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 through.

Translucent surfaces pass some light through.

translucency = 1 - opacity ($\alpha$)

opaque surface $\alpha = 1$
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

Revert to writing model
Writing Model

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 component source and destination colors

\[ s = [s_r, s_g, s_b, s_\alpha] \]
\[ d = [d_r, d_g, d_b, d_\alpha] \]

Source and destination blending factors

\[ b = [b_r, b_g, b_b, b_\alpha] \]
\[ c = [c_r, c_g, c_b, c_\alpha] \]

Blend as

\[ d' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_\alpha s_\alpha + c_\alpha d_\alpha] \]
OpenGL Blending

Must enable blending and pick source and destination factors

```gl
    gl.enable(gl.BLEND)
    glBlendFunc(gl.source_factor,
                  gl.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 more online
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 \((R_1, G_1, B_1, \alpha_1)\)

Select \texttt{GL\_SRC\_ALPHA} and \texttt{GL\_ONE\_MINUS\_SRC\_ALPHA} as the source and destination blending factors

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

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

See 7E/07/cubet.js
Opaque and Translucent

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 `glDepthMask(GL_FALSE)` which makes depth buffer read-only.

Sort polygons first to remove order dependency.
Compositing and HTML

- In desktop OpenGL, the A component has no effect unless blending is enabled.
- In WebGL, an A other than 1.0 has an effect because WebGL works with the HTML5 Canvas element.
- \( A = 0.5 \) will cut the RGB values by \( \frac{1}{2} \) 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
Line Aliasing
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

![Diagram showing no overlap and overlap cases]
Area Averaging

- Use average area $\alpha_1 + \alpha_2 - \alpha_1 \alpha_2$ as blending factor.
Blending, Antialiasing

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

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

- Note most hardware will automatically antialias
Part 2: Multirendering

- Composite multiple images (7.10.5)
- Image Filtering (convolution) (7.10.7)
  - add shifted and scaled versions of an image

- Whole scene antialiasing (7.10.6)
  - move primitives a little for each render
- Depth of Field (7.10.8)
  - move viewer a little for each render keeping one plane unchanged
- Motion effects (7.10.8)
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.
We note that the only purpose of the geometry is to trigger the execution of the imaging operations in the fragment shader.

Consequently, we can look at what we have done as large matrix operations rather than graphics operations.

Leads to the field of General Purpose Computing with a GPU (GPGPU).
Examples

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

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

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

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

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;
}