Programming Languages and Techniques (CIS1200)

Lecture 18

GUI library: Widgets, Layout, and Event Handling

Chapter 18

Announcements

- HW05 available **now**, due *Tuesday*, March 18th (at 11.59pm)
  - Start early! This homework is long and challenging.
  - Tasks 0-1 can be done after class today
  - Tasks 2-4 can be done after class on Monday
  - Tasks 5-6 can be done after class on Wednesday

- Final Exam
  - Wednesday, May 7th, 9-11 am

Objects and GUIs

Where we're going…

- HW 5: Build a GUI library and client application *from scratch* in OCaml

- Goals:
  - Practice with *first-class functions* and *hidden state (Ch 17)*
  - Bridge to object-oriented programming in Java
  - Illustrate the *event-driven programming* model
  - Give a feel for how GUI libraries (like Java's Swing) are put together
  - Apply everything we've seen so far to do some pretty serious programming

17: Have you ever used a GUI library (such as Java's Swing) to construct a user interface?

0

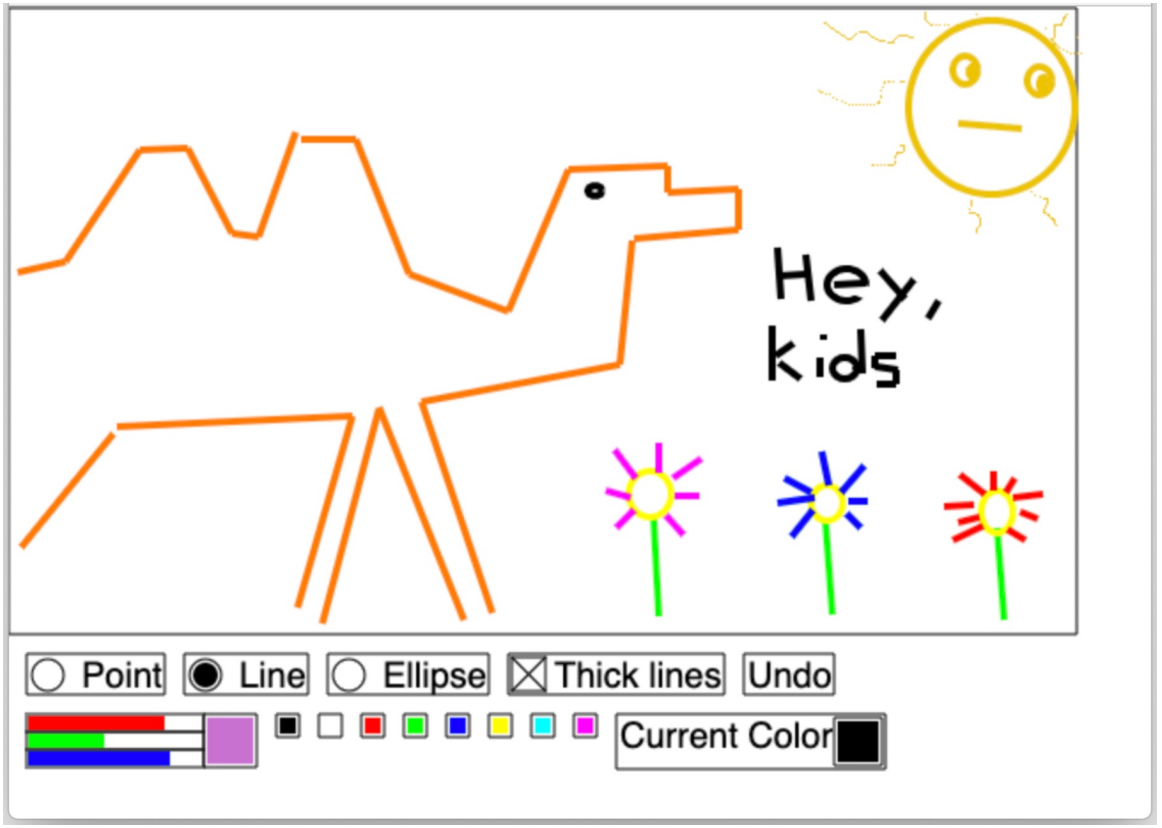
Yes

0%

No

0%

Building a GUI library & application

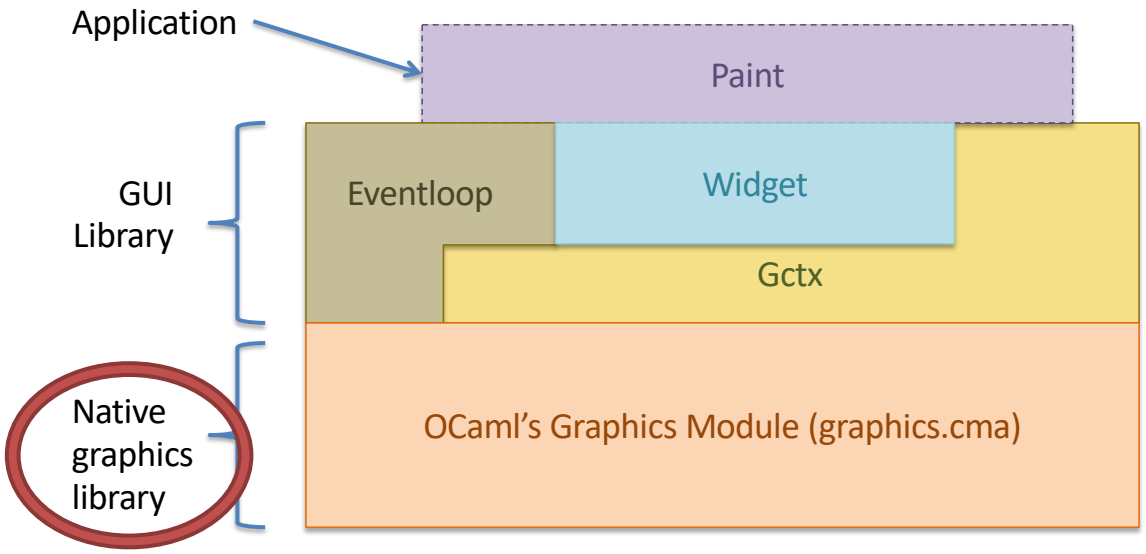


Step #1: Understand the Problem

- There are two separate parts of this homework: an *application* (Paint) and a *GUI library* (several files) used to build the application
- What are the concepts involved in *GUI libraries* and how do they relate to each other?
- How can we separate the various concerns on the project?
- Goal: The library should be *reusable*. It should be useful for other applications besides Paint.

Step #2, Interfaces: Project Architecture*

*program snippets will be 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.

Starting point: The low-level `Graphics` module

- OCaml's `Graphics` library provides *very basic* primitives for:
  - Creating an area in the screen for graphics
  - 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: https://ocaml.github.io/graphics/graphics/Graphics/

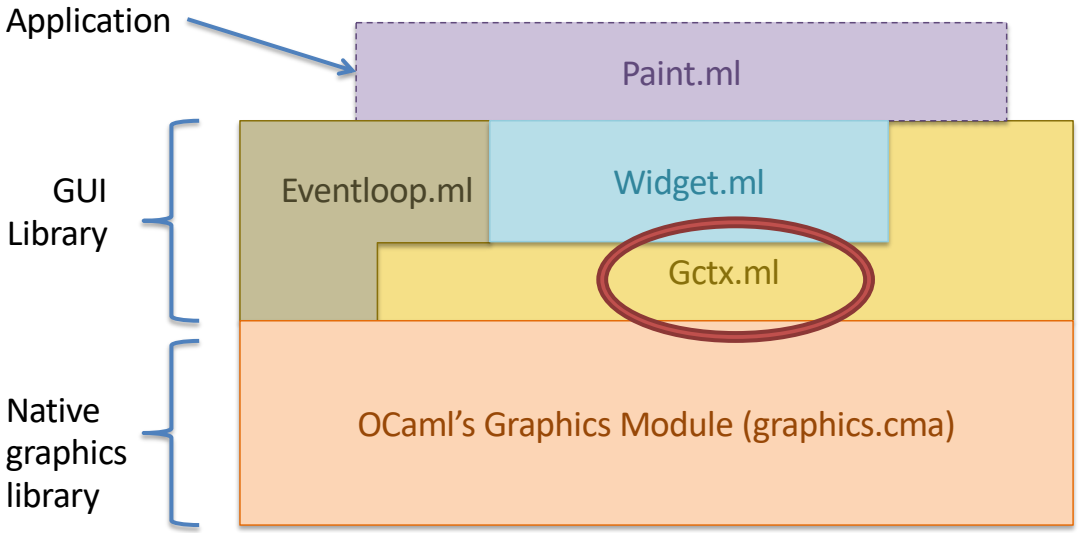
- How do we go from that to a full-blown GUI library?

Module: Gctx

"Contextualizes" graphics operations

Challenge: Widget Layout

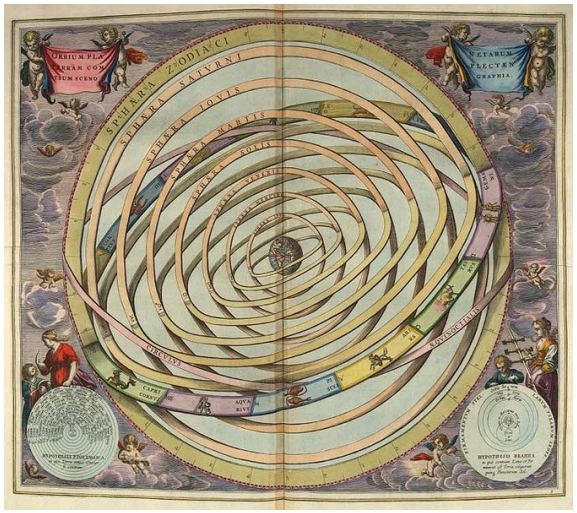
- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a *graphics context* to allow drawing *relative* to a widget’s current position



The graphics context isolates the widgets from the Graphics module.

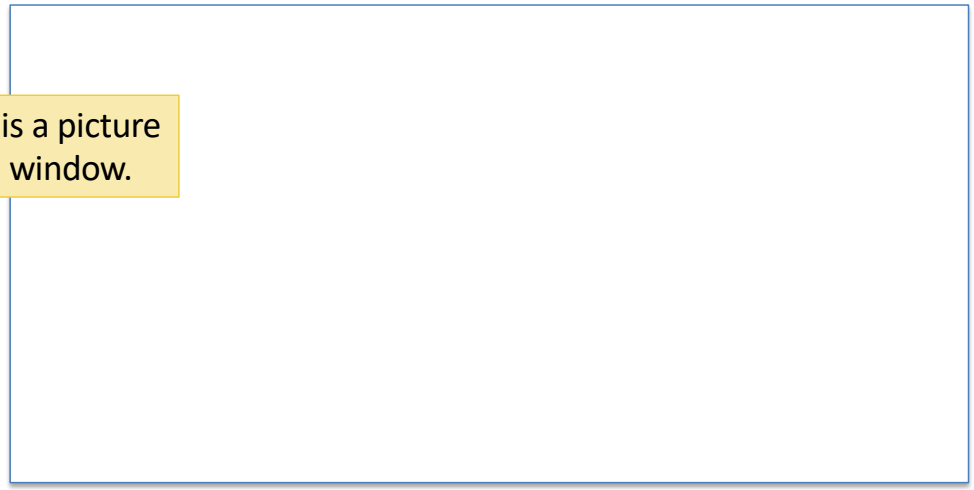
GUI terminology – Graphics Context

- **Translates** coordinates
  - *Translates* coordinates so all widgets can pretend that they are at the origin
  - *Flips* from OCaml to “standard” coordinates so origin is top-left
- Also carries information about **how** things should be drawn:
  - color
  - line width
- "Task 0" in the homework helps you understand the interaction between Gctx and OCaml's Graphics module



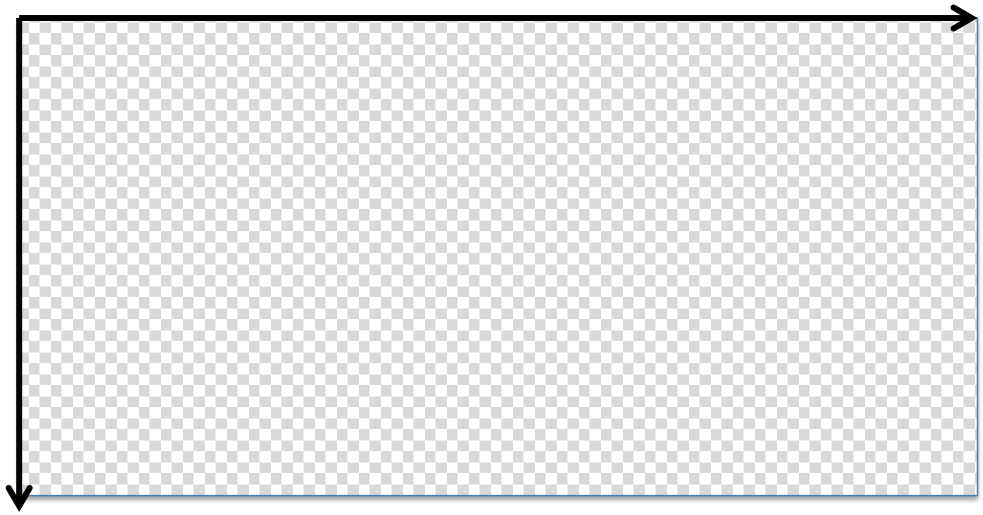
Graphics Contexts

This top box is a picture of the whole window.



```
let top  = Gctx.top_level in
```

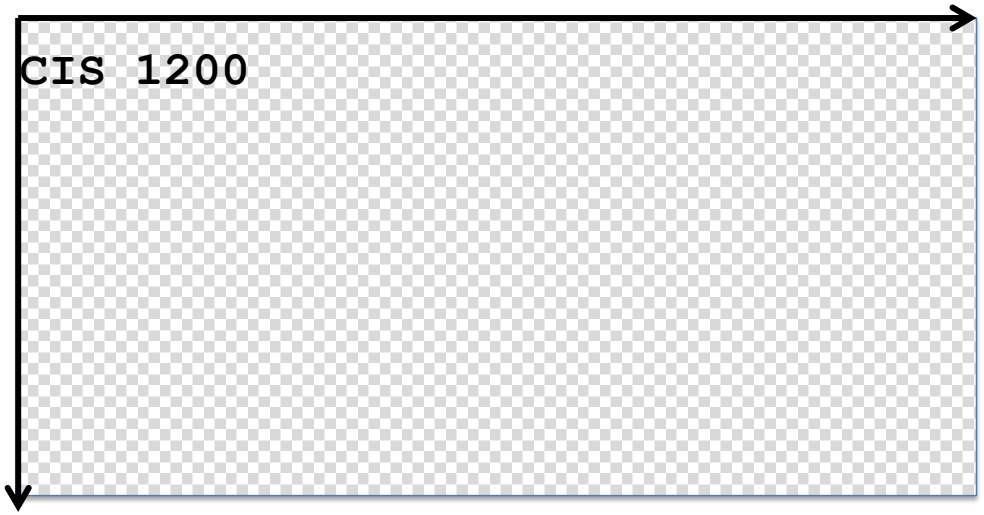
Graphics Contexts



```
let top  = Gctx.top_level
```

The top graphics context represents a coordinate system anchored at (0,0), with current pen color of black.

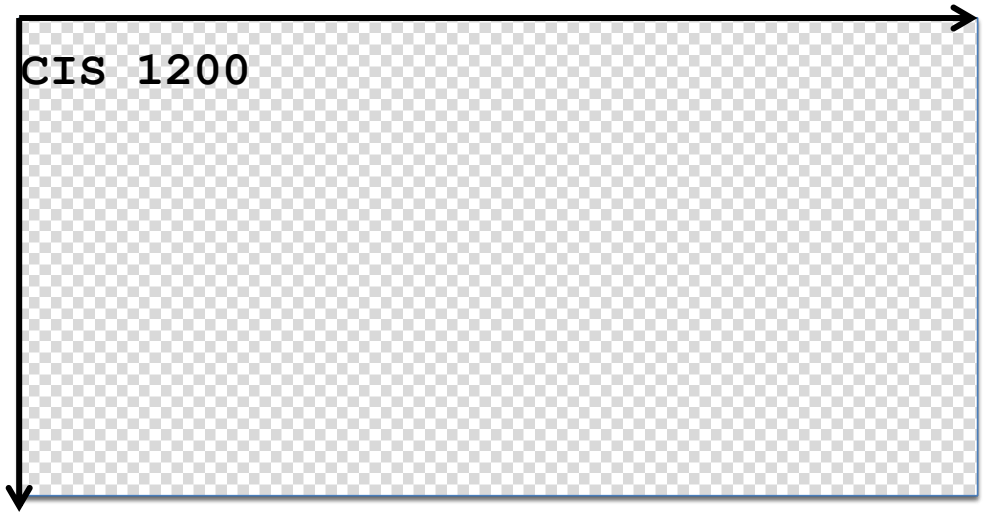
Graphics Contexts




```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200"
```

Drawing a string at (0,10) in this context positions it on the left edge and 10 pixels down.
The string is drawn in black.

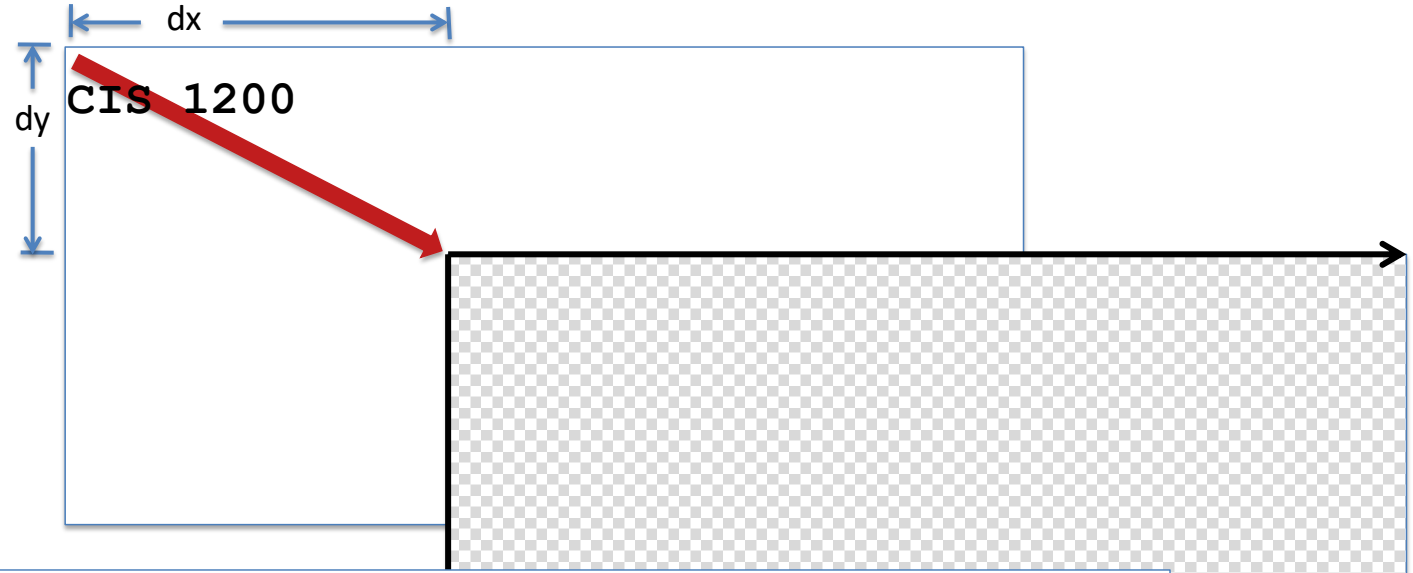
Graphics Contexts




```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200"

(* move origin and change the color *)
let nctx = Gctx.with_color
               (Gctx.translate top (dx,dy)) red
```

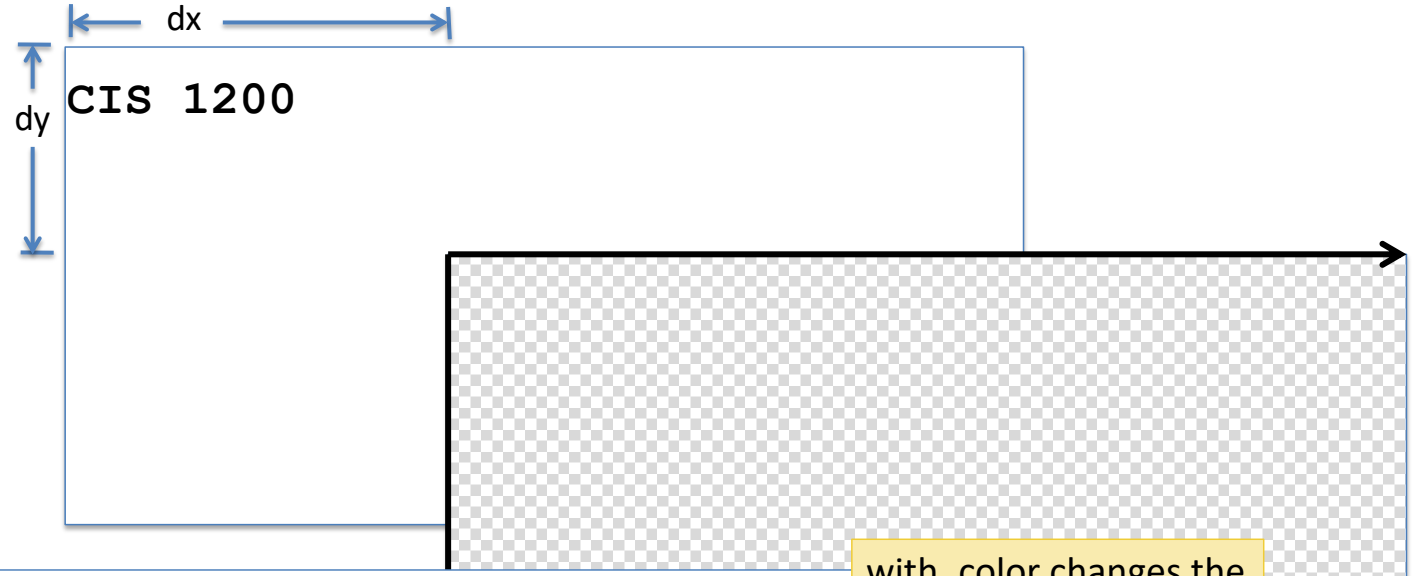
Translating the gctx has the effect of shifting the origin relative to the old origin.

Graphics Contexts




```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200"

(* move origin and change the color *)
let nctx = Gctx.with_color
               (Gctx.translate top (dx,dy)) red
```

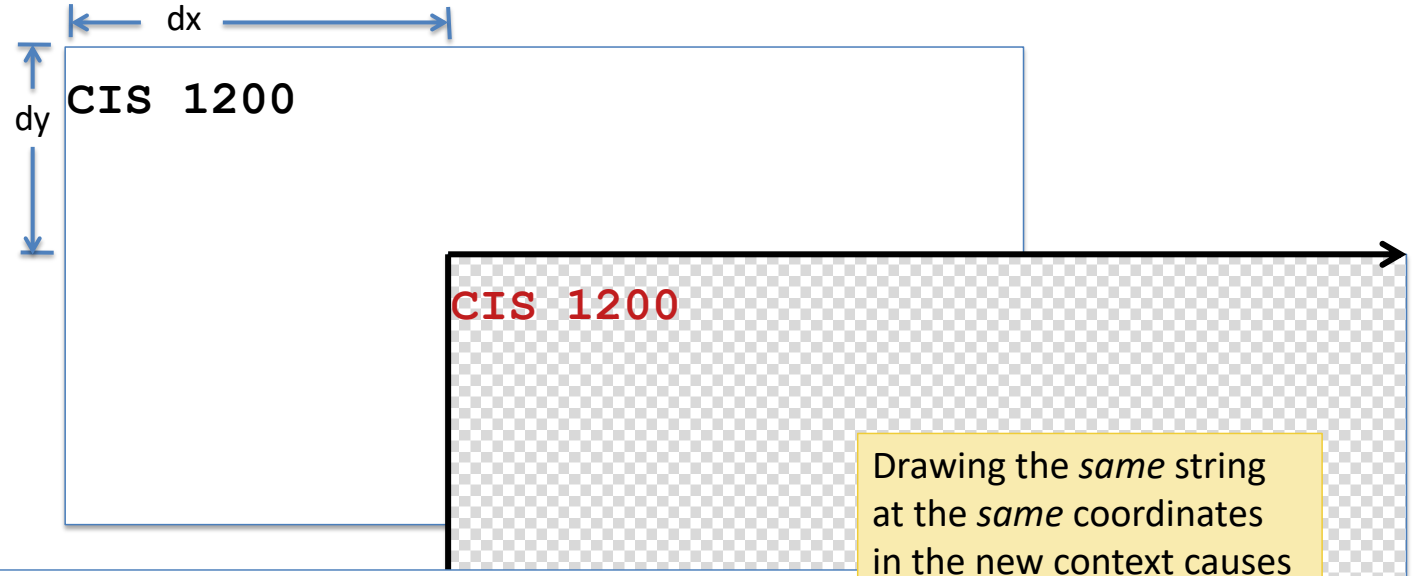
Graphics Contexts




```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200"

(* move origin and change the color *)
let nctx = Gctx.with_color
               (Gctx.translate top (dx,dy)) red
```

with_color changes the current drawing color…

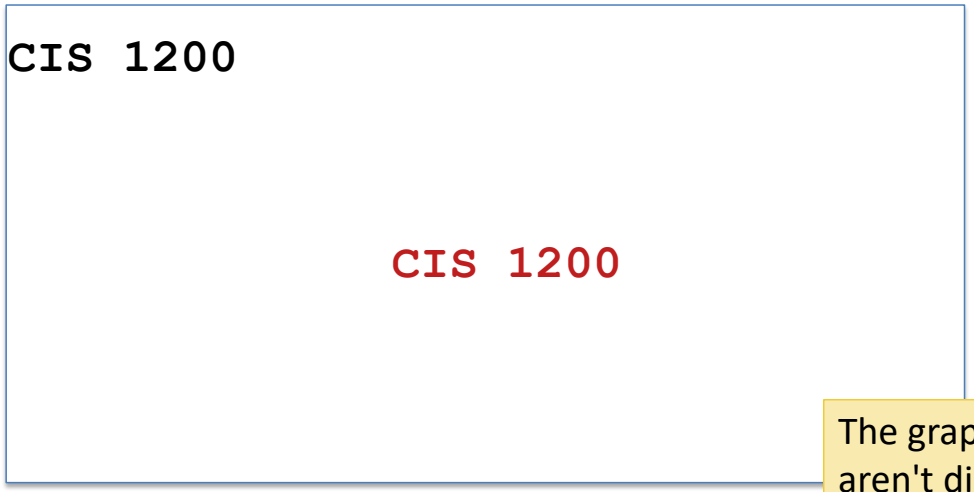
Graphics Contexts




```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200"

(* move origin and change the color *)
let nctx = Gctx.with_color
               (Gctx.translate top (dx,dy)) red
;; Gctx.draw_string nctx (0,10) "CIS 1200"
```

Drawing the *same* string at the *same* coordinates in the new context causes it to display at a translated location and in the new color.

Graphics Contexts

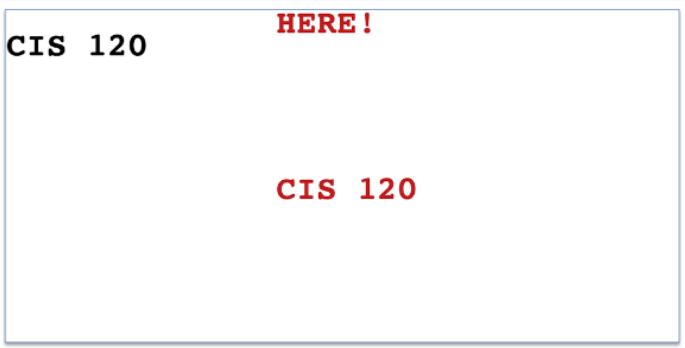


The graphics contexts aren't displayed anywhere: they only serve as frames of reference...

```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200"

(* move origin and change the color *)
let nctx = Gctx.with_color
               (Gctx.translate top (dx,dy)) red
;; Gctx.draw_string nctx (0,10) "CIS 1200"
```

18: Which of the following can we fill in for ??? to obtain the result shown?

0




```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 120"
let nctx = Gctx.with_color Color.red
             (Gctx.translate top (dx,dy))
;; Gctx.draw_string nctx (0,10) "CIS 120"
let ctx3 = ???
;; Gctx.draw_string ctx3 (0,0) "HERE!"
```

Gctx.translate top (dx,0)

0%

Gctx.translate top (0,-dy)

0%

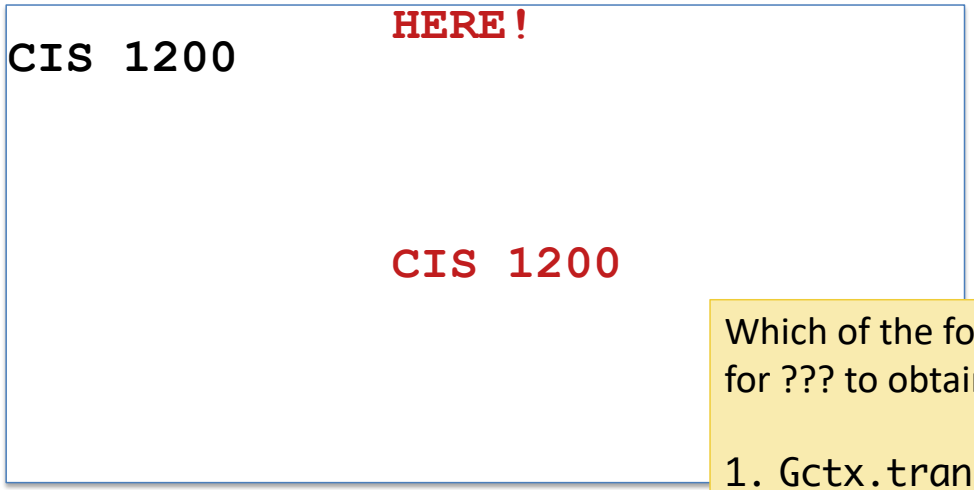
Gctx.translate nctx (dx,0)

0%

Gctx.translate nctx (0,-dy)

0%

Graphics Contexts



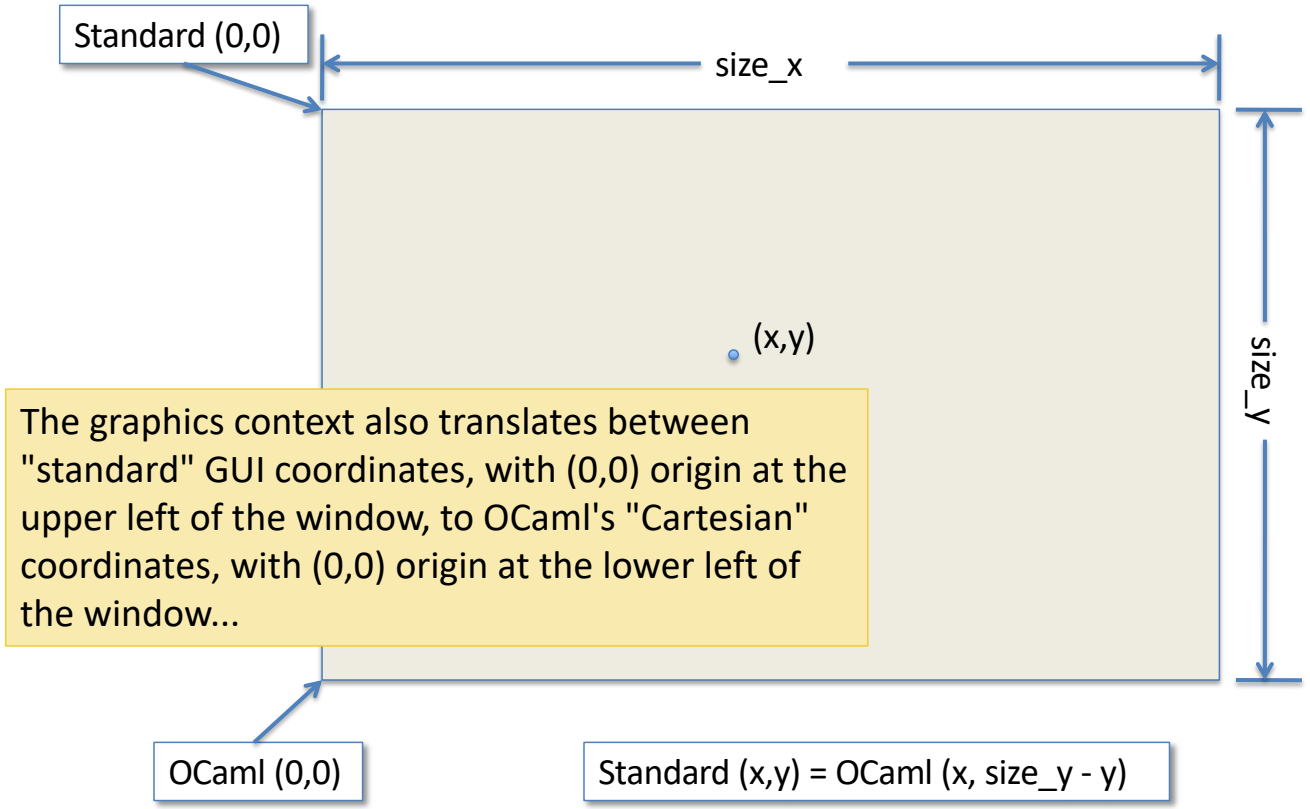

```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 1200"
let nctx = Gctx.with_color
              (Gctx.translate top (dx,dy)) red
;; Gctx.draw_string nctx (0,10) "CIS 1200"
let ctx3 = ???
;; Gctx.draw_string ctx3 (0,0) "HERE!"
```

Which of the following can we fill in for ??? to obtain the result shown?

1. `Gctx.translate top (dx,0)`
2. `Gctx.translate top (0,-dy)`
3. `Gctx.translate nctx (dx,0)`
4. `Gctx.translate nctx (0,-dy)`

Answer: 4

OCaml vs. “Standard” Coordinates



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 (relative) 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
```

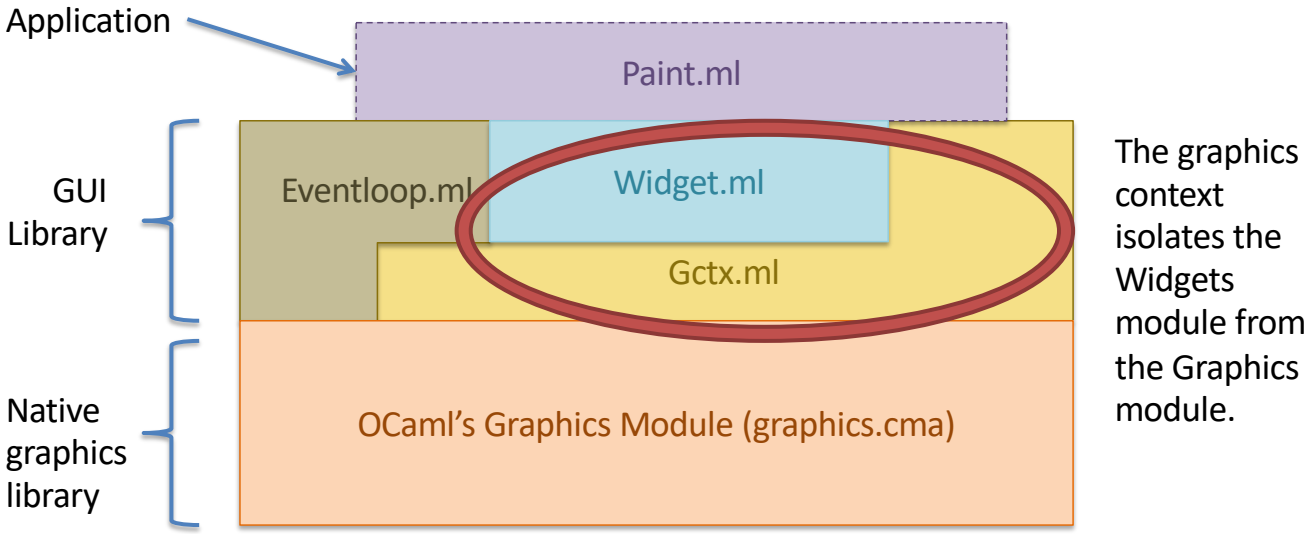
Widget Layout

Building blocks of GUI applications

see simpleWidget.ml in GUI Demo Code project

Widget Layout

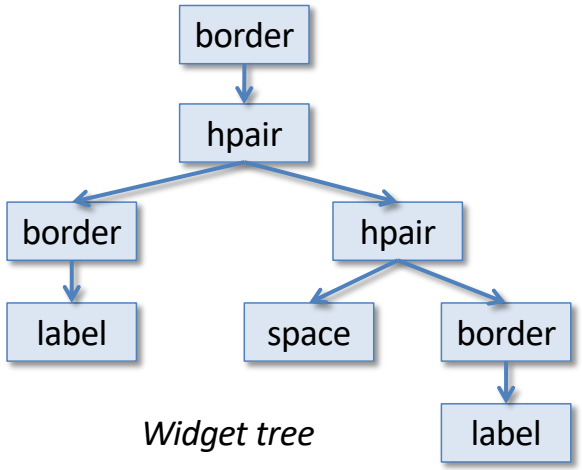
- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a *graphics context* to make drawing *relative* to the widget’s current position



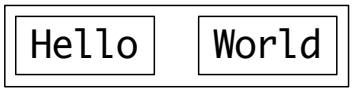
A “Hello World” application

hello.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

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 (relative to some graphics context)
- You can ask a simple widget to tell you its size
- (We'll talk about handling events later)

Widget Examples

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

simpleWidget.ml

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

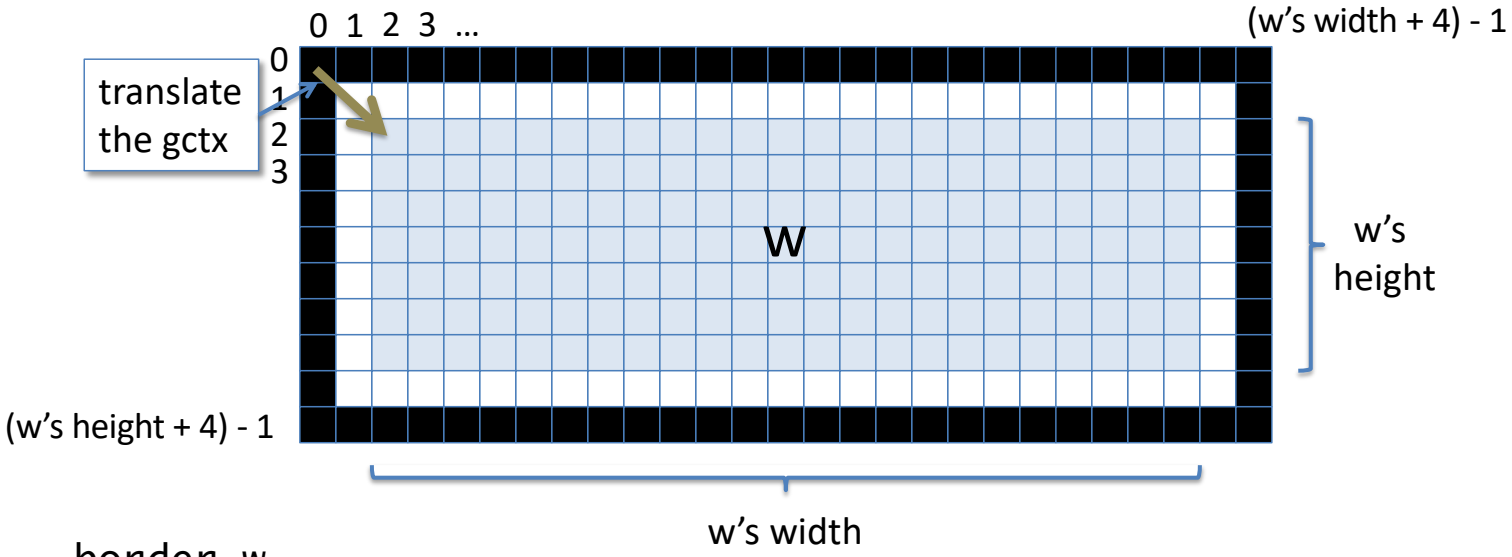
```
let canvas ((w,h):int*int) (r: Gctx.gctx -> unit) : widget =
{
  repaint = r;
  size = (fun () -> (w,h))
}
```

simpleWidget.ml

Nested Widgets

Containers and Composition

The Border Widget Container




```
let b = border w
```

- Draws a one-pixel-wide border (+ a one-pixel space) 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

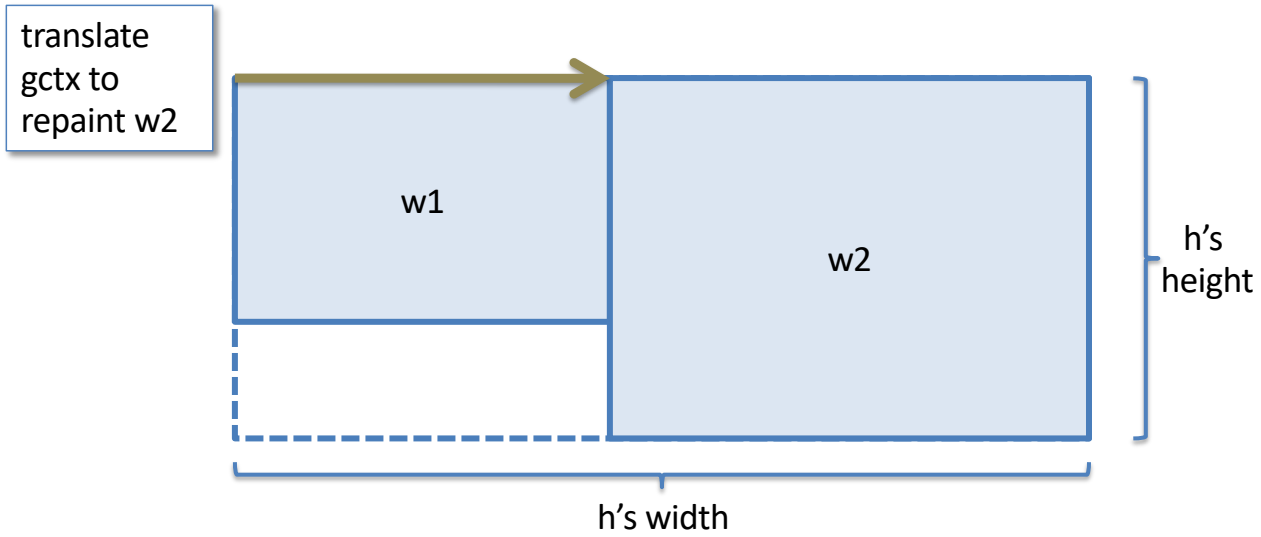
simpleWidget.ml

```
let border (w:widget):widget =
{
repaint = (fun (g:Gctx.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
- Size is the *sum* of their widths and *max* of their heights

The hpair Widget

simpleWidget.ml

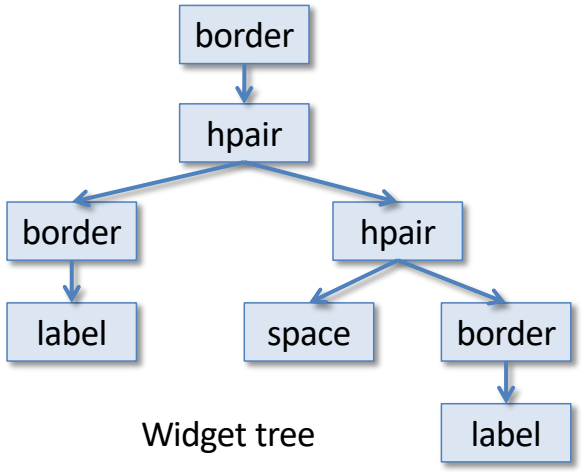
```
let hpair (w1: widget) (w2: widget) : widget =
  {
    repaint = (fun (g: Gctx.gctx) ->
            let (x1, _) = w1.size () in begin
              w1.repaint g;
              w2.repaint (Gctx.translate g (x1,0))
              (* Note translation of the Gctx *)
            end);

    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.

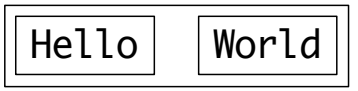
Widget Hierarchy Pictorially

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)))
```



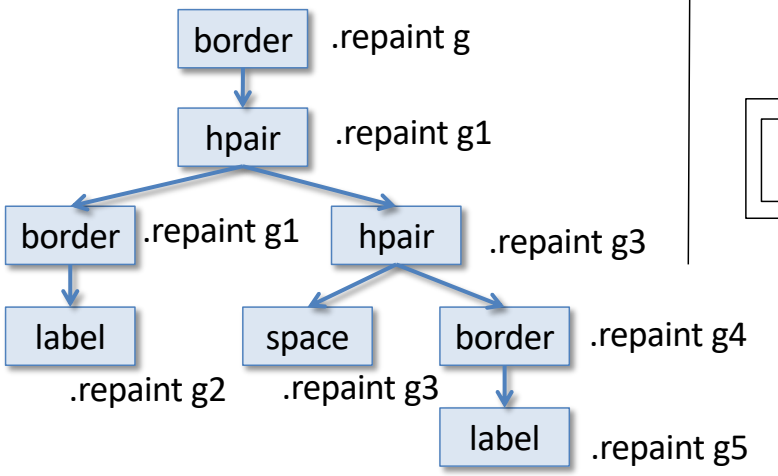
Widget tree

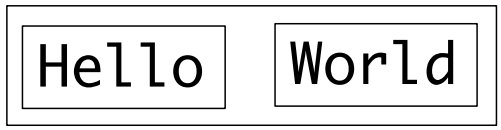


On the screen

Drawing: Containers

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



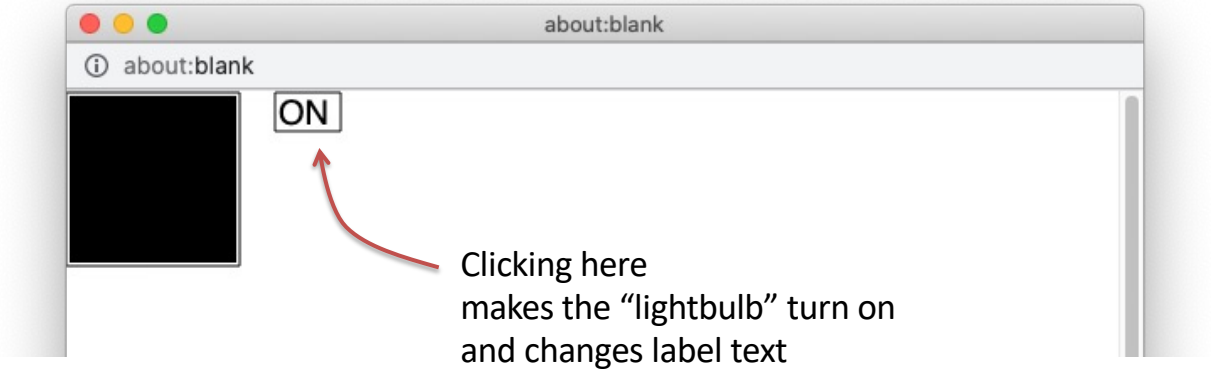


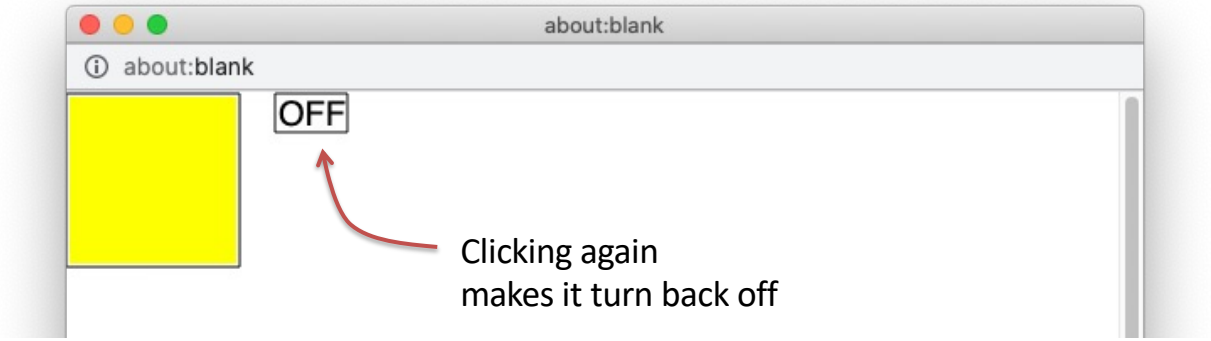
g1 = Gctx.translate g (2,2)
g2 = Gctx.translate g1 (2,2)
g3 = Gctx.translate g1 (hello_width,0)
g4 = Gctx.translate g3 (space_width,0)
g5 = Gctx.translate g4 (2,2)

Coding with Simple Widgets

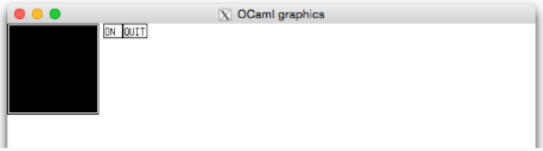
see swdemo.ml

"lightbulb" demo





19: Do you know how you would use the (simple) widget library to define the layout of this application?




```
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
```

I don't know how to start

0%

I may have it, but I'm not sure

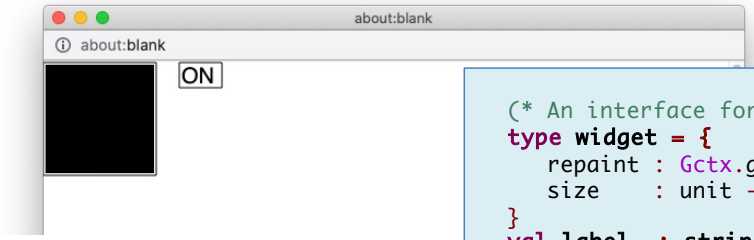
0%

I'm sure I've got it

0%

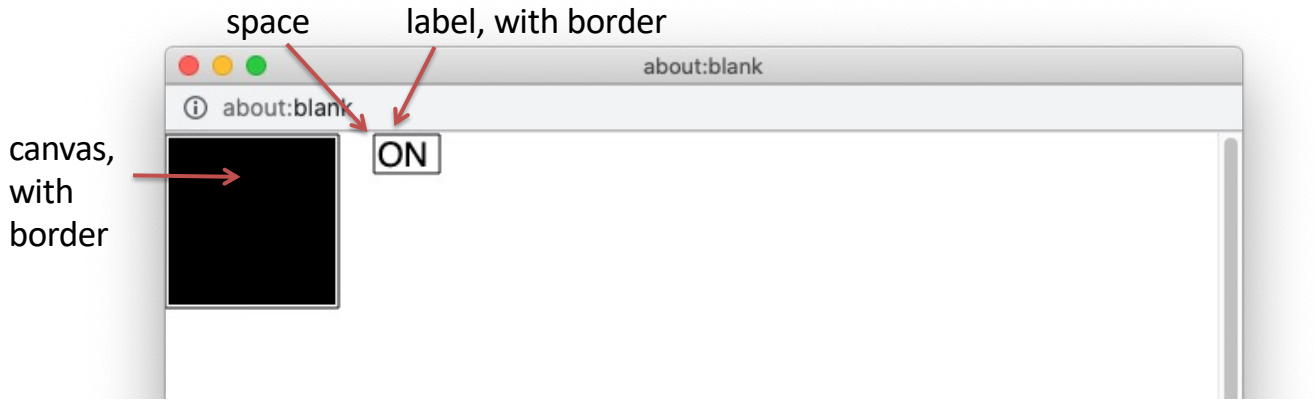
Do you know how you would use the (simple) widget library to define the layout of this lightbulb application?

1. I'm not sure how to start.
2. I may have it, but I'm not sure.
3. Sure! No problem.




```
(* 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
```

"lightbulb" demo layout



swdemo.ml

```
let onoff = border (label "ON")

let paint_bulb (g: Gctx.gctx) : unit = …

let bulb = border (canvas (100, 100) paint_bulb)

let top : widget = hpair bulb (hpair (space (20, 20)) onoff)
```