

Programming Languages and Techniques (CIS120)

Lecture 19

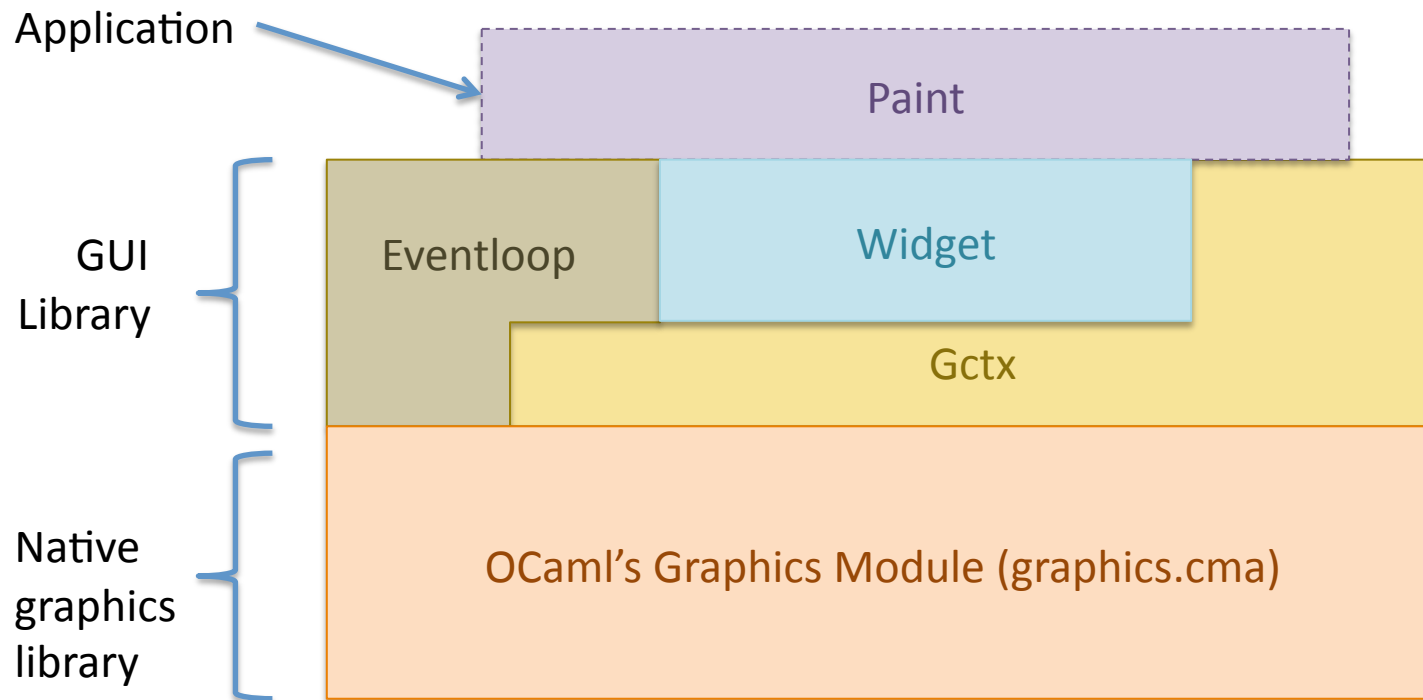
March 5, 2014

GUI Design III: Events

How far have you gotten in HW 6?

1. Task 1 (Better layout)
2. Task 2 (Drawing points)
3. Task 3 (Preview/drag-and-drop)
4. Task 4 (Drawing ellipses)
5. Task 5 (Checkboxes/line thickness)
6. Task 6 (Something cool)
7. Task 0 / Haven't started yet!

Project Architecture



User interaction

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) : 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.

Events

- Whenever user interaction happens, the GUI library generates an “event”
 - Mouse button press or release
 - Mouse movement
 - Keypress

gcxt.mli

```
type event
```

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

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

Reactive Widgets

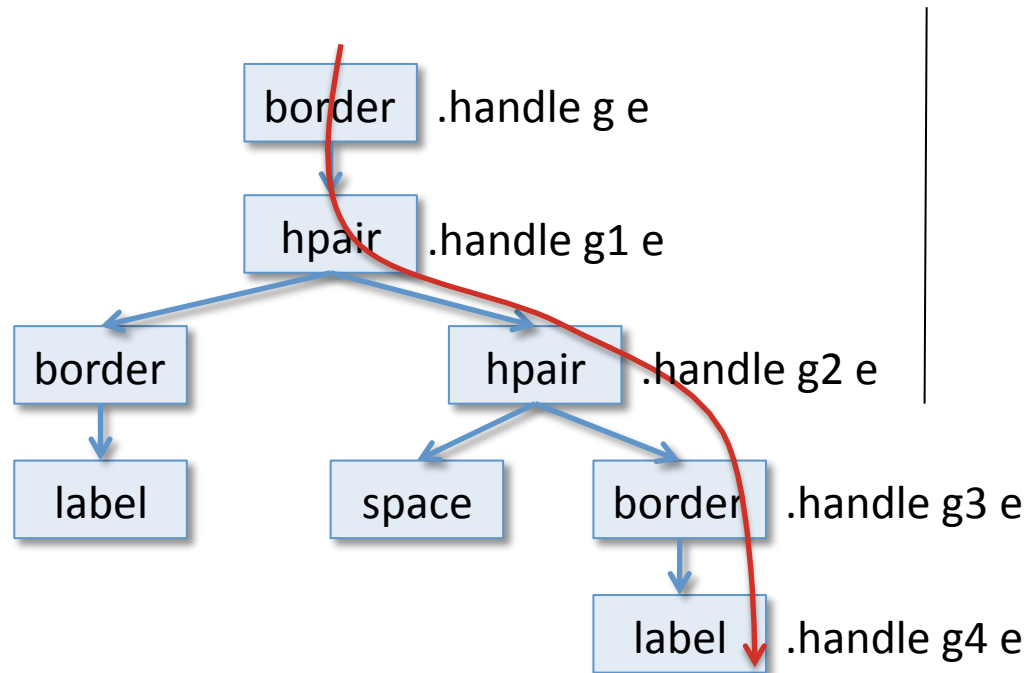
widget.mli

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

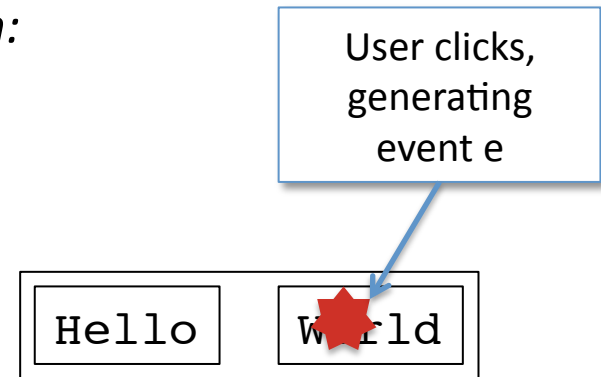
- Widgets have a “method” for handling events
 - The eventloop waits for an event and then gives it to the root widget
 - The widgets forward the event down the tree until some widget handles the event (or no suitable widget is found, in which case the event is ignored)

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 must be translated so the child can interpret the event in its own local coordinates.

widget.ml

```
let border (w:widget):widget =  
  { repaint = ...;  
    size = ...;  
    handle = (fun (g:Gctx.gctx) (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.gctx) (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 ());
```

Demo

eventdemo.ml

Stateful Widgets

What state do the event handlers modify?

How can widgets expose this state to the application?

A stateful `Label` Widget

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

Built-in mutable record type

- 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) : widget * label_controller =
  let r = { contents = s } in
  ({ repaint =
    (fun (g: Gctx.gctx) ->
      Gctx.draw_string g (0,0) r.contents);
    handle = (fun _ _ -> ());
    size = (fun (g: Gctx.gctx) ->
      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

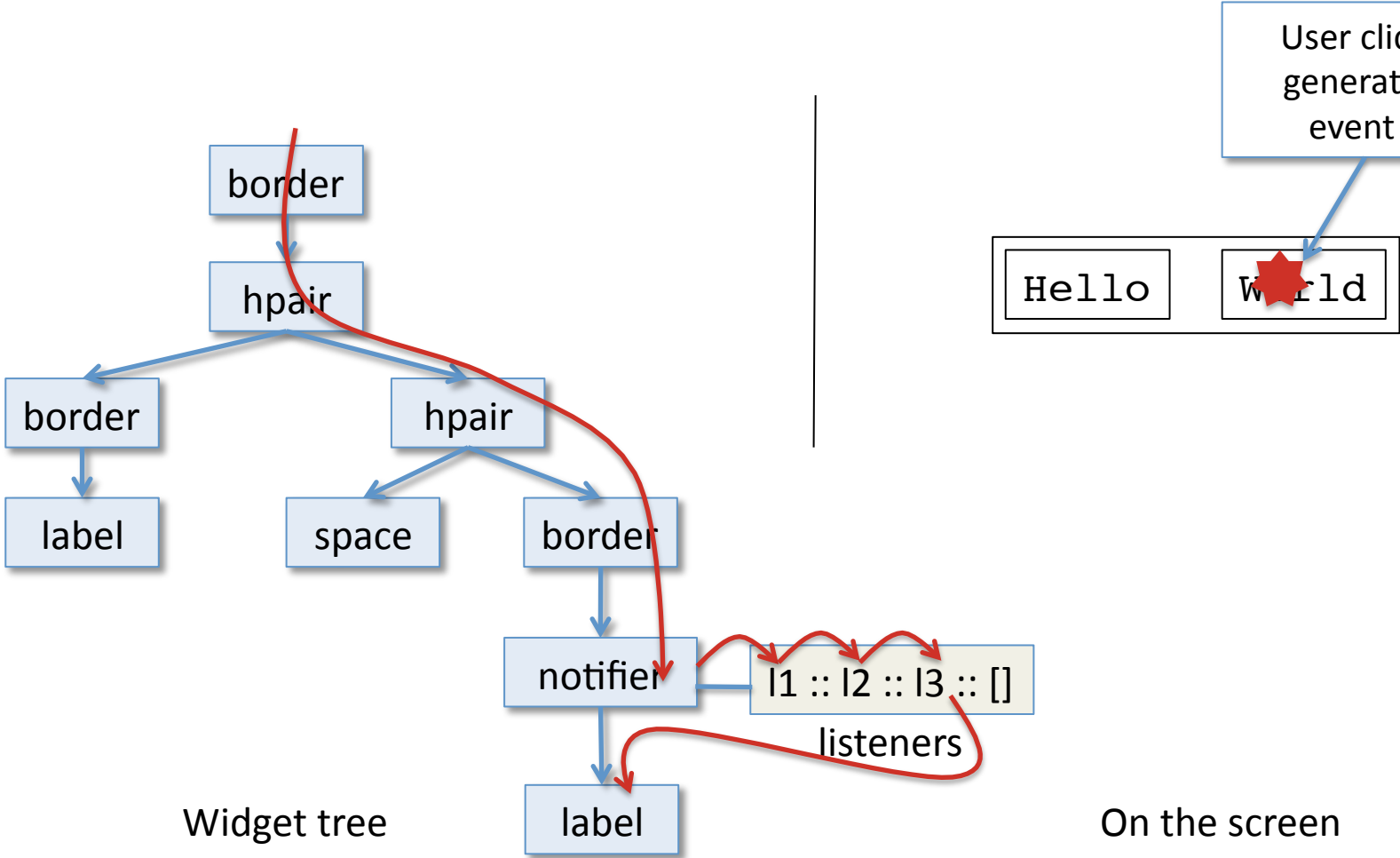
Event Listeners

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.gctx -> Gctx.event -> listener_result  
  
(* Performs an action upon receiving a mouse click. *)  
let mouseclick_listener (action: unit -> unit) : listener =  
  fun (g:Gctx.gctx) (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: widget) : widget * notifier_controller =
  let listeners = { contents = [] } in
  ({repaint = w.repaint;
   handle = (fun (g:Gctx.gctx) (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) : widget * 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 `mousedown_listener` to the button using the `notifier_controller`
- (For aesthetic purposes, you can put a border around the button widget.)

Did you attend class today?

1. Yes