Programming Languages and Techniques (CIS120)

Lecture 18

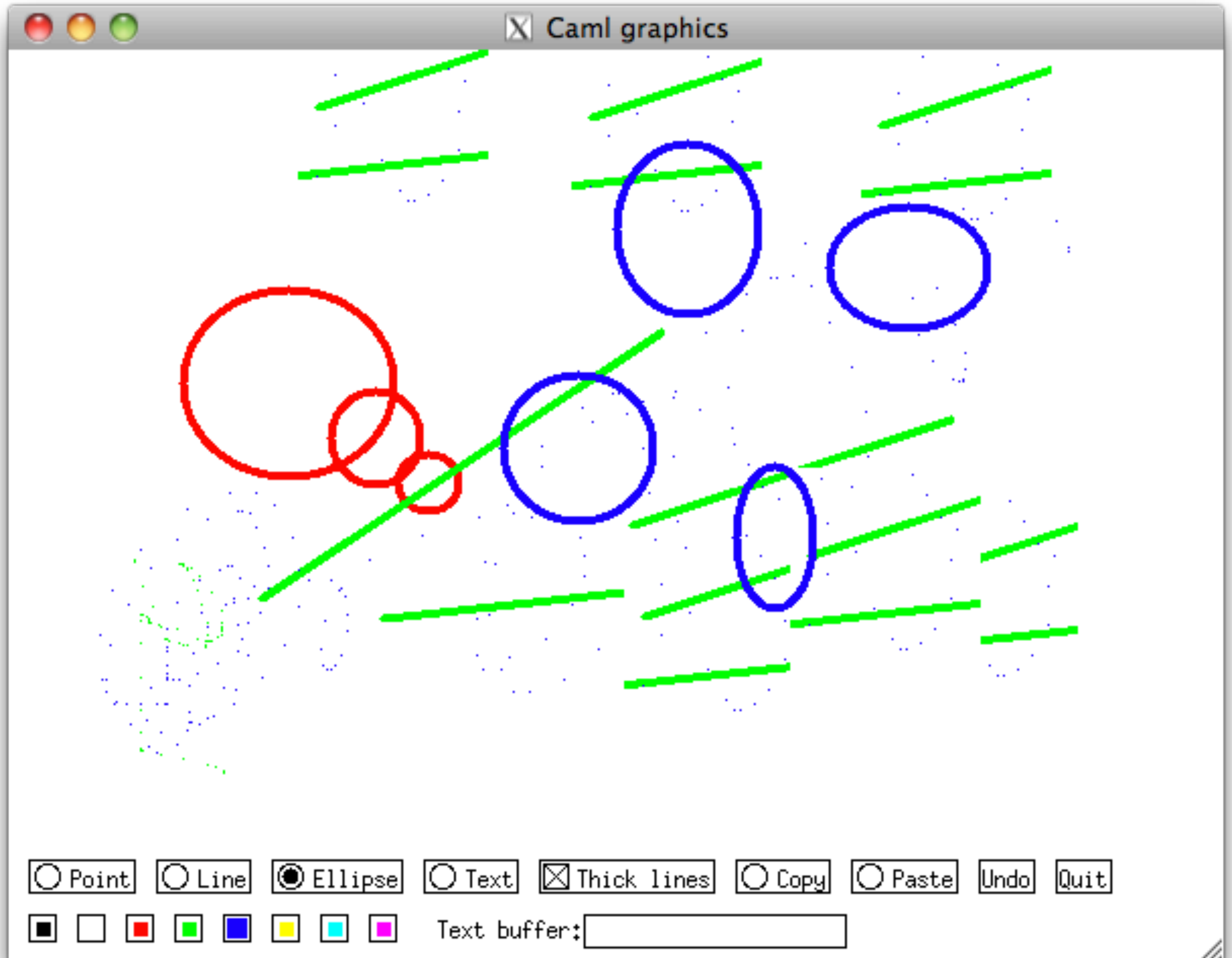
Feb. 22, 2013

GUI Design II: Layout

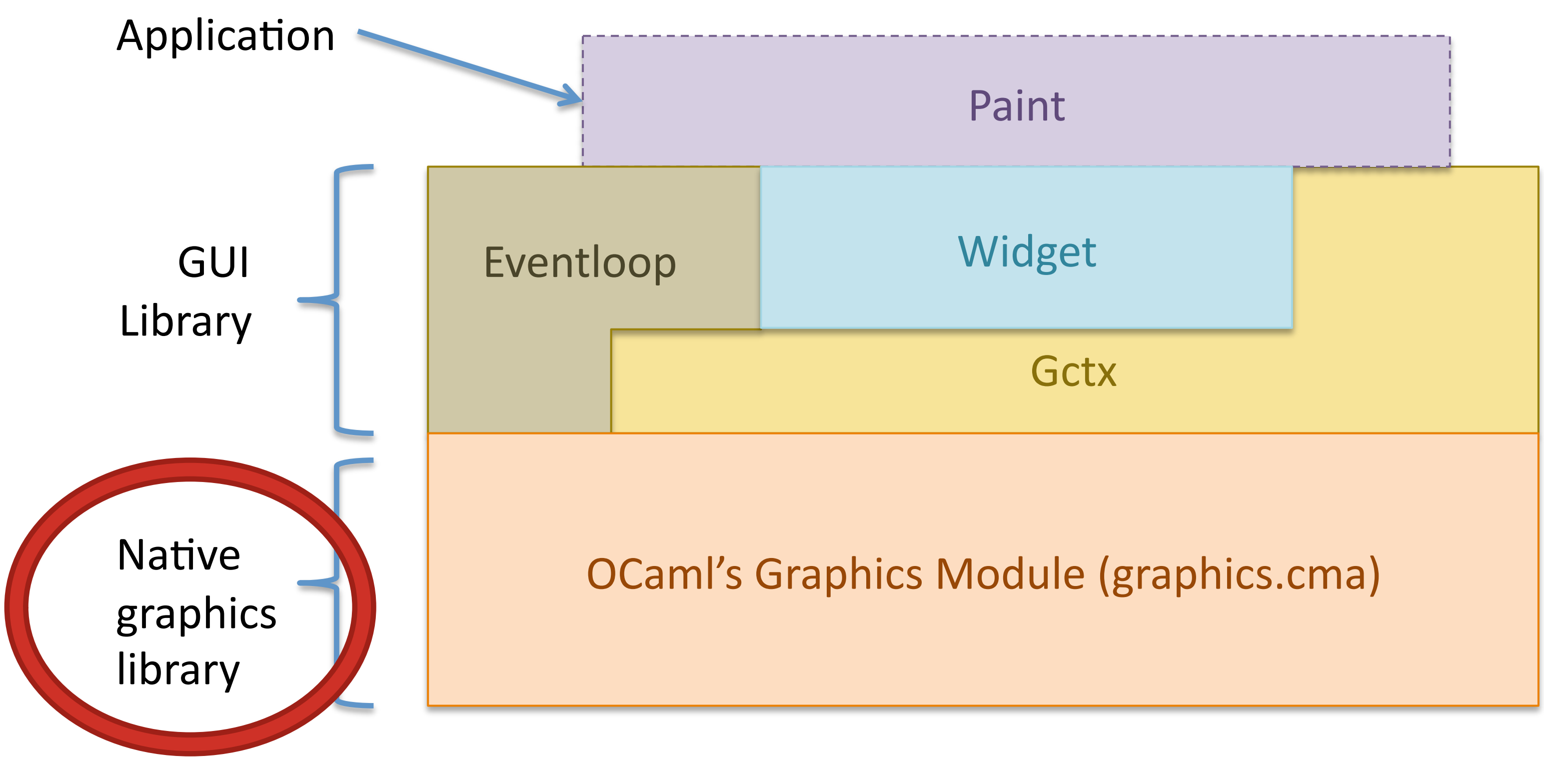
Announcements

- HW06: GUI programming available now
  - Due: Friday, *March 1st*

Designing a GUI library



Project Architecture



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

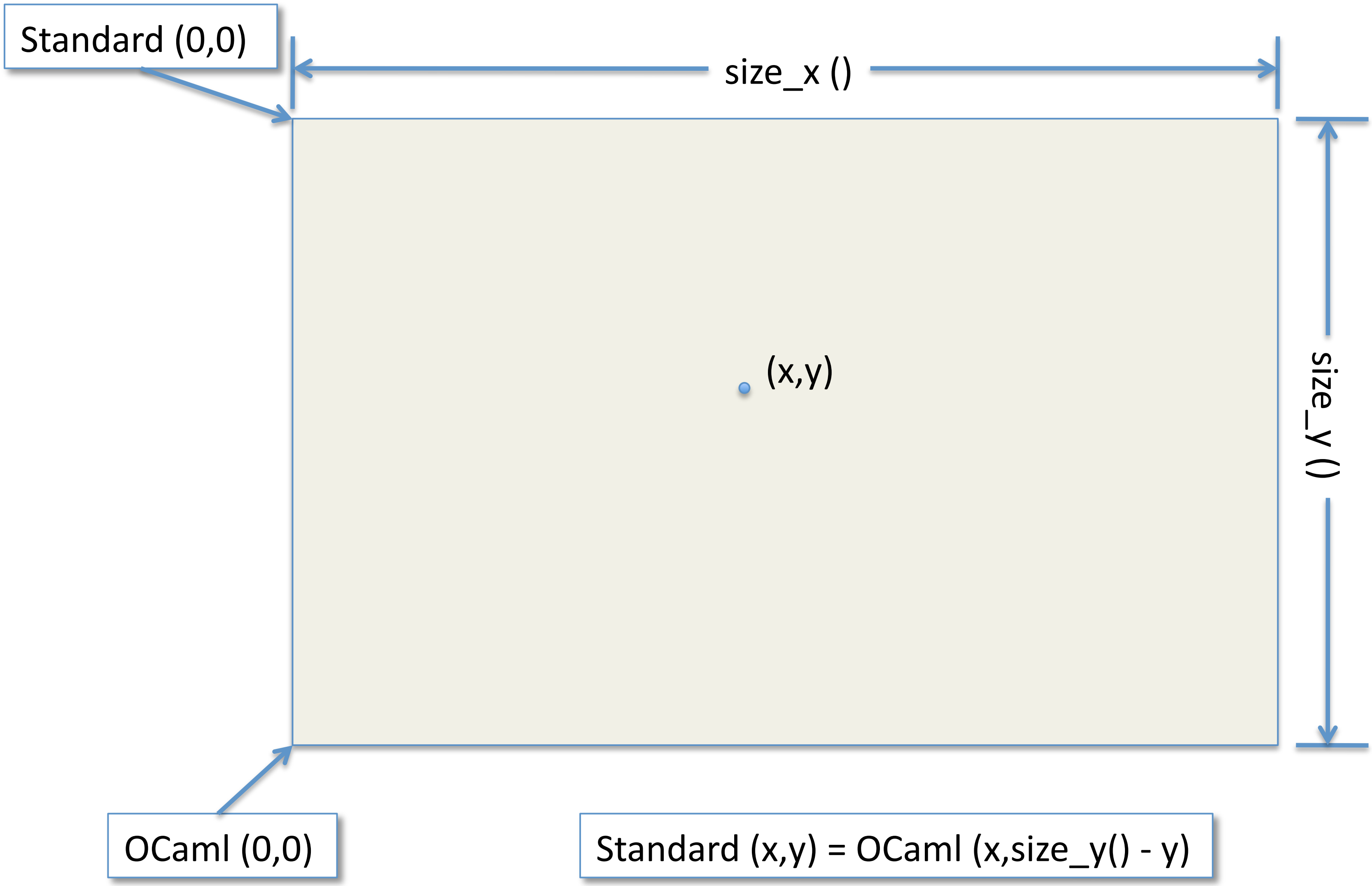
Designing a GUI library

- OCaml's Graphics library* provides very *simple* primitives for:
  - Creating a window
  - Drawing various shapes: points, lines, text, rectangles, circles, etc.
  - Getting the mouse position, whether the mouse button is pressed, what key is pressed, etc.

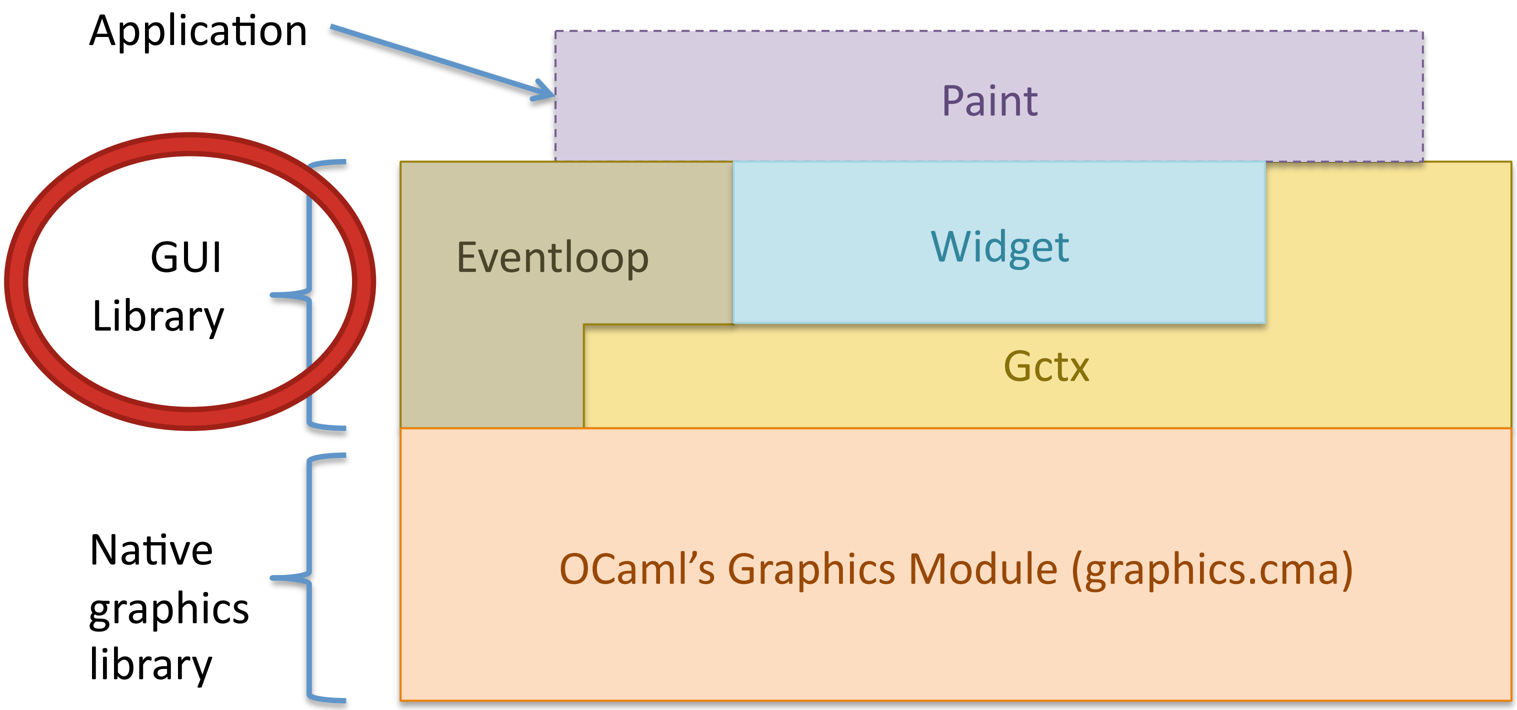
  - See: http://www.seas.upenn.edu/~cis120/current/ocaml-3.12-manual/libref/Graphics.html
- How do we go from that to a functioning, reusable GUI library?

*Pragmatic note: when compiling a program that uses the Graphics module, add graphics.cmxa (for native compilation) or graphics.cma (for bytecode compilation) to OCaml Build Flags under the Projects>Properties dialog in Eclipse.

OCaml vs. *Standard* Coordinates



Project Architecture



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

GUI terminology – Widget*

- Basic element of GUIs : buttons, checkboxes, windows, textboxes, canvases, scrollbars, labels
- All have a position on the screen and know how to display themselves
- May be composed of other widgets (for layout)
- Widgets are often modeled by objects
  - They often have hidden state (string on the button, whether the checkbox is checked)
  - They need functions that can modify that state

*Each GUI library uses its own naming convention for what we call "Widget". Java's Swing calls them "Components"; iOS UIKit calls them "UIViews"; WINAPI, GTK+, X11's widgets, etc….

GUI terminology - Eventloop

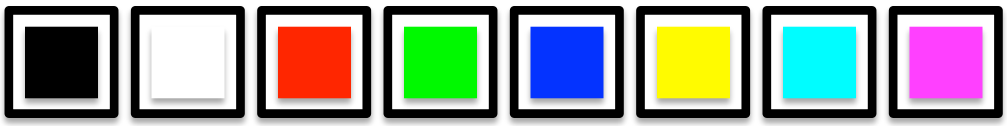
- Main loop of any GUI application

```
let run (w:Widget.t) : unit =
  Graphics.open_graph "";              (* open a new window *)
  Graphics.auto_synchronize false;

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

- Takes "top-level" widget w as argument. That widget *contains* all others in the application.

Container Widgets for layout



```
let color_toolbar : Widget.t = 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]
```

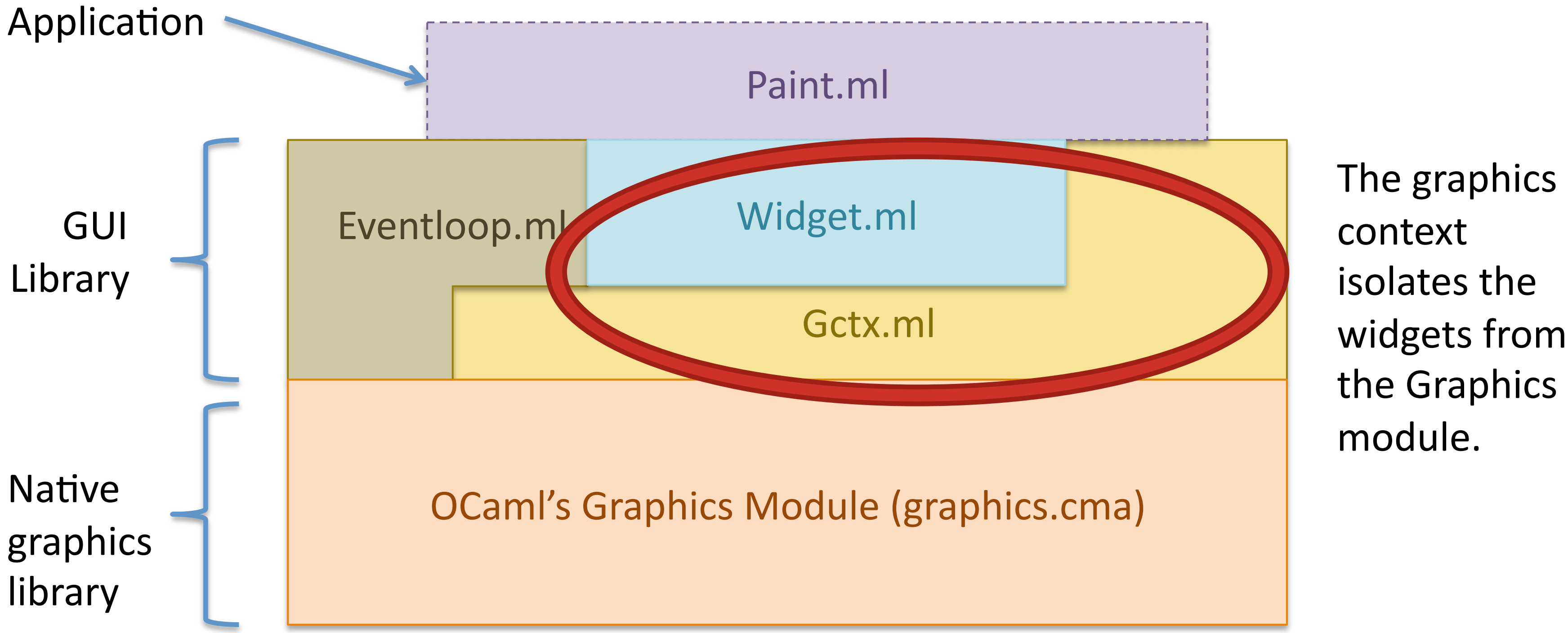
paint.ml

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

- Challenge: How can we make it so that the functions that draw widgets (buttons, check boxes, text, etc.) in different places on the window are location independent?

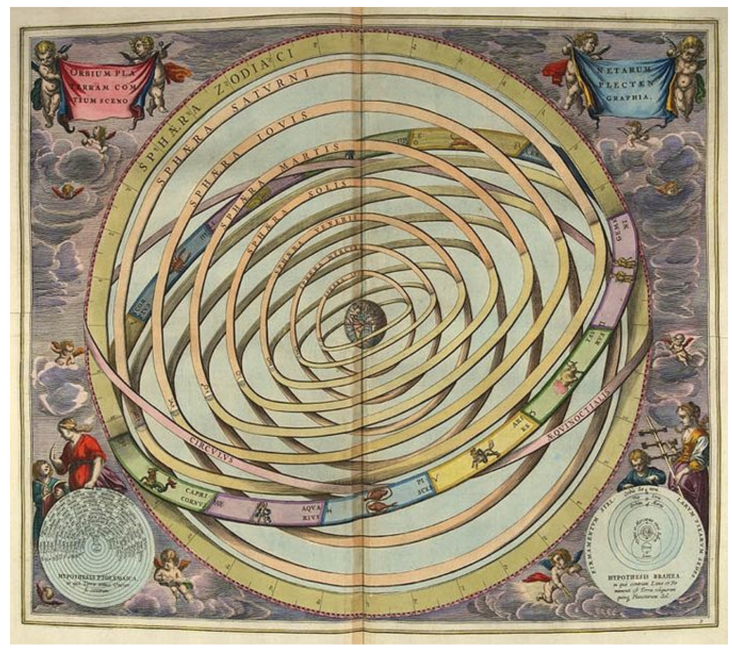
Challenge: Widget Layout

- Widgets are "things drawn on the screen". How to make them location independent?
- Idea: Use a graphics context to make drawing primitives *relative* to the widget's local coordinates.



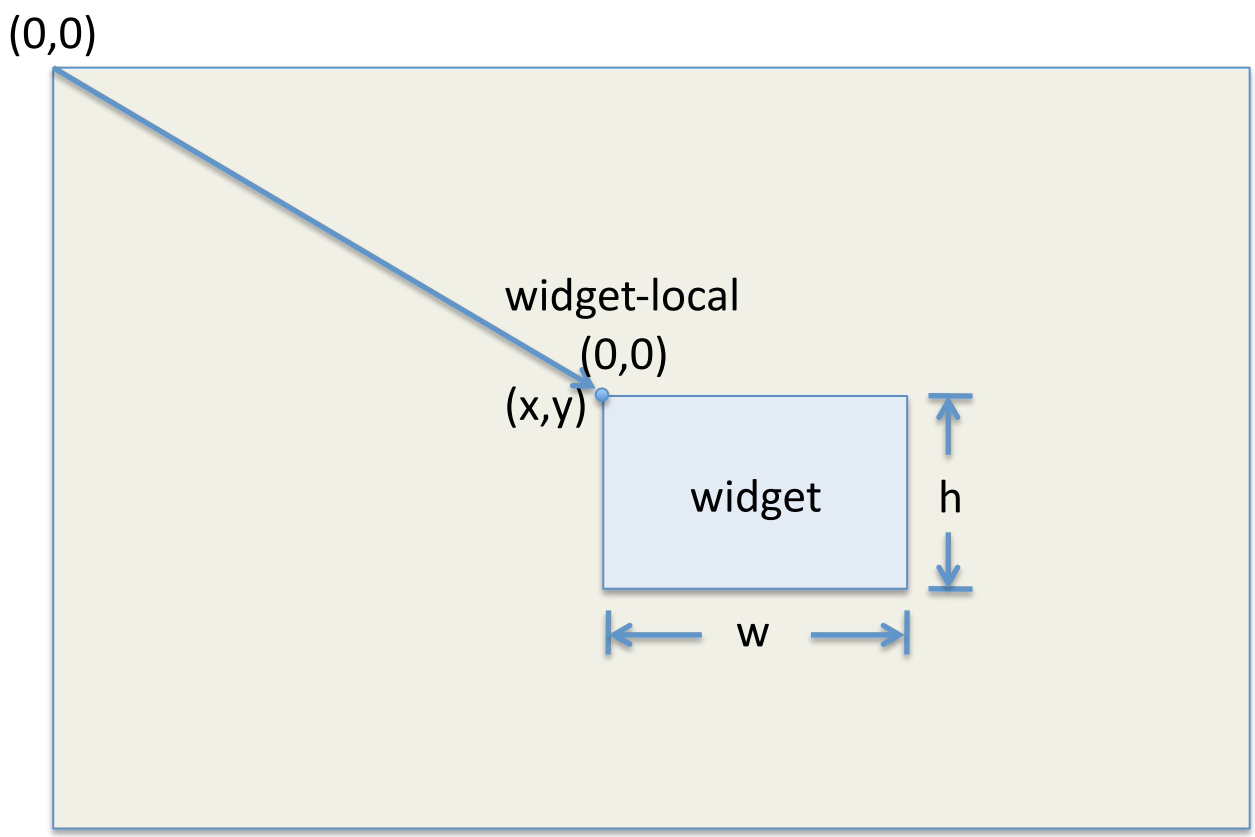
GUI terminology – Graphics Context

- Wrapper for OCaml Graphics library, putting operations "in context"
- Aggregates information about the way things are drawn, such as the foreground color or line width
- Translates coordinates of drawing commands
  - Flips between OCaml and "Standard coordinates" so origin is top-left
  - Translates coordinates so all widgets can pretend that they are at the origin



Graphics Contexts

Absolute (Flipped OCaml)



A graphics context Gctx.t represents a position within the window, relative to which the widget-local coordinates should be interpreted. We can add additional context information that should be "inherited" by children widgets (e.g. current pen color).

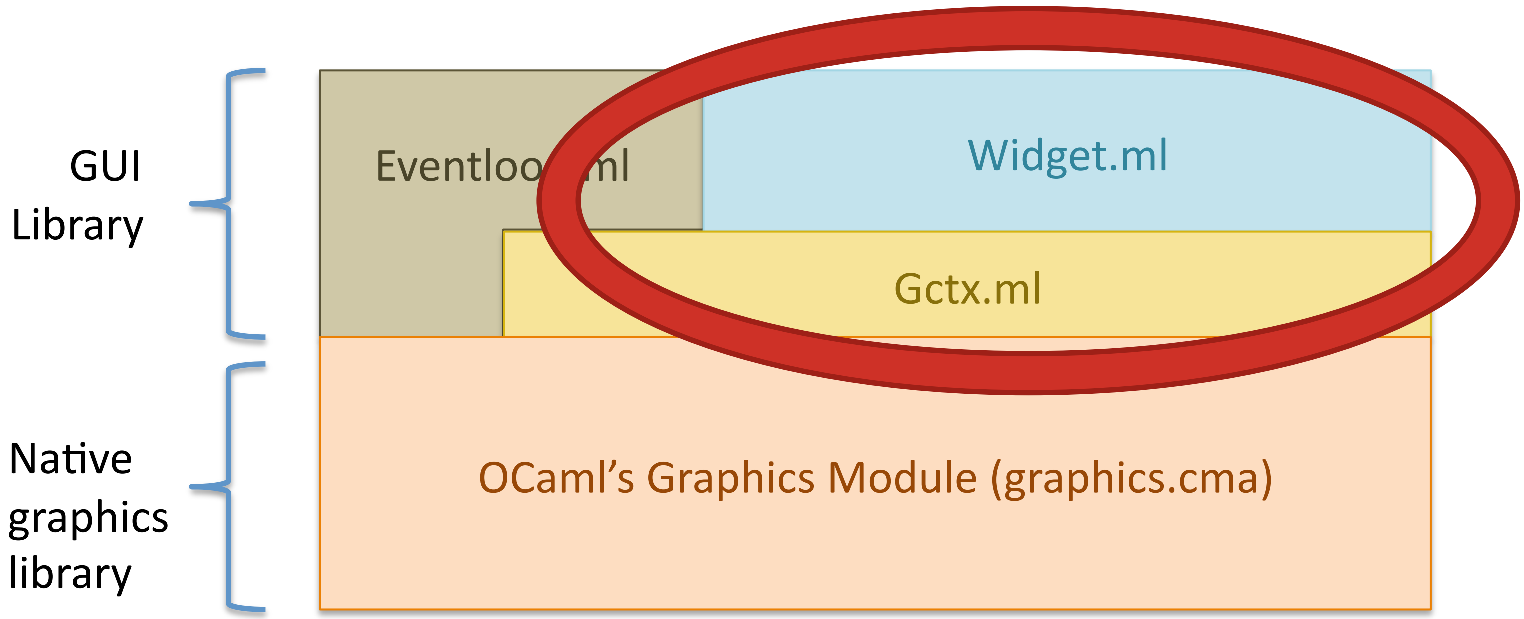
Module: Gctx

Contextualizes graphics drawing operations

Module: Widgets

Building blocks of GUI applications

GUI Library Architecture



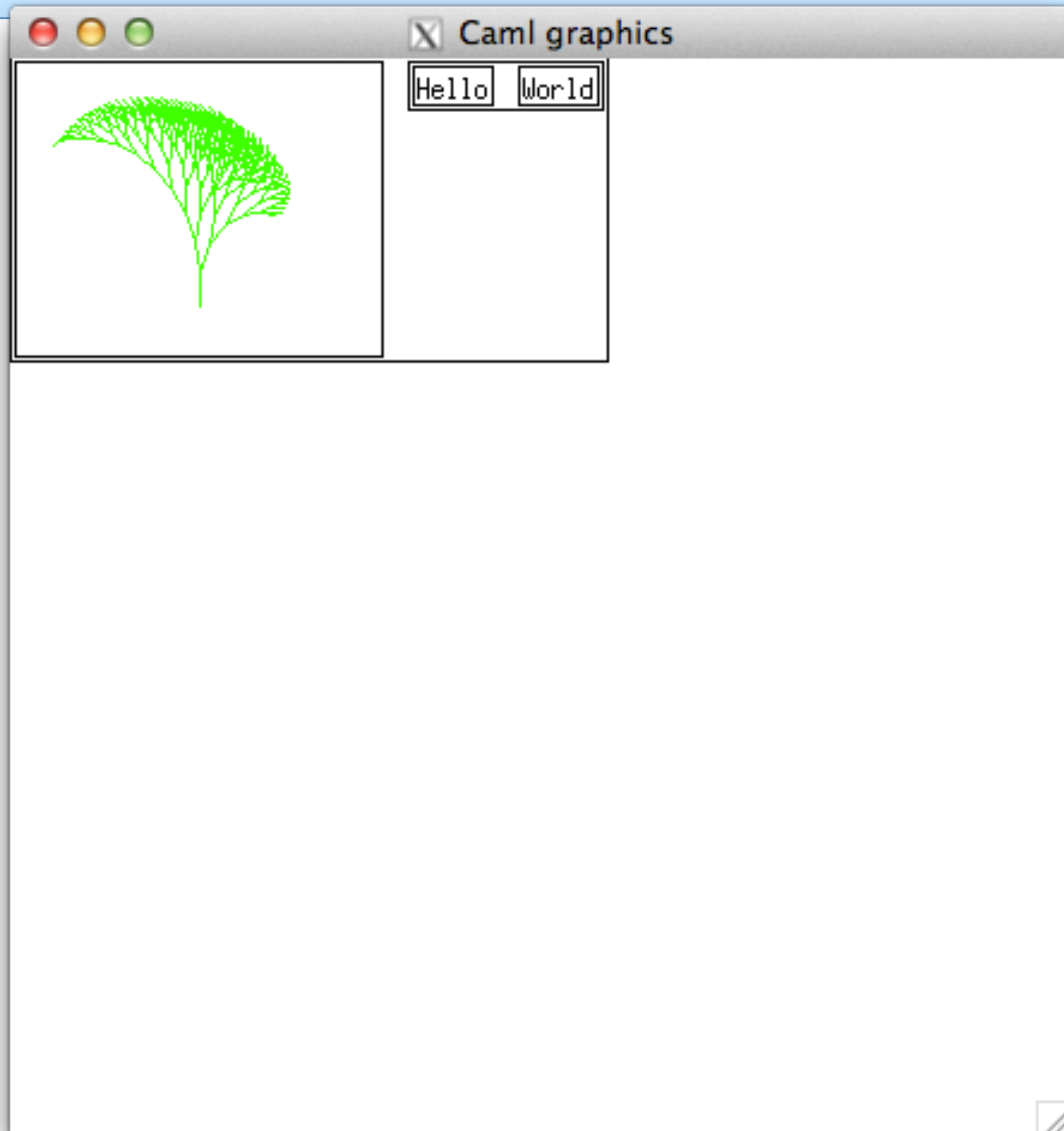
Simple Widgets

```
(* An interface for simple GUI widgets *)
type t = {
   repaint : Gctx.t -> unit;
   size    : Gctx.t -> (int * int)
}
```

- You can ask a simple widget to repaint itself.
- You can ask a simple widget to tell you its size.

- Both operations are relative to a graphics context

swdemo.ml



Widget Examples

simpleWidget.ml

```
(* Display a string on the screen. *)
let label (s:string) : t =
{
  repaint = (fun (g:Gctx.t) -> Gctx.draw_string g s);
  size    = (fun (g:Gctx.t) -> Gctx.text_size g s)
}
```

simpleWidget.ml

```
(* A region of empty space. *)
let space ((w,h):int*int) : t =
{
  repaint = (fun (_:Gctx.t) -> ());
  size    = (fun (_:Gctx.t) -> (w,h))
}
```

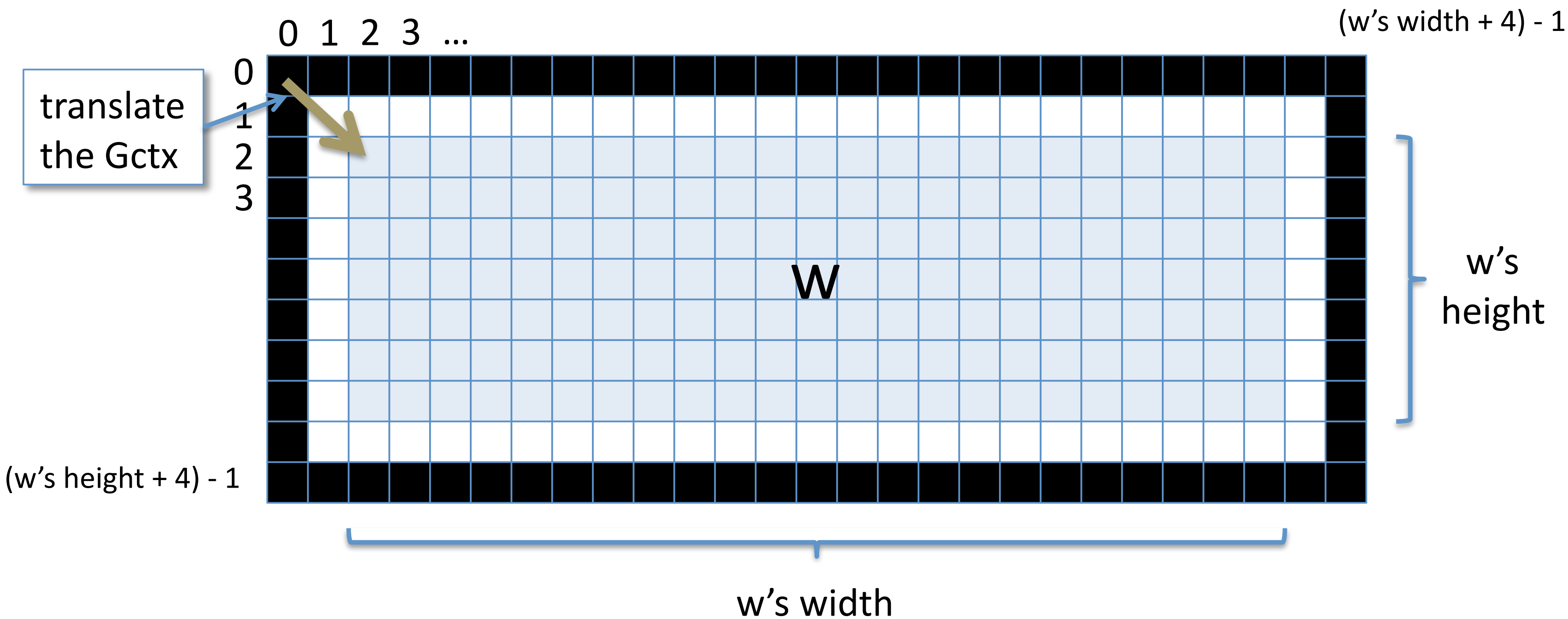
The canvas Widget

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

simpleWidget.ml

```
(* expose the graphics context as a widget *)
let canvas ((w,h):int*int)(repaint: Gctx.t -> unit) : t =
  {
    repaint = repaint;
    size    = (fun (_:Gctx.t) -> (w,h))
  }
```

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

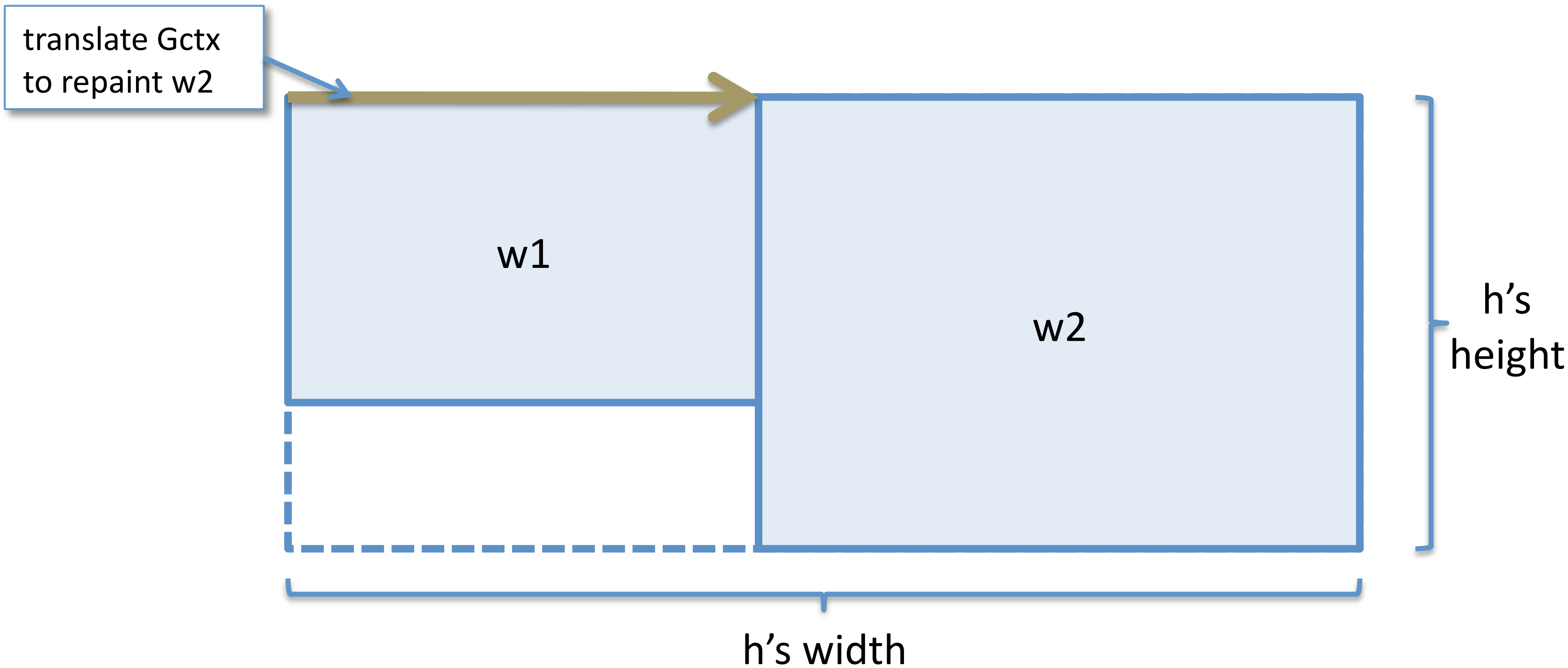
```
let border (w:t):t =
{
  repaint = (fun (g:Gctx.t) ->
    let (width,height) = w.size g in
    let x = width + 3 in
    let y = height + 3 in
    Gctx.draw_line g (0,0) (x,0);
    Gctx.draw_line g (0,0) (0,y);
    Gctx.draw_line g (x,0) (x,y);
    Gctx.draw_line g (0,y) (x,y);
    let g = Gctx.translate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

  size = (fun (g:Gctx.t) ->
    let (width,height) = w.size g 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 the right widget
- Size is the sum of their widths and max of their heights

The Widget Hierarchy

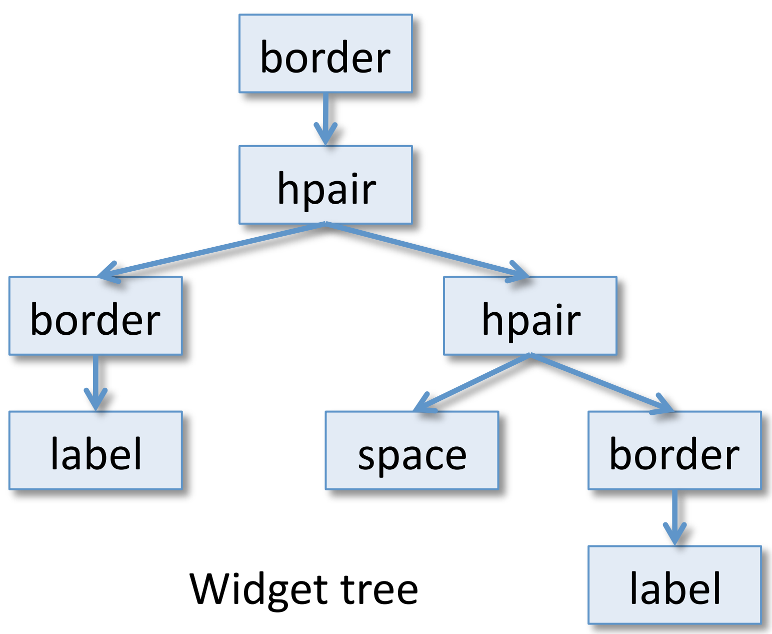
- Widget instances form a tree*:
  - Leaf widgets – don't contain any children
    - label, space, and canvas widgets are leaves
  - Container widgets – are "wrappers" for their children
    - border and hpair widgets are containers
- Build container widgets by passing in their children as arguments to their "constructor" functions
  - e.g. `let b = border w in …`
    `let h = hpair b1 b2 in …`
- The repaint method of the root widget initiates all the drawing and layout for the whole window

*If you draw the state of the abstract machine for a widget program, the tree will be visible in the heap – the saved stack of the "repaint" function for a container widget will contain references to its children.

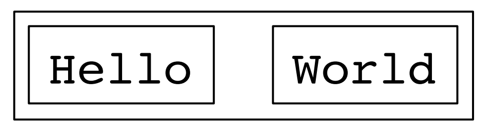
Widget Hierarchy Pictorially

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

Demo: swdemo.ml