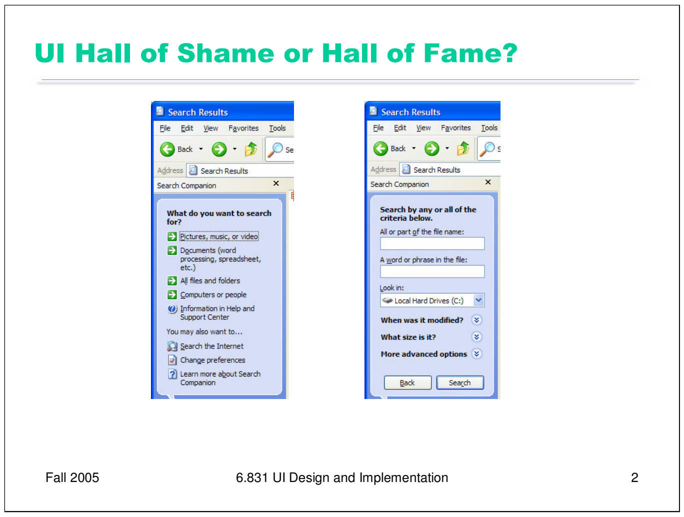
Lecture 17: Output Models 2



UI Hall of Shame or Hall of Fame?

Search Results
File Edit View Favorites Tools
Back
Address Search Results
Search Companion
What do you want to search for?
Pictures, music, or video
Documents (word processing, spreadsheet, etc.)
All files and folders
Computers or people
Information in Help and Support Center
You may also want to...
Search the Internet
Change preferences
Learn more about Search Companion
Search Results
File Edit View Favorites Tools
Back
Address Search Results
Search Companion
Search by any or all of the criteria below.
All or part of the file name:
A word or phrase in the file:
Look in:
Local Hard Drives (C:)
When was it modified?
What size is it?
More advanced options
Back
Search

Fall 2005 6.831 UI Design and Implementation 2

This is the Windows XP Search Companion. It appears when you press the Search button on a Windows Explorer toolbar, and is primarily intended for finding files on your hard disk.

An interesting feature of this interface is that, rather than giving a textbox for search keywords right away, it first asks you to specify what kind of file you're looking for. There's some logic to this design decision, because it turns out that different search criteria are appropriate for different kinds of files. For example, if you select "Picture, music, and video", the next step of the dialog won't both asking for a word or phrase *inside* the file, since these kinds of files are not textual. Similarly, if you select "Documents", the next step of the dialog will ask not only for search keywords, but also for the approximate time since you last edited the file, since most documents are sought for editing purposes (while most media files are sought for playing purposes).

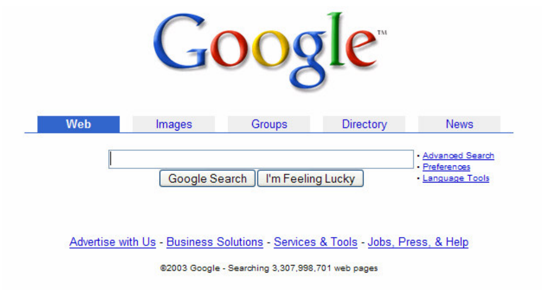
Unfortunately, to a frequent user, the demand that you specify the file's type *first* feels jarring and hard to answer. The categories are not disjoint, so the decision isn't always easy. Are HTML files and simple text files included in "Documents", or only Microsoft Office files? Some of the categories are bizarre – "computers or people"? Why is "Internet" a completely separate category, and why does Help get a different icon than the rest? Perhaps the worst problem in the category list is that the answer that frequent users are most likely to want – "All files and folders", to be sure that the search won't miss anything – is actually buried in the middle of the list, where it's hardest to find and click.

This interface is clearly designed for novice users. Hence the **wizard** design, a fixed sequence of carefully guided steps. And hence the cute animated cartoon dog, which some people in class found condescending by its mere presence. It's still an open question whether cartoon characters like this dog and the Paperclip are more helpful or harmful to good user interface design. So far, experiments with characters in serious commercial interfaces (designed for productivity rather than entertainment) have been largely unsuccessful.

The animated dog does have one advantage: it's a very visible mode status indicator. You won't accidentally leave the Windows Explorer in search mode, because the dog will get your attention and motivate you to find a way to get rid of it -- which is not trivial, since there's no obvious Cancel button.

Another problem with this wizard is that the Back button on toolbar is easy to confuse with the Back button in the dialog. The user thinks "this isn't what I want, I'll go Back", but then reaches habitually for the Back button in the toolbar, which backs up the main Explorer window instead of the Search Companion pane. This is probably a **capture error**, because of the effect of habit, but it also has some features of a **description error.**

UI Hall of Fame or Shame?


Google
Web
Images
Groups
Directory
News
Google Search
I'm Feeling Lucky
Advanced Search
Preferences
Language Tools
Advertise with Us - Business Solutions - Services & Tools - Jobs, Press, & Help
©2003 Google - Searching 3,307,998,701 web pages

In contrast to the previous example, here's Google's start page. Google is an outstanding example of a heuristic we'll see today: **Aesthetic and minimalist design**. Its interface is as simple as possible. Unnecessary features and hyperlinks are omitted, lots of whitespace is used. Google is fast to load and trivial to use.

But maybe Google goes a little too far! Take the perspective of a completely novice user coming to Google for the first time.

•What does Google actually do? The front page doesn't say.

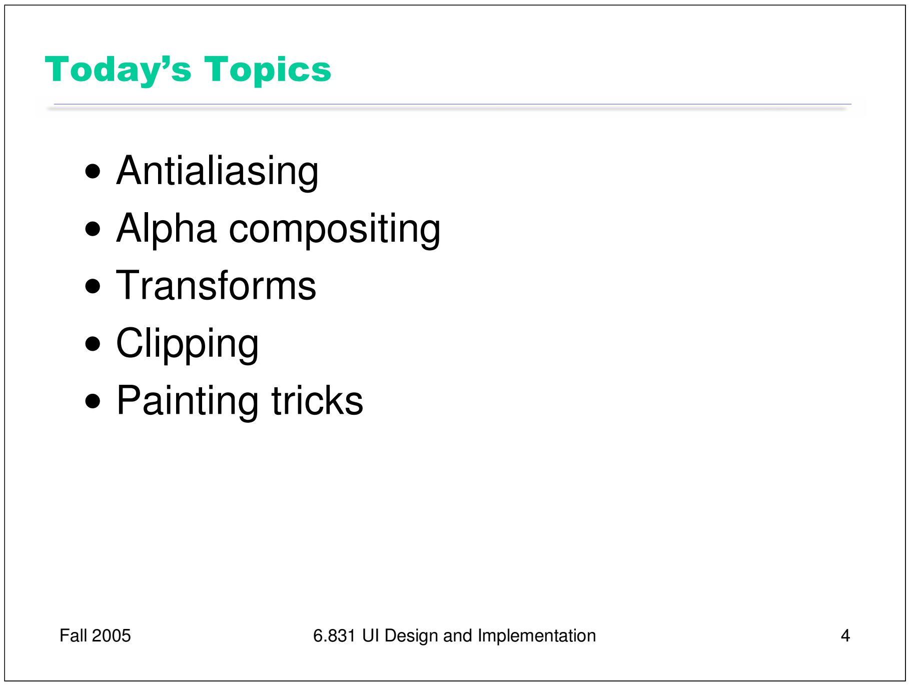
•What should be typed into the text box? It has no caption at all.

•The button labels are almost gibberish. "Google Search" isn't meaningful English (although it's gradually becoming more meaningful as *Google* enters the language as a noun, verb, and adjective). And what does "I'm Feeling Lucky" mean?

•Where is Help? Turns out it's buried at the bottom, along with "Jobs & Press".

Although these problems would be easy for Google to fix, they are actually minor, because Google's interface is simple enough that it can be learned by only a small amount of exploration. (Except perhaps for the I'm Feeling Lucky button, which probably remains a mystery until a user is curious enough to hunt for the help. After all, maybe it does a random choice from the search results!)

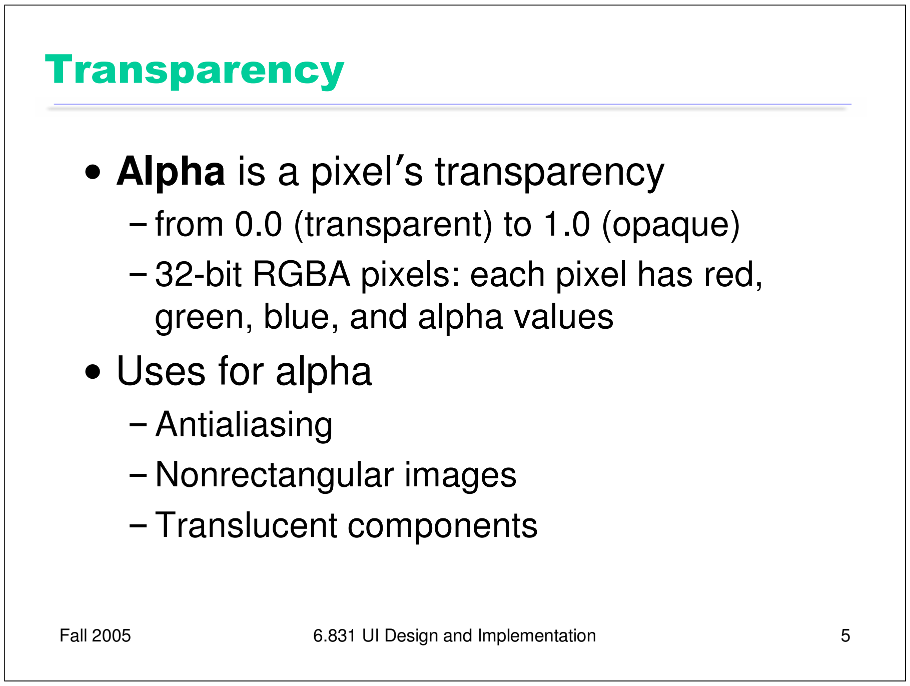
Notice that Google does not ask you to choose your search domain first. It picks a good default, and makes it easy to change.



Today's Topics

- Antialiasing
- Alpha compositing
- Transforms
- Clipping
- Painting tricks

Fall 2005 6.831 UI Design and Implementation 4



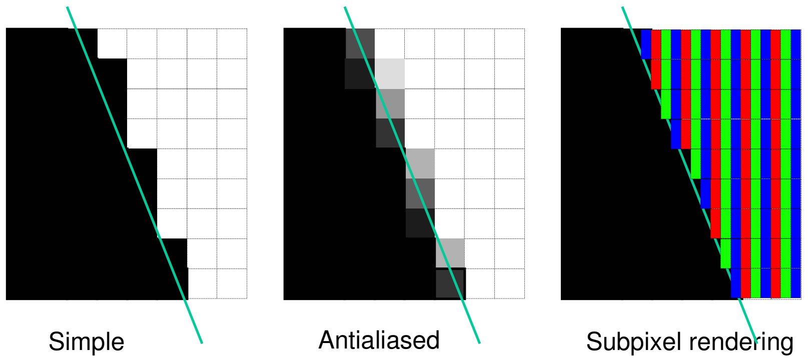
Transparency

- **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

Fall 2005 6.831 UI Design and Implementation 5

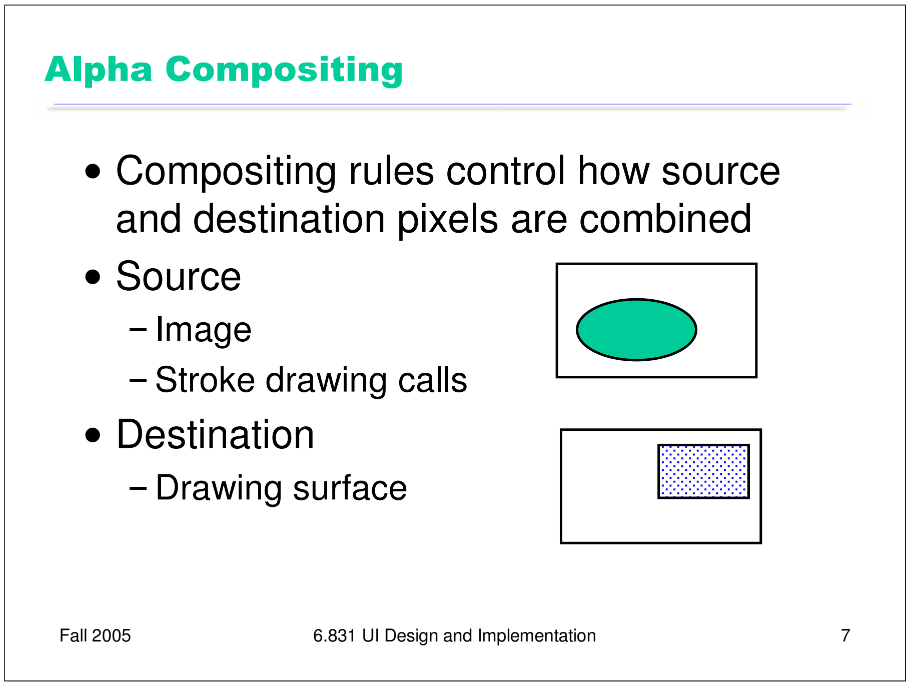
In many graphics toolkits, the pixel model includes a fourth channel in addition to red, green, and blue: the pixel's **alpha** value, which represents its degree of transparency.

Antialiasing and Subpixel Rendering



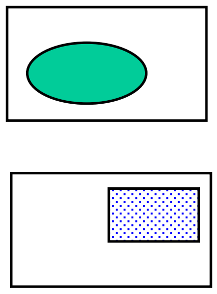
One use of the alpha channel is **antialiasing**, which is a way to make an edge look smoother. Instead of making a binary decision between whether to draw a pixel (opaque) or not (transparent), antialiasing uses an alpha value determined by the fraction of the pixel that is covered by the edge. The overall effect is a fuzzier but less jagged edge.

Subpixel rendering takes this a step further. Every pixel on an LCD screen consists of three discrete pixels side-by-side: red, green, and blue. So we can get a horizontal resolution which is three times the nominal pixel resolution of the screen, simply by choosing the colors of the pixels along the edge so that the appropriate subpixels are light or dark. It only works on LCD screens, not CRTs, because CRT pixels are often arranged in triangles, and because CRTs are analog, so the blue in a single "pixel" usually consists of a bunch of blue phosphor dots interspersed with green and red phosphor dots. You also have to be careful to smooth out the edge to avoid color fringing effects on perfectly vertical edges. And it works best for high-contrast edges, like this edge between black and white. Subpixel rendering is ideal for text rendering, since text is usually small, high-contrast, and benefits the most from a boost in horizontal resolution. Windows XP includes ClearType, an implementation of subpixel rendering for Windows fonts. (For more about subpixel rendering, see Steve Gibson, "Sub-Pixel Font Rendering Technology", http://grc.com/cleartype.htm)



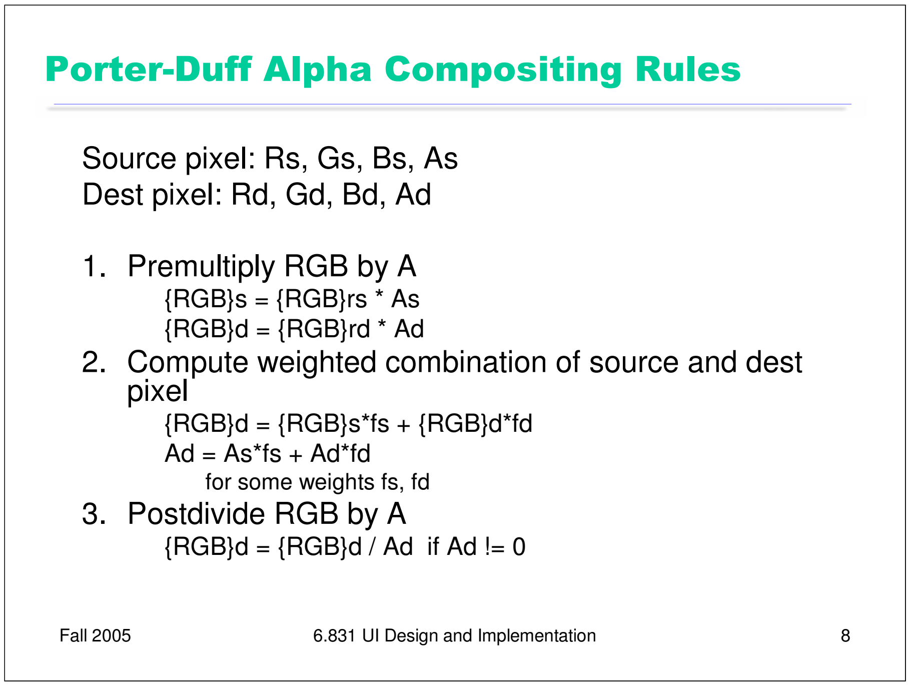
Alpha Compositing

- 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 image bitblitting 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: Rs, Gs, Bs, As
Dest pixel: Rd, Gd, Bd, Ad

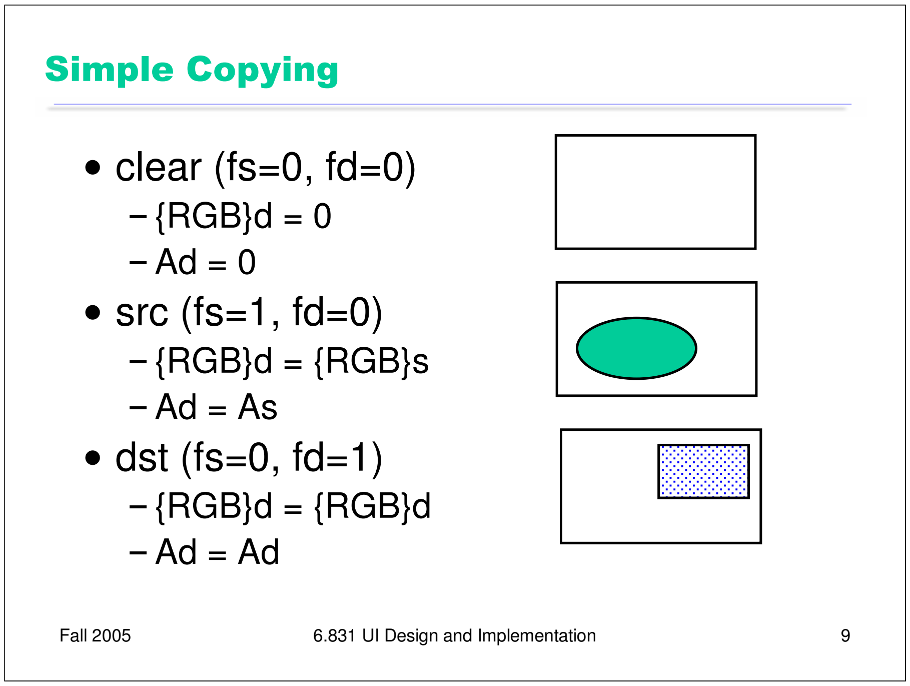
1. Premultiply RGB by A
   {RGB}s = {RGB}rs * As
   {RGB}d = {RGB}rd * Ad
2. Compute weighted combination of source and dest pixel
   {RGB}d = {RGB}s*fs + {RGB}d*fd
   Ad = As*fs + Ad*fd
   for some weights fs, fd
3. Postdivide RGB by A
   {RGB}d = {RGB}d / Ad if Ad != 0

Fall 2005 6.831 UI Design and Implementation 8

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; this representation will allow faster compositing.)



Simple Copying

- clear (fs=0, fd=0)
  - {RGB}d = 0
  - Ad = 0
- src (fs=1, fd=0)
  - {RGB}d = {RGB}s
  - Ad = As
- dst (fs=0, fd=1)
  - {RGB}d = {RGB}d
  - Ad = Ad

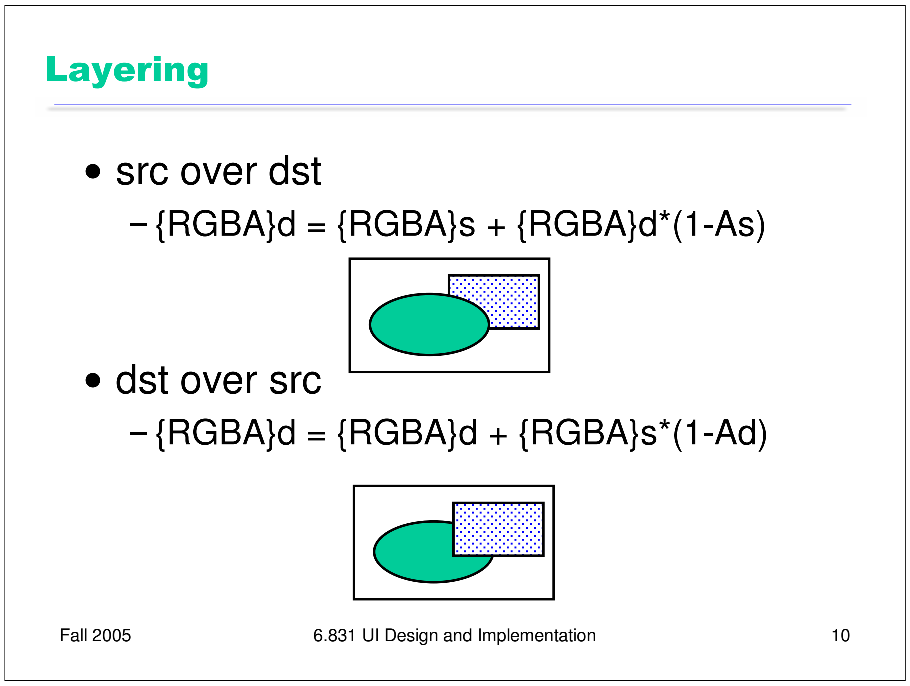
Fall 2005 6.831 UI Design and Implementation 9

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.)

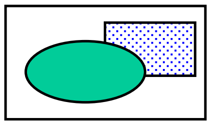
src replaces the destination image with the source image.

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

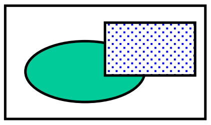


Layering

- src over dst
  - {RGBA}d = {RGBA}s + {RGBA}d*(1-As)



- dst over src
  - {RGBA}d = {RGBA}d + {RGBA}s*(1-Ad)

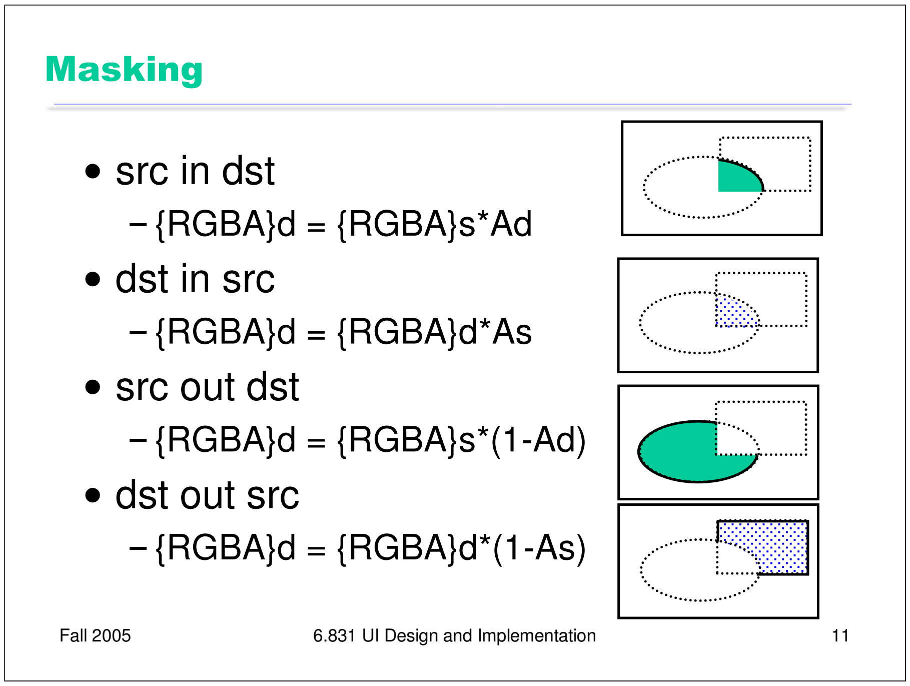


Fall 2005 6.831 UI Design and Implementation 10

The next two rules produce layering effects.

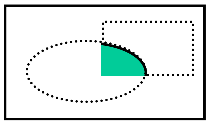
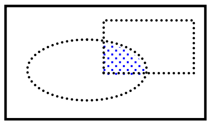
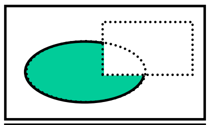
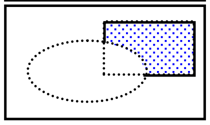
src over dst produces the effect of drawing the source pixels on top of the destination pixels. Wherever the source is opaque (As=1), the existing destination pixel is completely ignored; and wherever the source is transparent (As=0), only the destination pixel shows through. (Note that RGBs=0 when As=0, 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**.



Masking

- src in dst
  - {RGBA}d = {RGBA}s*Ad
- dst in src
  - {RGBA}d = {RGBA}d*As
- src out dst
  - {RGBA}d = {RGBA}s*(1-Ad)
- dst out src
  - {RGBA}d = {RGBA}d*(1-As)

Fall 2005 6.831 UI Design and Implementation 11

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.

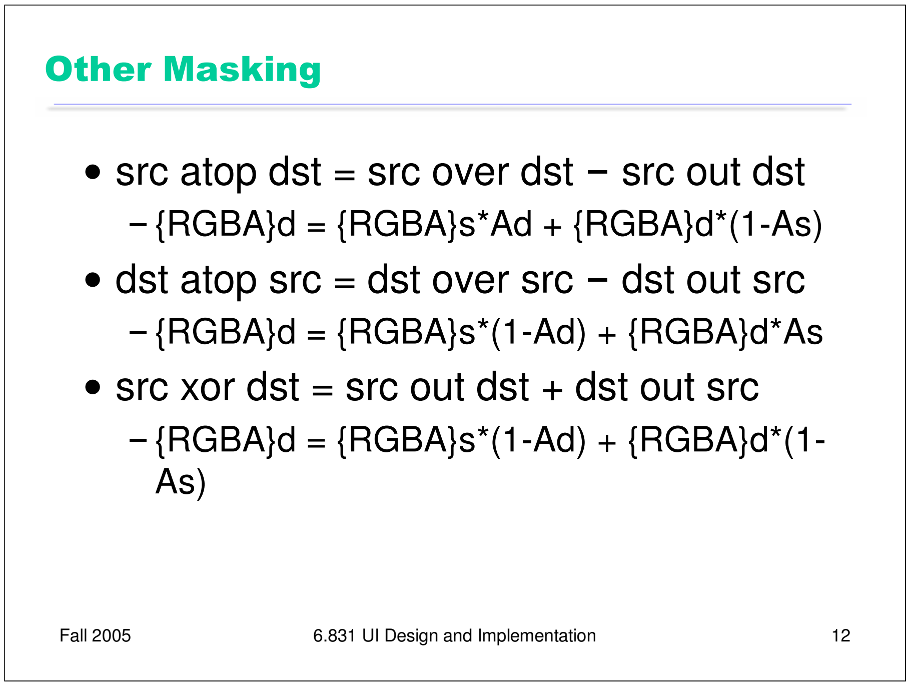
Notice that these masking use only the RGB values from one of the images (source or destination). The other image is used only for its alpha channel; its RGB values are ignored.

Here are some of the applications for masking:

- generating the drop shadow of a nonrectangular image (src is an image, dst is just filled gray – dst in src)

- 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)



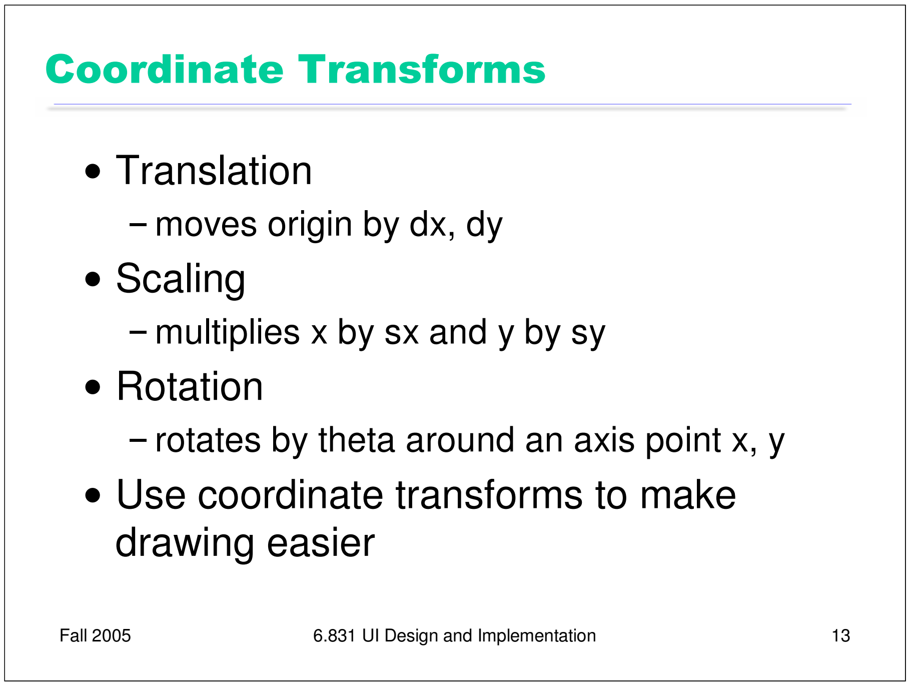
Other Masking

- src atop dst = src over dst − src out dst
  - {RGBA}d = {RGBA}s*Ad + {RGBA}d*(1-As)
- dst atop src = dst over src − dst out src
  - {RGBA}d = {RGBA}s*(1-Ad) + {RGBA}d*As
- src xor dst = src out dst + dst out src
  - {RGBA}d = {RGBA}s*(1-Ad) + {RGBA}d*(1-As)

Fall 2005 6.831 UI Design and Implementation 12

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.



Coordinate Transforms

- Translation
  - moves origin by dx, dy
- Scaling
  - multiplies x by sx and y by sy
- Rotation
  - rotates by theta around an axis point x, y
- Use coordinate transforms to make drawing easier

Fall 2005 6.831 UI Design and Implementation 13

Coordinate systems are relevant to all output models. In the component model, every component in a view hierarchy has its own local coordinate system, whose origin is generally at the top left corner of the component, with the y axis increasing down the screen. (Postscript is an exception to this rule; its origin is the bottom left, like conventional Cartesian coordinates.)

When you're drawing a component, you start with the component's local coordinate system. But you can change this coordinate system (a property of the graphics context) using three transformations:

Translation moves the origin, effectively adding (dx,dy) to every coordinate used in subsequent drawing.

Scaling shrinks or stretches the axes, effectively multiplying subsequent x coordinates by a scaling factor *sx* and subsequent y coordinates by *sy*.

Rotation rotates the coordinate system around some axis point, not necessarily the origin.

These operations are typically represented internally by a transform matrix which can be multiplied by a coordinate vector [x,y] to map it back to the original coordinate system. Scaling and rotation are easy to represent by matrix multiplication, but translation seems harder, since it involves vector addition, not multiplication. **Homogeneous transforms** offer a way around this problem, allowing translations to be represented homogeneously with the other transforms, so that the effect of a sequence of coordinate transforms can be multiplied together into a single matrix. Homogeneous transforms add a dummy element 1 to each coordinate vector:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

and represent each transform by a 3x3 matrix:

Translation

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix}$$

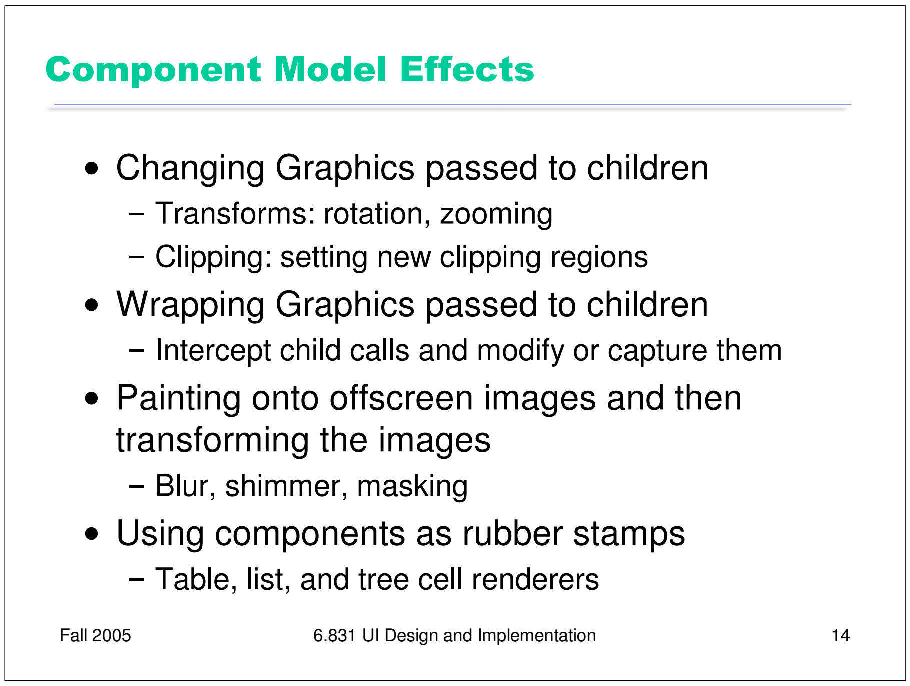
Scaling

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation around origin

$$\begin{bmatrix} \cos(theta) & -\sin(theta) & 0 \\ \sin(theta) & \cos(theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Rotation around another axis is done by first translating the origin to the axis point, rotating around the origin, and then inverting the translation.



Component Model Effects

- Changing Graphics passed to children
  - Transforms: rotation, zooming
  - Clipping: setting new clipping regions
- Wrapping Graphics passed to children
  - Intercept child calls and modify or capture them
- Painting onto offscreen images and then transforming the images
  - Blur, shimmer, masking
- Using components as rubber stamps
  - Table, list, and tree cell renderers

Fall 2005 6.831 UI Design and Implementation 14

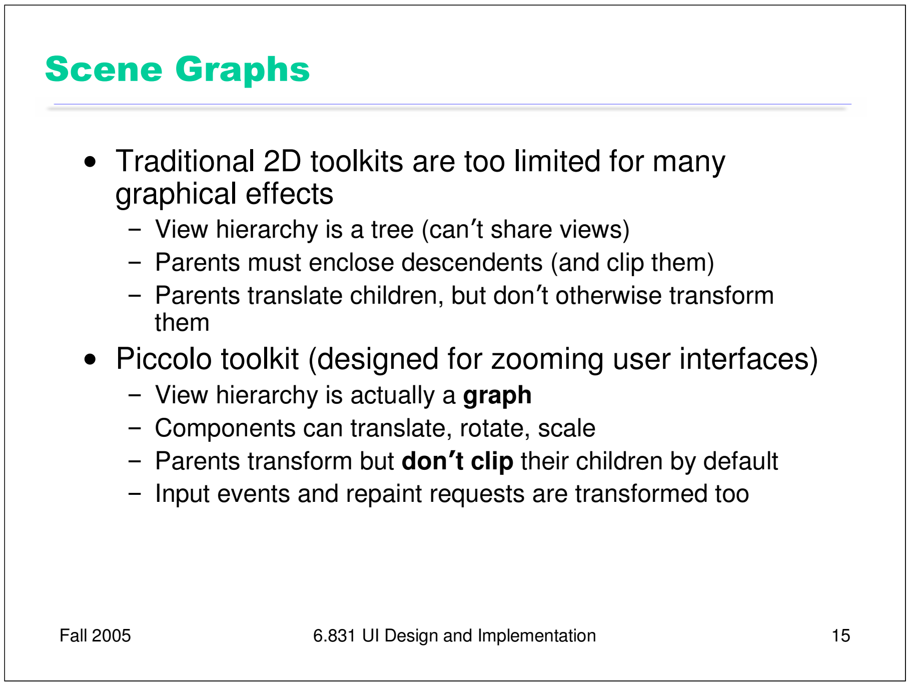
Using visual effects in the component model has some special problems, especially if you want your container to be decoupled from its component children – i.e., if you want it to handle arbitrary children who might draw themselves in arbitrary ways. Here are some tricks you can use to change the way your children draw themselves. (Some of these ideas come from a good paper: Edwards et al, "Systematic Output Modification in a 2D User Interface Toolkit", UIST '97.)

One technique is to change the defaults in the graphics context you pass down to your children. For example, you can apply transformations to the graphics context to persuade your children to draw in different places, or magnify or shrink their results. One problem with these kinds of transformations is that they can screw up input and automatic redraw. If a component is drawn transformed, you have to transform hit testing and input event coordinates in the same way; similarly, if the component asks to repaint itself, its repaint rectangle has to be transformed likewise. So if your toolkit doesn't support transforming input and redraw, you should restrict the use of this technique to components that don't expect input and that will notify *you* if they change.

Another trick is to put a wrapper around the Graphics object – a wrapper that delegates to the inner Graphics object, but changes the way certain kinds of drawing is done. For example, you could write a Graphics wrapper that produces a drop shadow underneath every stroke drawn by a child.

You can also create an offscreen image buffer, create a graphics context that uses it as a drawing surface, and then have your children paint themselves through this new graphics context. This gives you complete access to the pixel image produced by your children, so you can apply arbitrary effects to it. For example, you can create a drop shadow from the entire image, using masking; you can apply a Gaussian filter to it to blur the sharp edges; you can animate a shimmering effect. The result of these operations then gets copied to the onscreen drawing surface.

The final component-model technique is concerned not with components as children, but rather components as encapsulated drawing procedures – **rubber stamps** that, given some parameters, can paint a rendering of those parameters. For example, you can create a label widget, fill in its text, font, x, y, and size, and call its paint() method to paint it on an arbitrary graphics context, even though you never added it to a view hierarchy. Several Swing classes use this approach – JList, JTable, and JTree for example. These classes can be configured with *renderers* which are simply component factories, but the components are used only for stamping out output. This approach is even lighter-weight than the glyph pattern. You might need only one JLabel to stamp out all the text in a column, for example.



Scene Graphs

- Traditional 2D toolkits are too limited for many graphical effects
  - View hierarchy is a tree (can't share views)
  - Parents must enclose descendents (and clip them)
  - Parents translate children, but don't otherwise transform them
- Piccolo toolkit (designed for zooming user interfaces)
  - View hierarchy is actually a **graph**
  - Components can translate, rotate, scale
  - Parents transform but **don't clip** their children by default
  - Input events and repaint requests are transformed too

Fall 2005 6.831 UI Design and Implementation 15

Finally, let's look at Piccolo, a novel UI toolkit developed at University of Maryland. Piccolo is specially designed for building **zoomable** interfaces, which use smooth animated panning and zooming around a large space.

Piccolo has a view hierarchy consisting of PNode objects. But the hierarchy is not merely a tree, but in fact a **graph**: you can install camera objects in the hierarchy which act as viewports to other parts of the hierarchy, so a component may be seen in more than one place on the screen. Another distinction between Piccolo and other toolkits is that every component has an arbitrary transform relative to its parent's coordinate system – not just translation (which all toolkits provide), but also rotation and scaling. The toolkit automatically handles transforming not only output, but also input event coordinates, hit tests, and repaint requests.

Furthermore, in Piccolo, parents do not clip their children by default. If you want this behavior, you have to request it by inserting a special clipping object (a component) into the hierarchy. As a result, components in Piccolo have two bounding boxes – the bounding box of the node itself (getBounds()), and the bounding box of the node's entire subtree (getFullBounds()).

The widget set for Piccolo is fairly small by comparison with toolkits like Swing and .NET, probably because Piccolo is a research project with limited resources. It's worth noting, however, that Piccolo provides reusable components for shapes (e.g. lines, rectangles, ellipses, etc), which in other toolkits would require revering to the stroke model.

Piccolo home page: http://www.cs.umd.edu/hcil/piccolo/

Overview: http://www.cs.umd.edu/hcil/piccolo/learn/patterns.shtml

API documentation: http://www.cs.umd.edu/hcil/jazz/learn/piccolo/doc-1.1/api/