Programming Languages and Techniques (CIS120)

Lecture 19

February 26, 2016

GUI Library Design

Chapter 18

Are you here today?

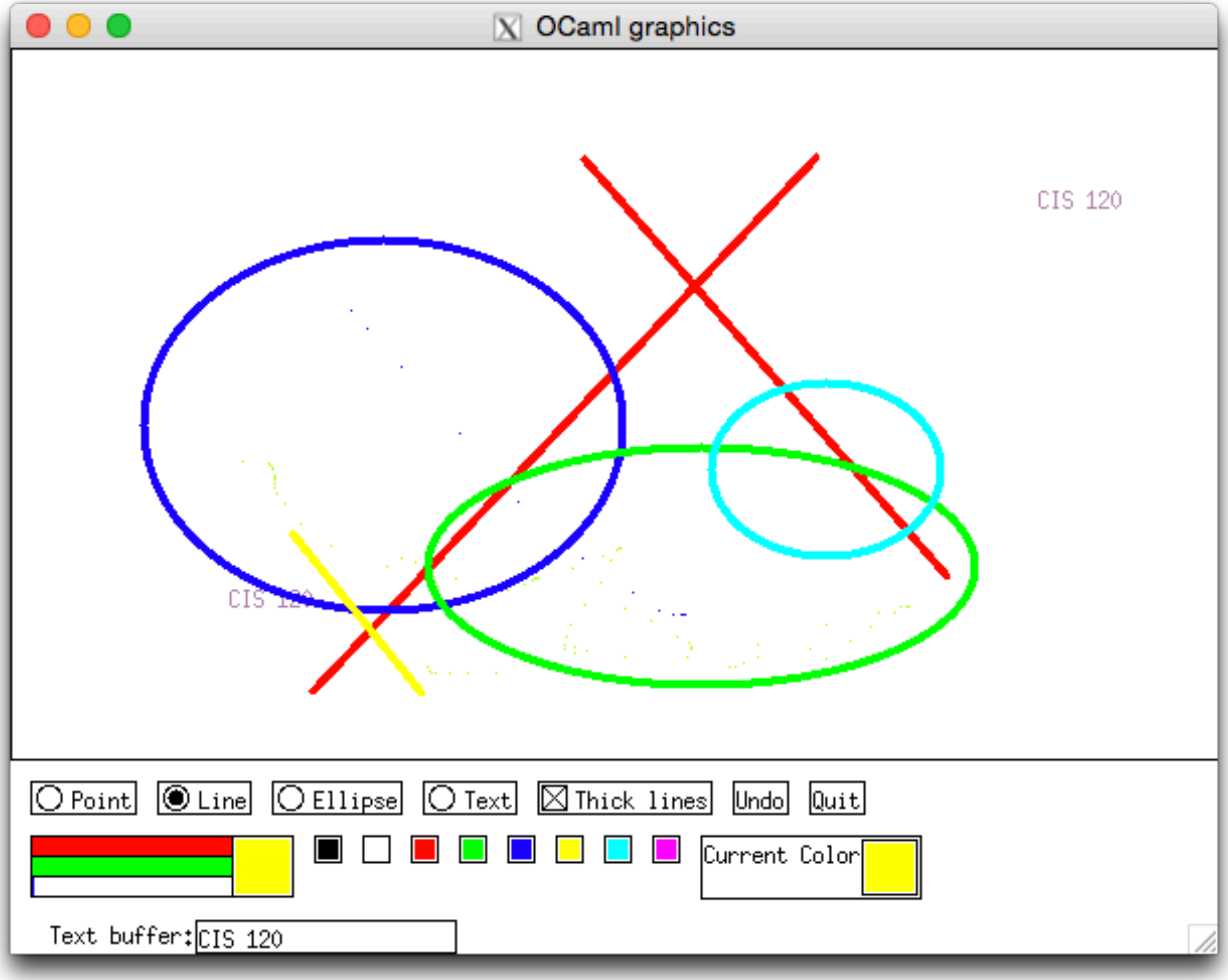
1. Yes

Announcements

HW05: GUI programming is available

- Due: THURSDAY March 3rd at 11:59:59pm
- *Graded manually*
  - *Submission only checks for compilation, no auto tests*
  - *Won't get scores immediately*
  - *Only LAST submission will be graded*
- This project is challenging:
  - Requires working with *multiple* levels of abstraction.
  - Managing state in the paint program is a bit tricky.

Building a GUI library & application



○ Point ● Line ○ Ellipse ○ Text ☒ Thick lines

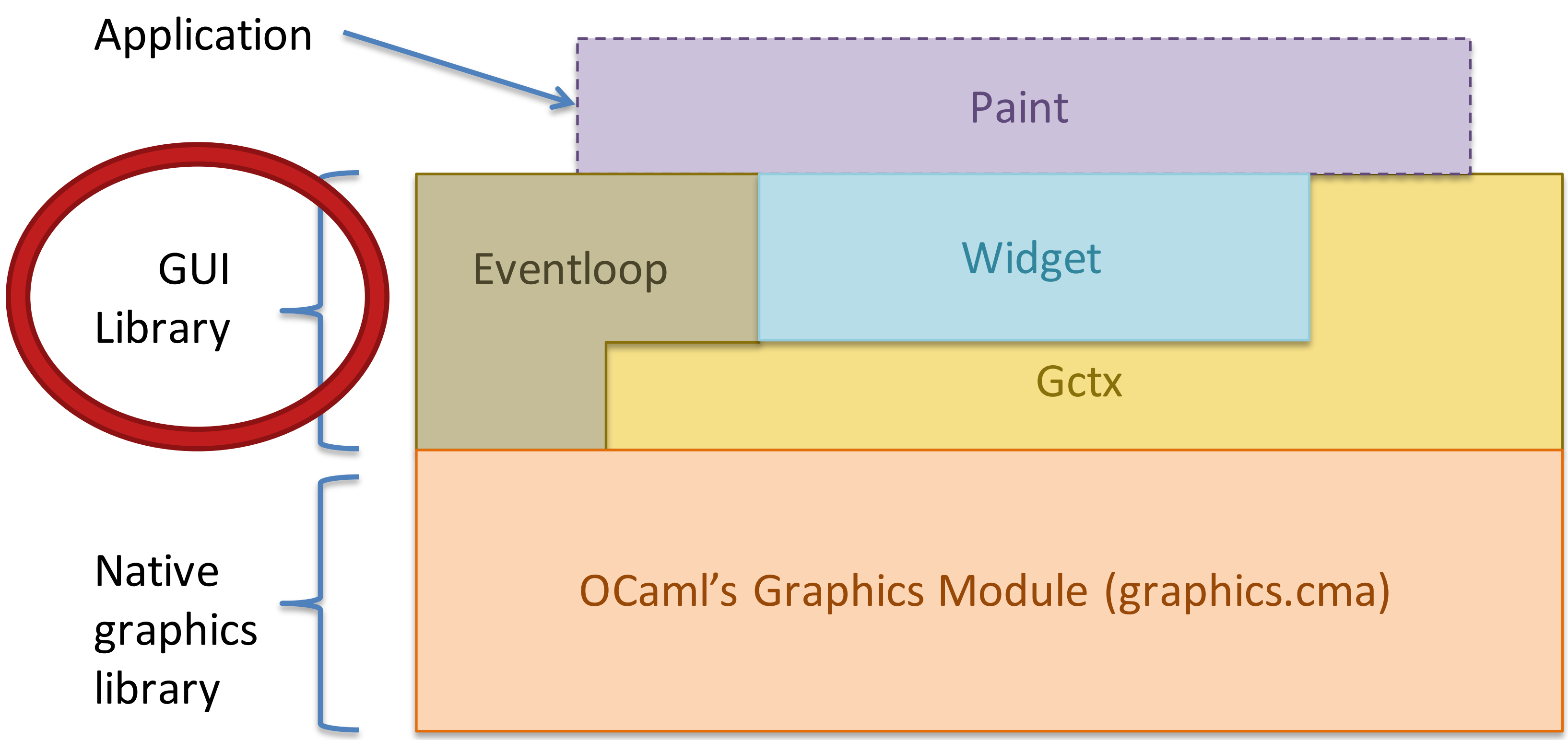
Text buffer: CIS 120

GUI Library Design

putting objects to work

Interfaces: Project Architecture*

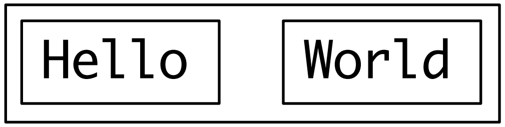
*Note: Subsequent program snippets are color-coded according to this diagram.



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)



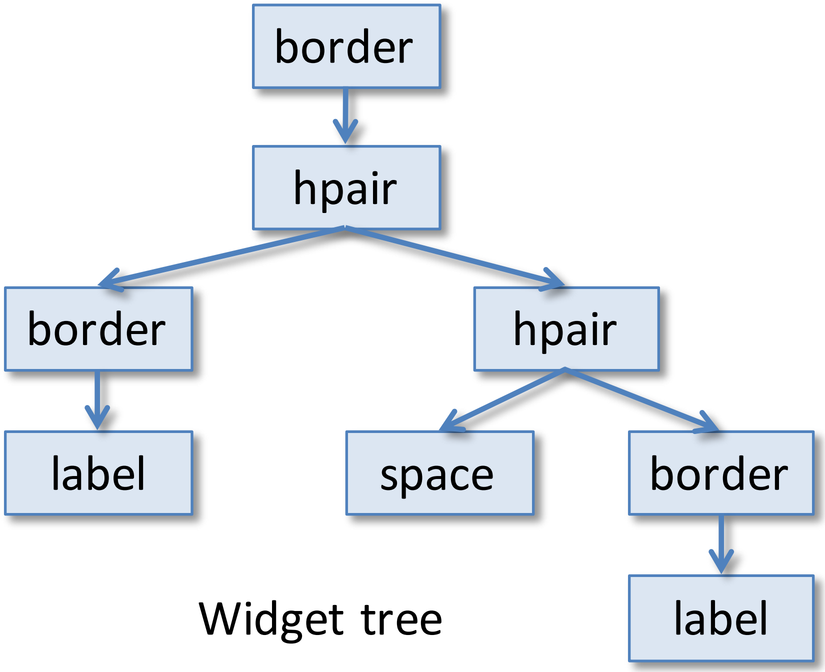
Hello World

*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.…

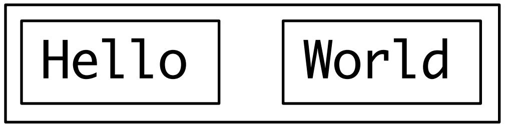
Widgets Pictorially

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

GUI terminology - Eventloop

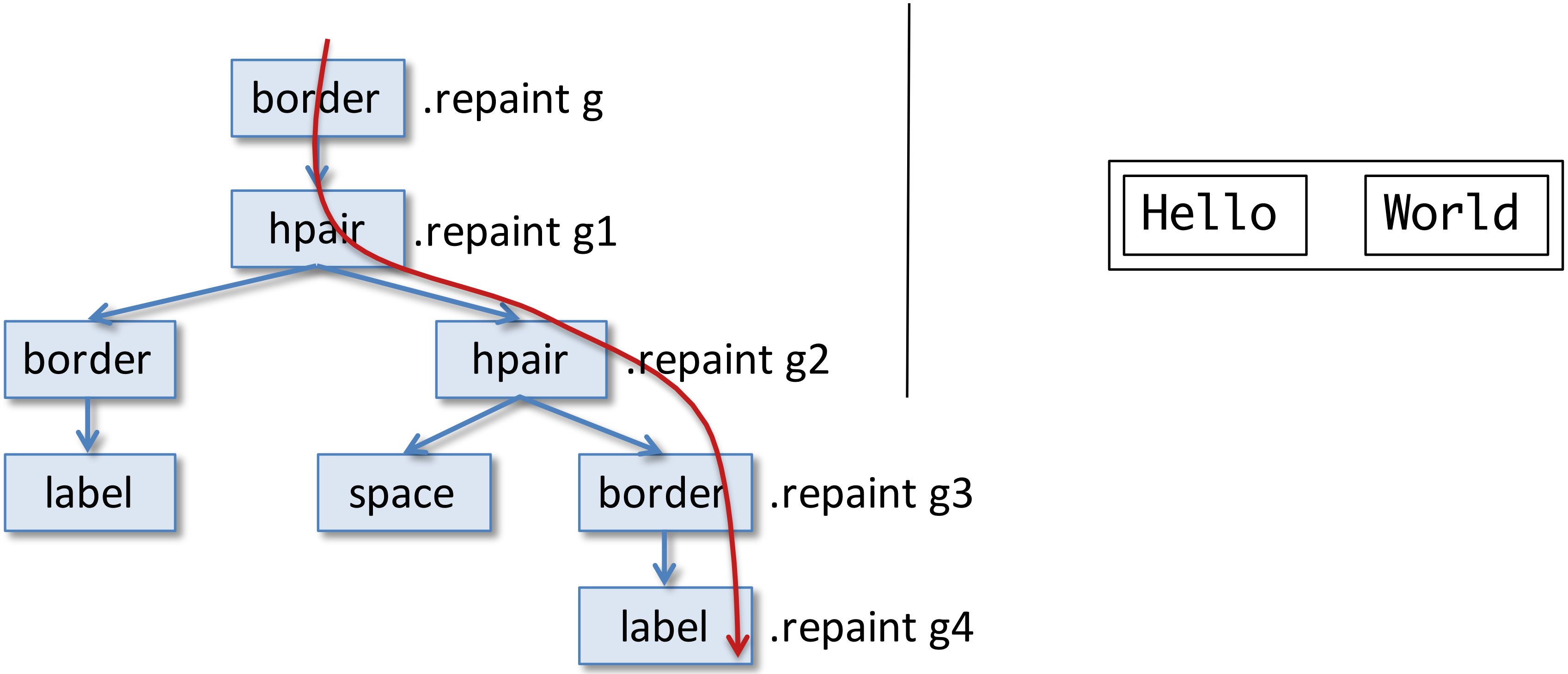
- Main loop of any GUI application

```
let run (w:widget) : unit =
  let g = Gctx.top_level in
  Gctx.open_graphics ();
  let rec loop () : unit =
    Graphics.clear_graph ();

    w.repaint g;

    Graphics.synchronize ();      (* force window update *)

    wait for user input (mouse movement, key press)
    inform w about the input 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.

Drawing: Containers

Container widgets propagate repaint commands to their children:



border .repaint g

hpair .repaint g1

border

hpair .repaint g2

label

space

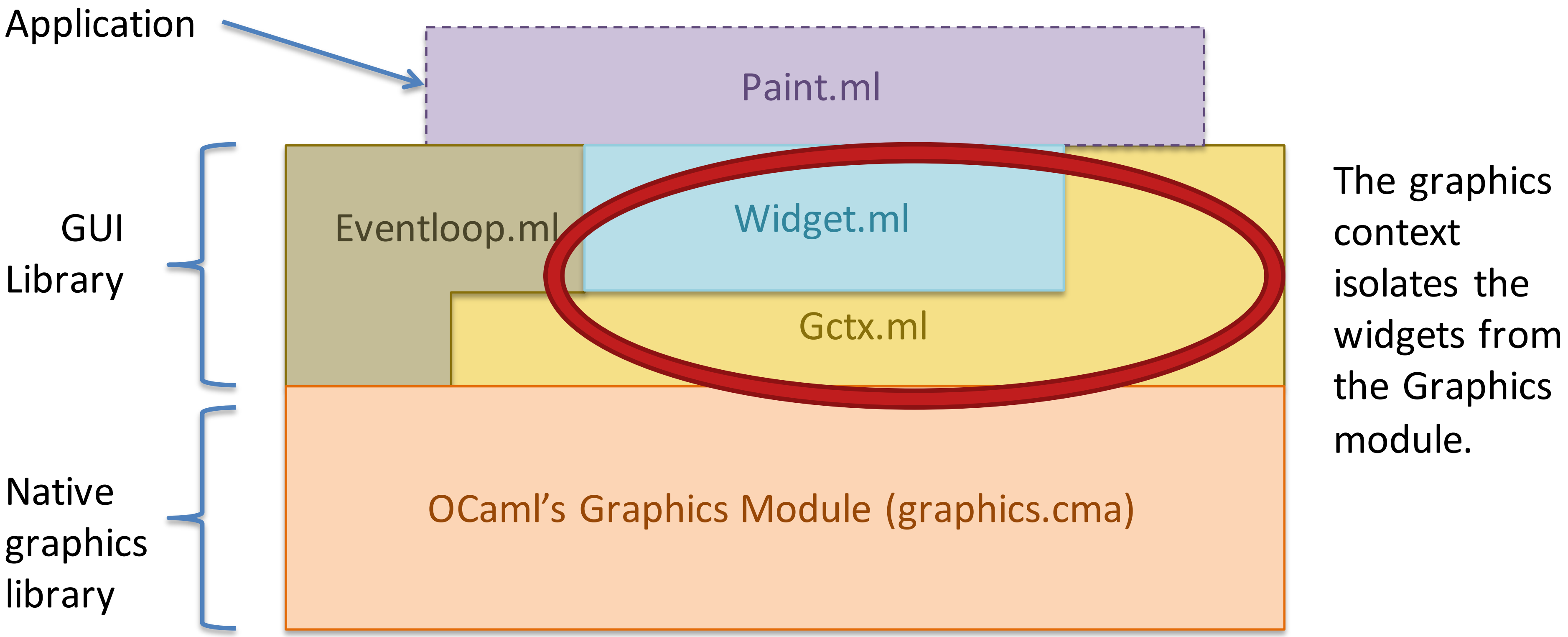
border .repaint g3

label .repaint g4

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

Challenge: Widget Layout

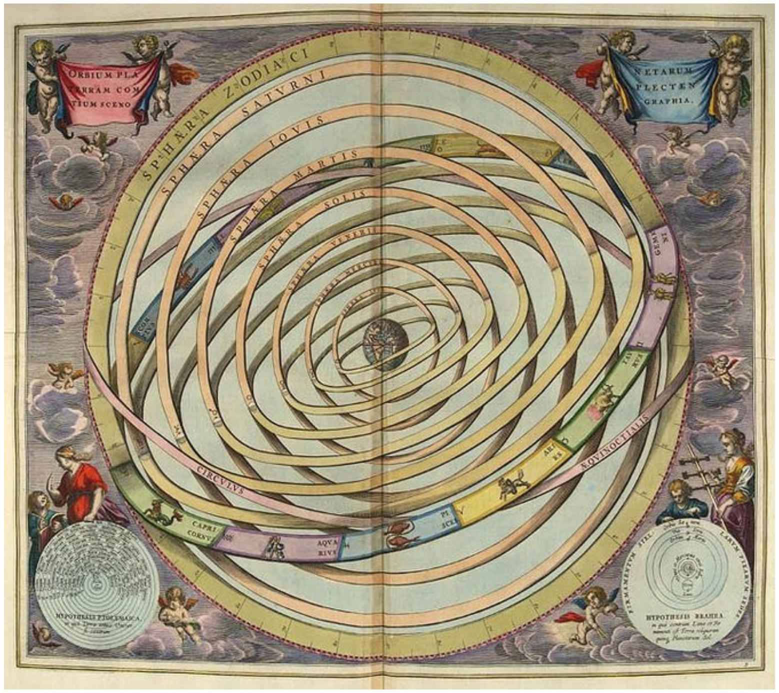
- Widgets are "things drawn on the screen". How to make them location independent?
- Idea: Use a *graphics context* to make drawing *relative* to the widget's current position



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



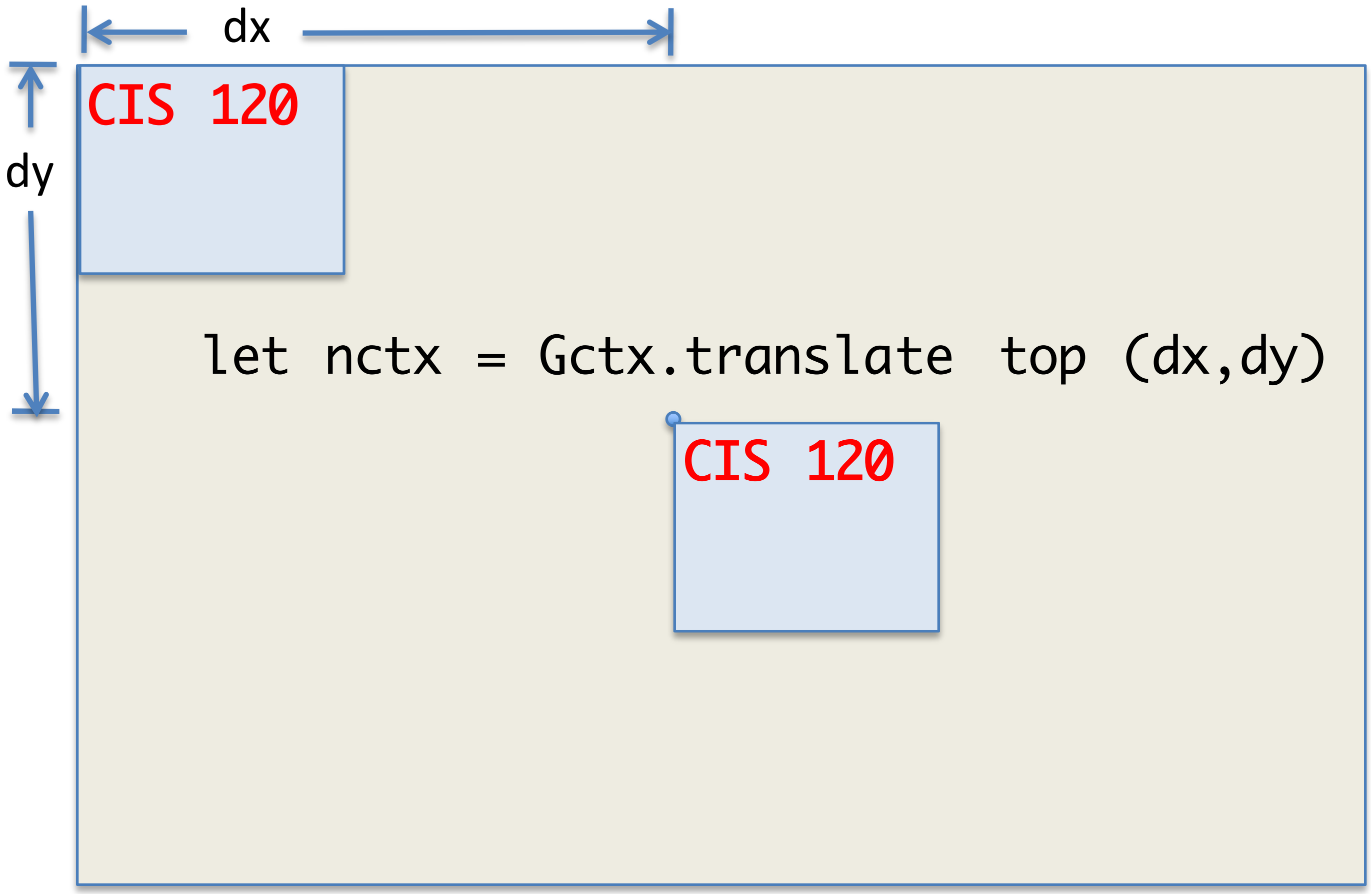
- Also aggregates information about the way things are drawn
  - foreground color
  - line width

Module: Gctx

Contextualizes graphics drawing operations

Graphics Contexts

```
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"
```

Module Gctx

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

(** The top-level graphics context *)
val top_level : gctx
(** A widget-relative position *)
type position = int * int

(** Display text at the given position *)
val draw_string : gctx -> position -> string -> unit
(** Draw a line between the two specified positions *)
val draw_line : gctx -> position -> position -> unit


(** Produce a new gctx shifted  by (dx,dy) *)
val translate : gctx -> int * int -> gctx
(** Produce a new gctx with a different pen color *)
val with_color : gctx -> color -> gctx
```

Module: Widgets

Building blocks of GUI applications

see simpleWidget.ml

Simple Widgets

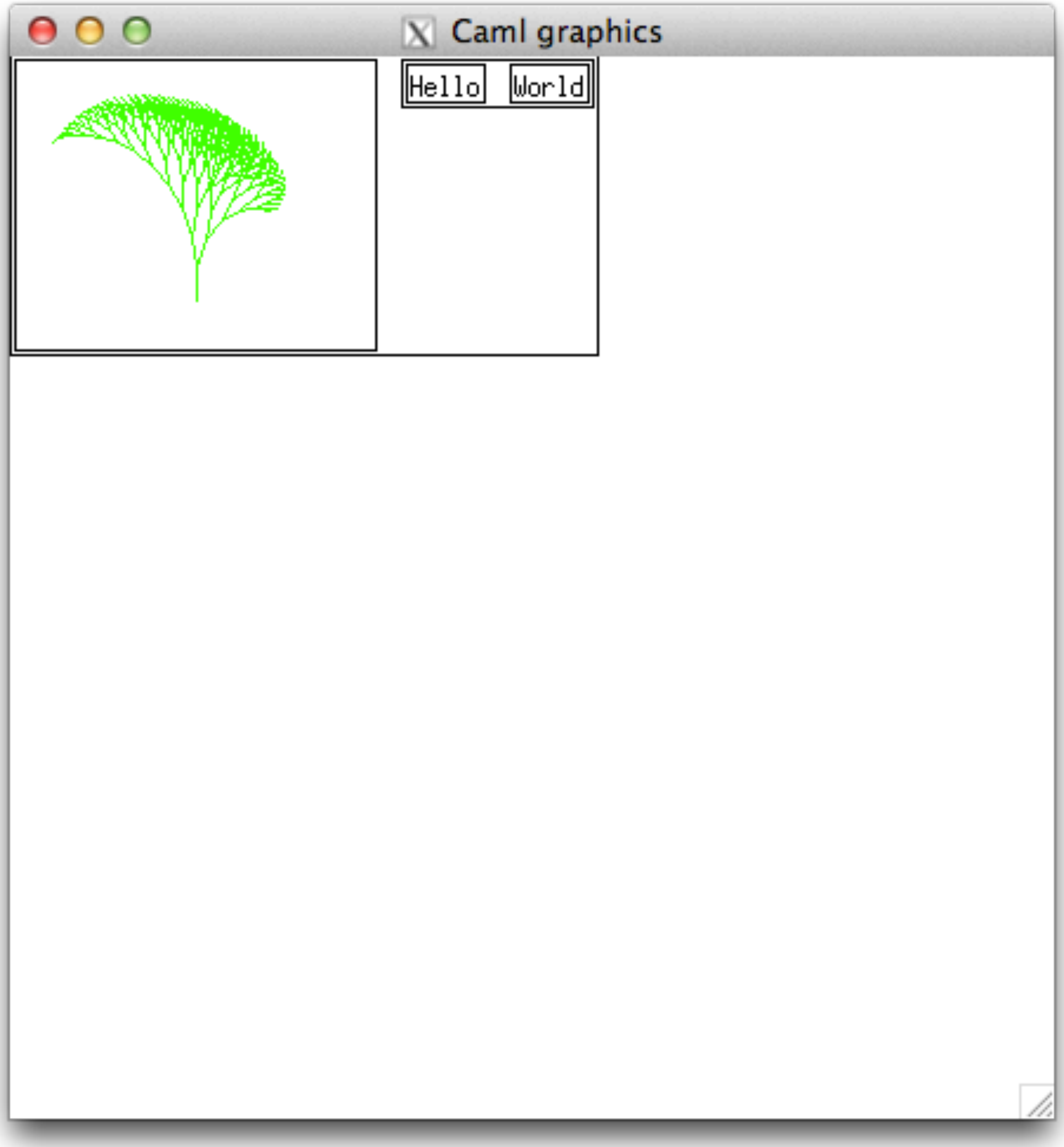
simpleWidget.mli

```
(* 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.

- Both operations are relative to a graphics context

swdemo.ml



Widget Examples

simpleWidget.ml

```
(* 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)
}
```

simpleWidget.ml

```
(* 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

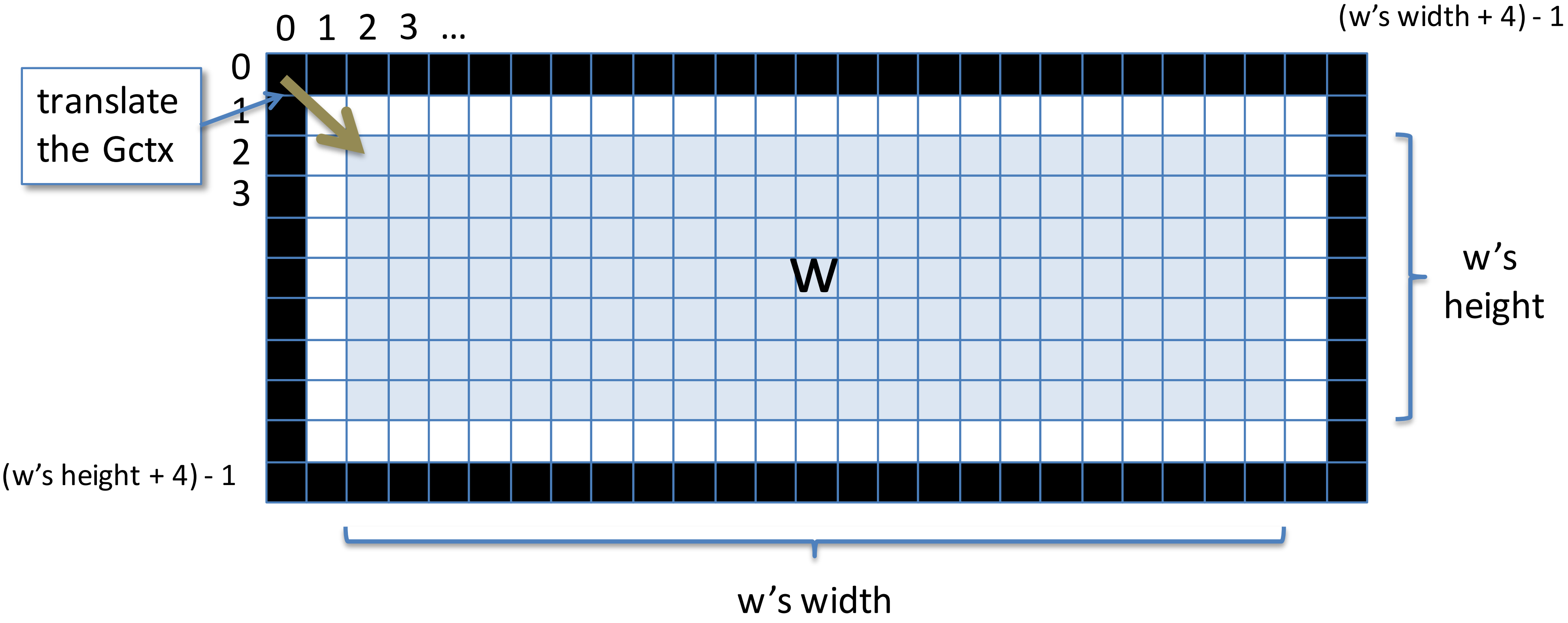
simpleWidget.ml

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

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

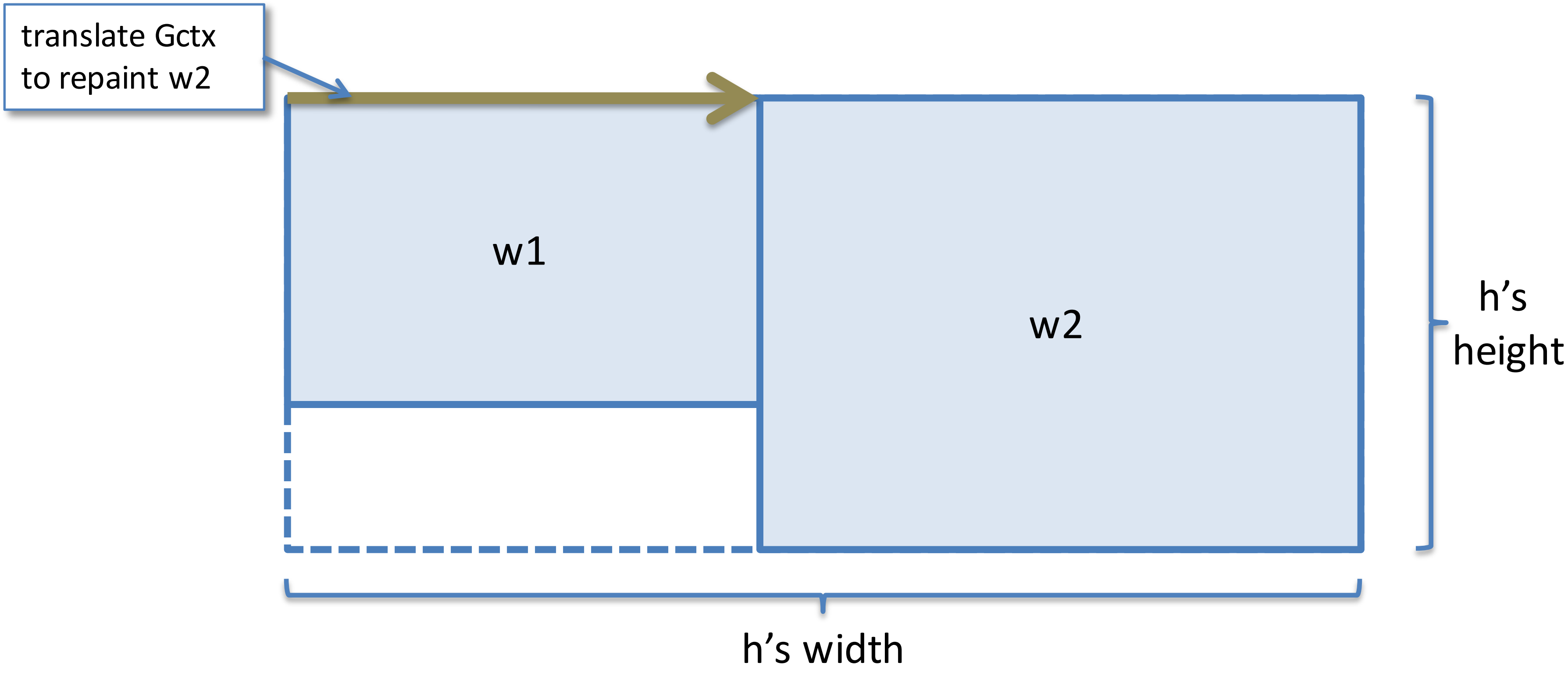
simpleWidget.ml

```
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 the right widget
- Size is the sum of their widths and max of their heights

The hpair Widget

simpleWidget.ml

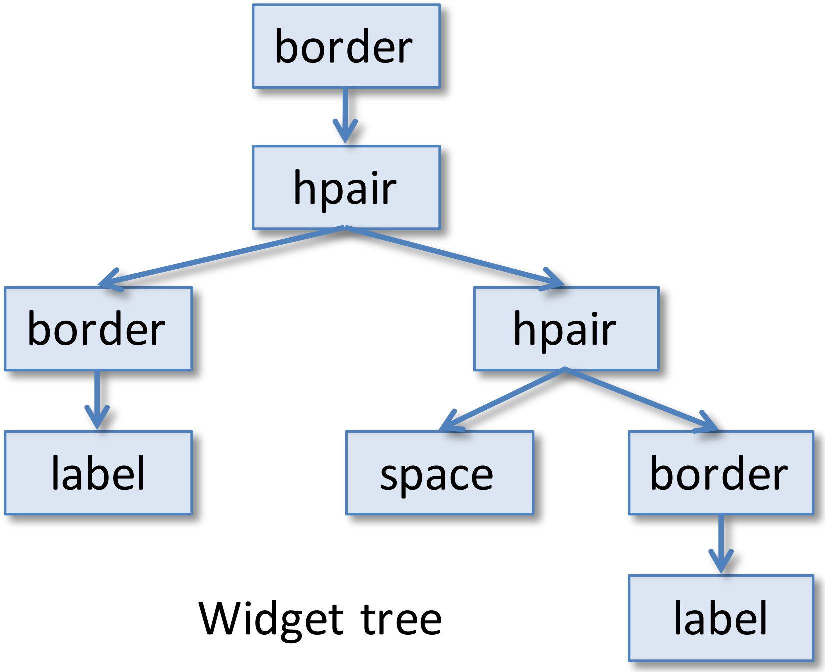
```
let hpair (w1: widget) (w2: widget) : widget =
  {
    repaint = (fun (g: Gctx.gctx) ->
            let (x1, _) = w1.size () in begin
              w1.repaint g;
              w2.repaint (Gctx.translate g (x1,0))
              (* Note translation of the Gctx *)
            end);

    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.

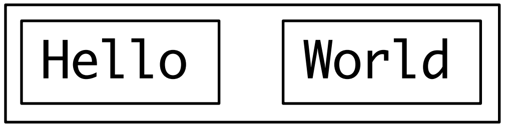
Widget Hierarchy Pictorially

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)))
```



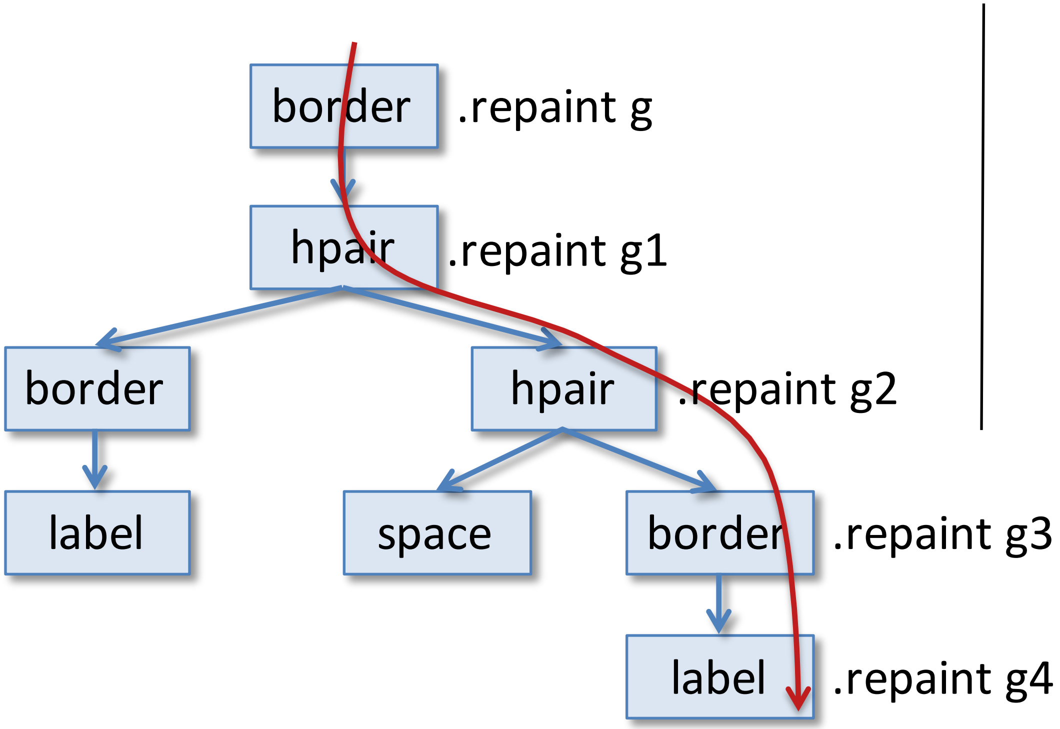
Widget tree

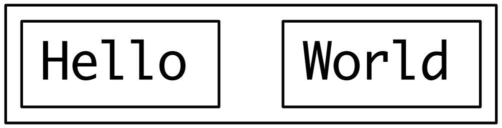


On the screen

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