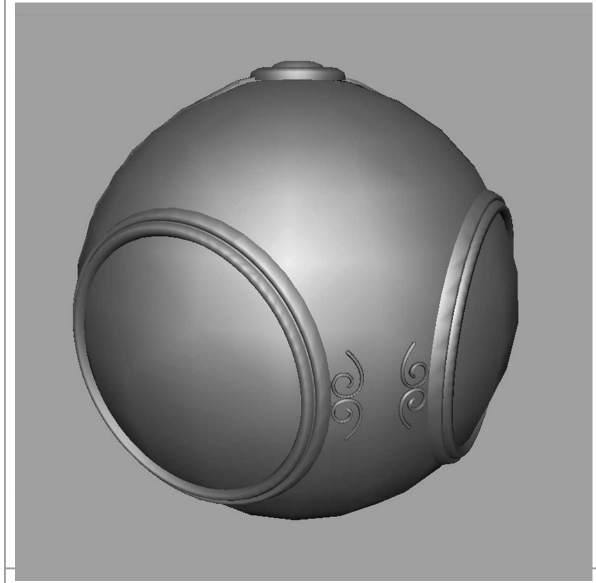
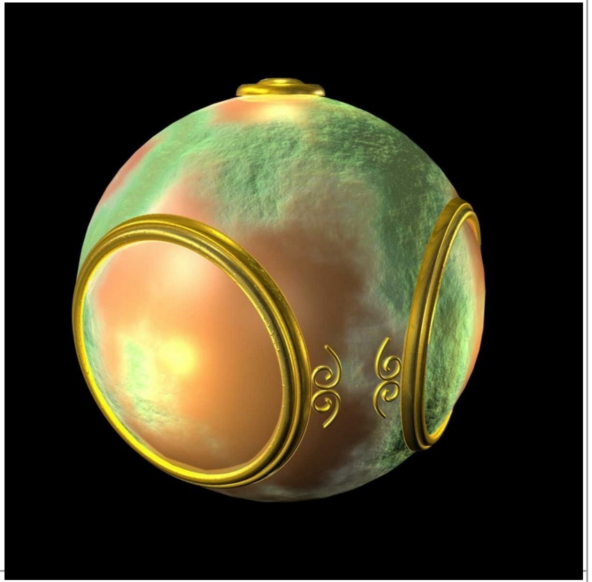
Reflection and Environment Maps

Sorin Babii
sorin.babii@cs.upt.ro

Objectives

- Texture Mapping Applications
- Reflection (Environment) Maps
    - Cube Maps
    - Spherical Maps
- Bump Maps

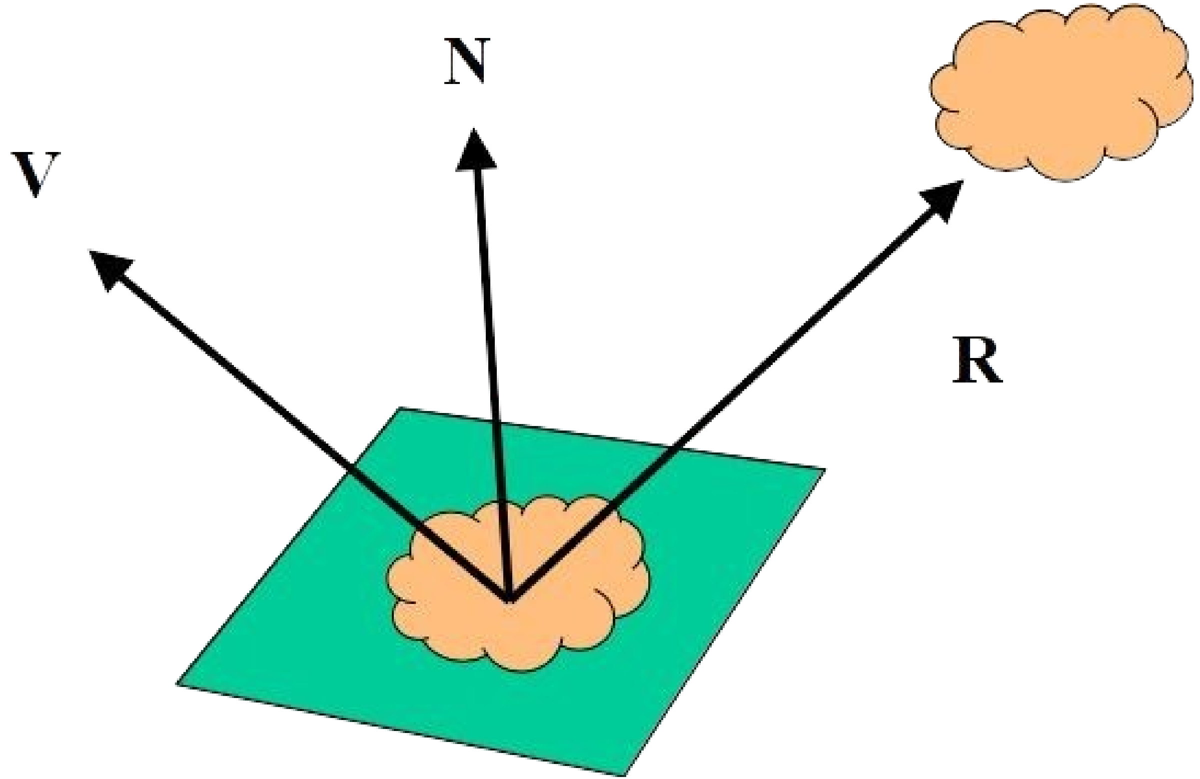
Mapping Variations

		
smooth shading	environment mapping	bump mapping

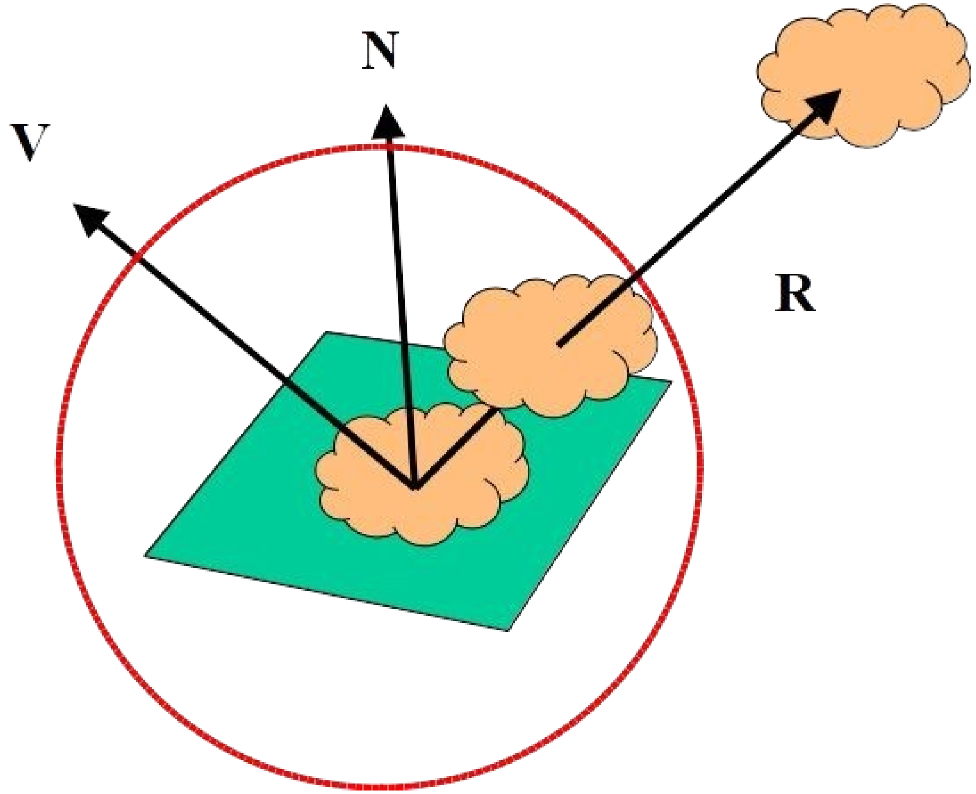
Environment Mapping

- Environmental (reflection) mapping is the way to create the appearance of highly reflective surfaces without ray tracing, which requires global calculations
- Introduced in movies such as The Abyss and Terminator 2
- Prevalent in video games
- It is a form of texture mapping

Reflecting the Environment



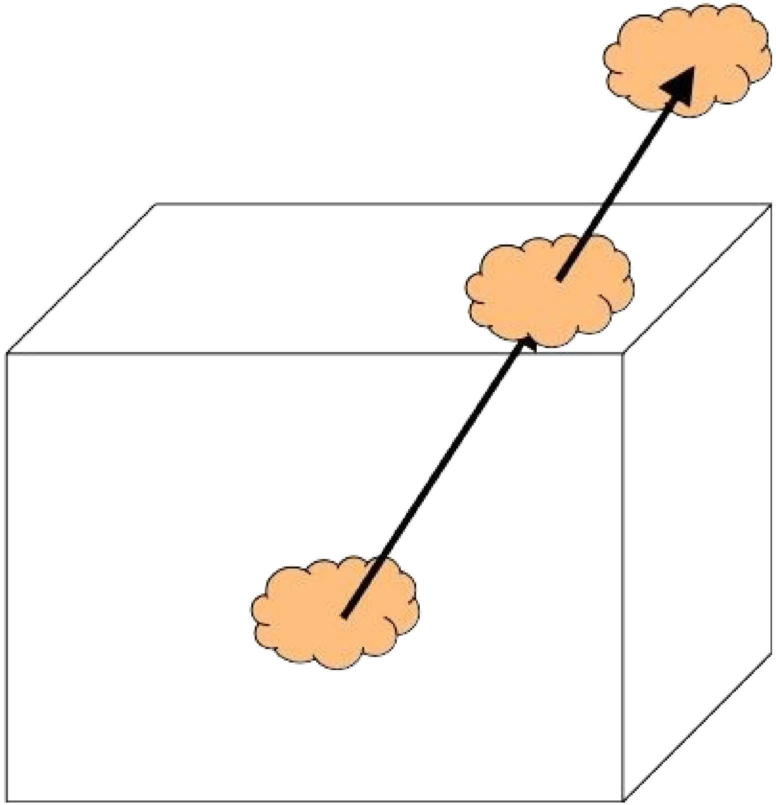
Mapping to a Sphere



Hemisphere Map as a Texture

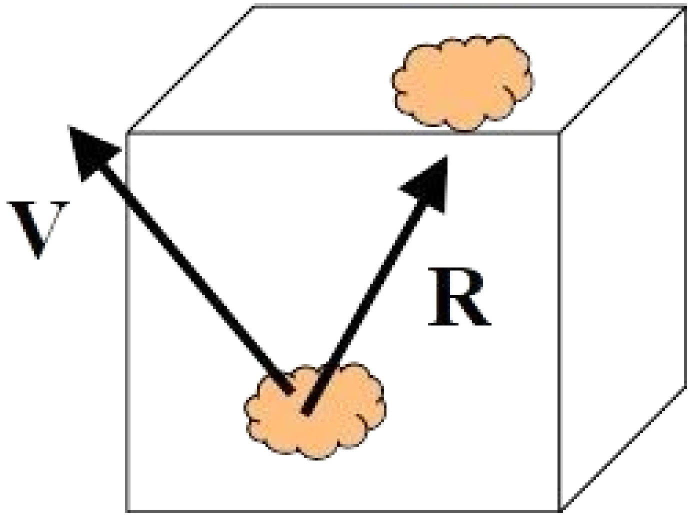
- If we map all objects to hemisphere, we cannot tell if they are on the sphere or anywhere else along the reflector
- Use the map on the sphere as a texture that can be mapped onto the object
- Can use other surfaces as the intermediate
    - Cube maps
    - Cylinder maps

Cube Map



Indexing into Cube Map

- Compute R = 2(N·V)N-V
- Object at origin
- Use largest magnitude component of R to determine face of cube
- Other two components give texture coordinates



OpenGL Implementation

- WebGL supports only cube maps
  - desktop OpenGL also supports sphere maps
- First must form map
  - Use images from a real camera
  - Form images with WebGL
- Texture map it to object

Cube Maps

- We can form a cube map texture by defining six 2D texture maps that correspond to the sides of a box
- Supported by WebGL through cubemap sampler

```
vec4 texColor = textureCube(mycube, texcoord);
```

- Texture coordinates must be 3D
  - usually are given by the vertex location ⇒ we don’t need to compute separate tex coords

Environment Maps with Shaders

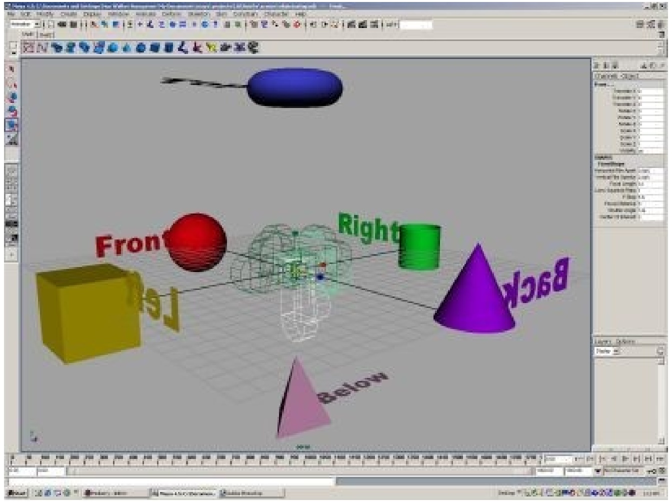
- Environment maps are usually computed in world coordinates which can differ from object coordinates because of the modeling matrix
  - May have to keep track of modeling matrix and pass it to the shaders as a uniform variable
- Can also use reflection map or refraction map for effects such as simulating water

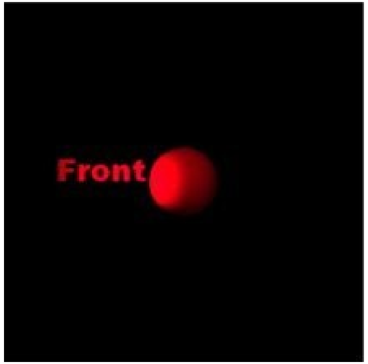
Issues

- Must assume environment is very far from object (equivalent to the difference between near and distant lights)
- Object cannot be concave (no self reflections possible)
- No reflections between objects
- Need a reflection map for each object
- Need a new map if viewer moves...

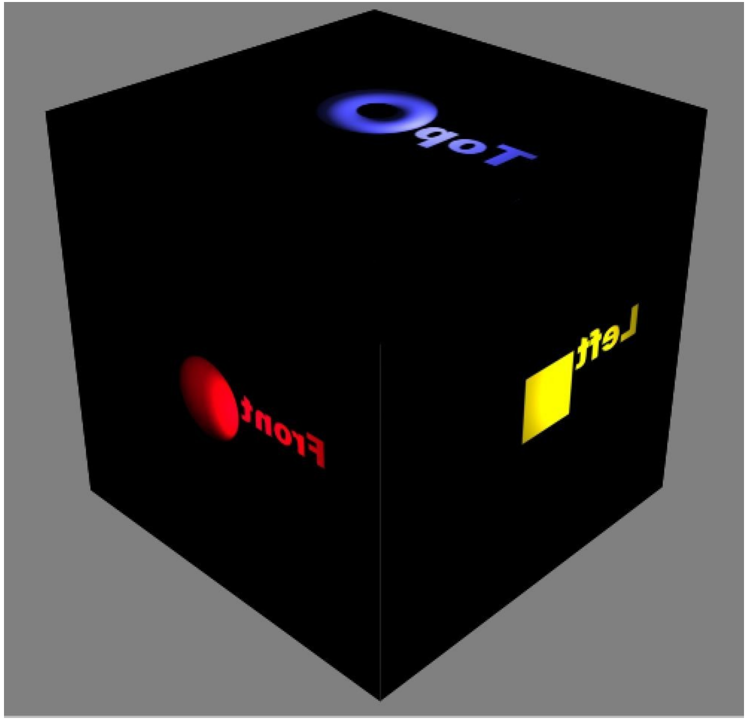
Building Cube Map...

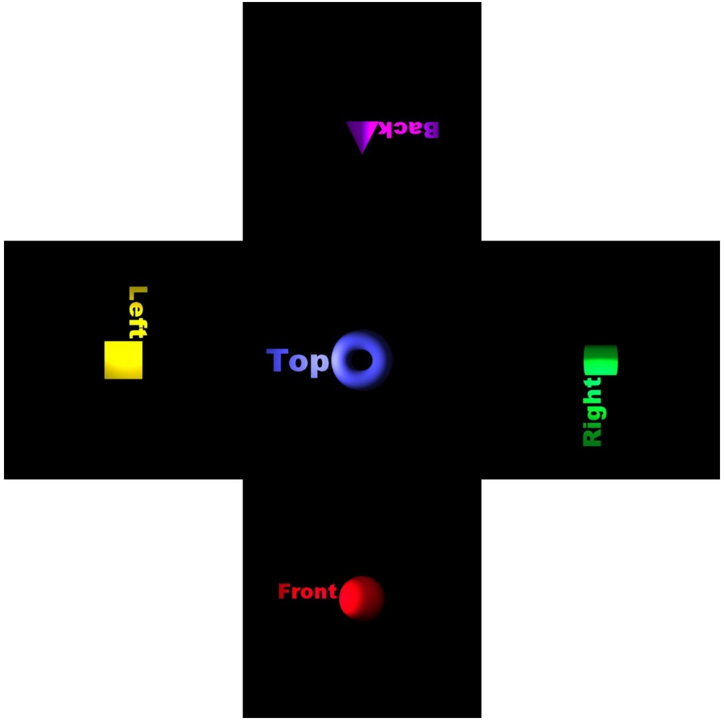
Use six cameras, each with a 90 degree angle of view





… and the Cube Image





WebGL style

```
gl.textureMap2D(
    gl.TEXTURE_CUBE_MAP_POSITIVE_X,
    level, rows, columns, border, gl.RGBA,
    gl.UNSIGNED_BYTE, image1)
```

- Same for other five images
- Make one texture object out of the six images

Example

- Consider rotating cube inside a cube that reflects the color of the walls
- Each wall is a solid color (red, green, blue, cyan, magenta, yellow)
  - Each face of room can be a texture of one texel

```
var red = new Uint8Array([255, 0, 0, 255]);
var green = new Uint8Array([0, 255, 0, 255]);
var blue = new Uint8Array([0, 0, 255, 255]);
var cyan = new Uint8Array([0, 255, 255, 255]);
var magenta = new Uint8Array([255, 0, 255, 255]);
var yellow = new Uint8Array([255, 255, 0, 255]);
```

Texture Object

```
cubeMap = gl.createTexture();
gl.bindTexture(gl.TEXTURE_CUBE_MAP, cubeMap);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_X, 0, gl.RGBA,
    1, 1, 0, gl.RGBA,gl.UNSIGNED_BYTE, red);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_X, 0, gl.RGBA,
    1, 1, 0, gl.RGBA,gl.UNSIGNED_BYTE, green);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_Y, 0, gl.RGBA,
    1, 1, 0, gl.RGBA,gl.UNSIGNED_BYTE, blue);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, gl.RGBA,
    1, 1, 0, gl.RGBA,gl.UNSIGNED_BYTE, cyan);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_POSITIVE_Z, 0, gl.RGBA,
    1, 1, 0, gl.RGBA,gl.UNSIGNED_BYTE, yellow);
gl.texImage2D(gl.TEXTURE_CUBE_MAP_NEGATIVE_Z, 0, gl.RGBA,
    1, 1, 0, gl.RGBA,gl.UNSIGNED_BYTE, magenta);
gl.activeTexture( gl.TEXTURE0 );
gl.uniform1i(gl.getUniformLocation(program, "texMap"),0);
```

Vertex Shader

```
varying vec3 R;
attribute vec4 vPosition;
attribute vec4 vNormal;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;
uniform vec3 theta;
void main(){
    vec3 angles = radians( theta );
    // compute rotation matrices rx, ry, rz here
    mat4 ModelViewMatrix = modelViewMatrix*rz*ry*rx;
    gl_Position = projectionMatrix*ModelViewMatrix*vPosition;
    vec4 eyePos = ModelViewMatrix*vPosition;
    vec4 N = ModelViewMatrix*vNormal;
    R = reflect(eyePos.xyz, N.xyz);   }
```

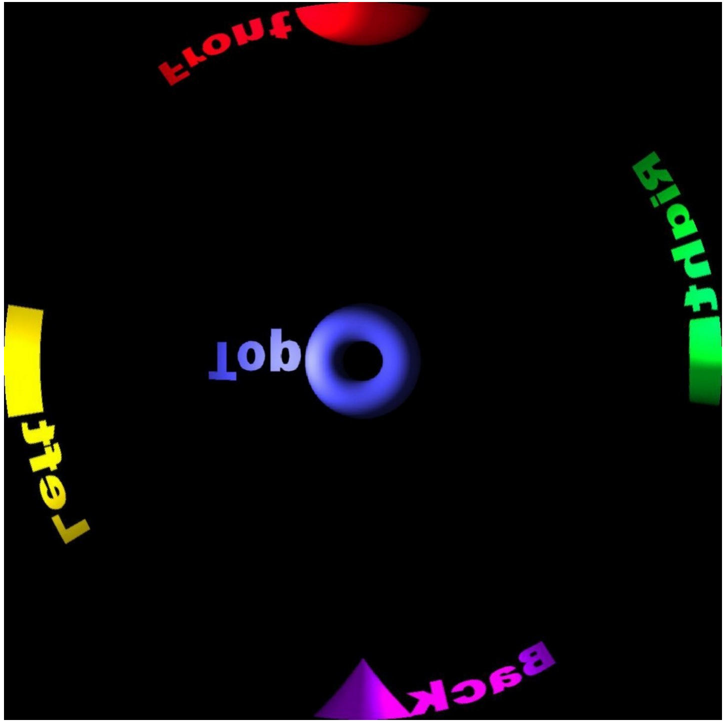
Fragment Shader

```
precision mediump float;

varying vec3 R;
uniform samplerCube texMap;

void main()
{
    vec4 texColor = textureCube(texMap, R);
    gl_FragColor = texColor;
}
```

Sphere Mapping

- Original environmental mapping technique proposed by Blinn and Newell based in using lines of longitude and latitude to map parametric variables to texture coordinates
- OpenGL supports sphere mapping which requires a circular texture map equivalent to an image taken with a fisheye lens

Sphere Map



Video

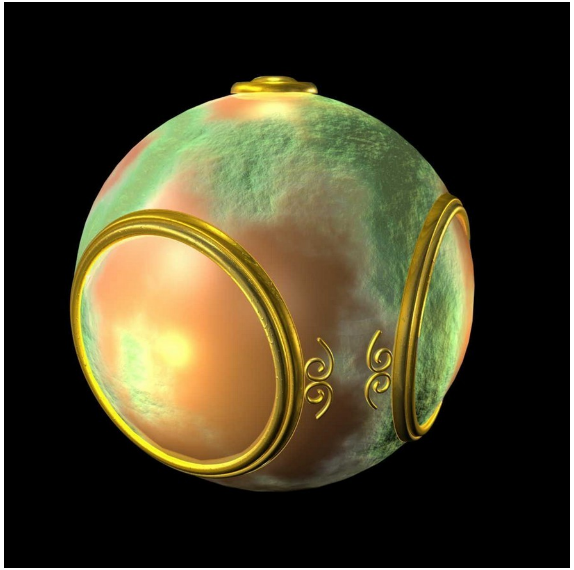
http://staff.cs.upt.ro/~sorin/webgl/Code/w07/reflectionMap.html

Bump Maps

Sorin Babii
sorin.babii@cs.upt.ro

Objectives

- Introduce bump mapping

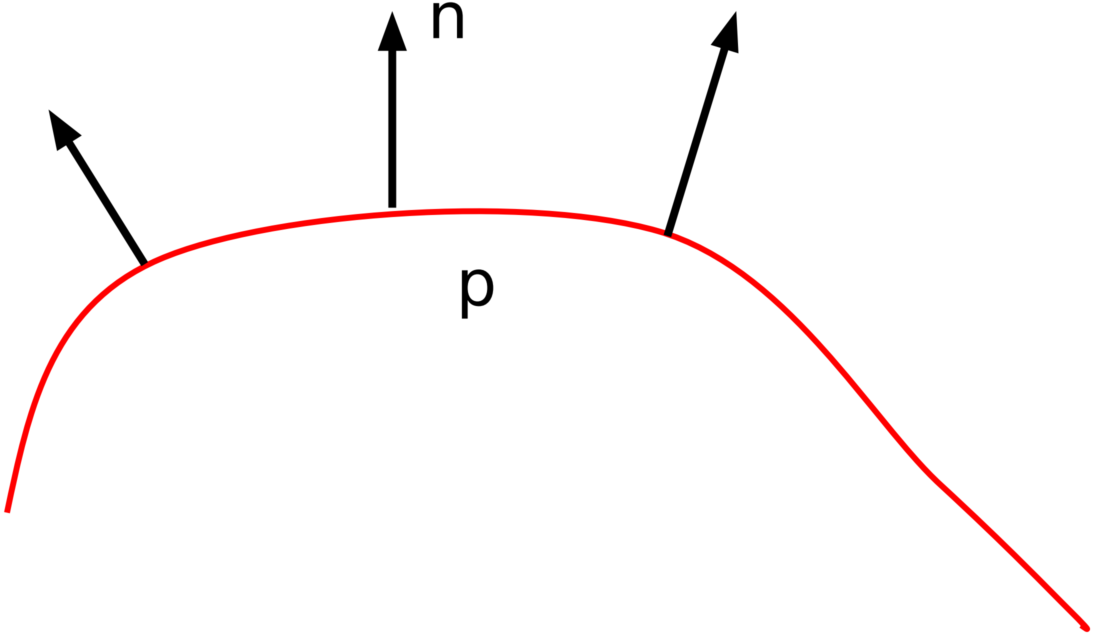


Modeling an Orange

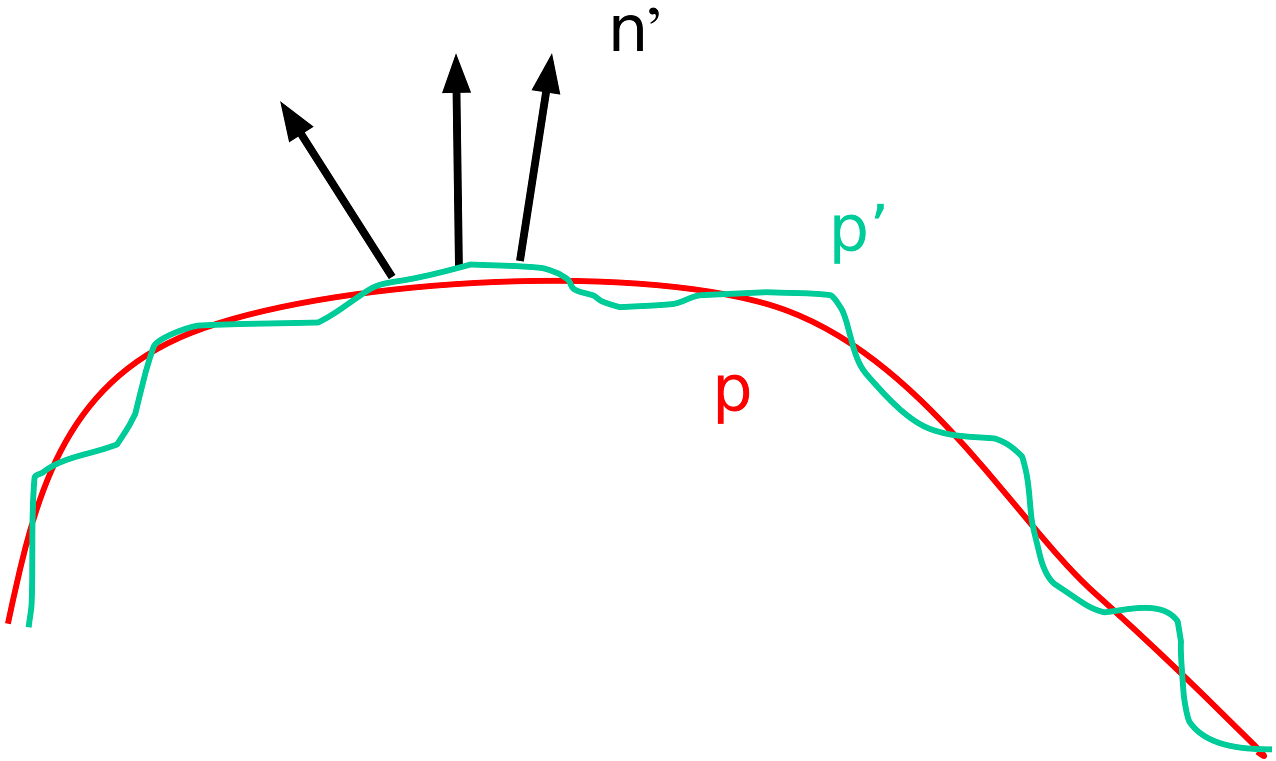
- Consider modeling an orange
- Texture map a photo of an orange onto a surface
  - Captures dimples
  - Will not be correct if we move viewer or light
  - We have shades of dimples rather than their correct orientation
- Ideally we need to perturb normal across surface of object and compute a new color at each interior point

Bump Mapping (Blinn)

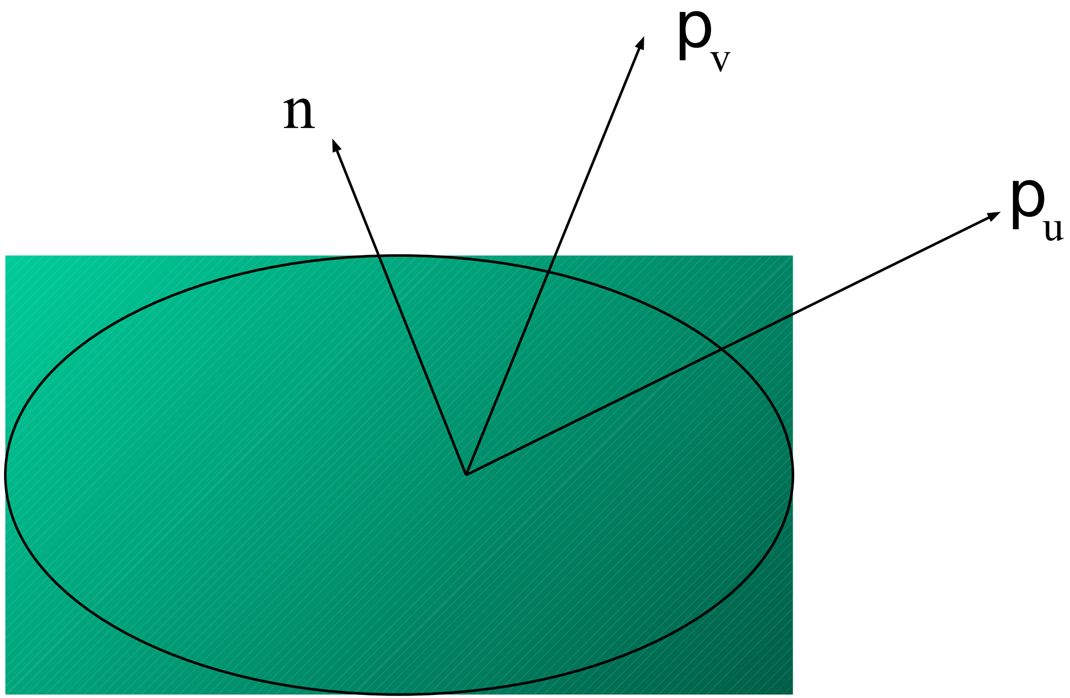
Consider a smooth surface



Rougher Version



Tangent Plane



Equations

$$\mathrm{p}(u,v) = [x(u,v),\ y(u,v),\ z(u,v)]^T$$

$$\mathrm{p}_u = [\ \partial x/\ \partial u,\ \partial y/\ \partial u,\ \partial z/\ \partial u]^T$$

$$\mathrm{p}_v = [\ \partial x/\ \partial v,\ \partial y/\ \partial v,\ \partial z/\ \partial v]^T$$

$$\mathrm{n} = (\mathrm{p}_u \times \mathrm{p}_v)\ /\ |\mathrm{p}_u \times \mathrm{p}_v|$$

Displacement Function

$$p' = p + d(u,v)\, n$$

$d(u,v)$ is the bump or displacement function

$$|d(u,v)| \ll 1$$

Perturbed Normal

$$n' = p'_u \times p'_v$$

$$p'_u = p_u + (\partial d/\partial u)n + d(u,v)n_u$$

$$p'_v = p_v + (\partial d/\partial v)n + d(u,v)n_v$$

If d is small, we can neglect last term

Approximating the Normal

$$n' = p'_u \times p'_v$$

$$\approx n + (\partial d/\partial u) n \times p_v + (\partial d/\partial v) n \times p_u$$

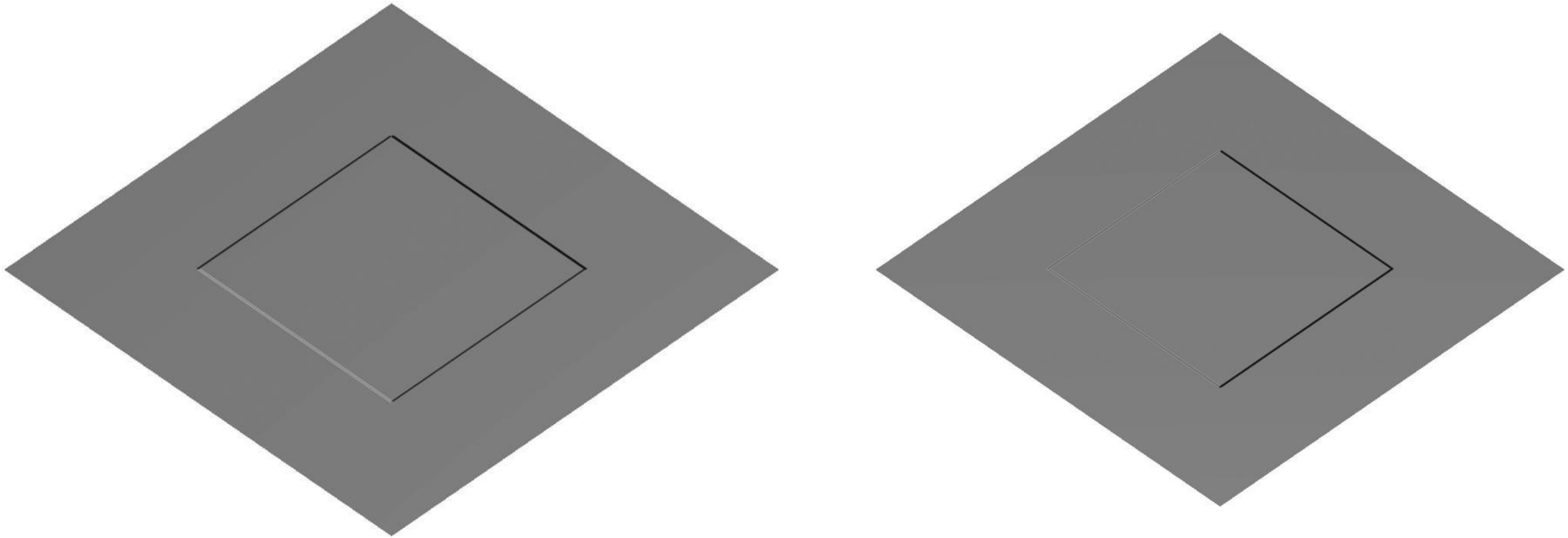
The vectors $n \times p_v$ and $n \times p_u$ lie in the tangent plane
Hence the normal is displaced in the tangent plane
Must precompute the arrays $\partial d/\partial u$ and $\partial d/\partial v$
Finally, we perturb the normal during shading

Image Processing

- Suppose that we start with a function $d(u,v)$
- We can sample it to form an array $D=[d_{ij}]$
- Then $\partial d / \partial u \approx d_{ij} - d_{i-1,j}$
  and $\partial d / \partial v \approx d_{ij} - d_{i,j-1}$
- Embossing: multipass approach using floating point buffer

Example

Single Polygon and a Rotating Light Source



How to do this?

- The problem is that we want to apply the perturbation at all points on the surface
- Cannot solve by vertex lighting (unless polygons are very small)
- Really want to apply to every fragment
- Can’t do that in fixed function pipeline
- But can do with a fragment program!!
- See bumpmap.html and bumpmap.js

Compositing and Blending

Sorin Babii
sorin.babii@cs.upt.ro

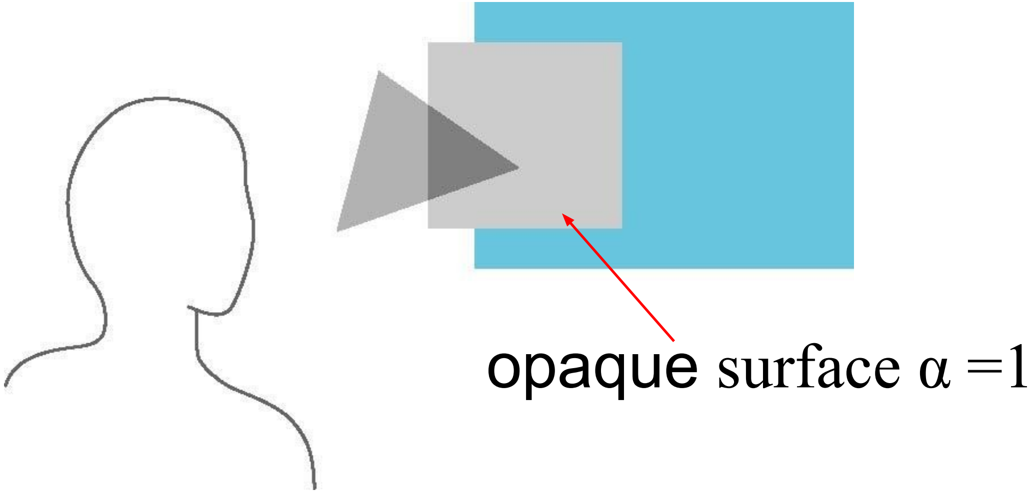
Objectives

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

Opacity and Transparency

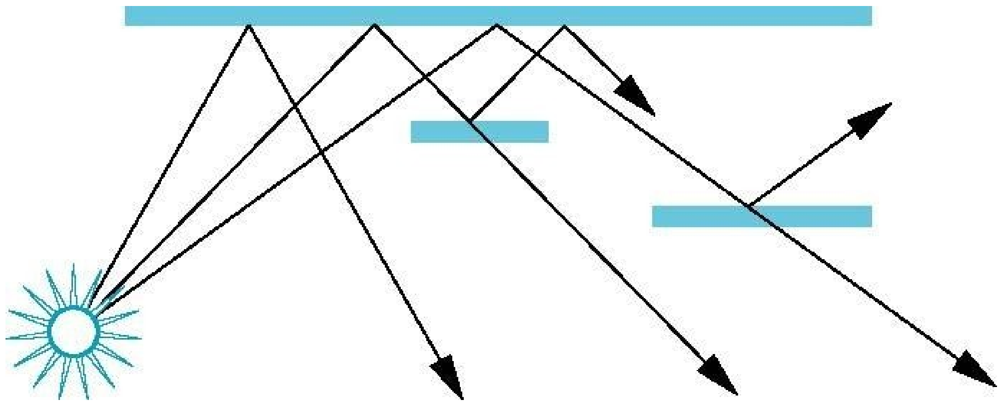
- Opaque surfaces permit no light to pass through
- Transparent surfaces permit all light to pass
- Translucent surfaces pass some light

  translucency = 1 – opacity (α)



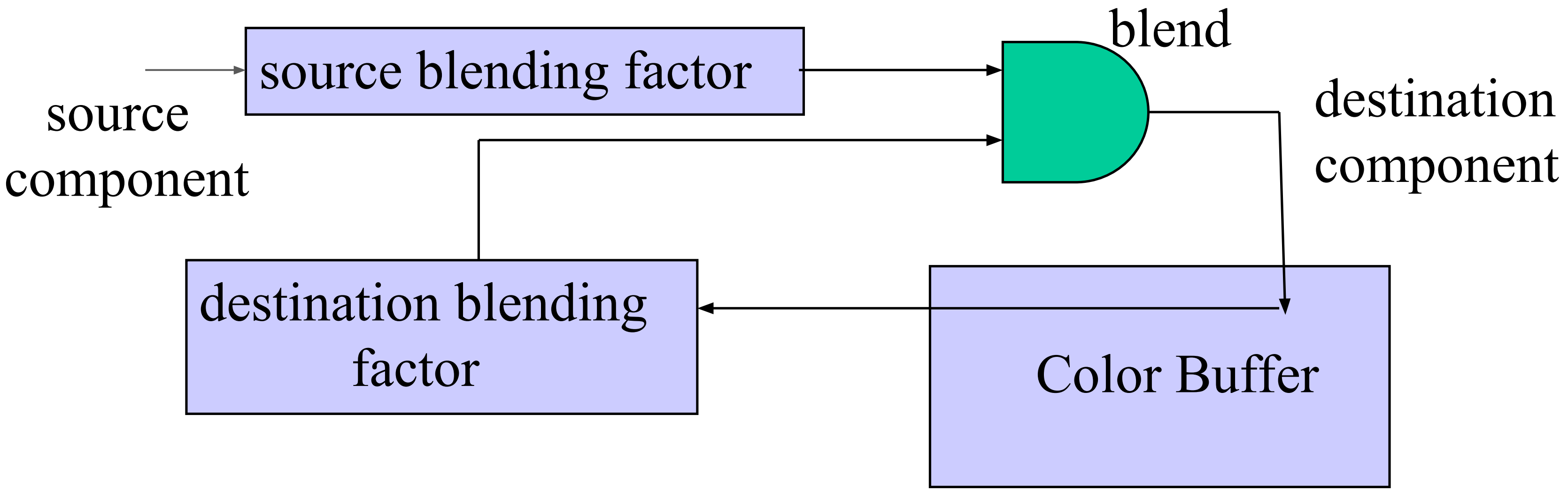
Physical Models

- 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

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

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

Suppose that the source and destination colors are

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

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

Blend as

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

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_ALPHAas the source and destination blending factors

  $R'_1 = \alpha_1 R_1 +(1- \alpha_1) R_0, \ldots\ldots$

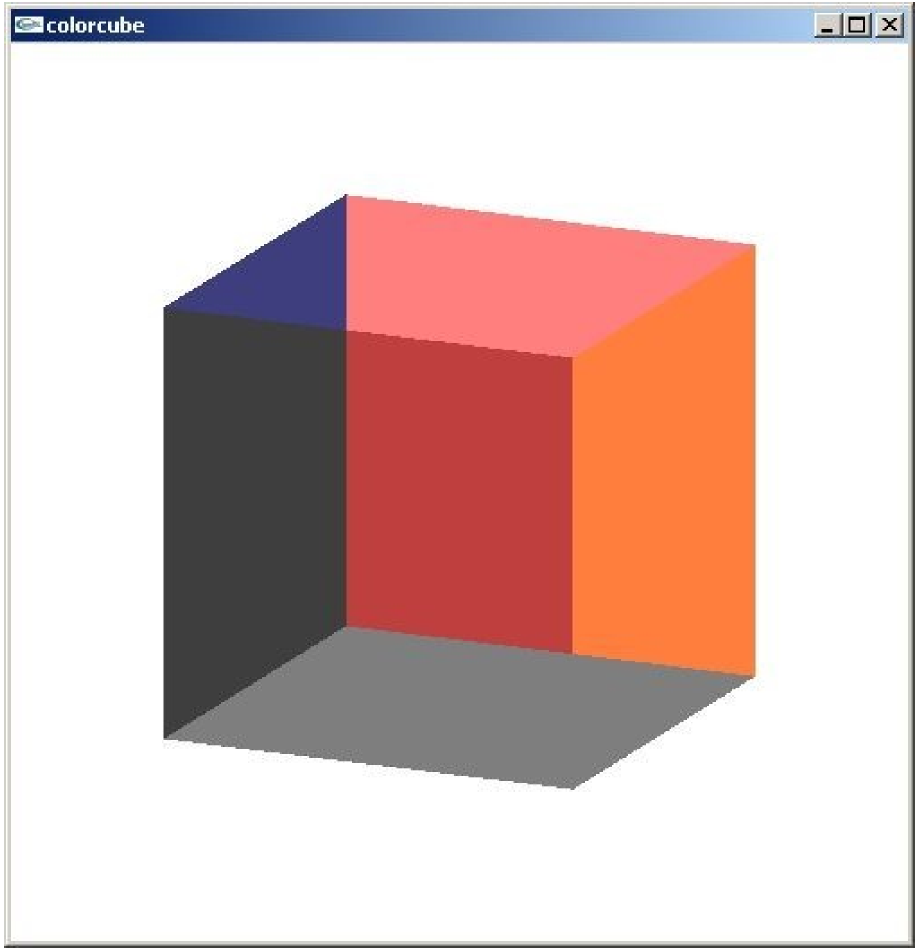
- Note this formula is correct if polygon is either opaque or transparent

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 lose 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



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

Video

http://staff.cs.upt.ro/~sorin/webgl/Code/w07/cubet.html

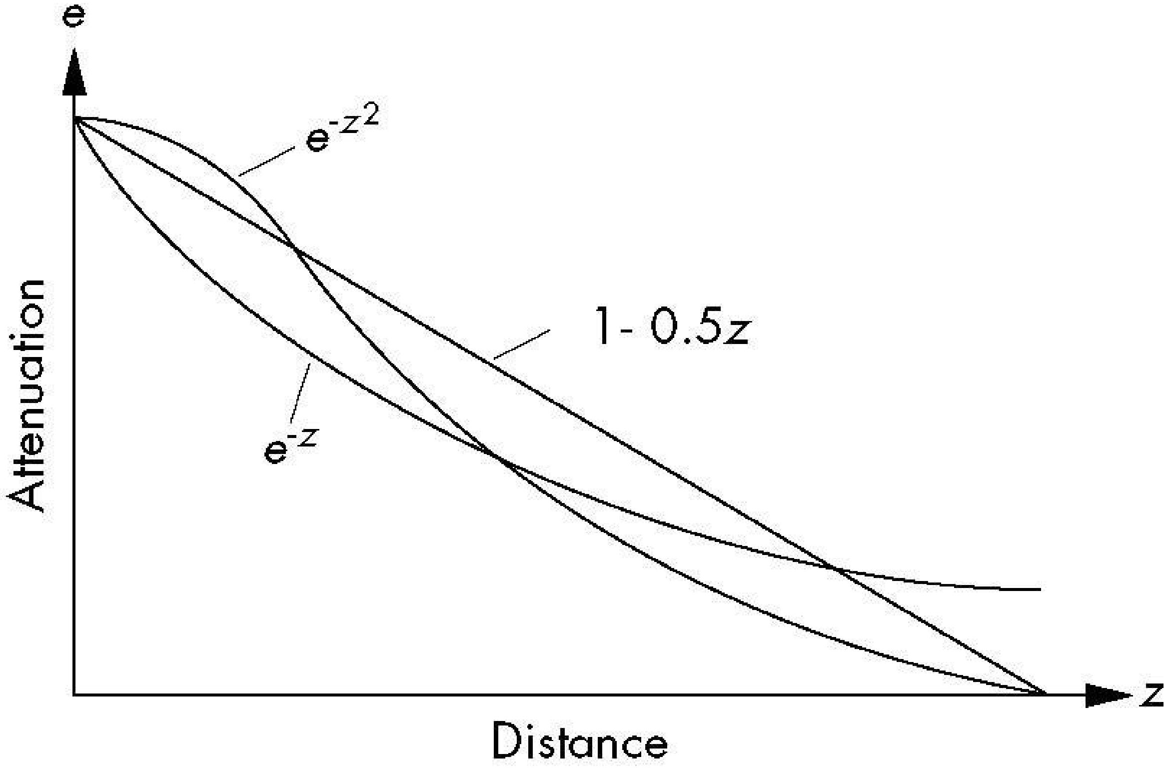
Fog

- 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

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

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

Fog Functions

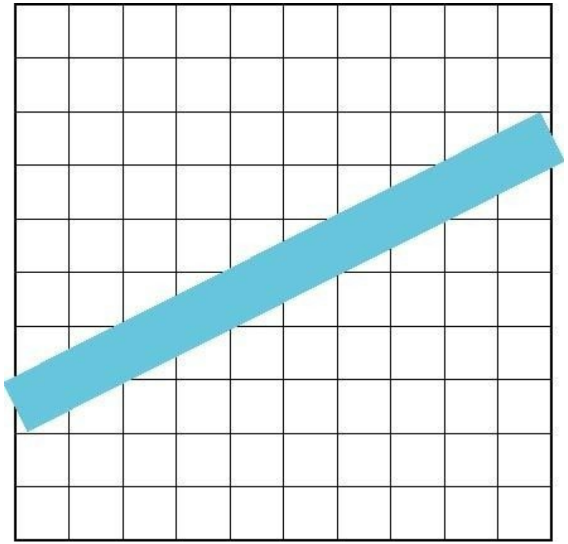


Compositing and HTML

- Desktop OpenGL: the A component has no effect unless blending is enabled
- 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

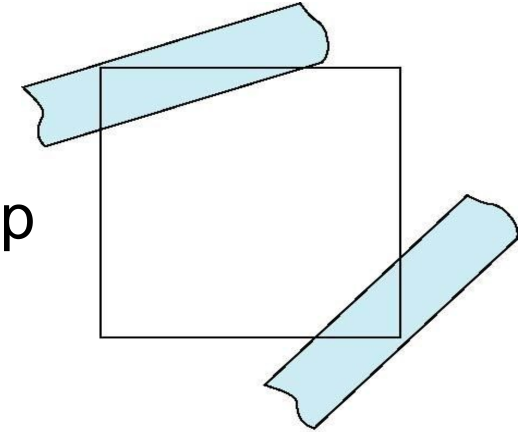
- Ideal raster line is one pixel wide
- All line segments, other than vertical and horizontal segments, partially cover pixels
- Simple algorithms color only whole pixels
- Lead to the “jaggies” or aliasing
- Similar issue for polygons



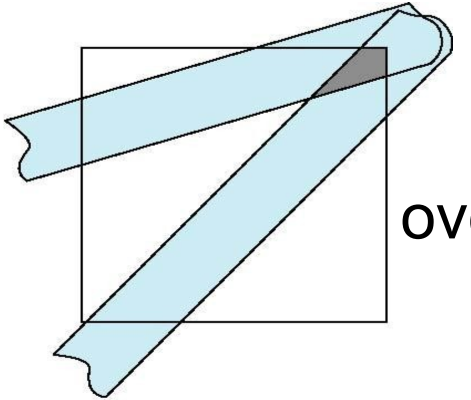
Antialiasing

- 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

no overlap



(a)

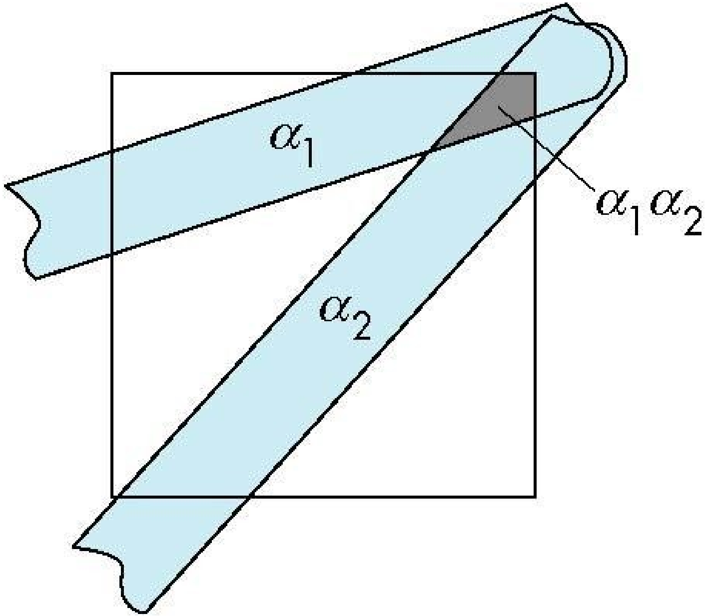


overlap

(b)

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

Imaging Applications

Sorin Babii
sorin.babii@cs.upt.ro

Objectives

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

Accumulation Techniques (OpenGL)

- 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 $n \times m$ texture map
- Suppose that in addition we use an $n \times m$ 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 1024×1024 texture in the texture object "image"

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

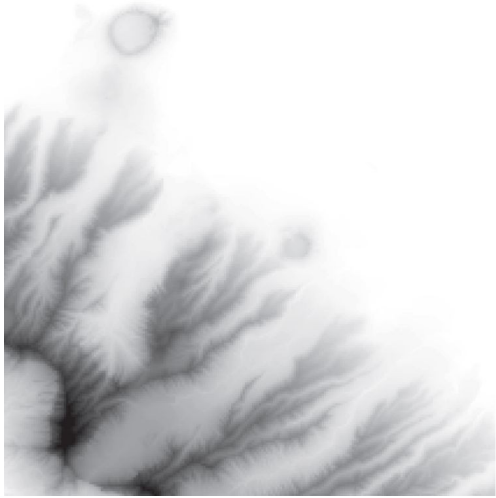
sampler2D(image, vec2(x+1.0/1024.0), y); returns the value of the texel to the right of (x,y)

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

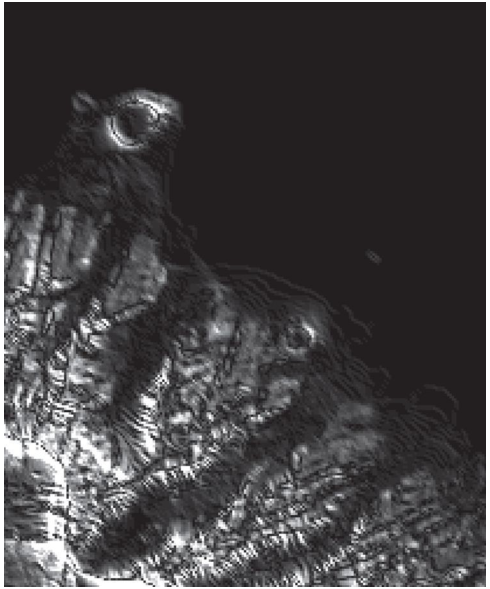
Image Enhancer

```
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 $8\mathrm{K} \times 8\mathrm{K}$
- For scalar imaging, we can do twice as well using all four color components

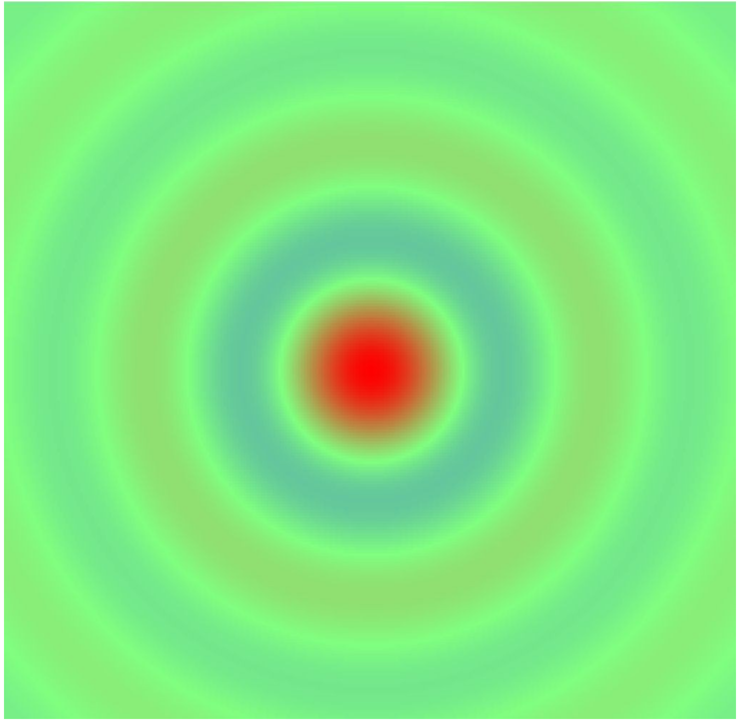
$$\begin{bmatrix} R & G \\ B & A \end{bmatrix}$$

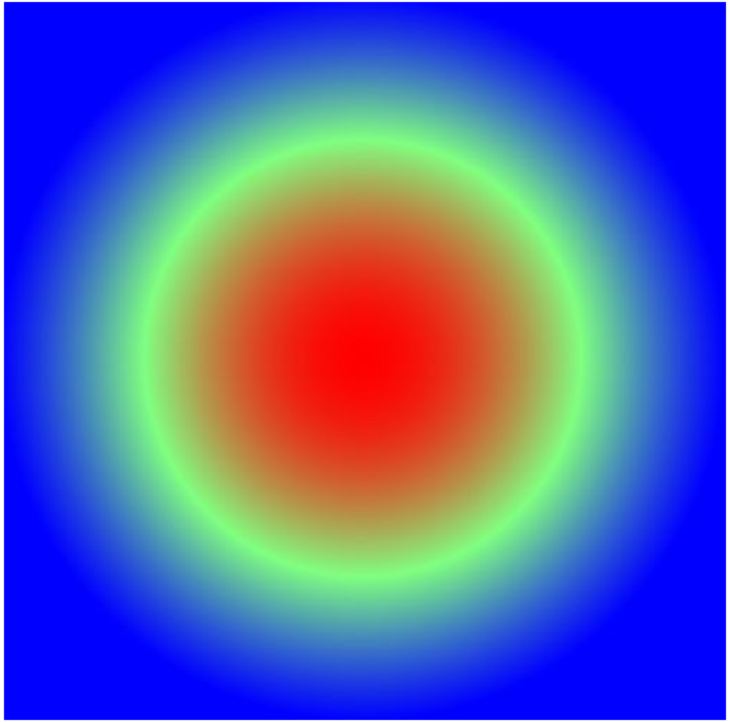
Indexed and Pseudo Color

- 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

Fractals - Mandelbrot

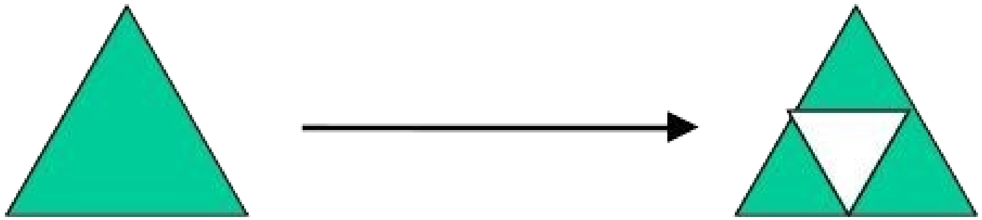
Sorin Babii
sorin.babii@cs.upt.ro

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:



Repeat n times.

As $n \to \infty$

  Area $\to 0$

  Perimeter $\to \infty$

Not a normal geometric object

More about fractal curves and surfaces later

Reminder: Complex Arithmetic

- Complex number defined by two scalars

$$z = x + jy$$
$$j^2 = -1$$

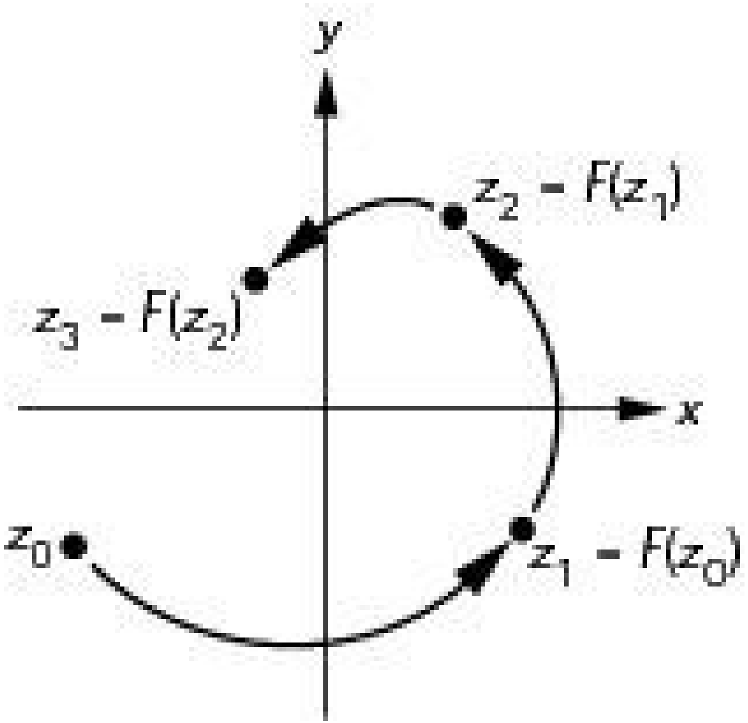
- Addition and Subtraction

$$z_1 + z_2 = x_1 + x_2 + j(y_1 + y_2)$$
$$z_1 * z_2 = x_1 * x_2 - y_1 * y_2 + j(x_1 * y_2 + x_2 * y_1)$$

- Magnitude

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

Iteration in the Complex Plane



Mandelbrot Set

iterate on $z_{k+1}=z_k{}^2+c$
with $z_0 = 0 + j0$

Two cases as $k \rightarrow\infty$

$|z_k| \rightarrow\infty$

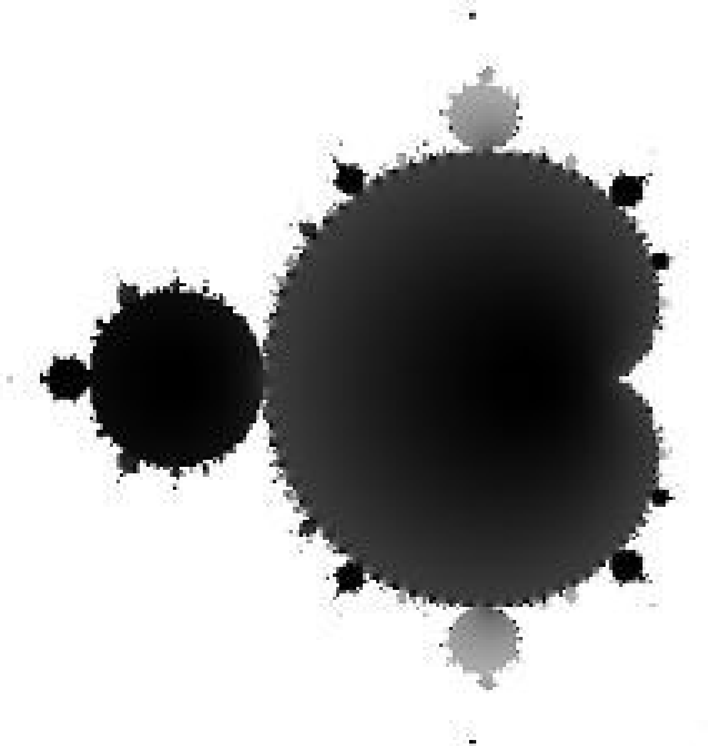
$|z_k|$ remains finite

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

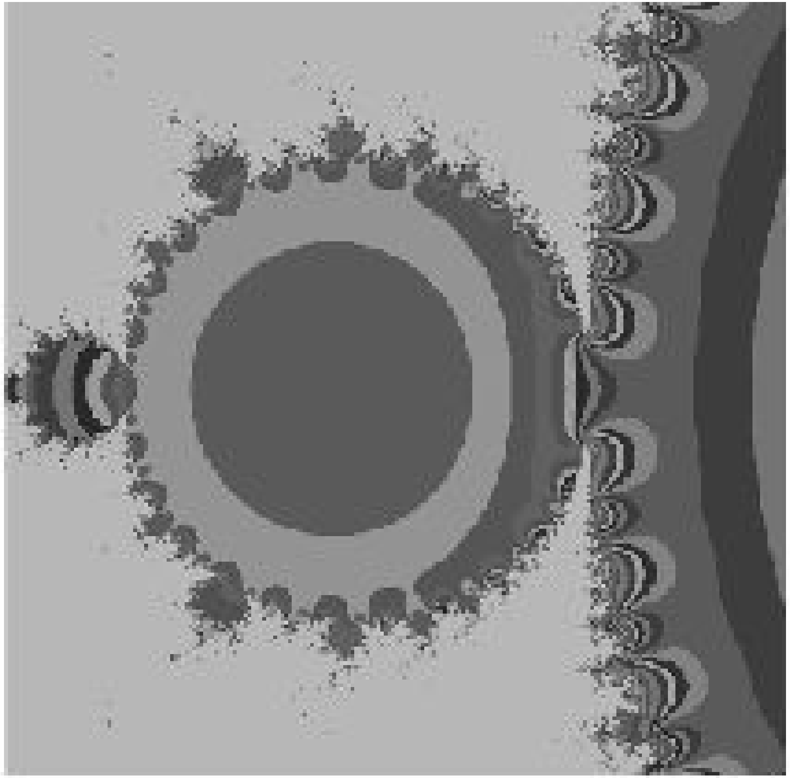
Mandelbrot Set



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 (1)

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

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

Computing in JS File (2)

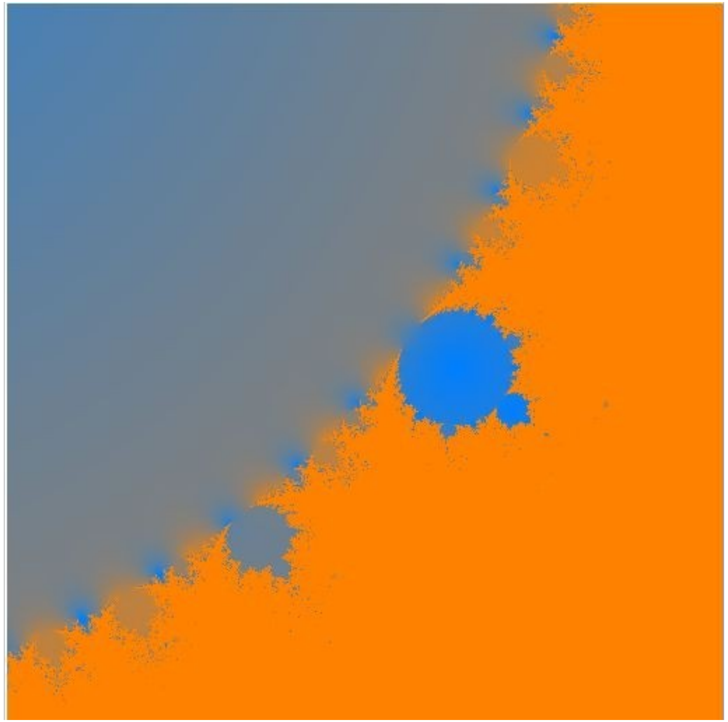
```
for ( var i = 0; i < n; i++ )
    for ( var j = 0; j < m; j++ ) {
         var x = i * ( width / (n - 1) ) + cx - width / 2;
         var y = j * ( height / ( m - 1 ) ) + cy - height / 2;
        var c = [ 0.0, 0.0 ];
        var p =  [ x, y ];

   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 (3)

```
// 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 (1)

```
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 (2)

```
   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: 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

Video

http://staff.cs.upt.ro/~sorin/webgl/Code/w10/mandelbrot2.html