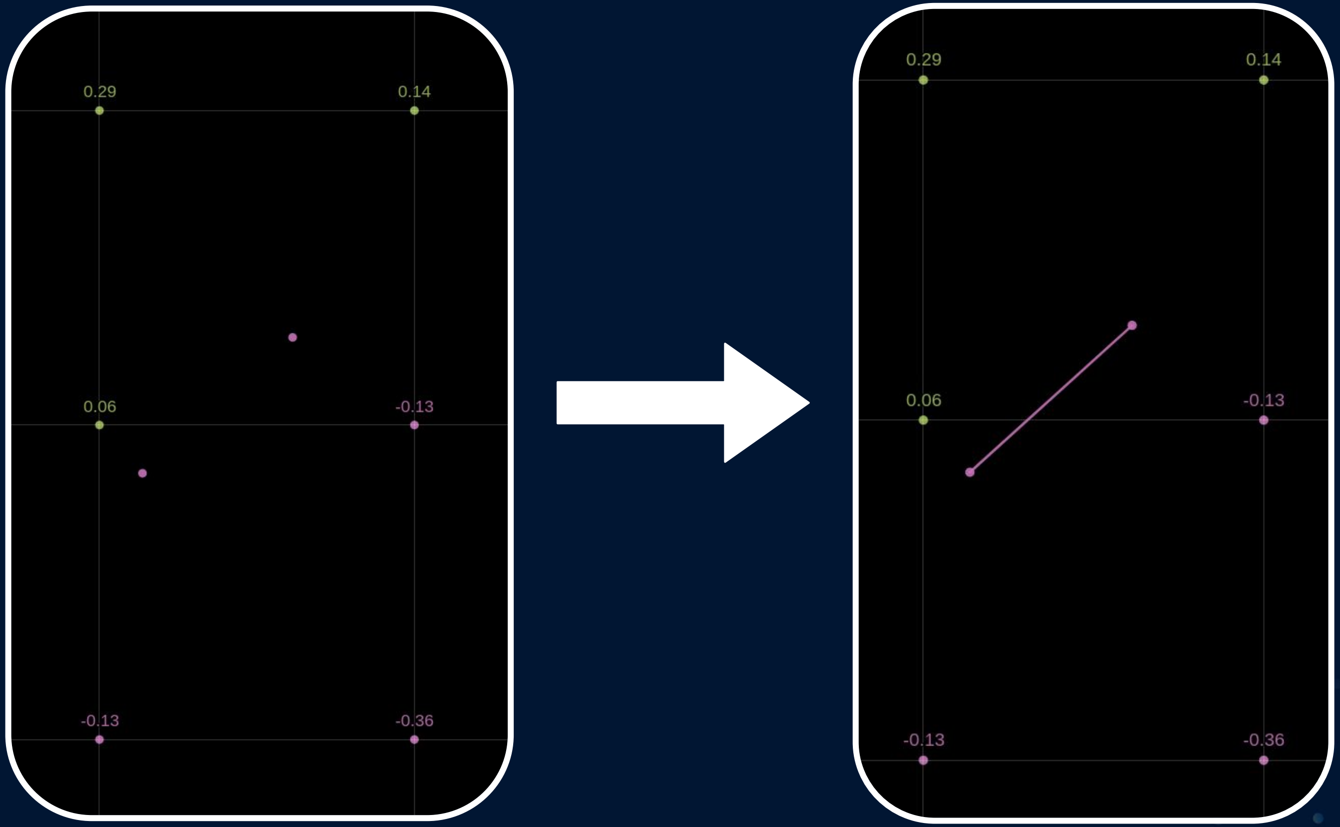
POLYGONIZATION: DUAL CONTOURING

Step 3: Connect the vertices in adjacent cells that **share a boundary with a sign change**.

In 2D, this is a line, but in 3D, this is a quad.

POLYGONIZATION: DUAL CONTOURING

Step 3:



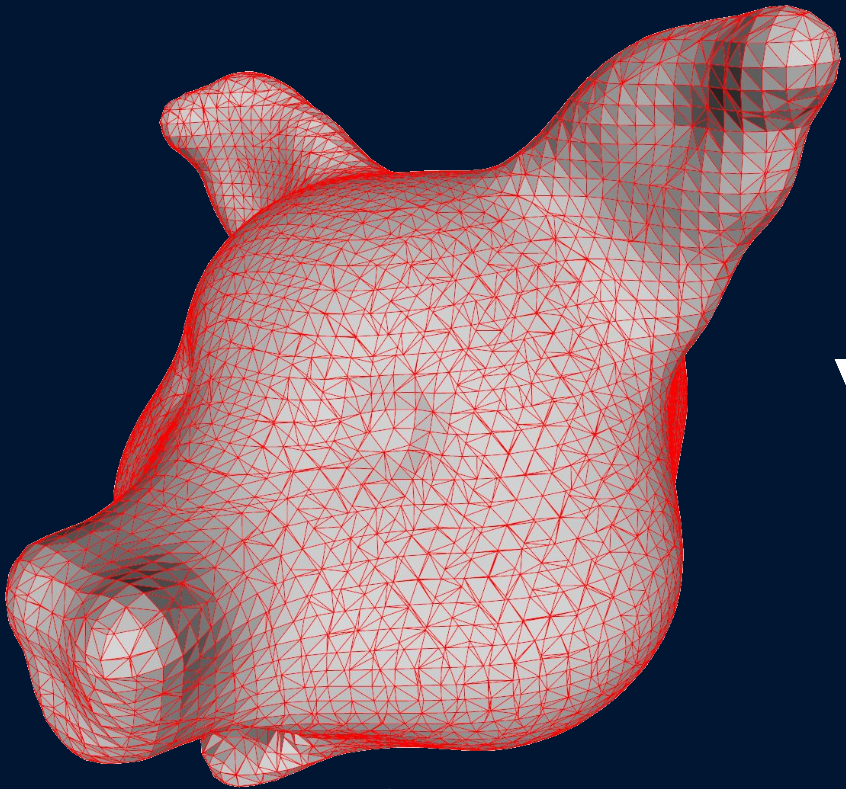
POLYGONIZATION: DUAL CONTOURING

Do this for the entire grid space, and you are done!

Minus any of the cool stuff:

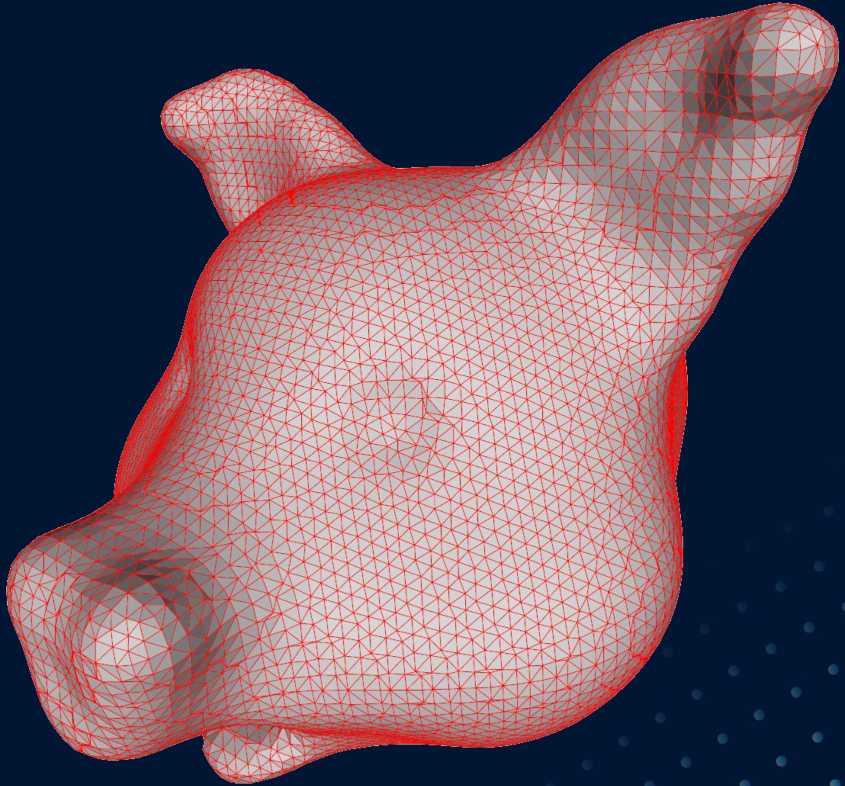
- *Multiple materials within the isosurface mesh*
- *Octrees and safe mesh simplification (LODs)*
- *How to solve the QEF in a stable fashion*
- *Handling manifold meshes.*

POLYGONIZATION: DUAL CONTOURING



Marching cubes

VS.



Dual Contouring

Summary

Ray-Marching:
- Great for continuous isosurfaces
- Directly renders isosurfaces
- Can’t render discrete data
- Infinite resolution (for SDFs)

Polygonization:
- Great for discrete data
- Allows isosurfaces to be used in traditional render pipelines.
- Discrete triangle meshes, limited resolution.

QUESTIONS?

REFERENCES

Lorensen, W.E. and Cline, H.E. (1987) “Marching cubes: A high resolution 3D surface construction algorithm,” ACM SIGGRAPH Computer Graphics, 21(4), pp. 163–169. DOI:10.1145/37402.37422.

Ju, T. et al. (2002) “Dual contouring of Hermite Data,” ACM Transactions on Graphics, 21(3), pp. 339–346. DOI: 10.1145/566654.566586.

Cool Resources:

Lague, S. (2019) Coding adventure: Ray marching, YouTube. YouTube. Available at: https://www.youtube.com/watch?v=Cp5WWtMoeKg (Accessed: November 30, 2022).

CodeParade (2018) How to make 3D fractals, YouTube. YouTube. Available at: https://www.youtube.com/watch?v=svLzmFuSBhk (Accessed: November 30, 2022).