\[ I_{\lambda} = \alpha I_{\lambda 1} + (1 - \alpha) I_{\lambda 2} \]
OPENGL BLENDING

\[ I_\lambda = B_1 I_{\lambda 1} + B_2 I_{\lambda 2} \]

• Let the source and destination blending factors be \((S_r, S_g, S_b, S_a)\) and \((D_r, D_g, D_b, D_a)\).

• The RGBA values of the source and destination is indicated with a subscript of \(s\) or \(d\).

• The final, blended RGBA values are given by

\[
(R_s S_r + R_d D_r, G_s S_g + G_d D_g, B_s S_b + B_d D_b, A_s S_a + A_d D_a)
\]
\[ I_\lambda = B_1 I_{\lambda 1} + B_2 I_{\lambda 2} \]

\[(R_s S_r + R_d D_r, G_s S_g + G_d D_g, B_s S_b + B_d D_b, A_s S_a + A_d D_a)\]

1) Each component is eventually clamped to \([0,1]\); glEnable(GL_BLEND);

2) void glBlendFunc(GLenum sfactor, GLenum dfactor);

3) Color values in the fragment (the source) are combined with those already stored in the framebuffer (the destination).

4) sfactor indicates how to compute a source blending factor; dfactor indicates how to compute a destination blending factor

\[ \text{GL\_SRC\_ALPHA} \quad \text{s. or d.} \quad (A_s, A_s, A_s, A_s) \]
\[ \text{GL\_ONE\_MINUS\_SRC\_ALPHA} \quad \text{s. or d.} \quad (1,1,1,1) - (A_s, A_s, A_s, A_s) \]

\textbf{J4_1 Blending}
void `glBlendFunc(GLenum sfactor, GLenum dfactor);`

<table>
<thead>
<tr>
<th>Constant Value</th>
<th>Relevant Factor</th>
<th>Computed Blend Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>GL_ZERO</td>
<td>Source or destination</td>
<td>(0,0,0,0)</td>
</tr>
<tr>
<td>GL_ONE</td>
<td>Source or destination</td>
<td>(1,1,1,1)</td>
</tr>
<tr>
<td>GL_DST_COLOR</td>
<td>Source</td>
<td>(Rd,Gd,Bd,Ad)</td>
</tr>
<tr>
<td>GL_SRC_COLOR</td>
<td>destination</td>
<td>(Rs,Gs,Bs,As)</td>
</tr>
<tr>
<td>GL_ONE_MINUS_DST_COLOR</td>
<td>source</td>
<td>(1,1,1,1)-(Rd,Gd,Bd,Ad)</td>
</tr>
<tr>
<td>GL_ONE_MINUS_SRC_COLOR</td>
<td>destination</td>
<td>(1,1,1,1)-(Rs,Gs,Bs,As)</td>
</tr>
<tr>
<td>GL_SRC_ALPHA</td>
<td>Source or destination</td>
<td>(As,As,As,As)</td>
</tr>
<tr>
<td>GL_ONE_MINUS_SRC_ALPHA</td>
<td>Source or destination</td>
<td>(1,1,1,1)-(As,As,As,As)</td>
</tr>
<tr>
<td>GL_DST_ALPHA</td>
<td>Source or destination</td>
<td>(Ad,Ad,Ad,Ad)</td>
</tr>
<tr>
<td>GL_ONE_MINUS_DST_ALPHA</td>
<td>Source or destination</td>
<td>(1,1,1,1)-(As,As,As,As)</td>
</tr>
<tr>
<td>GL_SRC_ALPHA_SATURATE</td>
<td>Source</td>
<td>(f,f,f,1): f-min(As,1-Ad)</td>
</tr>
</tbody>
</table>
Order Dependency

• Is this image correct?
  – Probably not.

  – Polygons rendered in the order they pass down the pipeline.

  – Blending functions are order dependent!
Transparency with the Depth Buffer

• If an opaque object hides any object, you want the depth buffer
• If the translucent object is closer, however, you want to blend it with the opaque object.

• draw opaque objects first with normal depth-buffer operation; then draw translucent object with the depth buffer read-only

• glDepthMask(GL_FALSE ); the buffer becomes read-only, whereas GL_TRUE restores the normal, writable operation.

J4_2_Opaque  J4_3_TransLight
Antialiasing

- Can enable separately for points, lines, or polygons.
  
  ```c
  glEnable(GL_POINT_SMOOTH);
  glEnable(GL_LINE_SMOOTH);
  glEnable(GL_POLYGON_SMOOTH);
  
  glEnable(GL_BLEND);
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  ```
Line antialiasing: `glEnable(GL_LINE_SMOOTH);`

- OpenGL calculates a **coverage** value for each **fragment** based on the pixel square.

  ![Diagram of coverage values]

  - In RGBA mode, OpenGL multiplies the fragment’s alpha by its coverage. You can then use the alpha to **blend** the fragment with the pixel already in the framebuffer.
• after the rasterization stage (including texturing), the data are not yet pixel, but are fragments
  – Fragment is all the data associated with a pixel, including coordinates, color, depth and texture coordinates.
• `glEnable(GL_BLEND);`

• In RGBA mode, the blending factors you most likely want to use are `GL_SRC_ALPHA` (source) and `GL_ONE_MINUS_SRC_ALPHA` (destination).

• For points, lines, and polygons:
  ```c
  glEnable(GL_POINT_SMOOTH);
  glEnable(GL_LINE_SMOOTH);
  glEnable(GL_POLYGON_SMOOTH);
  ```

• For polygons, we need to turn off depth buffer and sort the polygons according to the depth, which is quite cumbersome (a good topic to study)
void glHint(GLenum target, GLenum hint);

The target parameter indicates which behavior is to be controlled:
GL_POINT_SMOOTH_HINT,
GL_LINE_SMOOTH_HINT,
GL_POLYGON_SMOOTH_HINT, GL_FOG_HINT,
GL_PERSPECTIVE_CORRECTION_HINT

The hint parameter can be GL_FASTEST to indicate that the most efficient option should be chosen, GL_NICEST to indicate the highest-quality option, or GL_DONT_CARE. The interpretation of hints is implementation-dependent.
In RGBA mode, the **fog factor** \( f \) is used to calculate the final fogged color:

\[
I_\lambda = f \cdot I_{\lambda 1} + (1-f)I_{\lambda f},
\]

where \( I_{\lambda 1} \) represents the incoming fragment’s RGBA values and \( I_{\lambda f} \) is assigned with GL_FOG_COLOR.

\[
f = e^{-(\text{density} \cdot z)} \quad \text{(GL_EXP)}
\]
\[
f = e^{-(\text{density} \cdot z)^2} \quad \text{(GL_EXP2)}
\]
\[
f = \frac{\text{end} - z}{\text{end} - \text{start}} \quad \text{(GL_LINEAR)}
\]
\[ f = e^{-(density \cdot z)} \quad (GL_{\text{EXP}}) \]

\[ f = e^{-(density \cdot z)^2} \quad (GL_{\text{EXP2}}) \]

\[ f = \frac{end - z}{end - start} \quad (GL_{\text{LINEAR}}) \]
OpenGL Fog in RGBA Mode

• first glEnable(GL_FOG);
• Then call glFog{if}{v}(GLenum pname, TYPE param);
  – If pname is GL_FOG_COLOR, param is the color value GLfloat fcolor[4] = {……};
  – If pname is GL_FOG_MODE, param is either GL_EXP (the default), GL_EXP2, or GL_LINEAR.
  – If pname is GL_FOG_DENSITY, GL_FOG_START, or GL_FOG_END, then param is a value for density, start, or end in the equations.

Copyright @ 2002 by Jim X. Chen:
jchen@cs.gmu.edu
Accumulation Buffer

• Compositing and blending are limited by resolution of the frame buffer:
  – Typically 8 bits per color component.

• The *accumulation buffer* is a high resolution buffer (16 or more bits per component) that avoids this problem.

• Write into it or read from it with a scale factor.

• Slower than direct compositing into the frame buffer.
glClear(GL_ACCUM_BUFFER_BIT);
// just like clearing color or depth buffers

glAccum(GL_ACCUM, frac);
// the current color in the color buffer will be multiplied
// by frac and then added to the Accumulation Buffer

glAccum(GL_RETURN, 1.0);
// copies the accumulation buffer to the current color buffer

*Note: There are other arguments you can pass to glAccum: GL_LOAD, GL_ADD, GL_RETURN*
Applications

• Whole scene anti-aliasing (stochastic raytracing)
• Motion effects (blur)
• Depth-of-field.
HW

• Integrate transparency into your environment such as your objects are visible inside a transparent volume. It is also better that the front of the transparent volume is totally transparent.