Announcements

• HW05: GUI programming
  – Due: Tuesday, October 23rd at 11:59:59pm
  – Graded (mostly) manually
    • Submission checks for compilation, few auto tests
    • Only LAST submission will be graded
  – This project is challenging:
    • Requires working with multiple levels of abstraction.
    • Managing state in the paint program is tricky.
    • START IMMEDIATELY!!
Building a GUI library & application
Review: A “Hello World” application

(* 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)))

On the screen

Widget tree
Module: EventLoop

Top-level driver
The event loop is the main "driver" of a GUI application
- For now: focus on how widgets are drawn on the screen
- Later: deal with event handling
GUI terminology: “event loop”

- Main loop of any GUI application:

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

...wait for user input (mouse movement, key press)...
    Graphics.Button_up; Graphics.Key_pressed]
(fun status ->
    clear_graph ();
    w.repaint ();  ...update the widget's appearance...
    ...
)
```

```ocaml
let rec loop (es:event list) (f:status -> unit) : unit =
let status = wait_next_event es in
(f status); loop es f
```

- Takes “top-level” widget w as argument. That widget contains all others in the application.
Challenge: How can we make it so that the functions that draw widgets in different places on the window are location independent?
Container widgets propagate repaint commands to their children:

Challenge: How can we make it so that the functions that draw widgets in different places on the window are location independent?
Module: Gctx

Contextualizes graphics drawing operations
Challenge: Widget Layout

- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a *graphics context* to make drawing *relative* to the widget’s current position.

The graphics context isolates the widgets from the Graphics module.
GUI terminology – Graphics Context

• Wraps OCaml Graphics library; puts drawing operations “in context”

• Translates coordinates
  – *Flips* between OCaml and “standard” coordinates so origin is top-left
  – *Translates* coordinates so all widgets can pretend that they are at the origin

• Also aggregates information about the way things are drawn
  – foreground color
  – line width
let top = Gctx.top_level

let nctx = Gctx.translate top (dx,dy)

draw_string top (0,10) "CIS 120"
draw_string nctx (0,10) "CIS 120"

repaint = fun g ->
  draw_rect g (0,0) (20,20);
  draw_string g (0,10) "CIS 120"
OCaml vs. *Standard* Coordinates

Standard (0,0) vs. OCaml (0,0)

Size $x$ ()

Size $y$ ()

$(x, y)$

Standard $(x, y) = \text{OCaml } (x, \text{size}_y() - y)$
Module Gctx

(** The main (abstract) type of graphics contexts. *)

```ml
type gctx
```

(** The top-level graphics context *)

```ml
val top_level : gctx
```

(** A widget-relative position *)

```ml
type position = int * int
```

(** Display text at the given (relative) position *)

```ml
val draw_string : gctx -> position -> string -> unit
```

(** Draw a line between the two specified positions *)

```ml
val draw_line : gctx -> position -> position -> unit
```

(** Produce a new gctx shifted by (dx,dy) *)

```ml
val translate : gctx -> int * int -> gctx
```

(** Produce a new gctx with a different pen color *)

```ml
val with_color : gctx -> color -> gctx
```
Event loop with Gctx

- Main loop of any GUI application:

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

  (fun status ->
    clear_graph ();
    w.repaint g;  ...repaint relative to g...
    ...inform w about the event so widgets can react to it...
  )
```
Drawing containers using graphics contexts

*Container widgets propagate repaint commands to their children, with appropriately modified graphics contexts:*
Widget Layout

Building blocks of GUI applications

see simpleWidget.ml
Challenge: Widget Layout

- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a *graphics context* to make drawing *relative* to the widget’s current position

![Diagram of widget layout](image)

- The graphics context isolates the widgets from the Graphics module.
Simple Widgets

(* An interface for simple GUI widgets *)

```ocaml
type widget = {
  repaint : Gctx.gctx -> unit;
  size : unit -> (int * int)
}
val label : string -> widget
val space : int * int -> widget
val border : widget -> widget
val hpair : widget -> widget -> widget
val canvas : int * int -> (Gctx.gctx -> unit) -> widget
```

- You can ask a simple widget to repaint itself
- You can ask a simple widget to tell you its size
- Repainting is relative to a graphics context
(* A simple widget that puts some text on the screen *)
let label (s:string) : widget =
{
    repaint = (fun (g:Gctx.gctx) -> Gctx.draw_string g (0,0) s);
    size = (fun () -> Gctx.text_size s)
}

(* A "blank" area widget -- it just takes up space *)
let space ((x,y):int*int) : widget =
{
    repaint = (fun (_,Gctx.gctx) -> ());
    size = (fun () -> (x,y))
}
The canvas Widget

- Region of the screen that can be drawn upon
- Has a fixed width and height
- Parameterized by a repaint method
  - Use the Gctx drawing routines to draw on the canvas

```ocaml
let canvas ((w,h):int*int) (repaint: Gctx.gctx -> unit) : widget =
{
  repaint = repaint;
  size = (fun () -> (w,h))
}
```

simpleWidget.ml
Nested Widgets

Containers and Composition
The Border Widget Container

- **let** `b = border w`
- Draws a one-pixel wide border around contained widget `w`
- `b`’s size is slightly larger than `w`’s (+4 pixels in each dimension)
- `b`’s repaint method must call `w`’s repaint method
- When `b` asks `w` to repaint, `b` must *translate* the Gctx.t to (2,2) to account for the displacement of `w` from `b`’s origin
The Border Widget

simpleWidget.ml

```
let border (w:widget):widget =
{
  repaint = (fun (g:Gctx.gctx) ->
    let (width, height) = w.size () in
    let x = width + 3 in
    let y = height + 3 in
    Gctx.draw_line g (0,0) (x,0);
    Gctx.draw_line g (0,0) (0,y);
    Gctx.draw_line g (x,0) (x,y);
    Gctx.draw_line g (0,y) (x,y);
    let gw = Gctx.translate g (2,2) in
    w.repaint gw);

  size = (fun () ->
    let (width, height) = w.size () in
    (width+4, height+4))
}
```

Draw the border

Display the interior
The `hpair` Widget Container

- `let h = hpair w1 w2`
- Creates a horizontally adjacent pair of widgets
- Aligns them by their top edges
  - Must translate the Gctx when repainting `w2`
- Size is the `sum` of their widths and `max` of their heights
The hpair Widget

```
let hpair (w1: widget) (w2: widget) : widget =
{
  repaint = (fun (g: Gctx.gctx) ->
    let (x1, _) = w1.size () in begin
      w1.repaint g;
      w2.repaint (Gctx.translate g (x1,0))
      (* Note translation of the Gctx *)
    end);

  size = (fun () ->
    let (x1, y1) = w1.size () in
    let (x2, y2) = w2.size () in
    (x1 + x2, max y1 y2))
}
```

Translate the Gctx to shift w2’s position relative to widget-local origin.
(* 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)))
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
Container Widgets for layout

hlist is a container widget. It takes a list of widgets and turns them into a single one by laying them out horizontally (using hpair).

```ml
define (color_toolbar : widget) = hlist
    [ color_button black;   spacer;
    color_button white;    spacer;
    color_button red;      spacer;
    color_button green;    spacer;
    color_button blue;     spacer;
    color_button yellow;   spacer;
    color_button cyan;     spacer;
    color_button magenta ]
```

paint.ml
Events and Event Handling
Goal of the GUI library: provide a consistent layer of abstraction between the application (Paint) and the Graphics module.
Demo: onoff.ml

Reacting to events
lightbulb demo

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

Clicking again makes it turn back off.
Do you know how you would use the (simple) widget library to define the layout of this application?

I don't know how to start

I may have it, but I'm not sure

I'm sure I've got it
Do you know how you would use the (simple) widget library to define the layout of this application?

1. I don't know how to start
2. I may have it, but I'm not sure
3. I'm sure I've got it

```ocaml
type widget = {
  repaint : Gctx.gctx -> unit;
  size : unit -> (int * int)
}
val label : string -> widget
val space : int * int -> widget
val border : widget -> widget
val hpair : widget -> widget -> widget
val canvas : int * int -> (Gctx.gctx -> unit) -> widget
```
lightbulb demo

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

Clicking again makes it turn back off.
User Interactions

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

```ocaml
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 *)
```

swdemo.ml
User Interactions

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

• Answer: the eventloop!
Handling Events

• Main loop of any GUI application:

```ocaml
let run (w:widget) : unit =
  let g = Gctx.top_level in    ...create the initial gbcx...

    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

``` Ocaml
module gcxt = ...

type event
val wait_for_event : unit -> event

type event_type = ...
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

- 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 t = {
  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
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.

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

```
let hp = hpair w1 w2
```
Consider routing an event through an hpair widget constructed by:

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

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

1. True
2. False

Answer: False
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

```ocaml
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

```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
Demo: onoff.ml

Changing the label on a button click
When a widget's handle function receives an event, it should also call functions from the Gctx library to update the view of the widget.
When a widget's handle function receives an event, it should also call functions from the Gctx library to update the view of the widget.

1. True
2. False
3. Not sure

Answer: False