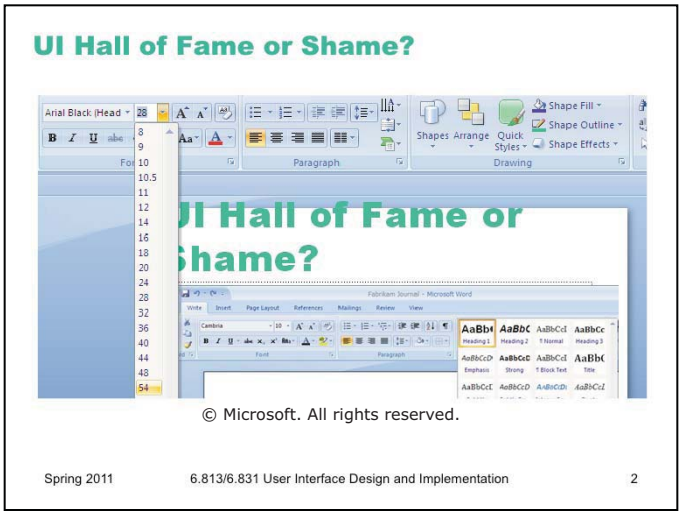
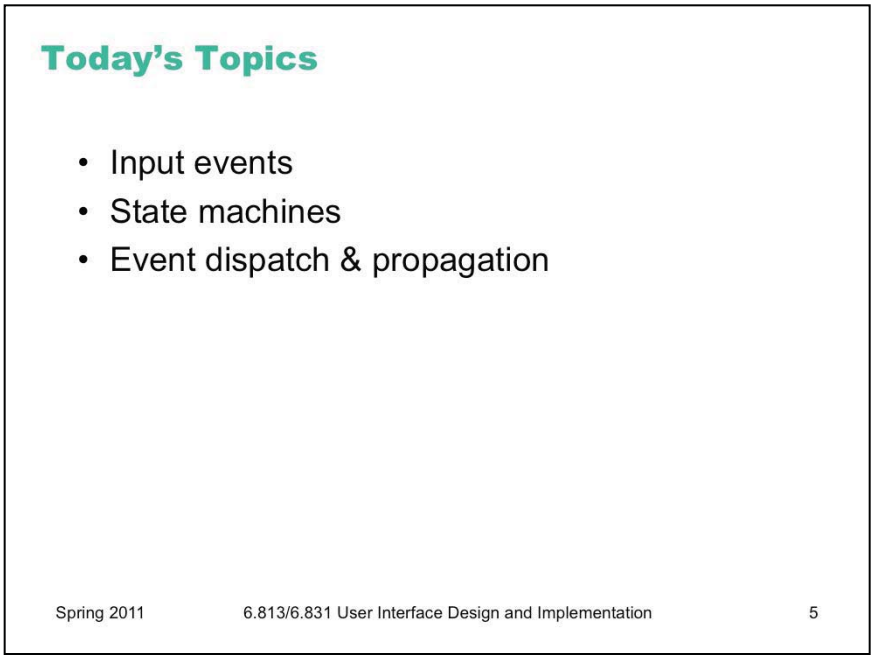
Lecture 12: Input

GR2 out, due Sunday



Today's Hall of Fame or Shame example is a feature of Microsoft Office 2007 that gives a preview of what a style command will do to the document while you're mousing over it. Here, the mouse is hovering over the 54-point choice on the font-size drop-down, and PowerPoint is showing what the selection would look like with that new style.

Let's discuss the pros and cons of this approach from a usability point of view.



Today's lecture finishes our look into the mechanics of implementing user interfaces, by examining **input** in more detail. We'll look mainly at keyboard and mouse input, but also multitouch interfaces like those on modern smartphones and tablets. This lecture has two key ideas for thinking about input. First, that **state machines** are a great way to think about and implement tricky input handling (like direct manipulation operations). Second, that events **propagate** through the view tree, and by understanding this process, you can make good design choices about where to attach the listeners that handle them.

Raw Input Events

- The usual input hardware has state:
  - ~100 keys on the keyboard (down or up)
  - (x,y) mouse cursor position on the screen
  - one, two, or three mouse buttons (down or up)
- A "raw" input event occurs when this state changes

	jQuery event
– key pressed or released	`keydown, keyup`
– mouse moved	`mousemove`
– button pressed or released	`mousedown, mouseup`

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

A raw event comes from a state transition in the input hardware. 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 game controls. And yet some toolkits like that did exist at one time, particularly in the bad old days of handheld and mobile phone programming.

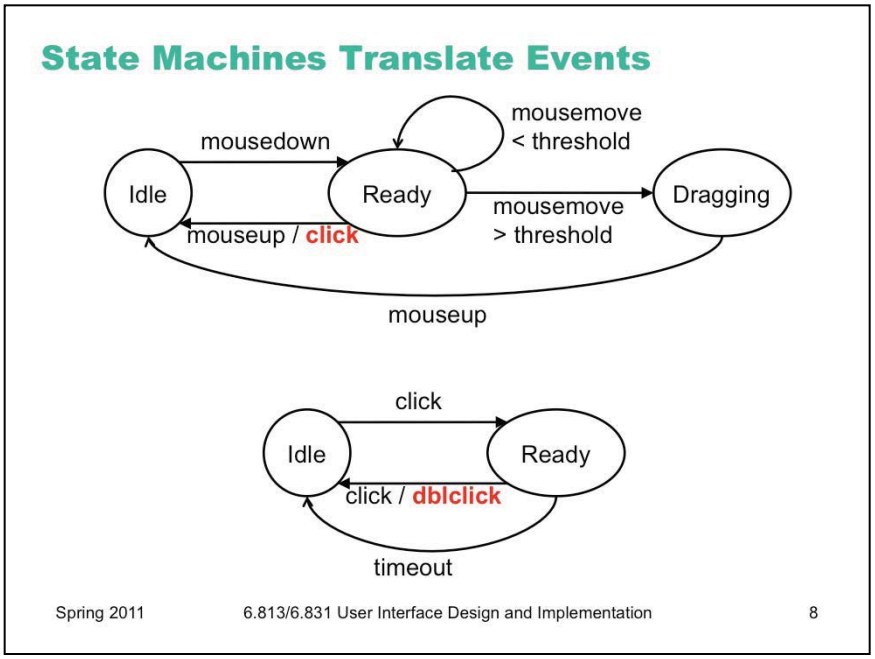
Translated Events

- Raw events are translated into higher-level events

	jQuery event
– Clicking	`click`
– Double-clicking	`dblclick`
– Character typed	`keypress`
– Entering or exiting an object's bounding box	`mouseenter`, `mouseleave`

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 interpreted as a drag rather than a click, so a click event isn't produced.

Key down and up events are translated into character-typed events, which take modifiers (Shift/Ctrl/Alt) and input methods (e.g. entering for Chinese characters on a standard keyboard) into account to produce a Unicode character rather than a physical keyboard key. In addition, if you hold a key down, multiple character-typed events may be generated by an autorepeat mechanism (usually built into the operating system or GUI toolkit). 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.



Here's our first example of using state machines for input handling. Inside the GUI toolkit, a state machine is handling the translation of raw events into higher-level events. Here's how the click event is generated – after a mousedown and mouseup, as long as the mouse hasn't moved (much) between those two events. Question for you: what is the threshold on your favorite GUI toolkit? If it's measured in pixels, how large is it? Does the mouse exiting the bounding box of the graphical object trigger the threshold regardless of pixel distance?

In this case, the raw events (down, up, move) are still delivered to your application, along with the translated event (click). This means that if your application is handling both the raw events and the translated events, it has to be prepared to expect this. This often comes up with double-click, for example: your application will see two click events before it sees the double-click event. As a result, you can't make click do something incompatible with double-click.

But occasionally, low-level events are **consumed** in the process of translating them to higher-level events. It's a difference you have to pay attention to in your particular toolkit.

Here's some HTML code to experiment with the translation of raw mouse events into clicks and doubleclicks:

```
<script src="http://code.jquery.com/jquery-1.5.min.js"></script>
<div id="A" style="width:100px;height:100px;background:lightYellow;padding:5px">
</div>
<script>
$(function() {
  $("#A").mousedown(function() { console.log("A down") })
  $("#A").mouseup(function() { console.log("A up") })
  $("#A").click(function() { console.log("A clicked") })
  $("#A").dblclick(function() { console.log("A double-clicked") })
})
</script>
```

Keyboard Focus

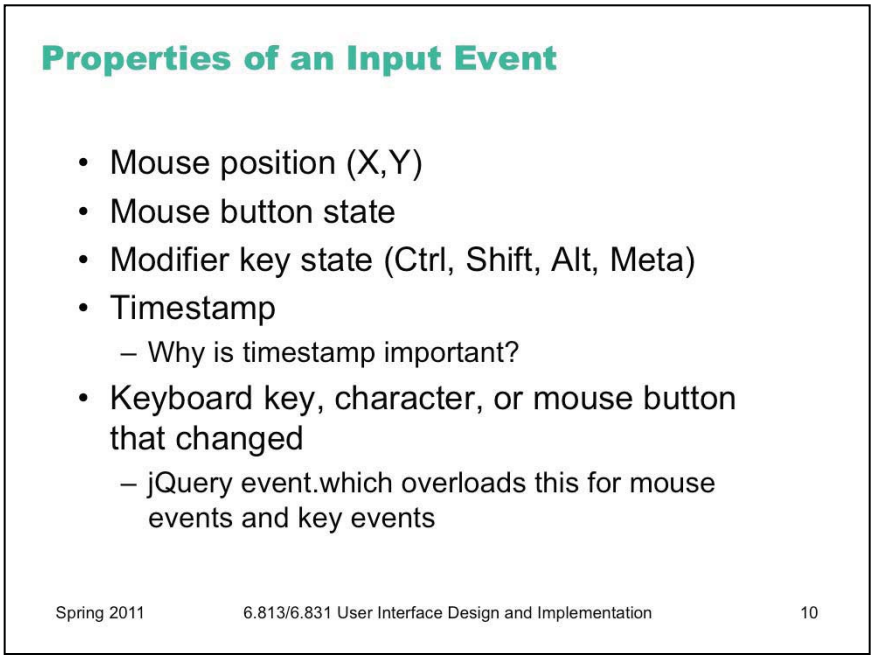
- An object in the view tree has the keyboard focus

	jQuery event
– Keyboard focus gained or lost	`focusin`, `focusout`

- Changing keyboard focus
  - by user input event: e.g. mouse down, Tab
  - programmatically by a method call
- Not all HTML elements can have keyboard focus by default

   to force ability to take focus

The keyboard focus is also part of the state of the input system, but it isn't in the input hardware – instead, the keyboard focus is a particular object in the view tree that currently receives keyboard events. On some X Windows window managers, you can configure the keyboard focus to follow the mouse pointer – whatever view object contains the mouse pointer has the keyboard focus as well. On most windowing systems (like Windows and Mac), however, a mouse down is the more common way to change the focus.



Input events carry with them some or all of these properties, which represent the state of the input hardware immediately after the event occurred.

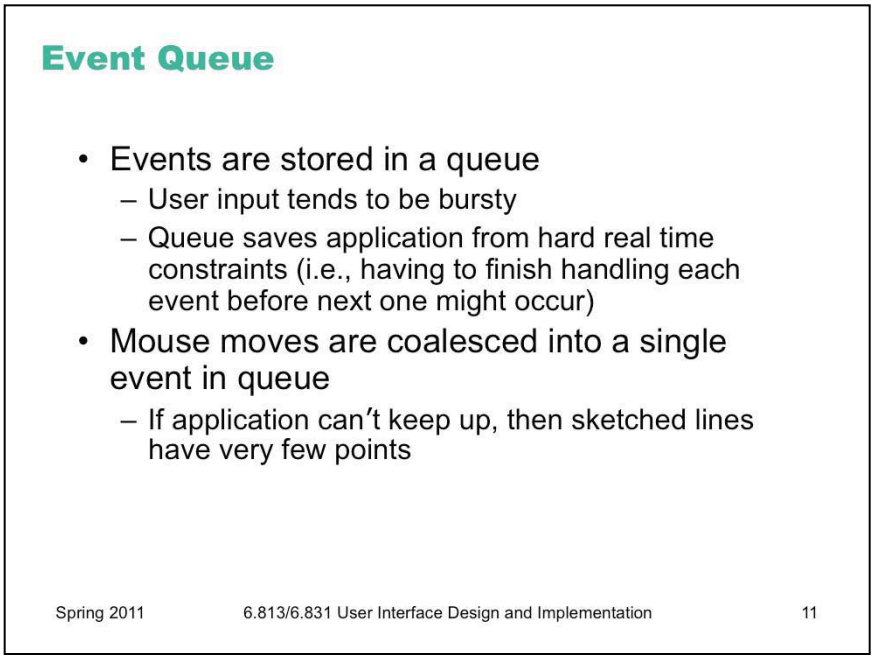
On most systems, all events include the modifier key state, since some mouse gestures are modified by Shift, Control, and Alt. Some systems include the mouse position and button state on all events; some put it only on mouse-related events.

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, so it's necessary for the time of the event to be captured as close to the event's actual occurrence (the press or release in the event object itself).

Keyboard events can be trickier to handle than mouse events because identifying the key involved in the event is not always easy. Particularly for cross-platform toolkits (HTML, Flash, Java), there may be a variety of different keyboard hardware with different sets of keys, and in HTML/Javascript, different browsers may work differently. There is the further complication that translated key events (the "character typed" event) do not represent a **keystroke** (like Shift or PgUp or the A key), but rather a **character** (like "a" or "A" or "%"). Keystrokes are identified by physical keys on the keyboard; characters are identified by values in a character set (like Unicode or ASCII). In jQuery, do not treat keydown/keyup and keypress as interchangeable; their names may be similar, but the parameters of the events are different.

Here's some code for experimenting with key event parameters:

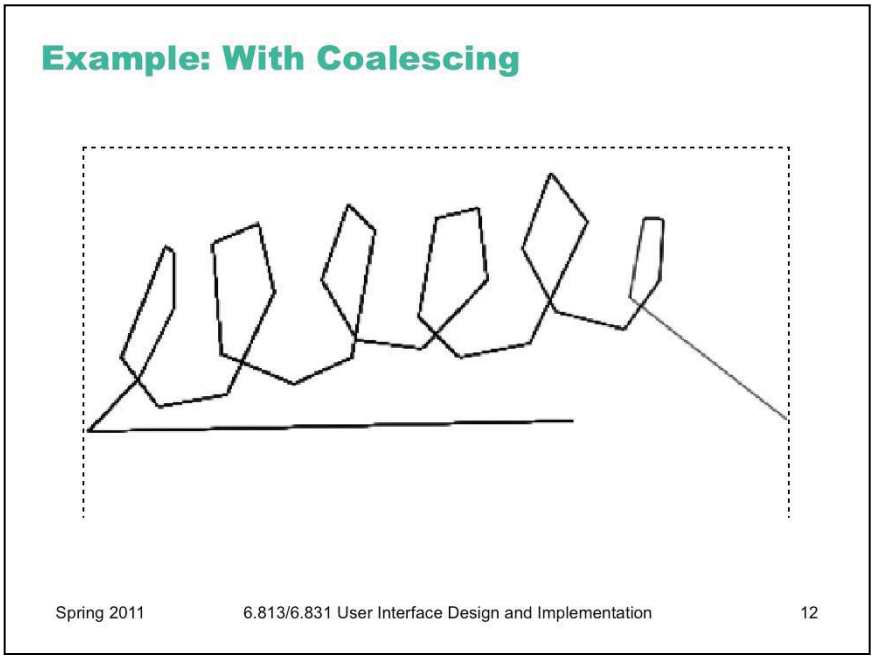
```
<script src="http://code.jquery.com/jquery-1.5.min.js"></script>
<textarea id="A"></textarea>
<script>
$(function() {
$("#A").keydown(function(event) { console.log("A down " + event.which) })
  $("#A").keyup(function(event) { console.log("A up " + event.which) })
  $("#A").keypress(function(event) { console.log("A press " + event.which) })
})
</script>
```



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.

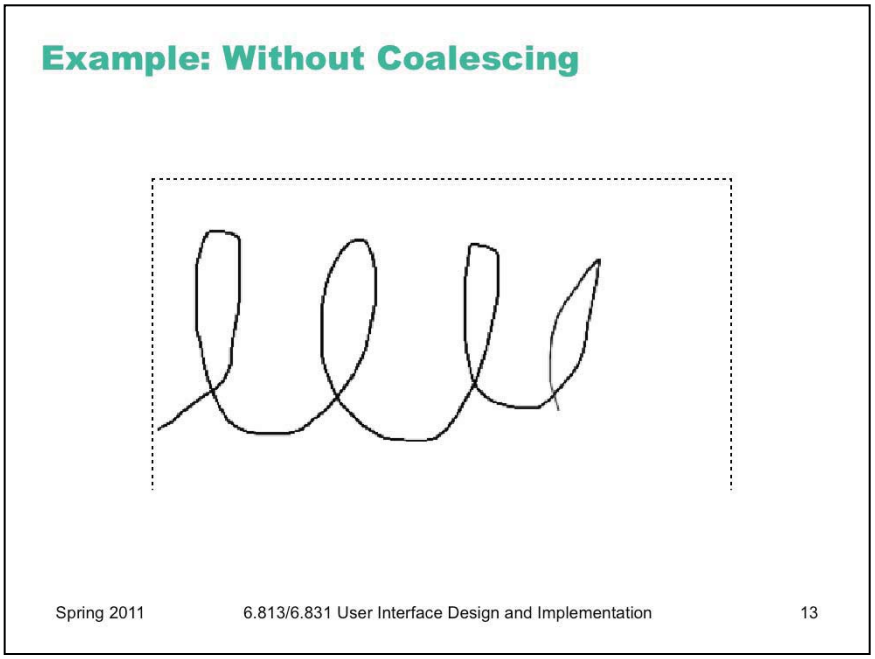
Edge events (button down and up events) are all kept in the queue unchanged. But multiple events that describe a continuing state – in particular, mouse movements – may be **coalesced** into a single event with the latest known state. Most of the time, this is the right thing to do. For example, if you're 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.



Here's some code for experimenting with mouse coalescing in HTML. Try it in http://htmledit.squarefree.com. First wave your mouse over the canvas; you'll see a curve following your mouse points. Now increase the sleep(0) to sleep(100), simulating 100 milliseconds of extra load on every mousemove event (which might be the cost of your application handling the event, or maybe from some other application that's hogging the CPU). What do you see?

```
<script src="http://code.jquery.com/jquery-1.5.min.js"></script>
<canvas id="A" width="400" height="300" style="border: 1px dashed"></canvas>
<script>
$(function() {
  var ctx = $("#A").get(0).getContext("2d")
  var pos = $("#A").offset()
  ctx.translate(-pos.left, -pos.top) // move origin of canvas to origin of document, so we can use event.pageX/pageY directly
  $("#A").mousemove(function(event) { sleep(0); ctx.lineTo(event.pageX, event.pageY); ctx.stroke() })
  function sleep(ms) { var wake = now() + ms; while (now() < wake) { } }
  function now() { return new Date().getTime() }
})
</script>
```



Event coalescing can't be easily disabled in HTML/Javascript, so we'll simulate what life would be like without coalescing by creating our own queue of mouse points (queue and pushEvent() in the code below) and an event handler that reads from it (eventLoop()). Try the code as given first, and then try increasing DELAY to 50 milliseconds. What happens?

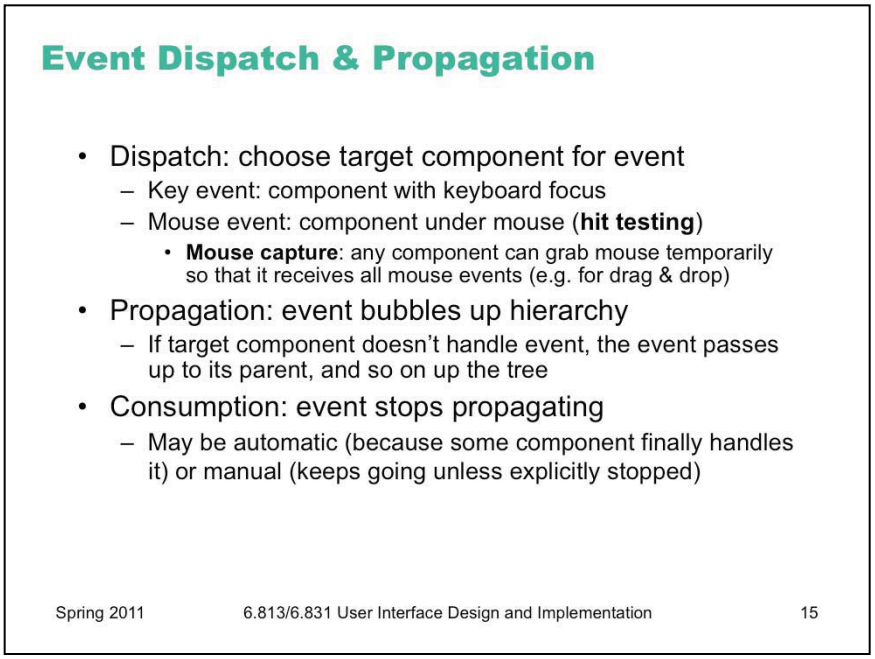
```
<script src="http://code.jquery.com/jquery-1.5.min.js"></script>
<canvas id="A" width="400" height="300" style="border: 1px dashed"></canvas>
<script>
$(function() {
  var ctx = $("#A").get(0).getContext("2d")
  var pos = $("#A").offset()
  ctx.translate(-pos.left, -pos.top) // move origin of canvas to origin of document, so we can use event.pageX/
pageY directly
  $("#A").mousemove(function(event) { pushEvent(event.pageX, event.pageY) })

  var queue = [];
  var DELAY = 1;
  function pushEvent(x,y) {
    if (!queue.length) setTimeout(eventLoop, DELAY)
    queue.push({x:x, y:y})
  }
  function eventLoop() {
    if (!queue.length) return;
    var pt = queue.shift();
    ctx.lineTo(pt.x, pt.y);
    ctx.stroke();
    setTimeout(eventLoop, DELAY)
  }
})
</script>
```

Event Loop

- While application is running
  - Block until an event is ready
  - Get event from queue
  - 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).



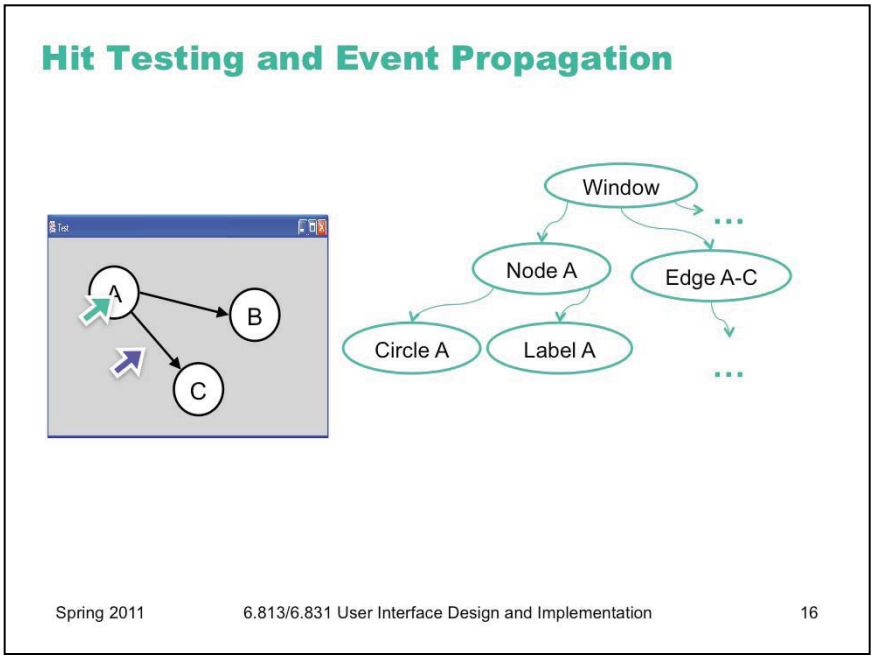
Event dispatch chooses a component to receive the event. Key events are sent to the component with the keyboard focus, and mouse events are generally sent to the component under the mouse, using hit testing to determine the visible component that contains the mouse position and is topmost (in z order).

An exception is **mouse capture**, which allows any component to grab all mouse events after a mouse button was pressed over that component, for as long as the button is held down. This is 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 the ability to turn it on or off – in the Windows API, for example, you'll find a SetCapture function.

If the target component has no handler for the event, the event propagates up the view hierarchy looking for some component able to handle it. If an event bubbles up to the top without being handled, it is discarded.

In many GUI toolkits, the event stops propagating automatically after reaching a component that handles it; none of that component's ancestors see the event. Java Swing behaves this way; an event propagates up through the tree until it finds a component with at least one listener registered for the event, and then propagation stops automatically. (Note that this doesn't necessarily mean that only one *listener* sees the event. The component that finally handles the event may have more than one listener attached to it, and all of those listeners will receive the event, in some arbitrary order. But no listeners attached to components higher in the tree will see it.)

In some toolkits, however, event propagation is under the control of the programmer, and events continue propagating up the tree unless explicitly stopped. HTML/Javascript behaves this way, as does Adobe Flex. In these toolkits, an event can be stopped by calling stopPropagation() on its event object.



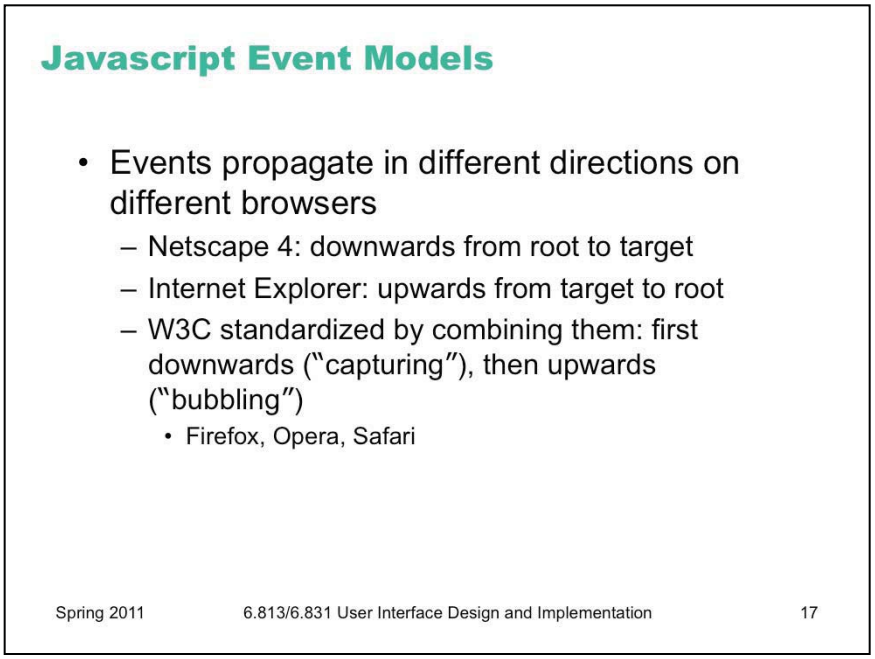
Here are some examples of how mouse events are dispatched and propagated. The window on the left has the view hierarchy shown on the right, in which each graph node is represented by a Node component with two children, a Circle (displaying a filled white circle with a black outline) and a text Label (displaying a text string, such as “A” or “B”).

First consider the green mouse cursor; suppose it just arrived at this point. Then a mouse-move event is created and dispatched to the topmost component whose bounding box contains that point, which is Label A. If Label A doesn’t handle the mouse-move event, then the event is propagated up to Node A; if that doesn’t handle the event either, it’s propagated to Window, and then discarded. Notice that Circle A never sees the event, because event propagation goes up the tree, not down through z-order layers.

Now consider the blue mouse cursor. What component will be the initial target for a mouse-move event for this point? The answer depends on how hit-testing is done by the toolkit. Some toolkits support only rectangular bounding-box hit testing, in which case Edge A-C (whose bounding box contains the mouse point) will be the event target. Other toolkits allow hit testing to be overridden and controlled by components themselves, so that Edge A-C could test whether the point actually falls on (or within some small threshold of) the actual line it draws. Java Swing supports this by overriding Component.contains(). If Edge A-C rejects the point, then the next component in z-order whose bounding box contains the mouse position is the window itself, so the event would be dispatched directly to the window.

Here’s some HTML code for experimenting with event dispatch and propagation. How does HTML/Javascript do mouse capture?

```
<script src="http://code.jquery.com/jquery-1.5.min.js"></script>
<div id="A" style="width:100px;height:100px;background:lightYellow;padding:5px">A
  <div id="B" style="margin:20px; padding:5px; background:lightPink;">B</div>
</div>
<script>
$(function() {
$("#B").bind("mousedown mouseup mousemove", function(event) { console.log(this.id +" got " +
event.target.id + " " + event.type) })
//$("#A").bind("mousedown mouseup mousemove", function(event) { console.log(this.id +" got " +
event.target.id + " " + event.type) })
//$(window).bind("mousedown mouseup mousemove", function(event) { console.log("window got " +
event.target.id + " " + event.type) })
})
</script>
```



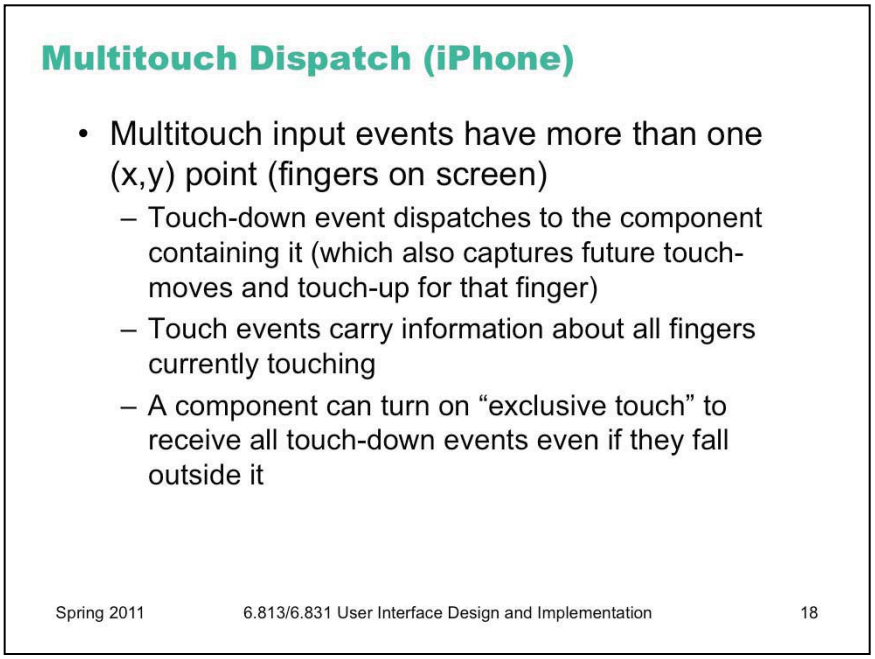
The previous slides describe how virtually all desktop toolkits do event dispatch and propagation. Alas, the Web is not so simple.

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 button or hyperlink has for click events), 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 either or both phases if you want. Modern standards-compliant browsers, like Firefox and Opera, support this model; so does Adobe Flex.

One advantage of this two-phase event propagation model is that it gives you a lot more flexibility as a programmer to override the behavior of other components. By attaching a capturing listener high up in the component hierarchy, you can handle the events yourself and prevent other components from even seeing them. For example, if you want to implement an "edit mode" for your UI, in which the user can click and drag around standard widgets like buttons and textboxes, you can do that easily with a single capturing listener attached to the top of your UI tree. In the traditional desktop event propagation model, it would be harder to prevent the buttons and textboxes from trying to interpret the click and drag events themselves, and you would have to add listeners to every single widget.



Multitouch interfaces like the Apple iPhone introduce a few wrinkles into the event dispatch story. Instead of having a single mouse position where the event occurs, a multitouch interface may have multiple points (fingers) touching the screen at once. Which of these points is used to decide which component gets the event?

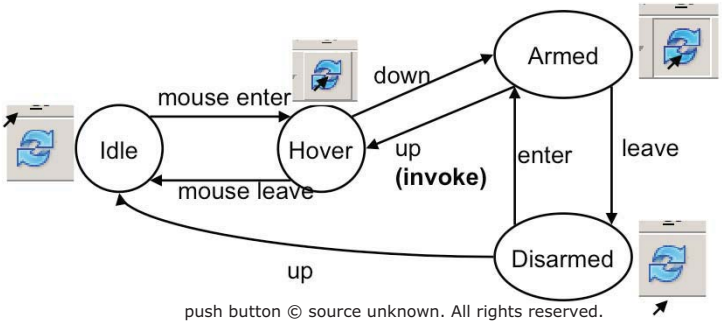
Here's how the iPhone does it. Each time a finger touches down on the screen, the location of the new touch-down is used to dispatch the touch-down event. All events carry along information about all the fingers that are currently touching the screen, so that the component can recognize multitouch gestures like pinching fingers together or rotating the fingers. (This is a straightforward extension of keyboard and mouse events, in fact – most input events carry along information about what keyboard modifiers are currently being held down, and often the current mouse position and mouse button state as well.)

Two kinds of event capture are used in the iPhone. First, after a touch-down event is dispatched to the component that it touched first, that component automatically captures the events about all future moves of that finger, even if it strays outside the bounds of the component, until the finger finally leaves the screen (touch-up). This is similar to the automatic mouse capture used by Java Swing when the mouse is dragged.

Second, a component can also turn on its "exclusive touch" property, which means that if the first touch on the screen (after a period of no fingers touching) is dispatched to that component, then all future touch events are captured by that component, until all fingers are released again. (Apple, "Event Handling", *iPhone Application Programming Guide*, 2007).

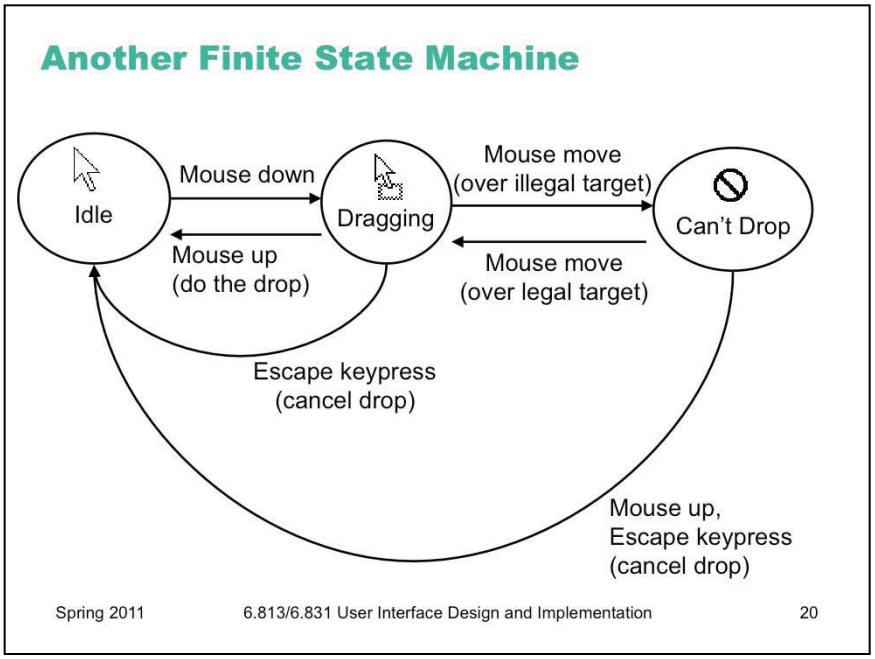
Designing a Controller

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



Now let's look at how components that handle input are typically structured. A controller in a direct manipulation interface is a **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 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).

Summary

- Input events are raw and translated
- Events are **dispatched** to a target view and **propagated** up (or down then up) the view hierarchy
- State machines are a useful pattern for thinking about input

6.831 / 6.813 User Interface Design and Implementation

Spring 2011