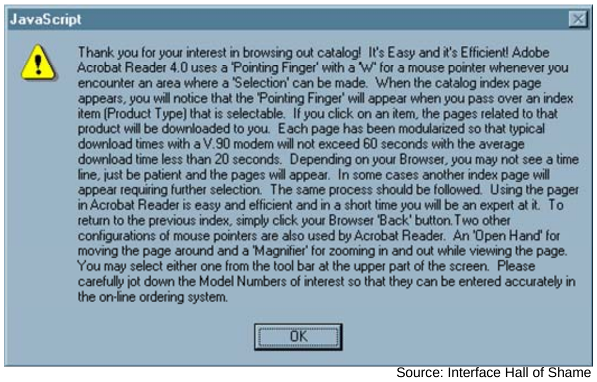
# Lecture 20: Alpha Compositing

## UI Hall of Fame or Shame?



Thank you for your interest in browsing out catalog! It's Easy and it's Efficient! Adobe Acrobat Reader 4.0 uses a 'Pointing Finger' with a 'W' for a mouse pointer whenever you encounter an area where a 'Selection' can be made. When the catalog index page appears, you will notice that the 'Pointing Finger' will appear when you pass over an index item (Product Type) that is selectable. If you click on an item, the pages related to that product will be downloaded to you. Each page has been modularized so that typical download times with a V.90 modem will not exceed 60 seconds with the average download time less than 20 seconds. Depending on your Browser, you may not see a time line, just be patient and the pages will appear. In some cases another index page will appear requiring further selection. The same process should be followed. Using the pager in Acrobat Reader is easy and efficient and in a short time you will be an expert at it. To return to the previous index, simply click your Browser 'Back' button.Two other configurations of mouse pointers are also used by Acrobat Reader. An 'Open Hand' for moving the page around and a 'Magnifier' for zooming in and out while viewing the page. You may select either one from the tool bar at the upper part of the screen. Please carefully jot down the Model Numbers of interest so that they can be entered accurately in the on-line ordering system.

Source: Interface Hall of Shame

Once upon a time, this bizarre help message was popped up by a website (Midwest Microwave) when users requested to view the site's product catalog. The message appears before the catalog is displayed. Clearly this message is a patch for usability problems in the catalog itself. But the message itself has a lot of usability problems of its own! How many problems can you find?

## Today's Topics

- Alpha channel
- Antialiasing
- Alpha compositing rules
- Masks

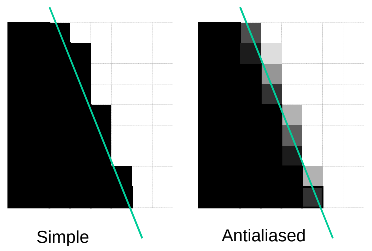
Today's lecture is about **alpha compositing** – the process of using the transparency value, alpha, to combine two images together.

## Transparency

- **Alpha** is a pixel's transparency
  - from 0.0 (transparent) to 1.0 (opaque)
  - 32-bit RGBA pixels: each pixel has red, green, blue, and alpha values
- Uses for alpha
  - Antialiasing
  - Nonrectangular images
  - Translucent components

In many graphics toolkits, the pixel model includes a fourth channel in addition to red, green, and blue: the pixel's **alpha** value, which represents its degree of transparency.

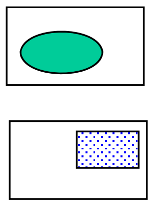
## Antialiasing



Recall that **antialiasing** is a way to make an edge look smoother. Instead of making a binary decision between whether to draw a pixel near the edge completely (opaque) or not at all (transparent), antialiasing uses an alpha value that varies from 0 to 1 depending on how much of the pixel is covered by the edge. The alpha value causes a blending between the background color and the drawn color. The overall effect is a fuzzier but smoother edge.

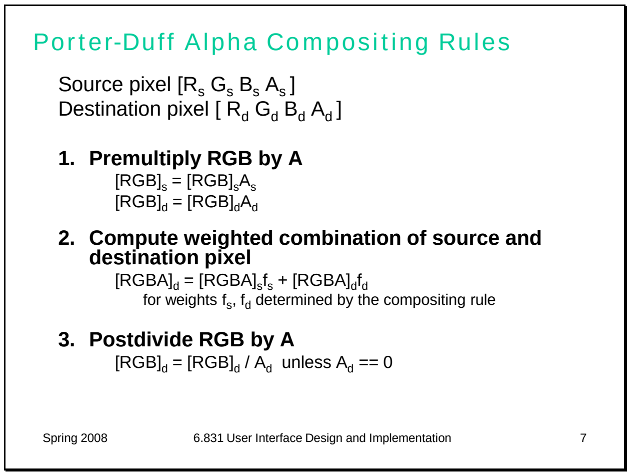
## Alpha Compositing

- Compositing rules control how source and destination pixels are combined
- Source
  - Image
  - Stroke drawing calls
- Destination
  - Drawing surface



When pixels include alpha values, drawing gets more interesting. When you draw on a drawing surface – whether it's using stroke calls such as drawRect(), or pixel copying like drawImage(), there are several ways that the alpha values of your drawing calls can interact with the alpha of the destination surface. This process is called **alpha compositing**.

Let's set up the situation. We have a rectangle of source pixels, which may be an image, or may be the pixels produced by some drawing call. We also have a rectangle of destination pixels, which is the drawing surface you want to modify. Alpha compositing determines what the resulting destination pixels will be after the source drawing is applied.



## Porter-Duff Alpha Compositing Rules

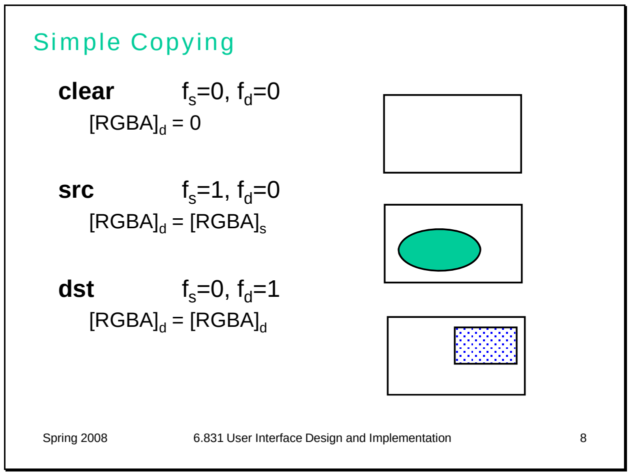
Source pixel $[R_s\ G_s\ B_s\ A_s]$
Destination pixel $[R_d\ G_d\ B_d\ A_d]$

1. **Premultiply RGB by A**
   $[RGB]_s = [RGB]_s A_s$
   $[RGB]_d = [RGB]_d A_d$
2. **Compute weighted combination of source and destination pixel**
   $[RGBA]_d = [RGBA]_s f_s + [RGBA]_d f_d$
   for weights $f_s$, $f_d$ determined by the compositing rule
3. **Postdivide RGB by A**
   $[RGB]_d = [RGB]_d / A_d$ unless $A_d == 0$

The compositing rules used by graphics toolkits were specified by Porter & Duff in a landmark paper (Porter & Duff, “Compositing Digital Images”, Computer Graphics v18 n3, July 1984). Their rules constitute an *algebra* of a few simple binary operators between the two images: over, in, out, atop, and xor. Altogether, there are 12 different operations, each using a different weighted combination of corresponding source pixel and destination pixel, where the weights are determined by alpha values.

The presentation of the rules is simplified if we assume that each pixel’s RGB value is **premultiplied** by its alpha value. For opaque pixels (A=1), this has no effect; for transparent pixels (A=0), this sets the RGB value to 0.

After the composition, the multiplication by alpha is undone by dividing each RGB value by the (final) alpha value of the pixel. If we were going to do a sequence of compositing operations, however, we might skip this step, deferring the division until the final composition is completed. (Java gives you an option, when you create an offscreen image buffer, whether you want the RGB values to be *stored* premultiplied by alpha; this representation will allow faster compositing.)



## Simple Copying

**clear** $f_s=0$, $f_d=0$
$[RGBA]_d = 0$

**src** $f_s=1$, $f_d=0$
$[RGBA]_d = [RGBA]_s$

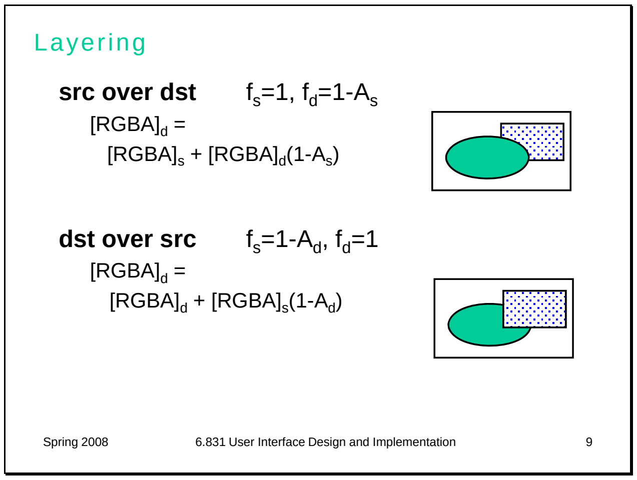
**dst** $f_s=0$, $f_d=1$
$[RGBA]_d = [RGBA]_d$

Here are the three simplest rules. They’re not particularly useful in practice, but they’re included to make the algebra complete.

**clear** combines the source and destination pixels with zero weights, so the effect is to fill the destination with transparent pixels. (The transparent pixels happen to be black, i.e. RGB=0, but the color of a transparent pixel is irrelevant.)

**src** replaces the destination image with the source image.

**dst** completely ignores the source image, and leaves the destination unchanged.



## Layering

**src over dst** $f_s=1$, $f_d=1-A_s$

$[RGBA]_d = [RGBA]_s + [RGBA]_d(1-A_s)$

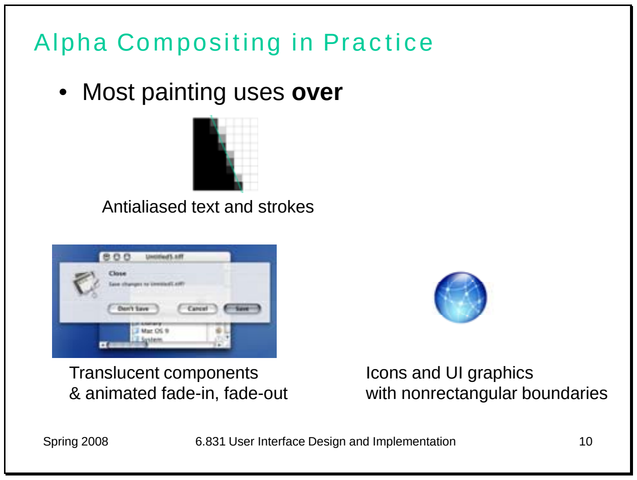
**dst over src** $f_s=1-A_d$, $f_d=1$

$[RGBA]_d = [RGBA]_d + [RGBA]_s(1-A_d)$

The next two rules produce layering effects.

**src over dst** produces the effect of drawing the source pixels on top of the destination pixels. Wherever the source is opaque (As=1), the existing destination pixel is completely ignored; and wherever the source is transparent (As=0), only the destination pixel shows through. (Note that RGBs=0 when As=0, because we have premultiplied by alpha). If the source is translucent (0 < As < 1), then the final pixel is a mix of the source and destination pixel.

**dst over src** produces the opposite effect – putting the source image *behind* the destination image. This is one way to affect drawing Z-order without having to change the actual order in which drawing calls are made. Be careful, though – in order for dst over src to succeed in a useful way, the destination image buffer must actually *have* an alpha channel, and it can't have been already been filled with opaque pixels. A typical drawing surface in Java (the Graphics object passed to your paintComponent() method) has already been filled with an opaque background, so you won't see any of your source drawing if you use **dst over src**.



## Alpha Compositing in Practice

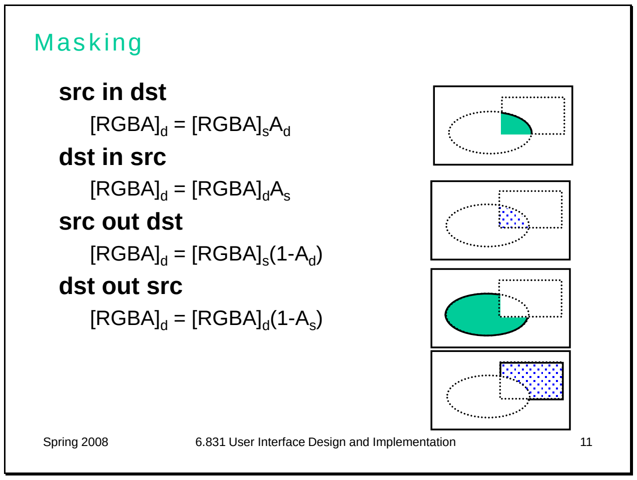
- Most painting uses **over**

Antialiased text and strokes

Translucent components
& animated fade-in, fade-out

Icons and UI graphics
with nonrectangular boundaries

In practice, most drawing uses **over**, including translucent components (like the Mac OS X modal sheet shown here) and nonrectangular images.



## Masking

**src in dst**

$[RGBA]_d = [RGBA]_s A_d$

**dst in src**

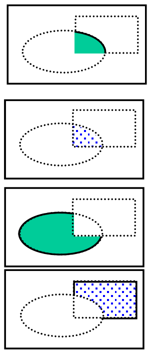
$[RGBA]_d = [RGBA]_d A_s$

**src out dst**

$[RGBA]_d = [RGBA]_s (1-A_d)$

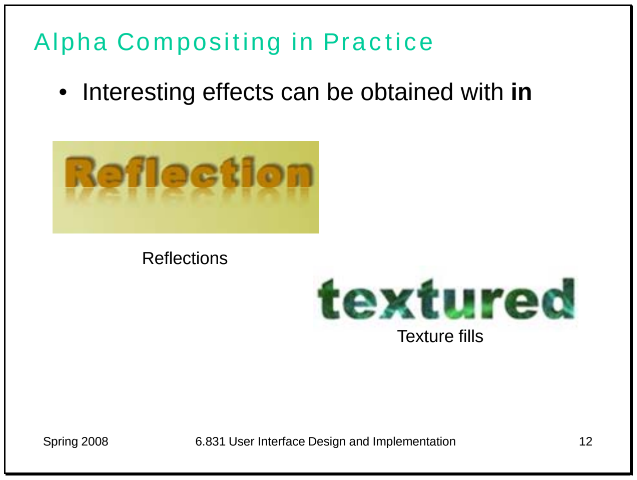
**dst out src**

$[RGBA]_d = [RGBA]_d (1-A_s)$



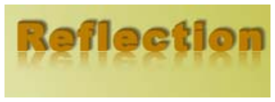
The next set of rules are for **masking**. Masking is like clipping – it restricts drawing to a certain area. But where clipping uses a shape (such as a rectangle) to describe the area, masking uses a pixel array. In older graphics systems, this pixel array was simply a bitmap: 1s for pixels that should be drawn, 0s for pixels that shouldn't be drawn. But with alpha compositing, the alpha channel represents the mask, a value ranging from 0.0 to 1.0 depending on how much of a pixel should be drawn.

Notice that these masking use the RGB values from only one of the images (source or destination). The other image is used only for its alpha channel; its RGB values are ignored.



## Alpha Compositing in Practice

- Interesting effects can be obtained with **in**



Reflections

Texture fills

Here are some of the applications for masking:

- generating the drop shadow of an arbitrary image (src is an image, dst is just filled gray – dst in src. Often a blur transform is added as well, to soften the edge.)

- generating a reflection of an arbitrary image (see the picture)

- pattern or texture filling (src is a pattern, like tiled images, dst is a filled or stroked shape – src in dst)

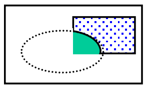
- clipping where the clip region should have antialiased borders (src is drawing calls, dst is filled clip region shape with antialiased borders, src in dst)

Many of these effects can be achieved in an image processing application like Photoshop, using the same kinds of techniques – creating an alpha-channel mask and using it with a source image in a compositing operation. But keep in mind that that process produces a static image, which might be useful for an icon or button label. If you want to create an effect dynamically, e.g. with the user's data, then you need to be able to program the compositing in a GUI toolkit, like Java2D.

## Other Masking

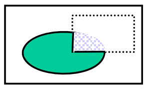
**src atop dst**

$[RGBA]_d =$
$[RGBA]_s A_d + [RGBA]_d(1-A_s)$



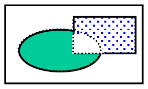
**dst atop src**

$[RGBA]_d =$
$[RGBA]_s(1-A_d) + [RGBA]_d A_s$



**src xor dst**

$[RGBA]_d =$
$[RGBA]_s(1-A_d) + [RGBA]_d(1-A_s)$



These are the last three rules. **src atop dst** is like src over dst, but it omits any source pixels where the destination is transparent. And **src xor dst** omits any pixels where both the source and the destination are nontransparent.

atop and xor aren’t terribly useful in practice; earlier versions of Java actually omitted them, but they’re present in Java 1.5.

## Summary

- Alpha channel is useful for smooth antialiased edges
- Alpha compositing rules allow for interesting effects