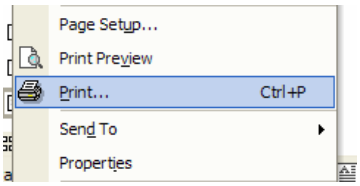
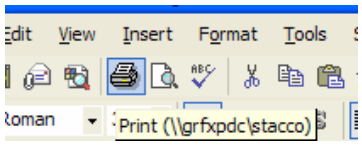
Lecture 11: Toolkits

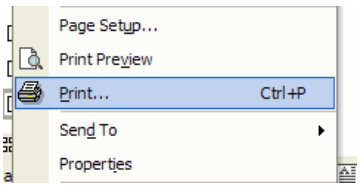
UI Hall of Fame or Shame?

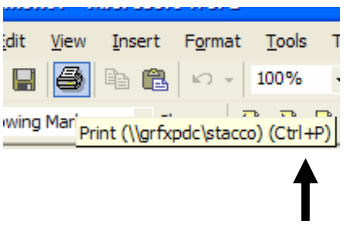
Power Point





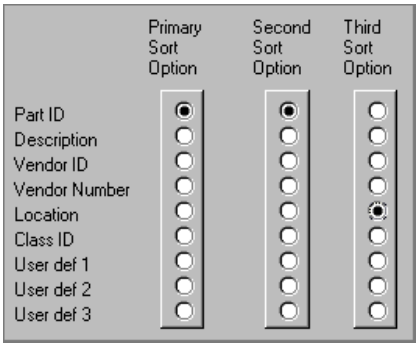
Word





Here's a followup on the Hall of Shame entry from last time, which compared the behavior of the Print menu item, toolbar button, and keyboard shortcut in Microsoft PowerPoint. It turns out that Word is even more misleading. Not only are both the menu item and the toolbar button named Print – despite doing different things -- but *both labels* mention the keyboard shortcut Ctrl-P. In fact, only the menu item is telling the truth. Ctrl-P in Word pops up the Print dialog, instead of printing immediately.

UI Hall of Fame or Shame?



Source: UI Hall of Shame

Our Hall of Shame candidate for the day is this interface for choose how a list of database records should be sorted. Like other sorting interfaces, this one allows the user to select more than one field to sort on, so that if two records are equal on the primary sort key, they are next compared by the secondary sort key, and so on.

On the plus side, this interface communicates one aspect of its model very well. Each column is a set of radio buttons, clearly grouped together (both by **gestalt proximity** and by an explicit raised border). Radio buttons have the property that only one can be selected at a time. So the interface has a clear **affordance** for picking only one field for each sort key.

But, on the down side, the radio buttons don't afford making NO choice. What if I want to sort by only one key? I have to resort to a trick, like setting all three sort keys to the same field. The interface model clearly doesn't map correctly to the task it's intended to perform. In fact, unlike typical model mismatch problems, both the user and the system have to adjust to this silly interface model – the user by selecting the same field more than once, and the system by detecting redundant selections in order to avoid doing unnecessary sorts.

The interface also fails on **minimalist** design grounds. It wastes a huge amount of screen real estate on a two-dimensional matrix, which doesn't convey enough information to merit the cost. The column labels are similarly redundant; "sort option" could be factored out to a higher level heading.

Today's Topics

- More output
- Widgets
- Toolkit layering

Today we'll finish up our survey of user interface implementation.

Last lecture, we discussed the component model, stroke model, and pixel model, and observed that virtually every graphical user interface uses all three of these models for output. The main decision is, at what point in a GUI application do you make the switch from one model to the next. The switch from component model to stroke model occurs at the leaves of the view hierarchy (although parent containers may also draw strokes, of course). The switch from stroke model to pixel model usually occurs within the system's graphics library.

Drawing Surface

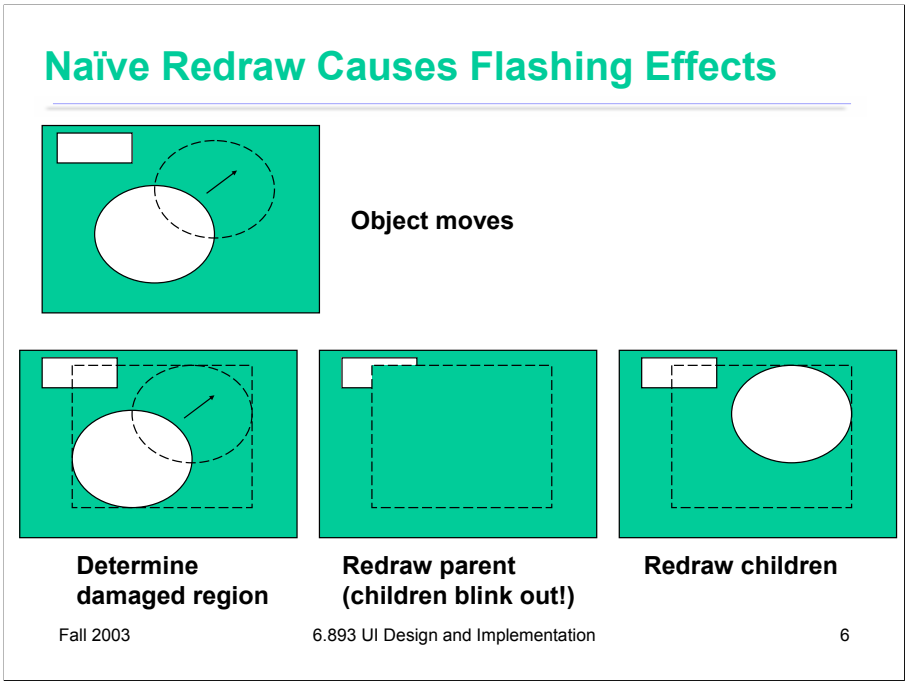
- Every view component has a method for drawing itself
  - In Java: void paint (Graphics g)
- Graphics is a reference to a drawing surface
  - Also called drawable (X Windows), GDI (MS Win)
- The drawing surface itself may vary:
  - Screen
  - Memory buffer
  - Print driver
  - File
  - Remote screen

We've already considered the component model (i.e., the view hierarchy) in some detail. So now, let's look at stroke models.

Every stroke model has some notion of a **drawing surface**. This is the object that is passed to a view component when it's told to draw itself. In Java, it's called Graphics or Graphics2D; it has other names in other toolkits. This object's interface provides the drawing calls of the stroke model, and is responsible for converting these strokes into onscreen pixels.

The screen is only one place where drawing might go, however. The same drawing surface interface can also be used to draw to a memory buffer, which is an array of pixels just like the screen. Unlike the screen, however, a memory buffer can have arbitrary dimensions.

Another target is a printer driver, which forwards the drawing instructions on to a printer. Although most printers have a pixel model internally (when the ink actually hits the paper), the driver virtually always uses a stroke model to communicate with the printer, for compact transmission. Postscript, for example, is a stroke model.

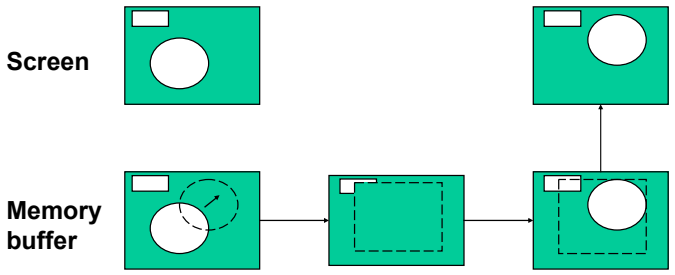


Let's look at one common reason we might want to draw on a memory buffer. If we draw a view hierarchy directly to the screen, then moving view objects can make the screen appear to flash – objects flickering while they move, and nearby objects flickering as well.

This is a side-effect of the automatic damage/redraw algorithm. When an object moves, it needs to be erased from its original position and drawn in its new position. The erasure is done by redrawing all the objects in the view hierarchy that intersect this damaged region. If the drawing is done directly on the screen, this means that all the objects in the damaged region temporarily *disappear*, before being redrawn. Depending on how screen refreshes are timed with respect to the drawing, these partial redraws may be briefly visible on the monitor, causing a perceptible flicker.

Double-Buffering

- Double-buffering solves the flashing problem



The double-buffering technique solves this flickering problem. An identical copy of the screen contents is kept in a memory buffer. (In practice, this may be only the part of the screen belonging to some subtree of the view hierarchy that cares about double-buffering.) This memory buffer is used as the drawing surface for the automatic damage/redraw algorithm. After drawing is complete, the damaged region is just copied to screen as a block of pixels. Double-buffering reduces flickering for two reasons: first, because the pixel copy is generally faster than redrawing the view hierarchy, so there's less chance that a screen refresh will catch it half-done; and second, because unmoving objects that happen to be caught, as innocent victims, in the damaged region are never erased from the screen, only from the memory buffer.

It's a waste for every individual view to double-buffer itself. If any of your ancestors is double-buffered, then you'll derive the benefit of it. So double-buffering is usually applied to top-level windows.

Why is it called double-buffering? Because it used to be implemented by two interchangeable buffers in video memory. While one buffer was showing, you'd draw the next frame of animation into the other buffer. Then you'd just tell the video hardware to switch which buffer it was showing, a very fast operation that required no copying and was done during the CRT's vertical refresh interval so it produced no flicker at all.

Graphics Context

- Graphics context encapsulates lots of drawing parameters
  - Foreground & background colors
  - Line width, line end caps, join styles
  - Fill color, fill pattern (also called stipple)
  - Font
  - Clipping region
  - Coordinate system
- Otherwise we'd have to pass all these rarely-changed parameters on every drawing call:
  - g.drawLine (x1, y1, x2, y2, color, width, end caps, …)
- Graphics context is usually combined with the drawing surface (Java, MS Win)
  - Sometimes a separate object passed with drawing calls (X Windows graphics context, Amulet Am_Style)
- Context can usually be saved & restored

Most stroke models include some kind of a **graphics context**, an object that bundles up drawing parameters like color, line properties (width, end cap, join style), fill properties (pattern), and font.

Most systems actually combine the graphics context with the drawing surface itself. Java's Graphics object is an excellent example of this approach. In other toolkits, the graphics context is an independent object that is passed along with drawing calls.

When the context is embedded in the drawing surface, the surface must provide some way to save and restore the context. A key reason for this is so that parent views can pass the drawing surface down to a child's draw method without fear that the child will change the graphics context. In Java, for example, the context can be saved by Graphics.create(), which makes a copy of the Graphics object. Notice that this only duplicates the graphics context; it doesn't duplicate the drawing surface, which is still the same.

Coordinate Transformation

- Most GUI systems put default origin in top left corner
  - Except Postscript: origin is bottom left
- 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

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.

Color Models

- RGB: cube
  - Red, green, blue
- HSV: hexagonal cone
  - Hue: kind of color
    - Angle around cone
  - Saturation: amount of pure color
    - 0% = gray, 100% = pure color
  - Value: brightness
    - 0% = dark, 100% = bright
- HLS: double-hexagonal cone
  - Hue, lightness, saturation
  - Pulls up center of HSV model, so that only white has lightness 1.0 and pure colors have lightness 0.5
- Cyan-Magenta-Yellow(-Black)
  - Used for printing, where pigments absorb wavelengths instead of generating them

We learned a bit about how humans perceive color back when we talked about human capabilities. Now let's look at how colors are represented in GUI software.

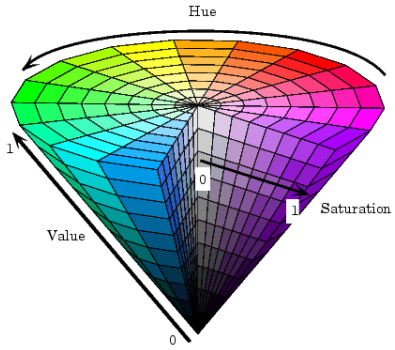
At the lowest level, the RGB model rules. The RGB model is a unit cube, with (0,0,0) corresponding to black, (1, 1, 1) corresponding to white, and the three dimensions measuring levels of red, green, and blue. The RGB model is used directly by CRT and LCD monitors for display, since each pixel in a monitor has separate red, green, and blue components.

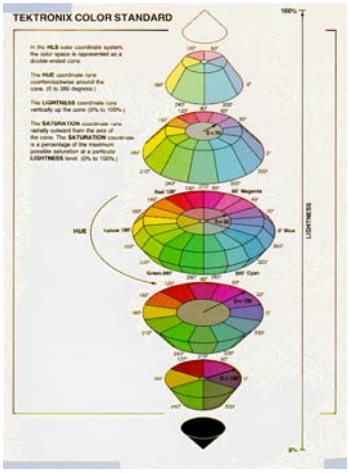
HSV (hue, saturation value) is a better model for how humans perceive color, and more useful for building usable interfaces. HSV is a cone. We've already encountered hue and value in our discussion of visual variables. Saturation is the degree of color, as opposed to grayness. Colors with zero saturation are shades of gray; colors with 100% saturation are pure colors.

HLS (hue, lightness, saturation) is a symmetrical relative of the HSV model, which is elegant. See the pictures on the next page.

Finally, the CMYK (cyan, magenta, yellow, and sometimes black) is similar to the RGB model, but used for print colors.

HSV & HLS





Here are some pictures of the HSV and HLS models.

Widgets

- Reusable user interface components
  - Also called controls, interactors
- Examples
  - Buttons, checkboxes, radio buttons
  - List boxes, combo boxes, drop-downs
  - Menus, toolbars
  - Scrollbars, splitters, zoomers
  - One-line text, multiline text, rich text
  - Trees, tables
  - Simple dialogs

Widgets are the last part of user interface toolkits we'll look at. Widgets are a success story for user interface software, and for object-oriented programming in general. Many GUI applications derive substantial reuse from widgets in a toolkit.

Widget reuse is beneficial in two ways, actually. First are the software engineering benefits, like shorter development time. But second are usability benefits, since widget reuse increases consistency among the applications on a platform.

Widget Design

- Widget is a view + controller
  - Embedded model
    - Application data must be copied into the widget
    - Selections must be copied out'
  - Linked model
    - Application provides model satisfying an interface
    - Enables "data-bound" widgets, e.g. a table showing thousands of database rows, or a combo box with thousands of choices

Widgets generally combine a view and a controller into a single tightly-coupled object. For the widget's model, however, there are two common approaches. One is to fuse the model into the widget as well, making it a little MVC complex. With this embedded model approach, application data must be copied into the widget to initialize it. When the user interacts with the widget, the user's changes or selections must be copied back out.

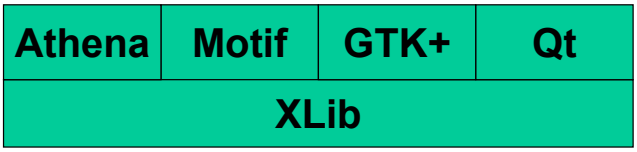
The other alternative is to leave the model separate from the widget, with a well-defined interface that the application can implement.

Embedded models are usually easier for the developer to understand and use for simple interfaces, but suffer from serious scaling problems. For example, suppose you want to use a table widget to show the contents of a database. If the table widget had an embedded model, you would have to fetch the entire database and load it into the table widget, which may be prohibitively slow and memory-intensive. Furthermore, most of this is wasted work, since the user can only see a few rows of the table at a time. With a well-designed linked model, the table widget will only request as much of the data as it needs to display.

The linked model idea is also called **data binding**.

Toolkits

- User interface toolkit consists of:
  - Widgets
  - View hierarchy
  - Stroke drawing
  - Input handling
- Toolkits are often layered

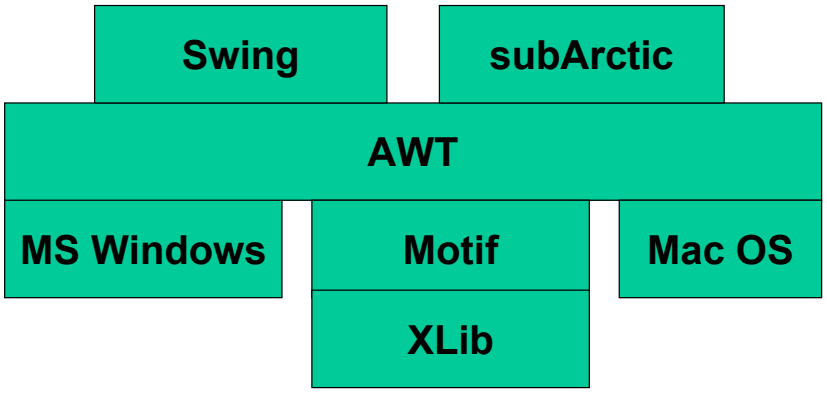


By now, we've looked at all the basic pieces of a user interface toolkit: widgets, view hierarchy, stroke drawing, and input handling. Every modern GUI toolkit provides these pieces in some form. Microsoft Windows, for example, has widgets (e.g., buttons, menus, text boxes), a view hierarchy (consisting of *windows* and *child windows*), a stroke drawing package (GDI), and input handling (messages sent to a *window procedure*).

User interface toolkits are often built on top of other toolkits, sometimes for portability or compatibility across platforms, and sometimes to add more powerful features, like a richer stroke drawing model or different widgets.

X Windows demonstrates this layering technique. The view hierarchy, stroke drawing, and input handling are provided by a low-level toolkit called XLib. But XLib does not provide widgets, so several toolkits are layered on top of XLib to add that functionality: Athena widgets and Motif, among others. More recent X-based toolkits (GTK+ and Qt) not only add widgets to XLib, but also hide XLib's view hierarchy, stroke drawing, and input handling with newer, more powerful models, although these models are implemented internally by calls to XLib.

Toolkit Layering



Here's what the layering looks like for some common Java user interface toolkits.

AWT (Abstract Windowing Toolkit) was the first Java toolkit. Although its widget set are rarely used today, AWT continues to provide drawing and input handling to more recent Java toolkits.

Swing is the second-generation Java toolkit, which appeared in the Java API starting in Java 1.2. Swing adds a new view hierarchy (JComponent) derived from AWT's view hierarchy (Component and Container). It also replaces AWT's widget set with new widgets that use the new view hierarchy.

subArctic was a research toolkit developed at Georgia Tech. Like Swing, subArctic relies on AWT for drawing and input handling, but provides its own widgets and views.

Not shown in the picture is SWT, IBM's Standard Windowing Toolkit. Like AWT, SWT is implemented directly on top of the native toolkits. It provides different (and incompatible) interfaces for widgets, views, drawing, and input handling.

Cross-Platform Widget Approaches

- AWT
  - Use native widgets, but only those common to all platforms
  - Consistent with other platform apps
- Swing, Amulet
  - Reimplement all widgets
  - Not constrained by least common denominator
  - Consistent behavior for app across platforms
  - Widget code is identical for all platforms
- SWT
  - Use native widgets where available
  - Reimplement missing native widgets

Cross-platform toolkits face a special issue: should the native widgets of each platform be reused by the toolkit? One reason to do so is to preserve consistency with other applications on the same platform, so that applications written for the cross-platform toolkit look and feel like native applications. This is what we've been calling external consistency.

One reason NOT to reuse the native widgets is so that the application looks and behaves consistently with itself across platforms – a variant of internal consistency, if you consider all the instantiations of an application on various platforms as being part of the same system. Cross-platform consistency makes it easier to deliver a well-designed, usable application on all platforms.

Another problem is that native widgets may not exist for all the widgets the cross-platform toolkit wants to provide. AWT throws up its hands at this problem, providing only the widgets that occur on every platform AWT runs on: e.g., buttons, menus, list boxes, text boxes. SWT takes the opposite tack; instead of limiting itself to the **intersection** of the native widget sets, SWT strives to provide the **union** of the native widget sets. SWT uses a native widget if it's available, but reimplements it if it's missing.