Programming Languages and Techniques (CIS120)

Lecture 18

March 3, 2014

GUI Design II: Layout

Are you going somewhere warm for Spring Break?

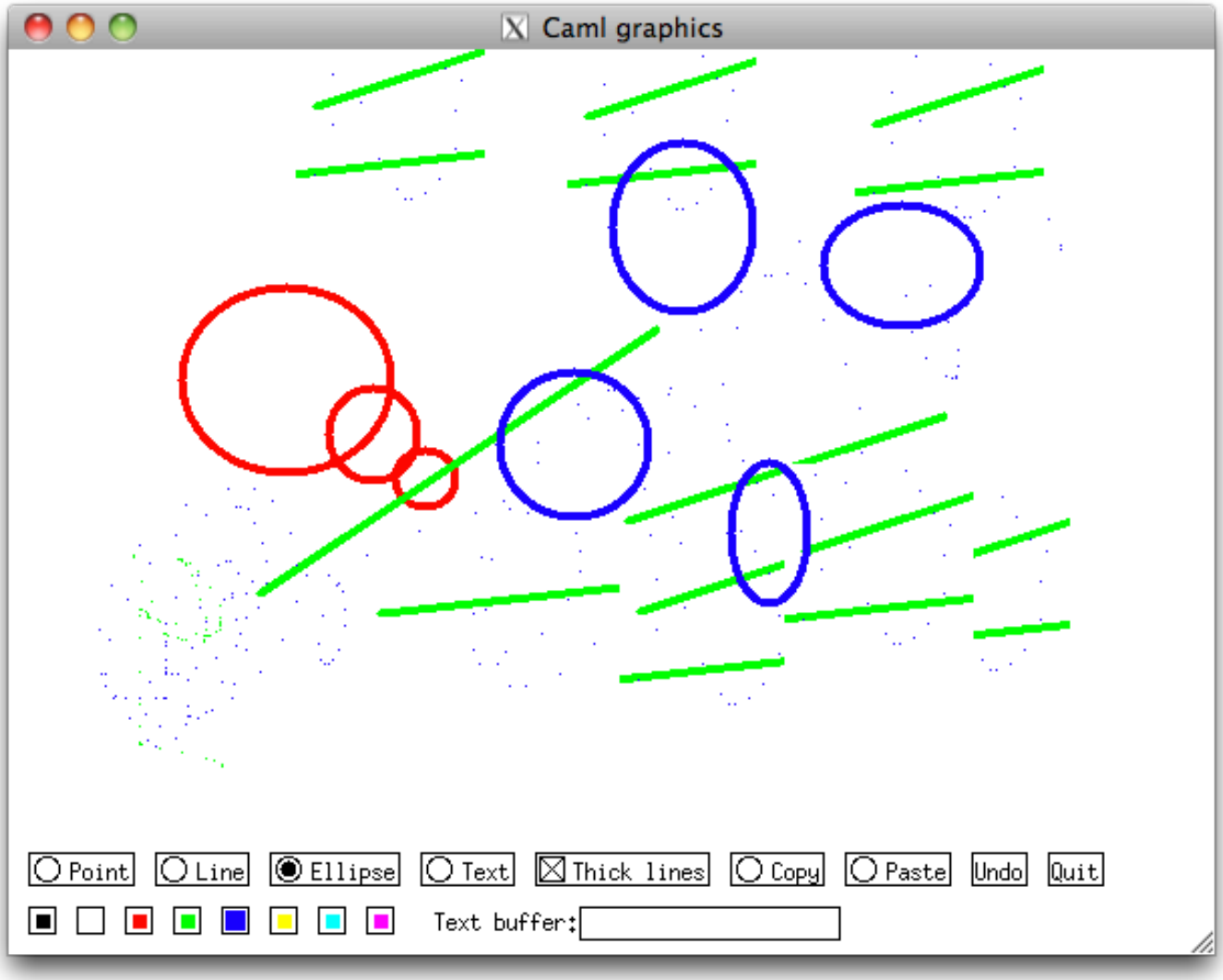
1. Yes!
2. No, the cold never bothered me anyway
3. Philly will be 70 degrees next week, right?!
4. Spring break?

Announcements

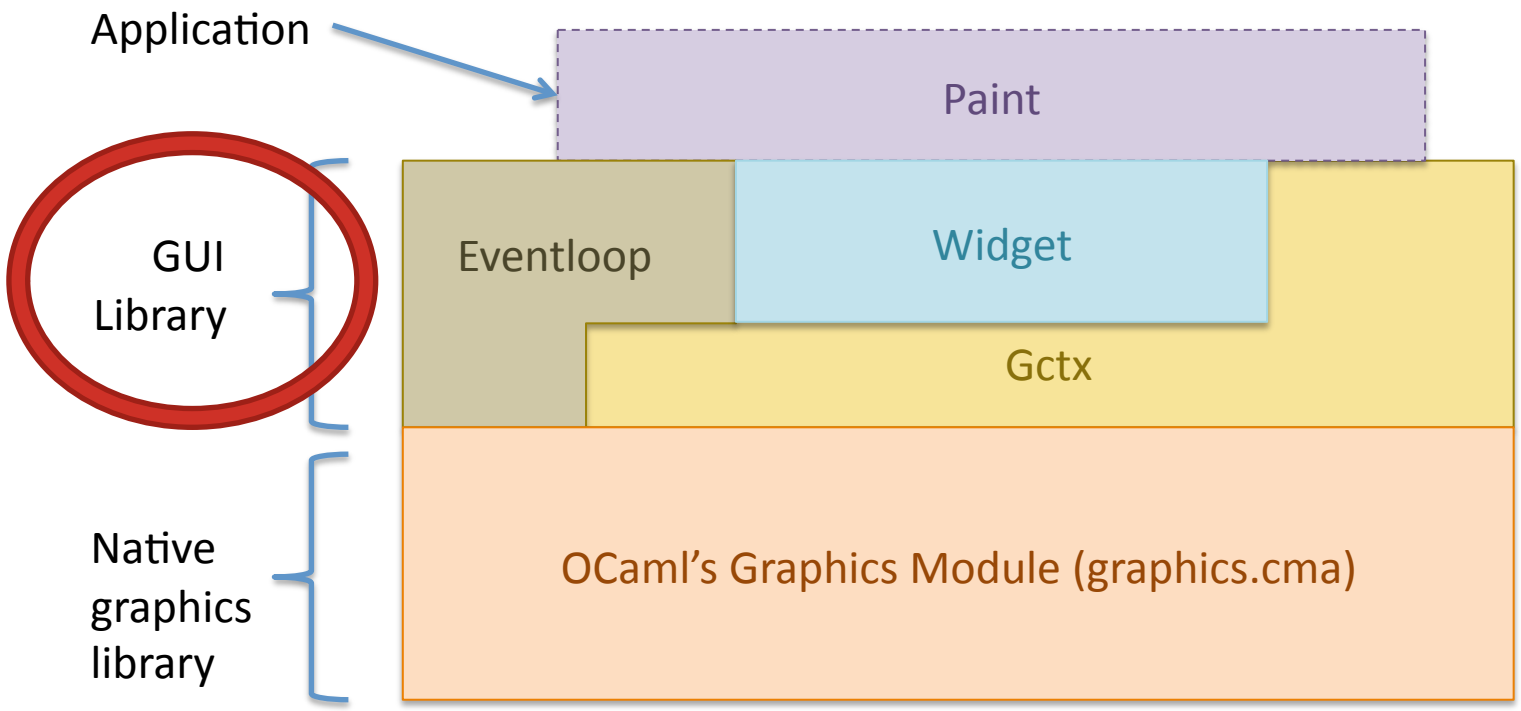
- HW 6 due FRIDAY at midnight
- Read Chapter 18 of the lecture notes
- Read over the provided code *before* getting started

- First Java assignment will be available *after* Spring Break
- Due Tuesday, March 25th

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.

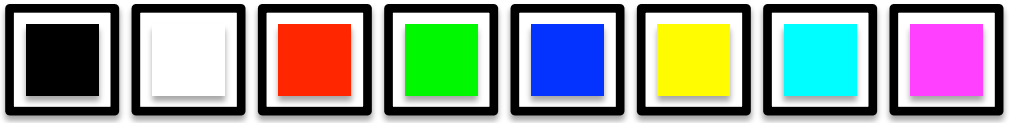
GUI terminology – Widget*

- 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….

Container Widgets for layout



```
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]
```

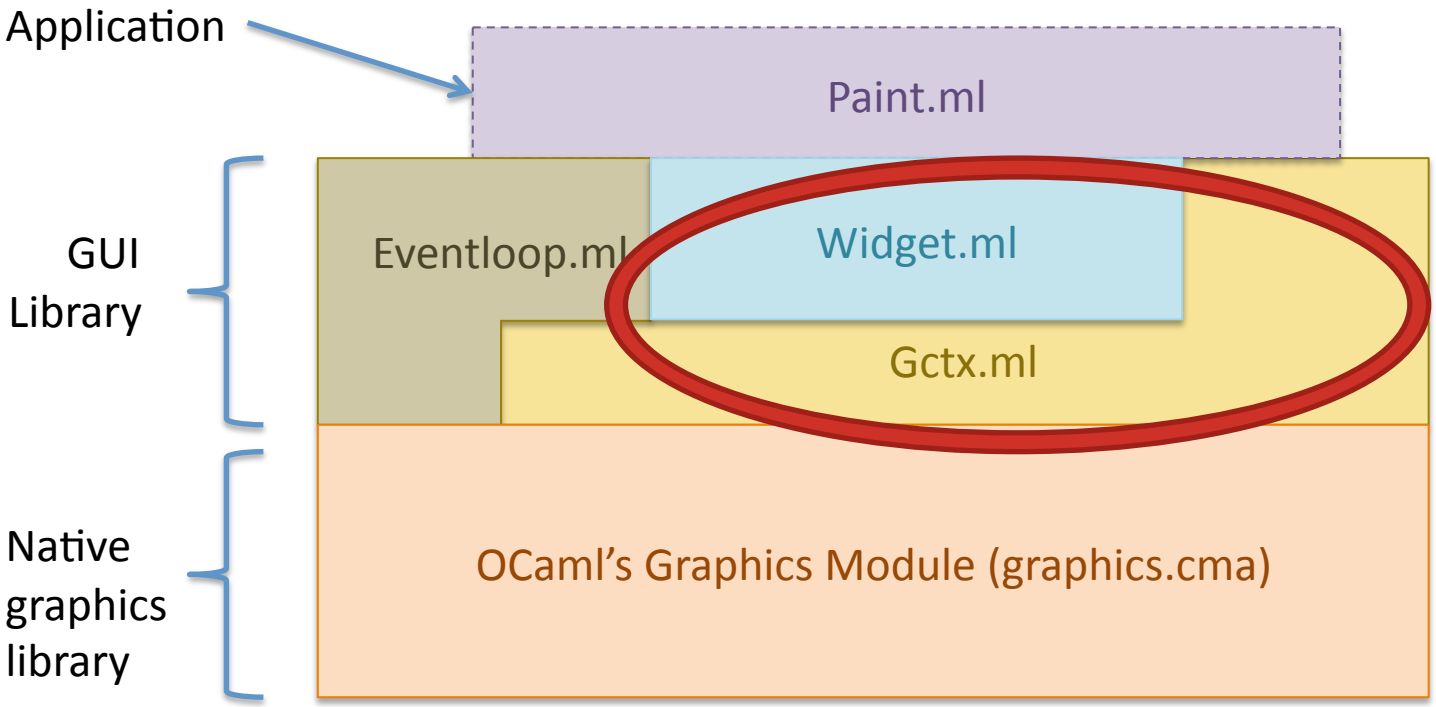
paint.ml

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

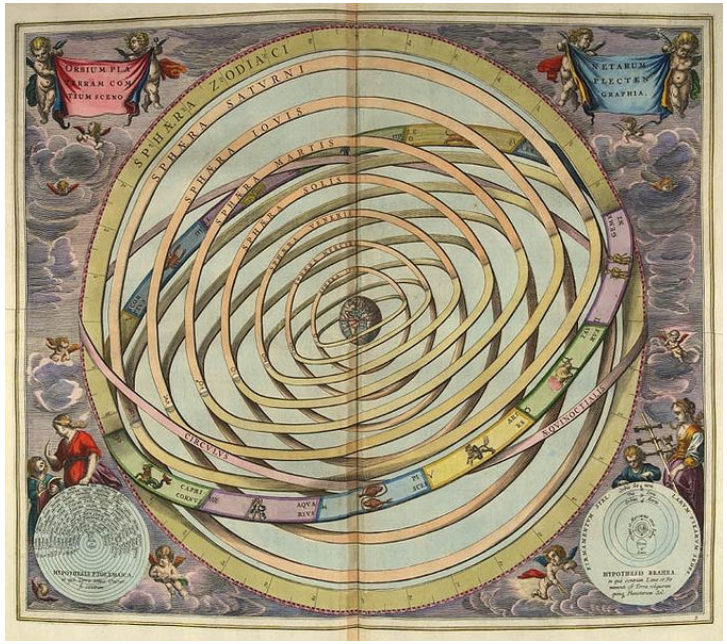
- Widgets are "things drawn on the screen". How to make them location independent?
- Idea: Use a graphics context to make drawing primitives *relative* to the widget's local coordinates.



The graphics context isolates the widgets from the Graphics module.

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



Widgets

Building blocks of GUI applications

Simple Widgets

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

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

- Both operations use a graphics context

Simple widget examples

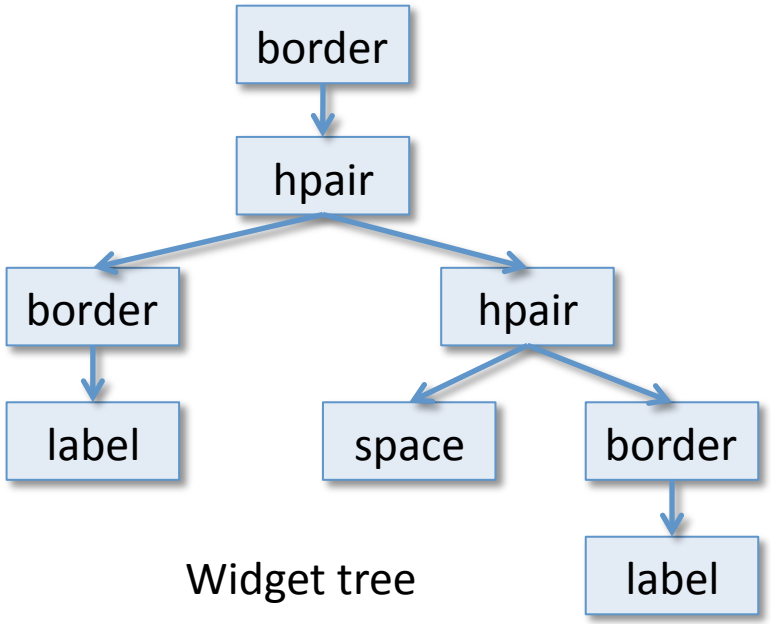
- Leaf widgets --- parts of the screen
  - *label*: piece of text on the screen
  - *canvas*: part of the screen that can be drawn on
  - *space*: blank part of the screen

- Container widgets – arrange other widgets
  - *border*: draw a border around another widget
  - *hpair*: put two widgets side-by-side

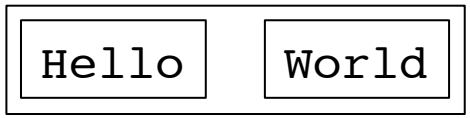
Widget Hierarchy Pictorially

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

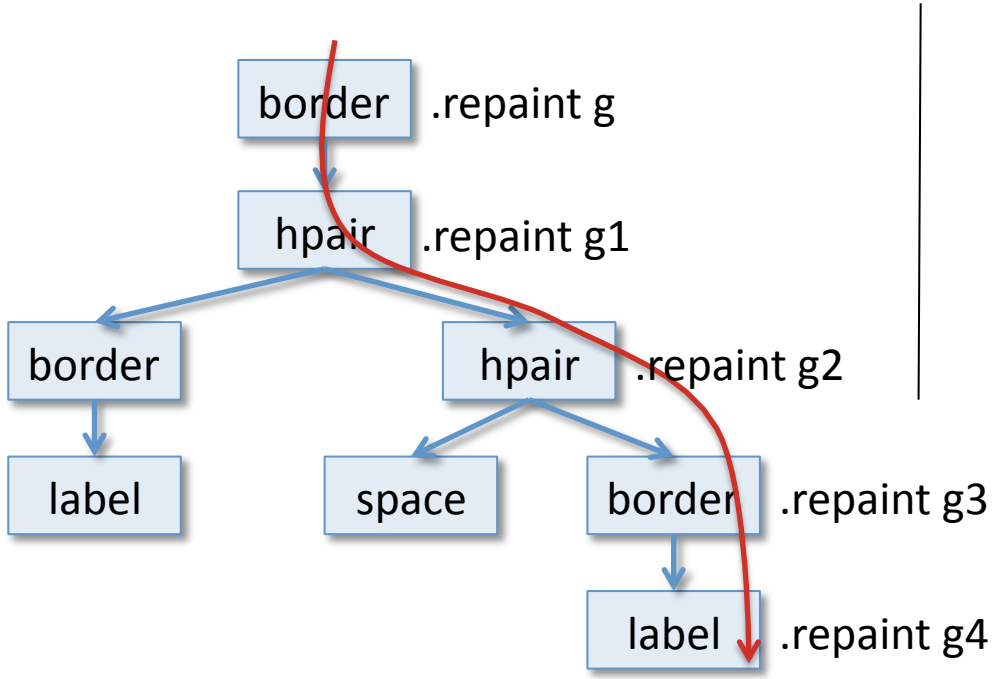
The Widget Hierarchy

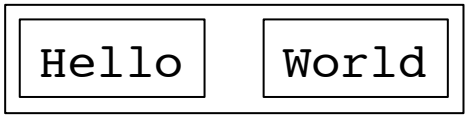
- Widget instances form a tree*:
  - Leaf widgets – don't contain any children
    - label, space, and canvas widgets are leaves
  - Container widgets – are "wrappers" for their children
    - border and hpair widgets are containers
- Build container widgets by passing in their children as arguments 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.

Drawing: Containers

Container widgets propagate repaint commands to their children:





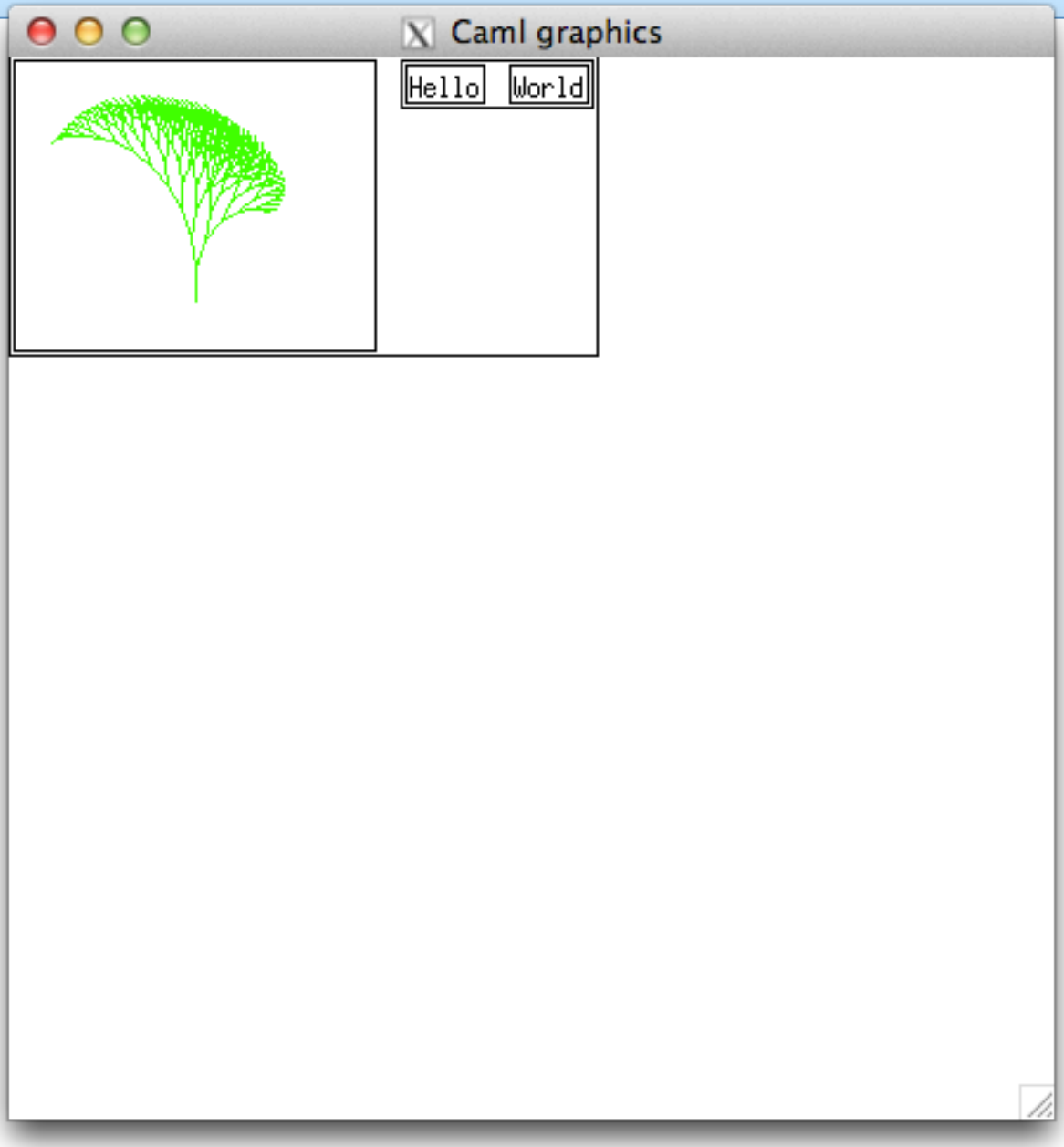
Widget tree

On the screen

swdemo.ml

Building blocks of GUI applications

swdemo.ml



Simple Widgets: implementation

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

- All widgets have a “constructor function” that returns a value of this type

Widget Examples

simpleWidget.ml

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

simpleWidget.ml

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

The canvas Widget

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

simpleWidget.ml

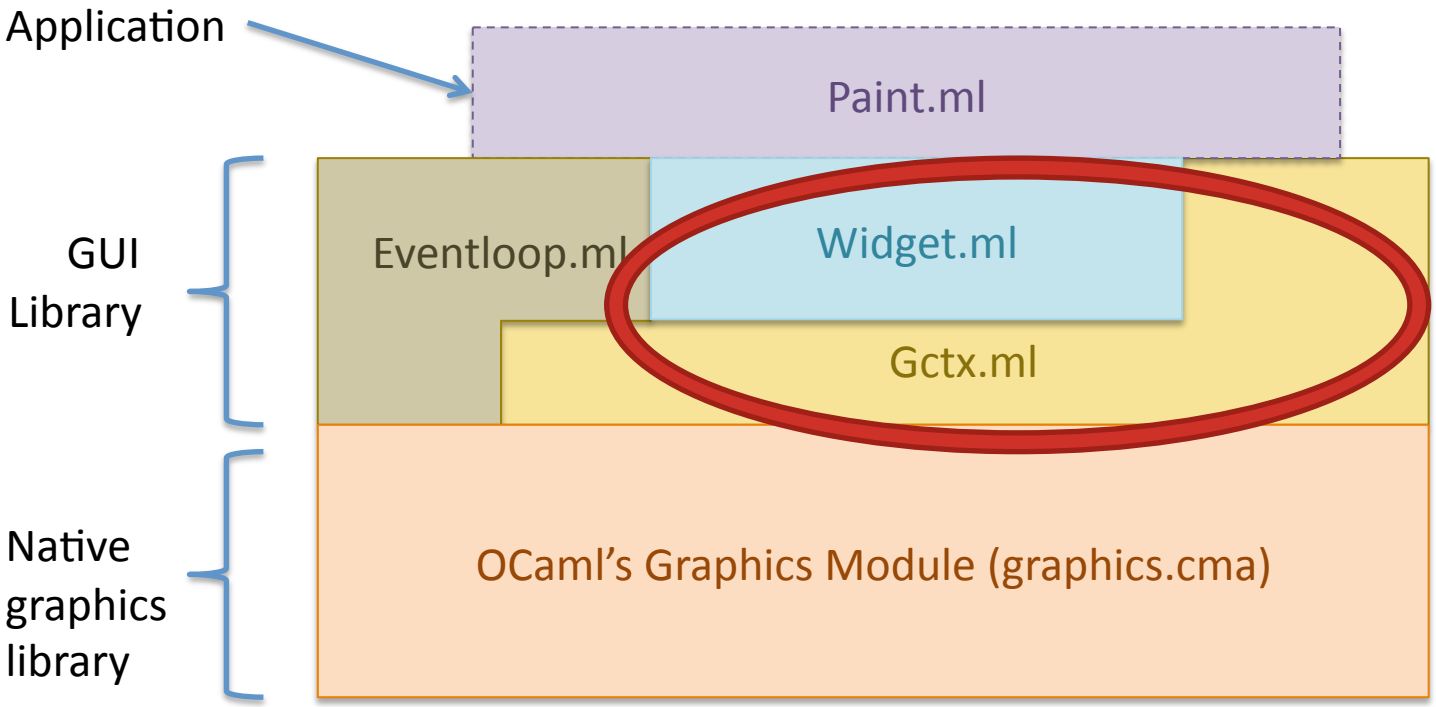
```
(* expose the graphics context as a widget *)
let canvas ((w,h):int*int)(repaint:gctx -> unit): widget =
  {
    repaint = repaint;
    size    = (fun (_:gctx) -> (w,h))
  }
```

Graphics Contexts

Why do we need a layer between the Graphics library and the Widget library?

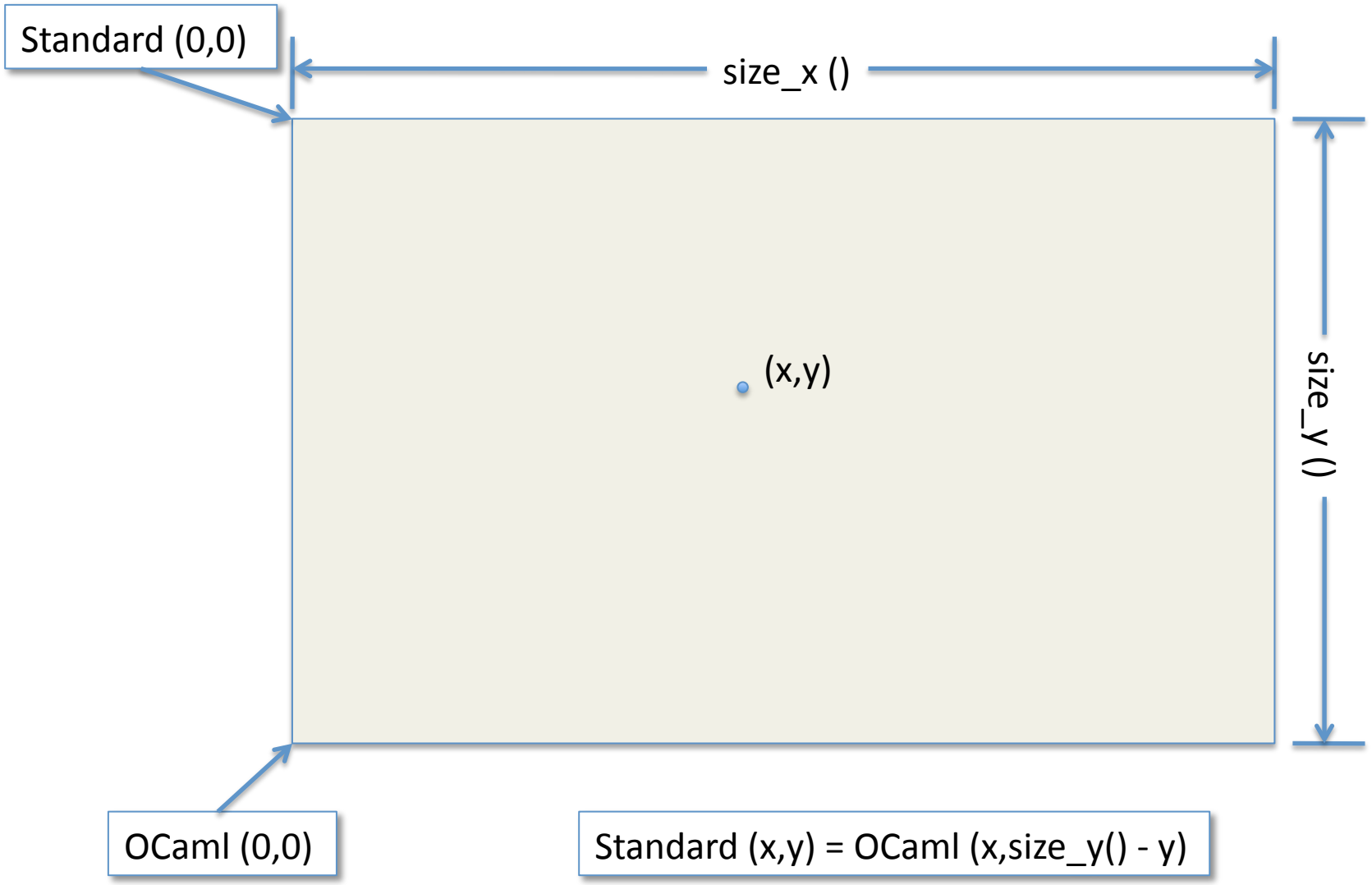
Challenge: Widget Layout

- 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.

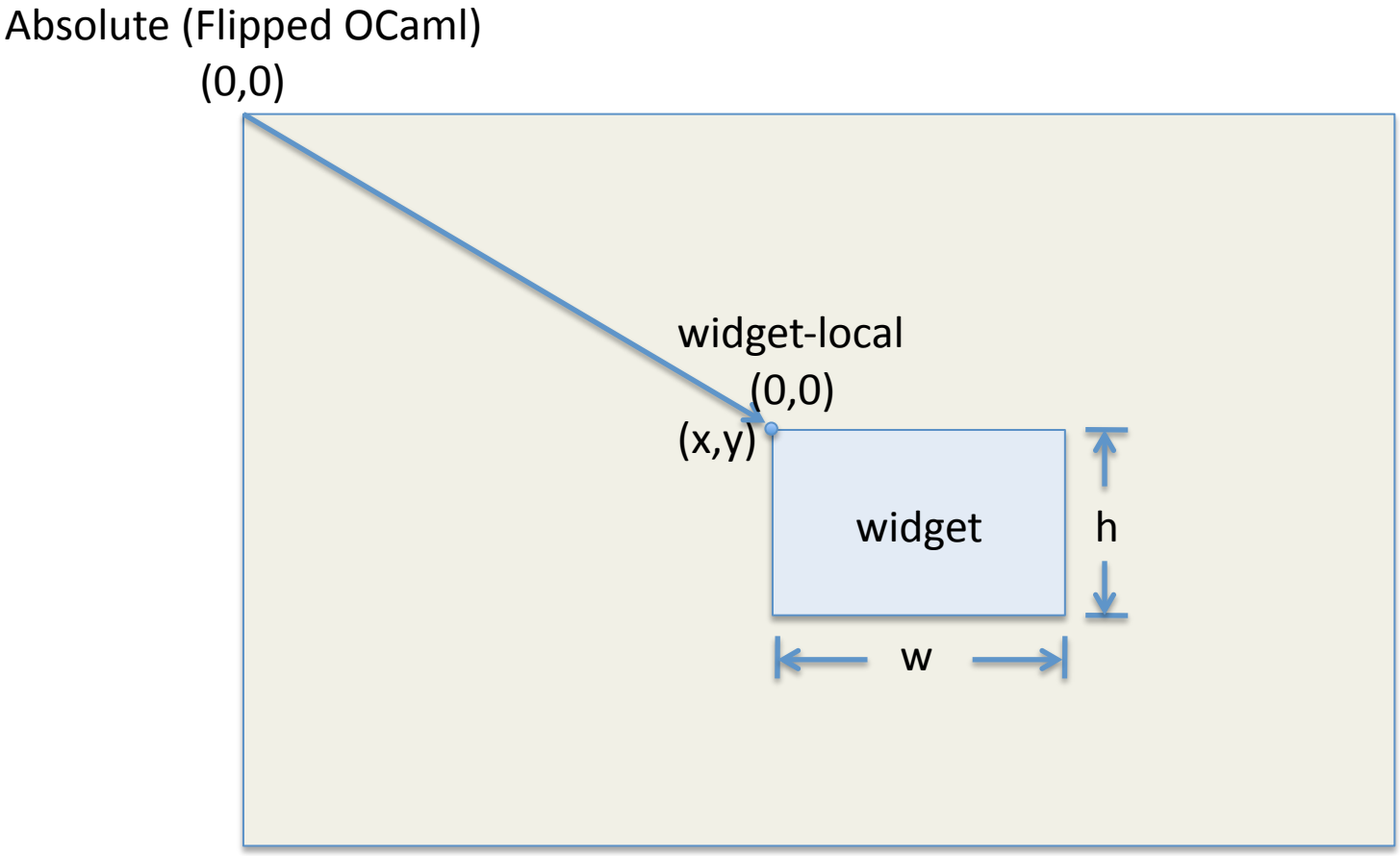


The graphics context isolates the widgets from the Graphics module.

OCaml vs. *Standard* Coordinates



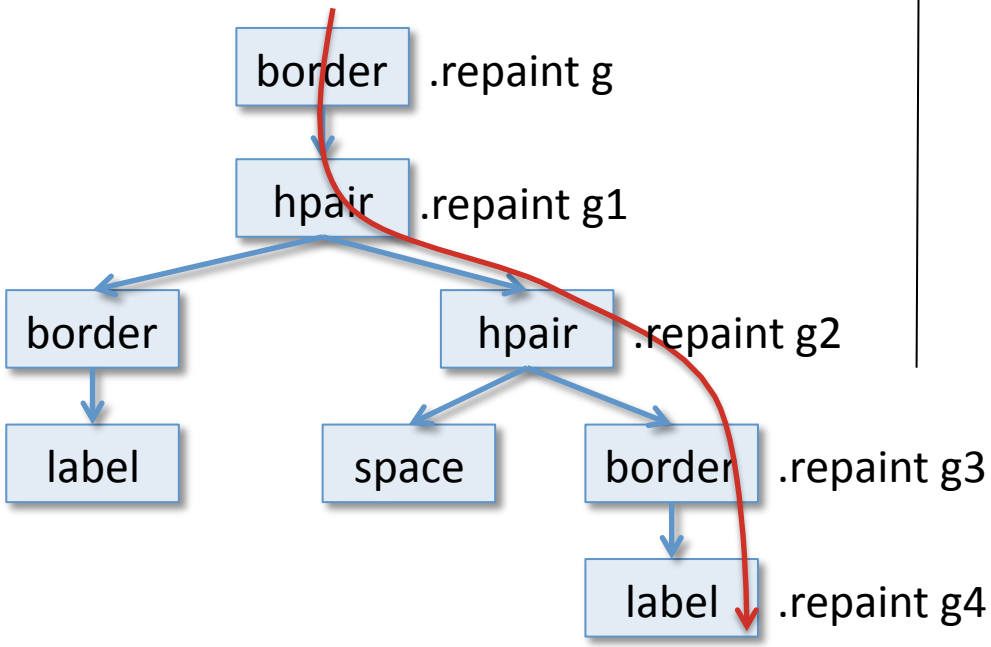
Graphics Contexts

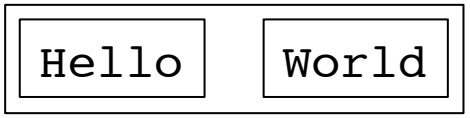


A graphics context gctx represents a position within the window, relative to which the widget-local coordinates should be interpreted. We can add additional context information that should be "inherited" by children widgets (e.g. current pen color).

Drawing: Containers

Container widgets propagate repaint commands to their children:



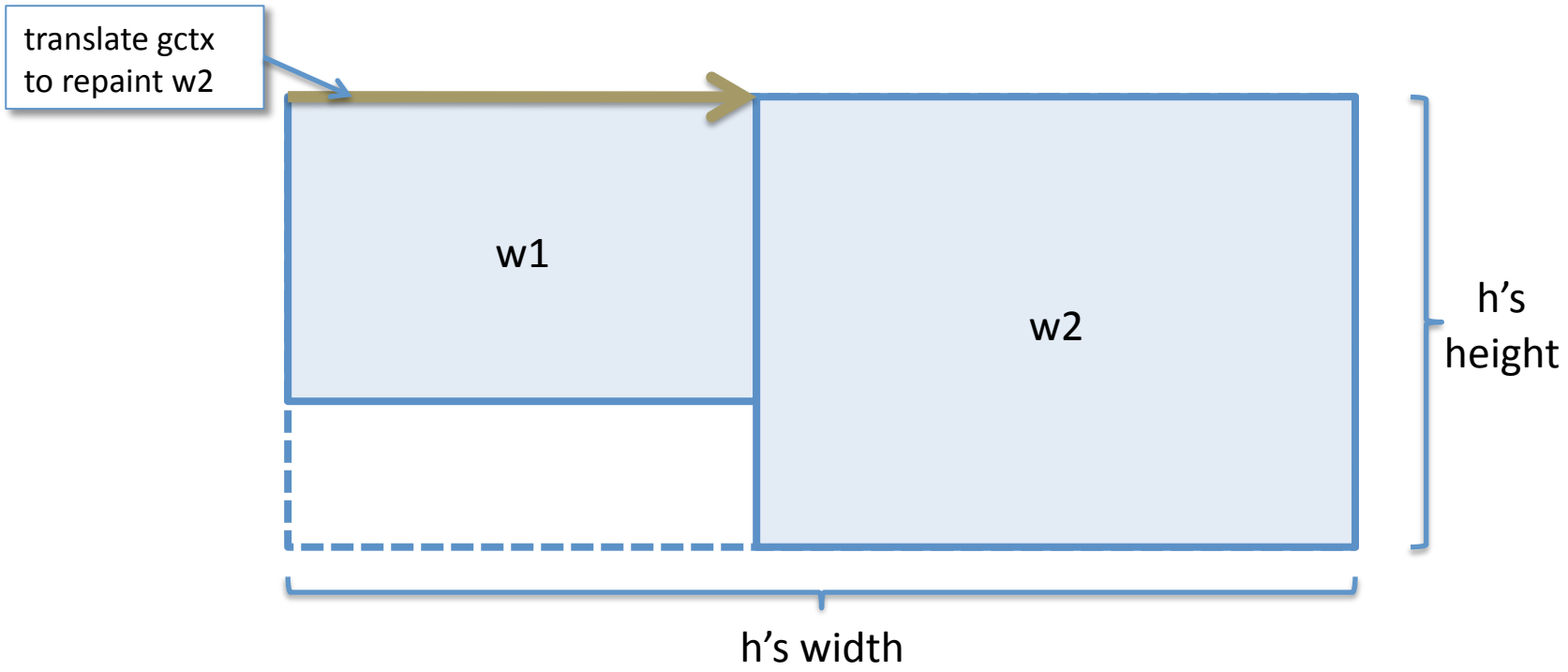


Widget tree

```
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

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

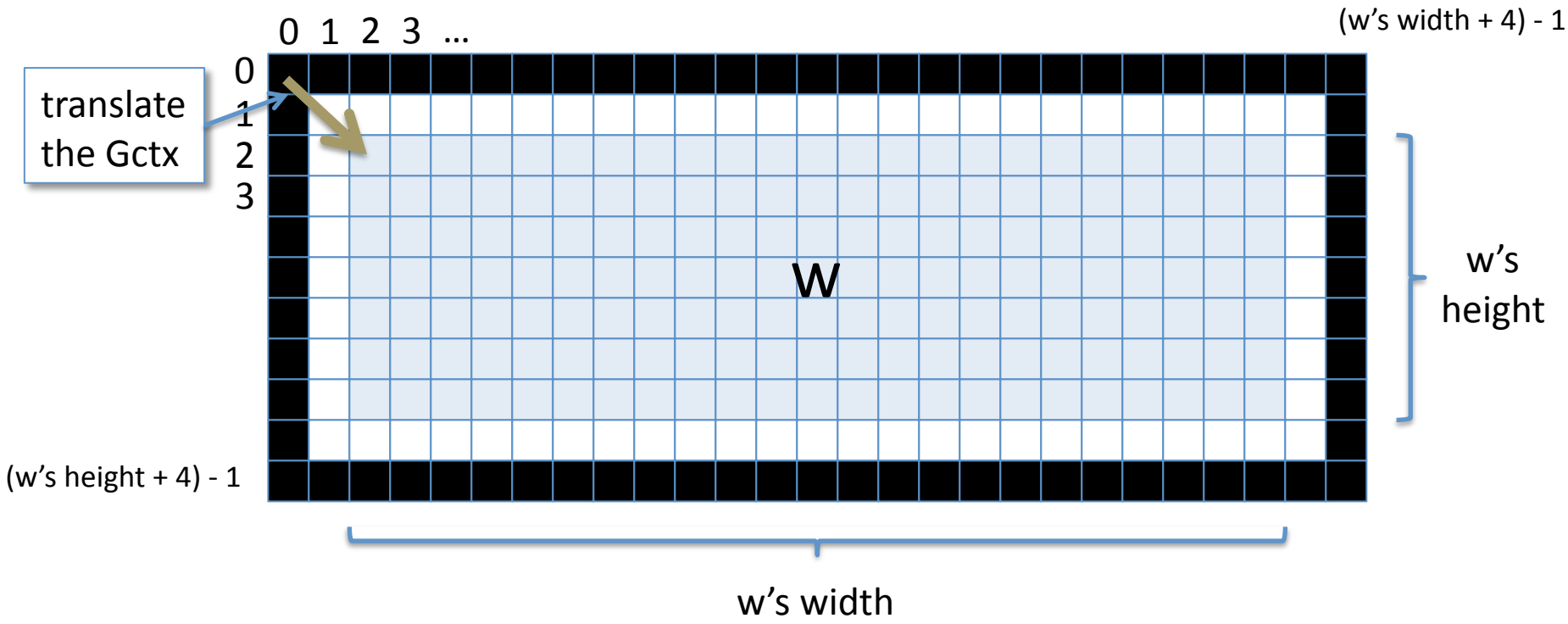
The Border Widget

simpleWidget.ml

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

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

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

Draw the border

Display the interior