10. Procedural Methods

Outline

- Particle Systems
- Marching Squares
- Agent Based Models
- Computing the Mandelbrot Set

Particle Systems

Introduction

- Most important of procedural methods
- Used to model
  - Natural phenomena
    - Clouds
    - Terrain
    - Plants
  - Crowd Scenes
  - Real physical processes

Newtonian Particle

- Particle system is a set of particles
- Each particle is an ideal point mass
- Six degrees of freedom
  - Position
  - Velocity
- Each particle obeys Newtons' law

$$f = ma$$

Particle Equations

$\mathbf{p}_i = (x_i, y_i\ z_i)$

$\mathbf{v}_i = d\mathbf{p}_i\ /dt = \mathbf{p}_i{}^{\text{‘}} = (dx_i\ /dt, dy_i\ /dt\ , z_i\ /dt)$

$m\ \mathbf{v}_i{}^{\text{‘}} = \mathbf{f}_i$

Hard part is defining force vector

Force Vector

- Independent Particles
  - Gravity
  - Wind forces
  - O(n) calulation
- Coupled Particles O(n)
  - Meshes
  - Spring-Mass Systems
- Coupled Particles $O(n^2)$
  - Attractive and repulsive forces

Solution of Particle Systems

```
float time, delta state[6n], force[3n];
state = initial_state();
for(time = t0; time<final_time, time+=delta) {
   force =  force_function(state, time);
   state = ode(force, state, time, delta);
   render(state, time)
}
```

Simple Forces

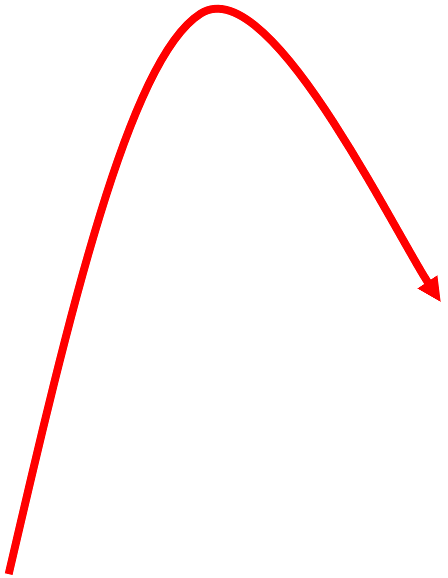
- Consider force on particle i

$$\mathbf{f}_i = \mathbf{f}_i(\mathbf{p}_i, \mathbf{v}_i)$$

- Gravity $\mathbf{f}_i = \mathbf{g}$

$$\mathbf{g}_i = (0, -g, 0)$$

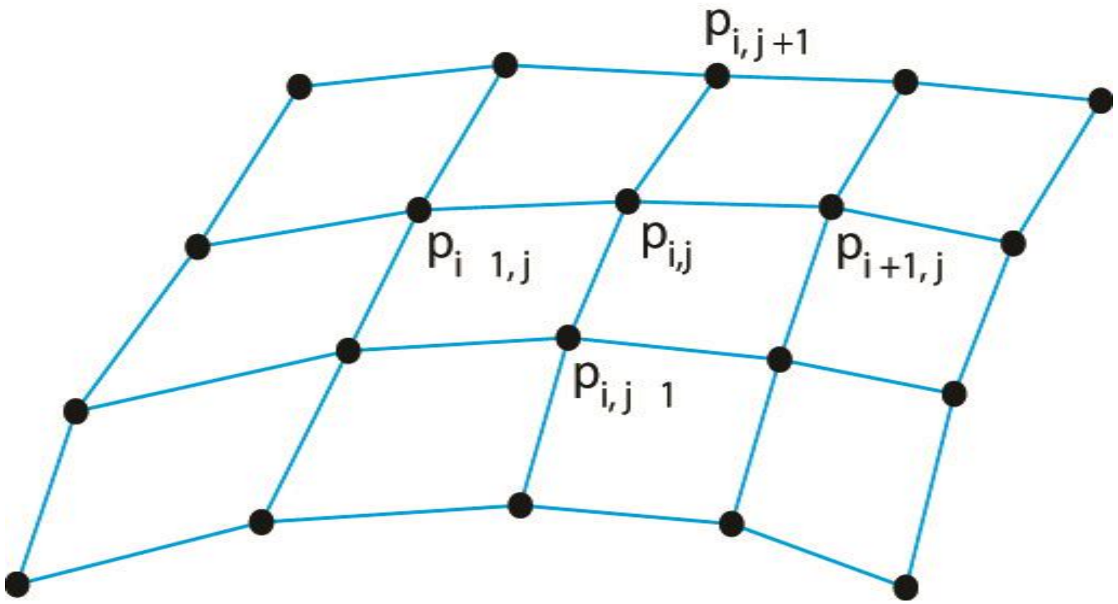
- Wind forces
- Drag



$\mathbf{p}_i(t_0), \mathbf{v}_i(t_0)$

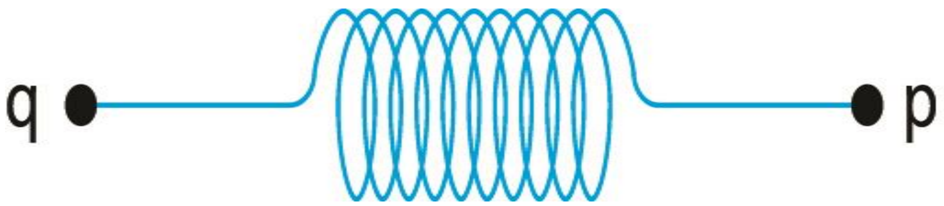
Meshes

- Connect each particle to its closest neighbors
  - O(n) force calculation
- Use spring-mass system



Spring Forces

- Assume each particle has unit mass and is connected to its neighbor(s) by a spring
- Hooke's law: force proportional to distance ($d = \|\mathbf{p} - \mathbf{q}\|$) between the points



Hooke's Law

- Let s be the distance when there is no force

$$\mathbf{f} = -k_s(|\mathbf{d}| - s)\, \mathbf{d}/|\mathbf{d}|$$

k_s is the spring constant

$\mathbf{d}/|\mathbf{d}|$ is a unit vector pointed from $\mathbf{p}$ to $\mathbf{q}$

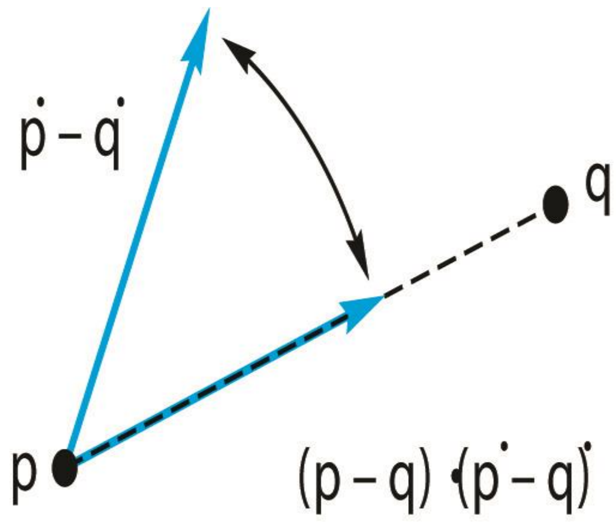
- Each interior point in mesh has four forces applied to it

Spring Damping

- A pure spring-mass will oscillate forever
- Must add a damping term

$$\mathbf{f} = -(k_s(|\mathbf{d}| - s) + k_d\, \mathbf{d}\cdot\dot{\mathbf{d}}/|\mathbf{d}|)\mathbf{d}/|\mathbf{d}|$$

- Must project velocity



Attraction and Repulsion

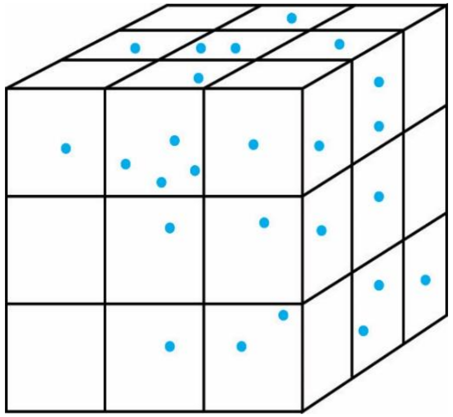
- Inverse square law

$$\mathbf{f} = -k_r\mathbf{d}/|\mathbf{d}|^3$$

- General case requires $O(n^2)$ calculation
- In most problems, the drop off is such that not many particles contribute to the forces on any given particle
- Sorting problem: is it $O(n \log n)$?

Boxes

- Spatial subdivision technique
- Divide space into boxes
- Particle can only interact with particles in its box or the neighboring boxes
- Must update which box a particle belongs to after each time step



Linked Lists

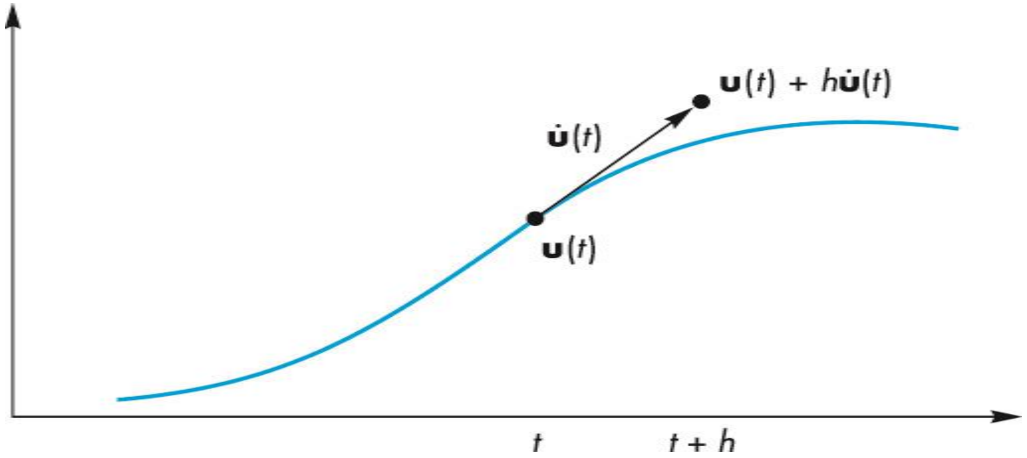
- Each particle maintains a linked list of its neighbors
- Update data structure at each time step
- Must amortize cost of building the data structures initially

Particle Field Calculations

- Consider simple gravity
- We don’t compute forces due to sun, moon, and other large bodies
- Rather we use the gravitational field
- Usually we can group particles into equivalent point masses

Solution of ODEs

- Particle system has 6n ordinary differential equations
- Write set as $d\mathbf{u}/dt = g(\mathbf{u},t)$
- Solve by approximations using Taylor's Thm



Euler's Method

$$\mathbf{u}(t + h) \approx \mathbf{u}(t) + h\, d\mathbf{u}/dt = \mathbf{u}(t) + h\mathbf{g}(\mathbf{u}, t)$$

Per step error is $O(h^2)$

Require one force evaluation per time step

Problem is numerical instability

depends on step size

Improved Euler

$$\mathbf{u}(t + h) \approx \mathbf{u}(t) + h/2(\mathbf{g}(\mathbf{u}, t) + \mathbf{g}(\mathbf{u}, t+h))$$

Per step error is $O(h^3)$

Also allows for larger step sizes

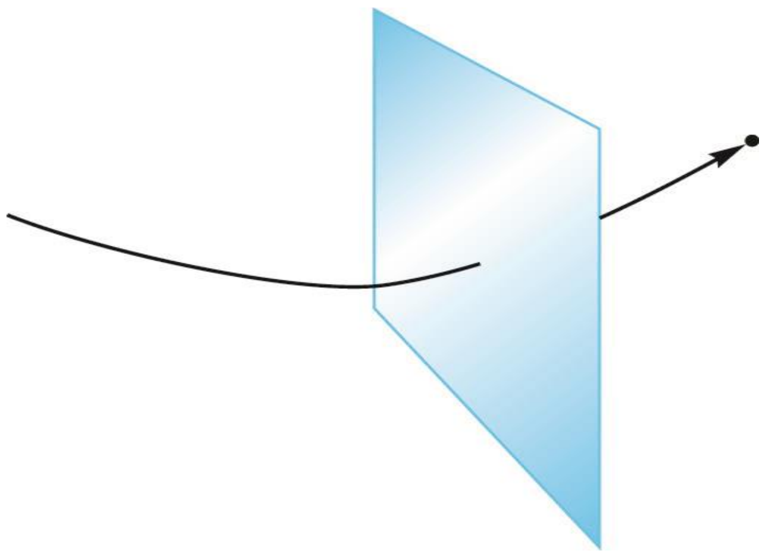
But requires two function evaluations per step

Also known as Runge-Kutta method of order 2

Contraints

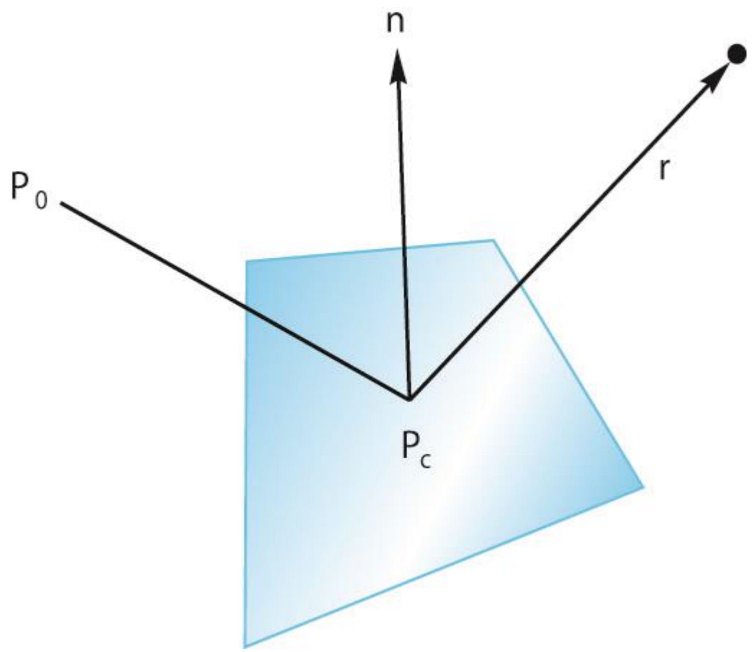
- Easy in computer graphics to ignore physical reality
- Surfaces are virtual
- Must detect collisions separately if we want exact solution
- Can approximate with

repulsive forces

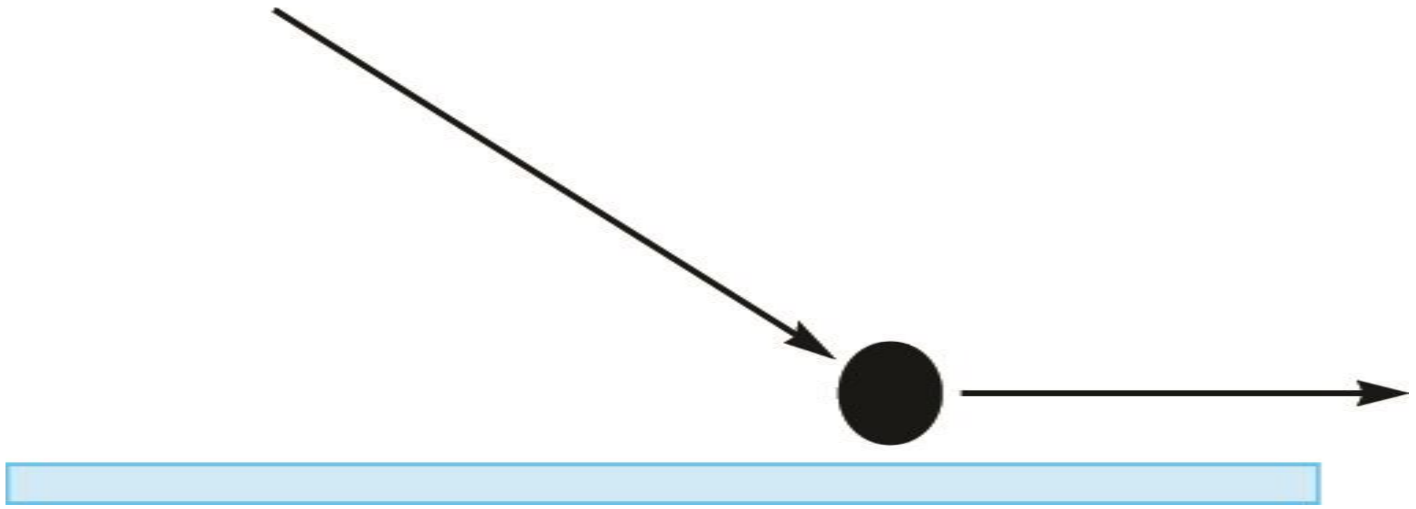


Collisions

- Once we detect a collision, we can calculate new path
- Use coefficient of resititution
- Reflect vertical component
- May have to use partial time step



Contact Forces



Marching Squares

Objectives

- Nontrivial two-dimensional application
- Important method for
    - Contour plots
    - Implicit function visualization
- Extends to important method for volume visualization
- This lecture is optional but should be interesting to most of you

Displaying Implicit Functions

- Consider the implicit function

$$g(x,y)=0$$

- Given an x, we cannot in general find a corresponding y
- Given an x and a y, we can test if they are on the curve

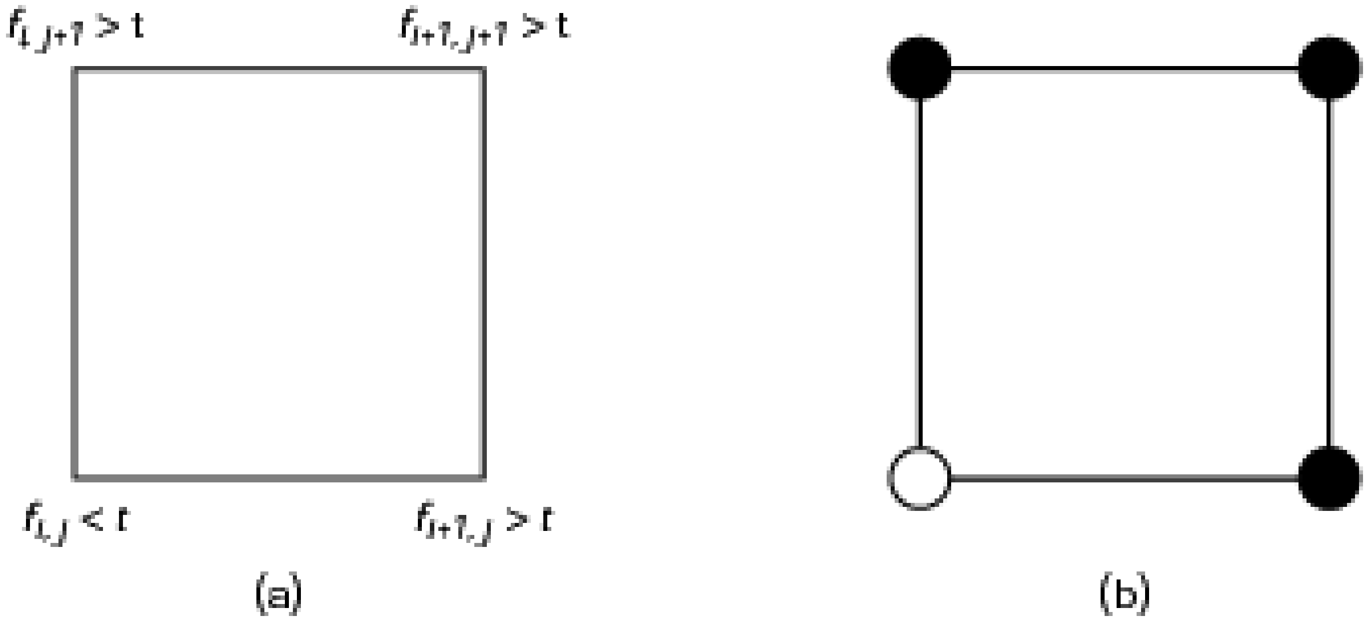
Height Fields and Contours

- In many applications, we have the heights given by a function of the form $z=f(x,y)$
- To find all the points that have a given height t, we have to solve the implicit equation $g(x,y)=f(x,y)-t=0$
- Such a function determines the **isocurves** or **contours** of f for the **isovalue** t

Marching Squares

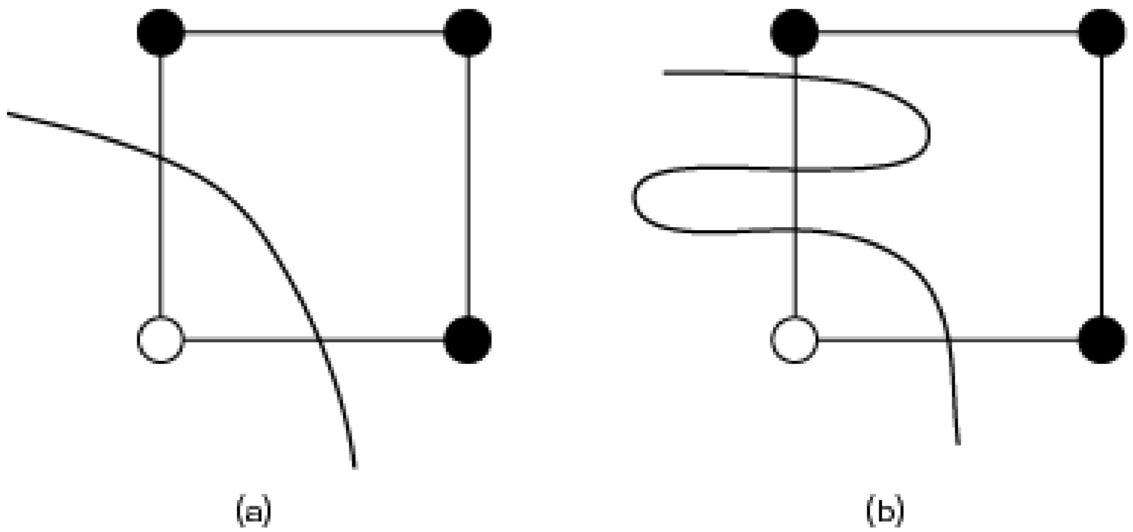
- Displays isocurves or contours for functions $f(x,y) = t$
- Sample $f(x,y)$ on a regular grid yielding samples $\{f_{ij}(x,y)\}$
- These samples can be greater than, less than, or equal to t
- Consider four samples $f_{ij}(x,y)$, $f_{i+1,j}(x,y)$, $f_{i+1,j+1}(x,y)$, $f_{i,j+1}(x,y)$
- These samples correspond to the corners of a cell
- Color the corners by whether they exceed or are less than the contour value t

Cells and Coloring

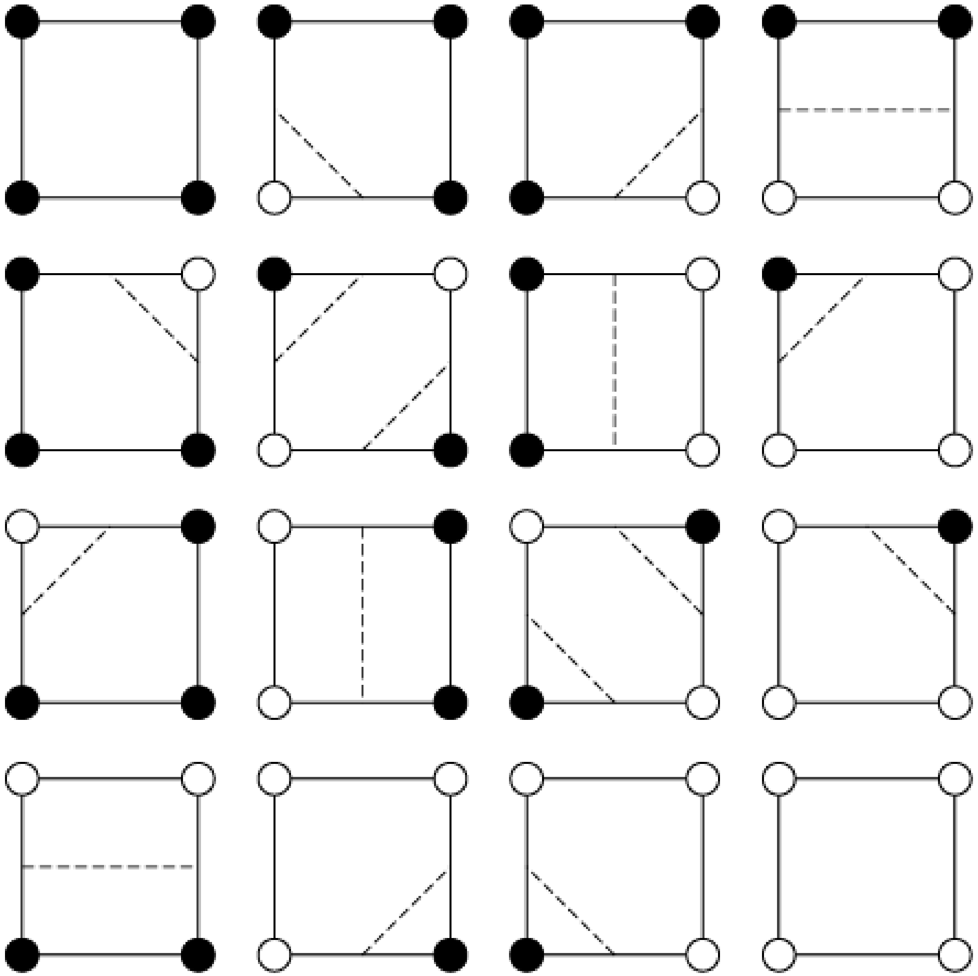


Occum’s Razor

- Contour must intersect edge between a black and white vertex an odd number of times
- Pick simplest interpretation: one crossing

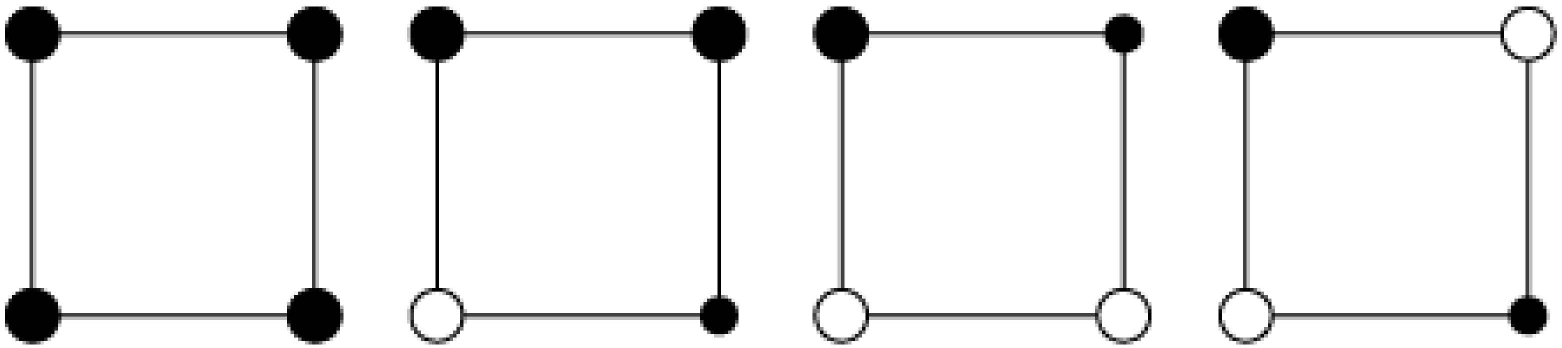


16 Cases



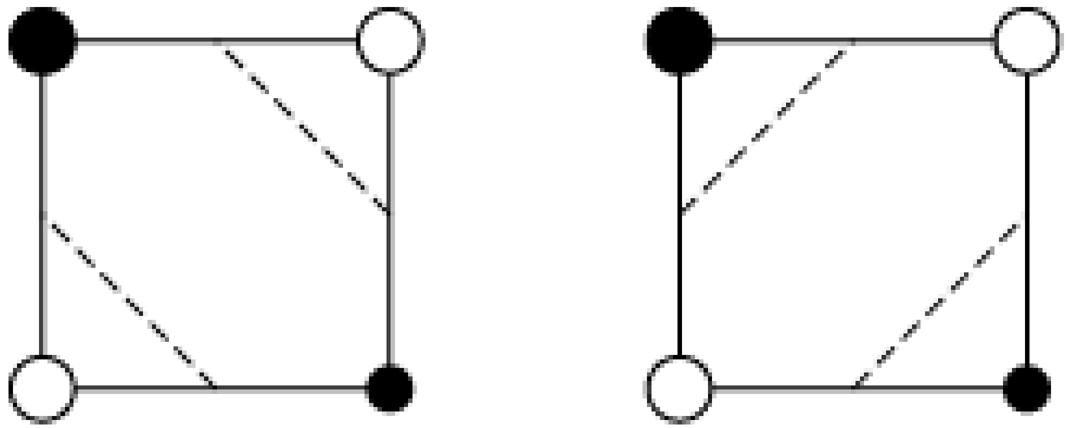
Unique Cases

- Taking out rotational and color swapping symmetries leaves four unique cases
- First three have a simple interpretation



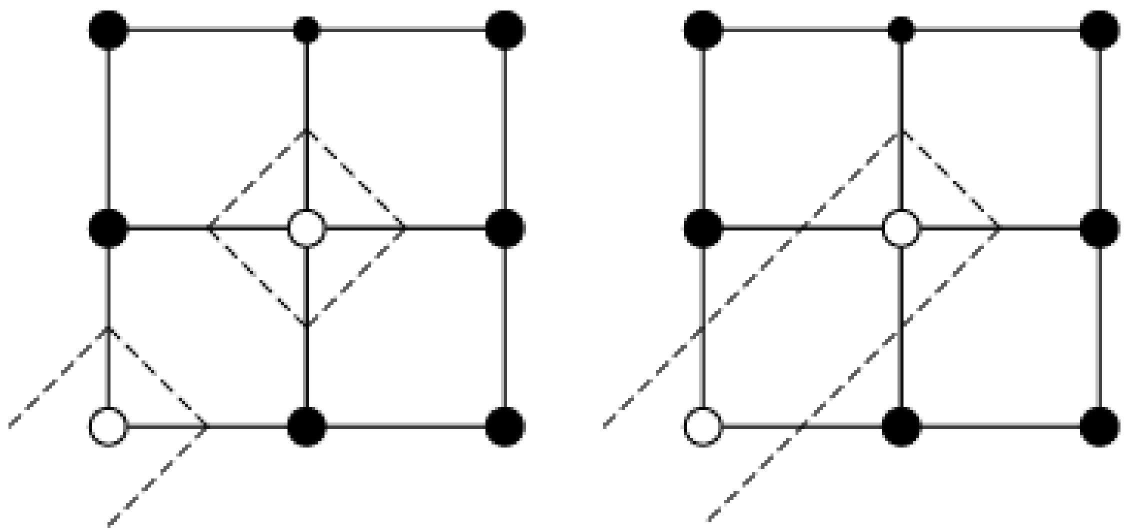
Ambiguity Problem

- Diagonally opposite cases have two equally simple possible interpretations

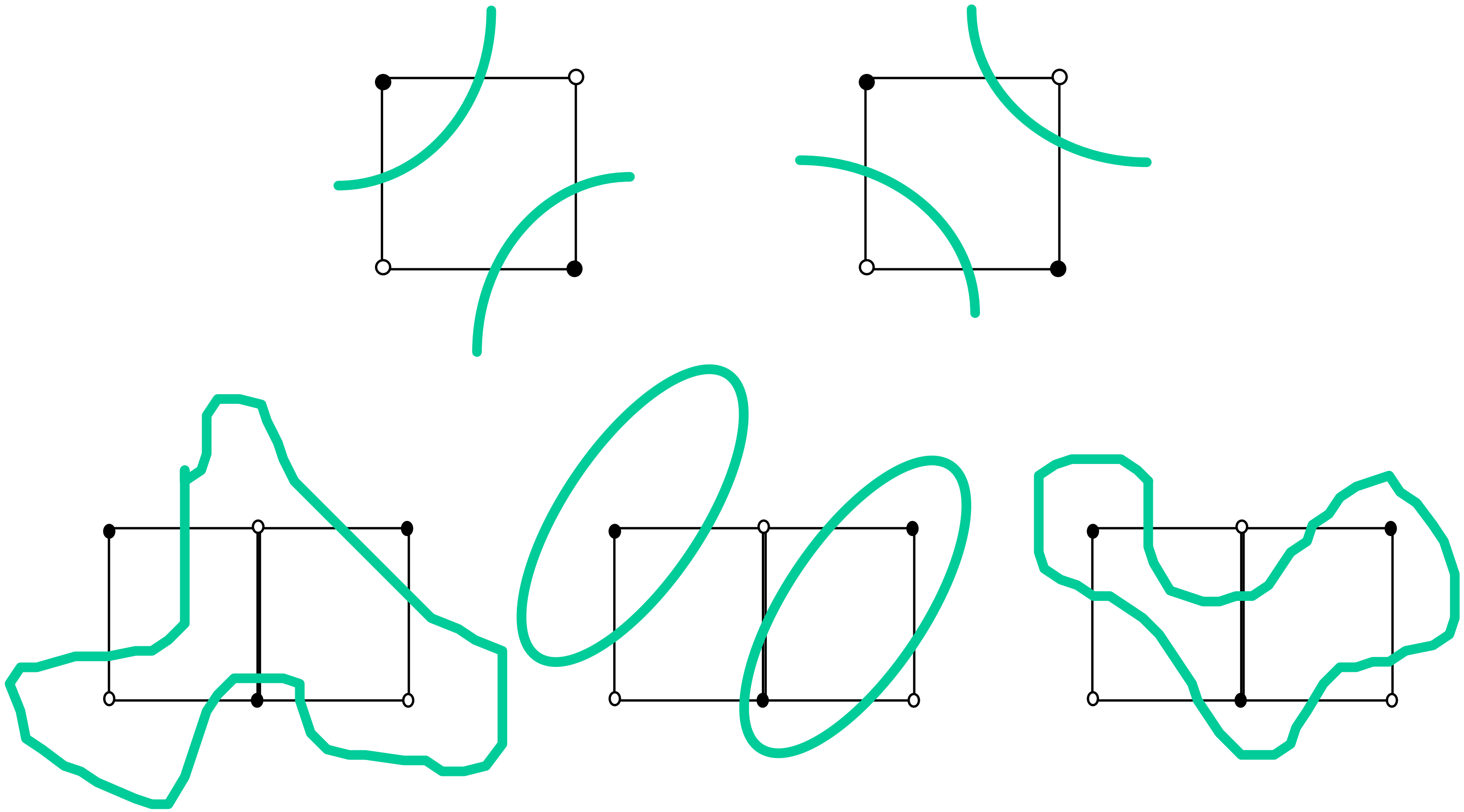


Ambiguity Example

- Two different possibilities below
- More possibilities on next slide



Ambiguity Problem

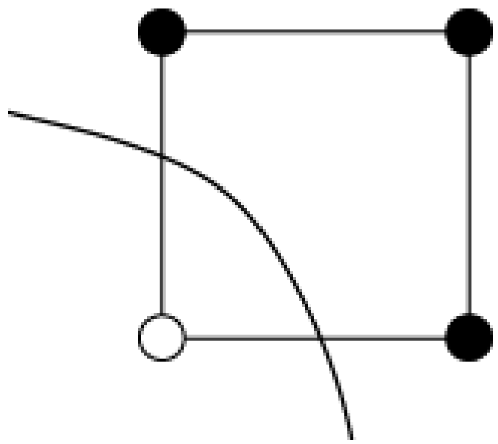


Is Problem Resolvable?

- Problem is a sampling problem
  - Not enough samples to know the local detail
  - No solution in a mathematical sense without extra information
- More of a problem with volume extension (marching cubes) where selecting “wrong” interpretation can leave a hole in a surface
- Multiple methods in literature to give better appearance
  - Supersampling
  - Look at larger area before deciding

Interpolating Edges

- We can compute where contour intersects edge in multiple ways
  - Halfway between vertics
  - Interpolated based on difference between contour value and value at vertices

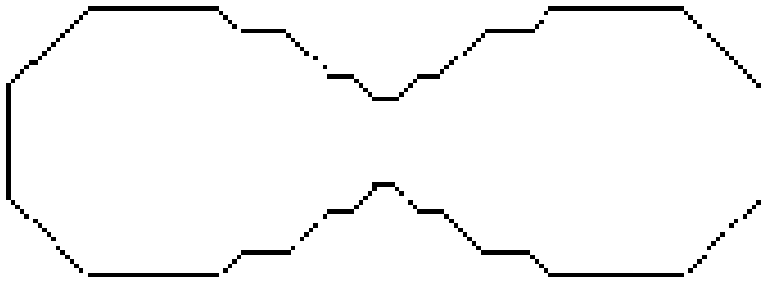


Example: Oval of Cassini

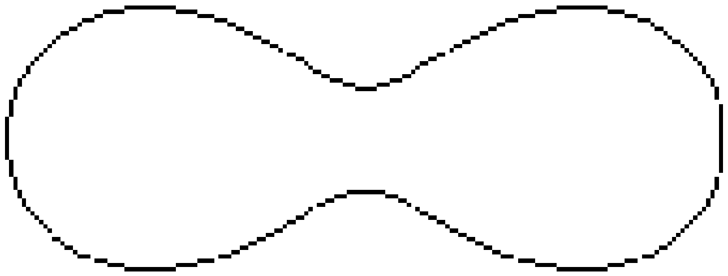
$f(x,y)=(x^2+y^2+a^2)^2-4a^2x^2-b^4$

Depending on a and b we can have 0, 1, or 2 curves

midpoint intersections

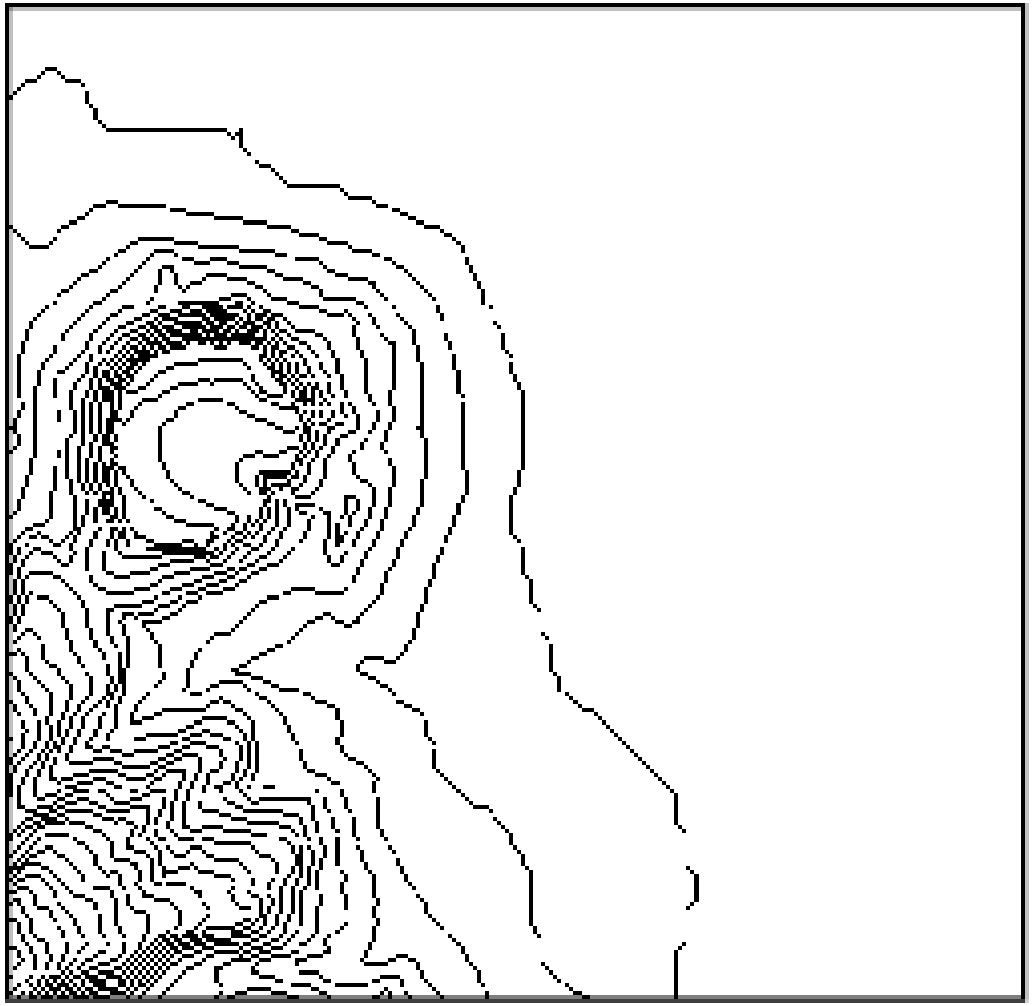


interpolating intersections



Contour Map

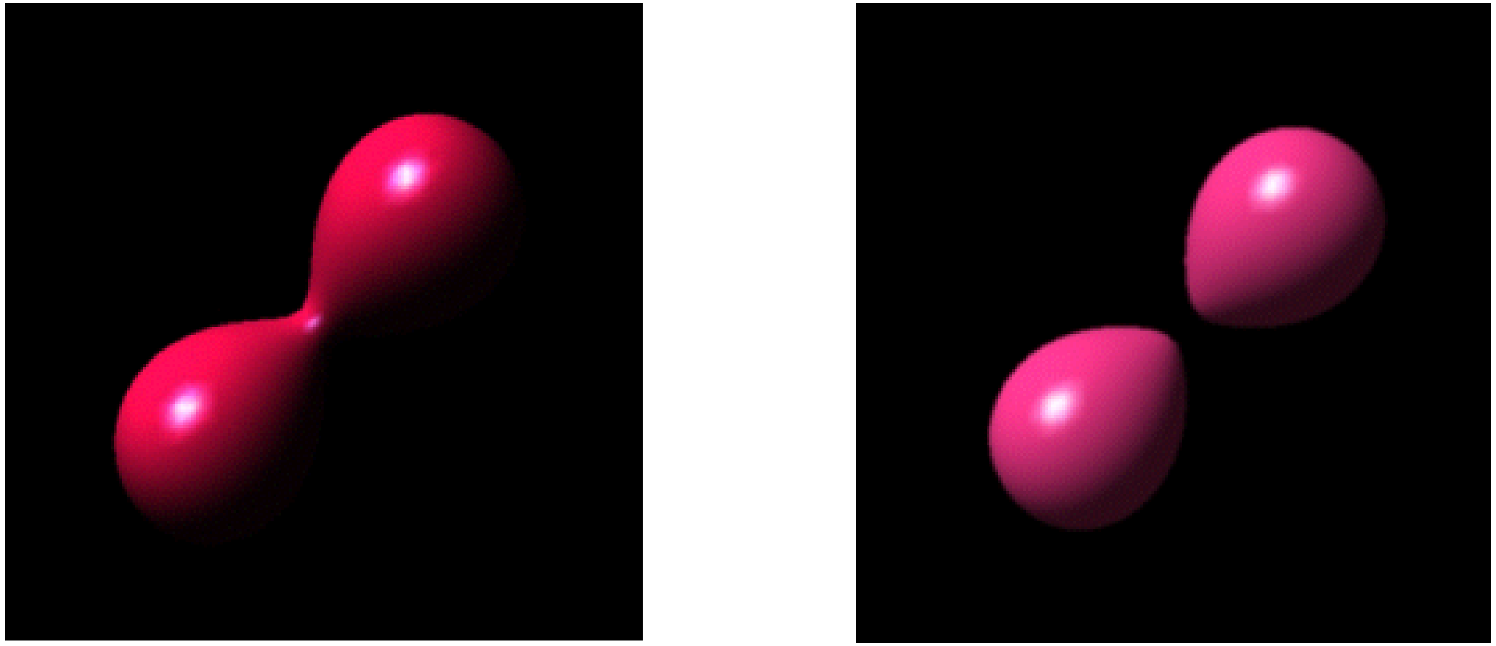
- Diamond Head, Oahu Hawaii
- Shows contours for many contour values



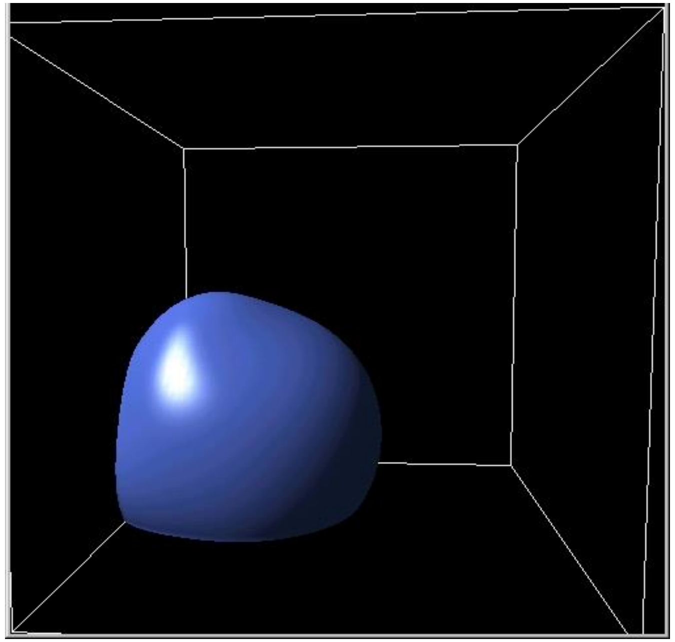
Marching Cubes

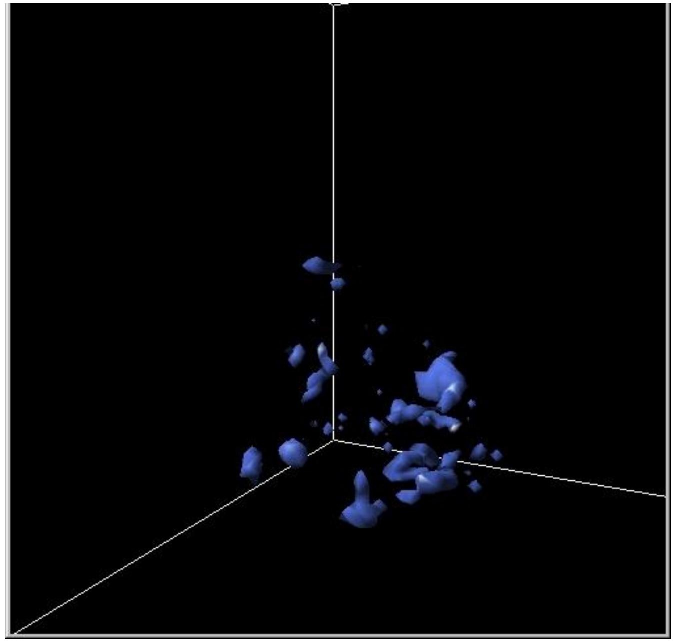
- Isosurface: solution of g(x,y,z)=c
- Use same argument to derive method but with a cubic cell (8 vertices, 256 colorings)
- Standard method of volume visualization
- Suggested by Lorensen and Kline before marching squares
- Note inherent parallelism of both marching cubes and marching squares

Marching Cubes

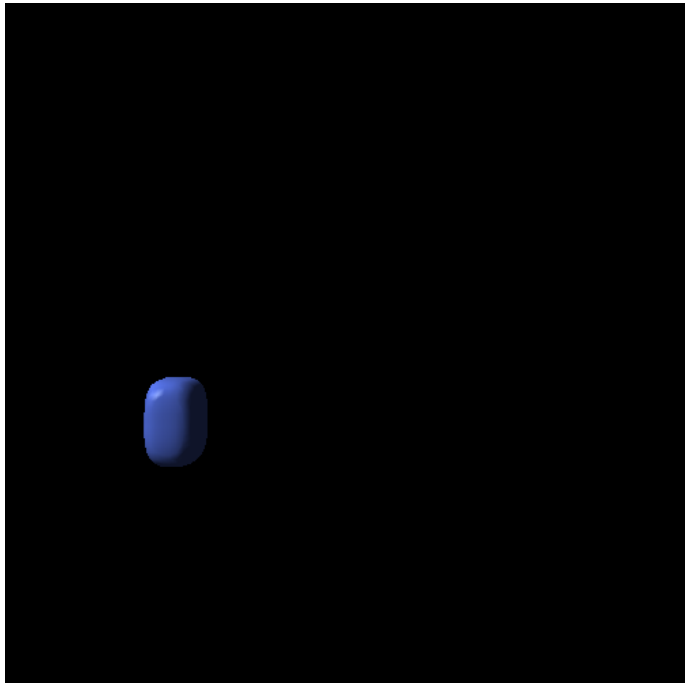
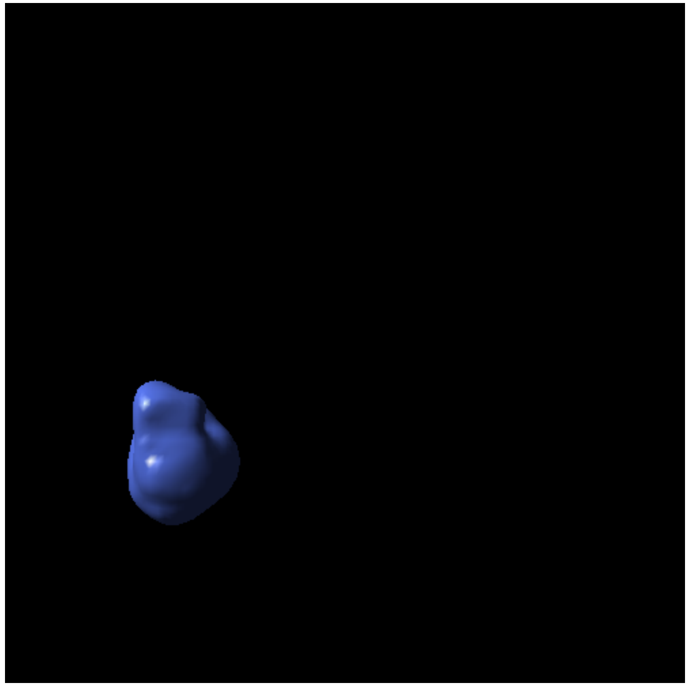


Marching Cubes

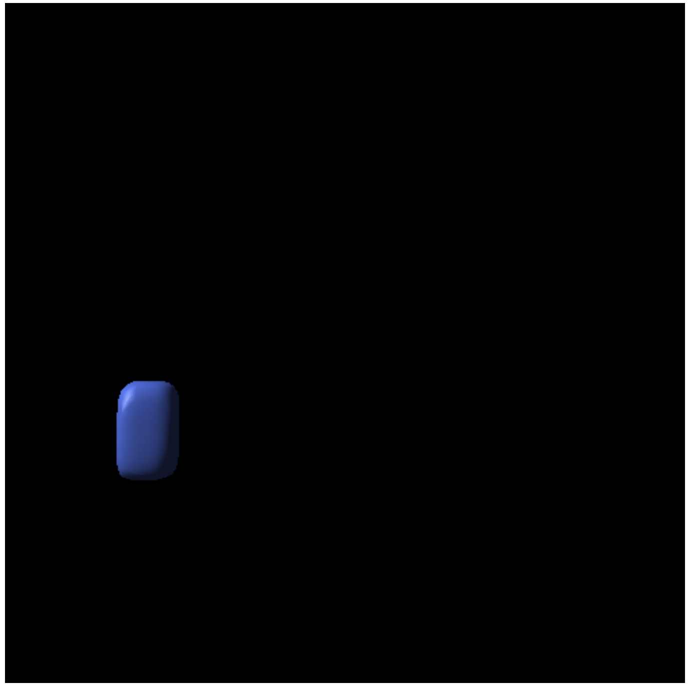


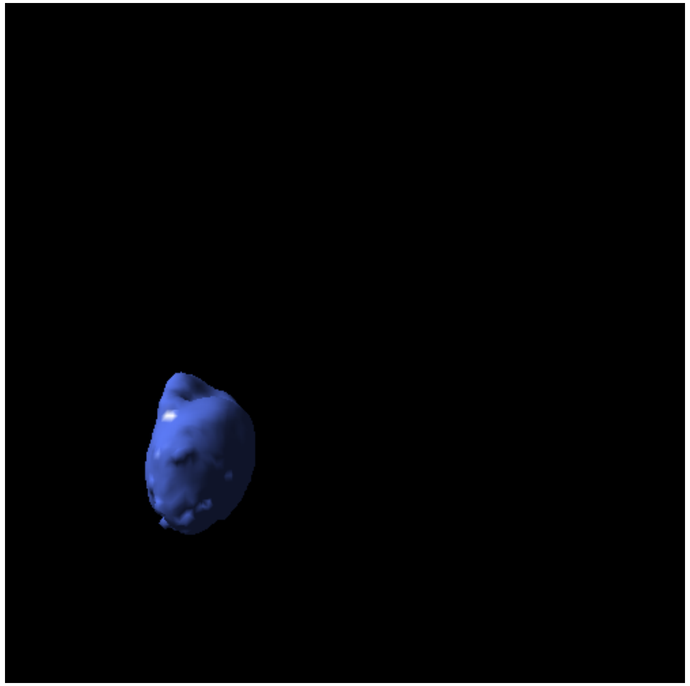


Marching Cubes

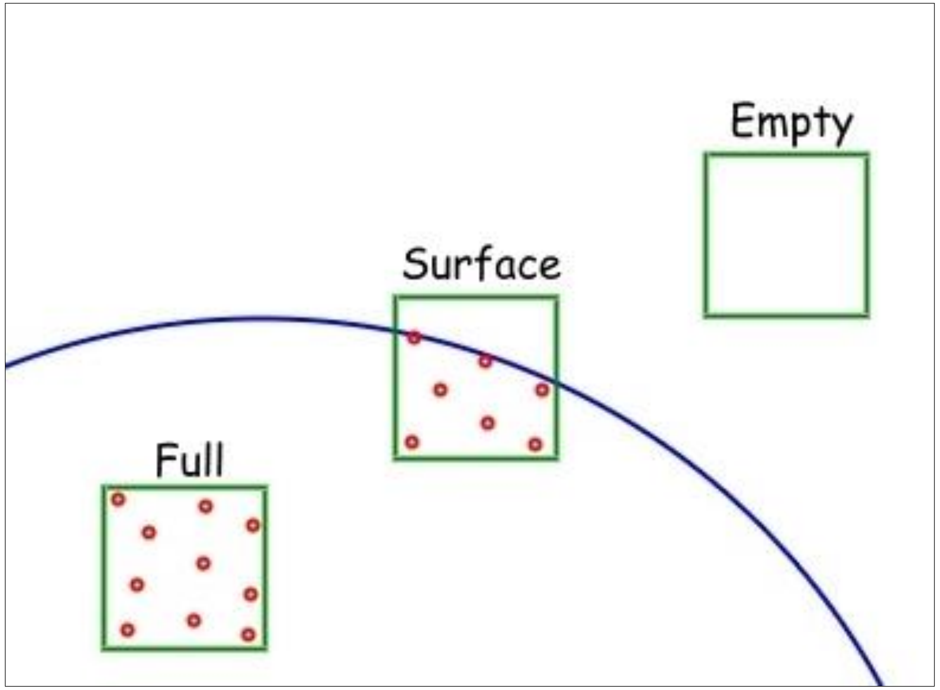
 

Marching Cubes

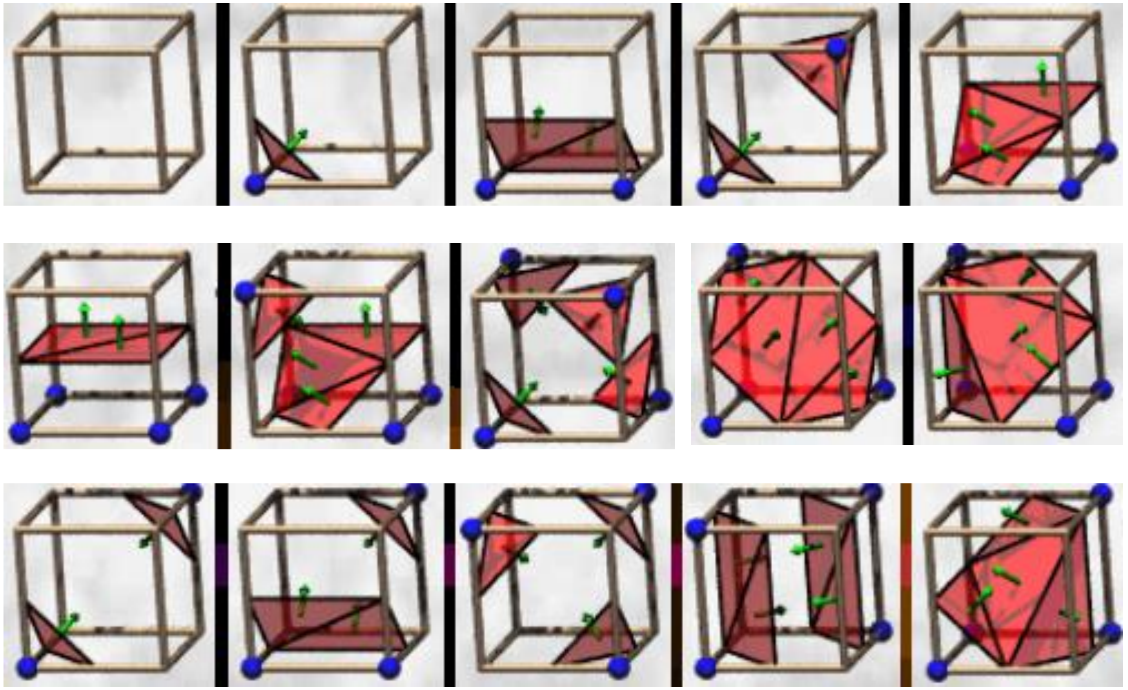




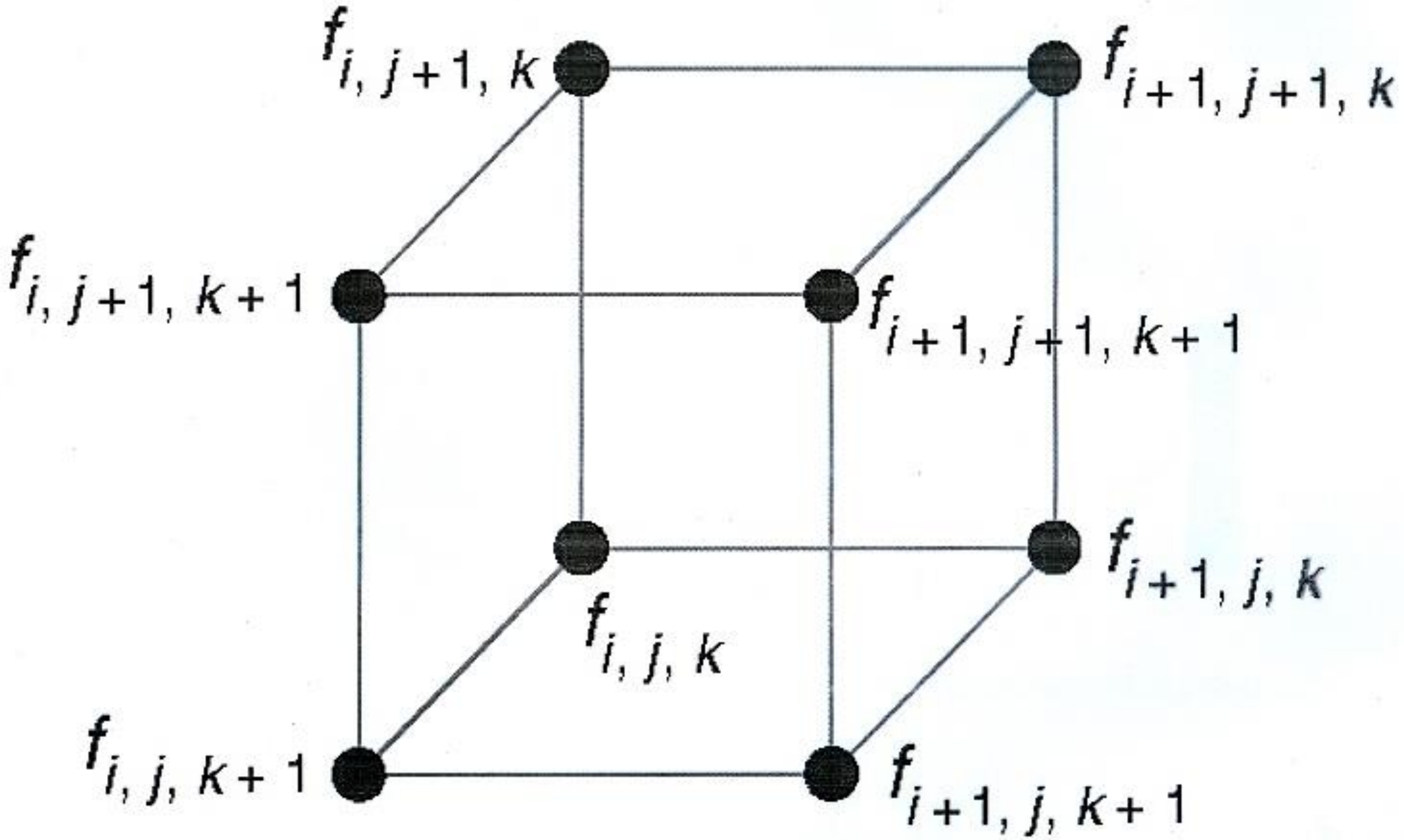
Marching Cubes



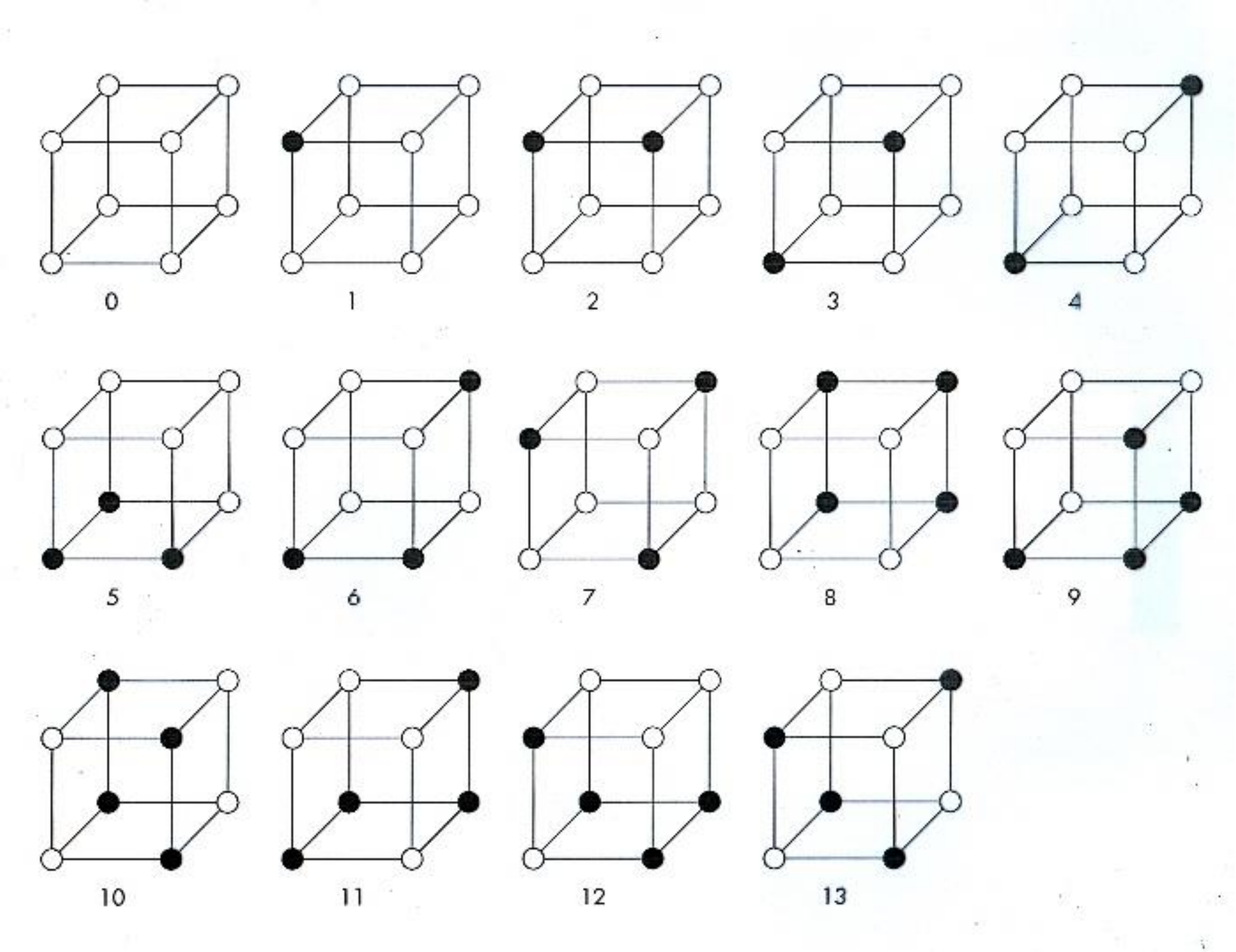
Marching Cubes



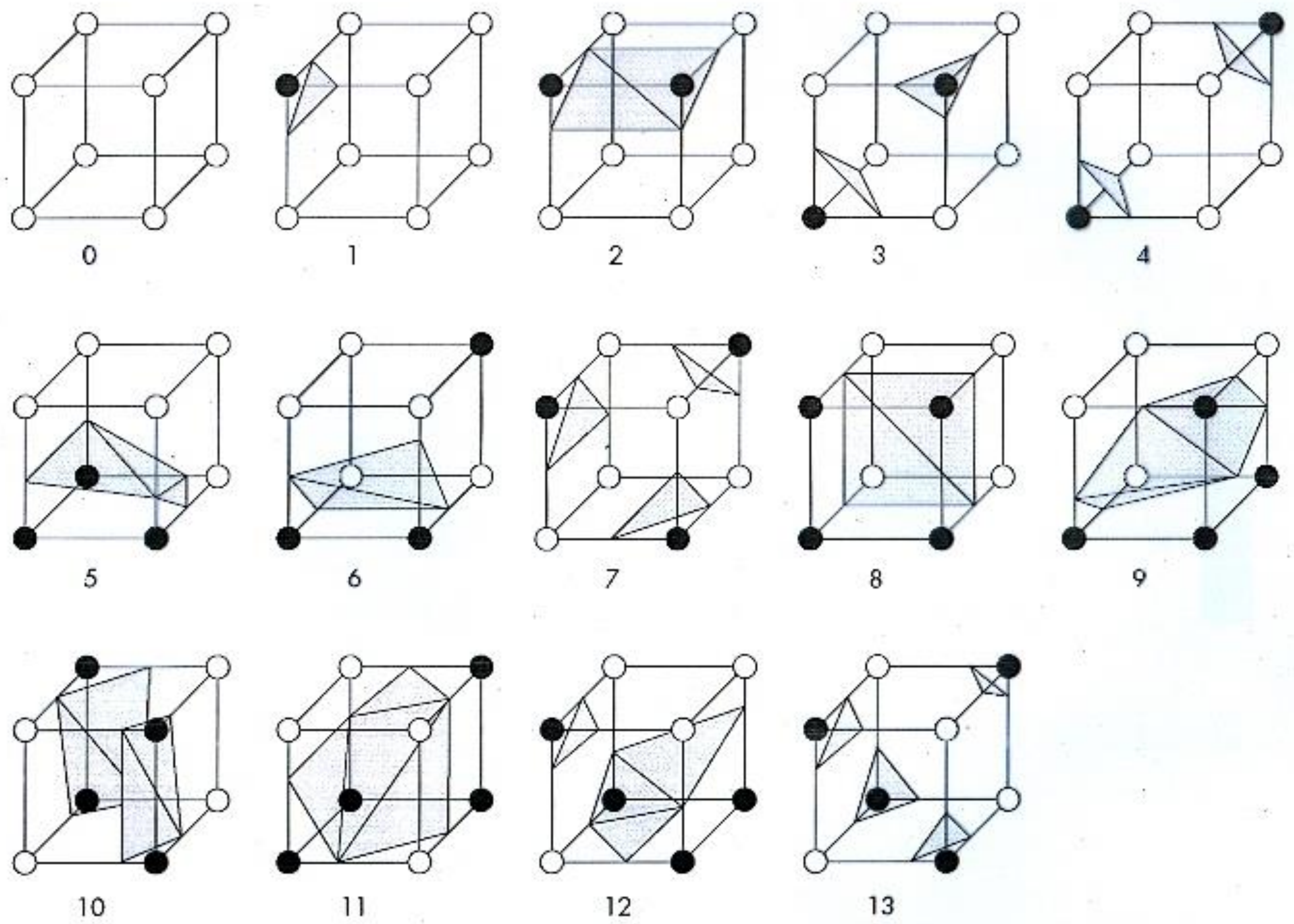
Marching Cubes



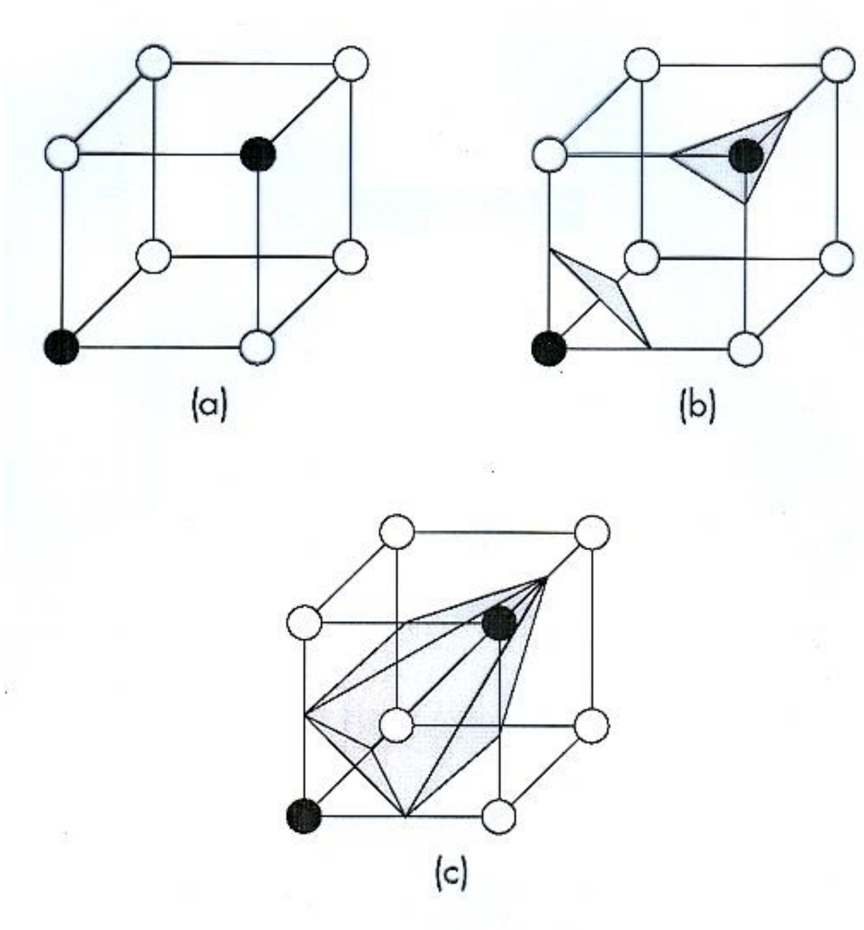
Marching Cubes



Marching Cubes



Marching Cubes



Agent Based Models

Objectives

- Introduce a powerful form of simulation
- Use render-to-texture for dynamic simulations using agent-based models
- Example of diffusion

Agent Based Models (ABMs)

- Consider a particle system in which particle can be programmed with individual behaviors and properties
  - different colors
  - different geometry
  - different rules
- Agents can interact with each other and with the environment

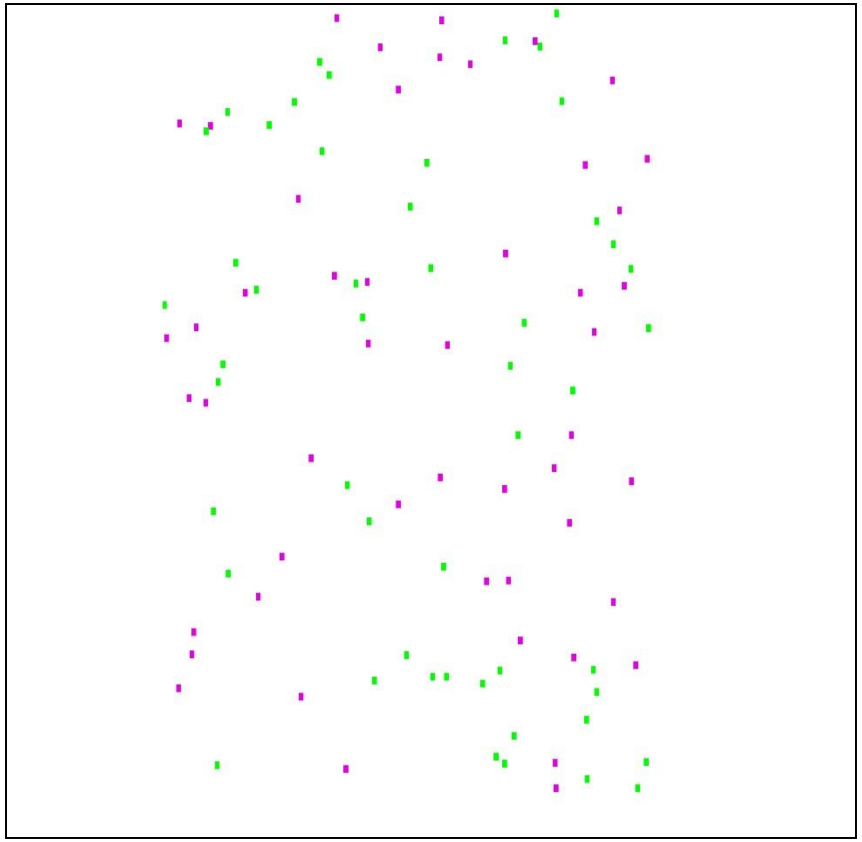
Simulating Ant Behavior

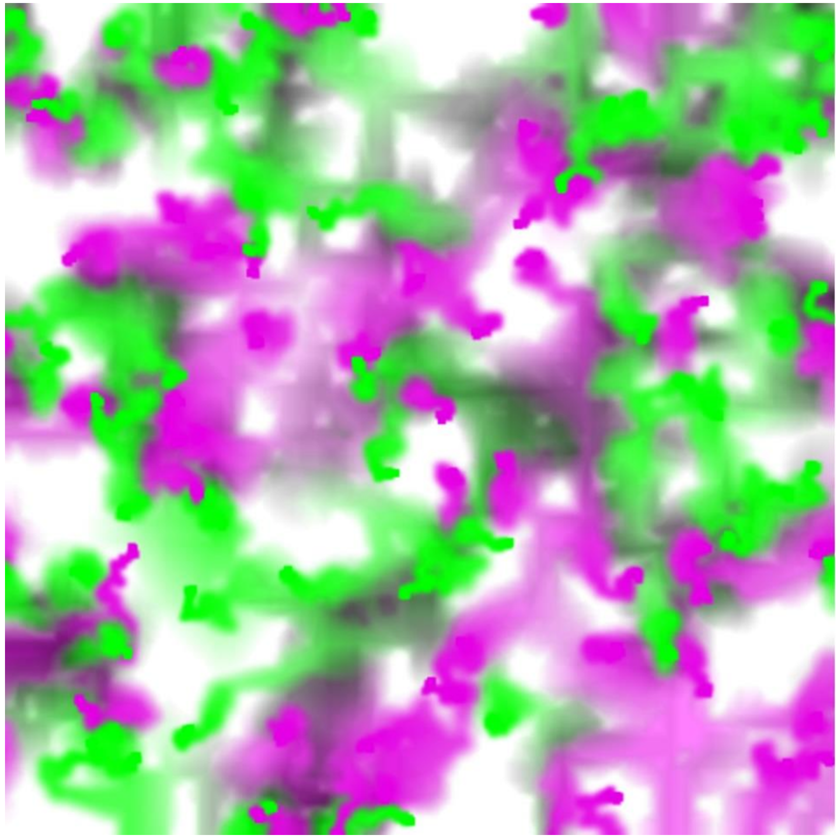
- Consider ants searching for food
- At the beginning, an ant moves randomly around the terrain searching for food
  - The ant can leave a chemical marker called a pheromone to indicate the spot was visited
  - Once food is found, other ants can trace the path by following the pheromone trail
- Model each ant as a point moving over a surface
- Render each point with arbitrary geometry

Diffusion Example I

- Two types of agents
  - no interaction with environment
  - differ only in color
- All move randomly
- Leave position information
  - need render-to-texture
- Diffuse position information
  - need buffer pingponging

Snapshots





Initialization

- We need two program objects
  - One for rendering points in new positions
  - One for diffusing texture map
- Initialization is standard otherwise
  - setup texture objects
  - setup framebuffer object
  - distribute particles in random locations

Vertex Shader 1

```
attribute vec4 vPosition1;
attribute vec2 vTexCoord;
varying vec2 fTexCoord;
void main()
{
    gl_Position = vPosition1;
    fTexCoord = vTexCoord;
}
```

Fragment Shader 1

```
precision mediump float;
uniform sampler2D texture;
uniform float d;
uniform float s;
varying vec2 fTexCoord;
void main()
 {
   float x = fTexCoord.x;
   float y = fTexCoord.y;
   gl_FragColor = (texture2D( texture, vec2(x+d, y))
                  +texture2D( texture, vec2(x, y+d))
                  +texture2D( texture, vec2(x-d, y))
                  +texture2D( texture, vec2(x, y-d)))/s;
 }
```

Vertex Shader 2

```
attribute  vec4 vPosition2;
uniform float pointSize;
void main()
 {
    gl_PointSize = pointSize;
    gl_Position = vPosition2;
}
```

Fragment Shader 2

```
precision mediump float;
uniform vec4 color;
void main()
{
   gl_FragColor = color;
}
```

Rendering Loop I

```
var render = function(){
  // render to texture
  // first a rectangle that is texture mapped
   gl.useProgram(program1);
   gl.bindFramebuffer( gl.FRAMEBUFFER, framebuffer);
   if(flag) {
       gl.bindTexture(gl.TEXTURE_2D, texture1);
       gl.framebufferTexture2D(gl.FRAMEBUFFER,
       gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D, texture2, 0);
   }
   else {
       gl.bindTexture(gl.TEXTURE_2D, texture2);
       gl.framebufferTexture2D(gl.FRAMEBUFFER,
          gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D, texture1, 0);
   }
  gl.drawArrays( gl.TRIANGLE_STRIP, 0, 4 );
```

Rendering Loop II

```
// render points
    gl.useProgram(program2);
    gl.vertexAttribPointer( vPosition2, 2, gl.FLOAT, false, 0, 0);
    gl.uniform4f( gl.getUniformLocation(program2, "color"), 0.9, 0.0, 0.9, 1.0);
     gl.drawArrays(gl.POINTS, 4, numPoints/2);
    gl.uniform4f( gl.getUniformLocation(program2, "color"), 0.0, 9.0, 0.0, 1.0);
    gl.drawArrays(gl.POINTS, 4+numPoints/2, numPoints/2);
 // render to display
    gl.useProgram(program1);
    gl.vertexAttribPointer( texLoc, 2, gl.FLOAT, false, 0, 32+8*numPoints);
    gl.generateMipmap(gl.TEXTURE_2D);
    gl.bindFramebuffer(gl.FRAMEBUFFER, null);
// pick texture
    if(flag) gl.bindTexture(gl.TEXTURE_2D, texture2);
    else gl.bindTexture(gl.TEXTURE_2D, texture1);
```

Rendering Loop III

```
    var r = 1024/texSize;
    gl.viewport(0, 0, r*texSize, r*texSize);
    gl.clear( gl.COLOR_BUFFER_BIT );
    gl.drawArrays( gl.TRIANGLE_STRIP, 0, 4 );
    gl.viewport(0, 0, texSize, texSize);
    gl.useProgram(program2);
// move particles in a random direction with wrap around
    for(var i=0; i<numPoints; i++) {
         vertices[4+i][0] += 0.01*(2.0*Math.random()-1.0);
        vertices[4+i][1] += 0.01*(2.0*Math.random()-1.0);
        if(vertices[4+i][0]>1.0) vertices[4+i][0]-= 2.0;
        if(vertices[4+i][0]<-1.0) vertices[4+i][0]+= 2.0;
        if(vertices[4+i][1]>1.0) vertices[4+i][1]-= 2.0;
        if(vertices[4+i][1]<-1.0) vertices[4+i][1]+= 2.0;
}
gl.bufferSubData(gl.ARRAY_BUFFER,  0, flatten(vertices));
```

Rendering Loop IV

```
// swap textures
    flag = !flag;
    requestAnimFrame(render);
}
```

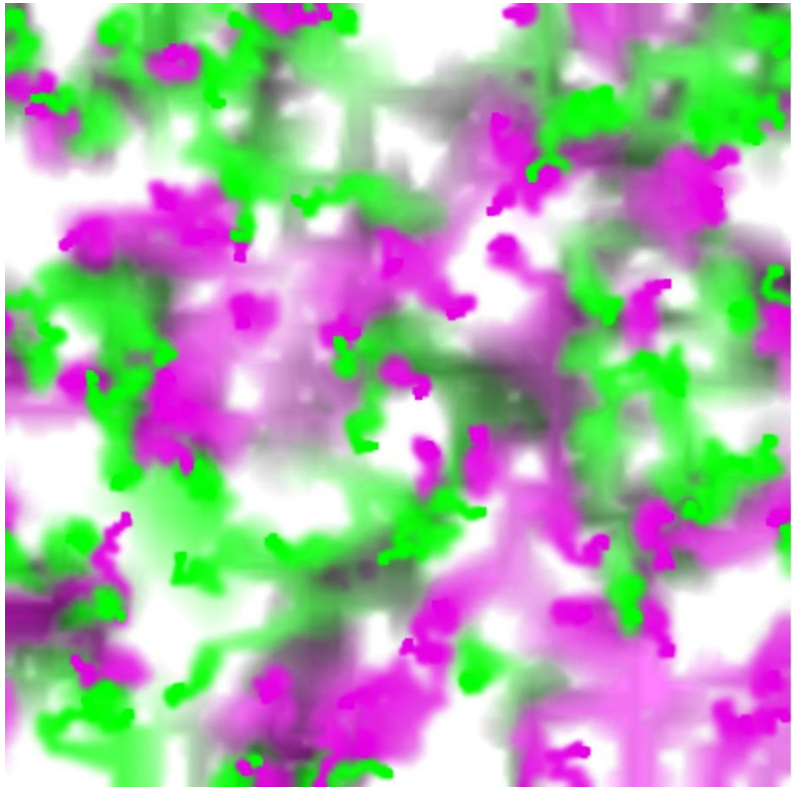
Add Agent Behavior

- Move randomly
- Check color where particle is located
- If green particle sees a green component over 128 move to (0.5, 0.5)
- If magenta particle sees a red component over 128 move to (-0.5, -0.5)

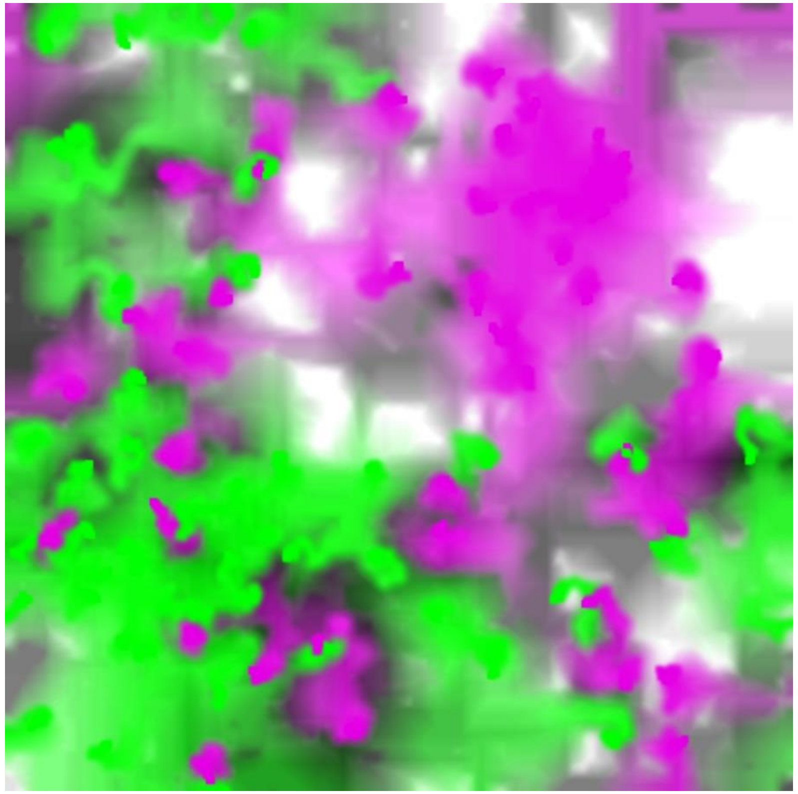
Diffusion Code

```
var color = new Uint8(4);
for(var i=0; i<numPoints/2; i++) {
        var x = Math.floor(511*(vertices[4+i][0]));
        var y = Math.floor(511*(vertices[4+i][1]));
        gl.readPixels(x, y, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, color);
       if(color[0]>128) {
             vertices[4+i][0] = 0.5;
             vertices[4+i][1] = 0.5;
    }
    for(var i=numPoints/2; i<numPoints; i++) {
        var x = Math.floor(511*(vertices[4+i][0]));
        var y = Math.floor(511*(vertices[4+i][1]));
        gl.readPixels(x, y, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, color);
        if(color[1]>128) {
            vertices[4+i][0] = -0.5;
            vertices[4+i][1] = -0.5;
    }
```

Snapshots



without reading color



with reading color

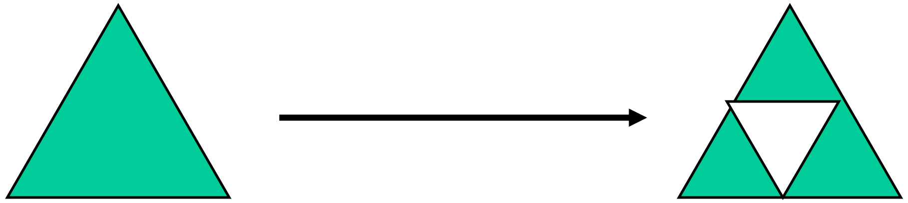
Computing the Mandelbrot Set

Objectives

- Introduce the most famous fractal object
  - more about fractal curves and surfaces later
- Imaging calculation
  - Must compute value for each pixel on display
  - Shows power of fragment processing

Sierpinski Gasket

Rule based:



Repeat n times. As n $\rightarrow\infty$

Area$\rightarrow$0

Perimeter $\rightarrow\infty$

Not a normal geometric object

More about fractal curves and surfaces later

Complex Arithmetic

- Complex number defined by two scalars

  $z = x + \mathbf{j}y$

  $j^2 = -1$

- Addition and Subtraction

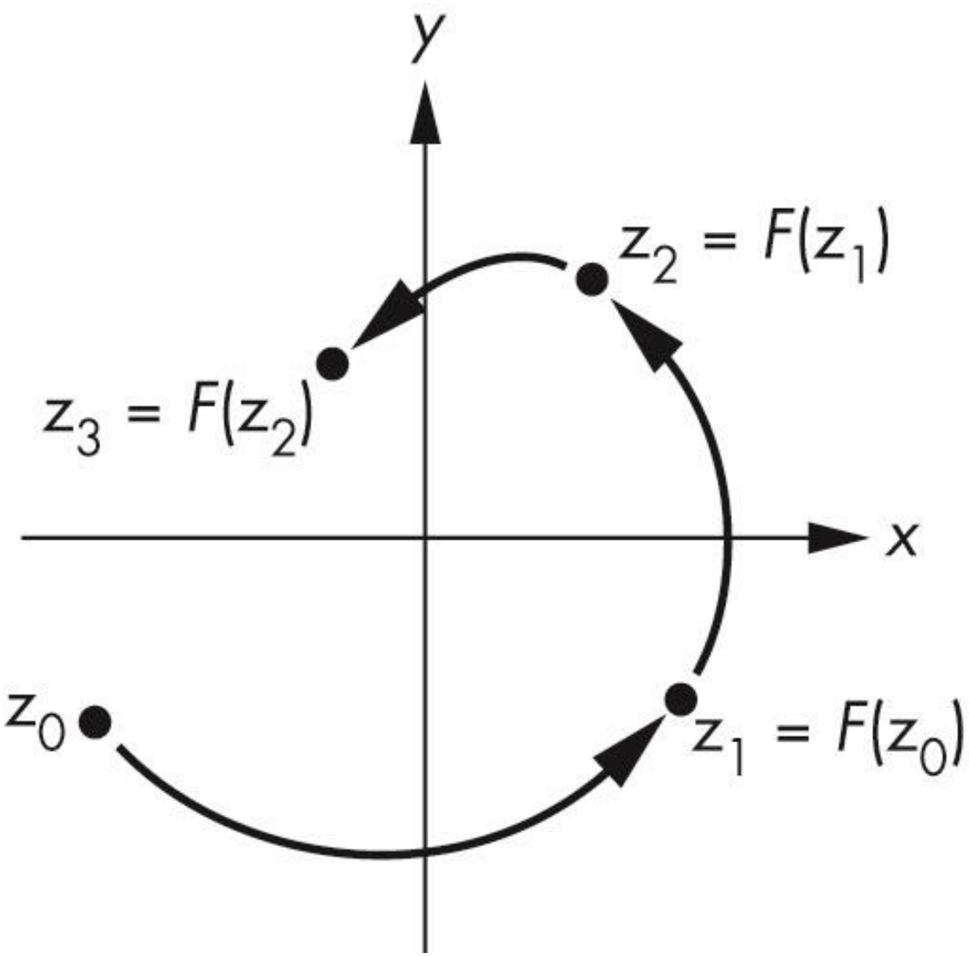
  $z_1+z_2 = x_1 + x_2 +\mathbf{j}(y_1+y_2)$

  $z_1*z_2 = x_1*x_2-y_1*y_2 + \mathbf{j}(x_1*y_2+x_2*y_1)$

- Magnitude

  $|z|^2 = x^2 + y^2$

Iteration in the Complex Plane



Mandelbrot Set

iterate on $z_{k+1}=z_k^2+c$

with $z_0 = 0 + \mathbf{j}0$

Two cases as $k \rightarrow \infty$

$|z_k| \rightarrow \infty$

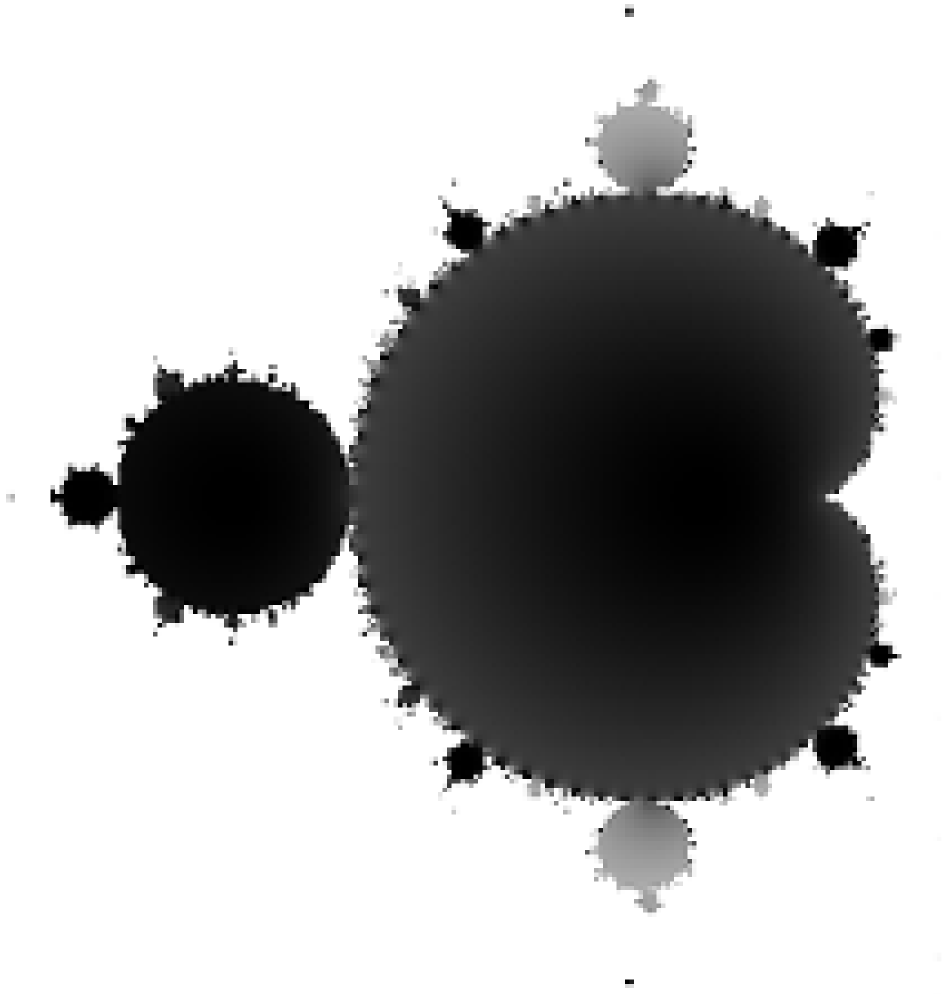
$|z_k|$ remains finite

If for a given c, $|z_k|$ remains finite, then c belongs to the Mandelbrot set

Computing the Mandelbrot Set

- Pick a rectangular region
- Map each pixel to a value in this region
- Do an iterative calculation for each pixel
  - If magnitude is greater than 2, we know sequence will diverge and point does not belong to the set
  - Stop after a fixed number of iterations
  - Points with small magnitudes should be in set
  - Color each point based on its magnitude

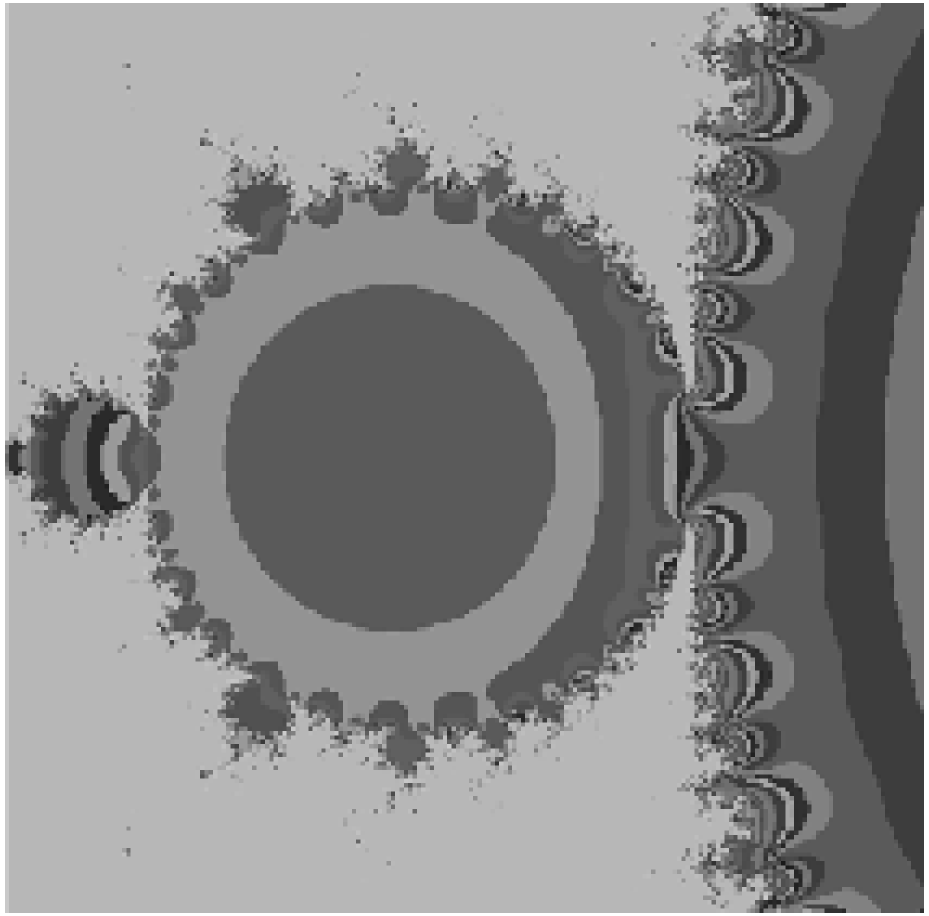
Mandelbrot Set



Exploring the Mandelbrot Set

- Most interesting parts are centered near (-0.5, 0.0)
- Really interesting parts are where we are uncertain if points are in or out of the set
- Repeated magnification these regions reveals complex and beautiful patterns
- We use color maps to enhance the detail

Mandelbrot Set



Computing in the JS File I

- Form a texture map of the set and map to a rectangle

```
var height = 0.5;
              // size of window in complex plane
var width = 0.5;var cx = -0.5;
             // center of window in complex plane
var cy = 0.5;var max = 100;
             // number of interations per point
var n = 512;
var m =512;
var texImage = new Uint8Array(4*n*m);
```

Computing in JS File II

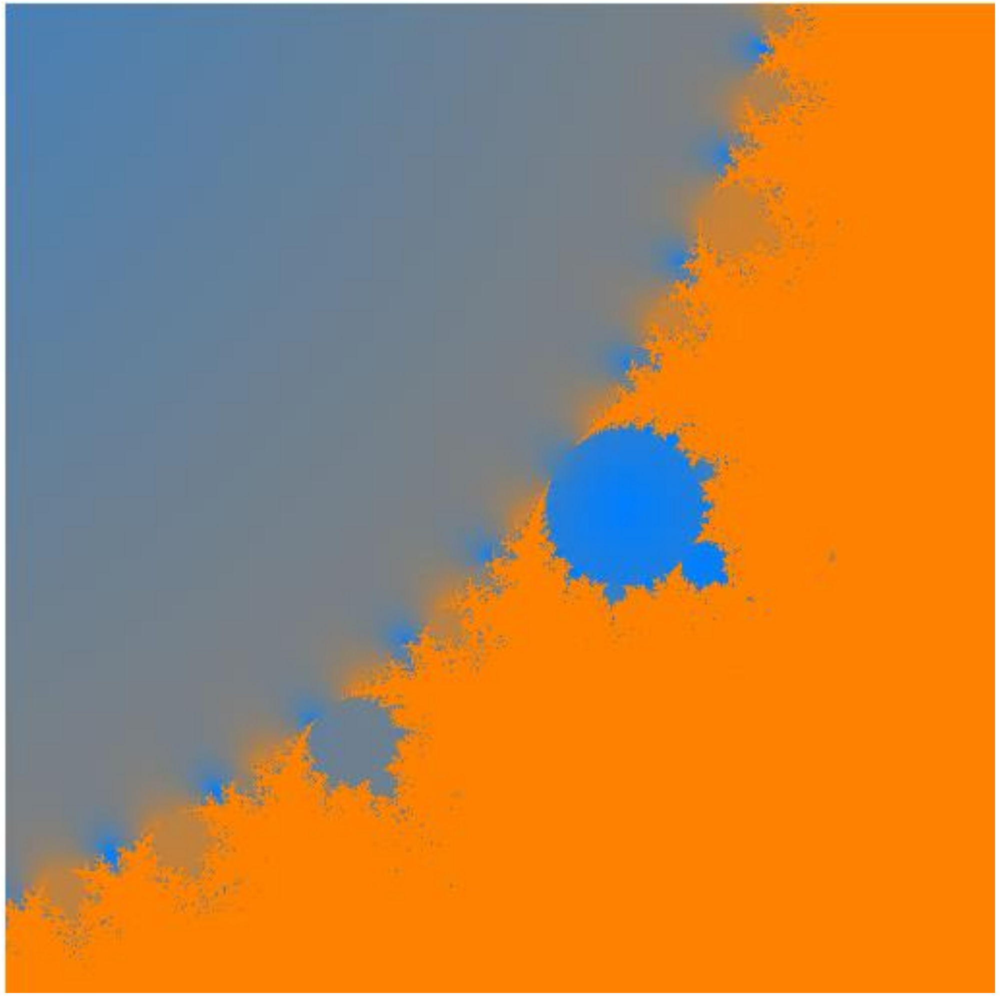
```
for ( var i = 0; i < n; i++ )
  for ( var j = 0; j < m; j++ ) {
      var x = i * ( width / (n - 1) ) + cx - width / 2;
      var y = j * ( height / ( m - 1 ) ) + cy - height / 2;
      var c = [ 0.0, 0.0 ];
      var p =  [ x, y ];

      for ( var k = 0; k < max; k++ ) {
        // compute c = c^2 + p
        c = [c[0]*c[0]-c[1]*c[1], 2*c[0]*c[1]];
        c = [c[0]+p[0],  c[1]+p[1]];
        v = c[0]*c[0]+c[1]*c[1];
        if ( v > 4.0 ) break;      /* assume not in set if mag > 2 */
      }
```

Computing in JS File III

```
// assign gray level to point based on its magnitude */
        if ( v > 1.0 ) v = 1.0;        /* clamp if > 1 */
        texImage[4*i*m+4*j] = 255*v;
        texImage[4*i*m+4*j+1] = 255*( 0.5* (Math.sin( v*Math.PI/180 ) + 1.0));
        texImage[4*i*m+4*j+2] = 255*(1.0 - v);
        texImage[4*i*m+4*j+3] = 255;
}
```

- Set up two triangles to define a rectangle
- Set up texture object with the set as data
- Render the triangles

Example



Fragment Shader

- Our first implementation is incredibly inefficient and makes no use of the power of the fragment shader
- Note the calculation is “embarrassingly parallel”
  - computation for the color of each fragment is completely independent
  - Why not have each fragment compute membership for itself?
  - Each fragment would then determine its own color

Interactive Program

- JS file sends window parameters obtained from sliders to the fragment shader as uniforms
- Only geometry is a rectangle
- No need for a texture map since shader will work on individual pixels

Fragment Shader I

```
precision mediump float;
uniform float cx;
uniform float cy;
uniform float scale;
float height;
float width;

void main() {

    const int max = 100;        /* number of iterations per point */
    const float PI = 3.14159;
    float n = 1000.0;
    float m = 1000.0;
```

Fragment Shader II

```
float v;
float x = gl_FragCoord.x  /(n*scale) + cx - 1.0 / (2.0*scale);
float y = gl_FragCoord.y/(m*scale) + cy - 1.0 / (2.0*scale);
float ax=0.0, ay=0.0;
float bx, by;
for ( int k = 0; k < max; k++ ) {
        // compute c = c^2 + p
        bx  = ax*ax-ay*ay;
        by = 2.0*ax*ay;
        ax = bx+x;
        ay = by+y;
       v = ax*ax+ay*ay;
       if ( v > 4.0 ) break;      // assume not in set if mag > 2
}
```

Fragment Shader

```
// assign gray level to point based on its magnitude //

// clamp if  > 1

        v = min(v, 1.0);
        gl_FragColor.r = v;
        gl_FragColor.g = 0.5* sin( 3.0*PI*v) + 1.0;
        gl_FragColor.b = 1.0-v;
        gl_FragColor.b = 0.5* cos( 19.0*PI*v) + 1.0;
        gl_FragColor.a = 1.0;
}
```

Analysis

- This implementation will use as many fragment processors as are available concurrently
- Note that if an iteration ends early, the GPU will use that processor to work on another fragment
- Note also the absence of loops over x and y
- Still not using the full parallelism since we are really computing a luminance image