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)))

Hello World

On the screen

Widget tree
Review: 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.
Simple Widgets

(* An interface for simple GUI widgets *)

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 ((w,h):int*int) : widget =
{
  repaint = (fun (_:Gctx.gctx) -> ());
  size = (fun () -> (w,h))
}
The canvas Widget

• Region of the screen that can be drawn upon
• Has a fixed width and height
• Parameterized by a repaint method
  – ...which will directly use the Gctx drawing routines to draw on the canvas

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

simpleWidget.ml
<table>
<thead>
<tr>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not started yet</td>
</tr>
<tr>
<td>Task 0 finished</td>
</tr>
<tr>
<td>Working on tasks 1-4</td>
</tr>
<tr>
<td>Working on task 5</td>
</tr>
<tr>
<td>Working on task 6</td>
</tr>
<tr>
<td>All done!!</td>
</tr>
</tbody>
</table>
Nested Widgets

Containers and Composition
let $b = \text{border } w$

- 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

```ocaml
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)))

Widget hierarchy pictorially:

```
.Border
  .Hpair
    .Border
      .Label
    .Hpair
      .Space
        .Hpair
          .Border
            .Label
```

On the screen:

```
Hello
World
```
Container widgets propagate repaint commands to their children:

```
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)
```
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
let 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]
```
Events and Event Handling
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.
lightbulb demo

canvas, with border

space

label, with border
Event loop with event handling

```
let run (w:widget) : unit =
  let g = Gctx.top_level in
  Graphics.loop
    (fun e ->
      clear_graph ();
      w.handle g e;
      w.repaint g)
```

...create the initial gctx...
...wait for user input

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

...inform widget about the event...
...update the widget's appearance...
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
Container widgets propagate events to their children:

User clicks, generating event e

Hello World

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.

```ocaml
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

```plaintext
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 ()
```