Programming Languages and Techniques (CIS120)

Lecture 19

February 25, 2013

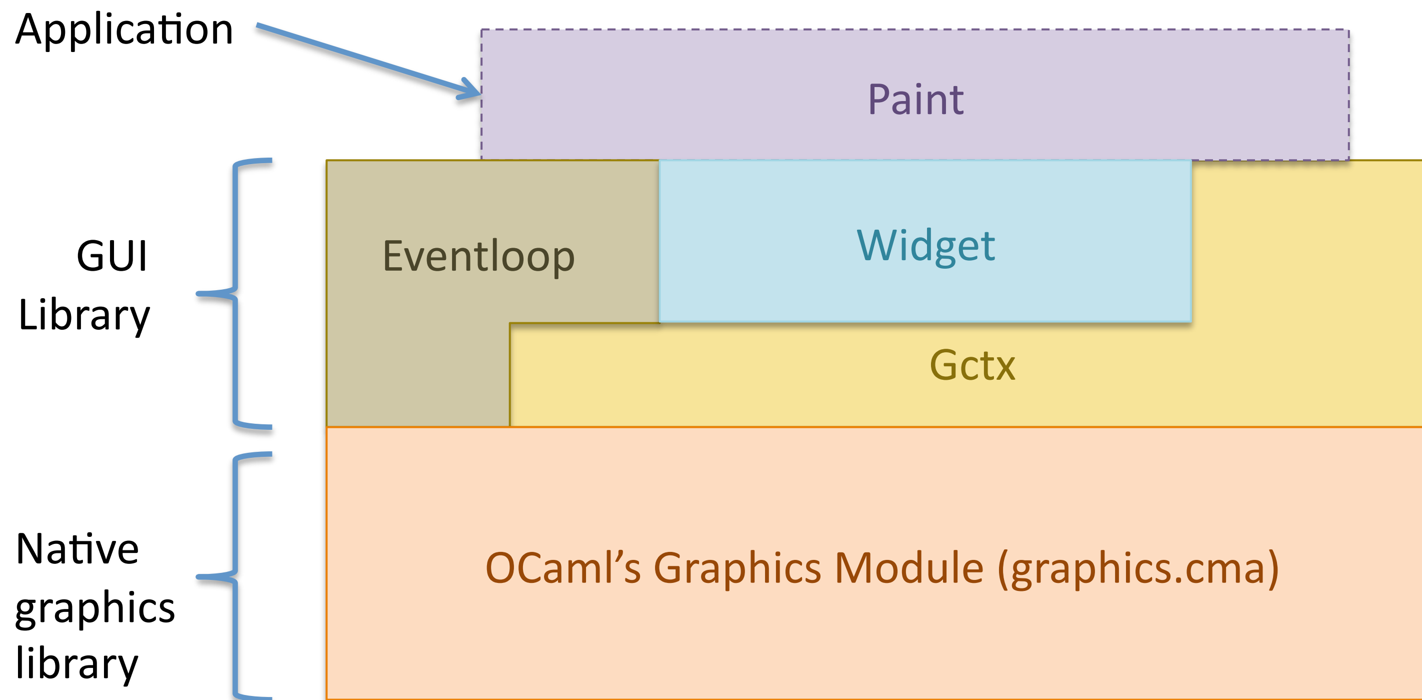
GUI Design III: Events

Announcements

HW06: GUI programming

- Due: *Friday, March 1st*
- *Note TAs will be unavailable during late period due to Spring Break*
- *Graded manually*
  - *Submission only checks for compilation, no auto tests*
  - *Won't get scores immediately*
  - *Only LAST submission will be graded*
- Weirich OH today (3:30-5PM)

Project Architecture



The Widget Hierarchy

- Widget instances 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" functions
  - e.g.

```
let b = border w in …
let h = hpair b1 b2 in …
```

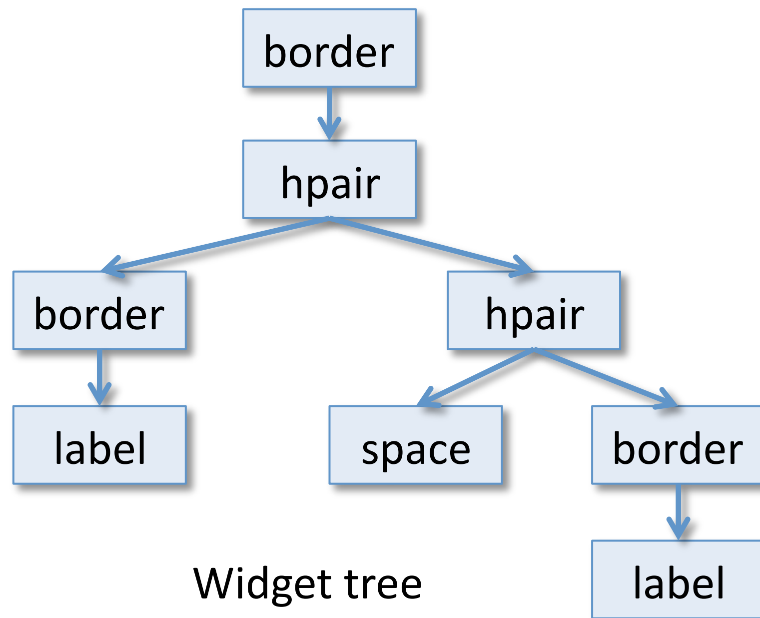
- The repaint method of the root widget initiates all the drawing and layout for the whole window

*If you draw the state of the abstract machine for a widget program, the tree will be visible in the heap – the saved stack of the "repaint" function for a container widget will contain references to its children.

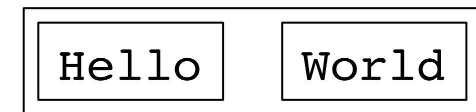
Widget Hierarchy Pictorially

swdemo.ml

```
(* 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)))
```



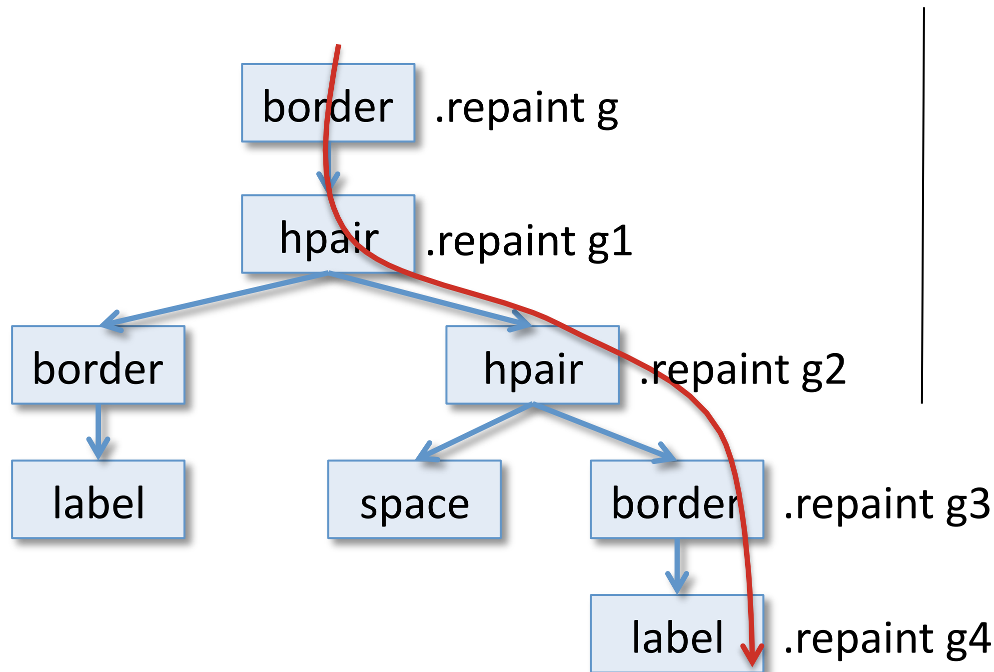
Widget tree

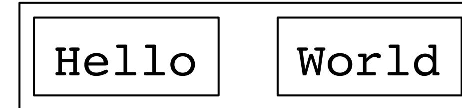


On the screen

Drawing: Containers

Container widgets propagate repaint commands to their children:





Widget tree

g1 = Gctx.translate g (2,2)
g2 = Gctx.translate g1 (hello_width,0)
g3 = Gctx.translate g2 (space_width,0)
g4 = Gctx.translate g3 (2,2)

On the screen

Events and Event Handling

User Interactions

- Problem: When a user moves the mouse, clicks the button, or presses a key, the application should react. How?

swdemo.ml

```
let w = …   (* top-level widget *)

let run () :unit =
    (* open the window *)
    Graphics.open_graph "";
    let g = Gctx.top_level in
    (* draw the widget *)
    w.repaint g;
    (* infinite loop so we can see the window. *)
    let rec loop () : unit = loop () in
    loop ()
```

Solution: The Event Loop

eventloop.ml

```
let run (w:Widget.t) : unit =
  Graphics.open_graph "";
  Graphics.auto_synchronize false;
  let g = Gctx.top_level in

  let rec loop () =
    Graphics.clear_graph ();
    w.repaint g;
    Graphics.synchronize ();
    let e = Gctx.wait_for_event g in    (* wait for user input *)
      w.handle g e;                     (* react to event *)
    loop ()
  in
    loop ()
```

- The run function takes in the root widget "w", creates the graphics window, and then enters an infinite loop.
- The loop clears the window, repaints it, waits for a user event, and then asks the root widget to *handle* that event.

Reactive Widgets

OCaml 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 *)
}
```

gcxt.mli

```
type event  (* Graphics.status internally *)

val wait_for_event : t -> event
val event_pos : t -> event -> position
val button_pressed : t -> event -> bool
val is_keypressed : t -> event -> bool
val get_key : t -> event -> char
```

The graphics context translates the location of the event to widget-local coordinates

Reactive Widgets

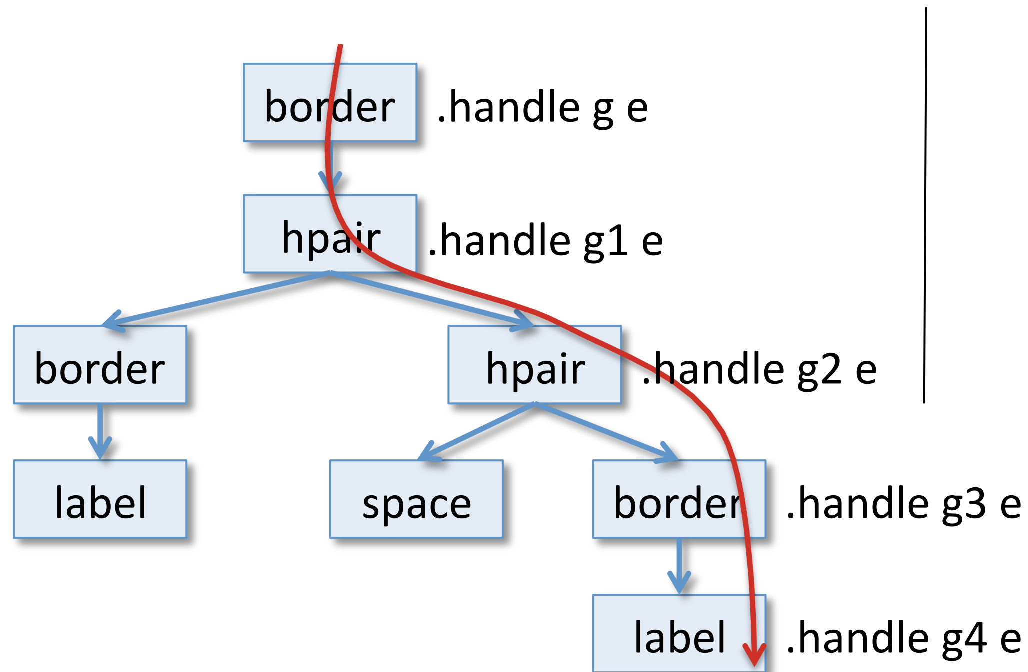
widget.mli

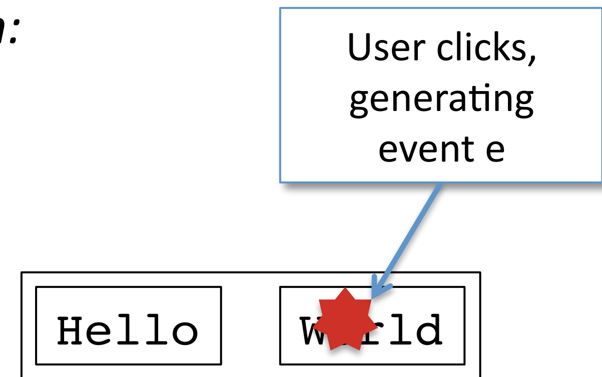
```
type t = {
  repaint : Gctx.t -> unit;
  size    : Gctx.t -> Gctx.dimension;
  handle  : Gctx.t -> Gctx.event -> unit     (* NEW! *)
}
```

- Widgets have a "method" for handling events
  - The eventloop 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 ignored

Event-handling: Containers

Container widgets propagate events to their children:





Widget tree

On the screen

Event Handling: Routing

- When a container widget handles an event, it passes the event to the appropriate child
- The Gctx.t must be translated so the child can interpret the event in its own local coordinates.

widget.ml

```
let border (w:t):t =
  { repaint = …;
    size = …;
    handle = (fun (g:Gctx.t) (e:Gctx.event) ->
       w.handle (Gctx.translate g (2,2)) e);
  }
```

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

```
handle =
(fun (g:Gctx.t) (e:Gctx.event) ->
   if event_within g e (w1.size g)
   then w1.handle g e
   else
   let g = (Gctx.translate g (fst (w1.size g), 0)) in
     if event_within g e (w2.size g)
     then w2.handle g e
     else ());
```

Stateful Widgets

What state do the event handlers modify?

How can widgets expose extra this state to the application?

A stateful `label` Widget

```
let label (s: string) : t =
  let r = { contents = s } in
      { repaint =
          (fun (g: Gctx.t) ->
                 Gctx.draw_string g (0,0) r.contents);
        handle = (fun _ _ -> ());
        size = (fun (g: Gctx.t) ->
                 Gctx.text_size g r.contents)
    }
```

- The label “object” can make its string mutable. The three “methods” can encapsulate that string.
- But what if the application wants to change this string in response to an event?

A stateful `label` Widget

widget.ml

```
type label_controller = { set_label: string -> unit }

let label (s: string) : t * label_controller =
  let r = { contents = s } in
    ({ repaint =
         (fun (g: Gctx.t) ->
            Gctx.draw_string g (0,0) r.contents);
       handle = (fun _ _ -> ());
       size = (fun (g: Gctx.t) ->
            Gctx.text_size g r.contents)
    },
    { set_label = fun (s: string) -> r.contents <- s })
```

- A *controller* object gives access to the shared 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 Handling Summary

- An *event* is a signal
  - e.g. a mouse click or release, mouse motion, or keypress
- Events carry data
  - e.g. state of the mouse button, the coordinates of the mouse, the key 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)
  - e.g. a button handles a mouse click event
- Typically, the widget that handles an event *updates some state* of the GUI
  - e.g. to record whether the light is on and change the label of the button
- User sees the reaction to the event when the GUI repaint itself
  - e.g. button has new label, canvas is a new color

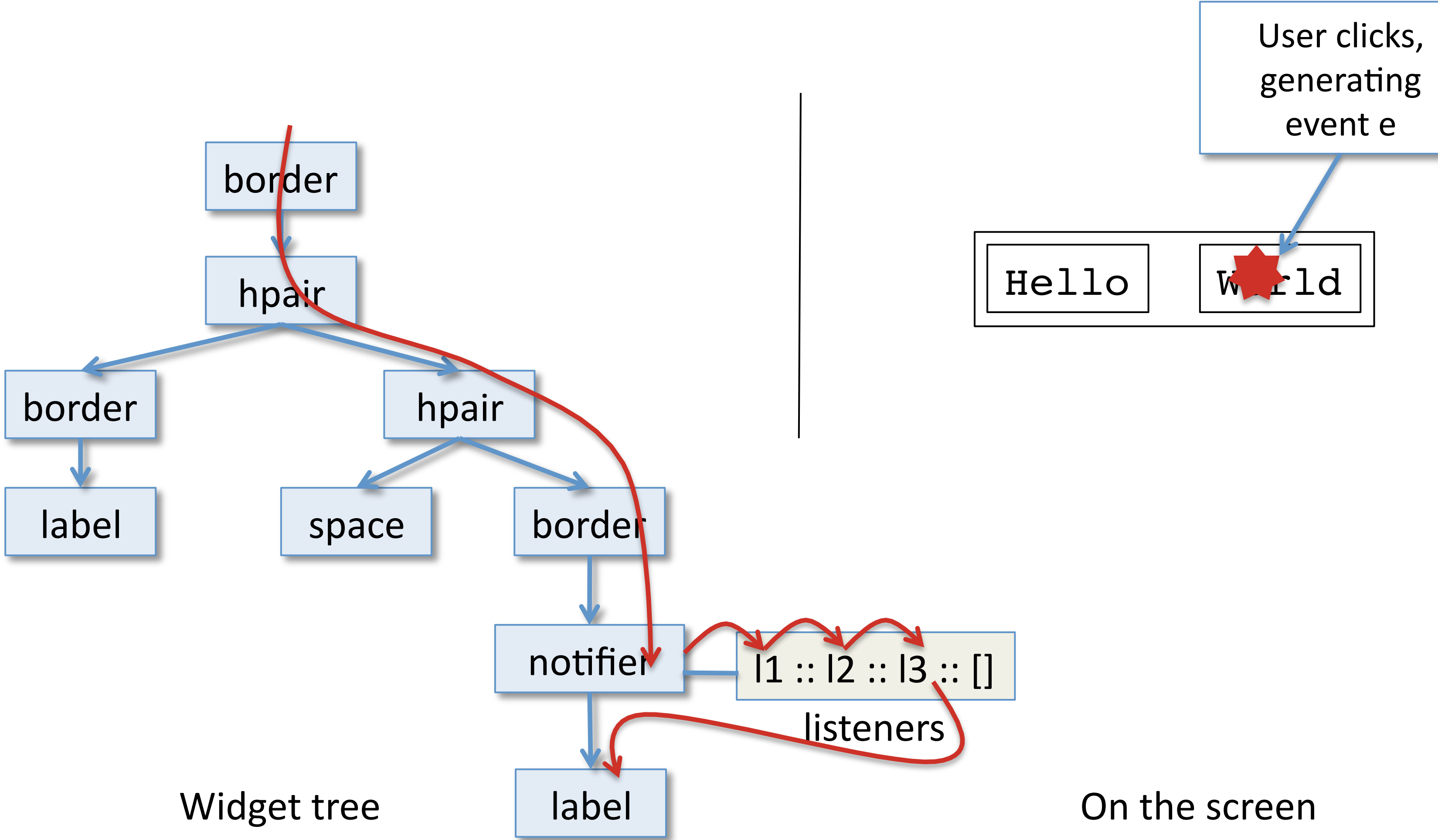
Event Listeners

How to react to events in a modular way?

Event Listeners

- Widgets may want to react to many *different* sorts of events
- Example: Button
  - button click: changes the state of the paint program and button label
  - mouse movement: tooltip? highlight?
  - key press: provide keyboard access to the button functionality?
- These reactions should be independent
  - Each sort of event handled by a different *event listener* (i.e. a first-class function*)*
  - Reactive widgets may have *several* listeners to handle a triggered event
  - Listeners react in sequence, earlier ones may prevent the event from propagating
- A *notifier* is a container widget that adds event listeners to a node in the widget hierarchy
- Note: this way of structuring event listeners is based on Java's Swing Library design (we use Swing terminology).

Listeners and Notifiers Pictorially



Notifiers

- A *notifier* is a container widget that adds event listeners to a node in the widget hierarchy.
- The *event listeners* "eavesdrop" on the events flowing through the node
  - The event listeners are stored in a list
  - They react in order, if one of them handles the event the later ones do not hear it
  - If none of the listeners handle the event, then the event continues to the child widget
- List of event listeners can be updated by using a notifier_controller

Listeners

widget.ml

```
type listener_result =
   | EventFinished
   | EventNotDone

type listener = Gctx.t -> Gctx.event -> listener_result

(* Performs an action upon receiving a mouse click. *)
let mouseclick_listener (action: unit -> unit) : listener =
  fun (g:Gctx.t) (e: Gctx.event) ->
    if Gctx.button_pressed g e
    then (action (); EventFinished)
    else EventNotDone
```

- A listener returns EventFinished if it handled the event (i.e. the event should not be passed on) and EventNotDone 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

widget.ml

```
type notifier_controller = { add_listener: listener -> unit }

let notifier (w: t) : t * notifier_controller =
  let listeners = { contents = [] } in
   ({repaint = w.repaint;
     handle = (fun (g:Gctx.t) (e: Gctx.event) ->
       let rec loop (l: listener list) : unit =
         begin match l with
         | [] -> w.handle g e
         | h::t -> begin match h g e with
                   | EventFinished -> ()
                   | EventNotDone -> loop t
                   end
         end in
         loop listeners.contents);
      size = w.size
    },
    { add_listener =
      fun newl -> listeners.contents <-
                      newl::listeners.contents }
```

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.

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

Buttons (at last!)

widget.ml

```
(* A text button *)
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 button widget is just a label wrapped in a notifier
- Add a mouseclick_listener to the button using the notifier_controller
- (For aesthetic purposes, you can but a border around the button widget.)

Demo: lightswitch.ml

Putting it all together.