GUI: Events & State
Announcements

• HW05: GUI programming
  – Due: Thursday Mar 1st at 11:59:59pm
  – Graded (mostly) manually
    • Submission checks for compilation, few auto tests
    • Only LAST submission will be graded
  – This project is challenging:
    • IT IS NOW TOO LATE TO START EARLY!
Review: Events and Event Handling
Project Architecture

Application

GUI Library

Native graphics library

OCaml’s Graphics Module (graphics.cma)

Gctx

Eventloop

Widget

Paint

Native graphics library

Native graphics library
lightbulb demo

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

Clicking again makes it turn back off.
Handling Events

- Main loop of any GUI application:

```ocaml
let run (w:widget) : unit =
  let g = Graphics.top in  ...create the initial gctx...

  Graphics.loop
      Graphics.Button_up; Graphics.Key_pressed]
    (fun status ->
      clear_graph ();
      w.repaint g ();  ...repaint relative to g...
      begin match event_of_status status with
        | None -> ()     ...spurious status update, do nothing...
        | Some e -> w.handle g e  ...let widget handle the event...
      end
    )
```

(eventloop.ml)
**Events**

### gcxt.mli

```ocaml
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
```

*Remember:*

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

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, according to the position of the event

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

Container widgets propagate events to their children:

User clicks, generating event e

On the screen

Widget tree

Hello  World
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.

```
let border (w:widget):widget =
{
  repaint = ...;
  size = ...;
  handle = (fun (g:Gctx.gctx) (e:Gctx.event) ->
    w.handle (Gctx.translate g (2,2)) e);
}
```
• 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 ())
  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 ()
)
New Stuff: Stateful Widgets

How can widgets react to events?
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?

```ocaml
define (label : 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) }
```
A stateful label 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?
Listeners

(* Performs an action upon receiving a mouse click. *)

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

let mouseclick_listener (action: unit -> unit) : event_listener =
  fun (g:Gctx.gctx) (e: Gctx.event) ->
    if Gctx.event_type e = Gctx.MouseDown
    then action ()
```

widget.ml
Handling multiple event types

• Problem: 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

• Solution: notifier
Listeners and Notifiers Pictorially

User clicks, generating event e

Widget tree

On the screen
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 node
  – The event listeners are stored in a list
  – They react in order
  – Even 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
Notifiers and Notifier Controllers

```ocaml
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.
Buttons (at last!)

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.)
lightbulb demo

Clicking here makes the "lightbulb" turn on and changes label text.

Clicking again makes it turn back off.
Demo: onoff.ml

Changing the label on a button click