Programming Languages and Techniques (CIS120)

Lecture 19

October 16, 2015

GUI Library Design

Announcements

HW06: GUI programming is available

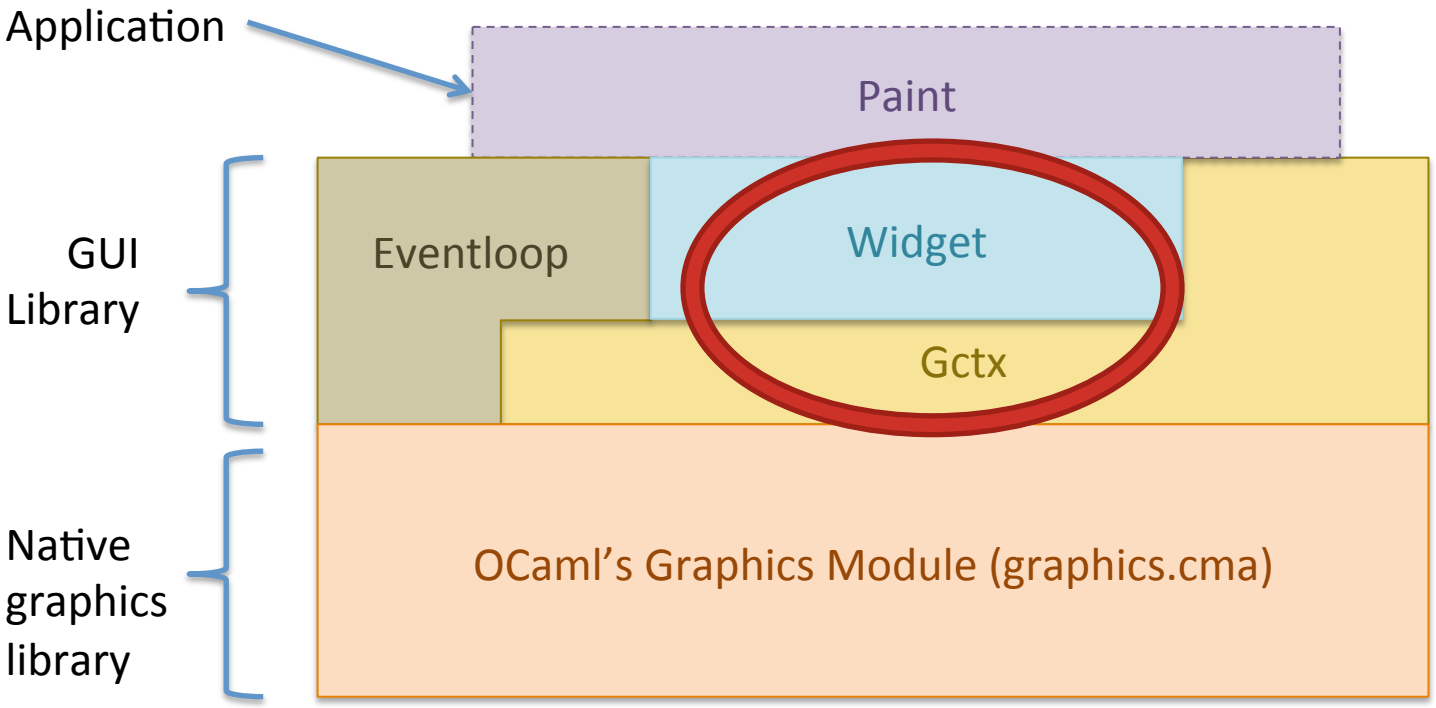
- Due: THURSDAY OCT. 22 at 11:59:59pm
- *Graded manually*
  - *Submission only checks for compilation, no auto tests*
  - *Won't get scores immediately*
  - *Only LAST submission will be graded*
- This project is challenging:
  - Requires working with *multiple* levels of abstraction.
  - Managing state in the paint program is a bit tricky.

GUI Design

putting objects to work

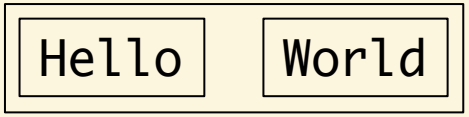
Project Architecture*

*Note: Subsequent program snippets are color-coded according to this diagram.



Goal of the GUI library: provide a consistent layer of abstraction *between* the application (Paint) and the Graphics module.

What code produces the following picture?



1
```
border (label "Hello World”)
```

2
```
border (hpair (label "Hello") (label "World"))
```

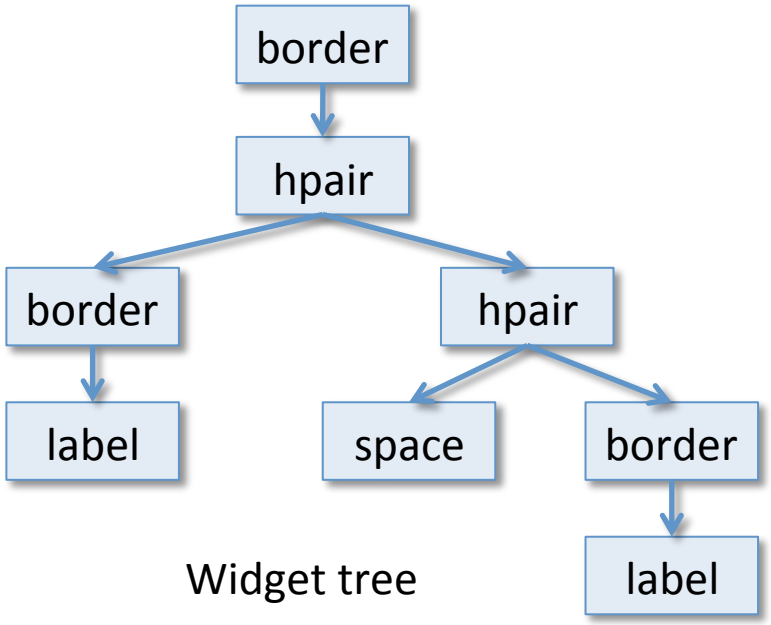
3
```
hpair (label "Hello" )
      (hpair (space (10,10)) (label "World" ))
```

4
```
border (hpair (border (label "Hello" ))
      (hpair (space (10,10))
             (border (label "World" ))))
```

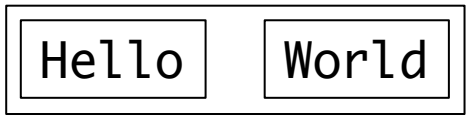
Widget Hierarchy Pictorially

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)))
```



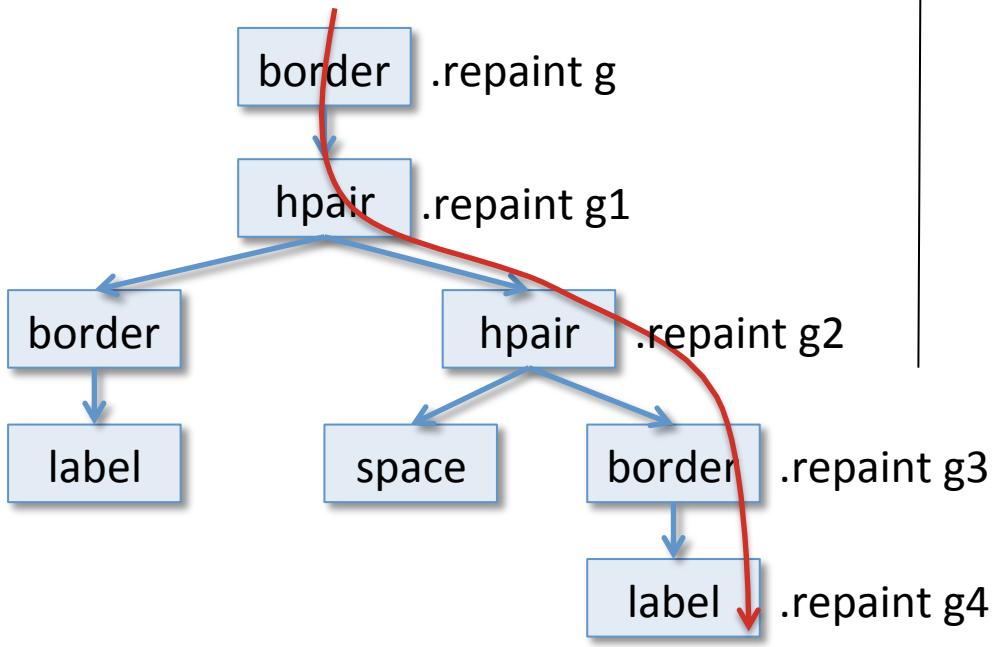
Widget tree

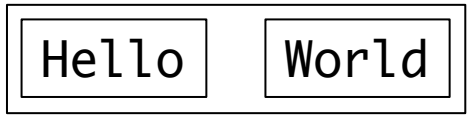


On the screen

Drawing: Containers

Container widgets propagate repaint commands to their children:





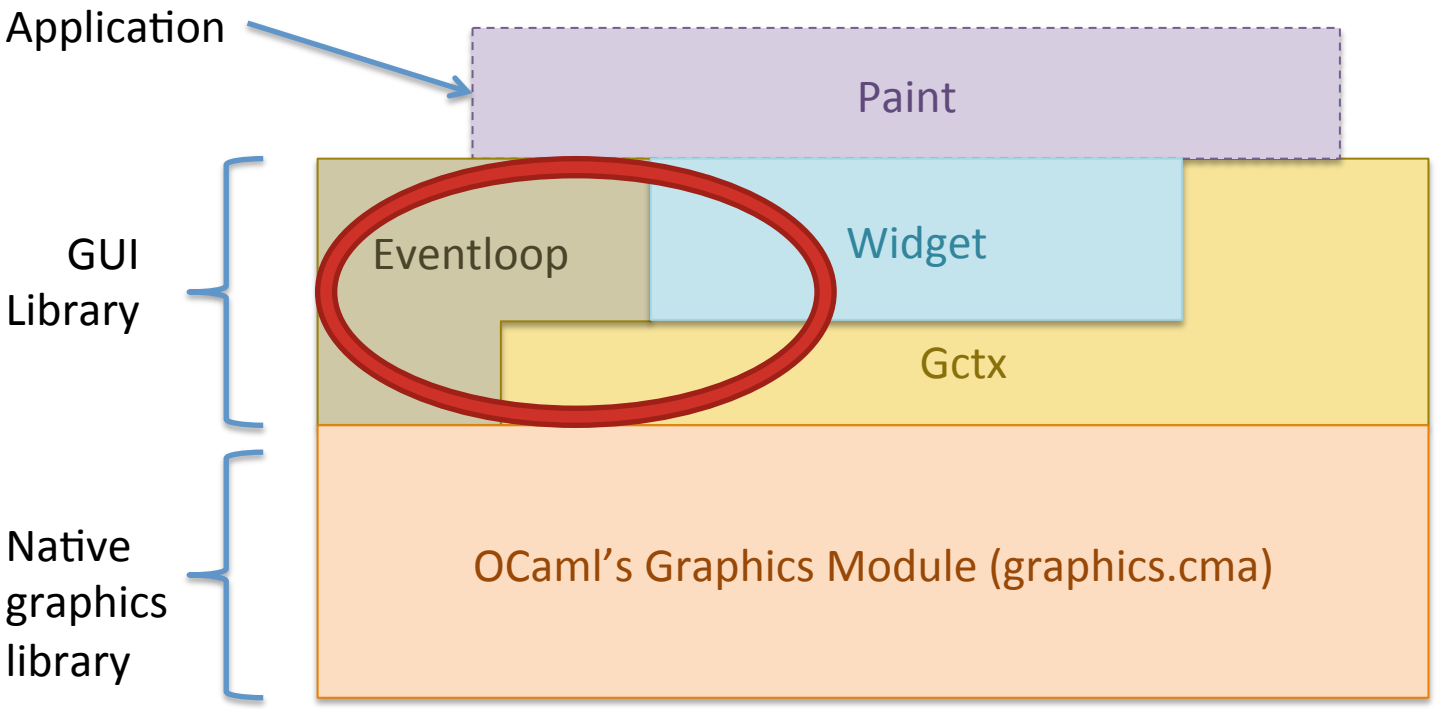
Widget tree

```
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

Project Architecture

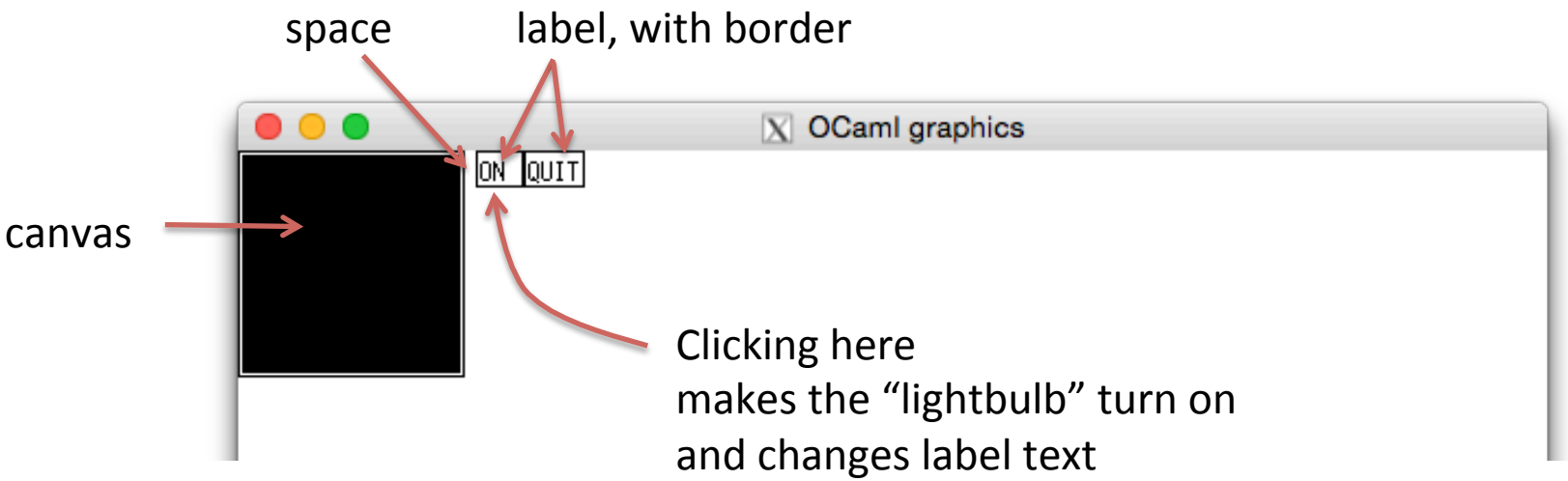


Goal of the GUI library: provide a consistent layer of abstraction *between* the application (Paint) and the Graphics module.

Demo: lightswitch.ml

Putting it all together.

lightbulb demo



User Interactions

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

swdemo.ml

```
let run (w:widget) : unit =

  Gctx.open_graphics ();              (* open graphics window *)

  let g = Gctx.top_level in
  w.repaint g;                      (* repaint the widget once *)

  Graphics.synchronize ();             (* force window update *)
  ignore (Graphics.read_key ())        (* wait for a keypress *)
```

GUI terminology - Eventloop

```
let run (w:widget) : unit =
  Gctx.open_graphics ();
  let g = Gctx.top_level in

  let rec loop () : unit =
    Graphics.clear_graph ();
    w.repaint g;
    Graphics.synchronize ();          (* force window update *)
    wait for user input (mouse movement, key press)
    inform w about the input so widgets can react to it;
    loop ()                           (* tail recursion! *)
  in
    loop ()
```

Solution: The Event Loop

eventloop.ml

```
let run (w:Widget.t) : unit =
  Gctx.open_graphics ();
  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 ()
```

Events

gcxt.mli

```
type event

val wait_for_event : unit -> event

type event_type =
  | KeyPress of char  (* User pressed a key                    *)
  | MouseDown         (* Mouse Button pressed, no movement     *)
  | MouseUp           (* Mouse button released, no movement    *)
  | MouseMove         (* Mouse moved with button up            *)
  | MouseDrag         (* Mouse moved with button down          *)

val event_type : event -> event_type
val event_pos  : event -> gctx -> position
```

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

Reactive Widgets

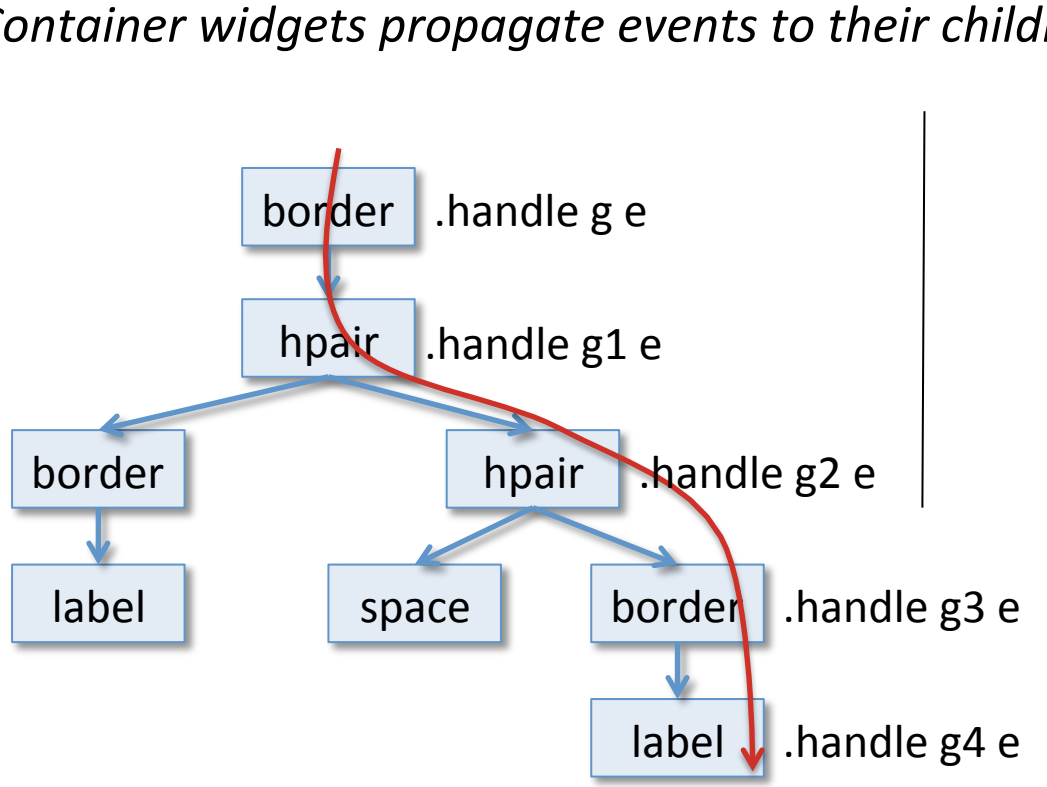
widget.mli

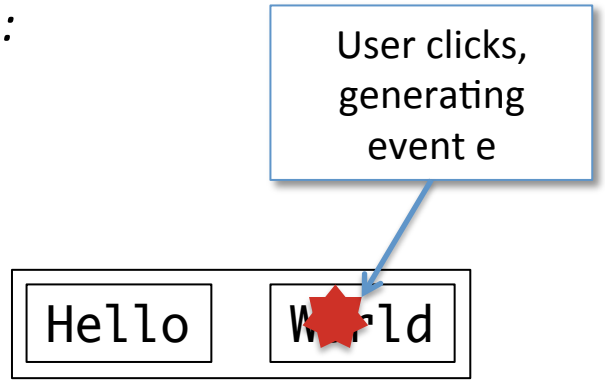
```
type t = {
  repaint : Gctx.gctx -> unit;
  size    : unit -> Gctx.dimension;
  handle  : Gctx.gctx -> Gctx.event -> unit  (* NEW! *)
}
```

- 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, according to the position of the event

Event-handling: Containers

Container widgets propagate events to their children:





Widget tree

On the screen

Routing events through container widgets

Event Handling: Routing

- When a container widget handles an event, it passes the event to the appropriate child
- The Gctx.gctx must be translated so that 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);
  }
```

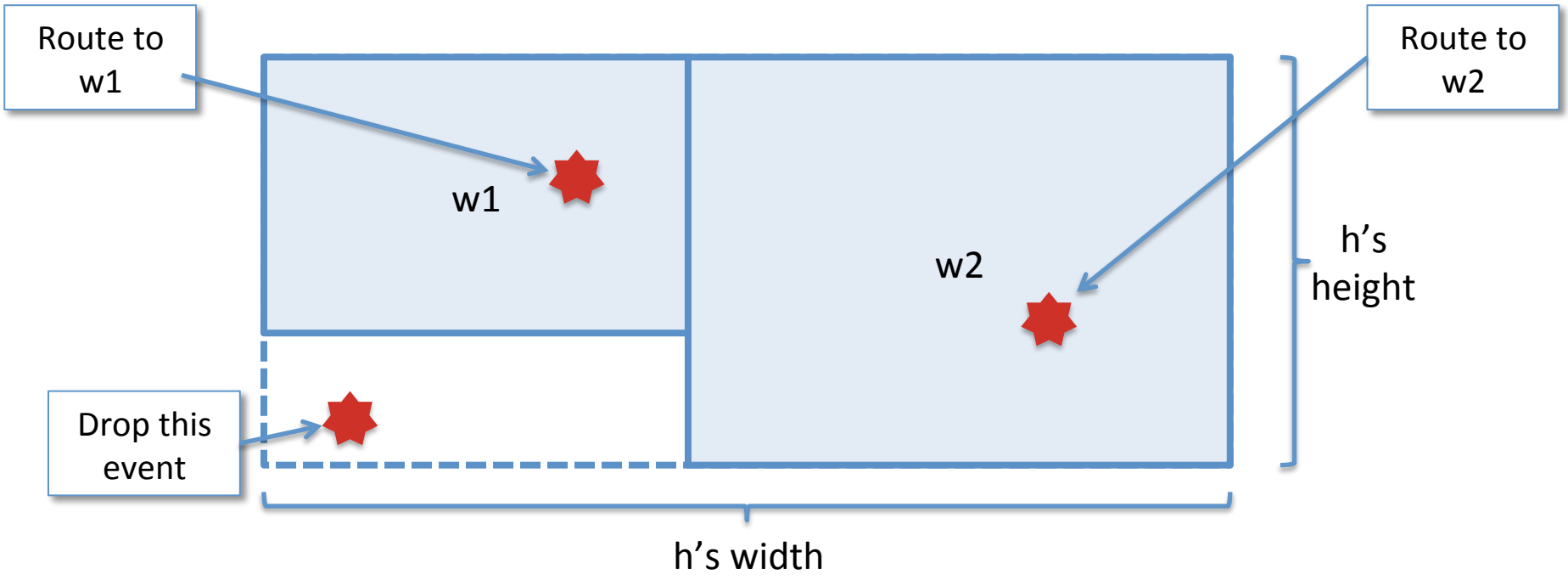
Consider routing an event through an hpair widget constructed by:

```
let hp = hpair w1 w2
```

The event will always be propagated either to w1 or w2.

1. True
2. False

Dropping Events in an HPair



- There are three cases for routing in an hpair.
- An event in the "empty area" should not be sent to either w1 or w2.

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 ())
```

Stateful Widgets

How can widgets react to events?

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 () ->
                Gctx.text_size r.contents)
     }
```

- The label “object” can make its string mutable. The “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 () ->
           Gctx.text_size r.contents)
    },
    { set_label = fun (s: string) -> r.contents <- s })
```

- A *controller* gives access to the shared state.
  - e.g. the `label_controller` object provides a way to set the label

Event Listeners

How to react to events in a modular way?

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

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, all have a chance to see the event
- 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

widget.ml

```
type event_listener = Gctx.gctx -> Gctx.event -> unit

(* Performs an action upon receiving a mouse click. *)
let mouseclick_listener (action: unit -> unit)
                        : event_listener =
  fun (g:Gctx.gctx) (e: Gctx.event) ->
    if Gctx.event_type e = Gctx.MouseDown
    then action ()
```

Notifiers and Notifier Controllers

widget.ml

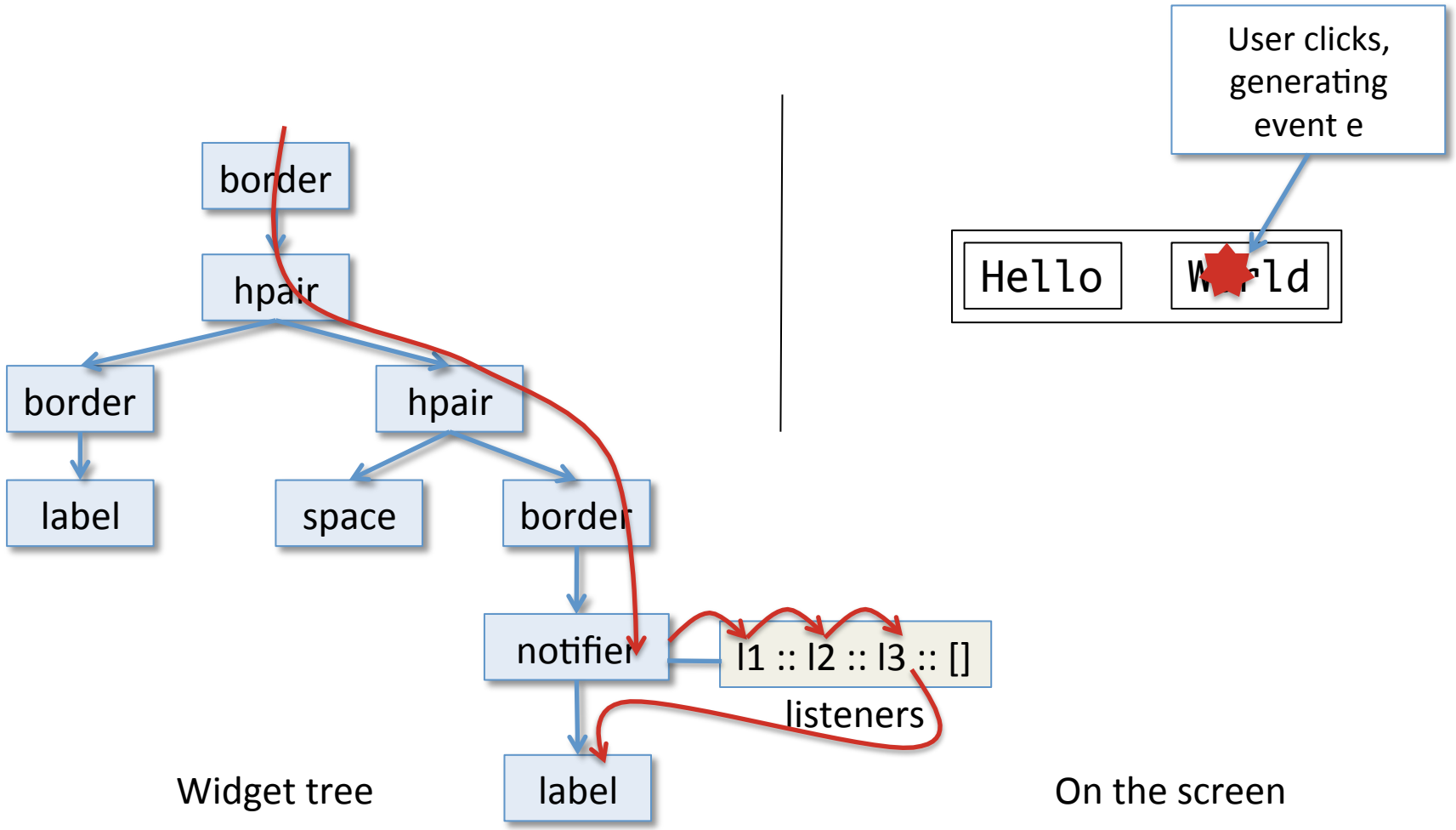
```
type notifier_controller =
     { add_listener : event_listener -> unit }

let notifier (w: widget) : widget * notifier_controller =
  let listeners = { contents = [] } in
  { repaint = w.repaint;
    handle =
      (fun (g: Gctx.gctx) (e: Gctx.event) ->
          List.iter (fun h -> h g e) listeners.contents;
          w.handle g e);
    size = w.size
  },
  { add_event_listener =
      fun (newl: event_listener) ->
          listeners.contents <-
                 newl :: listeners.contents
  }
```

Loop through the list of listeners, allowing each one to process the event. Then pass the event to the child.

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

Listeners and Notifiers Pictorially



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 mouseclick_listener to the button using the notifier_controller
- (For aesthetic purposes, you can but a border around the button widget.)