Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

The University of New Mexico

Compositing and Blending

Ed Angel
Professor Emeritus of Computer Science
University of New Mexico

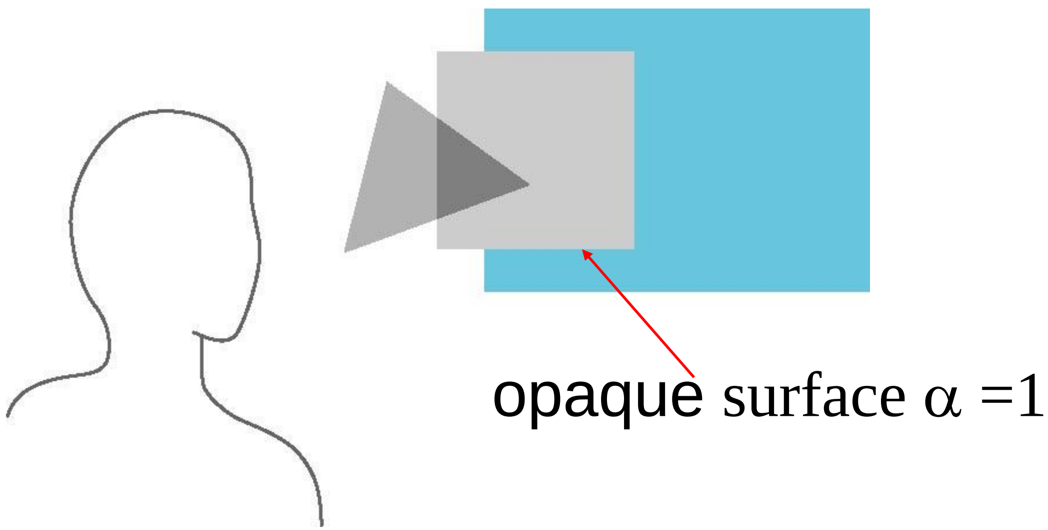
Objectives

- Learn to use the A component in RGBA color for
  - Blending for translucent surfaces
  - Compositing images
  - Antialiasing

Opacity and Transparency

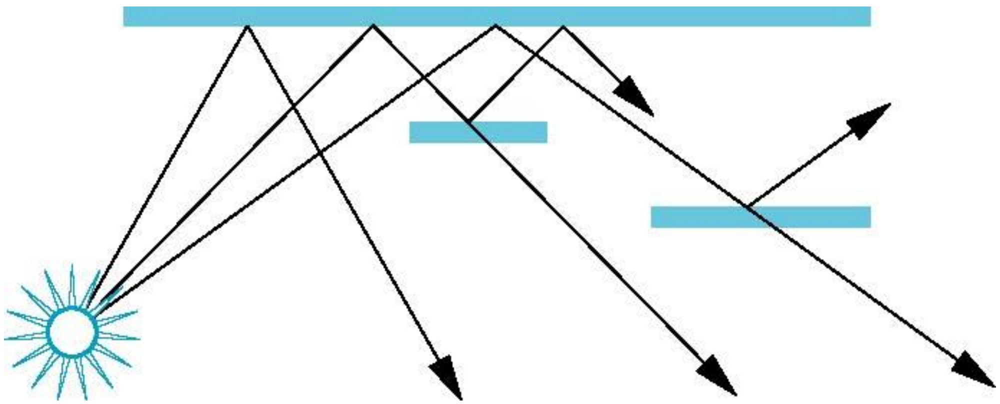
- Opaque surfaces permit no light to pass through
- Transparent surfaces permit all light to pass
- Translucent surfaces pass some light

translucency = 1 – opacity (α)



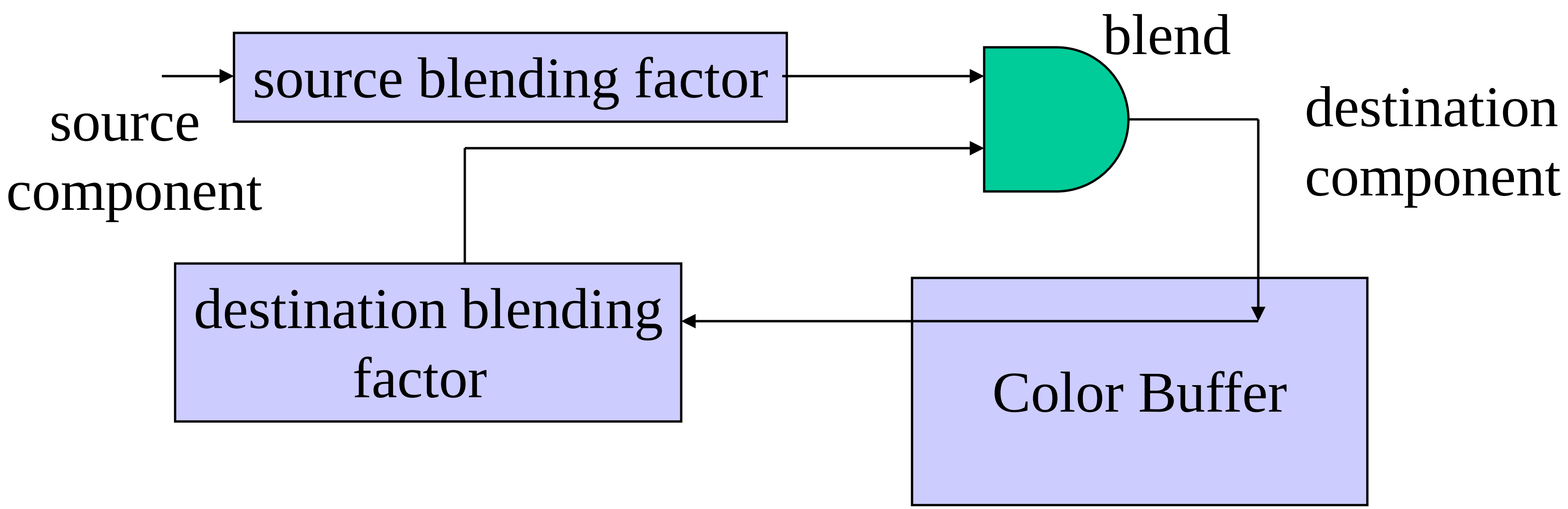
Physical Models

- Dealing with translucency in a physically correct manner is difficult due to
    - the complexity of the internal interactions of light and matter
    - Using a pipeline renderer



Writing Model

- Use A component of RGBA (or RGBα) color to store opacity
- During rendering we can expand our writing model to use RGBA values



Blending Equation

- We can define source and destination blending factors for each RGBA component

$$\mathbf{s} = [s_r, s_g, s_b, s_\alpha]$$

$$\mathbf{d} = [d_r, d_g, d_b, d_\alpha]$$

Suppose that the source and destination colors are

$$\mathbf{b} = [b_r, b_g, b_b, b_\alpha]$$

$$\mathbf{c} = [c_r, c_g, c_b, c_\alpha]$$

Blend as

$$\mathbf{c}' = [b_r\, s_r + c_r\, d_r,\ b_g\, s_g + c_g\, d_g\ ,\ b_b\, s_b + c_b\, d_b\ ,\ b_\alpha\, s_\alpha + c_\alpha\, d_\alpha\ ]$$

OpenGL Blending and Compositing

- Must enable blending and pick source and destination factors

```
gl.enable(gl.BLEND)
gl.blendFunc(source_factor,
  destination_factor)
```

- Only certain factors supported
  - `gl.ZERO, gl.ONE`
  - `gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA`
  - `gl.DST_ALPHA, gl.ONE_MINUS_DST_ALPHA`
  - See Redbook for complete list

Example

- Suppose that we start with the opaque background color $(R_0,G_0,B_0,1)$
  - This color becomes the initial destination color
- We now want to blend in a translucent polygon with color (R_1,G_1,B_1,α_1)
- Select **`GL_SRC_ALPHA`** and **`GL_ONE_MINUS_SRC_ALPHA`** as the source and destination blending factors

$$R'_1 = \alpha_1 R_1 + (1- \alpha_1) R_{0,} \ldots\ldots$$

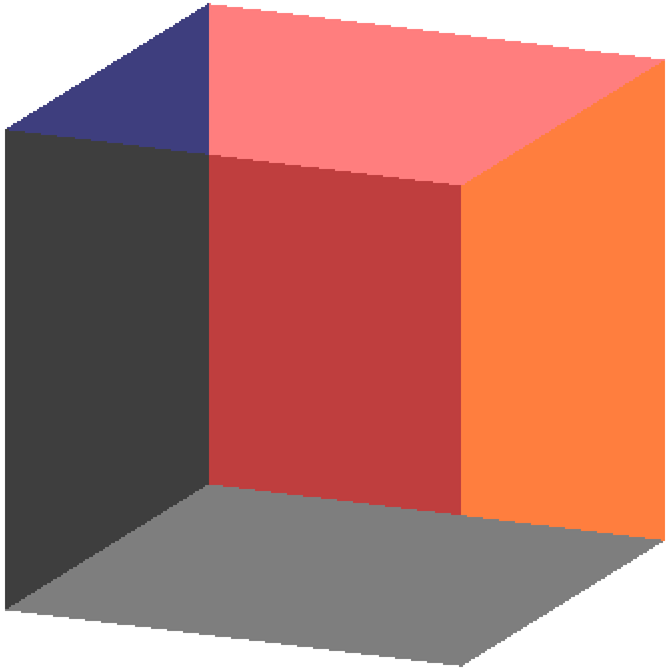
- Note this formula is correct if polygon is either opaque or transparent

Clamping and Accuracy

- All the components (RGBA) are clamped and stay in the range (0,1)
- However, in a typical system, RGBA values are only stored to 8 bits
  - Can easily loose accuracy if we add many components together
  - Example: add together n images
    - Divide all color components by n to avoid clamping
    - Blend with source factor = 1, destination factor = 1
    - But division by n loses bits

Order Dependency

- Is this image correct?
  - Probably not
  - Polygons are rendered in the order they pass down the pipeline
  - Blending functions are order dependent



The University of New Mexico

Opaque and Translucent Polygons

- Suppose that we have a group of polygons some of which are opaque and some translucent
- How do we use hidden-surface removal?
- Opaque polygons block all polygons behind them and affect the depth buffer
- Translucent polygons should not affect depth buffer
  - Render with `gl.depthMask(false)` which makes depth buffer read-only
- Sort polygons first to remove order dependency

The University of New Mexico

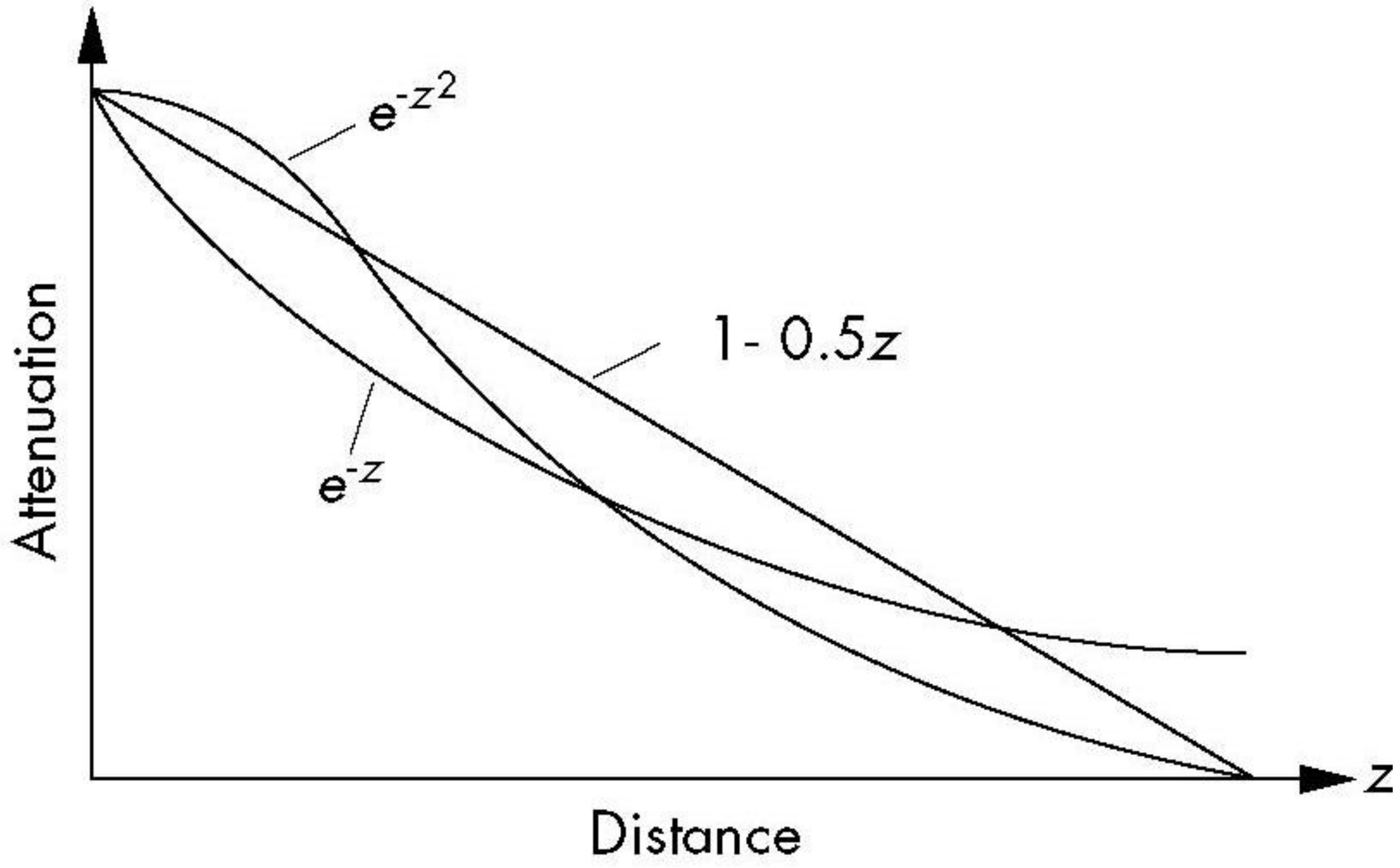
Fog

- We can composite with a fixed color and have the blending factors depend on depth
  - Simulates a fog effect
- Blend source color C_s and fog color C_f by

$$C_s' = f\,C_s + (1-f)\,C_f$$

- f is the *fog factor*
  - Exponential
  - Gaussian
  - Linear (depth cueing)
- Deprecated but can recreate

Fog Functions

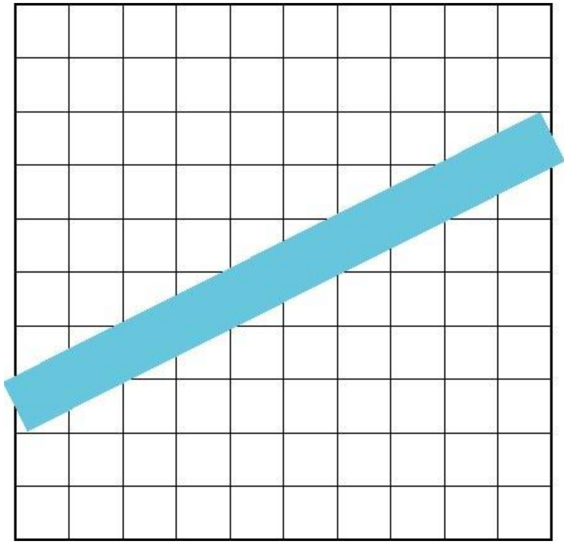


Compositing and HTML

- In desktop OpenGL, the A component has no effect unless blending is enabled
- In WebGL, an A other than 1.0 has an effect because WebGL works with the HTML5 Canvas element
- A = 0.5 will cut the RGB values by ½ when the pixel is displayed
- Allows other applications to be blended into the canvas along with the graphics

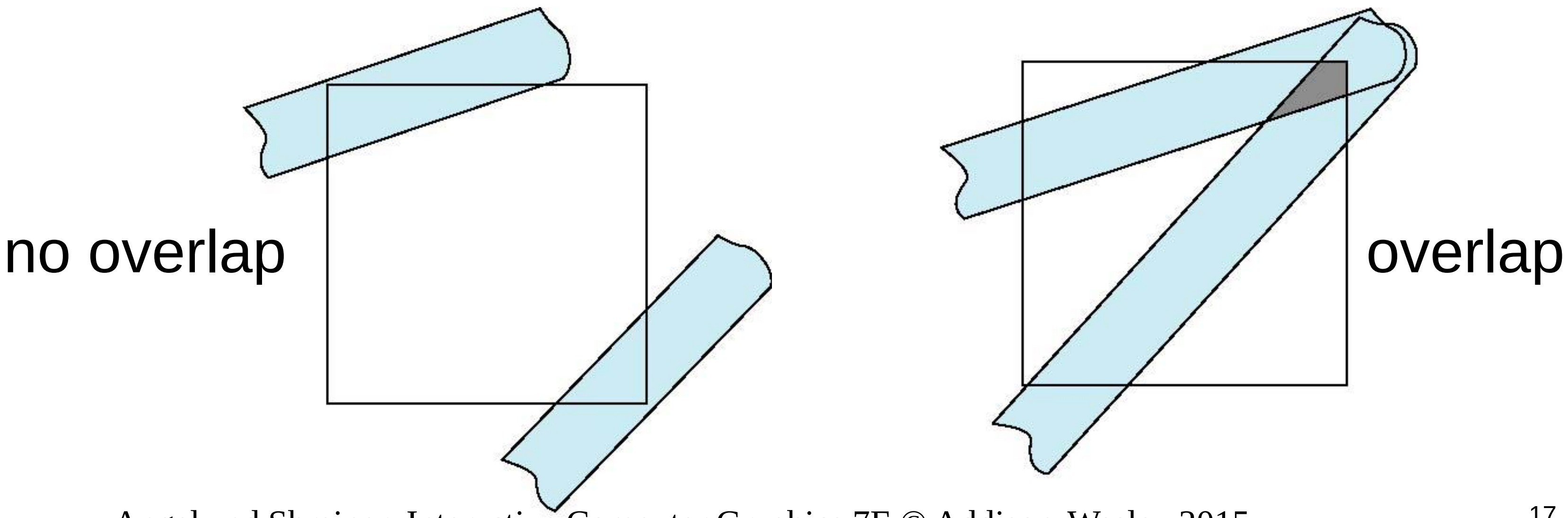
Line Aliasing

- Ideal raster line is one pixel wide
- All line segments, other than vertical and horizontal segments, partially cover pixels
- Simple algorithms color only whole pixels
- Lead to the "jaggies" or aliasing
- Similar issue for polygons



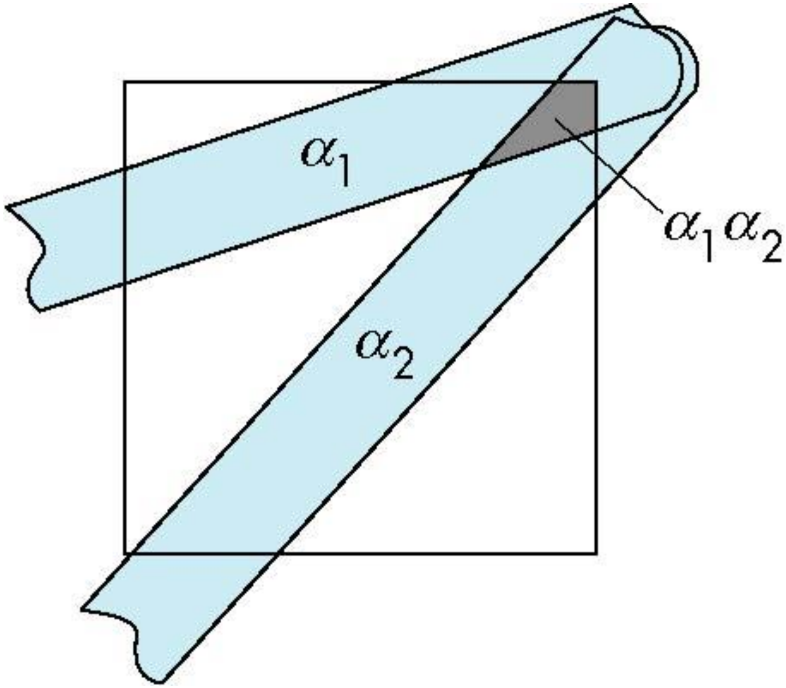
Antialiasing

- Can try to color a pixel by adding a fraction of its color to the frame buffer
  - Fraction depends on percentage of pixel covered by fragment
  - Fraction depends on whether there is overlap



Area Averaging

- Use average area $\alpha_1+\alpha_2-\alpha_1\alpha_2$ as blending factor



OpenGL Antialiasing

- Not (yet) supported in WebGL
- Can enable separately for points, lines, or polygons

```
glEnable(GL_POINT_SMOOTH);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);

glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

- Note most hardware will automatically antialias

Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

The University of New Mexico

Imaging Applications

Ed Angel

Professor Emeritus of Computer Science

University of New Mexico

Objectives

- Use the fragment shader to do image processing
  - Image filtering
  - Pseudo Color
- Use multiple textures
  - matrix operations
- Introduce GPGPU

Accumulation Techniques

- Compositing and blending are limited by resolution of the frame buffer
  - Typically 8 bits per color component
- The *accumulation buffer* was a high resolution buffer (16 or more bits per component) that avoided this problem
- Could write into it or read from it with a scale factor
- Slower than direct compositing into the frame buffer
- Now deprecated but can do techniques with floating point frame buffers

Multirendering

- Composite multiple images
- Image Filtering (convolution)
  - add shifted and scaled versions of an image
- Whole scene antialiasing
  - move primitives a little for each render
- Depth of Field
  - move viewer a little for each render keeping one plane unchanged
- Motion effects

Fragment Shaders and Images

- Suppose that we send a rectangle (two triangles) to the vertex shader and render it with an n x m texture map
- Suppose that in addition we use an n x m canvas
- There is now a one-to-one correspondence between each texel and each fragment
- Hence we can regard fragment operations as imaging operations on the texture map

GPGPU

- Looking back at these examples, we can note that the only purpose of the geometry is to trigger the execution of the imaging operations in the fragment shader
- Consequently, we can look at what we have done as large matrix operations rather than graphics operations
- Leads to the field of General Purpose Computing with a GPU (GPGPU)

Examples

- Add two matrices
- Multiply two matrices
- Fast Fourier Transform
- Uses speed and parallelism of GPU
- But how do we get out results?
  - Floating point frame buffers
  - OpenCL (WebCL)
  - Compute shaders

Using Multiple Texels

- Suppose we have a 1024 x 1024 texture in the texture object "image"

sampler2D(image, vec2(x,y)) returns the the value of the texture at (x,y)

sampler2D(image, vec2(x+1.0/1024.0), y); returns the value of the texel to the right of (x,y)

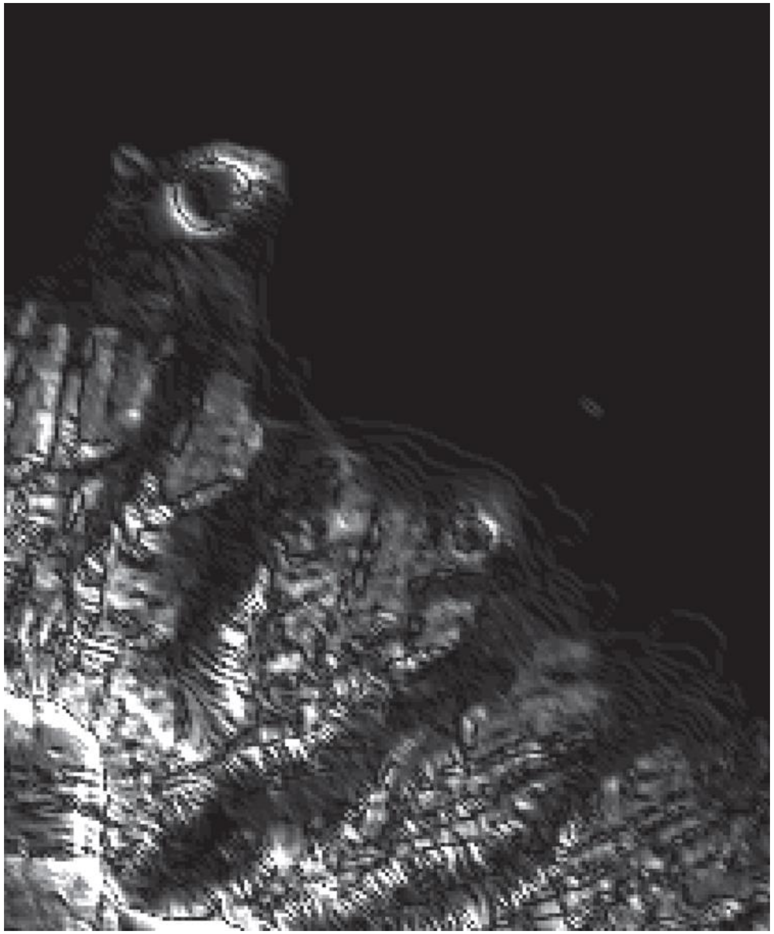
We can use any combination of texels surrounding (x, y) in the fragment shader

Image Enhancer

```
precision mediump float;
varying vec2 fTexCoord;
uniform sampler2D texture;
void main()
{
  float d = 1.0/256.0;  //spacing between texels
  float x = fTexCoord.x;
  float y = fTexCoord.y;

  gl_FragColor = 10.0*abs( texture2D( texture, vec2(x+d, y))
      - texture2D( texture, vec2(x-d, y)))
       +10.0*abs( texture2D( texture, vec2(x, y+d))
      - texture2D( texture, vec2(x, y-d)));
     gl_FragColor.w = 1.0;
}
```

Honolulu Image

original



enhanced

Sobel Edge Detector

- Nonlinear
- Find approximate gradient at each point
- Compute smoothed finite difference approximations to x and y components separately
- Display magnitude of approximate gradient
- Simple with fragment shader

Sobel Edge Detector

```
vec4 gx = 3.0*texture2D( texture, vec2(x+d, y))
        + texture2D( texture, vec2(x+d, y+d))
        + texture2D( texture, vec2(x+d, y-d))
        - 3.0*texture2D( texture, vec2(x-d, y))
        - texture2D( texture, vec2(x-d, y+d))
        - texture2D( texture, vec2(x-d, y-d));
vec4 gy = 3.0*texture2D( texture, vec2(x, y+d))
        + texture2D( texture, vec2(x+d, y+d))
        + texture2D( texture, vec2(x-d, y+d))
        - 3.0*texture2D( texture, vec2(x, y-d))
        - texture2D( texture, vec2(x+d, y-d))
        - texture2D( texture, vec2(x-d, y-d));
gl_FragColor = vec4(sqrt(gx*gx + gy*gy), 1.0);
gl_FragColor.w = 1.0;
```

Sobel Edge Detector

Using Multiple Textures

- Example: matrix addition
- Create two samplers, texture1 and texture2, that contain the data
- In fragment shader

gl_FragColor =

sampler2D(texture1, vec2(x, y))
+sampler2D(texture2, vec2(x,y));

Using 4 Way Parallelism

- Recent GPUs and graphics cards support textures up to 8K x 8K
- For scalar imaging, we can do twice as well using all four color components

$$\begin{bmatrix} R & G \\ B & A \end{bmatrix}$$

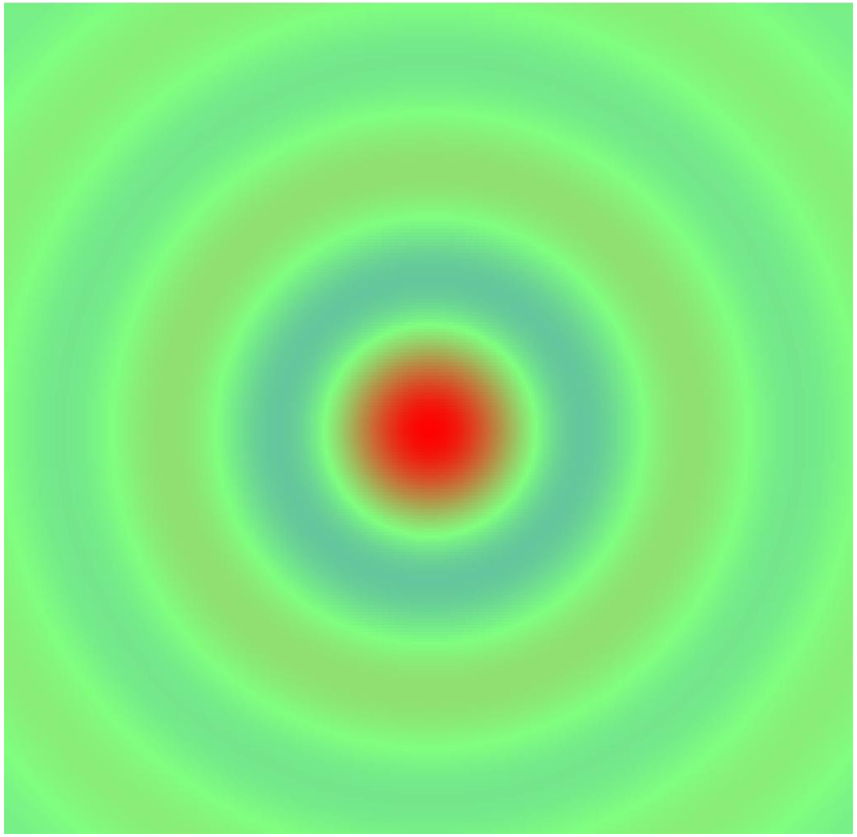
The University of New Mexico

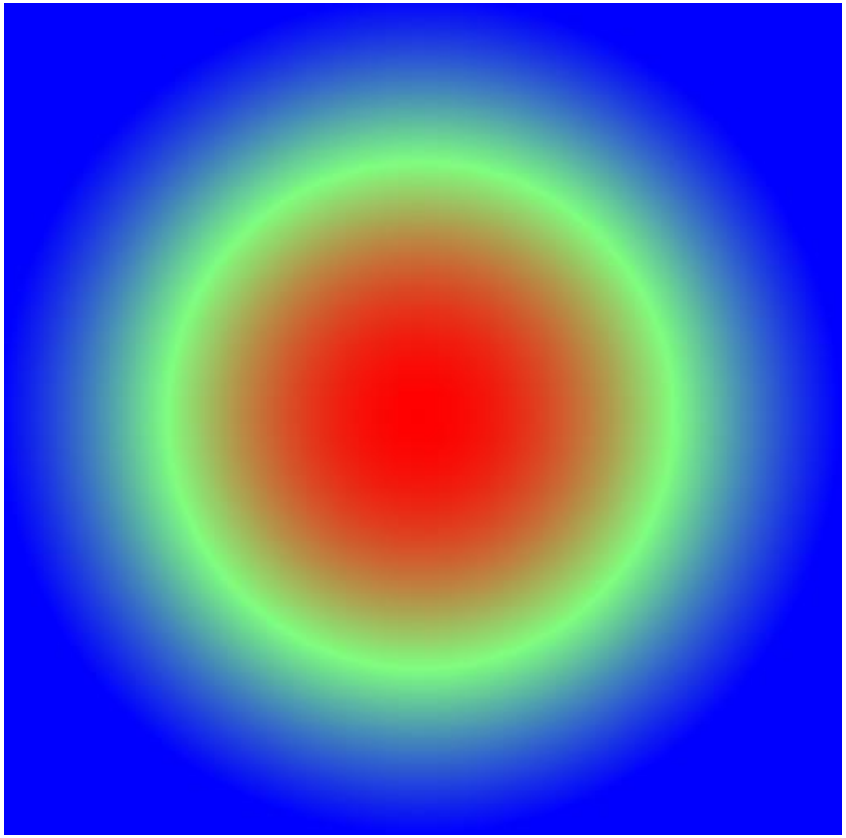
Indexed and Pseudo Color

- Display luminance (2D) image as texture map
- Treat pixel value as independent variable for separate functions for each color component

```
void main(){
    vec4 color = texture2D(texture, fTexCoord);
    if(color.g<0.5) color.g = 2.0*color.g;
        else color.g = 2.0 - 2.0*color.g;
    color.b = 1.0-color.b;
    gl_FragColor = color;
}
```

Top View of 2D Sinc





The Next Step

- Need more storage for most GPGPU calculations
- Example: filtering
- Example: iteration
- Need shared memory
- Solution: Use texture memory and off-screen rendering

Introduction to Computer Graphics with WebGL

Ed Angel

Professor Emeritus of Computer Science

Founding Director, Arts, Research, Technology and Science Laboratory

University of New Mexico

Computing the Mandelbrot Set

Ed Angel

Professor Emeritus of Computer Science

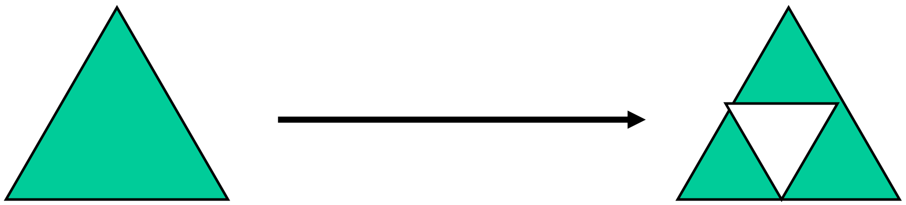
University of New Mexico

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

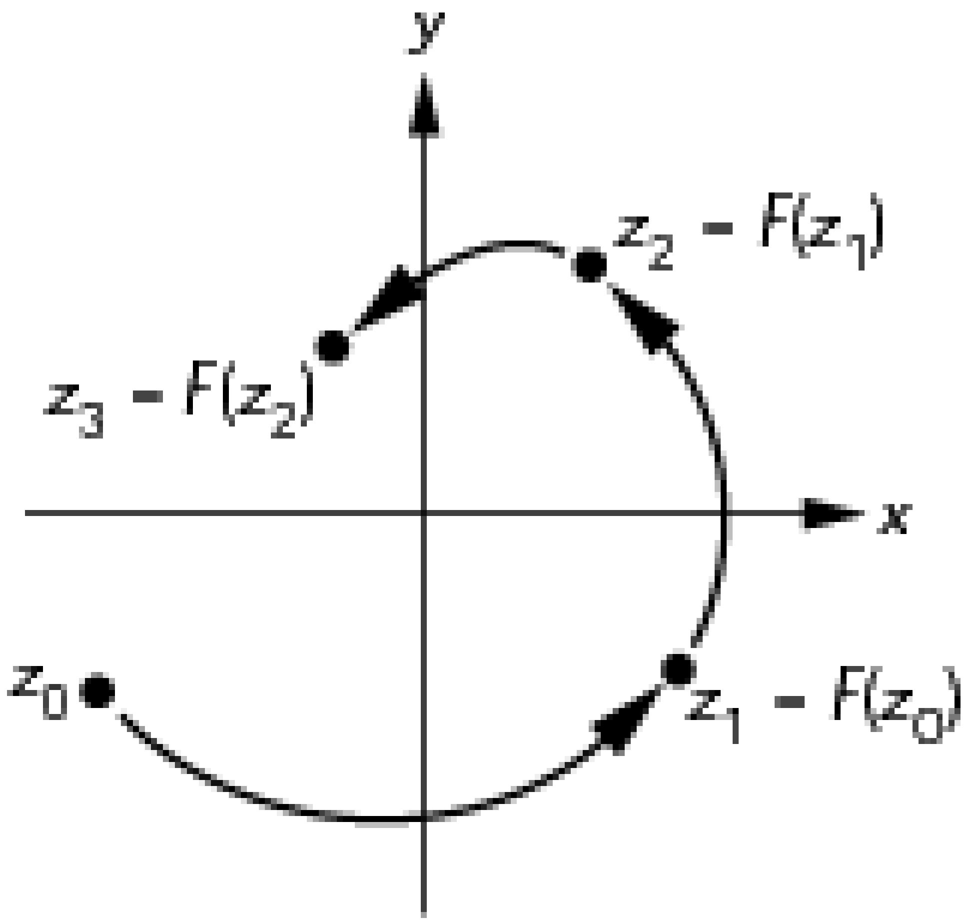
  $z_1+z_2 = x_1 + x_2 +\mathbf{j}(y_1+y_2)$

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

- 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 $k \rightarrow \infty$

$|z_k| \rightarrow \infty$

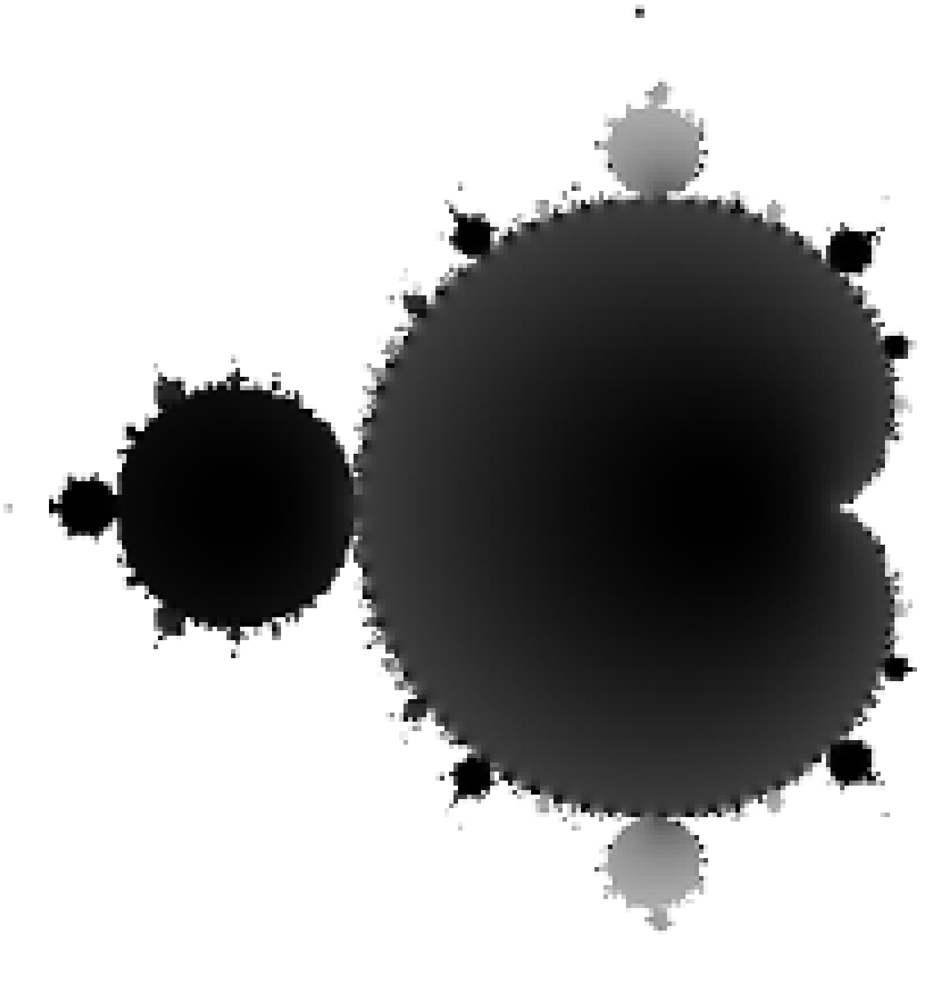
$|z_k|$ remains finite

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

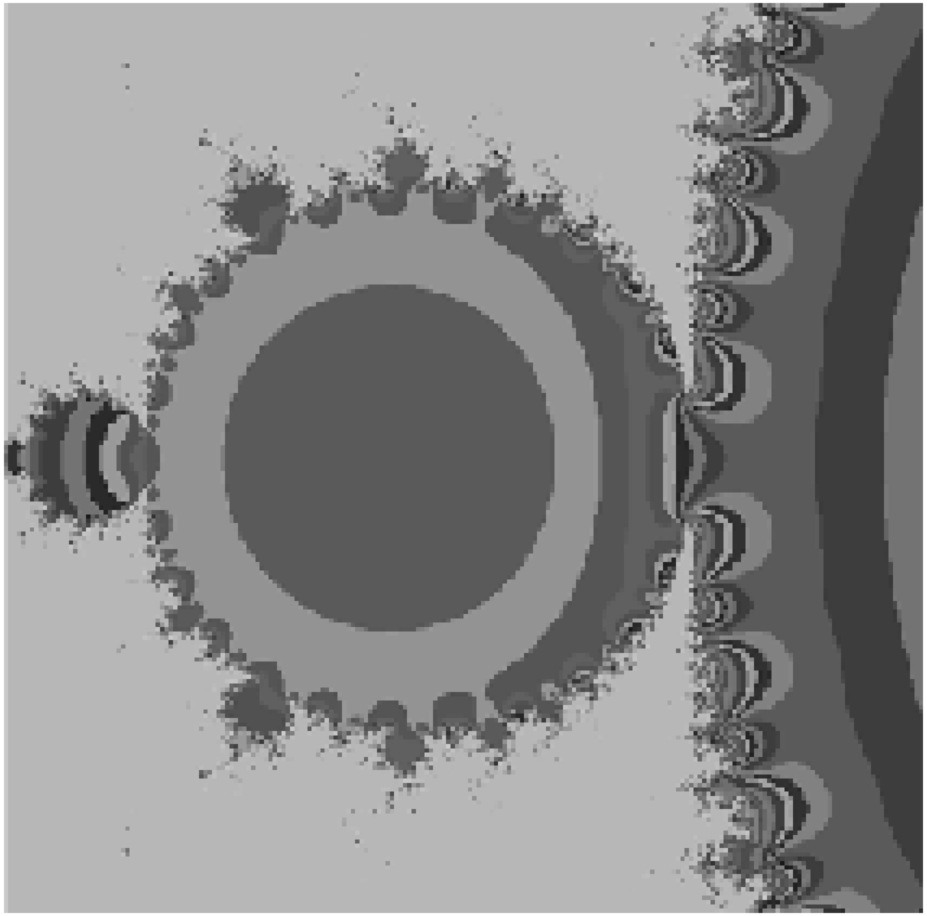
Mandelbrot Set



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);
```

The University of New Mexico

Computing in JS File II

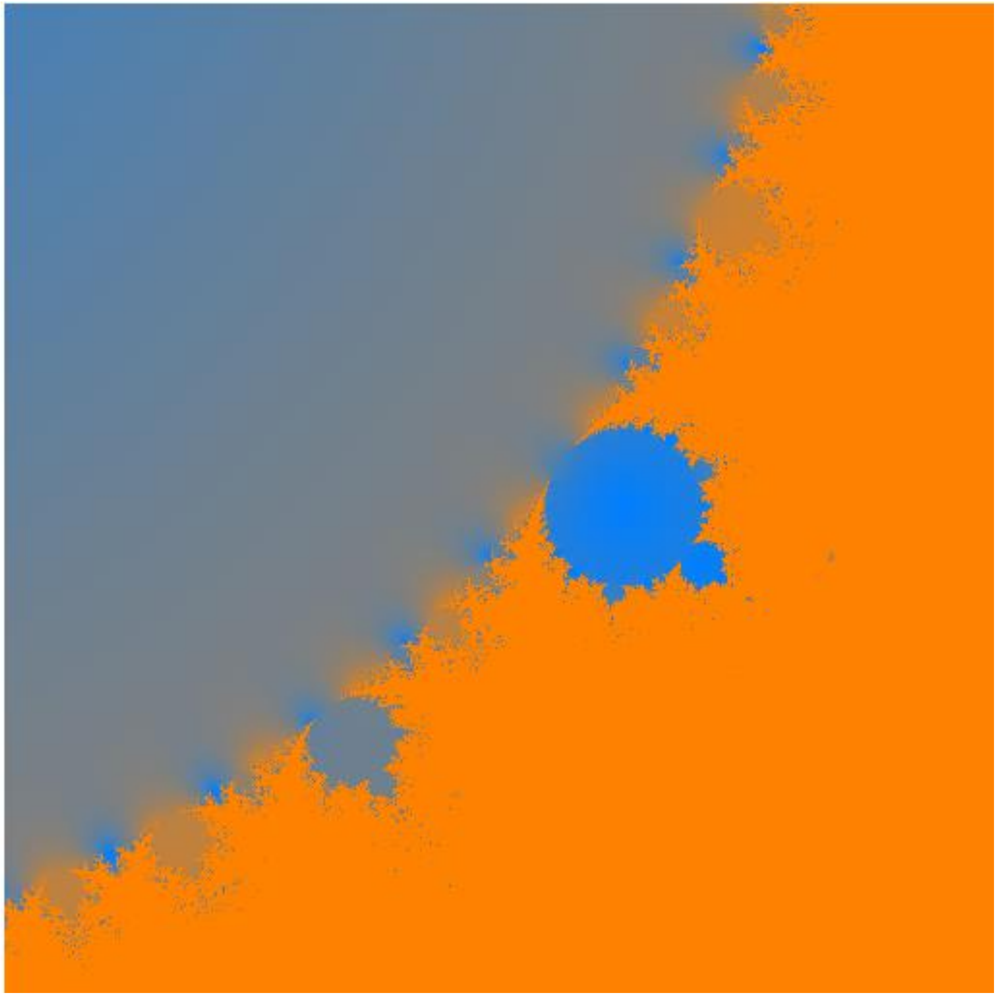
```
for ( var i = 0; i < n; i++ )
        for ( var j = 0; j < m; j++ ) {
            var x = i * ( width / (n - 1) ) + cx - width / 2;
            var y = j * ( height / ( m - 1 ) ) + cy - height / 2;
    var c = [ 0.0, 0.0 ];
    var p =  [ x, y ];

   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