Programming Languages and Techniques (CIS120e)

Lecture 17

Oct 20, 2010

GUI Framework Design II
A graphics context Gctx.t represents a position within the window, relative to which the widget-local coordinates should be interpreted. We can add additional context information that should be “inherited” by children widgets (e.g. current pen color).
(* An interface for non-interactive widgets *)

```ocaml
type t = {
  repaint : Gctx.t -> unit;
  size    : Gctx.t -> int * int
}
```

- SimpleWidget.t is the type of non-interactive widgets
- A simple widget knows how to repaint itself relative to a graphics context.
- It can report its size (also relative to a graphics context)

- Examples: label and space widgets
  - Recall code in simpleWidget.ml
The canvas Widget (Leaf)

(* expose the graphics context as a widget *)

let canvas ((w,h):int*int)(repaint: Gctx.t -> unit) : t =
{
  repaint = repaint;
  size = (fun (_:Gctx.t) -> (w,h))
}

• A canvas widget has a fixed width and height
• It is parameterized by a repaint method that has access to the graphics context.
  – Use the primitive Gctx drawing routines to draw on the canvas
let b = border w

- Draws a one-pixel wide border around the contained widget, w.
- b’s size is slightly than bigger than w’s (+4 pixels in each dimension).
- When b asks w to repaint, b must translate the Gctx.t to (2,2) to account for the displacement of w from b’s origin.
The hpair Widget (Container)

• \textbf{let} \ h = \textbf{hpair} \ w1 \ w2
• Creates a horizontally adjacent pair of widgets.
• Aligns them by their top edges.
  – Its size is the sum of their widths and max of their heights
• Must translate the Gctx when repainting the right widget.
The Widget Hierarchy

• Widgets form a tree*:
  – Leaf widgets – don’t contain any children
    • label, space, and canvas widgets are leaves
  – Container widgets – are “wrappers” for their children
    • border and hpair widgets are containers

• Build container widgets by passing in their children as arguments to their “constructor” function.
  – e.g. \texttt{let \ b = border \ w \ in ...}
    \texttt{let \ h = hpair \ b1 \ b2 \ in ...}

• The repaint method of the root widget initiates all the drawing and layout for the whole window.

*If you draw the state of the abstract machine for a widget program, the tree will be visible in the heap – the closure of the “repaint” method for a container widget will contain references to its children.
(* Create some simple label widgets *)
let l1 = label "Hello"
let l2 = label "World"

(* Compose them horizontally, adding some borders *)
let h = border (hpair (border l1)
    (hpair (space (10,10)) (border l2)))
Demo: swdemo.ml
Design Challenge #2: User Interactions

- When a user moves the mouse, clicks, the button, or presses a key, the GUI needs to be able to react.
- We need to replace the infinite loop in the main `run` function with something more useful:

```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 *)
    ???
  done
```
Events and Event Handling

widget.ml
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

Graphics library

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 *)
}

widget.mli

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

eventloop.ml

```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

User clicks, generating event e

Hello
World

Widget tree

On the screen
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.

```plaintext

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 = ...;
    size = ...;
    handle = (fun (e:event) ->
      w.handle (translate_event e (2,2)));
  }
```

widget.ml
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
Stateful Widgets

What state do the event handlers modify?
A stateful label Widget

- A stateful widget comes with a controller object that lets you modify the state.
  - e.g. the `label_controller` object provides a way to set the label
- Each kind of stateful widget gets its own kind of controller
  - As we’ll see, Java’s subtyping helps manage this complexity
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.
(* 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

```
module widget

let notifier (w: t) : t * notifier_controller = {
  let listeners = ref [] in
  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 !listeners;
}

{ add_listener = fun newl -> listeners := newl::!listeners }
```

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

Loop through the list of listeners, allowing each one to process the event. If they all pass on the event, send it to the child.
Listeners and Notifiers Pictorially

User clicks, generating event e

Widget tree

On the screen
Buttons (at last!)

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.