Lecture 8: Input

Hall of Fame or Shame?

Source: Daniel P.B. Smith, *Risks Digest*, v24 n44

Today's hall of fame or shame candidate is this DVD player. The arrow keys on a DVD player are supposed to move a cursor highlight around the screen. Unfortunately, this arrow diamond has been rotated by 45 degrees – so which arrow is which? Thanks to the rotation, this design destroys the normal **natural mapping** between the positions of the arrows and their functions. The up arrow isn't on top anymore.

Worse, the design also suffers from **internal inconsistency** – each button has two conflicting labels, one printed in white on the case, and the other embossed directly into the button. The white arrow is the correct label, and fortunately the white one wins by the "squint test" (which simulates a passing glance). But why have both labels? Clearly this is a case where a usability patch was slapped on after the fact, when it was too late to change the button molds.

The designers even missed an opportunity to recover at least a *little* of the direct mapping. They could have printed the up arrow *above* the up arrow button, and the down arrow *below* the down arrow button.

Today's Topics

- Input

Today's lecture continues our look into the mechanics of implementing user interfaces, by looking at **input** 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 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 interaction 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 (loss of focus is sometimes called "blur")
  - 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 something running in the background causes occasional long delays, 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 Swing, VB, C#, HTML)
  - Low-level toolkits require application to do it (MS Win, Palm, Java 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 Swing GUI program is System.exit(). This is true 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.

Javascript Event Models

- Events propagate in different directions on different browsers
  - Netscape 4: downwards from root to target
  - Internet Explorer: upwards from target to root
  - W3C standardized by combining them: first downwards ("capturing"), then upwards ("bubbling")
    - Firefox, Opera, Safari

The previous slide describes how virtually all desktop toolkits do event dispatch and propagation. Alas, the Web is not so simple.

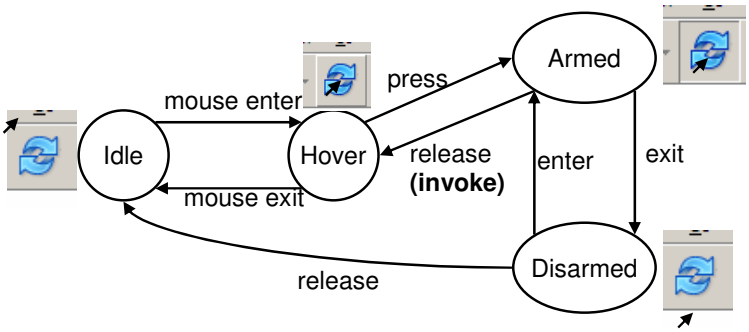
Early versions of Netscape propagated events *down* the view hierarchy, not up. (On the Web, the view hierarchy is a tree of HTML elements.) Netscape would first determine the target of the event (using mouse position or keyboard focus, as we explained earlier). But instead of sending the event directly to the target, it would first try sending it to the root of the tree, and so forth down the ancestor chain until it reached the target. Only if none of its ancestors wanted the event would the target actually receive it.

Alas, Internet Explorer's model was exactly the opposite – like the conventional desktop event propagation, IE propagated events upwards. If the target had no registered handler for the event (and no default behavior either, like a hyperlink does), then the event would propagate upwards through the tree.

The W3C consortium, in its effort to standardize the Web, combined the two models, so that events first propagate downwards to the target (a phase called "event capturing", not to be confused with mouse capture), and then back upwards again ("event bubbling"). You can register event handlers for both phases if you want. Modern standards-compliant browsers, like Firefox and Opera, support this model.

Designing a Controller

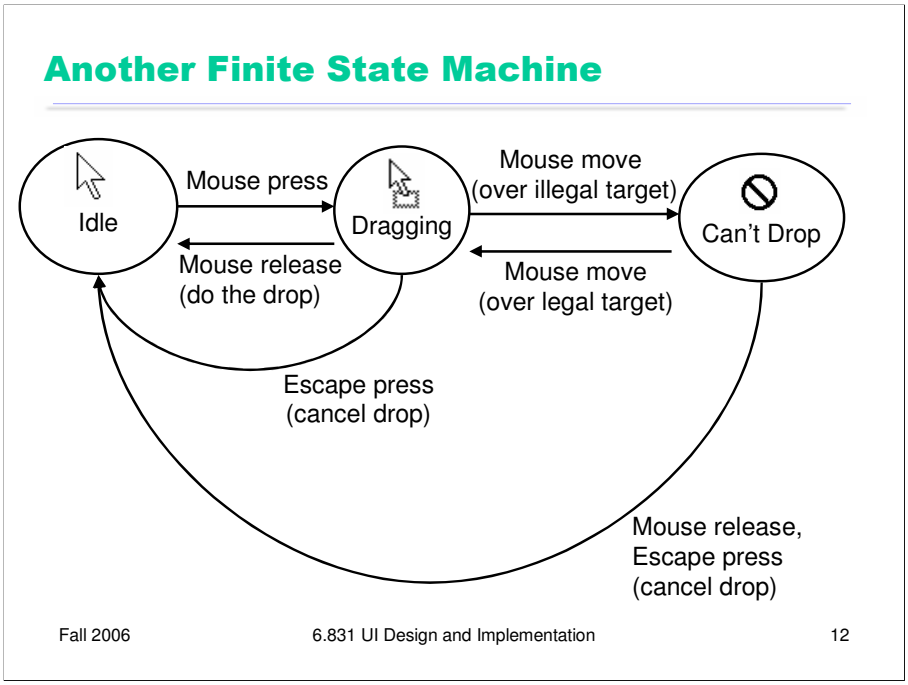
- A controller is a finite state machine
- Example: push button



Fall 2006 6.831 UI Design and Implementation 11

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.



Here's a finite state machine suitable for drag & drop.

Notice how each state of the machine produces different visual feedback, in this case the shape of the cursor. (The pushbutton on the last page had the same property.) This is a common case in input implementation, since different states of an input controller often represent different **modes** from the user's point of view, and distinguishing those modes with visual feedback helps reduce mode errors.

Visual feedback can also happen on the transitions, but it may have to be animated to be effective, because the transitions are very brief (like pressing or releasing a 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

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.