

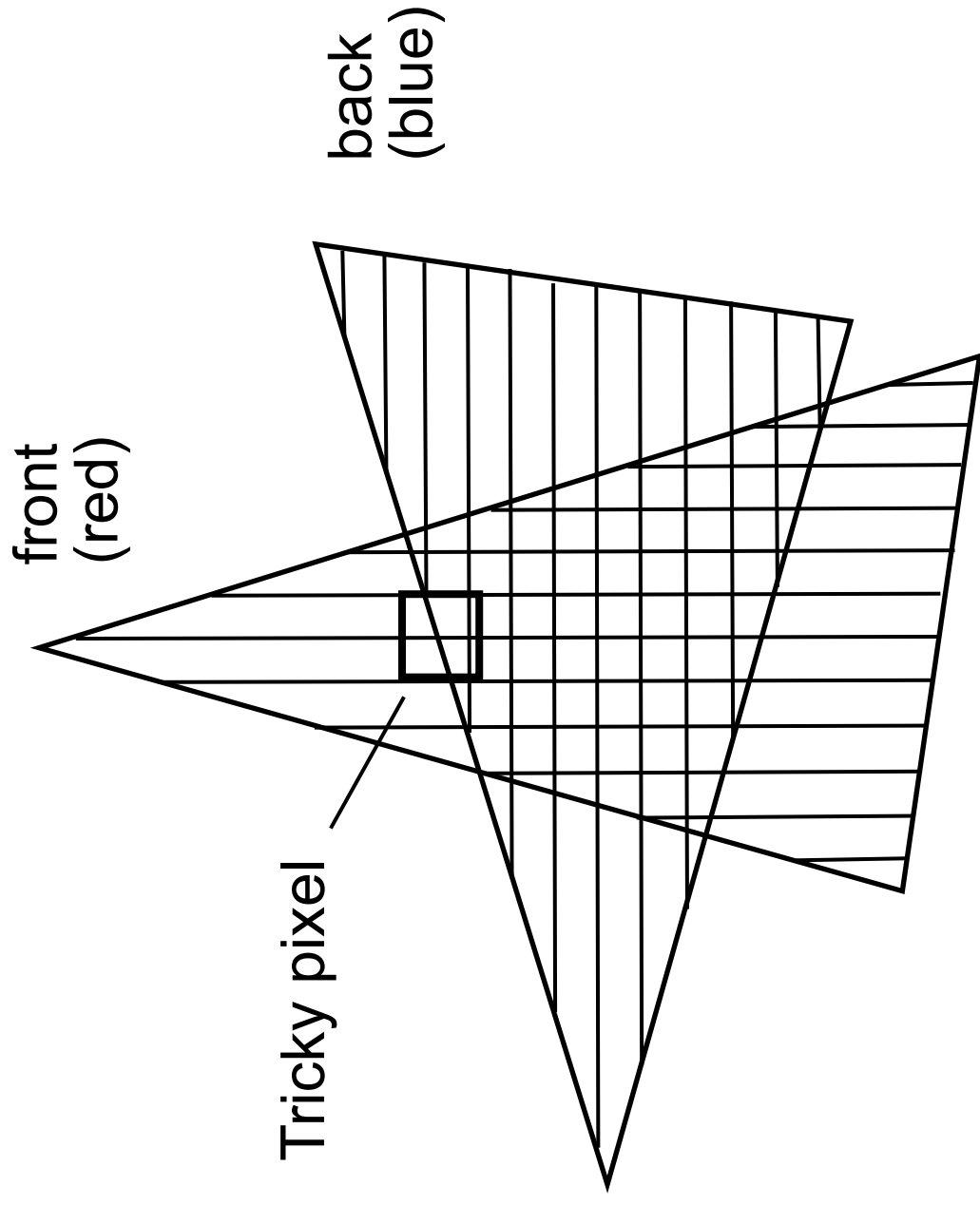
Imaging Compositing

- 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

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 front polygon but inside the back polygon? If we color it red only, aliasing artifacts will result. If we know that the front 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.



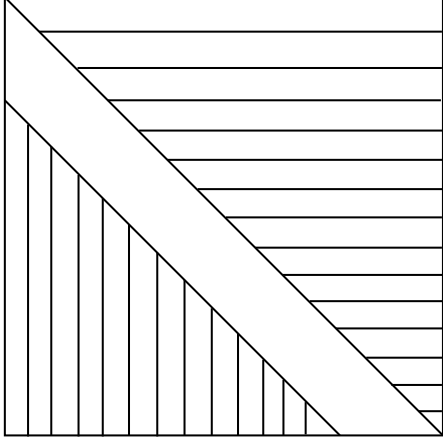
Compositing operations near an edge: How do we color the pixel?

The color associated with each pixel in the image is given an α *value* representing the *coverage* of the pixel.

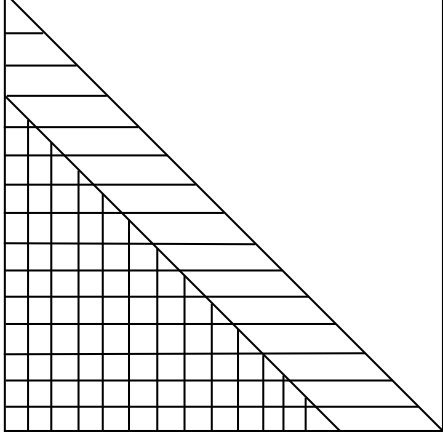
We need the α information at each pixel of the images being composited. Assume that, along with the RGB values of an image, we also have an α value encoding the coverage of each pixel. This collection of α values is often called the α *channel*.

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?

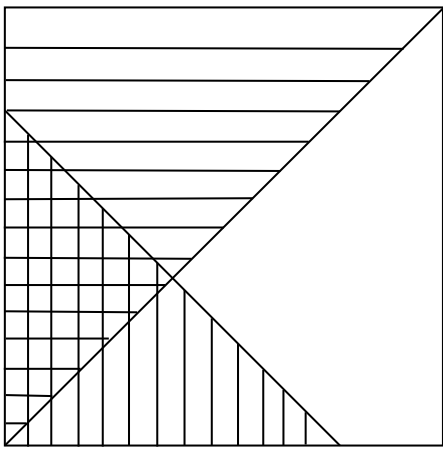
$$0.6\left(\frac{1}{3}\right)(1, 0, 0) + 0.4\left(\frac{1}{2}\right)(0, 0, 1) = (0.2, 0, 0.2)$$



Non overlap



Total overlap



Proportional overlap

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.

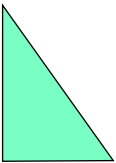
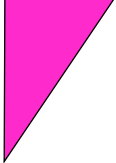
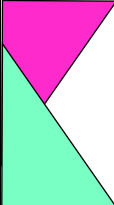
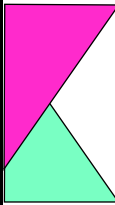
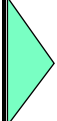
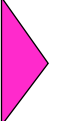
Areas and possible colors for regions of overlap in compositing

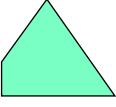
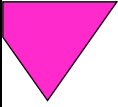
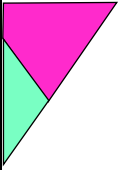
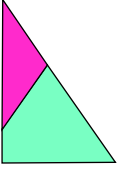
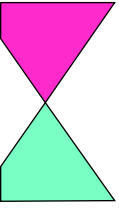
Region	Area	Possible colors
neither	$(1 - \alpha_A)(1 - \alpha_B)$	0
A alone	$\alpha_A(1 - \alpha_B)$	0, A
B alone	$\alpha_B(1 - \alpha_A)$	0, B
both	$\alpha_A\alpha_B$	0, A , B

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 $(\alpha R, \alpha G, \alpha B, \alpha)$ for each pixel, thus saving ourselves the trouble of performing the multiplications each time. When we refer to an $RGB\alpha$ value for a pixel, we mean exactly this.

How many possible ways of coloring this pixel are there?

A over B, *A in B*, and *A held out by B*.

operation	quadruple	diagram	F_A	F_B
clear	$(0, 0, 0, 0)$		0	0
<i>A</i>	$(0, A, 0, A)$		1	0
<i>B</i>	$(0, 0, B, B)$		0	1
<i>A over B</i>	$(0, A, B, A)$		1	$1 - \alpha_A$
<i>B over A</i>	$(0, A, B, B)$		$1 - \alpha_B$	1
<i>A in B</i>	$(0, A, B, A)$		α_B	0
<i>B in A</i>	$(0, A, B, A)$		0	α_A

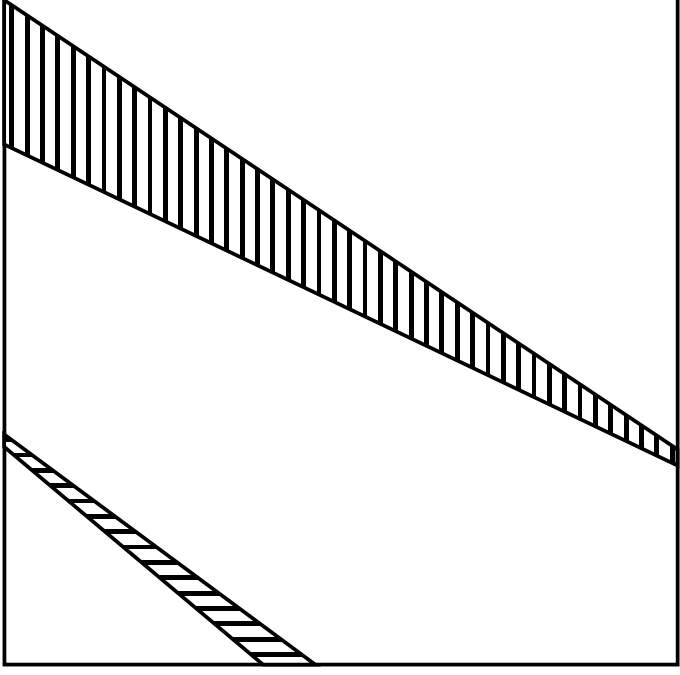
operation	quadruple	diagram	F_A	F_B
A held out by B	$(0, A, B, A)$		$1 - \alpha_B$	0
B held out by A	$(0, A, B, A)$		0	$1 - \alpha_A$
A atop B	$(0, A, B, A)$		α_B	$1 - \alpha_A$
B atop A	$(0, A, B, A)$		$1 - \alpha_B$	α_A
A xor B	$(0, A, B, A)$		$1 - \alpha_B$	$1 - \alpha_A$

$F_A(F_B)$ denotes the fraction of the pixel from image A (B), resulting color will be $F_{AC_A} + F_{BC_B}$. Quadruple in above table indicates the colors for the “neither”, “ A ”, “ B ”, and “both” regions.

darken $(A, \rho) := (\rho R_A, \rho G_A, \rho B_A, \alpha_A) \quad 0 \leq \rho \leq 1,$

fade $(A, \delta) := (\delta R_A, \delta G_A, \delta B_A, \delta \alpha_A) \quad 0 \leq \delta \leq 1$

opaque $(A, \omega) := (R_A, G_A, B_A, \omega \alpha_A).$



A case where the overlap assumptions fail in compositing.

Alternate Compositing Methods

1. First is to composite images in compressed form, by doing very simple operations (***A over B***, ***A and B***, ***A xor B***) on the compressed forms.
2. Other frame-buffer hardware to implement compositing.

Entry number (decimal)	Entry number (binary)	Contents of look-up table (decimal)
0	0 0 0	0
1	0 0 1	0
2	0 1 0	0
3	0 1 1	0
4	1 0 0	7
5	1 0 1	7
6	1 1 0	7
7	1 1 1	7
viewing $\longrightarrow$	$\overline{\text{image 2}}$ $\overline{\text{image 1}}$ $\overline{\text{image 0}}$	0 = black 7 = white

look-up table to display an image defined by the high-order bit of each pixel

Entry number (decimal)	Entry number (binary)	Contents of look-up table (decimal)
0	0 0 0	0
1	0 0 1	2
2	0 1 0	2
3	0 1 1	4
4	1 0 0	2
5	1 0 1	4
6	1 1 0	4
7	1 1 1	6
viewing $\longrightarrow$	$\begin{array}{c} 0 \\ 00 \\ 00 \\ 00 \\ 00 \\ 00 \\ 00 \\ 00 \end{array}$	

look-up table to display a sum of three 1-bit images

3. To use the look-up table to store a weighted sum of the intensities of two images, creating a double-exposure effect. If the weight applied to one image is decreased over time as the weight applied to the other is increased, we achieve the fade-out, fade-in effect called a *lap-dissolve*. The colors displayed during the fade sequence depend on the color space in which the weighted sum is calculated.

Entry number (decimal)	Entry number (binary)	Contents of look-up table (decimal)
0	0 0 0	0 no image present
1	0 0 1	3 image 0 visible
2	0 1 0	5 image 1 visible
3	0 1 1	5 image 1 visible
4	1 0 0	7 image 2 visible
5	1 0 1	7 image 2 visible
6	1 1 0	7 image 2 visible
7	1 1 1	7 image 2 visible
viewing	$\begin{matrix} 2 & 1 & 0 \\ \text{image} & \text{image} & \text{image} \end{matrix}$ $\longrightarrow$	

look-up table to assign priorities to three 1-bit images.

Generating α Values with Fill Mechanisms

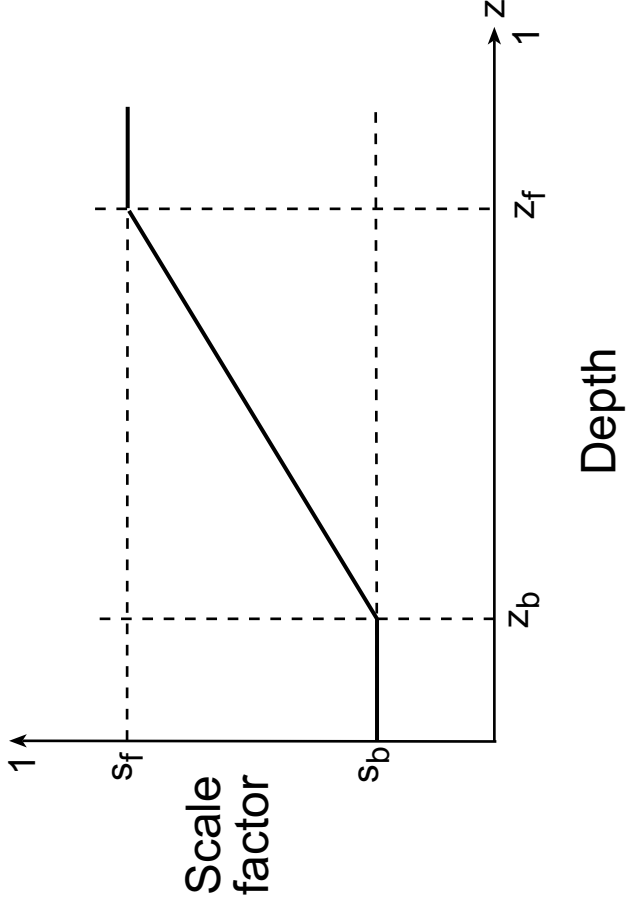
What if an image is produced by scanning of a photograph or is provided by some other source lacking this information? Can it still be composited? If we can generate an α channel for the image, we can use that channel for compositing. Even if we merely assign an α value of zero to black pixels and an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

Atmospheric Attenuation

To simulate the atmospheric attenuation from the object to the viewer, many systems provide *depth-cueing*. In this technique, which originated with vector-graphics hardware, more distant objects are rendered with lower intensity. Also attempt to approximate the shift in colors caused by the intervening atmosphere.

Front and back depth-cue reference planes are defined in NPC; each of these planes is associated with a scale factor, s_f and s_b , respectively, that ranges between 0 and 1. The scale factors determine the blending of the original intensity with that of a depth-cue color, $I_{dc\lambda}$. The goal is to modify I_λ to yield depth-cued value I'_λ that is displayed. Given z_0 , the object's z coordinate, a scale factor s_0 is used to interpolate between I_λ and $I_{dc\lambda}$, to determine

$$I'_\lambda = s_0 I_\lambda + (1 - s_0) I_{dc\lambda}.$$



Computing the scale factor for atmospheric attenuation

If z_o is in front of the front depth-cue reference plane's z coordinate z_f , then $s_o = s_f$. If z_o is behind the back depth-cue reference plane's z coordinate z_b , then $s_o = s_b$. Finally, if z_o is between the two planes, then

$$s_o = s_b + \frac{(z_o - z_b)(s_f - s_b)}{z_f - z_b}$$

Blending and Compositing in OpenGL

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

```
glEnable(GL_BLEND);
```

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

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

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 not be composited. 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.

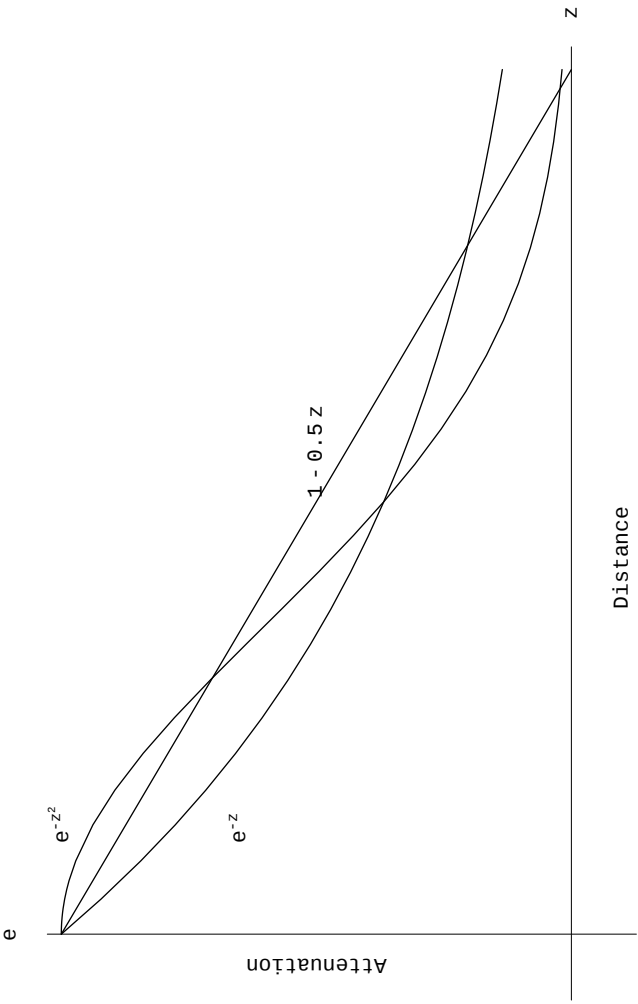
Fog Affects Achieved by Compositing

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 C_s and the fog is assigned a color C_f , then we can use the color

$$C_{s'} = fC_s + (1 - f)C_f$$

in the rendering. If f varies linearly between some minimum and maximum values, we have a depth-cueing effect. If this factor varies exponentially, then we get 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.