Programming Languages and Techniques (CIS120)

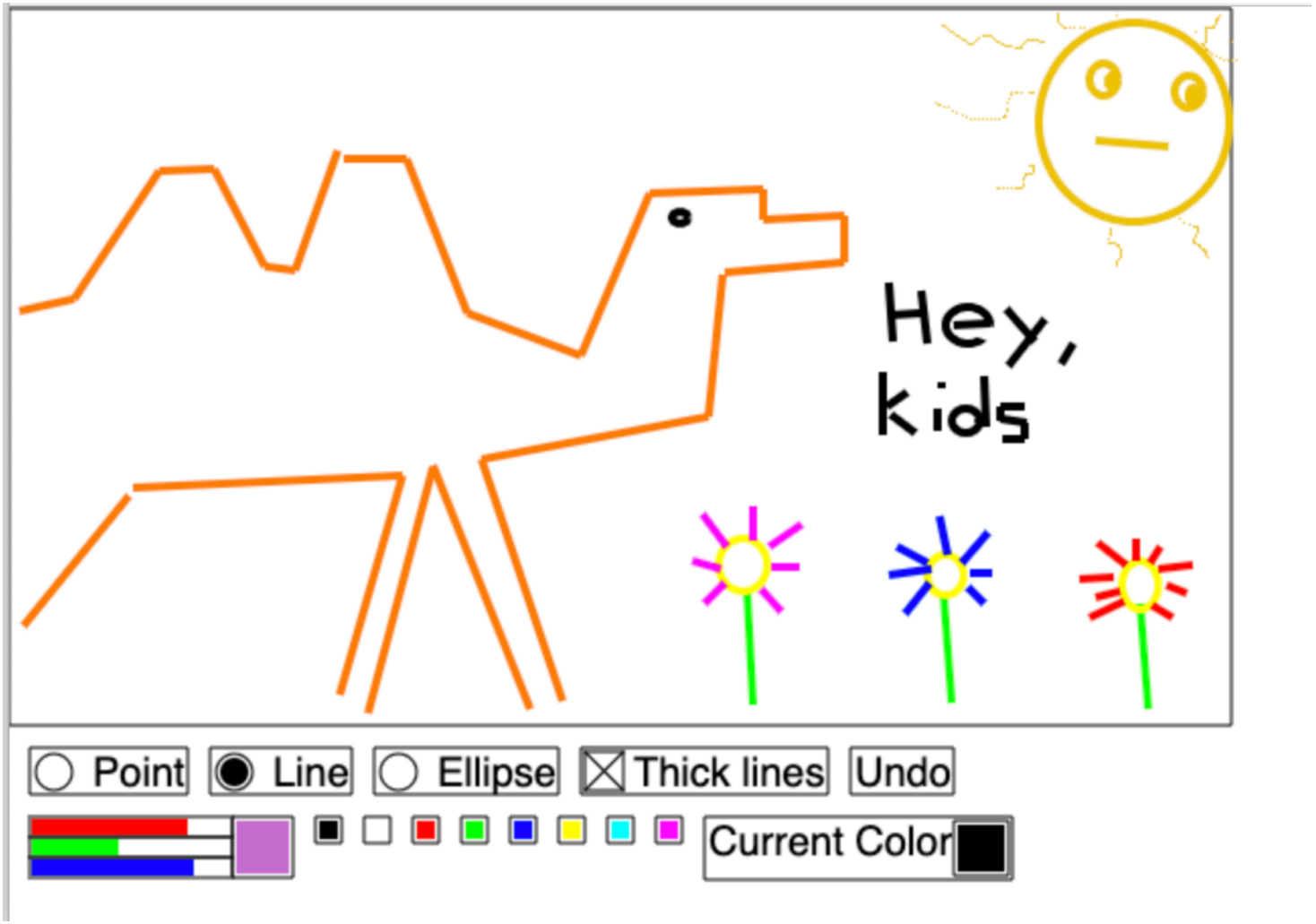
Lecture 18

GUI Library Design

Chapter 18

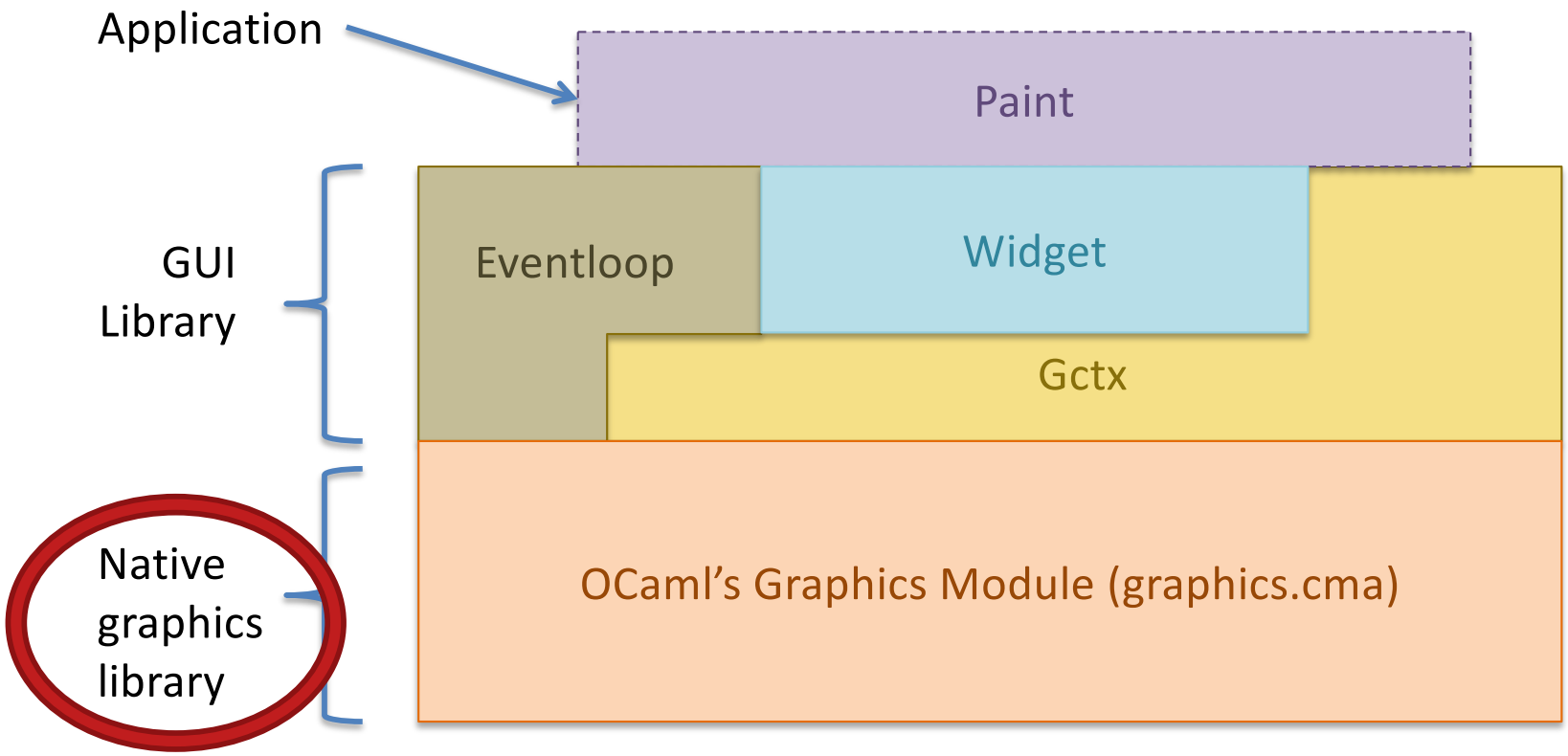
Where we're going…

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



Project Architecture*

*Subsequent 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 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: https://ocaml.github.io/graphics/graphics/Graphics/

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

*Note: We actually have *two* Graphics libraries, one for running "natively" and one for running in the browser. We have configured the project so that you can refer to either one using the module alias Graphics.

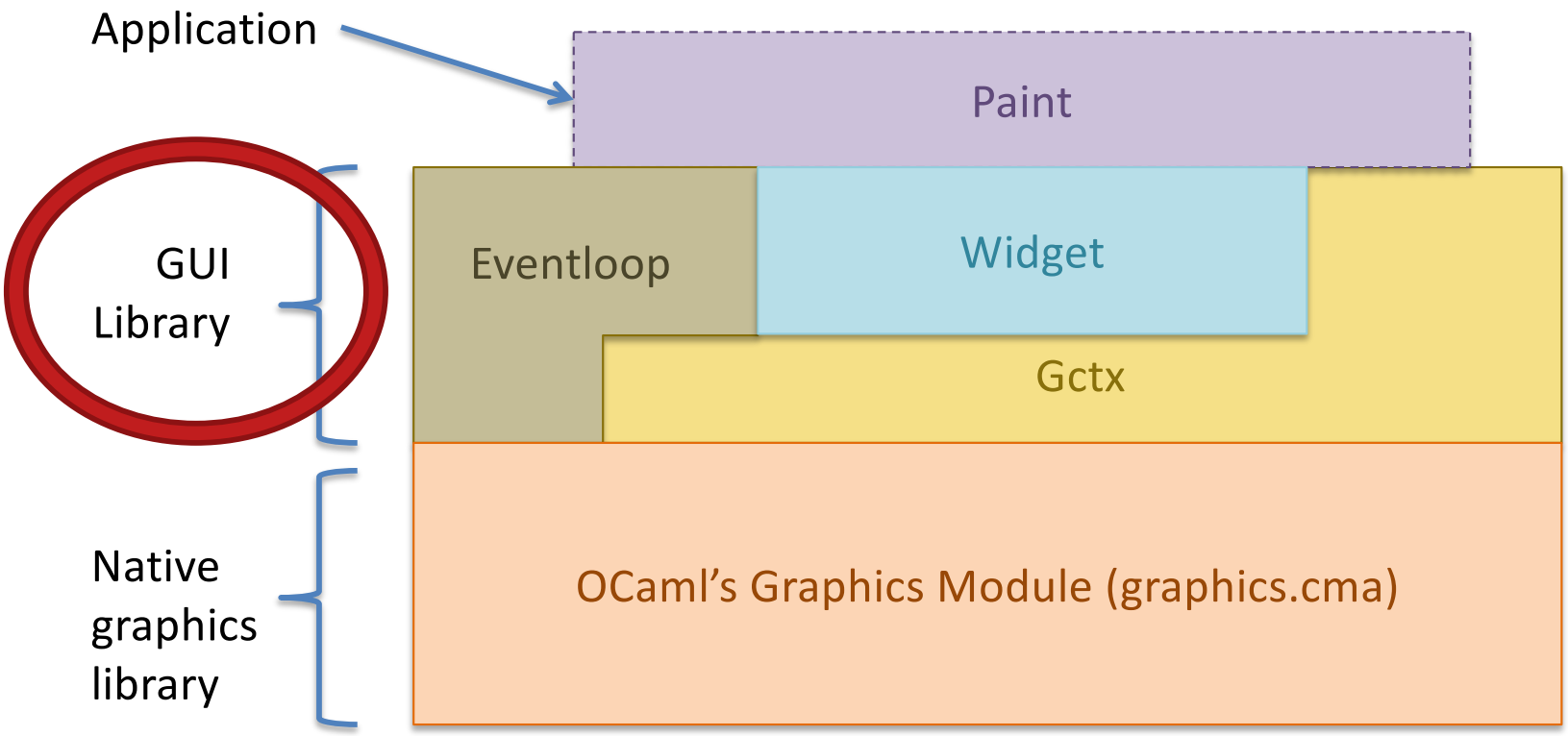
For use within the browser, we use a tool called js_of_ocaml that translates OCaml-compiled bytecode into javascript. There are some rendering differences between the native and browser versions.

GUI Library Design

Abstractions for graphical interfaces

Interfaces: Project Architecture*

*The background color of code in the following slides is color coded according to this picture.



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

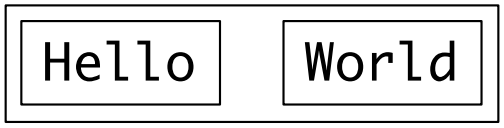
GUI terminology – Widget*

- Basic element of GUIs: examples include buttons, checkboxes, windows, textboxes, canvases, scrollbars, labels
- Every widget
  - has a size
  - knows how to display itself
  - knows how to react to events like mouse clicks

Simplified!

```
type widget = {
  repaint: unit -> unit;
  handle: event -> unit;
  size: unit -> int*int
}

val label  : string -> widget
val border : widget -> widget
val hpair  : widget -> widget -> widget
```

- May be composed of other sub-widgets, for laying out complex interfaces

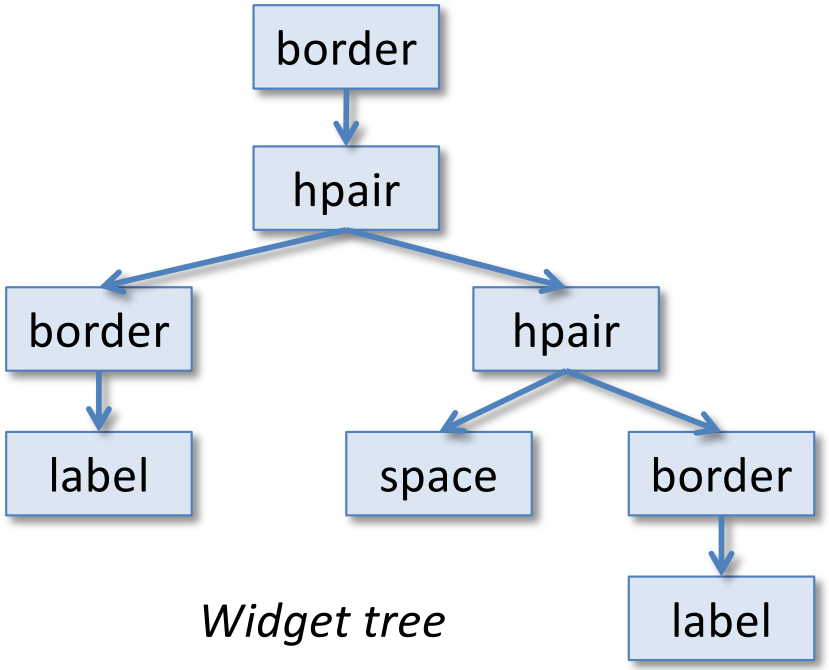


*Each GUI library uses its own naming convention for what we call "widgets." Java Swing calls them "Components"; iOS UIKit calls them "UIViews"; WINAPI, GTK+, X11's widgets, etc….

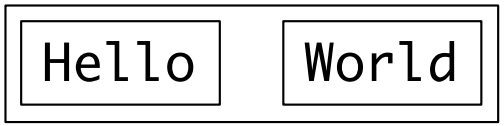
A “Hello World” application

swdemo.ml

```
(* Create some simple label (string) widgets *)
let l1 : widget = label "Hello"
let l2 : widget = label "World"
(* Compose them horizontally, adding some borders *)
let h : widget =
    border (hpair (border l1)
                  (hpair (space (10,10)) (border l2)))
```



Widget tree



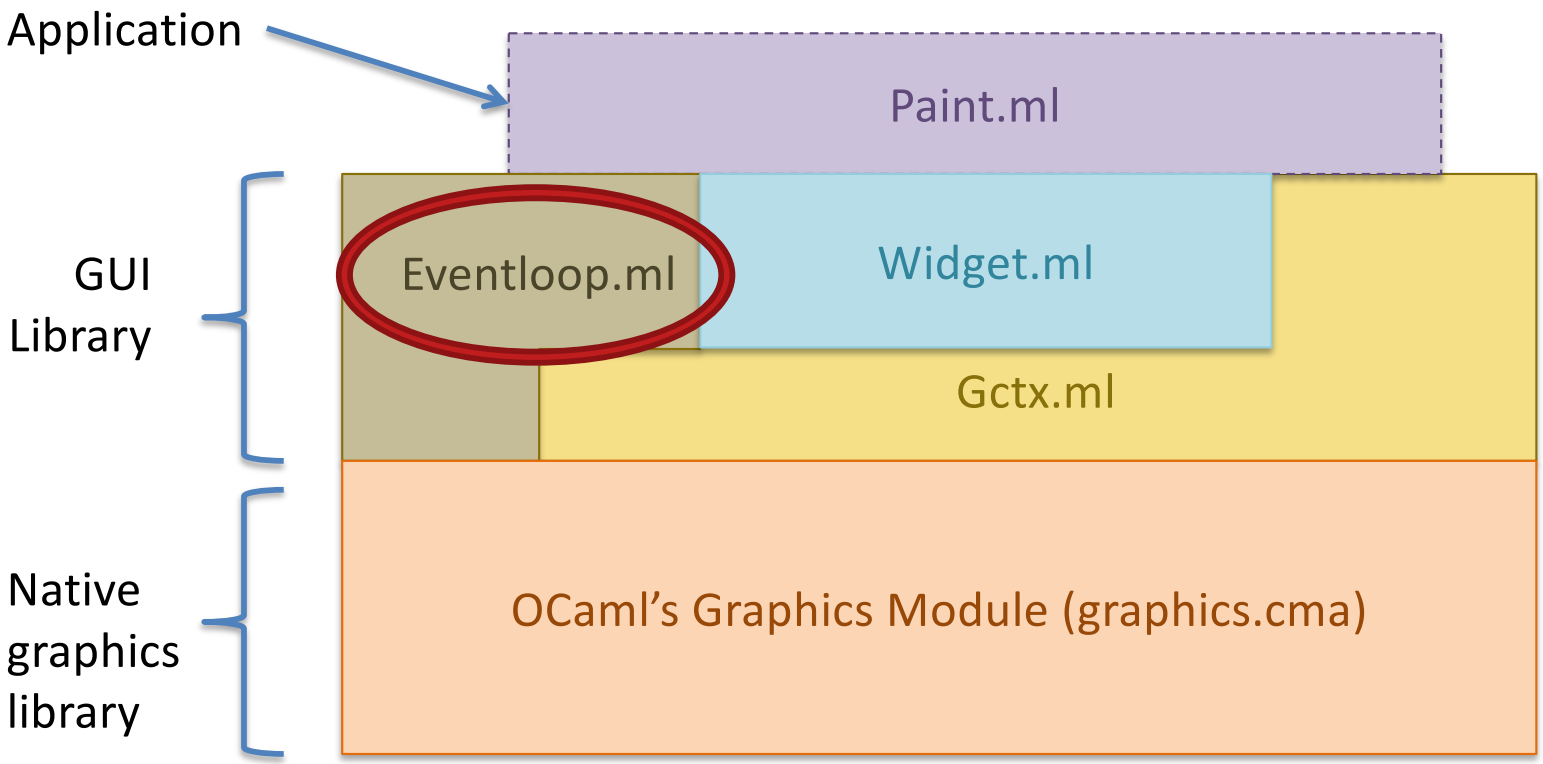
On the screen

Module: EventLoop

Top-level driver

GUI Architecture

- The eventloop is the main "driver" of a GUI application
  - For now: focus on how widgets are drawn on the screen
  - Later: deal with event handling



The event loop manages the top-level interactions and causes the display to be repainted.

GUI terminology: “event loop”

- Main loop for all GUI applications (simplified)
  - “run” function takes top-level widget w as argument, containing all other widgets in the application.

```
let run (w:widget) : unit =
  Graphics.loop                 …wait for user input (mouse click, etc)
    (fun e ->
       clear_graph ();
       w.handle e;              …inform widget about the event…
       w.repaint ()             …update the widget's appearance…
    )
```

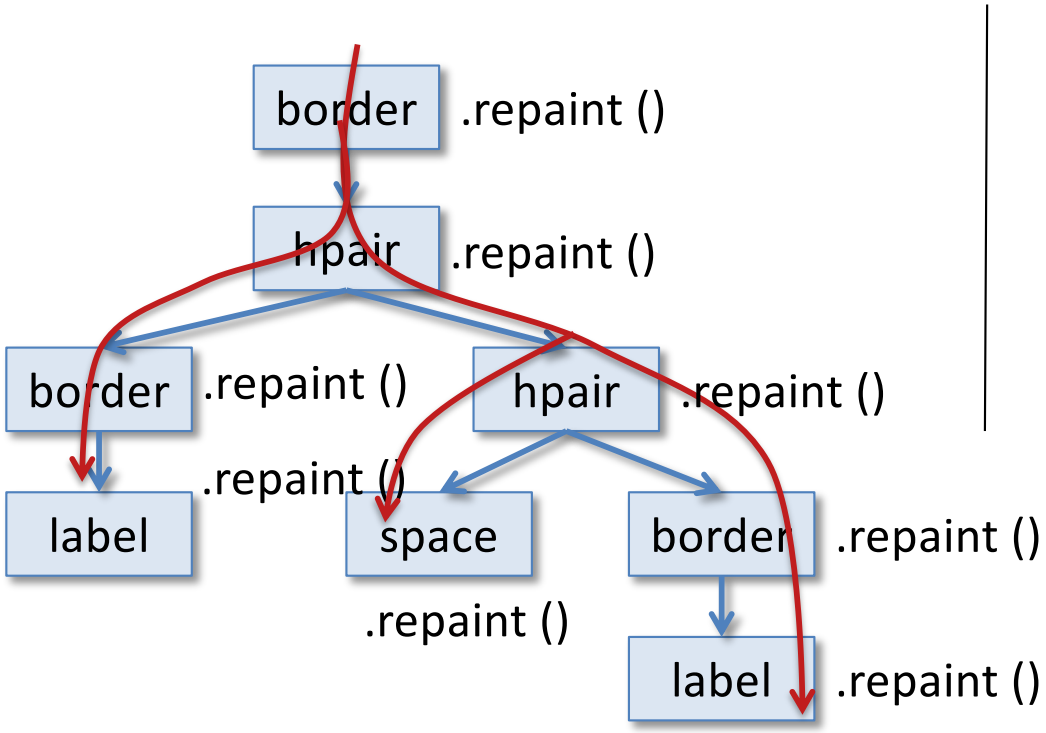
Eventloop

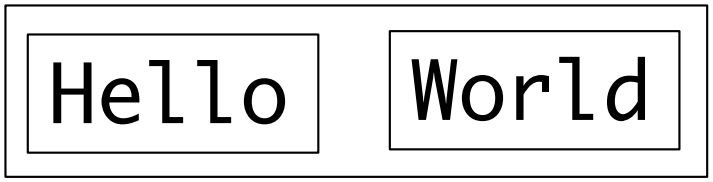
```
let rec loop (f: event -> unit) : unit =
  let e = wait_next_event () in
  f e;
  loop f
```

Graphics

Drawing: Containers

Container widgets propagate `repaint` *commands to their children:*





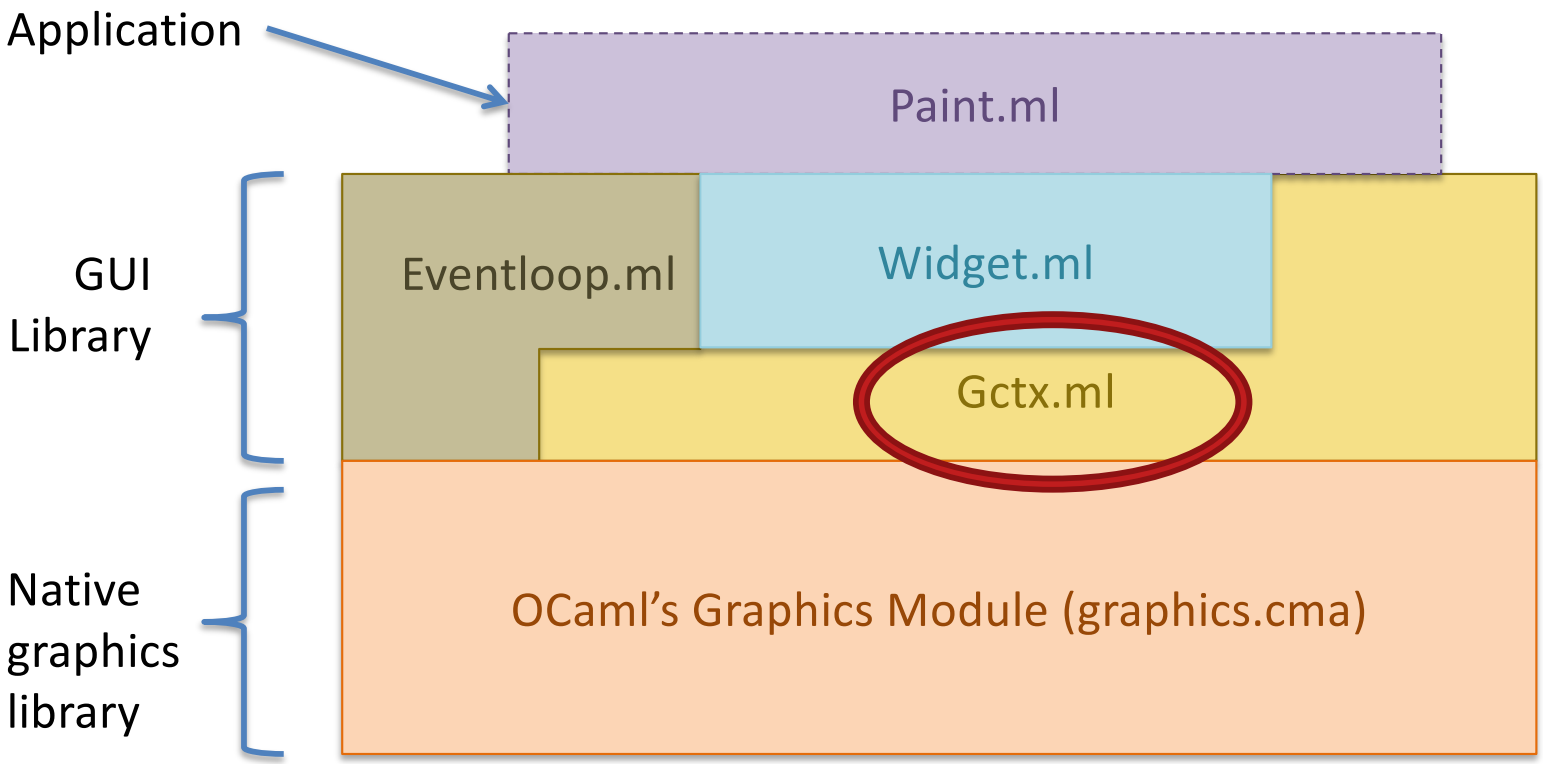
Challenge: How can we make it so that the functions that draw widgets in different places on the window are *location independent*?

Module: Gctx

“Contextualizes” graphics operations

Challenge: Widget Layout

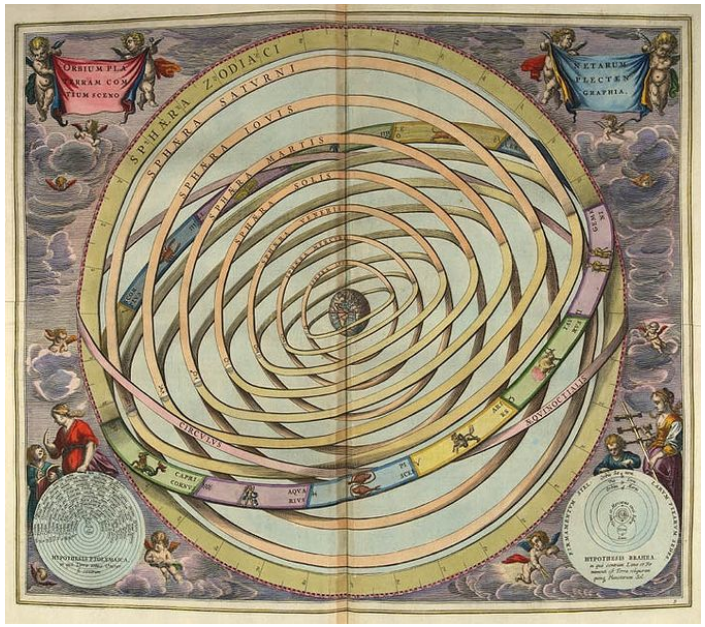
- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a *graphics context* to make drawing *relative* to a widget’s current position



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 when they repaint

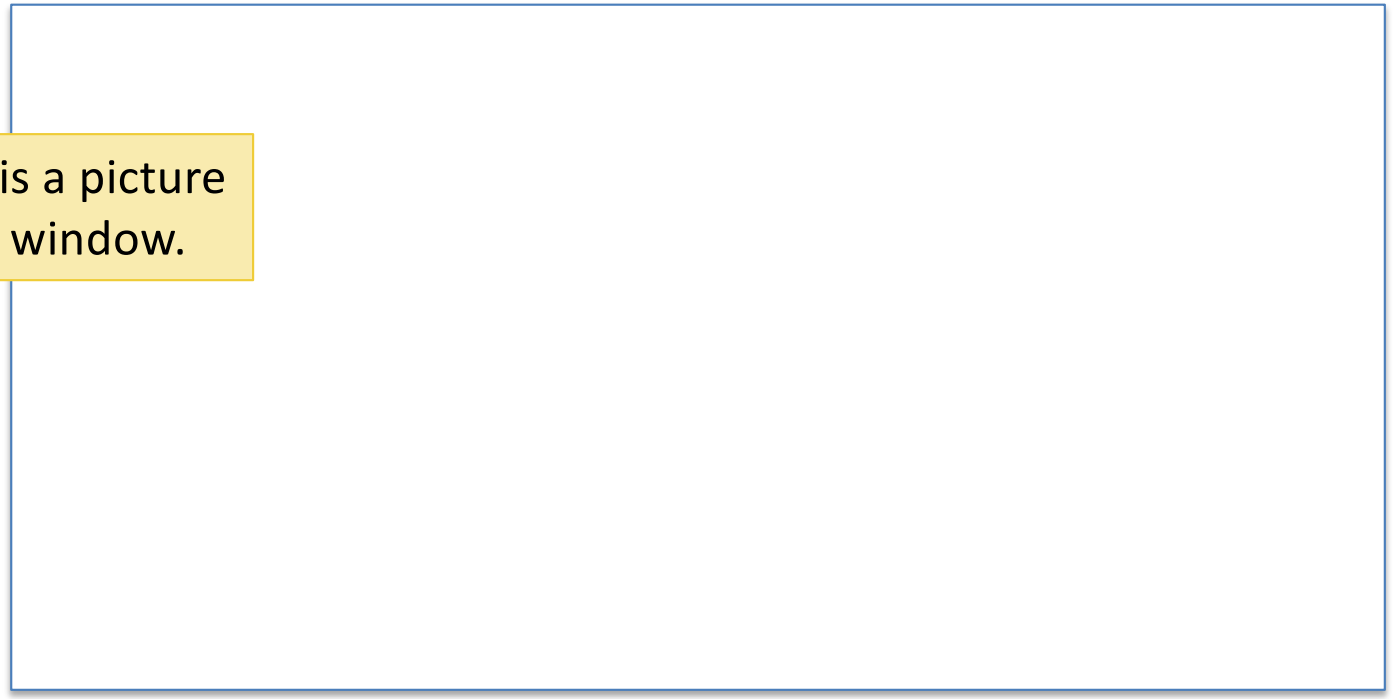


- Carries information about the way things should be drawn
  - color
  - line width
- "Task 0" in the homework helps you understand the interaction between Gctx and OCaml's Graphics module

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 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
```

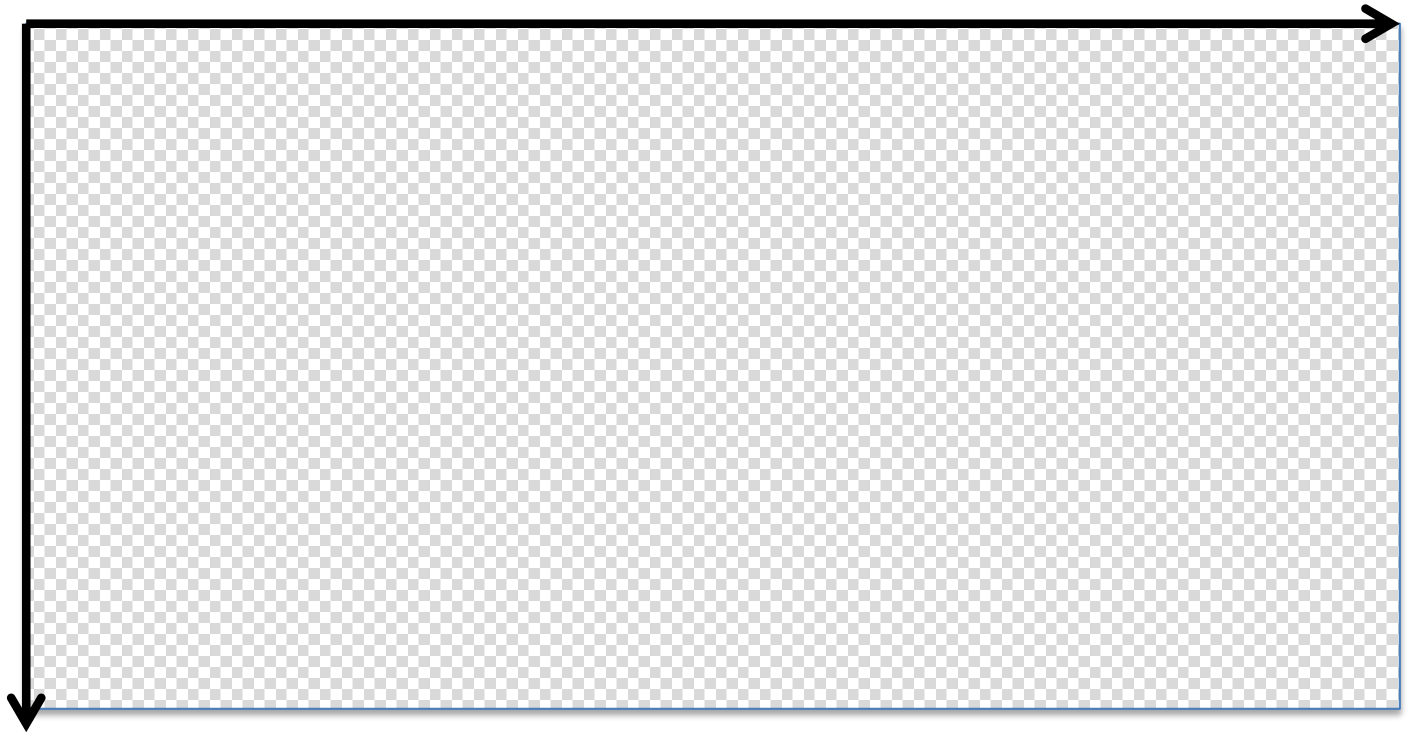
Graphics Contexts



This top box is a picture of the whole window.

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

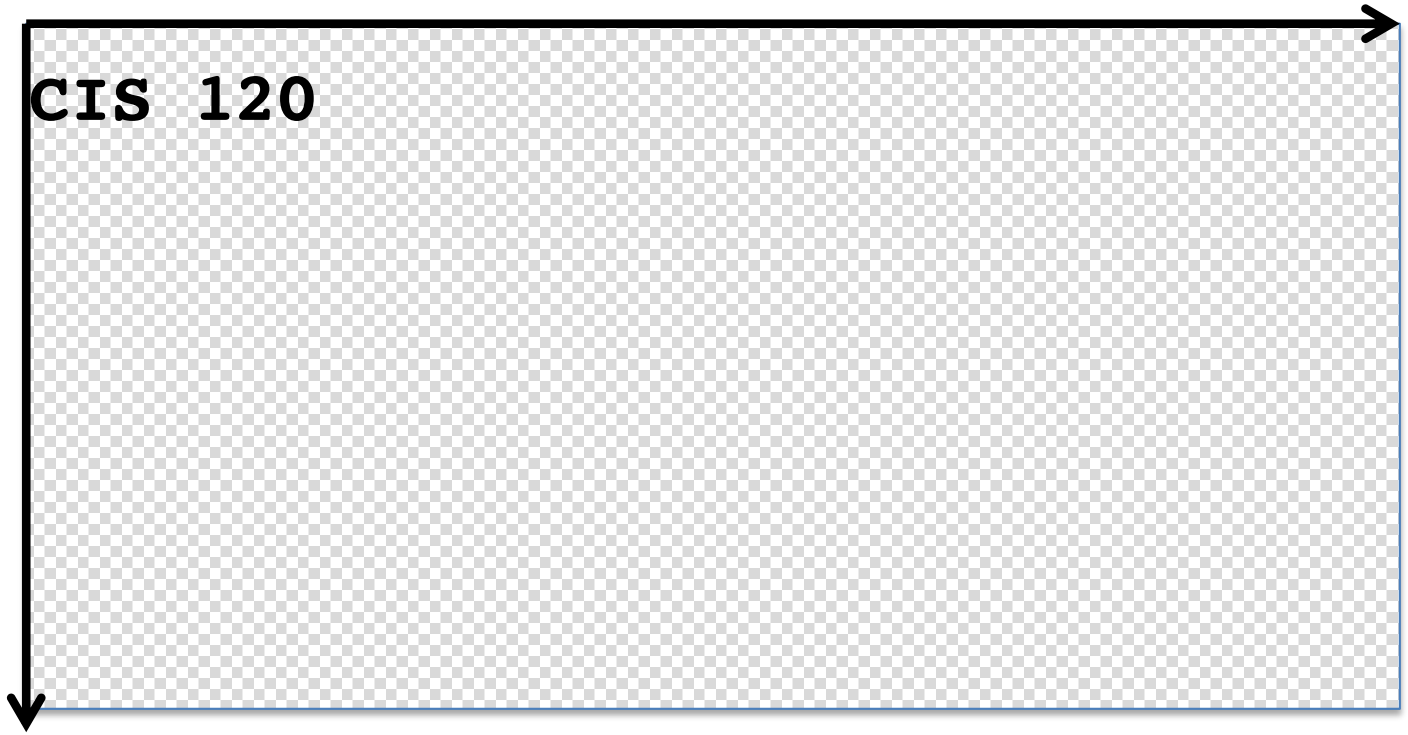
Graphics Contexts



```
let top : Gctx.gctx = Gctx.top_level
```

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

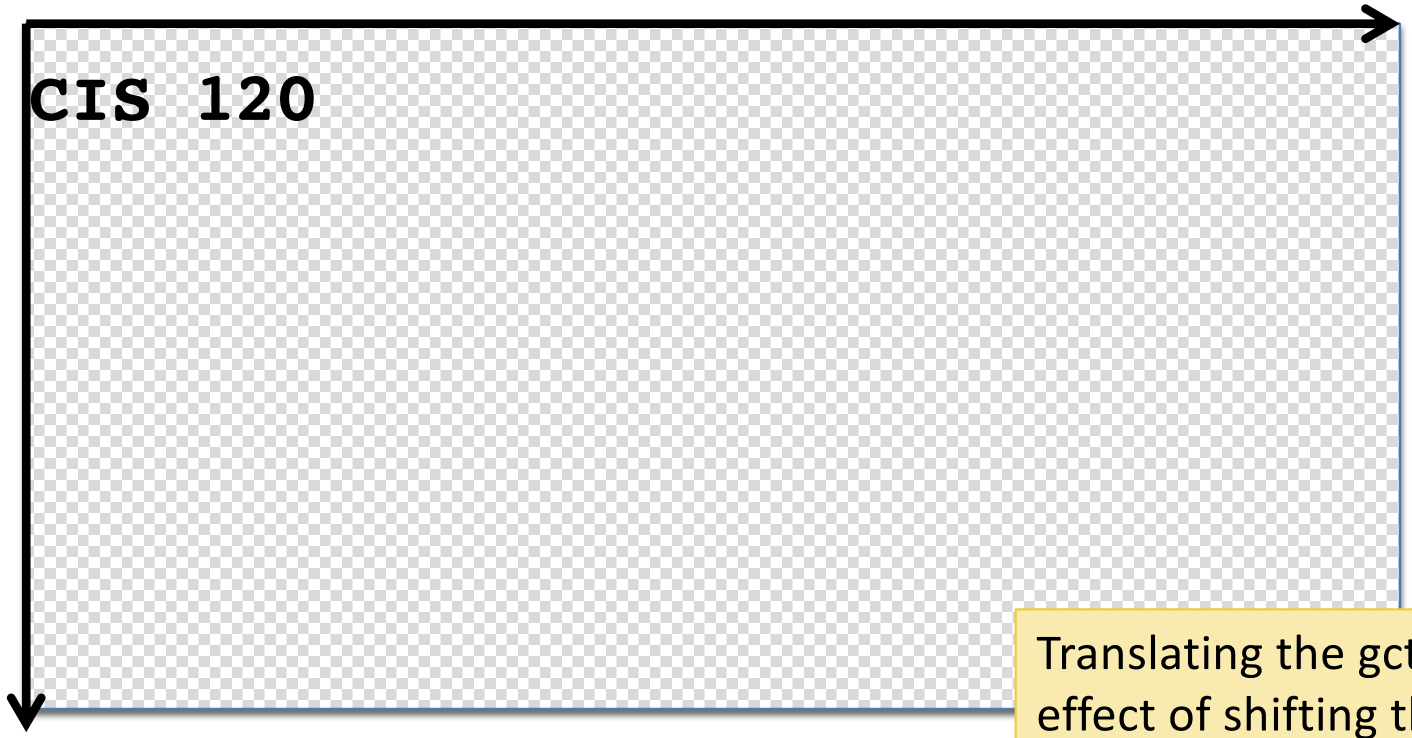
Graphics Contexts




```
let top : Gctx.gctx = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 120"
```

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.

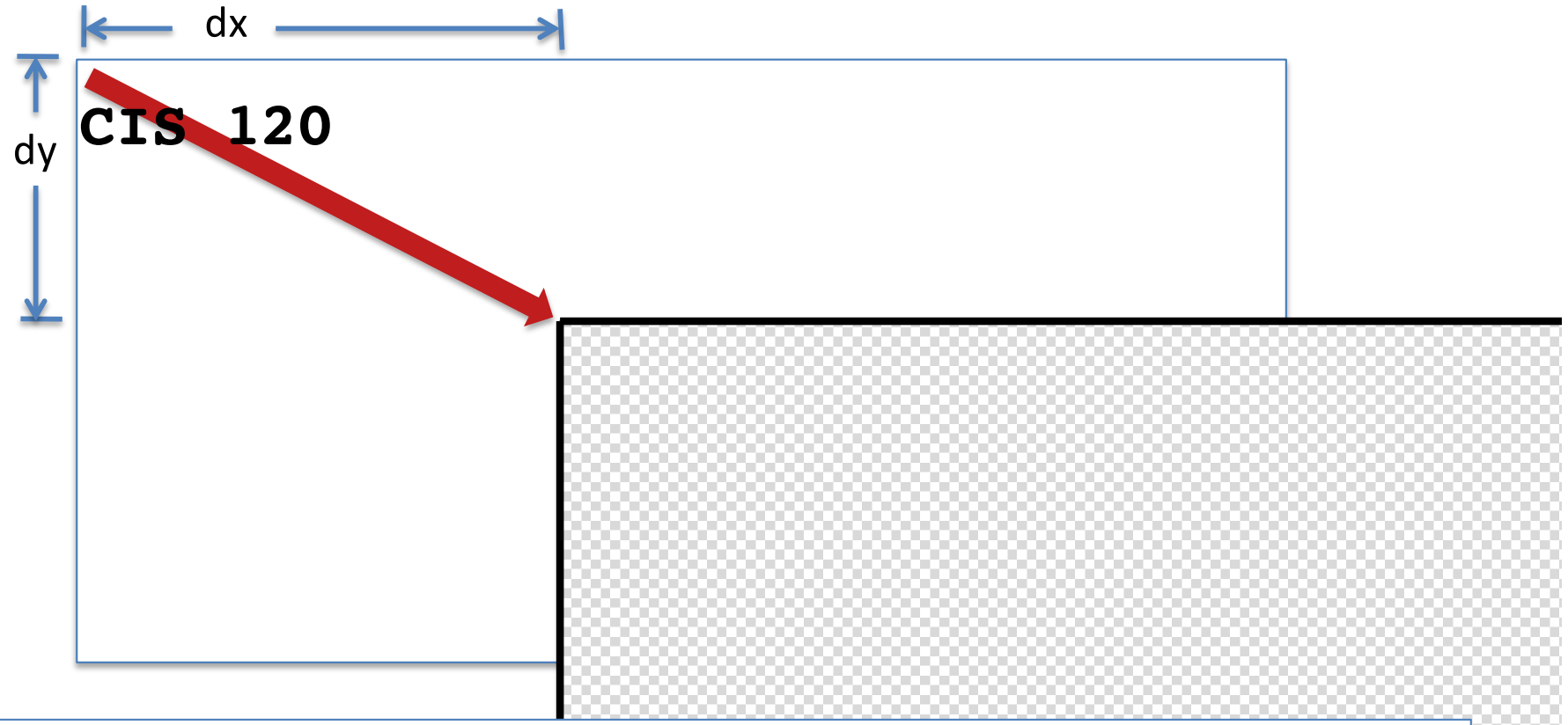
Graphics Contexts



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

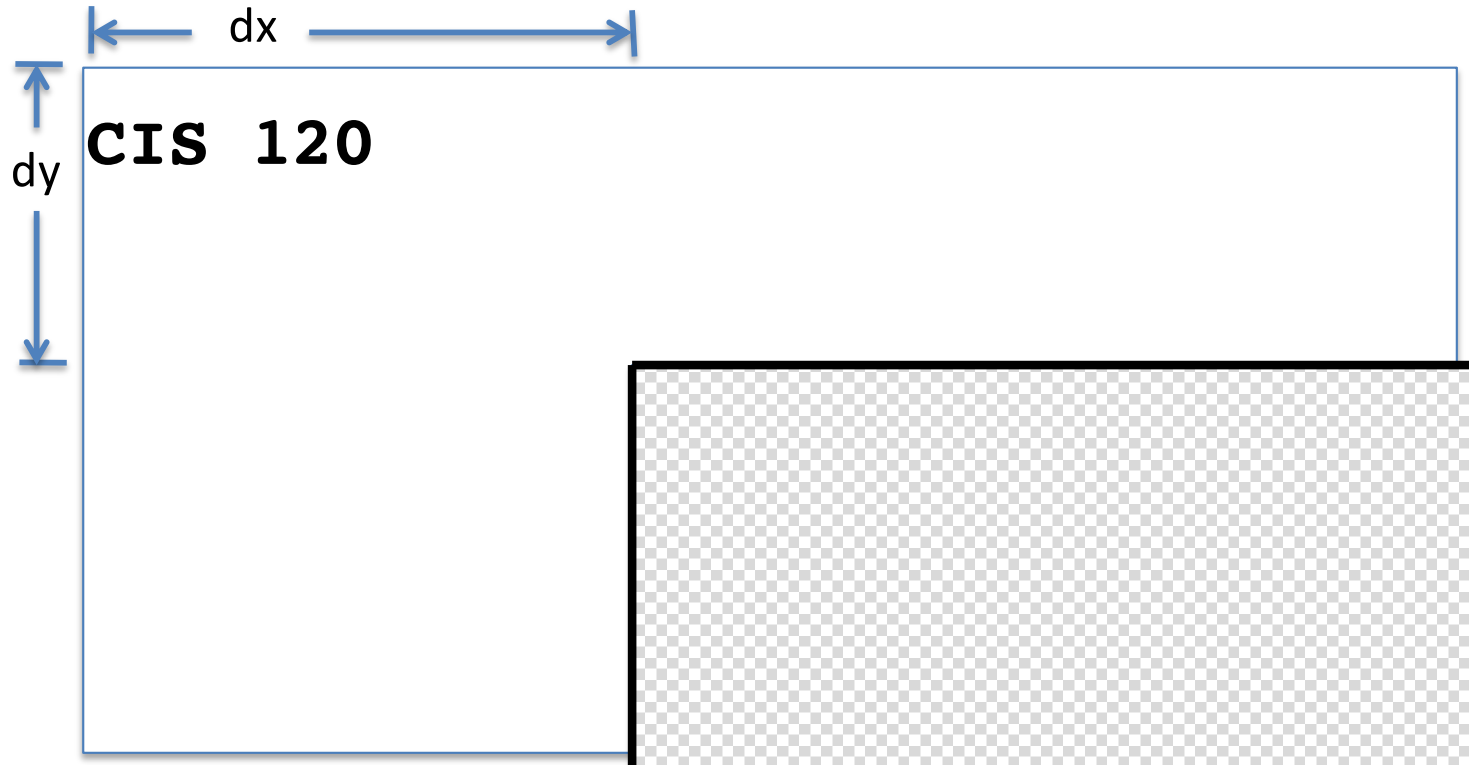
```
let top : Gctx.gctx = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 120"

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

Graphics Contexts




```
let top : Gctx.gctx = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 120"

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

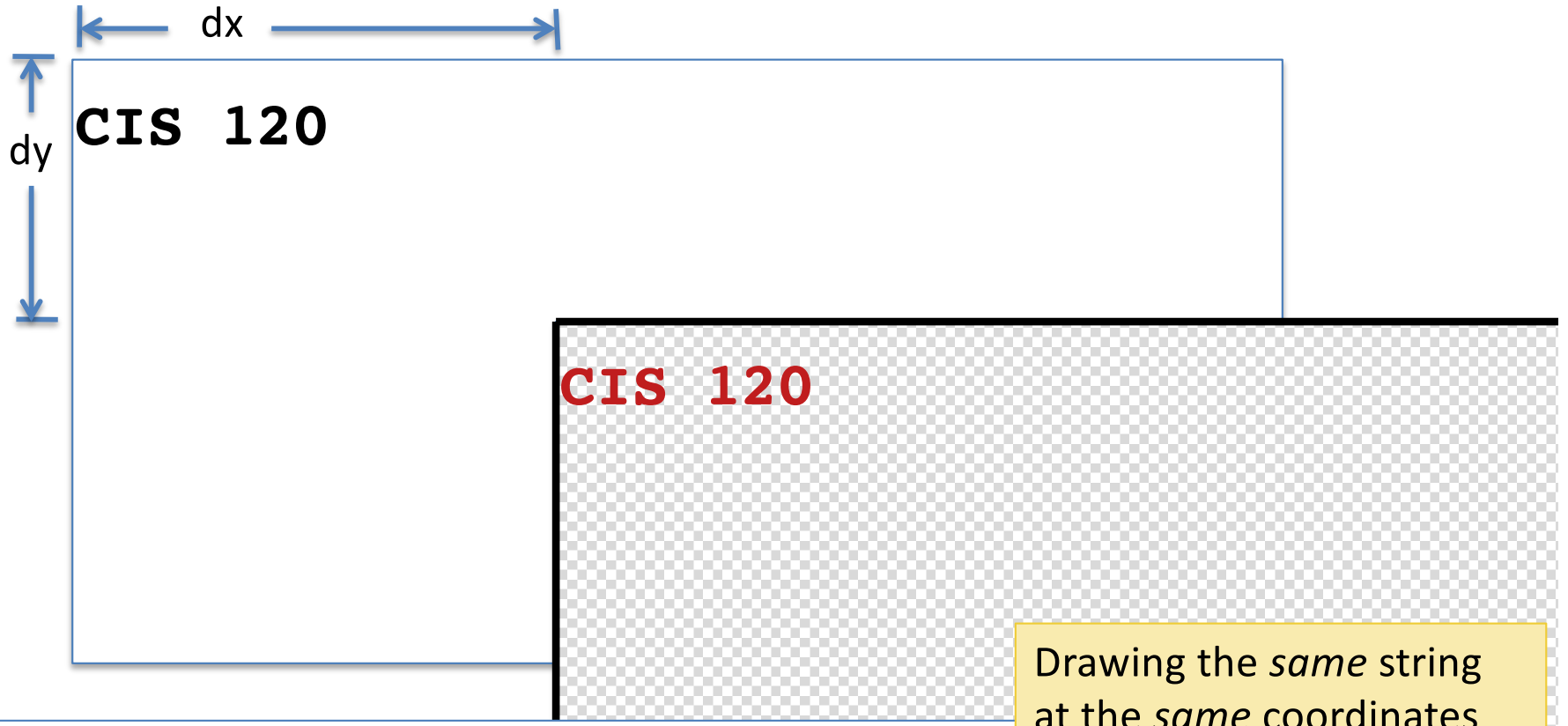
Graphics Contexts




```
let top : Gctx.gctx = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 120"

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

with_color changes the current drawing color...

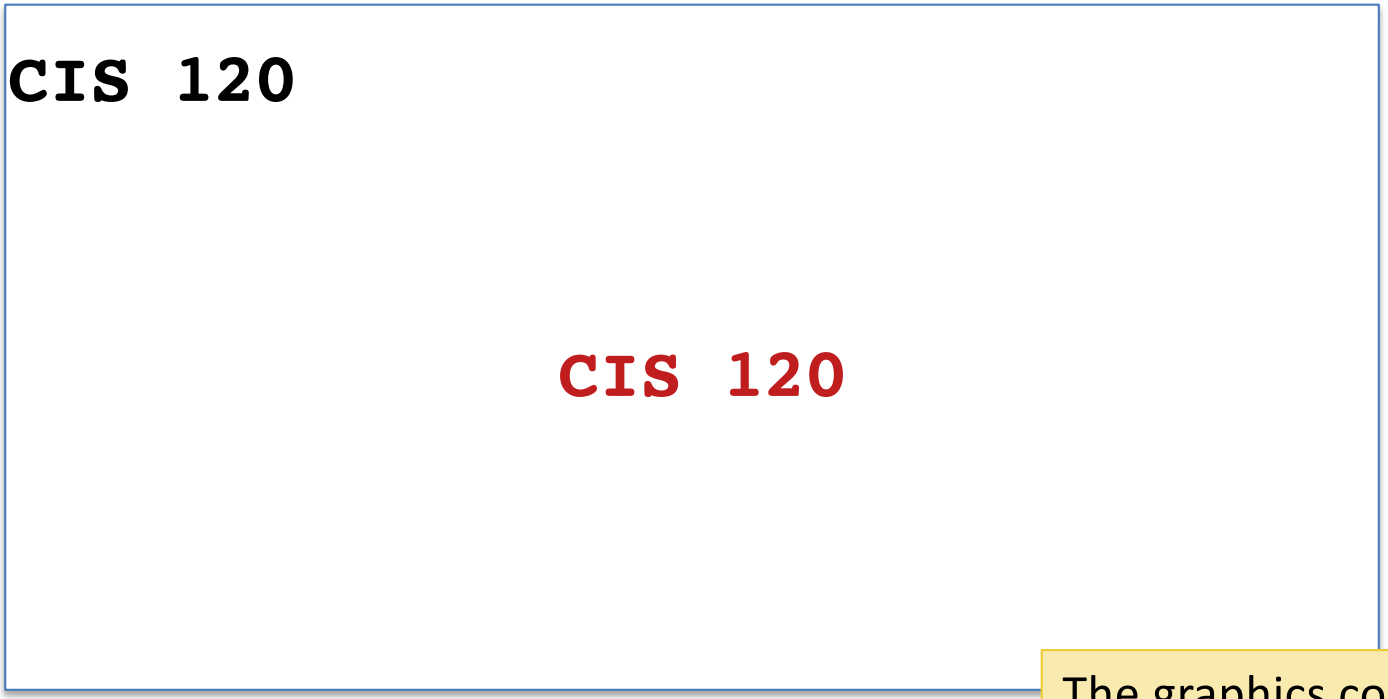
Graphics Contexts




```
let top : Gctx.gctx = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 120"

(* move origin and change the color *)
let nctx : Gctx.gctx = Gctx.with_color
                (Gctx.translate top (dx,dy)) red
;; Gctx.draw_string nctx (0,10) "CIS 120"
```

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.

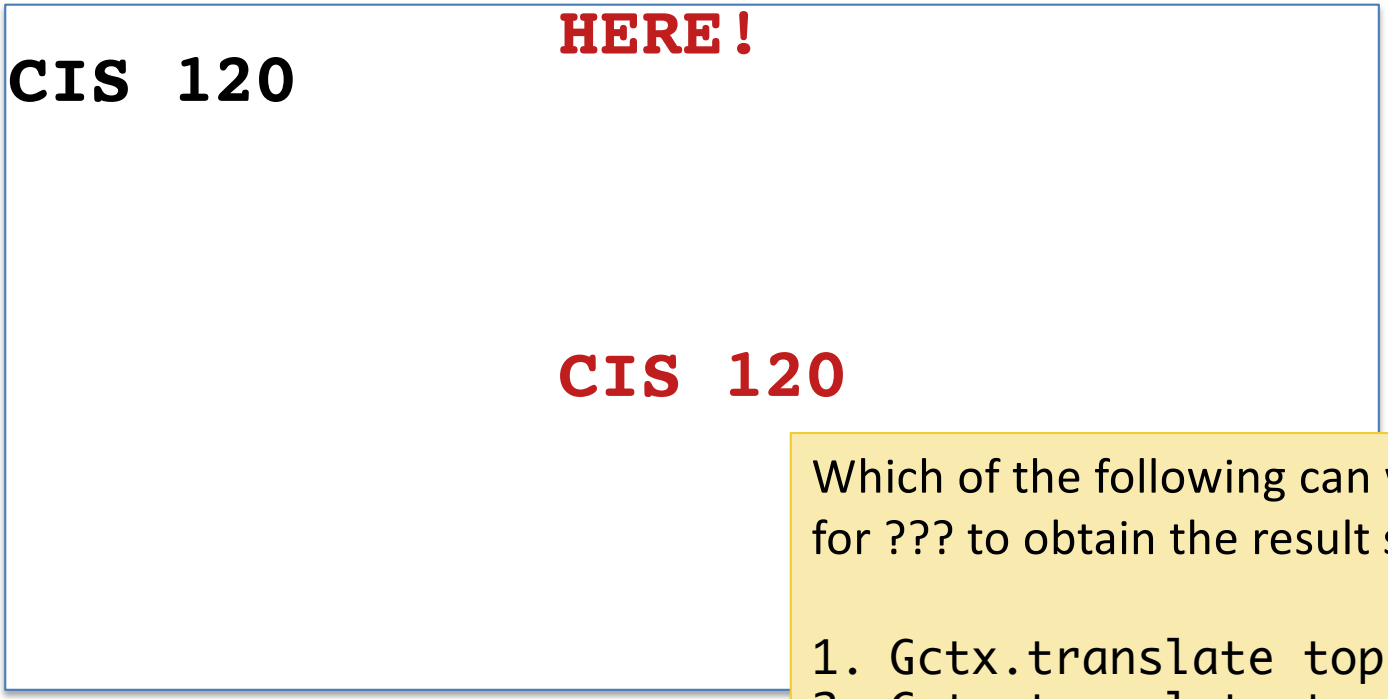
Graphics Contexts



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

```
let top : Gctx.gctx = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 120"

(* move origin and change the color *)
let nctx : Gctx.gctx = Gctx.with_color
                (Gctx.translate top (dx,dy)) red
;; Gctx.draw_string nctx (0,10) "CIS 120"
```

Graphics Contexts



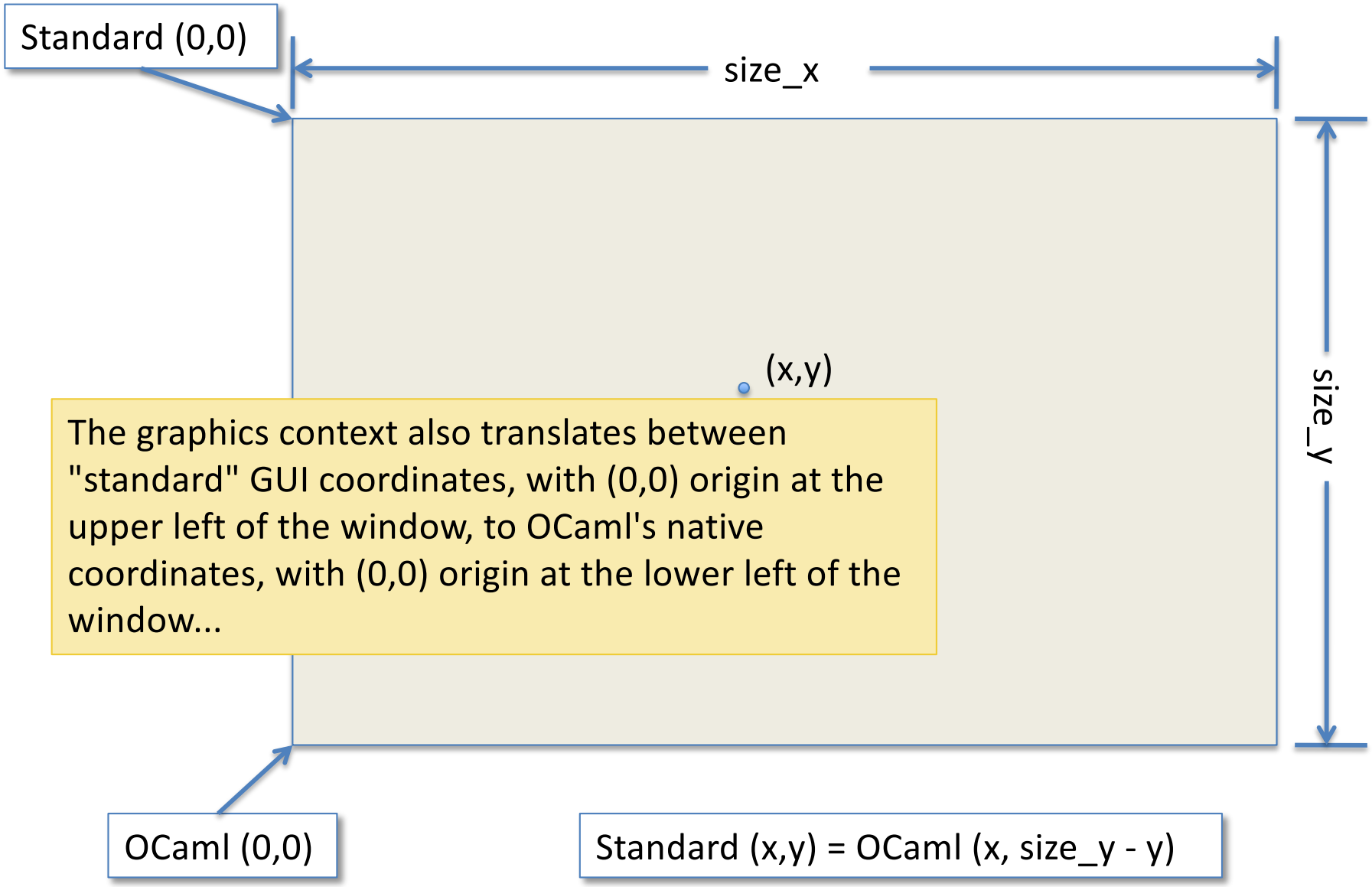

```
let top  = Gctx.top_level
;; Gctx.draw_string top  (0,10) "CIS 120"
let nctx = Gctx.with_color
               (Gctx.translate top (dx,dy)) red
;; Gctx.draw_string nctx (0,10) "CIS 120"
let ctx3 = ???
;; Gctx.draw_string ctx3 (0,0) "HERE!"
```

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



The (real) `widget` type

Recall: A *widget* is an object with three methods…

1. it can *repaint* itself (given an appropriate graphics context)
2. it can handle *events*
3. it knows its current *size*

```
type widget = {
  repaint: Gctx.gctx -> unit;
  handle: Gctx.gctx -> Gctx.event -> unit;
  size: unit -> Gctx.dimension
}
```

Event loop with graphics context

```
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

…inform widget about the event…
…update the widget's appearance…

Eventloop

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

Graphics

Widget Layout

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
  - Repainting is relative to a graphics context
- You can ask a simple widget to tell you its size
- (For now, we ignore event handling...)

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

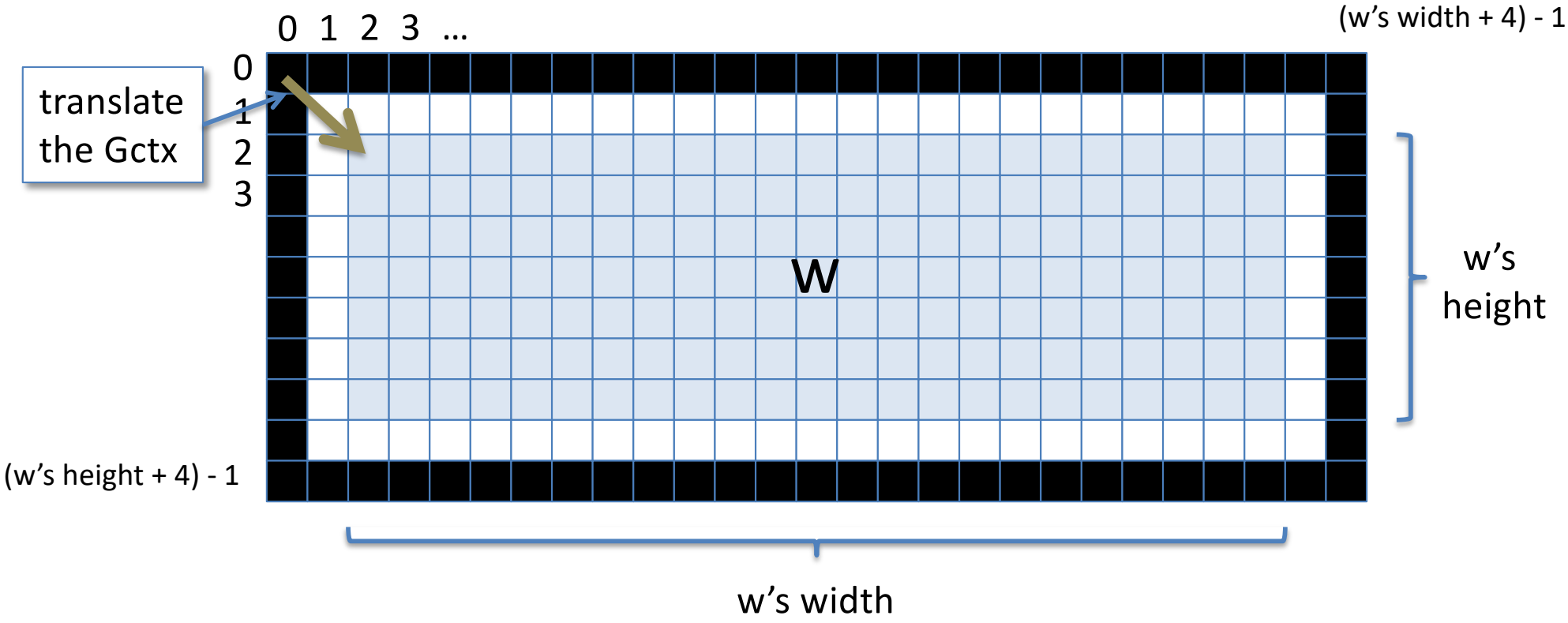
```
let canvas ((w,h):int*int) (r: Gctx.gctx -> unit) : widget =
{
  repaint = r;
  size = (fun () -> (w,h))
}
```

simpleWidget.ml

Nested Widgets

Containers and Composition

The Border Widget Container



- `let b = border w`
- Draws a one-pixel wide border 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

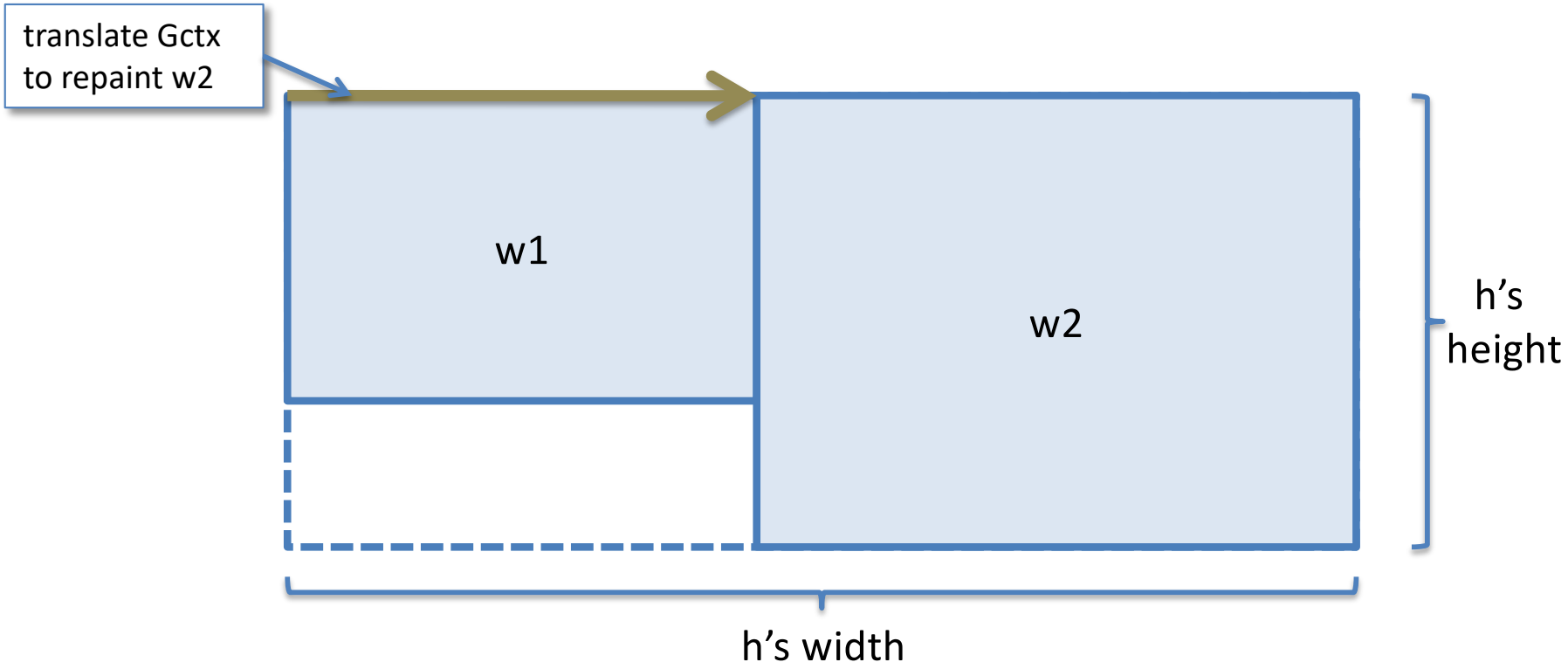
simpleWidget.ml

```
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 w2
- Size is the *sum* of their widths and *max* of their heights

The hpair Widget

simpleWidget.ml

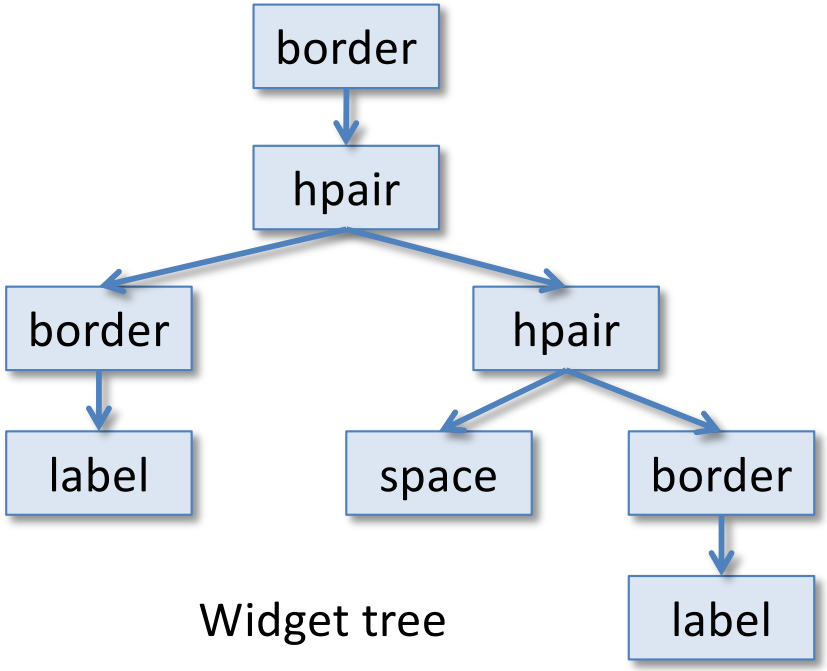
```
let hpair (w1: widget) (w2: widget) : widget =
  {
    repaint = (fun (g: Gctx.gctx) ->
            let (x1, _) = w1.size () in begin
              w1.repaint g;
              w2.repaint (Gctx.translate g (x1,0))
              (* Note translation of the Gctx *)
            end);

    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.

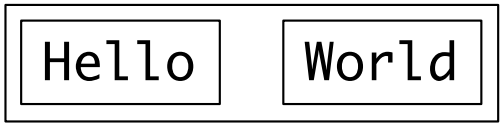
Widget Hierarchy Pictorially

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)))
```



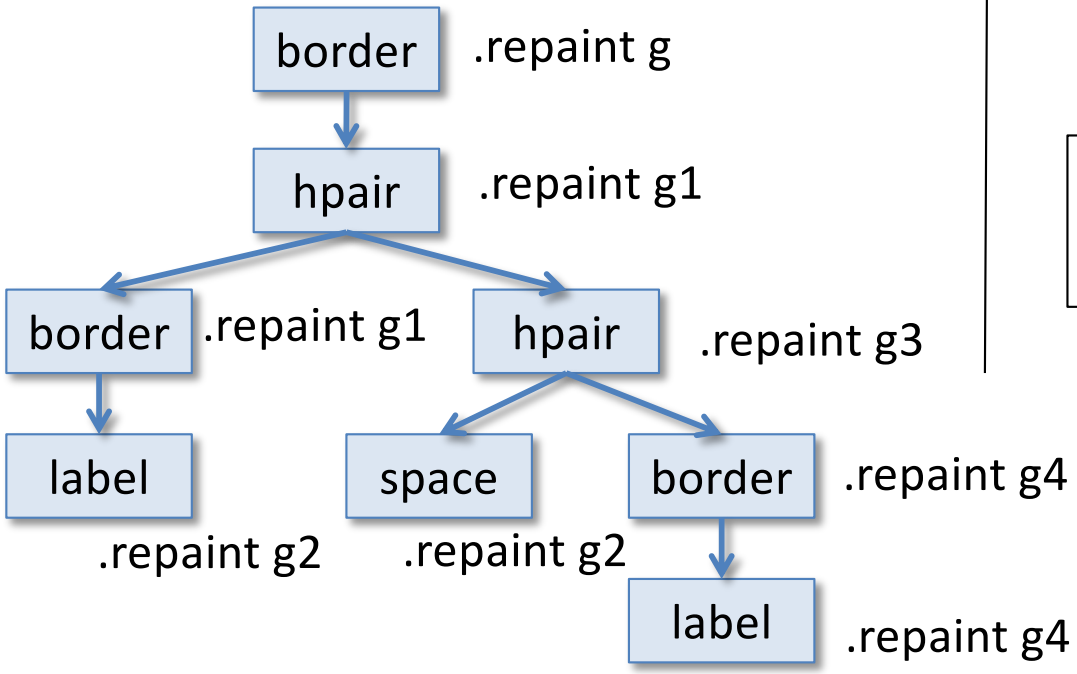
Widget tree

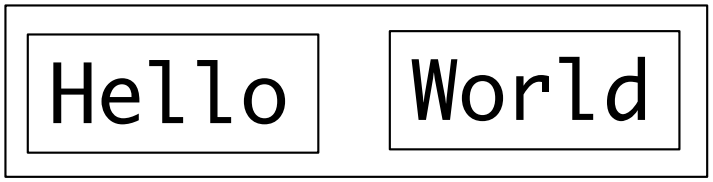


On the screen

Drawing: Containers

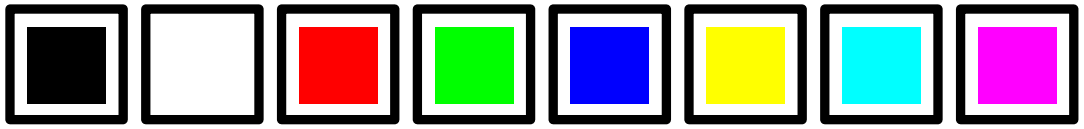
Container widgets propagate repaint commands to their children, with appropriately modified graphics contexts:






```
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)
```

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 (using hpair).