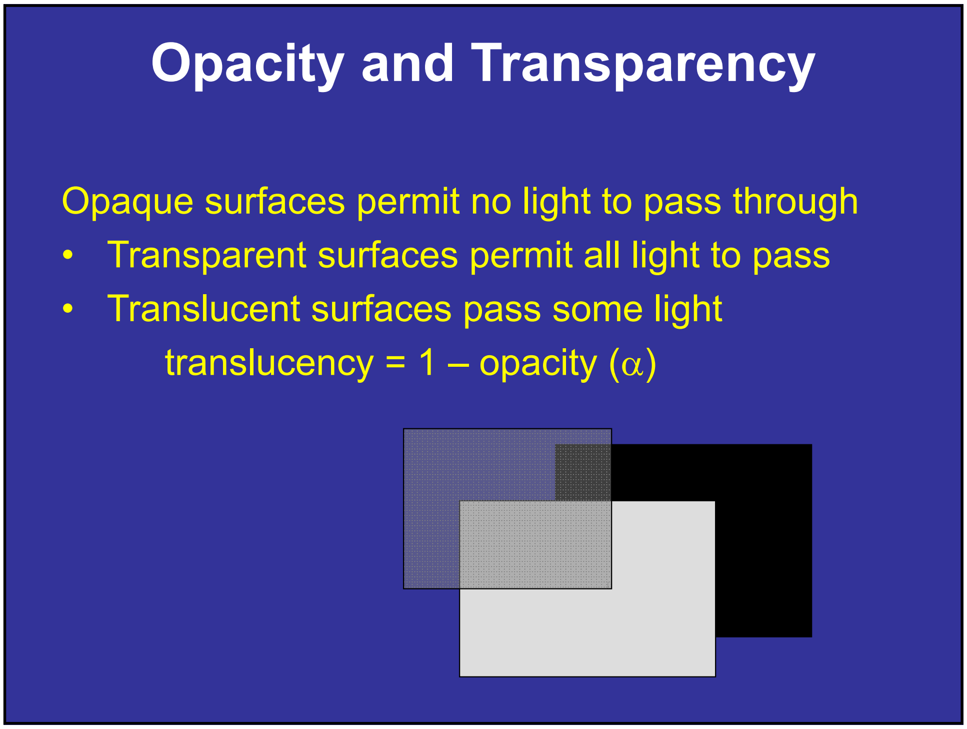
Blending

Blending

Learn to use the A component in RGBA color for

- - Blending for translucent surfaces
- - Compositing images
- - Antialiasing

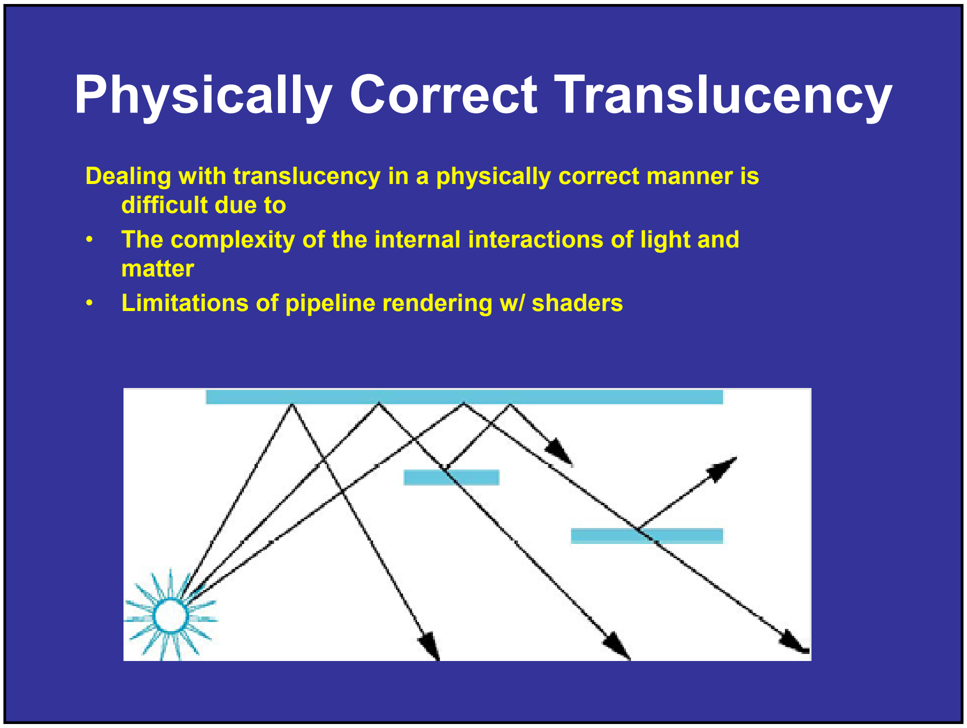


Opacity and Transparency

Opaque surfaces permit no light to pass through

- Transparent surfaces permit all light to pass
- Translucent surfaces pass some light

translucency = 1 – opacity (α)



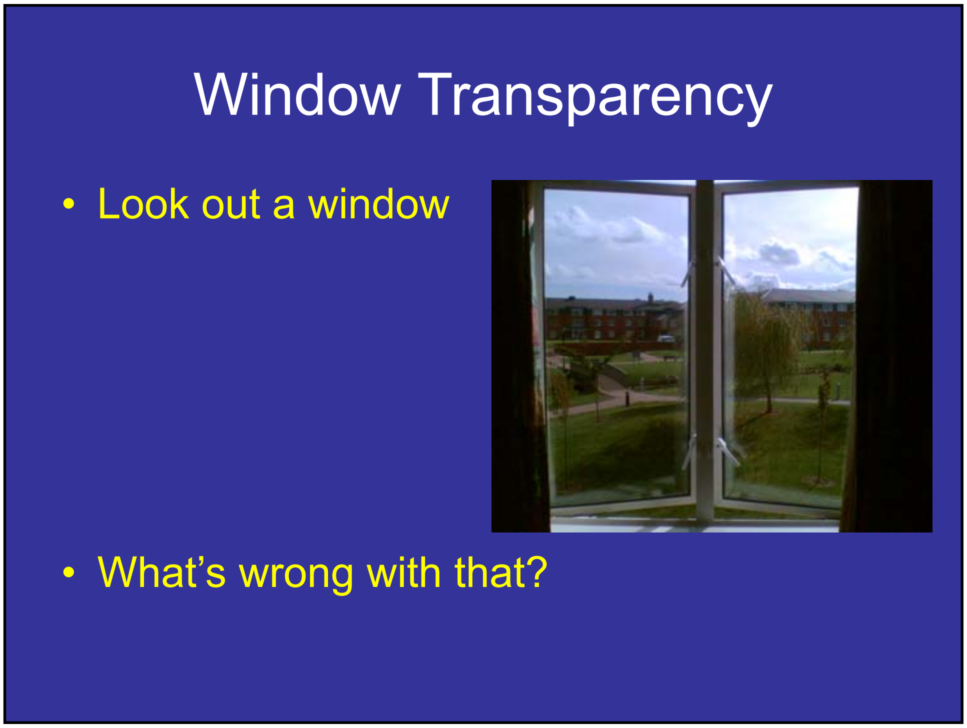
Physically Correct Translucency

Dealing with translucency in a physically correct manner is difficult due to

- The complexity of the internal interactions of light and matter
- Limitations of pipeline rendering w/ shaders

Window Transparency

- Look out a window

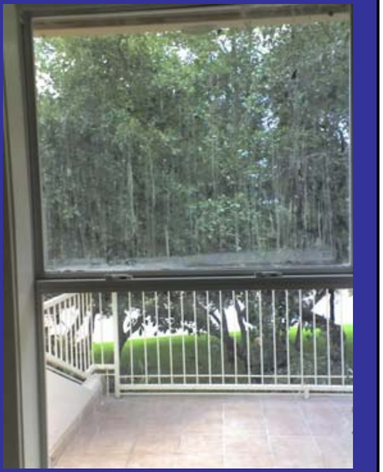


Window Transparency

- Look out a window
- What’s wrong with that?

Window Transparency

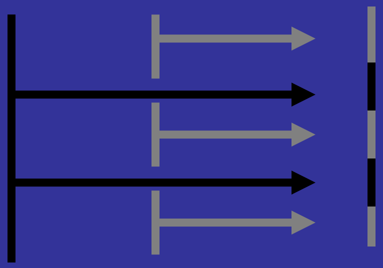
- Look out a window

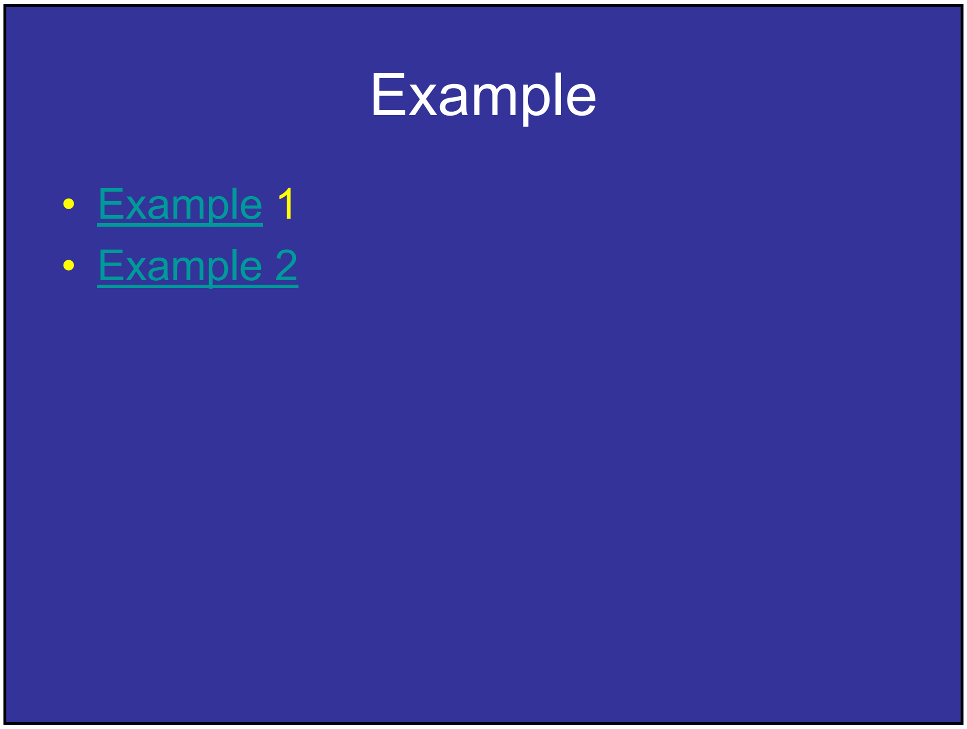


- What’s wrong with that?

Screen Door Transparency

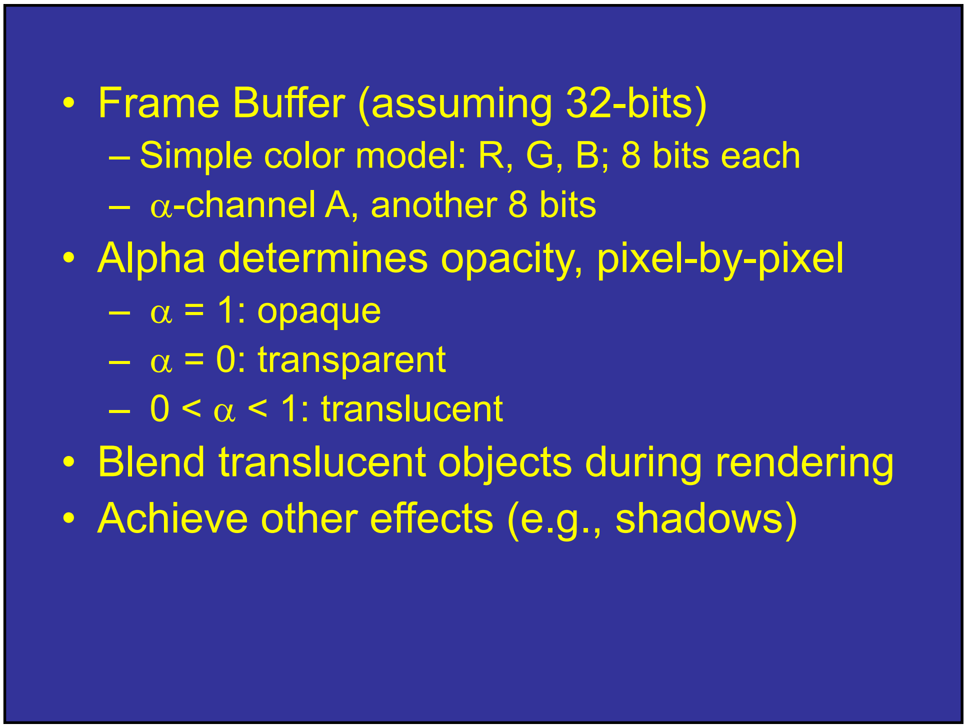
- glEnableGL_POLYGON_STIPPLE(GL_POLYGON_STIPPLE)





Example

- Example 1
- Example 2



- Frame Buffer (assuming 32-bits)
  - Simple color model: R, G, B; 8 bits each
  - α-channel A, another 8 bits
- Alpha determines opacity, pixel-by-pixel
  - $\alpha = 1$: opaque
  - $\alpha = 0$: transparent
  - $0 < \alpha < 1$: translucent
- Blend translucent objects during rendering
- Achieve other effects (e.g., shadows)

Compositing

- Back to Front

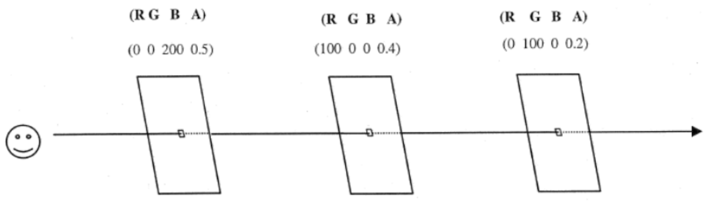
$$C_{out} = (1-\alpha_c)C_{in} + \alpha_c C_c$$

- Front to Back

$$C_{out} = C_{in} + C_c \alpha_c (1-\alpha_{in})$$

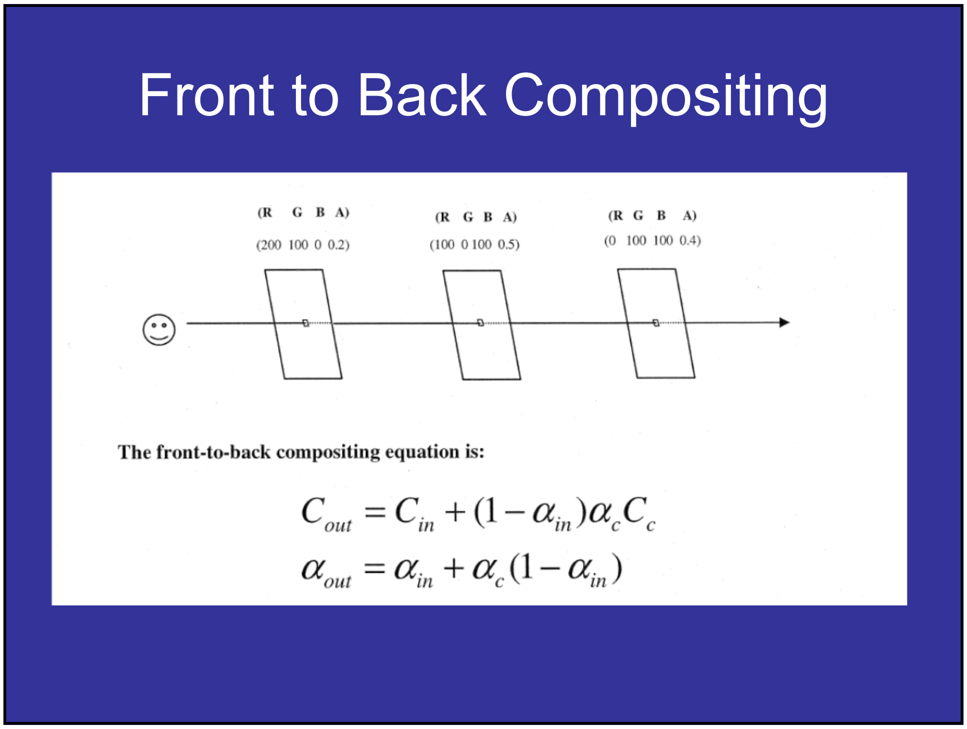
$$\alpha_{out} = \alpha_{in} + \alpha_c (1-\alpha_{in})$$

Back to Front Compositing



The back-to-front compositing equation is:

$$C_{out} = (1-\alpha_c)C_{in} + \alpha_c C_c$$

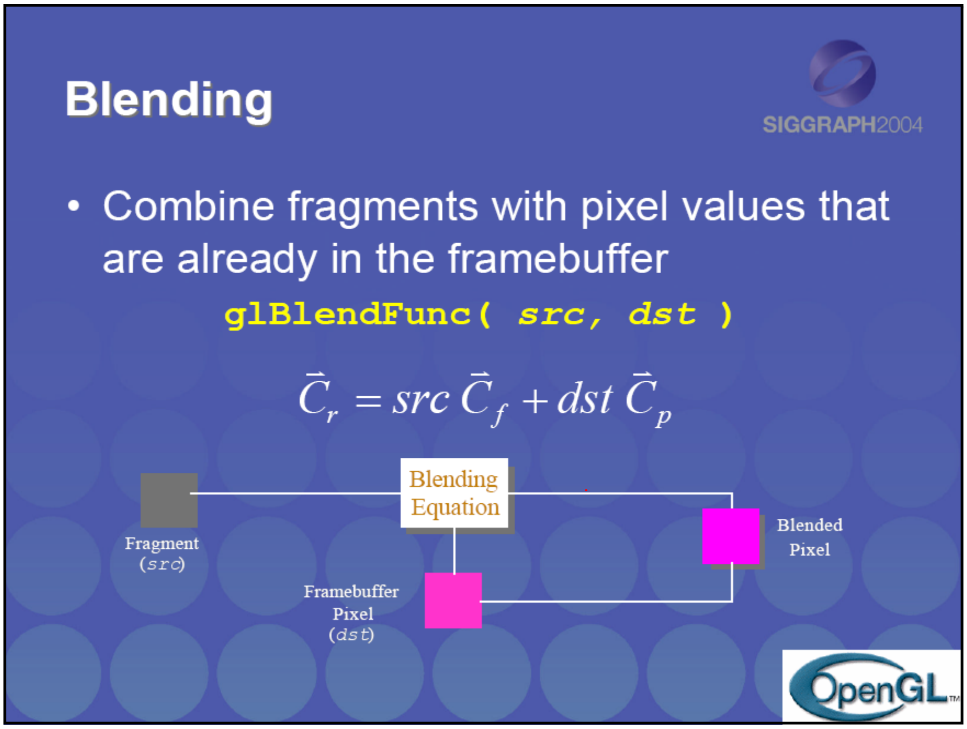


Front to Back Compositing

The front-to-back compositing equation is:

$$C_{out} = C_{in} + (1 - \alpha_{in})\alpha_c C_c$$

$$\alpha_{out} = \alpha_{in} + \alpha_c (1 - \alpha_{in})$$

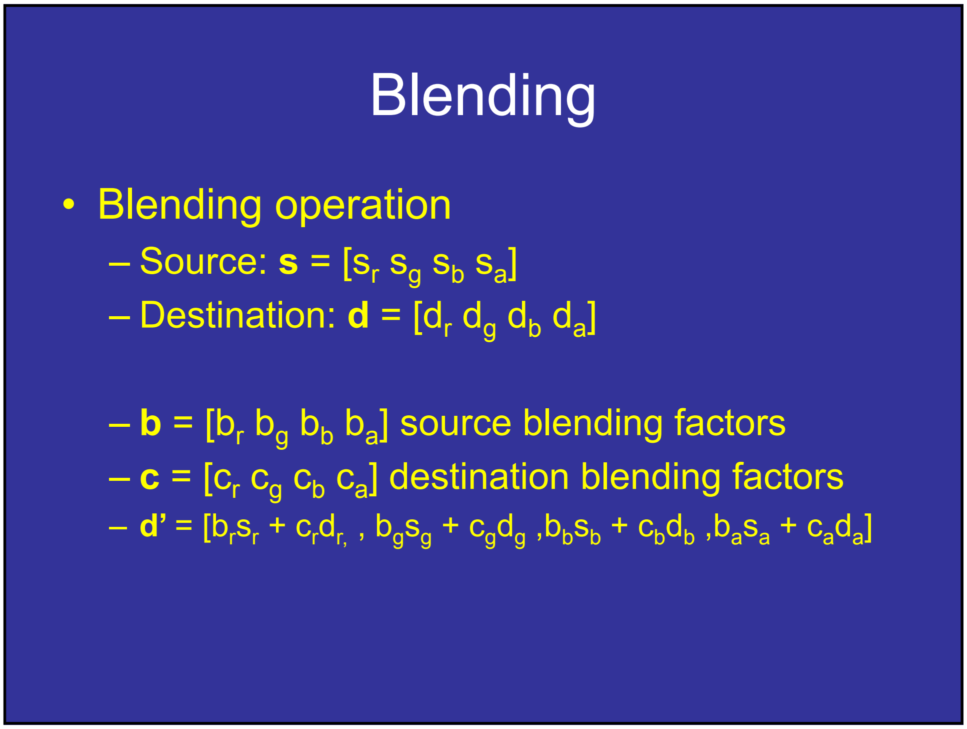


Blending

- Combine fragments with pixel values that are already in the framebuffer

```
glBlendFunc( src, dst )
```

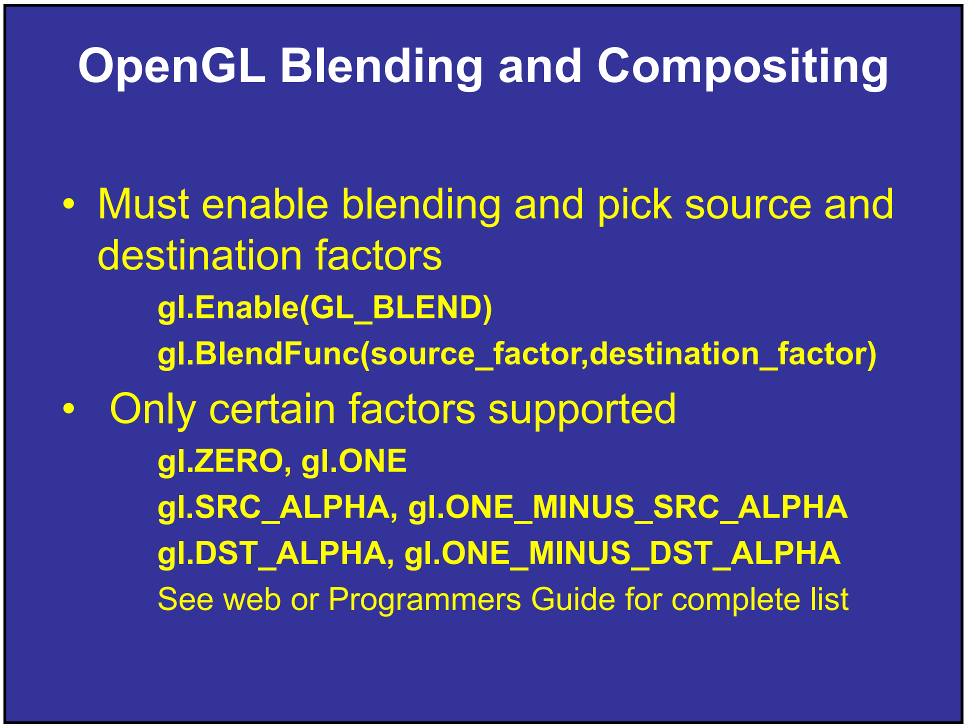
$$\vec{C}_r = src\,\vec{C}_f + dst\,\vec{C}_p$$



Blending

- Blending operation
  - Source: $\mathbf{s} = [s_r\ s_g\ s_b\ s_a]$
  - Destination: $\mathbf{d} = [d_r\ d_g\ d_b\ d_a]$

  - $\mathbf{b} = [b_r\ b_g\ b_b\ b_a]$ source blending factors
  - $\mathbf{c} = [c_r\ c_g\ c_b\ c_a]$ destination blending factors
  - $\mathbf{d'} = [b_r s_r + c_r d_{r,}\ ,\ b_g s_g + c_g d_g\ , b_b s_b + c_b d_b\ , b_a s_a + c_a d_a]$



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 web or Programmers Guide for complete list

Blending Factors

/* BlendingFactorSrc */

gl.ZERO

gl.ONE

gl.DST_COLOR

gl.SRC_COLOR

gl.ONE_MINUS_DST_COLOR

gl.ONE_MINUS_SRC_COLOR

gl.SRC_ALPHA

gl.ONE_MINUS_SRC_ALPHA

gl.DST_ALPHA

gl.ONE_MINUS_DST_ALPHA

gl.SRC_ALPHA_SATURATE

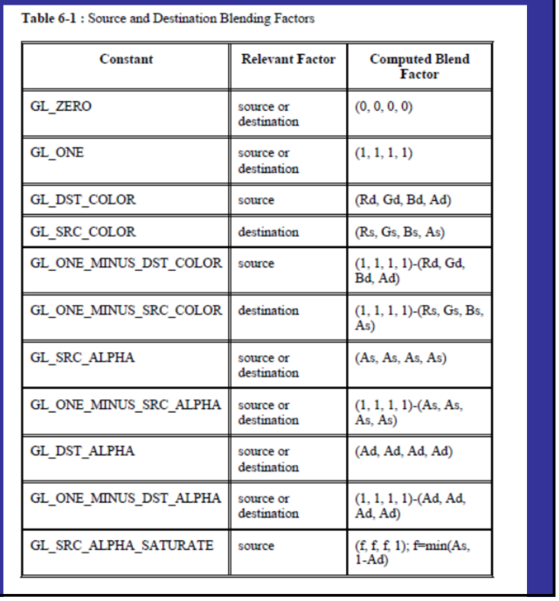


Table 6-1 : Source and Destination Blending Factors

Constant	Relevant Factor	Computed Blend Factor
GL_ZERO	source or destination	(0, 0, 0, 0)
GL_ONE	source or destination	(1, 1, 1, 1)
GL_DST_COLOR	source	(Rd, Gd, Bd, Ad)
GL_SRC_COLOR	destination	(Rs, Gs, Bs, As)
GL_ONE_MINUS_DST_COLOR	source	(1, 1, 1, 1)-(Rd, Gd, Bd, Ad)
GL_ONE_MINUS_SRC_COLOR	destination	(1, 1, 1, 1)-(Rs, Gs, Bs, As)
GL_SRC_ALPHA	source or destination	(As, As, As, As)
GL_ONE_MINUS_SRC_ALPHA	source or destination	(1, 1, 1, 1)-(As, As, As, As)
GL_DST_ALPHA	source or destination	(Ad, Ad, Ad, Ad)
GL_ONE_MINUS_DST_ALPHA	source or destination	(1, 1, 1, 1)-(Ad, Ad, Ad, Ad)
GL_SRC_ALPHA_SATURATE	source	(f, f, f, 1); f=min(As, 1-Ad)

gl.blendEquation(…)

- gl. FUNC_ADD
- gl. BLEND_EQUATION
- gl. BLEND_EQUATION_RGB
  - /* same as BLEND_EQUATION */
- gl. BLEND_EQUATION_ALPHA
- /* BlendSubtract */
- gl. FUNC_SUBTRACT
- gl. FUNC_REVERSE_SUBTRACT

Blending Example

Given the following:

Fragment: (R,G,B,A)=
(0.0, 0.0, 1.0, 0.25)

Framebuffer: (0.0, 1.0, 0.0, 0.75)

Assume blending is enabled and the state is correctly setup.

What is the result of the following:

RGBA blend with:

gl.blendFunc(
gl.ONE_MINUS_SRC_COLOR
, gl.SRC_ALPHA)?

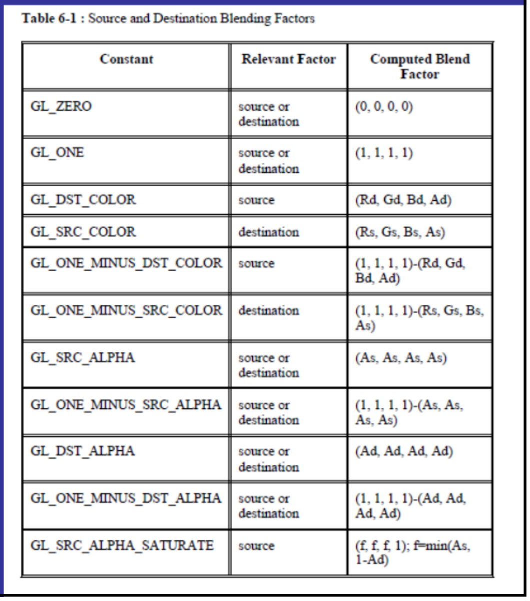


Table 6-1 : Source and Destination Blending Factors

Constant	Relevant Factor	Computed Blend Factor
GL_ZERO	source or destination	(0, 0, 0, 0)
GL_ONE	source or destination	(1, 1, 1, 1)
GL_DST_COLOR	source	(Rd, Gd, Bd, Ad)
GL_SRC_COLOR	destination	(Rs, Gs, Bs, As)
GL_ONE_MINUS_DST_COLOR	source	(1, 1, 1, 1)-(Rd, Gd, Bd, Ad)
GL_ONE_MINUS_SRC_COLOR	destination	(1, 1, 1, 1)-(Rs, Gs, Bs, As)
GL_SRC_ALPHA	source or destination	(As, As, As, As)
GL_ONE_MINUS_SRC_ALPHA	source or destination	(1, 1, 1, 1)-(As, As, As, As)
GL_DST_ALPHA	source or destination	(Ad, Ad, Ad, Ad)
GL_ONE_MINUS_DST_ALPHA	source or destination	(1, 1, 1, 1)-(Ad, Ad, Ad, Ad)
GL_SRC_ALPHA_SATURATE	source	(f, f, f, 1); f=min(As, 1-Ad)

Sorting

Correct

Magenta
Yellow
Gray
Cyan

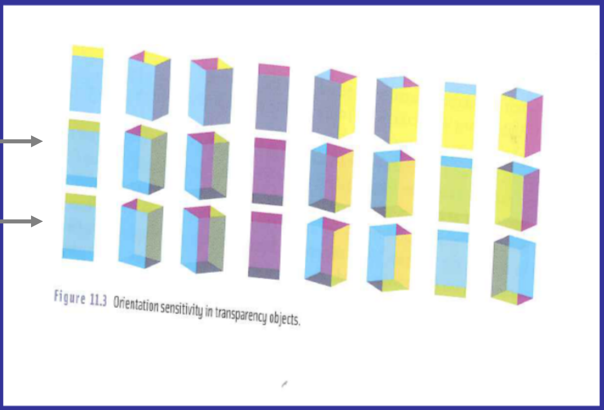
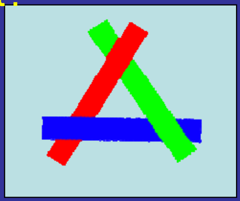


Figure 11.3 Orientation sensitivity in transparency objects.

Sorting

- General Solution?
  - Just sort polygons
    - Which Space?
  - What About?



  - Depth Peeling

Blending Errors

- Operations are not commutative
- Operations are not idempotent
- Interaction with hidden-surface removal
  - Polygon behind opaque one should be hidden
  - Translucent in front of others should be composited
  - Show Demo of the problem
  - Solution?

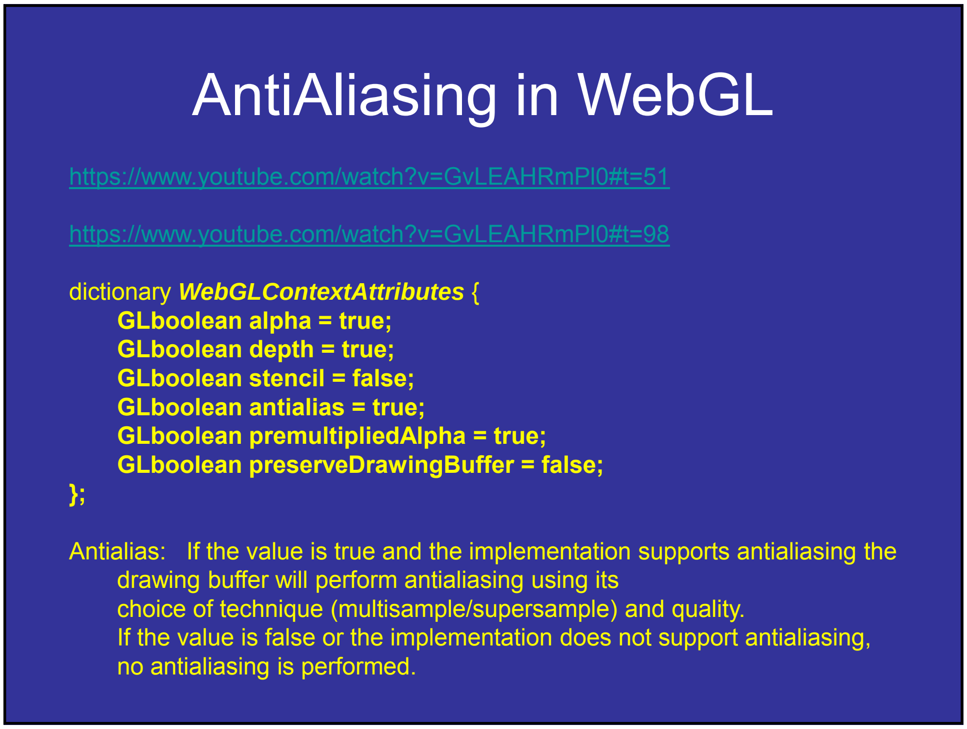
Blending Errors

- Interaction with hidden-surface removal
  - Polygon behind opaque one should be hidden
  - Translucent in front of others should be composited
  - Solution?
    - Two passes using *alpha testing* (gl.AlphaFunc): 1st pass
    - alpha=1 accepted, and 2nd pass alpha<1 accepted
    - make z-buffer read-only for translucent polygons (alpha<1) with **gl.depthMask(gl.FALSE);**
  - Demo

AntiAliasing in WebGL

The optional WebGLContextAttributes object may be used to change whether or not the buffers are defined. It can also be used to define whether the color buffer will include an alpha channel. If defined, the alpha channel is used by the HTML compositor to combine the color buffer with the rest of the page. The WebGLContextAttributes object is only used on the first call to getContext. No facility is provided to change the attributes of the drawing buffer after its creation.

The depth, stencil and antialias attributes, when set to true, are requests, not requirements. The WebGL implementation should make a best effort to honor them. When any of these attributes is set to false, however, the WebGL implementation must not provide the associated functionality. Combinations of attributes not supported by the WebGL implementation or graphics hardware shall not cause a failure to create a WebGLRenderingContext. The actual context parameters are set to the attributes of the created drawing buffer. The alpha, premultipliedAlpha and preserveDrawingBuffer attributes must be obeyed by the WebGL implementation.



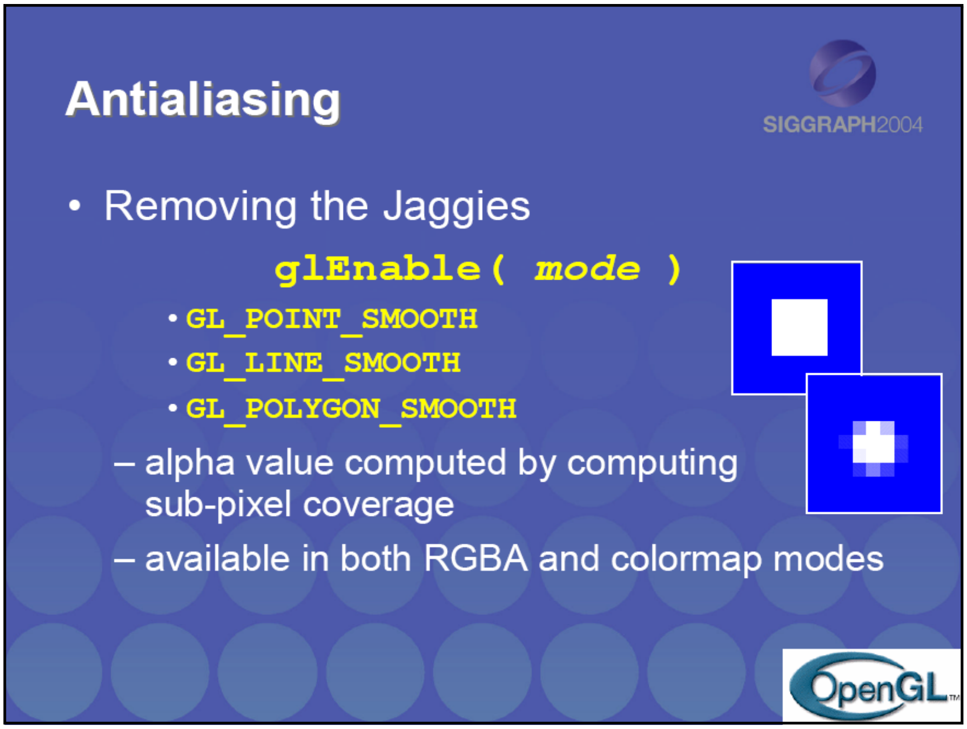
AntiAliasing in WebGL

https://www.youtube.com/watch?v=GvLEAHRmPl0#t=51

https://www.youtube.com/watch?v=GvLEAHRmPl0#t=98

```
dictionary WebGLContextAttributes {
    GLboolean alpha = true;
    GLboolean depth = true;
    GLboolean stencil = false;
    GLboolean antialias = true;
    GLboolean premultipliedAlpha = true;
    GLboolean preserveDrawingBuffer = false;
};
```

Antialias: If the value is true and the implementation supports antialiasing the drawing buffer will perform antialiasing using its choice of technique (multisample/supersample) and quality. If the value is false or the implementation does not support antialiasing, no antialiasing is performed.



Antialiasing

- Removing the Jaggies

  glEnable(*mode*)

  - **GL_POINT_SMOOTH**
  - **GL_LINE_SMOOTH**
  - **GL_POLYGON_SMOOTH**

  – alpha value computed by computing sub-pixel coverage

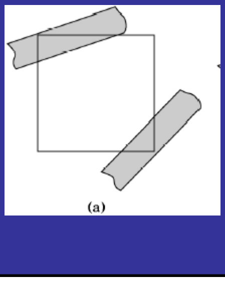
  – available in both RGBA and colormap modes

Antialiasing Revisited

- Single-polygon case first
- Set α value of each pixel to covered fraction
- Use destination factor of "1 – α"
- Use source factor of "α"
- This will blend background with foreground
- Overlaps can lead to blending errors

Antialiasing with Multiple Polygons

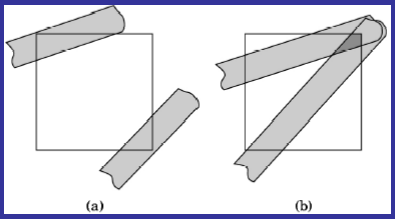
- Initially, background color $\mathbf{C}_0$, $a_0 = 0$
- Render first polygon; color C_1 fraction α_1
  - $\mathbf{C}_d = (1 - \alpha_1)\mathbf{C}_0 + \alpha_1\mathbf{C}_1$
  - $\alpha_d = \alpha_1$
- Render second polygon; assume fraction α_2
- If no overlap (case a), then
  - $\mathbf{C'}_d = (1 - \alpha_2)\mathbf{C}_d + \alpha_2\mathbf{C}_2$
  - $\alpha'_d = \alpha_1 + \alpha_2$

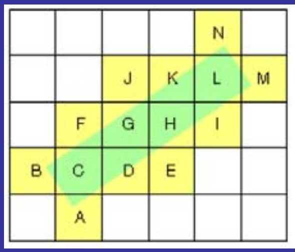


(a)

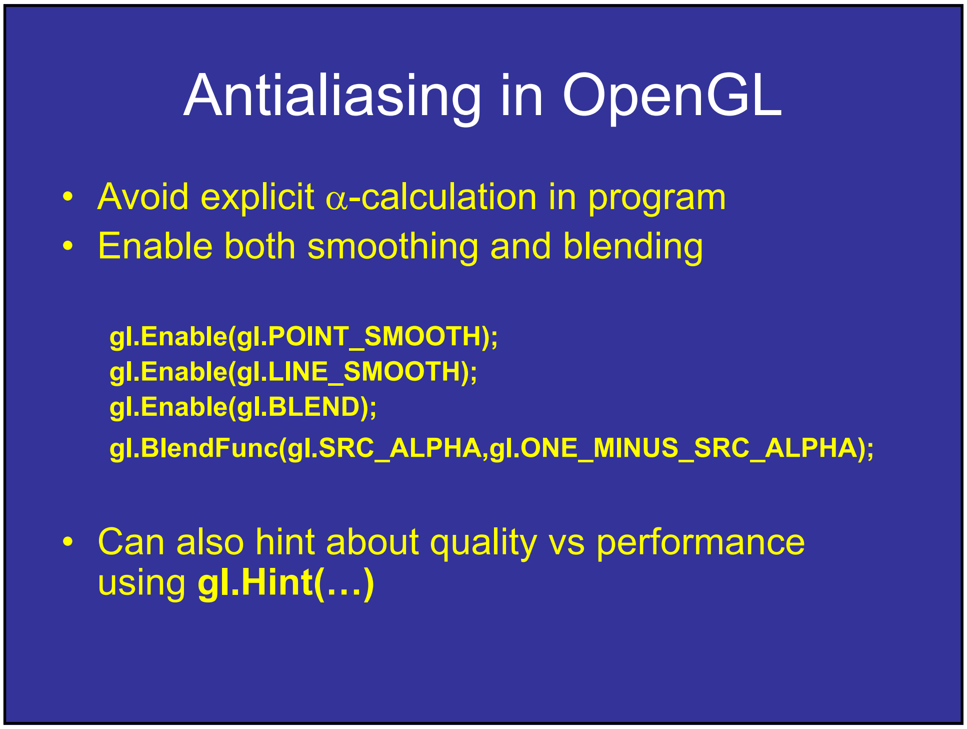
Antialiasing with Multiple Polygons

- Now assume overlap (case b)
- Average overlap is a_1a_2
- So $a_d = a_1 + a_2 – a_1a_2$
- Make front/back decision for color as usual





A	.040510
B	.040510
C	.878469
D	.434259
E	.007639
F	.141435
G	.759952
H	.759952
I	.141435
J	.007639
K	.434259
L	.878469
M	.040510
N	.040510



Antialiasing in OpenGL

- Avoid explicit α-calculation in program
- Enable both smoothing and blending

```
gl.Enable(gl.POINT_SMOOTH);
gl.Enable(gl.LINE_SMOOTH);
gl.Enable(gl.BLEND);
gl.BlendFunc(gl.SRC_ALPHA,gl.ONE_MINUS_SRC_ALPHA);
```

- Can also hint about quality vs performance using **gl.Hint(…)**