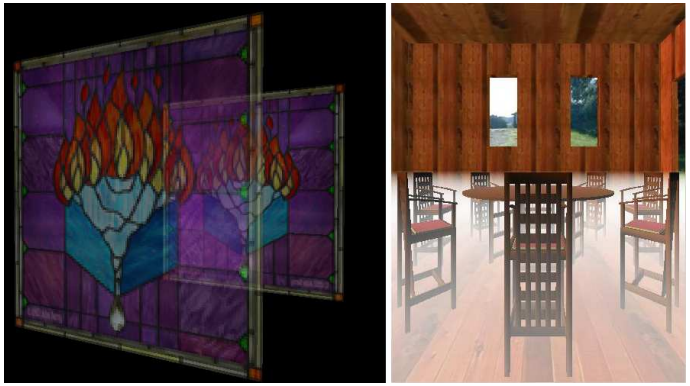
Image Compositions

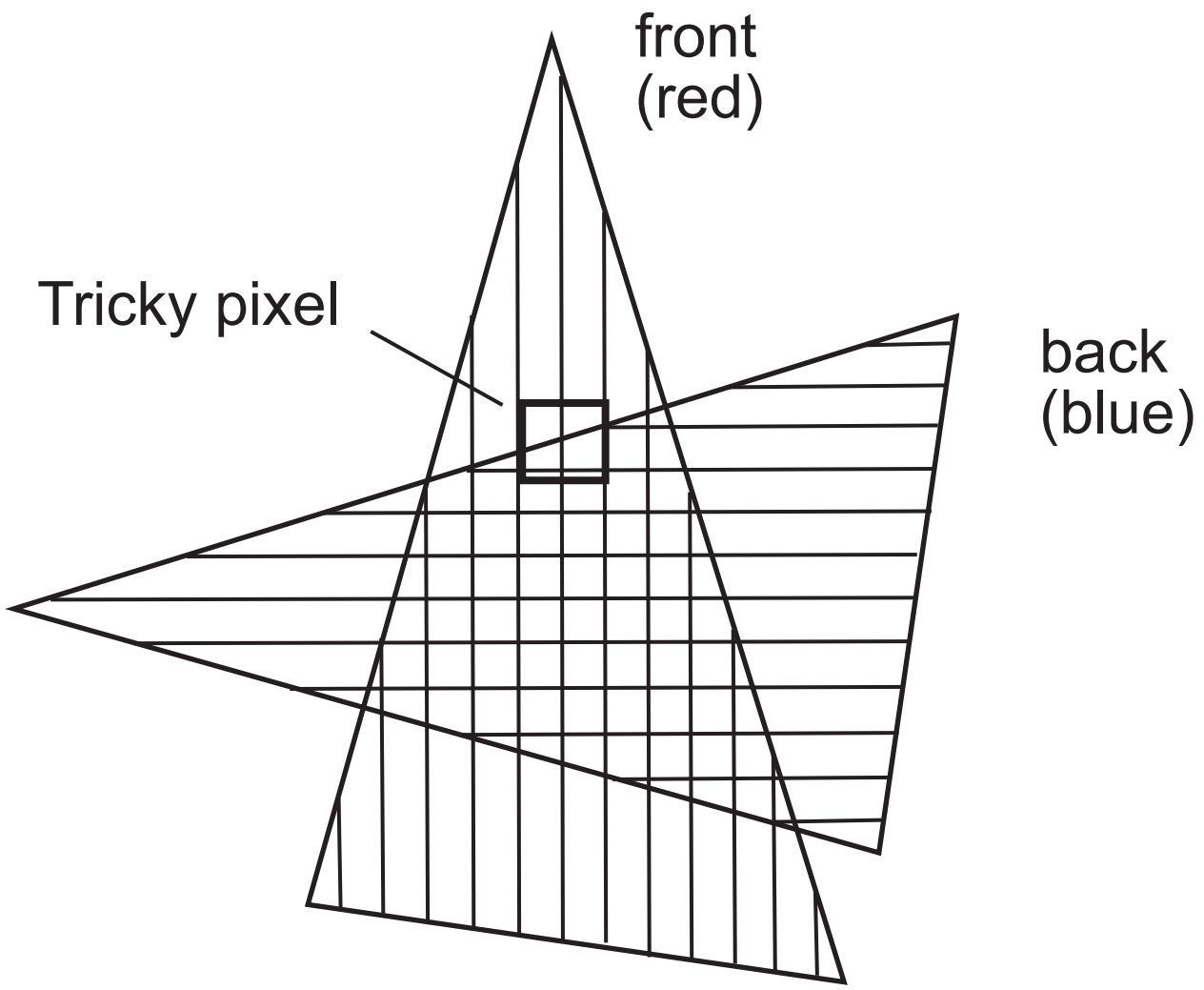
- Compositing of images — combining images to create new images.
- With compositing
  1. one portion of the image needs alteration, the whole image does not need to be regenerated
  2. some portions of an image are not rendered but have been optically scanned into memory instead, compositing may be the only way to incorporate them in the image
  3. special effects (fog, transparency etc)
  4. meta-buffer (parallel image rendering)



α-Channel Compositing

A pixel's value in the composited image is taken from the background image unless the foreground image has a nontransparent value at that point, in which case the value is taken from the foreground image. A *blending* of two images, the resulting pixel value is a linear combination of the value of the two component pixels.

Consider a pixel lying on the edge of the back red polygon but inside the front (transparent) blue polygon? If we color it red only, aliasing artifacts will result. If we know that the back polygon covers 70 percent of the pixel, we can make the composited pixel 70 percent red and 30 percent blue and get a much more attractive result.



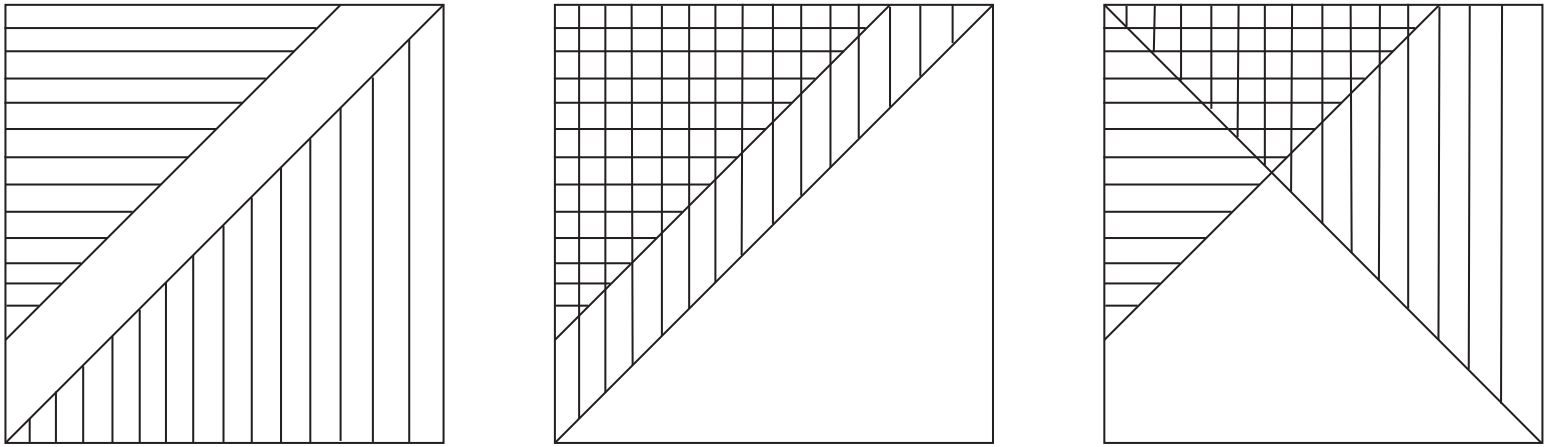
Compositing operations near an edge: How do we color the pixel?

The color associated with each pixel in the image is given an α *value* representing the *coverage* of the pixel. For an image that is to become the foreground element of a composited image, many of the pixels are registered as having coverage zero (they are transparent); the remainder, which constitue the important content of the foreground image, have larger coverage values (usually one).

We need tha α information at each pixel of the images being composited. Assume that, along with the RGB values of an image, we also have an α value encoding the coverage of each pixel. This collection of α values is often called the α *channel*.

How do α values combine? Suppose we have a red polygon covering one-third of the area of a pixel, and a blue polygon that, taken separately, covers one-half of the area of the pixel. How much of the first polygon is covered by the second? How do we compute the color of the pixel resulting from 60–40 blend of these 2 pixels?

$$0.6(\frac{1}{3})(1,0,0) + 0.4(\frac{1}{2})(0,0,1) = (0.2,0,0.2)$$



The ways in which polygons can overlap within a pixel. In image composition, the first two cases are considered exceptional; the third is treated as the rule.

Areas and possible colors for regions of overlap in compositing

Region	Area	Possible colors
neither	$(1-\alpha_A)(1-\alpha_B)$	0
A alone	$\alpha_A(1-\alpha_B)$	0, A
B alone	$\alpha_B(1-\alpha_A)$	0, B
both	$\alpha_a\alpha_b$	0, A, B

Whenever we combine a pixels, we use the product of the α value and the color of each pixel. This suggests that, when we store an image (within a compositing program), we should store not (R, G, B, α), but rather (αR, αG, αB, α) for each pixel, thus saving ourselves the trouble of performing the multiplications each time. When we refer to an RGBα value for a pixel, we mean exactly this.

Generating α Values with Fill Mechanisms

What if an image is produced by scanning of a photograph or is provided by some other source lacking this information? Can it still be composited? If we can generate an α channel for the image, we can use that channel for compositing. Even if we merely assign an α value of zero to black pixels and an α value of 1 to all others, we can use the preceding algorithms, although ragged-edge problems will generally arise.

Blending and Compositing in OpenGL

The mechanics of blending in OpenGL are straightforward. We enable blending by

```
glEnable(GL_BLEND);
glBlendFunc(GLenum sfactor, GLenum dfactor);
```

OpenGL has a number of blending factors defined, including the values 1 (`GL_ONE`) and 0 (`GL_ZERO`), the source α and $1 - \alpha$ (`GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA`), and the destination α and $1 - \alpha$ (`GL_DST_ALPHA` and `GL_ONE_MINUS_DST_ALPHA`). The application program specifies the desired operations and then uses RGBA color.

The major difficulties with compositing are that the order in which we render the polygons affects the image. For example, many applications use the source α as the source blending factor and $1 - \alpha$ for the destination factor. The resulting color and opacity are

$$(R_{d'}, G_{d'}, B_{d'}, \alpha_{d'}) = (\alpha_s R_s + (1 - \alpha_s) R_d, \alpha_s G_s + (1 - \alpha_s) G_d, \\ \alpha_s B_s + (1 - \alpha_s) B_d, \alpha_s \alpha_s + (1 - \alpha_s) \alpha_d).$$

This formula ensures that both transparent and opaque polygons are handled correctly and that neither colors nor opacities can saturate. However, the color and α values depend on the order in which the polygons are rendered.

A more subtle but visibly apparent problem occurs when we combine opaque and translucent objects in a scene. In a scene with both opaque and transparent polygons, any polygon behind an opaque polygon should not be rendered, but translucent polygons in front of opaque polygons should be composited but their depth information not registered. There is a simple solution to this problem that does not require the application program to order the polygons. We can enable hidden-surface removal as usual and can make the z-buffer read-only for any polygon that is translucent. We do so by

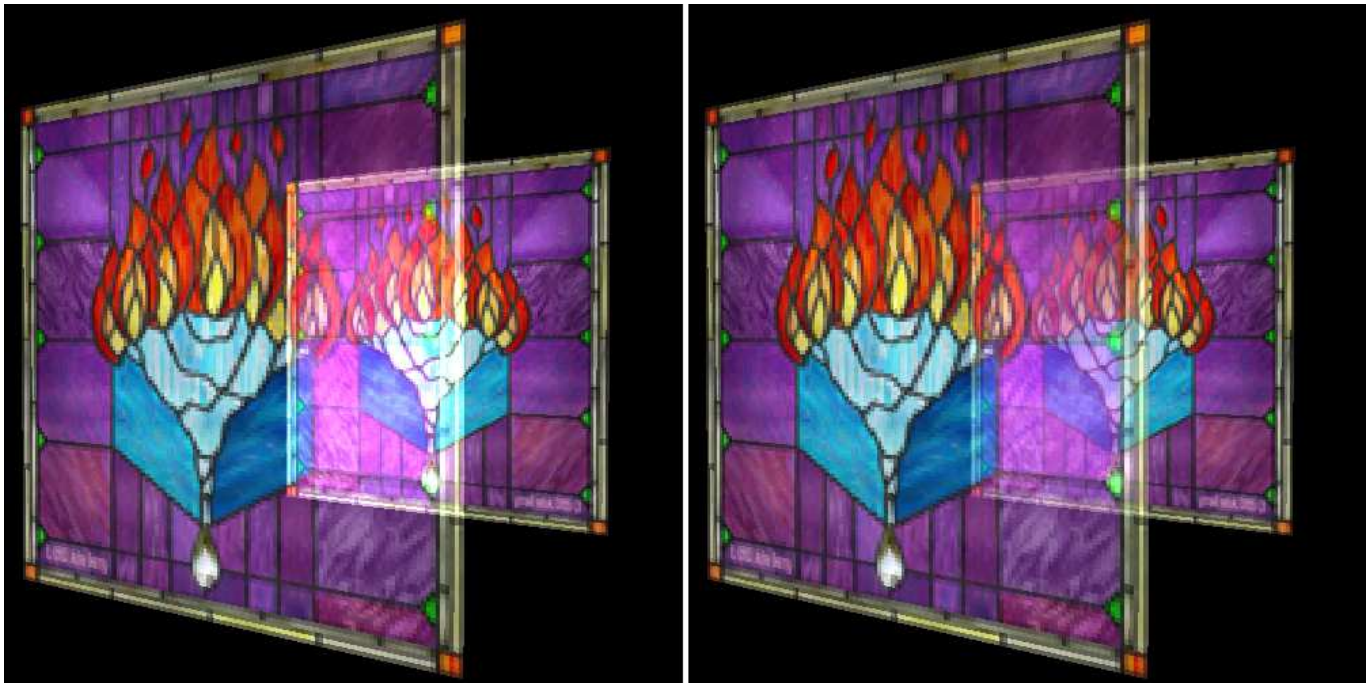
```
glDepthMask(GL_FALSE);
```

When the depth buffer is read-only, a translucent polygon that lies behind any opaque polygon already rendered is discarded. A translucent polygon that lies in front of any polygon that has already been rendered is blended with the color of the polygons of which it is in front. However, because the z-buffer is read-only for this polygon, the depth values in the buffer are unchanged. Opaque polygons set the depth mask to true and are rendered normally.

OpenGL Blending Examples

(a) (b)

(a) No Blending (b) glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)



(a) (b)

(a) `glBlendFunc(GL_ONE, GL_ONE)` (b) `glBlendFunc(GL_ONE, GL_SRC_ALPHA)`

Fog Affects Achieved by Compositing

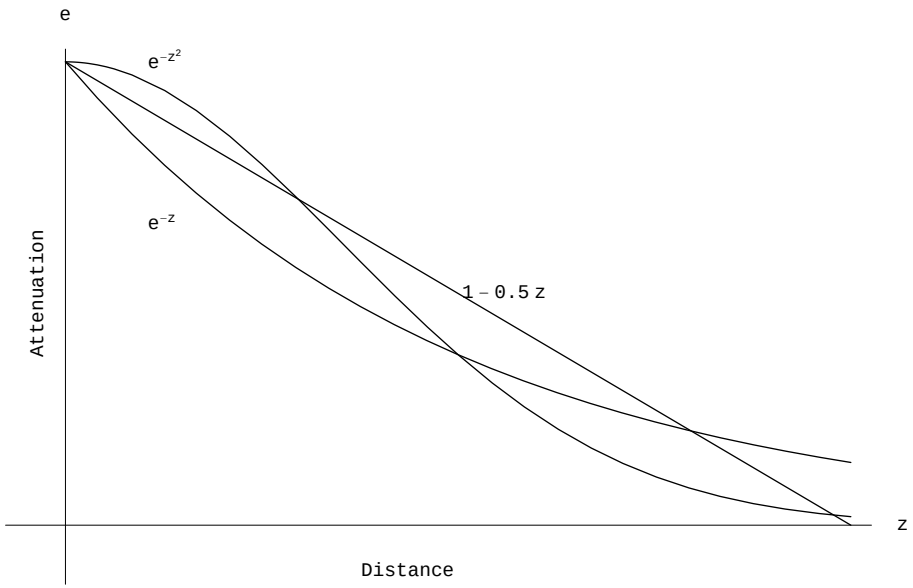
Let f denote a **fog factor**, and let z be the distance between a fragment being rendered and the viewer. If the fragment has a color $\mathbf{C}_s$ and the fog is assigned a color $\mathbf{C}_f$, then we can use the color

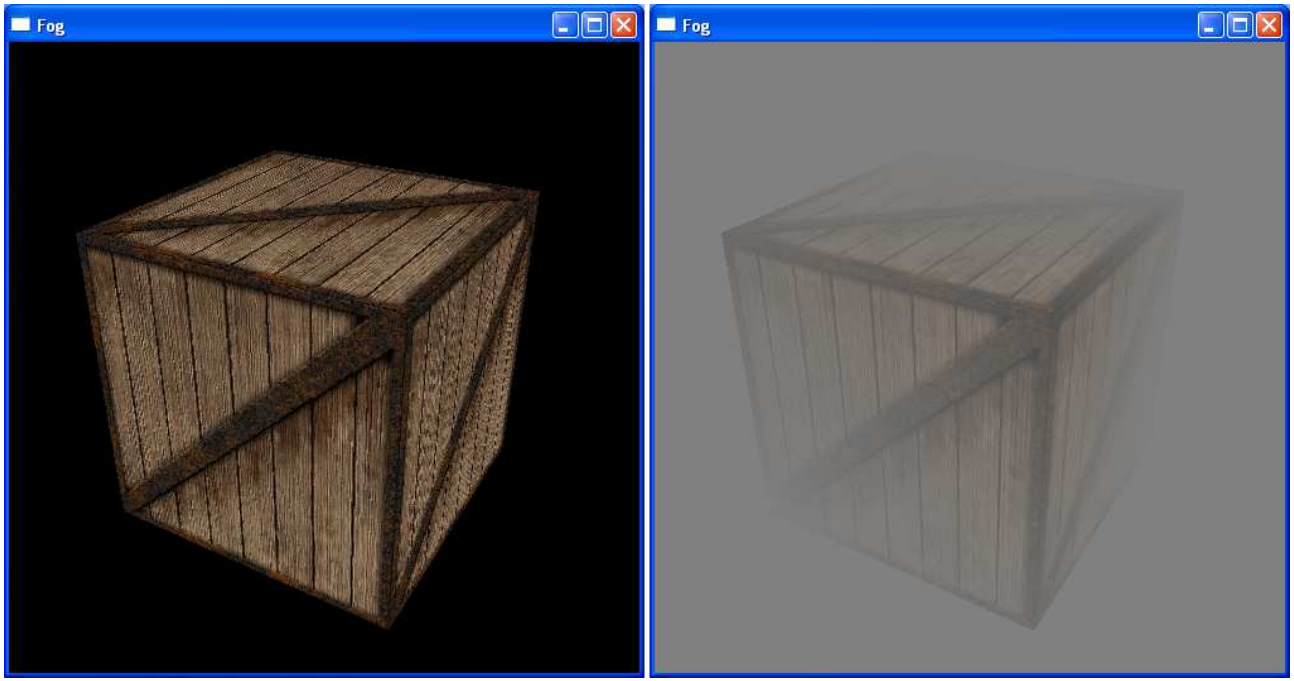
$$\mathbf{C}_{s'} = f\mathbf{C}_s + (1 - f)\mathbf{C}_f$$

in the rendering. If f varies linearly between some minimum and maximum values, we achieve a depth-cueing effect. If this factor varies exponentially, then we obtain effects that look more like fog. OpenGL supports linear, exponential, and Gaussian fog densities. For example, in RGBA mode, we can set up a fog-density function $f = e^{-0.5z^2}$ by using the function calls

```
GLfloat fcolor[4] = { ... };
glEnable(GL_FOG);
glFogf(GL_FOG_MODE, GL_EXP);
glFogf(GL_FOG_DENSITY, 0.5);
glFogfv(GL_FOG_COLOR, fcolor);
```

Fog Density


e
e^{-z^2}
e^{-z}
$1 - 0.5\,z$
Attenuation
z
Distance


Fog
Fog

Reading Assignment and News

Please review the appropriate sections related to this lecture in chapter 7, and associated exercises, of the recommended text.

(Recommended Text: Interactive Computer Graphics, by Edward Angel, Dave Shreiner, 6th edition, Addison-Wesley)

Please track Blackboard for the most recent Announcements and Project postings related to this course.

(http://www.cs.utexas.edu/users/bajaj/graphics2012/cs354/)