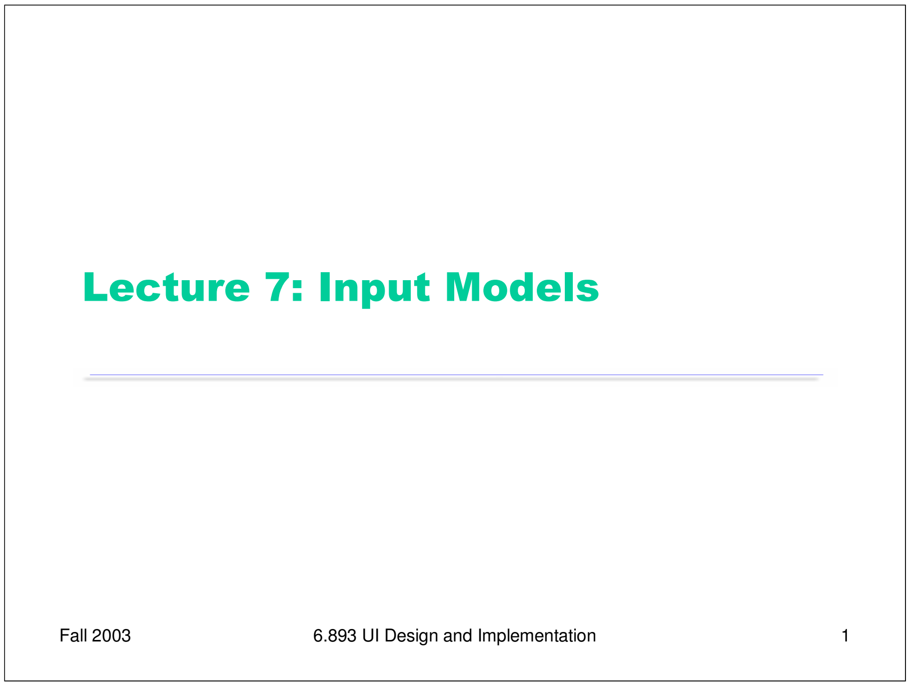
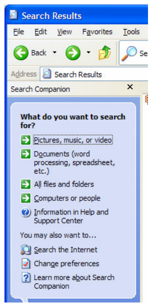
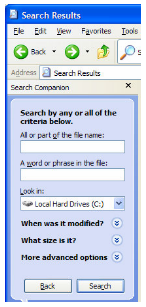


Lecture 7: Input Models

Fall 2003 6.893 UI Design and Implementation 1

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



Fall 2003 6.893 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.

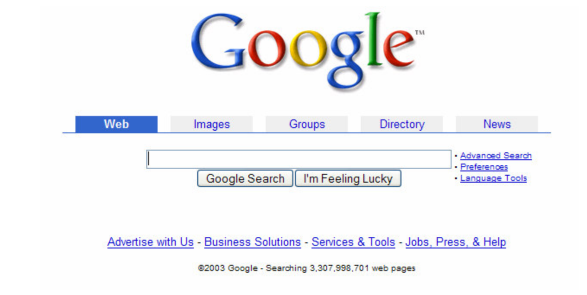
This interface is clearly designed for novice users. Hence the **wizard** design, a fixed sequence of carefully guided steps. And hence the cute animated cartoon dog, which some people in class found condescending by its mere presence. It's still an open question whether cartoon characters like this 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.**

It turns out that "Change preferences" leads to a menu where you can turn off the dog. He doesn't disappear instantly, but turns insouciantly and trots off in a huff. The preferences menu also offers

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

Fall 2003 6.893 UI Design and Implementation 3

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.

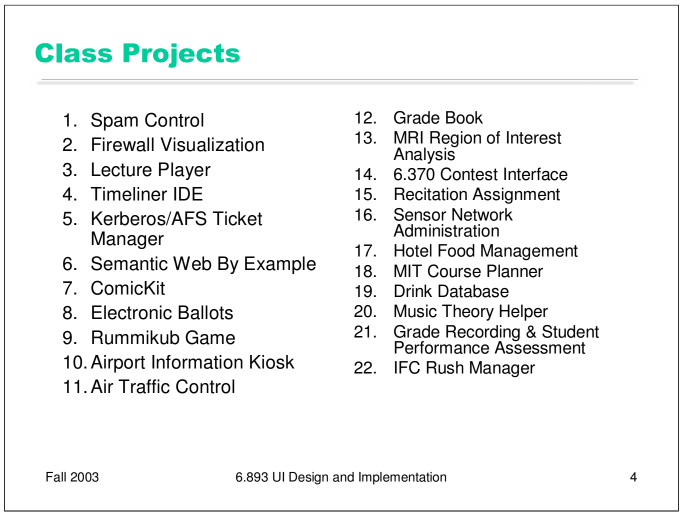
•What should be typed into the text box? It has no caption at all.

•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.



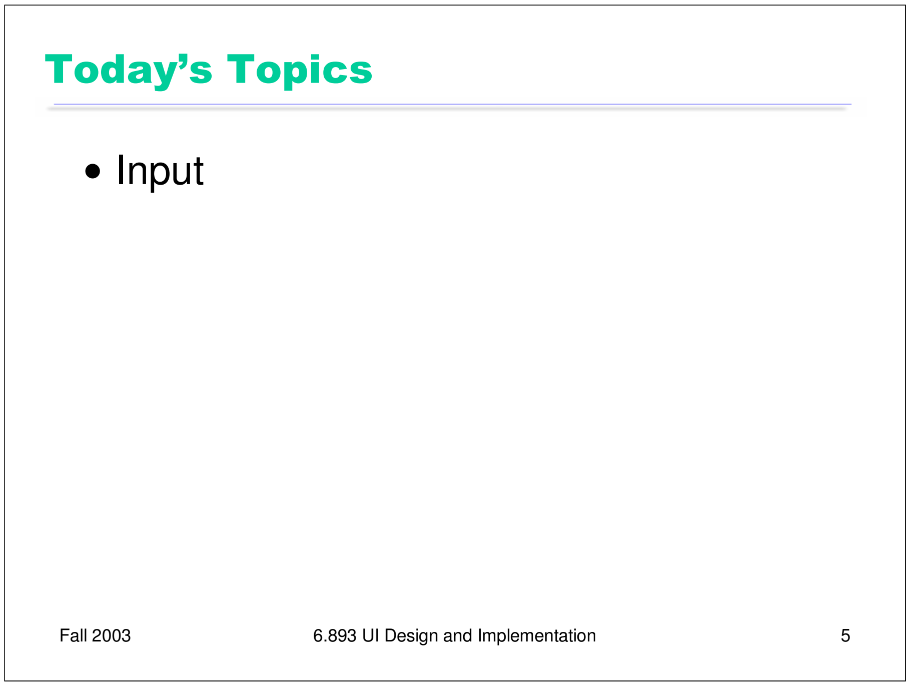
Class Projects

1. Spam Control
2. Firewall Visualization
3. Lecture Player
4. Timeliner IDE
5. Kerberos/AFS Ticket Manager
6. Semantic Web By Example
7. ComicKit
8. Electronic Ballots
9. Rummikub Game
10. Airport Information Kiosk
11. Air Traffic Control
12. Grade Book
13. MRI Region of Interest Analysis
14. 6.370 Contest Interface
15. Recitation Assignment
16. Sensor Network Administration
17. Hotel Food Management
18. MIT Course Planner
19. Drink Database
20. Music Theory Helper
21. Grade Recording & Student Performance Assessment
22. IFC Rush Manager

Fall 2003 6.893 UI Design and Implementation 4

In case you're curious, here are the projects that your classmates are working on. You'll have several opportunities to see what everybody is doing: some in paper prototype testing in 2 weeks, others when you do heuristic evaluation of computer prototypes, and all of them in the final presentations at the end of the course.

Incidentally, the original version of this slide used bullets instead of numbers. Then I thought about one natural question that people would ask – how many projects are there? Although it's *possible* to answer that question from a bulleted list, it's trivial when the list is numbered. Every kind of communication you do has a user interface, whether it's a talk or a paper or a homework assignment. The effectiveness of a communication is strongly influenced by its usability.

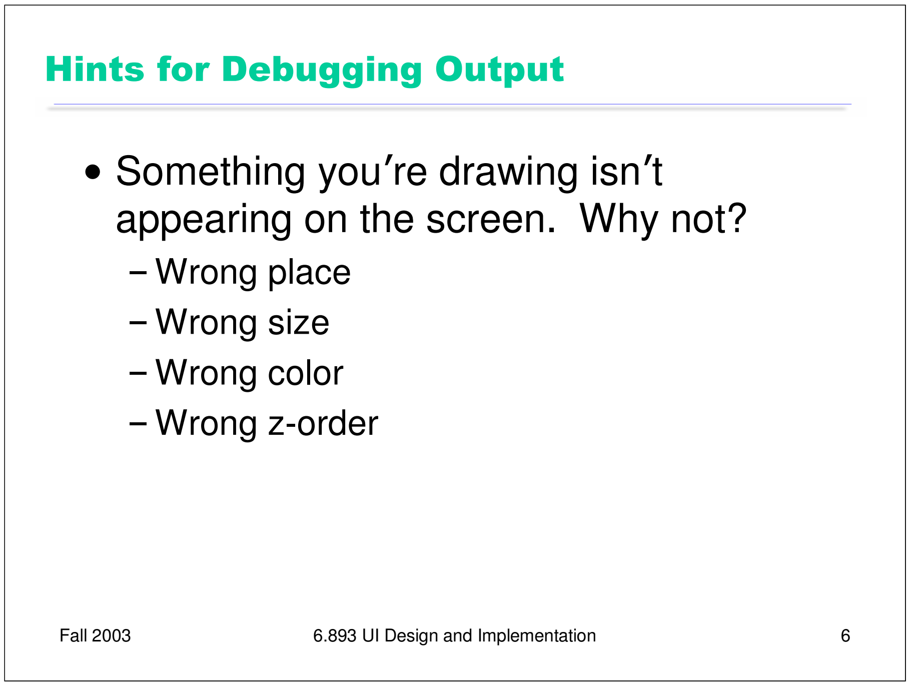


Today's Topics

- Input

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.



Hints for Debugging Output

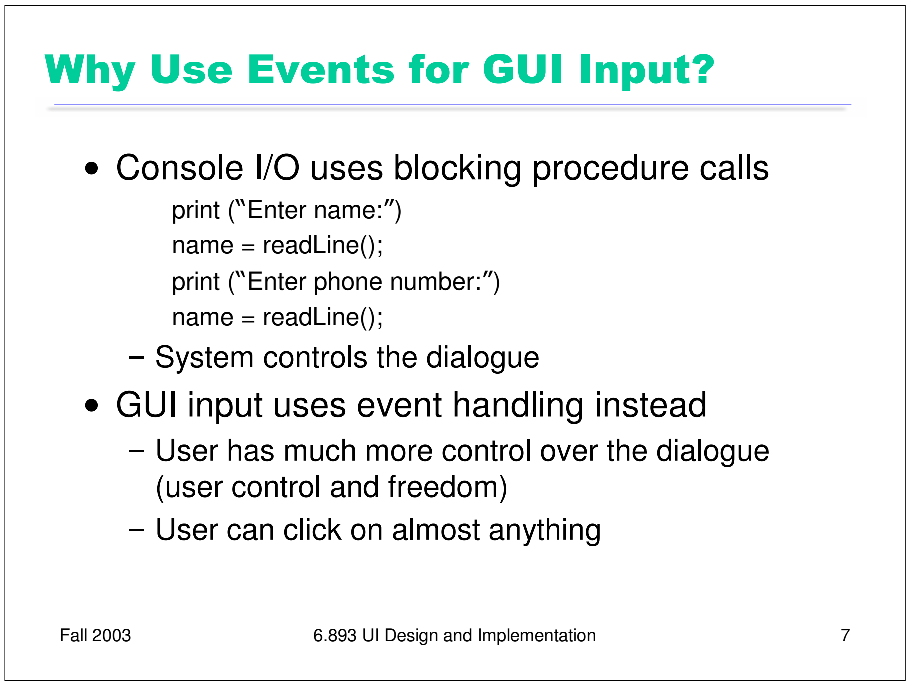
- Something you're drawing isn't appearing on the screen. Why not?
  - Wrong place
  - Wrong size
  - Wrong color
  - Wrong z-order

Wrong place: what's the origin of the coordinate system? What's the scale? Where is the component located in its parent?

Wrong size: if a component has 0 width and 0 height, it will be completely invisible no matter what it tries to draw– everything will be clipped. 0 width and 0 height is the default for all components in Swing – you have to use automatic layout or manual setting to make it a more reasonable size. Check whether the component (and its ancestors) have nonzero sizes.

Wrong color: is the drawing using the same color as the background? Is it using 100% alpha?

Wrong z-order: is something else drawing on top?


Why Use Events for GUI Input?

- Console I/O uses blocking procedure calls

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

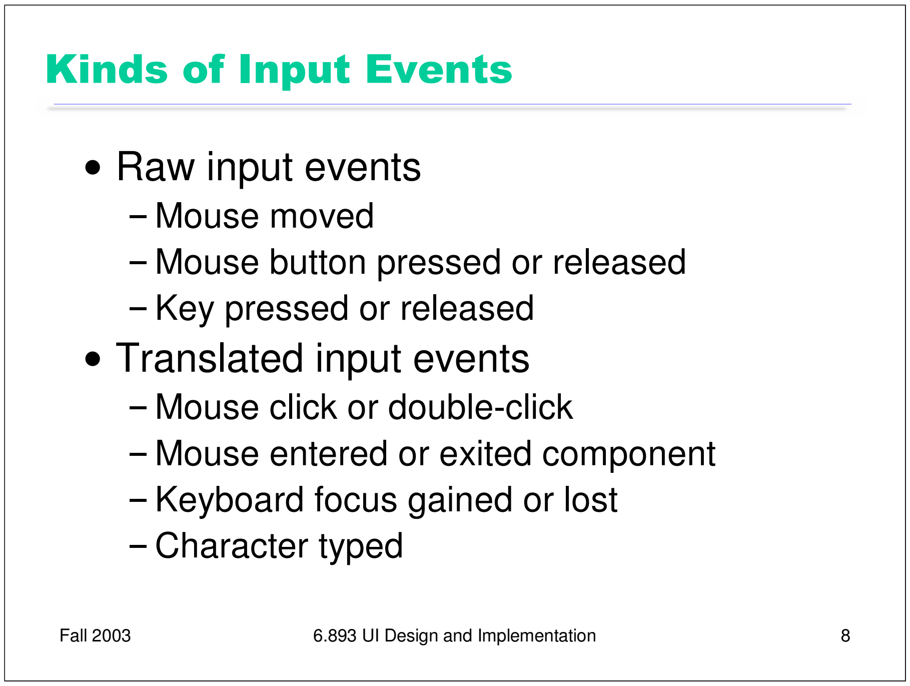
  – System controls the dialogue
- GUI input uses event handling instead
  – User has much more control over the dialogue (user control and freedom)
  – User can click on almost anything

Fall 2003 6.893 UI Design and Implementation 7

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

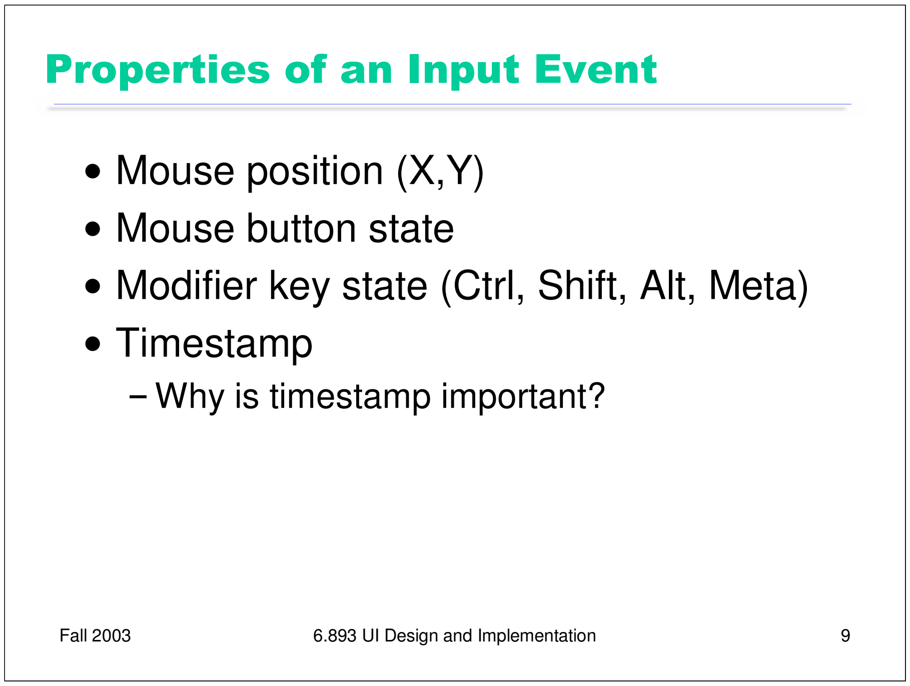
- Raw input events
  - Mouse moved
  - Mouse button pressed or released
  - Key pressed or released
- Translated input events
  - Mouse click or double-click
  - Mouse entered or exited component
  - Keyboard focus gained or lost
  - Character typed

Fall 2003 6.893 UI Design and Implementation 8

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.



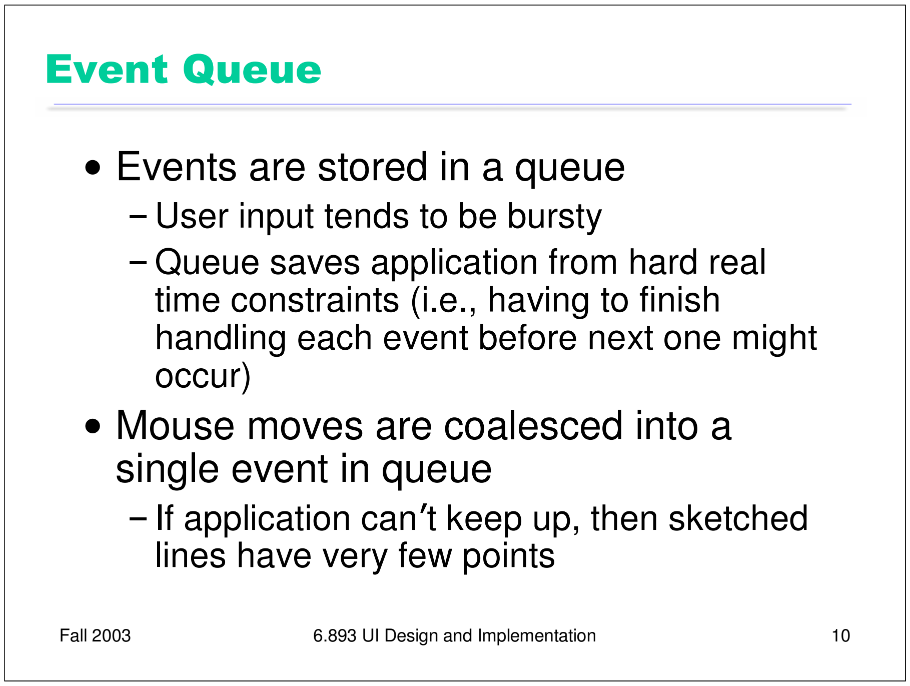
Properties of an Input Event

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

Fall 2003 6.893 UI Design and Implementation 9

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

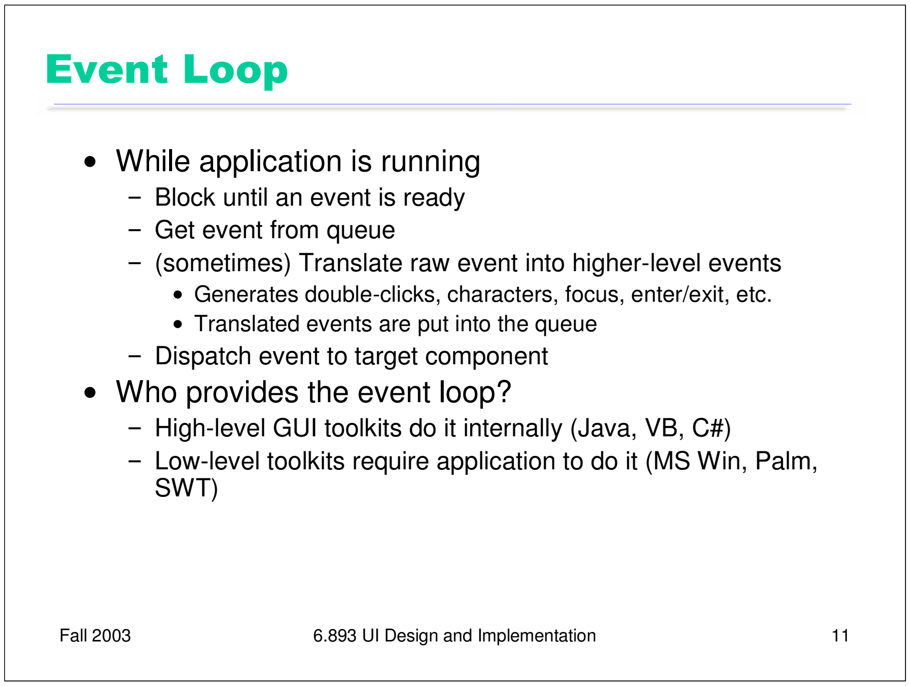
- Events are stored in a queue
  - User input tends to be bursty
  - Queue saves application from hard real time constraints (i.e., having to finish handling each event before next one might occur)
- Mouse moves are coalesced into a single event in queue
  - If application can't keep up, then sketched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

Fall 2003 6.893 UI Design and Implementation 10

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, VB, C#)
  - Low-level toolkits require application to do it (MS Win, Palm, SWT)

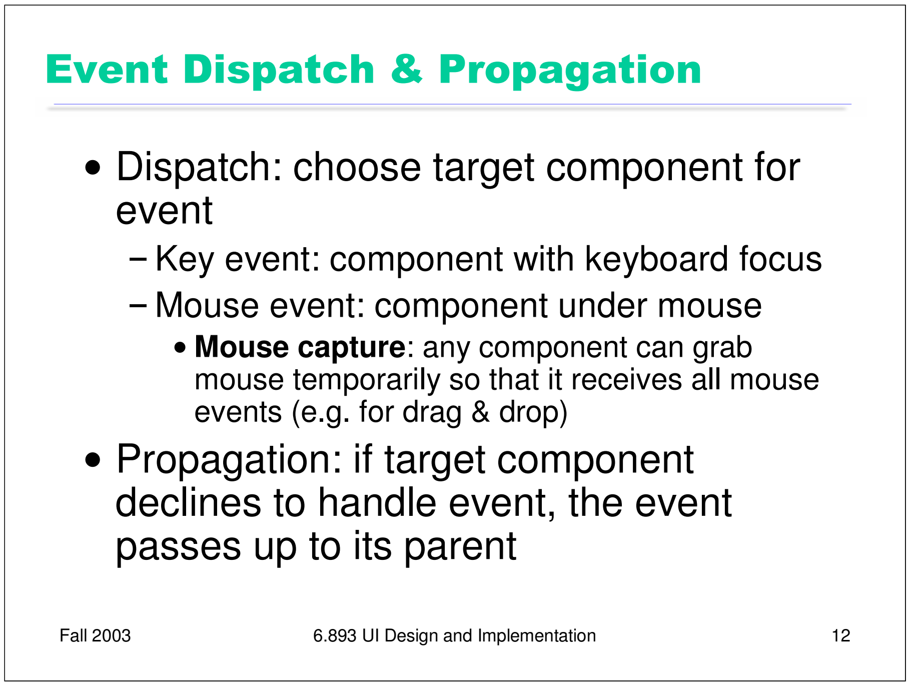
Fall 2003 6.893 UI Design and Implementation 11

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().



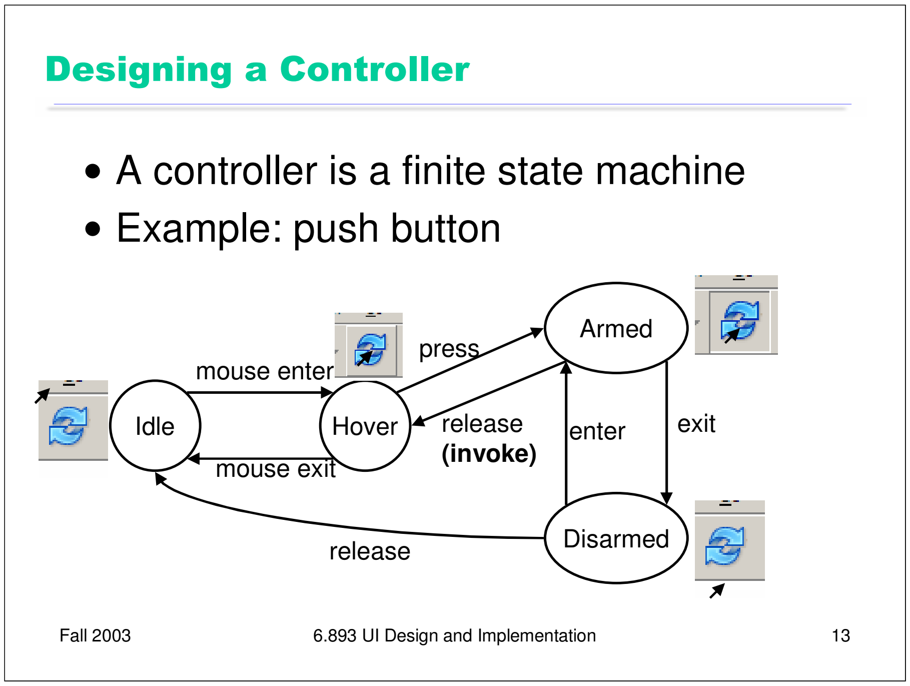
Event Dispatch & Propagation

- 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

Fall 2003 6.893 UI Design and Implementation 12

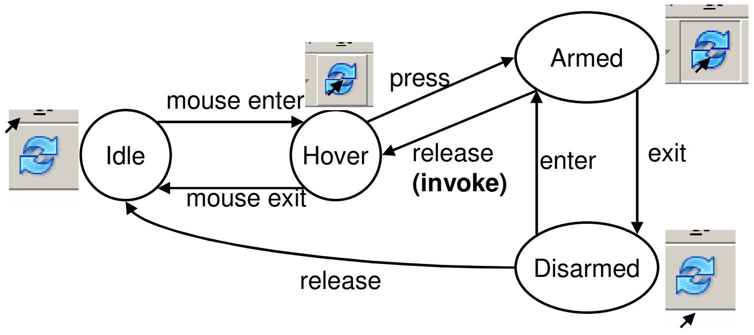
Event dispatch chooses a component to receive the event. Key events are sent to the component with the keyboard focus, and mouse events are generally sent to the component under the mouse. An exception is **mouse capture**, which allows any component to grab all mouse events (essentially a mouse analogue 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.



Designing a Controller

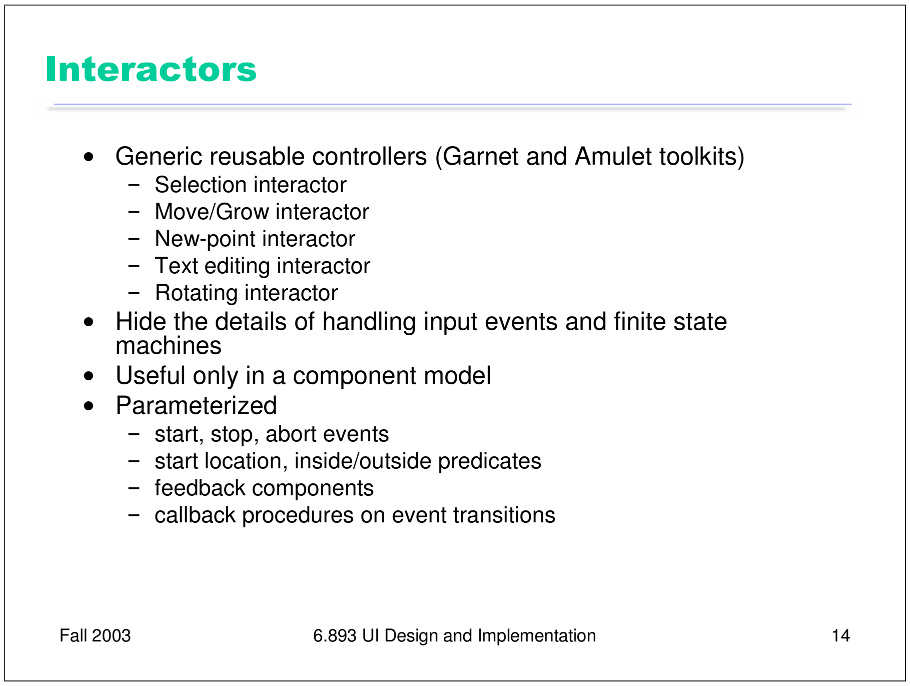
- A controller is a finite state machine
- Example: push button



Fall 2003 6.893 UI Design and Implementation 13

Now let’s look at how components that handle input are typically structured. A controller in a direct manipulation interface is a **finite state machine**. Here’s an example of the state machine for a push button’s controller. **Idle** is the normal state of the button when the user isn’t directing any input at it. The button enters the **Hover** state when the mouse enters it. It might display some feedback to reinforce that it affords clickability. If the mouse button is then pressed, the button enters the **Armed** state, to indicate that it’s being pushed down. The user can cancel the button press by moving the mouse away from it, which goes into the **Disarmed** state. Or the user can release the mouse button while still inside the component, which invokes the button’s action and returns to the **Hover** state.

Transitions between states occur when a certain input event arrives, or sometimes when a timer times out. Each state may need different feedback displayed by the view. Changes to the model or the view occur on transitions, not states: e.g., a push button is actually invoked by the release of the mouse button.



Interactors

- Generic reusable controllers (Garnet and Amulet toolkits)
  - Selection interactor
  - Move/Grow interactor
  - New-point interactor
  - Text editing interactor
  - Rotating interactor
- Hide the details of handling input events and finite state machines
- Useful only in a component model
- Parameterized
  - start, stop, abort events
  - start location, inside/outside predicates
  - feedback components
  - callback procedures on event transitions

Fall 2003 6.893 UI Design and Implementation 14

An alternative approach to handling low-level input events is the **interactor** model, introduced by the Garnet and Amulet research toolkits from CMU. Interactors are generic, reusable controllers, which encapsulate a finite state machine for a common task. They're mainly useful for the component model, in which the graphic output is represented by objects that the interactors can manipulate.