Programming Languages and Techniques (CIS120e)

Lecture 18
Oct 22, 2010

GUI Framework Design III

Events and Event Handling

- An event is a signal
  - e.g. a mouse click, mouse motion, or keypress

- Events carry data
  - e.g. the coordinates of the mouse, state of the button, which key is pressed

- An event can be handled by some widget
  - The top-level loop waits for an event and then gives it to the root widget.
  - The widgets forward the event down the tree until some widget handles the event (or no suitable widget is found, in which case the event is just dropped).

- Typically, the widget that handles an event updates some state of the GUI in response.
  - e.g. to record where the mouse was clicked, or to animate a button

Reactive Widgets

```plaintext
type status = {
  mouse_x : int; (* X coordinate of the mouse *)
  mouse_y : int; (* Y coordinate of the mouse *)
  button : bool; (* true if a mouse button is pressed *)
  keypressed : bool; (* true if a key has been pressed *)
  key : char; (* the character for the key pressed *)
}

type event = Graphics.status

type t = {
  repaint : Gctx.t -> unit;
  size : Gctx.t -> int * int;
  handle : event -> unit (* NEW: event handler *)
}
```
Revised Top-level Loop

```ocaml
let run (w:Widget.t) : unit =
  open_graph "";
let g = Gctx.create() in
while true do
  clear_graph ();
  w.Widget.repaint g;
  (* wait for a user input event and process it *)
  let e = wait_next_event [Mouse_motion; Button_down; Button_up; Key_pressed] in
  w.Widget.handle (invert_y_coordinate e)
done
```

- Use the Graphics library function `wait_next_event` to wait for something to happen and then ask the root widget to handle it.
  - Note: we have to translate the event’s OCaml coordinates into standard coordinates
- In the real code, there are a few more calls to the Graphics library to help eliminate flicker… see the documentation about “double buffering” if you’re curious.

Event-handling Pictorially

Widget tree

User clicks, generating event `e`

On the screen

Event-handling Pictorially

Routing events through `hpair` widgets

- The event handler of an `hpair` must check to see whether the event should be handled by the left or right widget.
  - Check the event’s coordinates against the size of the left widget
  - If the event is within the left widget, let it handle the event
  - Otherwise check the event’s coordinates against the right child’s
  - If the right child gets the event, don’t forget to translate its coordinates.

- Unfortunately, calculating the size of a widget requires a `Gctx.t`, which is not available in the handle method.
  - Solution: remember the sizes of the two children in reference cells and update them each time the `repaint` or size methods are called.

- Another design possibility: modify the type of handlers
  - `handle: Gctx.t -> event -> unit`
  - Possibly a good way to go; leads to different tradeoffs/complexities

Event Handling: Routing

- When a container widget handles an event, it passes the event to the appropriate child.
  - Event coordinates must be translated so that they are relative to the child’s local coordinates.

```ocaml
let translate_event (e:event) ((x,y):int*int) : event =
  {
    mouse_x = e.mouse_x - x; (* subtract -- why? *)
    mouse_y = e.mouse_y - y;
    button = e.button;
    keypressed = e.keypressed;
    key = e.key
  }

let border (w:t):t =
  {
    repaint = ...;
    handle = (fun (e:event) ->
      w.handle (translate_event e (2,2)));}
```
Stateful Widgets

What state do the event handlers modify?

Event Listeners

- Need to be able to add additional event processing.
  - Create a widget
  - Create some state associated with the widget
  - Add some event handling to process events associated with that widget, modifying the associated state

- An event listener “eavesdrops” on the events flowing through the widget hierarchy
  - It can react to the events it listens for (i.e. by changing some state)
  - It can selectively prevent an event from continuing on its route
  - (If the event is allowed to continue, subsequent event listeners might “hear” the event too.)

- A notifier is a container widget that adds event listeners to a path in the hierarchy.

A stateful label Widget

```
(* The type of listeners *)
type listener = event -> bool

(* Create a listener for mouse click events *)
let mouseclick_listener (action:unit -> unit) : listener =
  fun (e: event) ->
    if e.button then (action(); true)
    else false
```

- A listener returns true if the event should not be passed on, and false otherwise.
- A mouseclick_listener performs an action and stops the event when it “hears” a mouse click, and passes on the event to later listeners otherwise.
Notifiers and Notifier Controllers

```
let notifier (w: t) : t * notifier_controller =
  let listerners = ref [] in
  { repaint = w.repaint;
    size = w.size;
    handle = (fun (e: event) ->
      let rec loop (l: listener list) =
        begin match l with
        | [] -> w.handle e
        | h::t -> if h e then () else loop t
        end in
      loop !listerners);
    },
  { add_listener = fun newl -> listeners := newl::!listerners }
```

The controller allows new listeners to be added to the list.

Listeners and Notifiers Pictorially

---

Buttons (at last!)

```
let button (s: string) : t * label_controller *
  notifier_controller =
  let (w, lc) = label s in
  let (w', nc) = notifier w in
  (w', lc, nc)
```

- A text button widget is just a label wrapped in a notifier.
- One would typically add a mouseclick_listener to the button using the notifier_controller.

Demo: wdemo

A simple light-switch app.