Objectives

- Introduce fragment tests and operations
- Learn to use blending
- Introduce additional WebGL buffers
- Reading and writing buffers
- Buffers and Images

Fragment Tests and Operations

- After the fragment shader is executed a series of tests and operations are performed on the fragment
- Determine how and whether a fragment color is drawn into the frame buffer

Tests and operations are performed in the following order:
- Scissor test
- Stencil test
- Depth test
- Blending
- Dithering

On/off `gl.enable()`, `gl.disable()`
Pixel Tests

- **Scissor**
  - Only draw in a rectangular portion of screen
  - `gl.scissor()` – Specify rectangle
  - Default rectangle matches window

- **Depth**
  - Draw based on depth value and comparison function
  - `gl.depthFunc()` – Specify comparison function
  - Default is `gl.LESS`

Dithering

- Dithering may be enabled (`gl.DITHER`) on some systems with limited color resolution
- System/hardware-dependent

Opacity and Transparency using Blending

- Opaque surfaces permit no light to pass through
- Transparent surfaces permit all light to pass
- Translucent surfaces pass some light
  - `translucency = 1 – opacity (\(\alpha\))`

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 for Blending

- Use a component of RGBA (or RGB\(\alpha\)) 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
  
  \[ s = [s_r, s_g, s_b, s_a] \]
  
  \[ d = [d_r, d_g, d_b, d_a] \]

- Suppose that the source and destination colors are
  
  \[ b = [b_r, b_g, b_b, b_a] \]
  
  \[ c = [c_r, c_g, c_b, c_a] \]

- Blend as
  
  \[ c' = s \cdot b + d \cdot c \]
  
  \[ c' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_a s_a + c_a d_a] \]

WebGL Blending

- Must enable blending and set source and destination factors
  
  \[ \text{gl.enable}(\text{gl.BLEND}) \]
  
  \[ \text{gl.blendFunc}(\text{source_factor}, \text{destination_factor}) \]

- Only certain factors supported
  
  - \text{gl.ZERO}, \text{gl.ONE}
  
  - \text{gl.SRC_ALPHA}, \text{gl.ONE_MINUS_SRC_ALPHA}
  
  - \text{gl.DST_ALPHA}, \text{gl.ONE_MINUS_DST_ALPHA}
  
  - See WebGL spec 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 \((R_1, G_1, B_1, a_1)\)

- Select \text{gl.SRC_ALPHA} and \text{gl.ONE_MINUS_SRC_ALPHA}
  
  as the source and destination blending factors

- \(R'_0 = a_1 R_1 + (1 - a_1) R_0\) ...

- 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

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 \text{gl.depthMask(false)} which makes depth buffer read-only

- Sort polygons first to remove order dependency!

- Draw back to front
Blending 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

Fragment Tests and Operations

• Tests and operations are performed in the following order
  - Scissor test
  - Stencil test
  - Depth test
  - Blending
  - Dithering
• On/off gl.enable(), gl.disable()
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.

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).

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

\[
\text{sampler2D(image, vec2(x,y)) returns the value of the texture at (x,y)}
\]

\[
\text{sampler2D(image, vec2(x+1.0/1024.0), y); returns the value of the texel to the right of (x,y)}
\]

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

```
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;
}
```
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

\[
\begin{align*}
vec4 \, gx &= 3.0*\text{texture2D}( \text{texture}, \text{vec2}(x+d, y)) \\
&+ \text{texture2D}( \text{texture}, \text{vec2}(x+d, y-d)) \\
&+ \text{texture2D}( \text{texture}, \text{vec2}(x-d, y-d)) \\
&- 3.0*\text{texture2D}( \text{texture}, \text{vec2}(x-d, y)) \\
&- \text{texture2D}( \text{texture}, \text{vec2}(x-d, y+d)) \\
&- \text{texture2D}( \text{texture}, \text{vec2}(x+d, y-d));
\end{align*}
\]

\[
\begin{align*}
vec4 \, gy &= 3.0*\text{texture2D}( \text{texture}, \text{vec2}(x, y+d)) \\
&+ \text{texture2D}( \text{texture}, \text{vec2}(x+d, y+d)) \\
&+ \text{texture2D}( \text{texture}, \text{vec2}(x-d, y+d)) \\
&- 3.0*\text{texture2D}( \text{texture}, \text{vec2}(x, y-d)) \\
&- \text{texture2D}( \text{texture}, \text{vec2}(x+d, y)) \\
&- \text{texture2D}( \text{texture}, \text{vec2}(x-d, y-d));
\end{align*}
\]

\[
mag = \sqrt{gx*gx + gy*gy};
\]

\[
gl\_\text{FragColor} = \text{vec4}(\text{mag}, \text{mag}, \text{mag}, 1.0);
\]

Using Multiple Textures

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

\[
gl\_\text{FragColor} = \text{sampler2D}(\text{texture1}, \text{vec2}(x, y)) + \text{sampler2D}(\text{texture2}, \text{vec2}(x, y));
\]

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 = \text{texture2D}(\text{texture}, \text{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\_\text{FragColor} = color;
}
```
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

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 → ∞
Area → 0
Perimeter → ∞
Not a normal geometric object

Complex Arithmetic

- Complex number defined by two scalars
  \( z = x + iy \)
  \( |z|^2 = -1 \)
- Addition and Subtraction
  \( z_1 + z_2 = x_1 + x_2 + i(y_1 + y_2) \)
  \( z_1 - z_2 = x_1 - x_2 + i(y_1 - y_2) \)
- Magnitude
  \( |z|^2 = x^2 + y^2 \)

Iteration in the Complex Plane

\[ z_0 \]
\[ z_1 = f(z_0) \]
\[ z_2 = f(z_1) \]
\[ z_3 = f(z_2) \]
\[ z_4 = f(z_3) \]
Mandelbrot Set

iterate on $z_{k+1} = z_k^2 + c$
with $z_0 = 0 + i0$

Two cases as $k \to \infty$
- $|z_k| \to \infty$
- $|z_k|$ 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

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

Computing in the JS File I

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

```javascript
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 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

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

• 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

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 = 10000.0;
    float m = 10000.0;
    // 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 */
}
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 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
}

// 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;
Where are the Buffers?

- HTML5 Canvas
  - Default front and back color buffers
  - Under control of local window system
  - Physically on graphics card
- Depth buffer also on graphics card
- Stencil buffer
  - Holds masks
- Most RGBA buffers 8 bits per component
- Latest are floating point (IEEE)

Other Buffers

- Desktop OpenGL supported other buffers
  - auxiliary color buffers
  - accumulation buffer
  - these were on application side
  - now deprecated
- GPUs have their own or attached memory
  - texture buffers
  - off-screen buffers
    - not under control of window system
    - may be floating point

Images

- Framebuffer contents are unformatted
  - usually RGB or RGBA
  - one byte per component
  - no compression
- Standard Web Image Formats
  - jpeg, gif, png
- WebGL has no conversion functions
  - Understands standard Web formats for texture images

Writing into Buffers

- WebGL does not contain a function for writing bits into frame buffer
  - Use texture functions instead
- We can use the fragment shader to do bit level operations on graphics memory
- Bit Block Transfer (BitBit) operations act on blocks of bits with a single instruction

BitBit

- Conceptually, we can consider all of memory as a large two-dimensional array of pixels
- We read and write rectangular block of pixels
  - Bit block transfer (bitbit) operations
- The frame buffer is part of this memory

The (Old) Pixel Pipeline

- OpenGL has a separate pipeline for pixels
  - Writing pixels involves
    - Moving pixels from processor memory to the frame buffer
    - Format conversions
    - Mapping, Lookups, Tests
  - Reading pixels
    - Format conversion

Images

- Framebuffer contents are unformatted
  - usually RGB or RGBA
  - one byte per component
  - no compression
- Standard Web Image Formats
  - jpeg, gif, png
- WebGL has no conversion functions
  - Understands standard Web formats for texture images

Writing into Buffers

- WebGL does not contain a function for writing bits into frame buffer
  - Use texture functions instead
- We can use the fragment shader to do bit level operations on graphics memory
- Bit Block Transfer (BitBit) operations act on blocks of bits with a single instruction

BitBit

- Conceptually, we can consider all of memory as a large two-dimensional array of pixels
- We read and write rectangular block of pixels
  - Bit block transfer (bitbit) operations
- The frame buffer is part of this memory
Packing and Unpacking

- Compressed or uncompressed
- Indexed or RGB
- Bit Format
  - little or big endian
- WebGL (and shader-based OpenGL) lacks most functions for packing and unpacking
  - use texture functions instead
  - can implement desired functionality in fragment shaders

Buffer Reading

- WebGL can read pixels from the current framebuffer with gl.readPixels
- Returns only 8 bit RGBA values
- In general, the format of pixels in the frame buffer is different from that of processor memory and these two types of memory reside in different places
  - Need packing and unpacking
  - Reading can be slow
- Drawing through texture functions and off-screen memory (frame buffer objects)

WebGL Pixel Function

```
gl.readPixels(x, y, width, height, format, type, myimage)
```

- start pixel in frame buffer
- size
- type of pixels
- type of image
- pointer to processor memory

```
var myimage[512*512*4];
gl.readPixels(0, 0, 512, 512, gl.RGBA, gl.UNSIGNED_BYTE, myimage);
```

Formats & Types

- `gl.RGBA`
- `gl.RGB`
- `gl.ALPHA`
- `gl.UNSIGNED_BYTE`
- `gl.UNSIGNED_SHORT_5_6_5`
- `gl.UNSIGNED_SHORT_4_4_4_4`
- `gl.UNSIGNED_SHORT_5_5_5_2`
- `gl.FLOAT`

Clearing Buffers

- A clear (default) value may be set for each buffer
  - `gl.clearColor()`
  - `gl.clearDepth()`
  - `gl.clearStencil()`
- `gl.clear(Glbitfield mask)`
  - Clears the specified buffer
    - `gl.COLOR_BUFFER_BIT, gl.DEPTH_BUFFER_BIT, gl.STENCIL_BUFFER_BIT`
  - Can be or’ed together in one `clear()` call

Masking Buffers

- A buffer may be `mask`’ed, i.e. enabled or disabled
  - `gl.colorMask(red, green, blue, alpha)`
    - Arguments are booleans
  - `gl.depthMask(flag)`
  - `gl.stencilMask(mask)`
  - `gl.stencilMaskSeparate(face, mask)`
    - Stencil specific sides (front & back) of triangles
Render to Texture

- GPUs now include a large amount of texture memory that we can write into
- Advantage: fast (not under control of window system)
- Using frame buffer objects (FBOs) we can render into texture memory instead of the frame buffer and then read from this memory
  - Image processing
  - GPGPU

Framebuffer Objects

Objectives

- Look at methods that use memory on the graphics card
- Introduce off screen rendering
- Learn how to create framebuffer objects
  - Create a renderbuffer
  - Attach resources

Discrete Processing in WebGL

- Recent GPUs contain large amounts of memory
  - Texture memory
  - Framebuffer
  - Floating point
- Fragment shaders support discrete operations at the pixel level
- Separate pixel (texel) pipeline

Accessing the Framebuffer

- Pre 3.1 OpenGL had functions that allowed access to the framebuffer and other OpenGL buffers
  - Draw Pixels
  - Read Pixels
  - Copy Pixels
  - BitBlt
  - Accumulation Buffer functions
- All deprecated

Going between CPU and GPU

- We will see that we can write pixels as texels to texture memory
- Texture objects reduce transfers between CPU and GPU
- Transfer of pixel data back to CPU slow
- Want to manipulate pixels without going back to CPU
  - Image processing
  - GPGPU
Framebuffer Objects

- Framebuffer Objects (FBOs) are buffers that are created by the application
  - Not under control of window system
  - Cannot be displayed
  - Can attach a renderbuffer to a FBO and can render off screen into the attached buffer
  - Attached buffer can then be detached and used as a texture map for an on-screen render to the default frame buffer

Steps

- Create an Empty Texture Object
- Create a FBO
- Attach renderbuffers to create and store texture image
- Bind FBO
- Render scene
- Detach renderbuffer
- Bind window system frame buffer
- Bind texture
- Render with new texture

Creating a FBO

- We create a framebuffer object in a similar manner to other objects
- Creating an FBO creates an empty FBO
- Must add needed resources
  - Can add a renderbuffer to render into
  - Can add a texture which can also be rendered into
  - For hidden surface removal we must add a depth buffer attachment to the renderbuffer
  - Can also add a stencil buffer to FBO

Render to Texture

- Textures are shared by all instances of the fragment shader
- If we render to a texture attachment we can create a new texture image that can be used in subsequent renderings
- Use a double buffering strategy for operations such as convolution

Empty Texture Object

```javascript
var texture1 = gl.createTexture();
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, texture1);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, 512, 512, 0, gl.RGBA, gl.UNSIGNED_BYTE, null);
gl.generateMipmap(gl.TEXTURE_2D);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST_MIPMAP_LINEAR);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
```

Frame Buffer Object

```javascript
var framebuffer = gl.createFramebuffer();
gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
framebuffer.width = 512;
framebuffer.height = 512;
renderbuffer = gl.createRenderbuffer();
gl.bindRenderbuffer(gl.RENDERBUFFER, renderbuffer);
gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT, gl.RENDERBUFFER, renderbuffer);
```

// Attach color buffer
```javascript
gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D, texture1, 0);
```
Rest of Initialization

• Same as previous examples
  - Allocate VAO
  - Fill VAO with data for render to texture
• Initialize two program objects with different shaders
  - First for render to texture
  - Second for rendering with created texture

Objectives

• Examples of render-to-texture
• Render a triangle to texture, then use this texture on a rectangle
• Introduce buffer pingponging

Program Objects and Shaders

• For most applications of render-to-texture we need multiple program objects and shaders
  - One set for creating a texture
  - Second set for rendering with that texture
• Applications that we consider later such as buffer pingponging may require additional program objects

Program Object 1 Shaders

pass through vertex shader:
attribute vec4 vPosition;
void main()
{
  gl_Position = vPosition;
}

fragment shader to get a red triangle:
precision mediump float;
void main()
{
  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0);
}

Program Object 2 Shaders

// vertex shader
attribute vec4 vPosition;
attribute vec2 vTexCoord;
varying vec2 fTexCoord;
void main()
{
  gl_Position = vPosition;
  fTexCoord = vTexCoord;
}

// fragment shader
precision mediump float;
varying vec2 fTexCoord;
uniform sampler2D texture;
void main()
{
  gl_FragColor = texture2D( texture, fTexCoord);
}
First Render (to Texture)

```javascript
// Initialize the vertex position attribute from the vertex shader
var vPosition = gl.getAttribLocation(program1, "vPosition");
gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vPosition);
```

Set Up Second Render

```javascript
// Bind to default window system framebuffer
```

Data for Second Render

```javascript
// Assume we have already set up a texture object with null texture image
```

Render a Quad with Texture

```javascript
gl.uniform1i(gl.getUniformLocation(program2, "texture"), 0);
```

Dynamic 3D Example

```
```

Buffer Ping-ponging

- Iterative calculations can be accomplished using multiple render buffers
- Original data in texture buffer 1
- Render to texture buffer 2
- Swap buffers and rerender to texture
Buffer Applications

Anti-aliasing

- Aliasing – artifacts produced from inadequate sampling
  - Jagged edges
  - Missing thin objects/features

- Anti-aliasing – removing artifacts via super-sampling, filtering, blurring, smoothing
- OpenGL offers a number of ways to perform anti-aliasing
- More limited in WebGL

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

Area Averaging

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

Area Averaging

- Color a pixel by adding in a fraction of the fragment’s color
  - Fraction depends on percentage of pixel covered by object
  - Fraction depends on whether there is overlap

no overlap

overlap
OpenGL Antialiasing

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

```cpp
// Compute fractional alpha values along edges
// Based on pixel coverage
glEnable(GL_POINT_SMOOTH);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_POLYGON_SMOOTH);

// Note most hardware will automatically antialias
```

WebGL Antialiasing

• Full-screen antialiasing
• Multiple renderings with texture pipping
• Jitter view
• Average several jittered images together

Fog

• We can blend 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
  \[ C'_s = f \cdot C_s + (1-f) \cdot C_f \]
• \( f \) is the fog factor based on depth
  - Exponential
  - Gaussian
  - Linear (depth cueing)
• Hard-coded fog deprecated but can recreate

Fog Functions

Fog Effect

http://www.engin.swarthmore.edu/~jshin1

Interactive Depth-of-Field

• Jitter camera
• Each frustum has common plane “in focus”
• Accumulate & blend images
Reflections

- One of the most noticeable effects of inter-object lighting
- Direct calculation of the physics (ray tracing) is too expensive
- Our focus is to capture the most significant reflection while minimizing overhead via rendering the "virtual object"

Planar Reflections

- The most common reflection – flat mirror, floor, wall, etc
- Creating virtual objects (or reflected objects) is much easier
- A view-independent operation – only consider the relative position of the object and the reflector
- The virtual object is created by transforming the object across the reflector plane

Image vs. Object Space Methods

- Image space methods: create a texture from a view of the reflected objects and apply it to the reflector
  - Advantage: does not depend on the object geometry
  - Disadvantage: sampling issue and also only an approximation (environment mapping as an example)
- Object space methods: create the actual geometry of the object to be reflected and render it to the reflector
  - Disadvantage: Limited to planar reflections
  - Advantage: more accurate reflection (for nearby objects)
- Both methods need to create the virtual objects

Planar Reflections

- An important task: clip the reflected geometry so it is only visible on the reflector surface
  - Beyond the reflector boundaries and in front of reflector
Clipping using the stencil

- The key is you only want the reflected geometry to appear on the reflector surface
- Use stencil buffer:
  - Clear the stencil buffer
  - Render the reflector and set the stencil
  - Render the reflected geometry only where the stencil pixels have been set
- The above algorithm uses the stencil buffer to control where to draw the reflection

Another method: render the reflected object first, and then render the reflector to set the stencil buffer, then clear the color buffer everywhere except where the stencil is set

- This method is to use the stencil buffer to control where to erase the incorrect reflection
- Advantage: when it is faster to use stencil to control clearing the scene than drawing the entire scene with stencil tests

The stencil erase algorithm

Reflection Effect

Motion Effects

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 pingingpong

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);
    if (flag)
    {
        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);
    }
    else
    {
        gl.bindTexture(gl.TEXTURE_2D, texture2);
        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);
    }
var r = 1024/texSize;

gl.viewport(0, 0, r*texSize, r*texSize);

gl.clearColor(0, 0, 0, 1);

gl.drawArrays(gl.TRIANGLE_STRIP, 0, 4);

// 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));

// swap textures

flag = !flag;
requestAnimFrame(render);

// 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;
        }
    }
}

// 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)
Picking by Color

Objectives

- Use off-screen rendering for picking
- Example: rotating cube with shading
  - indicate which face is clicked on with mouse
  - normal rendering uses vertex colors that are interpolated across each face
  - Vertex colors could be determined by lighting calculation or just assigned
  - use console log to indicate which face (or background) was clicked

Algorithm

- Assign a unique color to each object
- When the mouse is clicked:
  - Do an off-screen render using these colors and no lighting
  - use gl.readPixels to obtain the color of the pixel where the mouse is located
  - map the color to the object id
  - do a normal render to the display

Shaders

- Only need one program object
- Vertex shader: same as in previous cube examples
  - includes rotation matrices
  - gets angle as uniform variable
- Fragment shader
  - Stores face colors for picking
  - Gets vertex color for normal render from rasterizer
- Send uniform integer to fragment shader as index for desired color

Fragment Shader

```cpp
precision mediump float;

uniform int i;

varying vec4 fColor;
void main() {
  vec4 c[7];
  c[0] = fColor;
  c[1] = vec4(1.0, 0.0, 0.0, 1.0);
  c[2] = vec4(0.0, 1.0, 0.0, 1.0);
  c[3] = vec4(0.0, 0.0, 1.0, 1.0);
  c[4] = vec4(1.0, 1.0, 0.0, 1.0);
  c[5] = vec4(0.0, 1.0, 1.0, 1.0);
  c[6] = vec4(1.0, 0.0, 1.0, 1.0);

  // no case statement in GLSL
  if (i==0) gl_FragColor = c[0];
  else if (i==1) gl_FragColor = c[1];
  else if (i==2) gl_FragColor = c[2];
  else if (i==3) gl_FragColor = c[3];
  else if (i==4) gl_FragColor = c[4];
  else if (i==5) gl_FragColor = c[5];
  else if (i==6) gl_FragColor = c[6];
}
```
Setup

// Allocate a frame buffer object
framebuffer = gl.createFramebuffer();
gl.bindFramebuffer( gl.FRAMEBUFFER, framebuffer );
// Attach color buffer
// Must first define empty texture object “texture”
gl.framebufferTexture2D( gl.FRAMEBUFFER,
    gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D, texture, 0 );
gl.bindFramebuffer( gl.FRAMEBUFFER, null );

Event Listener

canvas.addEventListener("mousedown", function(){
    gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
    gl.clear( gl.COLOR_BUFFER_BIT );
    for(var i=0; i<6; i++) {
        gl.uniform1i(gl.getUniformLocation(program, "i"), i+1);
        gl.drawArrays( gl.TRIANGLES, 6*i, 6 );
    }
    var x = event.clientX;
    var y = canvas.height - event.clientY;   // Flipping y !!!
    gl.readPixels(x, y, 1, 1, gl.RGBA,
        gl.UNSIGNED_BYTE, color);
    var color = canvas.toDataURL();
    console.log(color);
});

Picking by Selection

• Possible with render-to-texture
• When mouse clicked do an off-screen rendering with new viewing conditions that render only a small area around mouse
• Or render full scene and just sample off-screen image at mouse click position
• Keep track of what gets rendered to this off-screen buffer
• Know what was picked by returned color

Event Listener

if(color[0]===255)
    if(color[1]===255) console.log("yellow");
else if(color[2]===255) console.log("magenta");
else console.log("red");
else if(color[1]===255)
    if(color[2]===255) console.log("cyan");
else console.log("green");
else if(color[2]===255)
    console.log("blue");
else console.log("background");

HW9 Suggestions

• Create an off-screen frame buffer
  - With color(texture) and depth buffers
• Draw your three objects to this frame buffer, with each object having a unique, constant color
  - This color will act as the object’s ID
• Draw normally to on-screen frame buffer
• Allow user to click in the graphics window
HW9 Suggestions

- Read color at click point out of the off-screen frame buffer
- The color will tell you if an object was selected and which one
- Randomly change the diffuse color of the picked object
- YourEventListener should draw to both the off-screen and on-screen buffers