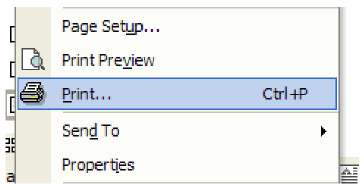
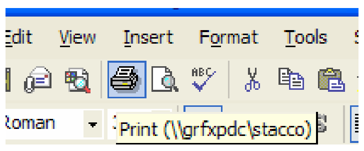
Lecture 10: Input and Output

UI Hall of Fame or Shame?

- Three ways to print in Microsoft Office
  - File/Print menu item
  - Print toolbar button
  - Ctrl-P keyboard shortcut





There are three ways to print in Microsoft Office applications: a menu command, a toolbar button, and a keyboard shortcut. That's OK, because we want to provide shortcuts for experienced users (**flexibility & efficiency**).

Unfortunately, the three commands don't all do the same thing:

•File/Print brings up the Print dialog

•The Print toolbar button prints immediately using the latest print settings

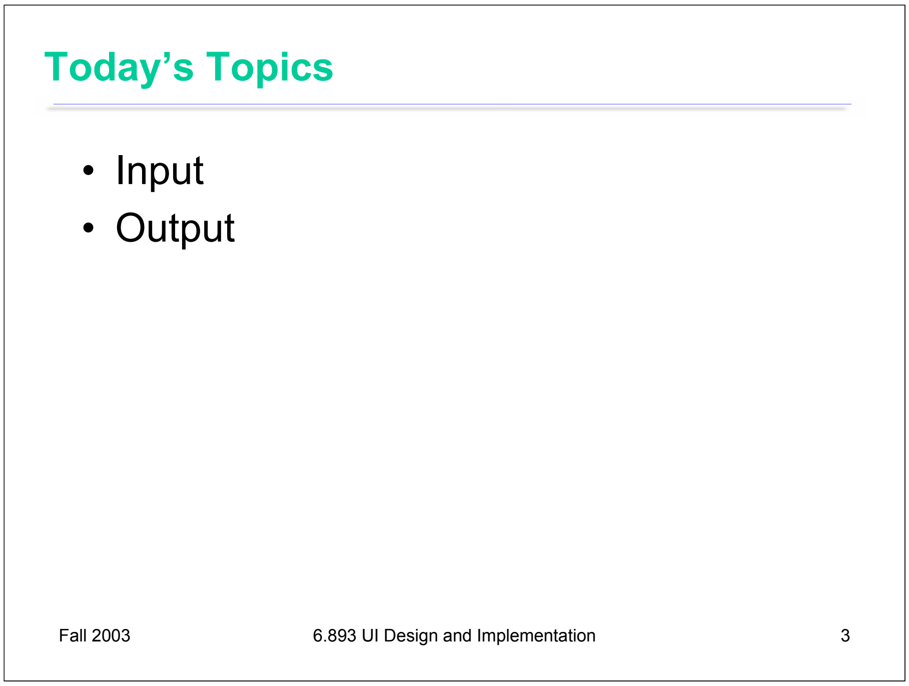
•Ctrl-P brings up the Print dialog

So there are three commands named Print that do different things (**internal inconsistency**).

But the toolbar button's behavior is useful! There *should* be an easy way to just print. Why? **Flexibility and efficiency**. Most of the time, for most users, for most copies of a document, you only want to print it one way. You don't need to specify the printer (most people have only 1), you don't need to pick color or grayscale or duplex or paper source. The default settings, or the last settings you chose for this particular document, should work. Don't ask me all those questions, just give me a printout! That's what the toolbar button is trying to do.

But the fact that all three are simply named Print is disturbing. Furthermore, there's a natural hierarchy among these shortcuts, starting with the menu item, which is clearly labeled knowledge in the world; then the toolbar button, which is always visible, faster to reach than the menu item, less descriptive than a label, but still allows you to recognize rather than recall; and finally the keyboard shortcut, which is mnemonic (Ctrl-P for Print) but requires knowledge in the head.

The print-now command is **unnaturally mapped** on this hierarchy. It's mapped to the medium-shortcut toolbar button, but not to the extreme-shortcut Ctrl-P. As a result, I (for one) hesitate to use either of the print shortcuts, because I'm frequently surprised by what it does.

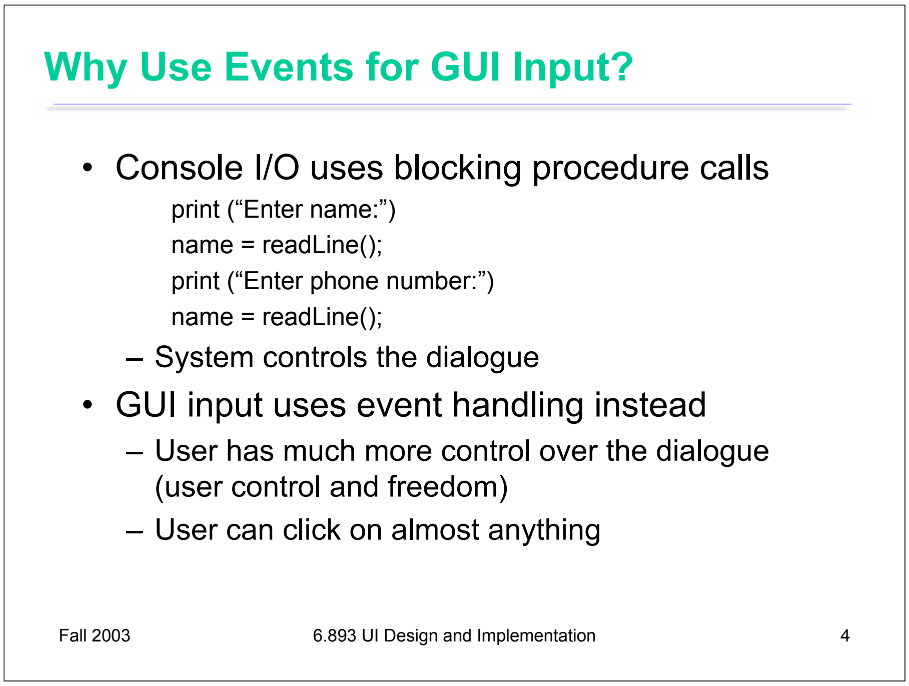


Today's Topics

- Input
- Output

Today's lecture continues our look into the mechanics of implementing user interfaces, by looking at **input** and **output** in more detail.

Our goal for these implementation lectures is not to teach any one particular GUI system or toolkit, but to give a survey of the issues involved in GUI programming and the range of solutions adopted by various systems. Presumably you've already encountered at least one GUI toolkit, probably Java Swing. These lectures should give you a sense for what's common and what's unusual in the toolkit you already know, and what you might expect to find when you pick up another GUI toolkit.


Why Use Events for GUI Input?

- Console I/O uses blocking procedure calls

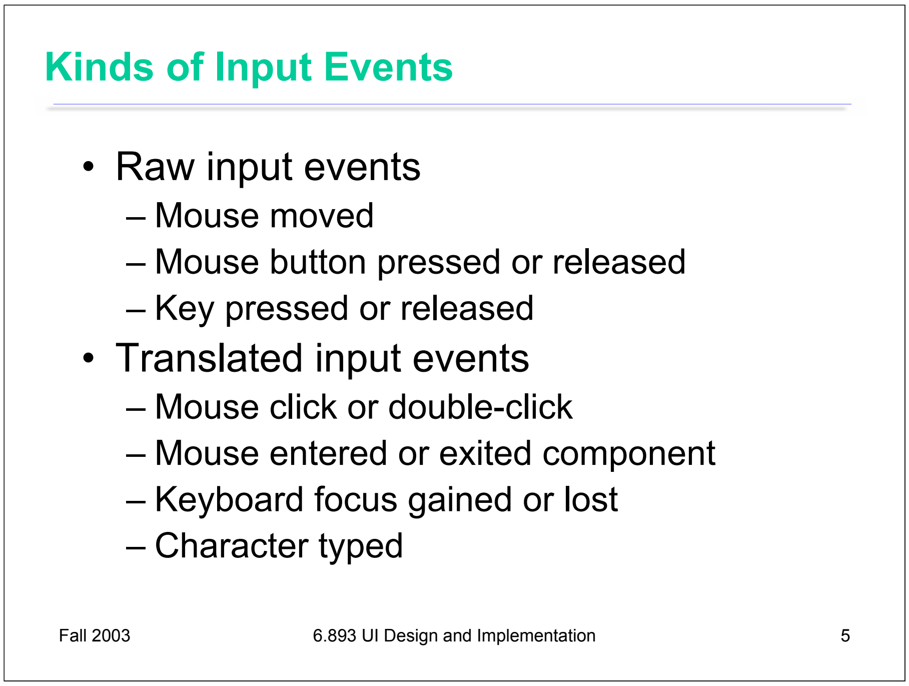
```
print (“Enter name:”)
name = readLine();
print (“Enter phone number:”)
name = readLine();
```

  – System controls the dialogue
- GUI input uses event handling instead
  – User has much more control over the dialogue (user control and freedom)
  – User can click on almost anything

Virtually all GUI toolkits use event handling for input. Why? Recall, when you first learned to program, you probably wrote user interfaces that printed a prompt and then waited for the user to enter a response. After the user gave their answer, you produced another prompt and waited for another response. Command-line interfaces (e.g. the Unix shell) and menu-driven interfaces (e.g., Pine) have interfaces that behave this way. In this user interface style, the system has complete control over the dialogue – the order in which inputs and outputs will occur.

Interactive graphical user interfaces can’t be written this way – at least, not if they care about giving the user control and freedom. One of the biggest advantages of GUIs is that a user can click anywhere on the window, invoking any command that’s available at the moment, interacting with any view that’s visible. In a GUI, the balance of power in the dialogue swings strongly over to the user’s side.

As a result, GUI programs can’t be written in a synchronous, prompt-response style. A component can’t simply take over the entire input channel to wait for the user to interact with it, because the user’s next input may be directed to some other component on the screen instead. So GUI programs are designed to handle input asynchronously, receiving it as events.



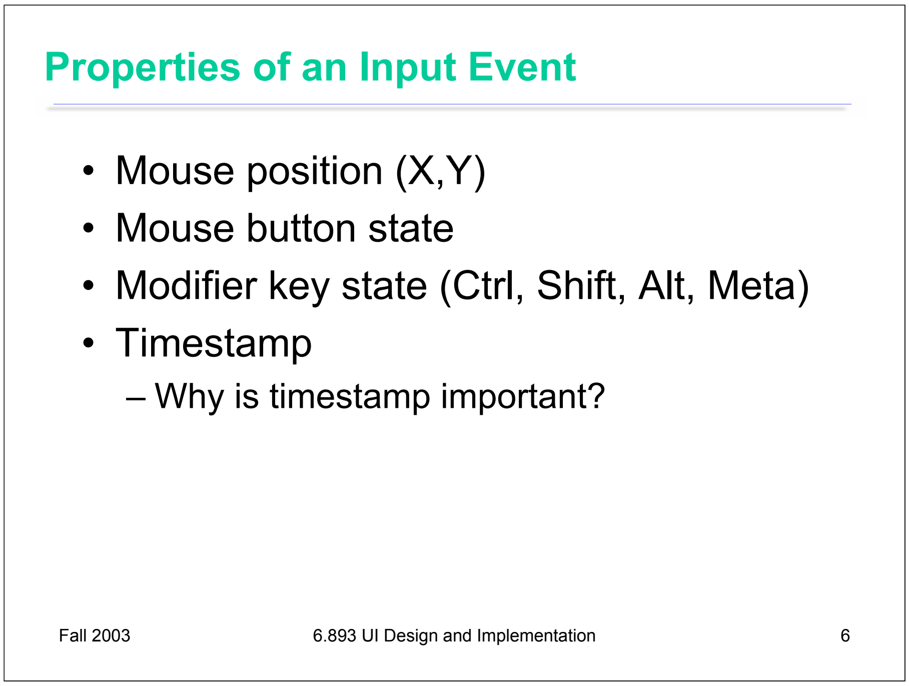
Kinds of Input Events

- Raw input events
  - Mouse moved
  - Mouse button pressed or released
  - Key pressed or released
- Translated input events
  - Mouse click or double-click
  - Mouse entered or exited component
  - Keyboard focus gained or lost
  - Character typed

There are two major categories of input events: raw and translated.

A raw event comes right from the device driver. Mouse movements, mouse button down and up, and keyboard key down and up are the raw events seen in almost every capable GUI system. A toolkit that does not provide separate events for down and up is poorly designed, and makes it difficult or impossible to implement input effects like drag-and-drop or video game controls.

For many GUI components, the raw events are too low-level, and must be translated into higher-level events. For example, a mouse button press and release is translated into a mouse click event (assuming the mouse didn't move much between press and release – if it did, these events would be translated into a drag rather than a click). Key down and up events are translated into character typed events, which take modifiers into account to produce an ASCII character rather than a keyboard key. If you hold a key down, multiple character typed events may be generated by an autorepeat mechanism. Mouse movements and clicks also translate into keyboard focus changes. When a mouse movement causes the mouse to enter or leave a component's bounding box, entry and exit events are generated, so that the component can give feedback – e.g., visually highlighting a button, or changing the mouse cursor to a text I-bar or a pointing finger.

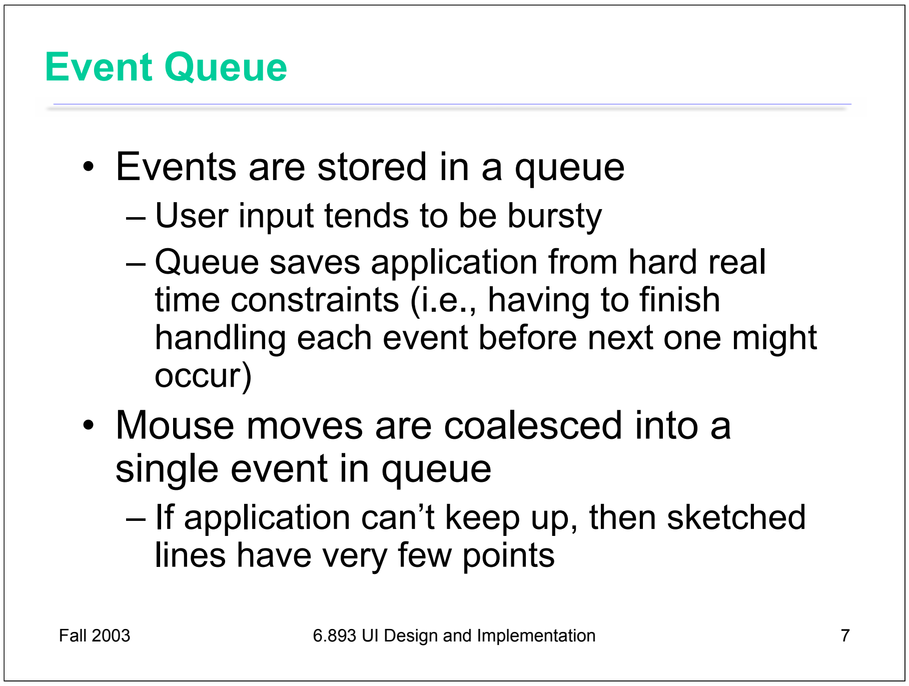


Properties of an Input Event

- Mouse position (X,Y)
- Mouse button state
- Modifier key state (Ctrl, Shift, Alt, Meta)
- Timestamp
  - Why is timestamp important?

Input events have some or all of these properties. On most systems, all events include the modifier key state, since some mouse gestures are modified by Shift, Control, and Alt. Some systems include the mouse position and button state on all events; some put it only on mouse-related events.

The timestamp indicates when the input was received, so that the system can time features like autorepeat and double-clicking. It is essential that the timestamp be a property of the event, rather than just read from the clock when the event is handled. Events are stored in a queue, and an event may languish in the queue for an uncertain interval until the application actually handles it.



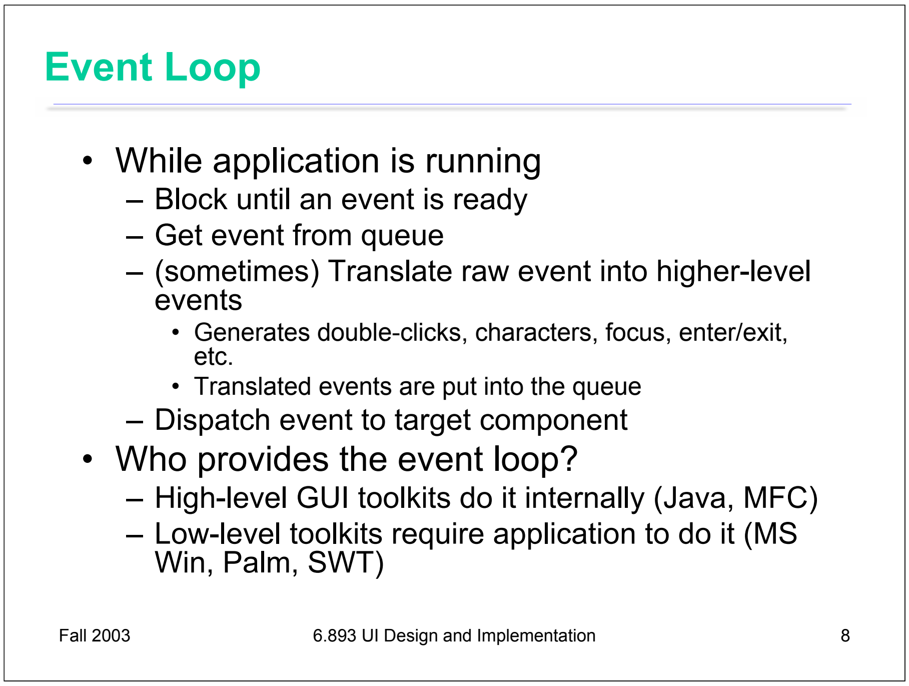
Event Queue

- Events are stored in a queue
  - User input tends to be bursty
  - Queue saves application from hard real time constraints (i.e., having to finish handling each event before next one might occur)
- Mouse moves are coalesced into a single event in queue
  - If application can't keep up, then sketched lines have very few points

User input tends to be bursty – many seconds may go by while the user is thinking, followed by a flurry of events. The event queue provides a buffer between the user and the application, so that the application doesn't have to keep up with each event in a burst. Recall that perceptual fusion means that the system has 100 milliseconds in which to respond.

Edge events (button down and up events) are all kept in the queue unchanged. But multiple events that describe a continuing state – in particular, mouse movements – may be **coalesced** into a single event with the latest known state. Most of the time, this is the right thing to do. For example, if you're dragging a big object across the screen, and the application can't repaint the object fast enough to keep up with your mouse, you don't want the mouse movements to accumulate in the queue, because then the object will lag behind the mouse pointer, diligently (and foolishly) following the same path your mouse did.

Sometimes, however, coalescing hurts. If you're sketching a freehand stroke with the mouse, and some of the mouse movements are coalesced, then the stroke may have straight segments at places where there should be a smooth curve. If application delays are bursty, then coalescing may hurt even if your application can usually keep up with the mouse.



Event Loop

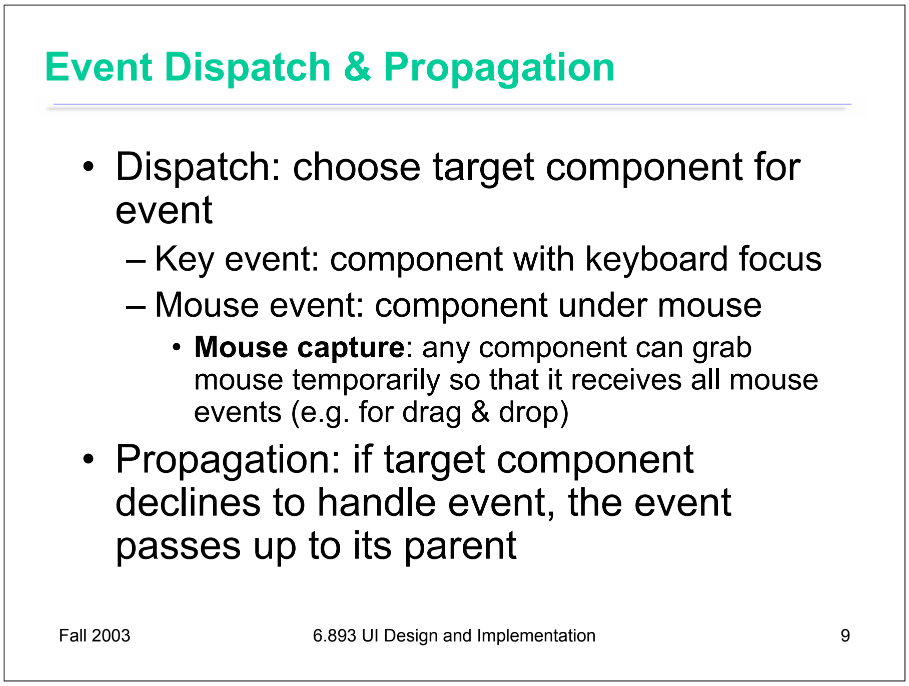
- While application is running
  - Block until an event is ready
  - Get event from queue
  - (sometimes) Translate raw event into higher-level events
    - Generates double-clicks, characters, focus, enter/exit, etc.
    - Translated events are put into the queue
  - Dispatch event to target component
- Who provides the event loop?
  - High-level GUI toolkits do it internally (Java, MFC)
  - Low-level toolkits require application to do it (MS Win, Palm, SWT)

The event loop reads events from the queue and dispatches them to the appropriate components in the view hierarchy. On some systems (notably Microsoft Windows), the event loop also includes a call to a function that translates raw events into higher-level ones. On most systems, however, translation happens when the raw event is added to the queue, not when it is removed.

Every GUI program has an event loop in it somewhere. Some toolkits require the application programmer to write this loop (e.g., Win32); other toolkits have it built-in (e.g., Java Swing).

Unfortunately, Java's event loop is written as essentially an infinite loop, so the event loop thread never cleanly exits. As a result, the normal clean way to end a Java program – waiting until all the threads are finished – doesn't work for GUI programs. The only way to end a Java GUI program is System.exit(). This despite the fact that Java best practices say *not* to use System.exit(), because it doesn't guarantee to garbage collect and run finalizers.

Swing lets you configure your application's main JFrame with EXIT_ON_CLOSE behavior, but this is just a shortcut for calling System.exit().

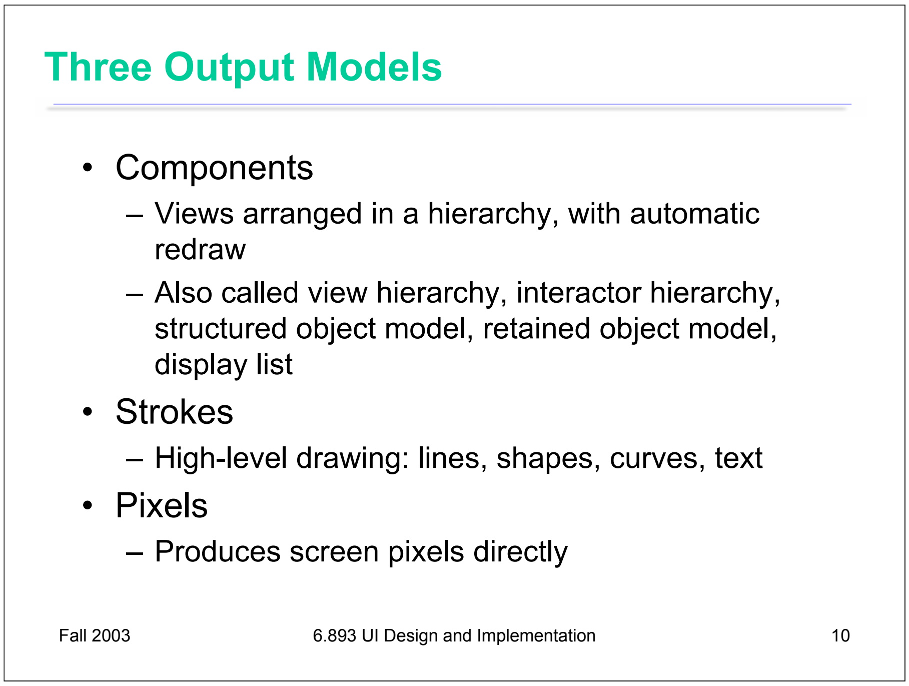


Event Dispatch & Propagation

- Dispatch: choose target component for event
  - Key event: component with keyboard focus
  - Mouse event: component under mouse
    - **Mouse capture**: any component can grab mouse temporarily so that it receives all mouse events (e.g. for drag & drop)
- Propagation: if target component declines to handle event, the event passes up to its parent

Event dispatch chooses a component to receive the event. Key events are sent to the component with the keyboard focus, and mouse events are generally sent to the component under the mouse. An exception is **mouse capture**, which allows any component to grab all mouse events (essentially a mouse analogue for keyboard focus). Mouse capture is done automatically by Java when you hold down the mouse button to drag the mouse. Other UI toolkits give the programmer direct access to mouse capture – in the Windows API, for example, you'll find a SetMouseCapture function.

If the target component declines to handle the event, the event propagates up the view hierarchy until some component handles it. If an event bubbles up to the top without being handled, it is ignored.



Three Output Models

- Components
  - Views arranged in a hierarchy, with automatic redraw
  - Also called view hierarchy, interactor hierarchy, structured object model, retained object model, display list
- Strokes
  - High-level drawing: lines, shapes, curves, text
- Pixels
  - Produces screen pixels directly

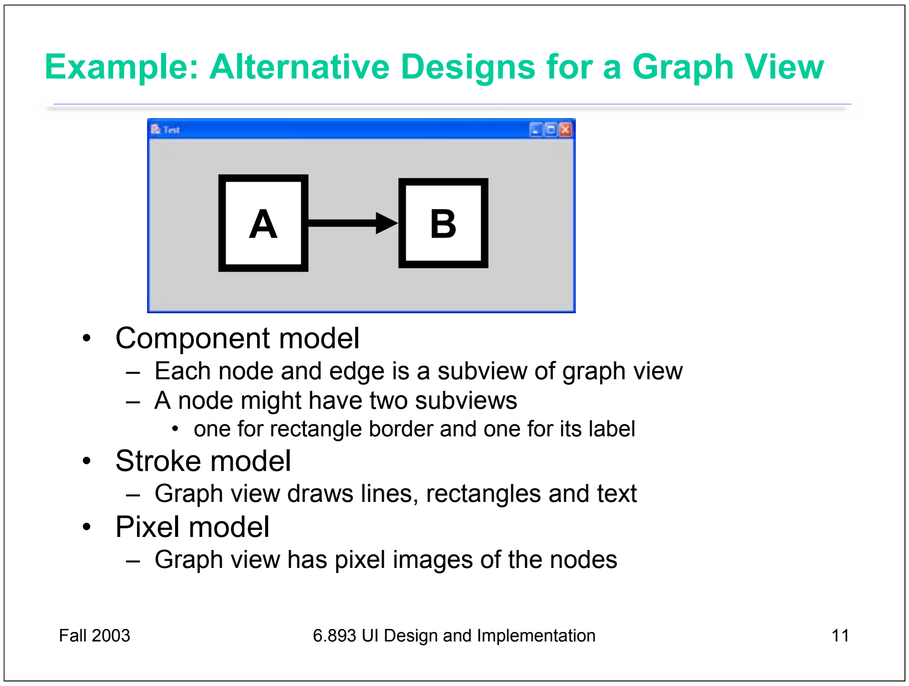
Now we'll turn our attention to output. There are basically three ways to structure the output of a GUI application.

Components is the same as the view hierarchy we've been talking about. The pieces of the display are represented by view objects arranged in a spatial hierarchy, with automatic redraw propagating down the hierarchy. There have been many names for this idea over the years; the GUI community hasn't managed to settle on a single preferred term.

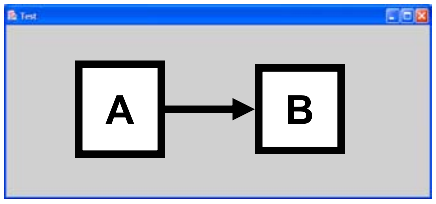
Strokes draws output by making calls to high-level drawing primitives, like drawLine, drawRectangle, drawArc, and drawText.

Pixels produces screen pixels directly, by treating the screen as an array of pixels and setting the pixels directly.

All three output models appear in virtually every modern GUI application. The component model always appears at the very top level, for windows, and often for components within the windows as well. At some point, we reach the leaves of the view hierarchy, and the leaf views draw themselves with stroke calls. A graphics package then converts those strokes into pixels for display on the screen.



Example: Alternative Designs for a Graph View



- Component model
  - Each node and edge is a subview of graph view
  - A node might have two subviews
    - one for rectangle border and one for its label
- Stroke model
  - Graph view draws lines, rectangles and text
- Pixel model
  - Graph view has pixel images of the nodes

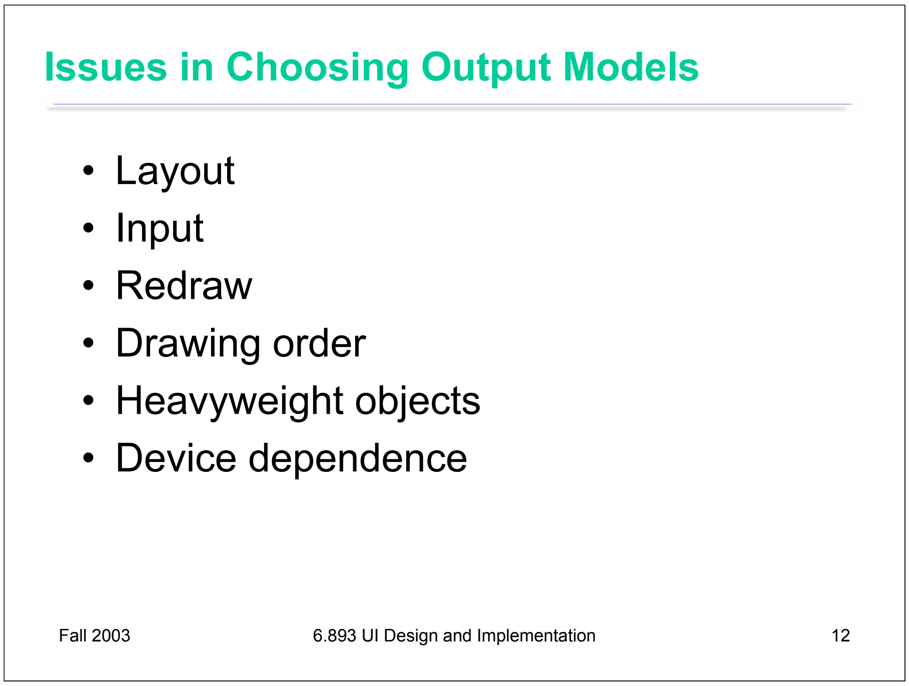
Since every application uses all three models, the design question becomes: at which points in your application do you want to step down into a lower-level output model? Here's an example. Suppose you want to build a view that displays a graph of nodes and edges.

One approach would represent each node and edge in the graph by a component. Each node in turn might have two components, a rectangle and a label. Eventually, you'll get down to primitive components available in your GUI toolkit. (Most GUI toolkits provide a label component; most don't provide a primitive rectangle component. One notable exception is Amulet, which has component equivalents for all the common drawing primitives.) This would be a **pure component model**, at least from your application's point of view – stroke output and pixel output would still happen, but inside primitive components.

Alternatively, the top-level window might have *no* subcomponents. Instead, it would draw the entire graph by a sequence of stroke calls: drawRectangle for the node outlines, drawText for the labels, drawLine for the edges. This would be a **pure stroke model**.

Finally, your graph view might bypass stroke drawing and set pixels in the window directly. The text labels might be assembled by copying character images to the screen. This **pure pixel model** is rarely used nowadays, because it's the most work for the programmer, but it used to be the only way to program graphics.

Hybrid models for the graph view are certainly possible, in which some parts of the output use one model, and others use another model. The graph view might use components for nodes, but draw the edges itself as strokes. It might draw all the lines itself, but use label components for the text.



Issues in Choosing Output Models

- Layout
- Input
- Redraw
- Drawing order
- Heavyweight objects
- Device dependence

Layout: Components remember where they were put, and draw themselves there. They also support automatic layout. With stroke or pixel models, you have to figure out (at drawing time) where each piece goes, and put it there.

Input: Components participate in event dispatch and propagation, and the system automatically does **hit-testing** (determining whether the mouse is over the component when an event occurs) for components, but not for strokes. If a graph node is a component, then it can receive its own click and drag events. If you stroked the node instead, then you have to write code to determine which node was clicked or dragged.

Redraw: The automatic damage/redraw algorithm means that components redraw themselves automatically when they have to. Furthermore, the redraw algorithm is efficient: it only redraws components whose extents intersect the damaged region. The stroke or pixel model would have to do this test by hand. In practice, most stroked components don't bother, simply redrawing everything whenever some part of the view needs to be redrawn.

Drawing order: It's easy for a parent to draw before (underneath) or after (on top of) all of its children. But it's not easy to interleave parent drawing with child drawing. So if you're using a hybrid model, with some parts of your view represented as components and others as strokes, then the components and strokes generally fall in two separate layers, and you can't have any complicated z-ordering relationships between strokes and components.

Heavyweight objects: Every component must be an object (and even a null object is 20 bytes in Java). As we've seen, the view hierarchy is overloaded not just with drawing functions but also with event dispatch, automatic redraw, and automatic layout, so that further bulks up the class. The flyweight pattern used by InterView's Glyphs can reduce this cost somewhat. But views derived from large amounts of data – say, a 100,000-node graph – generally can't use a component model.

Device dependence: The stroke model is largely device independent. In fact, it's useful not just for displaying to screens, but also to printers, which have dramatically different resolution. The pixel model, on the other hand, is extremely device dependent. A directly-mapped pixel image won't look the same on a screen with a different resolution.