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

• Determine how and whether a fragment color is drawn into the frame buffer

This is where the depth test is being performed
**Pixel Tests**

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

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

**Stencil**

- Draw based on values in stencil buffer, if available and enabled
- Used for drawing into an irregular region of color buffer
- `gl.stencilFunc()` – Specifies comparison function, reference value and mask
- `gl.stencilOp()` – Specifies how fragments can modify stencil buffer
- Used for reflections, capping, and stippling

**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
  \[ \text{translucency} = 1 - \text{opacity} (\alpha) \]

**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 RGBα) color to store opacity
- During rendering, we can expand our writing model to use RGBA values

- **Source Blending Factor**
  - `source blending factor`

- **Blend**
  - `blend`

- **Destination Blending Factor**
  - `destination blending factor`

- **Color Buffer**
### Blending Equation

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

\[
\begin{align*}
  s &= [s_r, s_g, s_b, s_a] \\
  d &= [d_r, d_g, d_b, d_a]
\end{align*}
\]

Suppose that the source and destination colors are

\[
\begin{align*}
  b &= [b_r, b_g, b_b, b_a] \\
  c &= [c_r, c_g, c_b, c_a]
\end{align*}
\]

Blend as

\[
c' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_a s_a + c_a d_a]
\]

### WebGL Blending

- Must enable blending and set source and destination factors

\[
\begin{align*}
  &\text{gl.enable}(\text{gl.BLEND}) \\
  &\text{gl.blendFunc(source\_factor, destination\_factor)}
\end{align*}
\]

- 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, α_1)
- Select \text{gl.SRC\_ALPHA} and \text{gl.ONE\_MINUS\_SRC\_ALPHA} as the source and destination blending factors

\[
R'_1 = α_1 R_1 + (1 - α_1) R_0
\]

- 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 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

- 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()`

Imaging Applications

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 $n \times m$ 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 our 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 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**

```c
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)))
                   +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

vec4 gx = 3.0*texture2D( texture, vec2(x+d, y)) + texture2D( texture, vec2(x-d, y+d)) + texture2D( texture, vec2(x+d, y-d)) + texture2D( texture, vec2(x-d, y+d)) - 3.0*texture2D( texture, vec2(x+d, y-d)) - texture2D( texture, vec2(x-d, y+d));
vec4 gy = 3.0*texture2D( texture, vec2(x+d, y)) + texture2D( texture, vec2(x-d, 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;

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

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;
}
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

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

Complex Arithmetic

- Complex number defined by two scalars
  \[ z = x + jy \]
  \[ |z|^2 = -1 \]

- Addition and Subtraction
  \[ z_1 + z_2 = x_1 + x_2 + j(y_1 + y_2) \]
  \[ z_1 \cdot z_2 = x_1 \cdot x_2 - y_1 \cdot y_2 + j(x_1 \cdot y_2 + x_2 \cdot y_1) \]

- Magnitude
  \[ |z|^2 = x^2 + y^2 \]

Sierpinski Gasket

Rule based:

- Repeat \( n \) times. As \( n \to \infty \)
- Area \( \to 0 \)
- Perimeter \( \to \infty \)
- Not a normal geometric object

Computing the Mandelbrot Set

- 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
Iteration in the Complex Plane

Mandelbrot Set

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

Two cases as \( k \to \infty \)
- \( |z_k| \to \infty \)
- \( |z_k| \) 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; // 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 iterations per point
var n = 512;
var m = 512;
var texImage = new Uint8Array(4*n*m);
```

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 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 II

```javascript
// 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;
```

Computing in JS File III

- 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

```c
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;
    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;
    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;
}
```

Fragment Shader II

```c
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
}
```

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.

Buffers

- Define a buffer by its spatial resolution \((n \times m)\) and its depth (or precision) \(k\), the number of bits per pixel.
WebGL Frame Buffer

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 (BitBlt) operations act on blocks of bits with a single instruction

BitBlt

- 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 (bitblt) 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

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 masked, 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
  - BitBit
  - 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

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

Render to Texture

- Textures are shared by all instances of the fragment shade
- 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

Steps

- Create an Empty Texture Object
- Create a FBO
- Attach renderbuffer for texture image
- Bind FBO
- Render scene
- Detach renderbuffer
- Bind texture
- Render with new texture

Empty Texture Object

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 )

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
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.renderbufferStorage(gl.RENDERBUFFER,
// gl.DEPTH_COMPONENT16, 512, 512);
// Attach color buffer
gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
gl.TEXTURE_2D, texture1, 0);
//gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT,
gl.RENDERBUFFER, renderbuffer);
// check for completeness
var status = gl.checkFramebufferStatus(gl.FRAMEBUFFER);
if (status != gl.FRAMEBUFFER_COMPLETE)
  alert('Frame Buffer Not Complete');
```

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

```glsl
pass through vertex shader:
attribute vec4 vPosition;
void main()
{
  gl_Position = vPosition;
}
```
fragment shader to get a red triangle:
```glsl```
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;
void main() {
  gl_Position = vPosition;
fTexCoord = vTexCoord;
}

// fragment shader
precision mediump float;
uniform sampler2D texture;
void main() {
  gl_FragColor = texture2D(texture, fTexCoord);

First Render (to Texture)

gl.useProgram(program1);
var buffer1 = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer1);
gl.bufferData(gl.ARRAY_BUFFER, flatten(pointsArray), gl.STATIC_DRAW);

var vPosition = gl.getAttribLocation(program1, "vPosition");
gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vPosition);

gl.viewport(0, 0, 64, 64);
gl.clearColor(0.5, 0.5, 0.5, 1.0);
gl.clear(gl.COLOR_BUFFER_BIT);
gl.drawArrays(gl.TRIANGLES, 0, 3);

Set Up Second Render

// Bind to default window system framebuffer
  gl.bindFramebuffer(gl.FRAMEBUFFER, null);
gl.disableVertexAttribArray(vPosition);
gl.useProgram(program2);

// Assume we have already set up a texture object with null texture image
  gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, texture1);

// set up vertex attribute arrays for texture coordinates and rectangle as usual

Data for Second Render

var buffer2 = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer2);
gl.bufferData(gl.ARRAY_BUFFER, new flatten(vertices), gl.STATIC_DRAW);

var vPosition = gl.getAttribLocation(program2, "vPosition");
gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vPosition);

var buffer3 = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer3);
gl.bufferData(gl.ARRAY_BUFFER, flatten(texCoord), gl.STATIC_DRAW);

var vTexCoord = gl.getAttribLocation(program2, "vTexCoord");
gl.vertexAttribPointer(vTexCoord, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vTexCoord);

Render a Quad with Texture

gl.uniform1i(gl.getActiveUniformLocation(program2, "texture"), 0);
gl.viewport(0, 0, 512, 512);
gl clearColor(0.0, 0.0, 1.0, 1.0);
gl.getActiveUniformLocation(program2, "vTexCoord");
gl.vertexAttribPointer(vTexCoord, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vTexCoord);

Dynamic 3D Example

gl.getActiveUniformLocation(program2, "texture"), 0);
gl.getActiveUniformLocation(program2, "vPosition");
gl.getActiveUniformLocation(program2, "vTexCoord");
gl.getActiveUniformLocation(program2, "vTexCoord");
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 render to texture

Buffer Applications

Anti-aliasing

- Aliasing – artifacts produced from inadequate sampling
  - Jagged edges
  - Missing thin objects/features
- Anti-aliasing – removing artifacts via supersampling, 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

Anti-aliasing

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

Area Averaging
### OpenGL Antialiasing

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

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

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

- Note most hardware will automatically antialias

### 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
  - Exponential
  - Gaussian
  - Linear (depth cueing)
- Hard-coded fog deprecated but can recreate

### Fog Functions

- $z^2 \cdot e^{-z}$
- $1 - 0.5z$

### 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 the overhead via rendering the "virtual object"

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 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

- 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

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 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 object
  - setup framebuffer object
  - distribute particles in random locations
Vertex Shader 1

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

Vertex Shader 2

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

Fragment Shader 1

```glsl
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;
}
```

Fragment Shader 2

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

Rendering Loop I

```javascript
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

```javascript
// 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);
// render to display
gl.useProgram(program1);
gl.bindTexture(gl.TEXTURE_2D, texture2);
// 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.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));

// 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;
    }
}

Snapshots

without reading color with reading color

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

```
precision medium float;
uniform int i;
varying vec4 iColor;
void main()
{
    vec4 c[7];
    c[0] = iColor;
    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);
}
```

Setup

```
// Allocates a frame buffer object
framebuffer = gl.createFramebuffer();
gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
// Attach color buffer
gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D, texture, 0);
gl.bindFramebuffer(gl.FRAMEBUFFER, null);
```
```javascript
canvas.addEventListener("mousedown", function() {
  gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
  gl.clear(gl.COLOR_BUFFER_BIT);
  gl.uniform3fv(thetaLoc, theta);
  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;
  gl.readPixels(x, y, 1, 1, gl.RGBA,
    gl.UNSIGNED_BYTE, color);
});
```

```javascript
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");
```

```javascript
// return to default framebuffer
gl.bindFramebuffer(gl.FRAMEBUFFER, null);
// send index 0 to fragment shader
  gl.uniform1i(gl.getUniformLocation(program, "i"), 0);
// normal render
  gl.clear(gl.COLOR_BUFFER_BIT);
  gl.uniform3fv(thetaLoc, theta);
  gl.drawArrays(gl.TRIANGLES, 0, 36);
});
```

**HW8 Suggestions**
- Create an off-screen frame buffer
- 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

**Picking by Selection**
- Possible with render-to-texture
- When mouse clicked do a off screen rendering with new viewing conditions that render only a small area around mouse
- Keep track of what gets rendered to this off screen buffer

**HW8 Suggestions**
- Read color at click point out of the off-screen buffer
- The color will tell you if an object was selected and which one
- Randomly change the diffuse color of the picked object
- Your EventListener should draw to both the off-screen and on-screen buffers