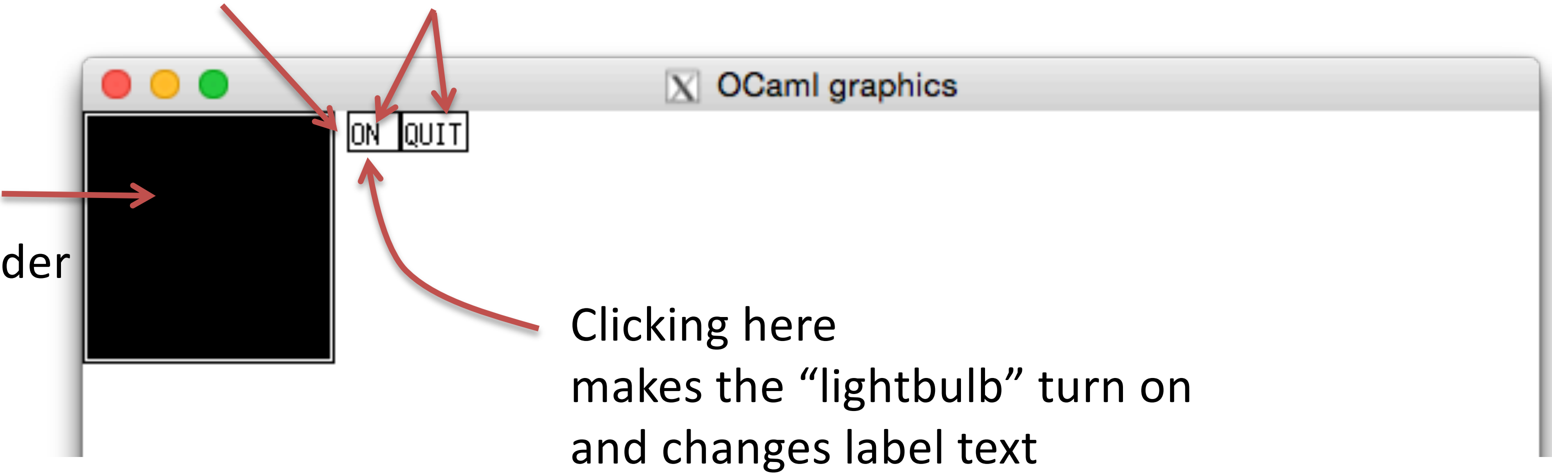
Programming Languages and Techniques (CIS120)

Lecture 20

GUI: Events & State
Chapter 18

lightbulb demo

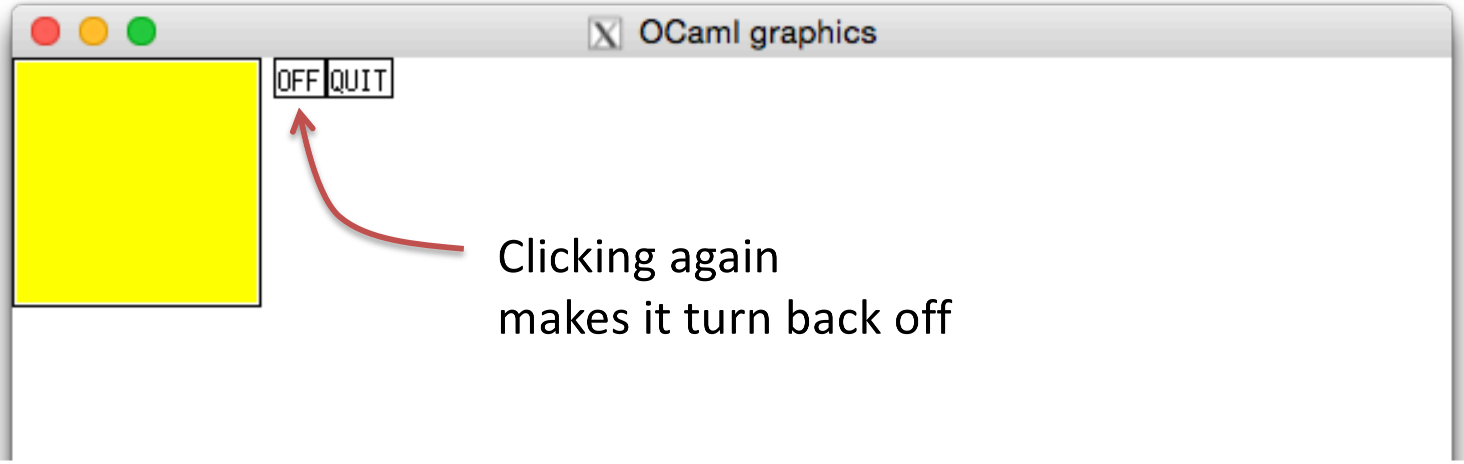


space

label, with border

canvas,
with border

Clicking here
makes the “lightbulb” turn on
and changes label text



Clicking again
makes it turn back off

Events and Event Handling

Event loop with event handling

```
let run (w:widget) : unit =
  let g = Gctx.top_level  in     …create the initial gctx…
  Graphics.loop                  …wait for user input
    (fun e ->
       clear_graph ();
       w.handle g e;             …inform widget about the event…
       w.repaint g)             …update the widget's appearance…
```

Eventloop

```
let rec loop (f: event -> unit) : unit =
  let e = wait_next_event () in
  f e;
  loop f
```

Graphics

Events

gctx.mli

```
type 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
```

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

Reactive Widgets

widget.mli

```
type widget = {
  repaint : Gctx.gctx -> unit;
  size    : unit -> Gctx.dimension;
  handle  : Gctx.gctx -> Gctx.event -> unit
}
```

- Widgets now 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
  - In a widget's handle function, **`Gctx.event_pos`** determines the location of the event relative to widget's coordinate system

gctx.mli

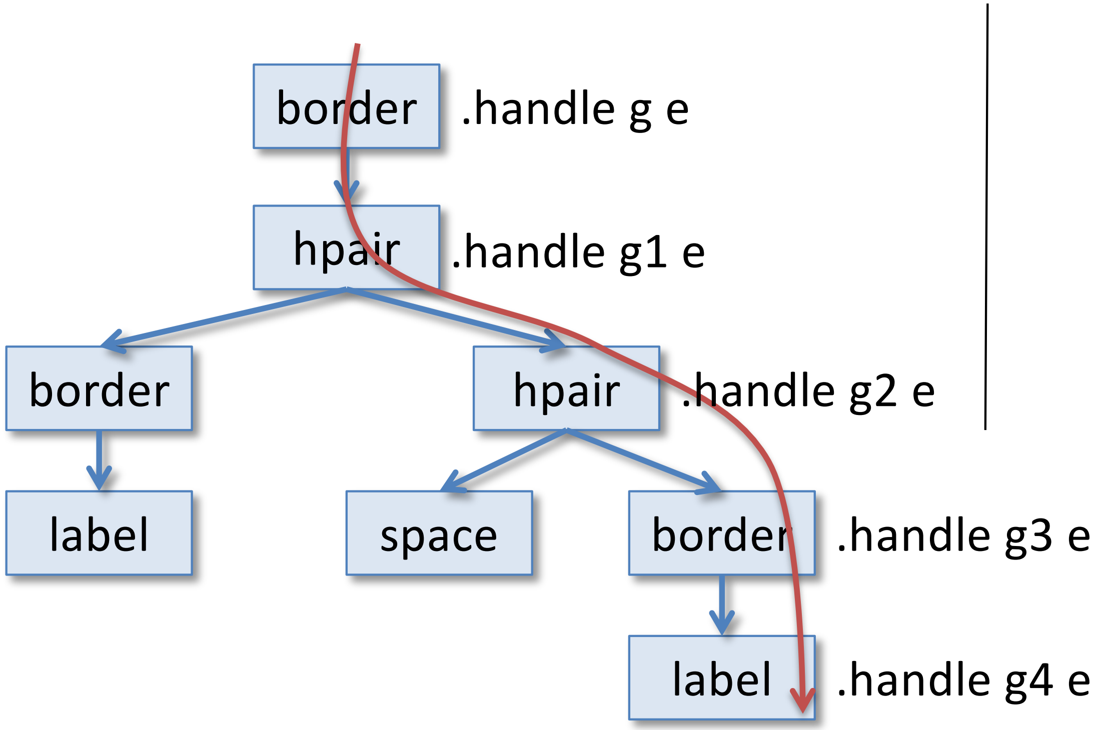
```
type event

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

Routing events through container widgets

Event-handling: Containers

Container widgets propagate events to their children:



border .handle g e

hpair .handle g1 e

border

hpair .handle g2 e

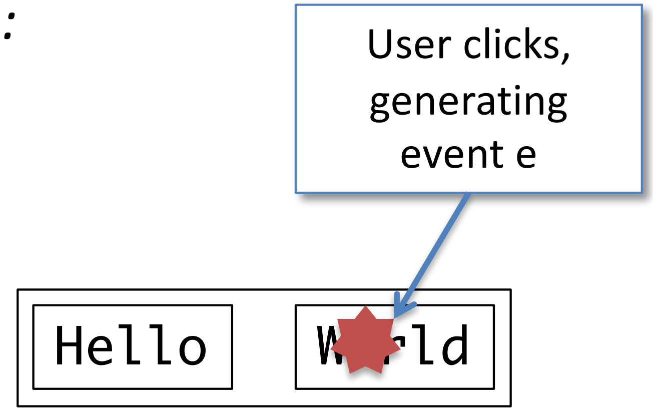
label

space

border .handle g3 e

label .handle g4 e

Widget tree



User clicks, generating event e

Hello

World

On the screen

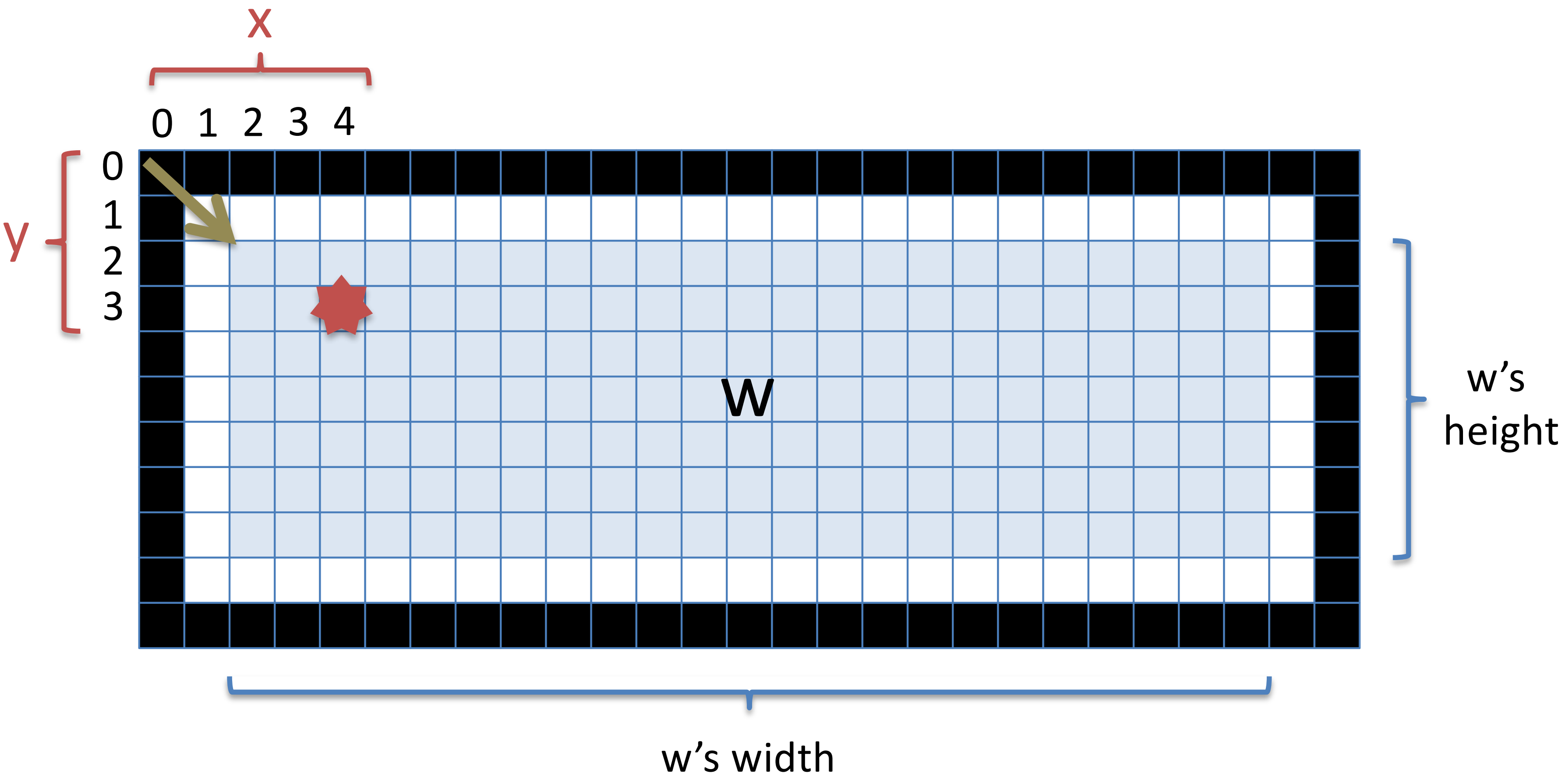
Event Handling: Routing

- 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);
  }
```

Routing events through Border Widgets



`let b = border w`

Say `b` is asked to handle an event, at some position `(x,y)`

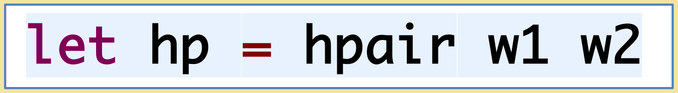
Then `b` should call `w's` event handler, with an updated gctx so that `w will` see the event at `(x-2,y-2)`

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 ())
   then w1.handle g e
   else
   let g = (Gctx.translate g (fst (w1.size ()), 0)) in
     if event_within g e (w2.size ())
     then w2.handle g e
     else ())
```

Consider routing an event through an hpair widget constructed by:



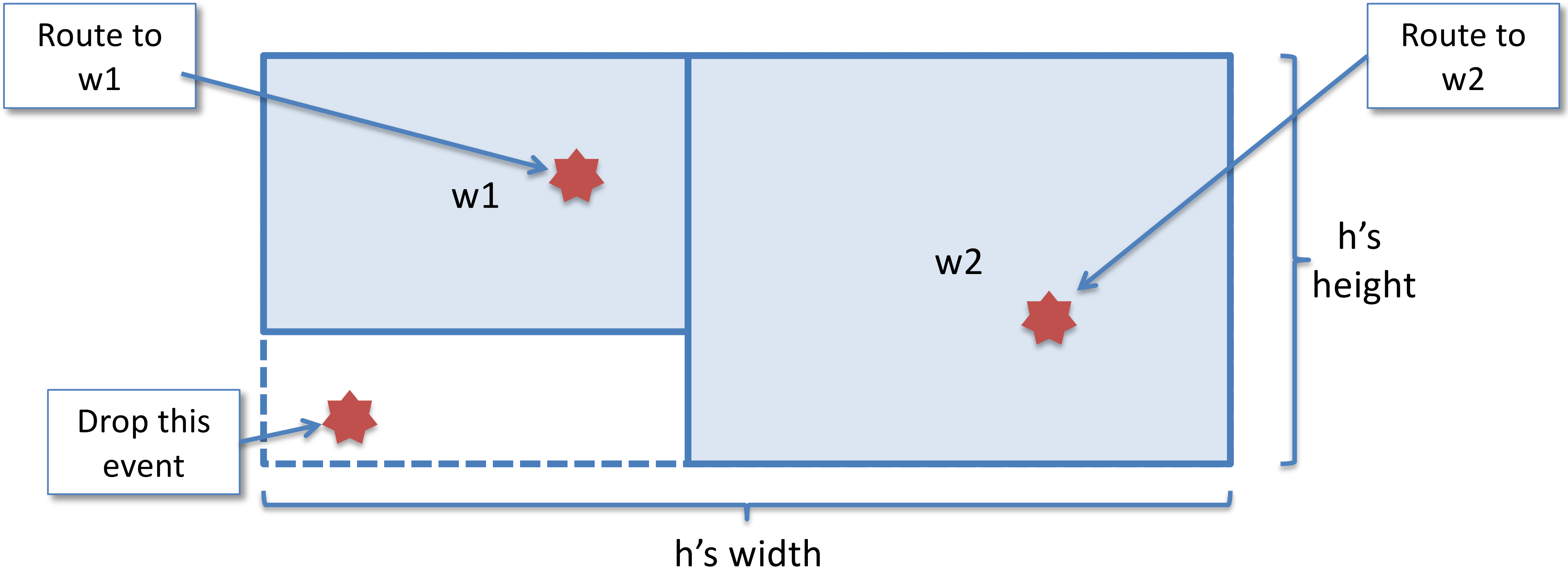
```
let hp = hpair w1 w2
```

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

1. True
2. False

Answer: False

Routing events through hpair widgets



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

Stateful Widgets

How can widgets react to events?

(not very useful first stab at a)

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 store its string in a mutable location. The methods can refer to this mutable string.
- But how can applications change this string?

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 label consists of two parts: the widget and its controller
- A *controller* gives access to the shared state.
  - Here, the `label_controller` object returned by `label` provides a way to modify the label string

Event Listeners

How can we make handling events more flexible?

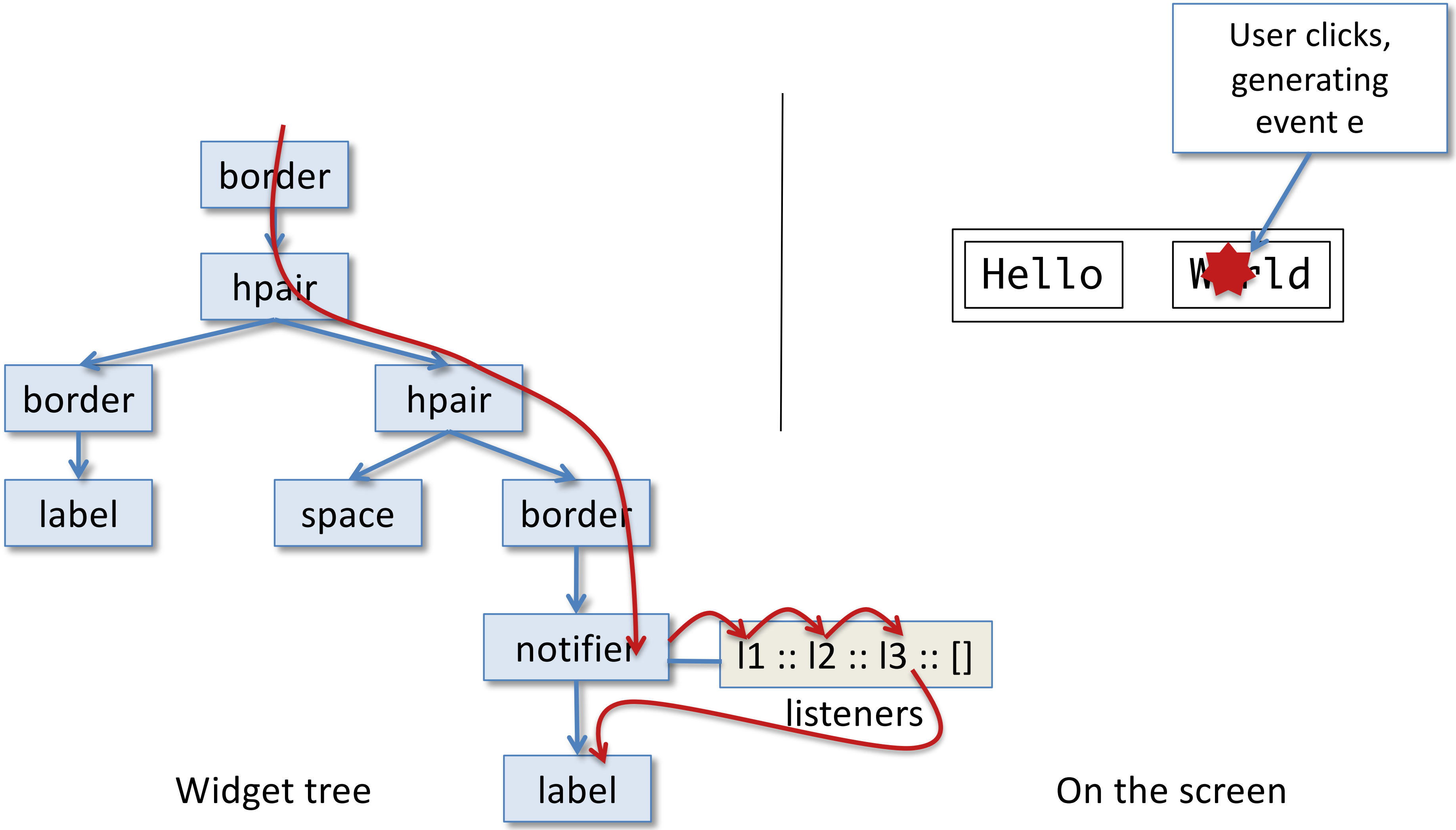
See notifierdemo.ml

(distributed with the homework)

Handling multiple event types

- Problem: *Widgets may want to react to many different 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
- Solution: notifier

Listeners and Notifiers Pictorially



User clicks, generating event e

Widget tree

On the screen

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

- 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).
- *Event listeners* “eavesdrop” on the events flowing through the notifier
  - The event listeners are stored in a list
  - They react in order
  - Then the event is passed down to the child widget
- Event listeners can be added by using a notifier_controller

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;
    size    = w.size
    handle  =
      (fun (g: Gctx.gctx) (e: Gctx.event) ->
          List.iter (fun h -> h g e) listeners.contents;
          w.handle g e);
},
  { 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.

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, we could also put a border around the label widget.)

True or False: One can use a notifier and label to create a button that toggles the states of *two* separate lightbulb canvases.

Answer: True

onoff.ml — changing state on a button click

DEMO: ONOFF