# Programming Languages and Techniques (CIS120)

Lecture 17

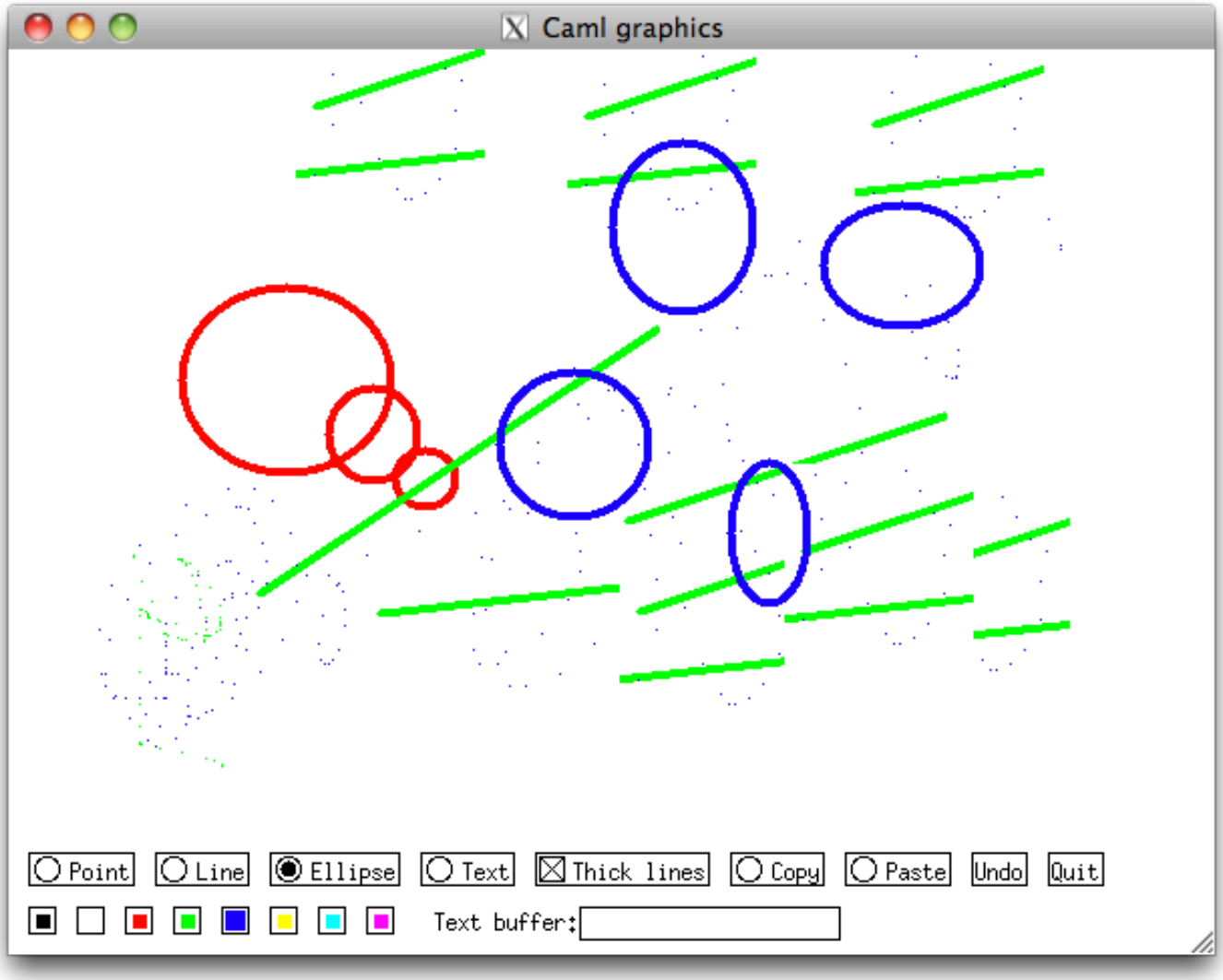
Feb 22, 2012

## GUI Design II: Layout

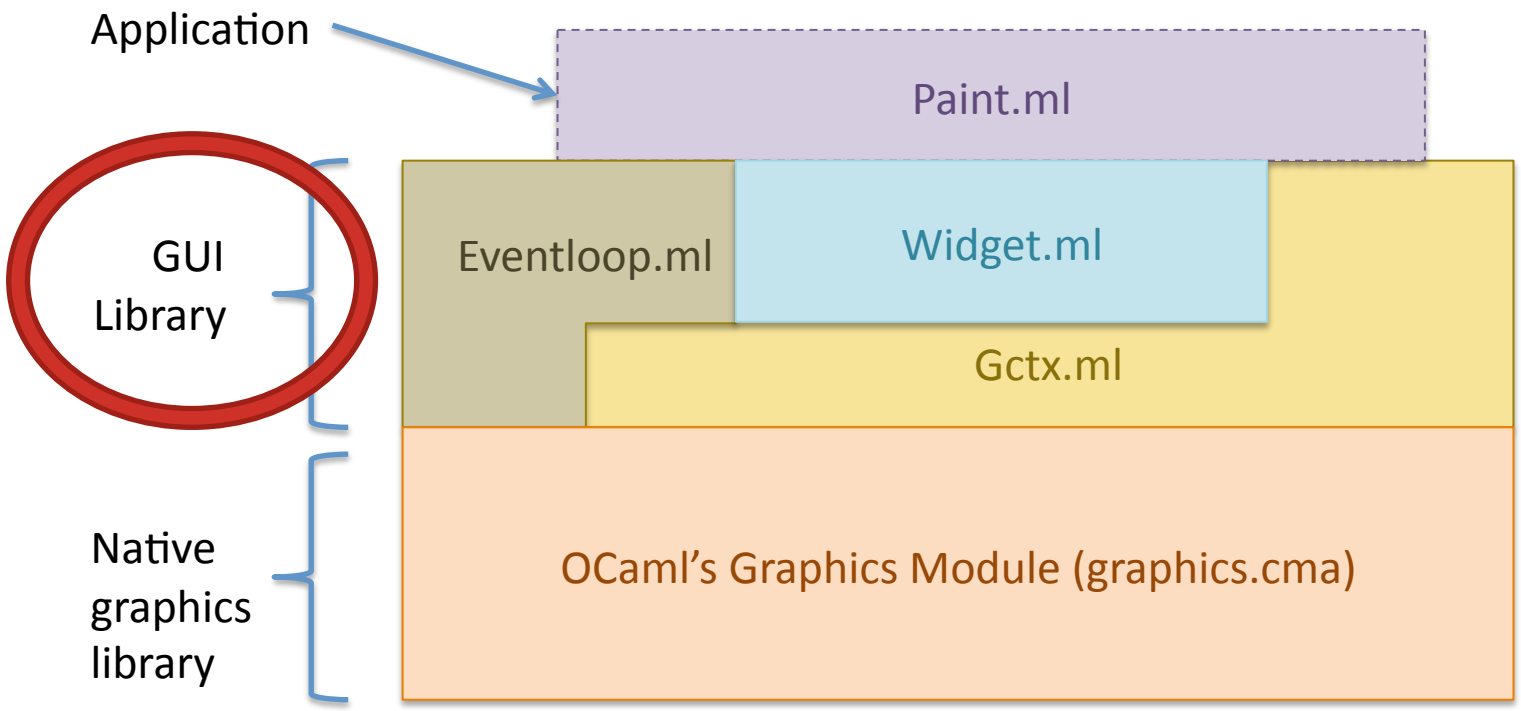
# Announcements

- Weirich Office hours today, 3:30-5PM in Levine 510
- Lab today: Graphing Calculator & Midsemester survey
- HW06: Building a GUI from scratch
  - Will be available this afternoon.
  - Is officially due *Thursday,* Mar 1 at 11:59:59pm
  - … but grace period until Friday, Mar 2 at 11:59:59pm (free late day)
  - NOTE: TAs will not be available after the due date

# Building a GUI and GUI Applications

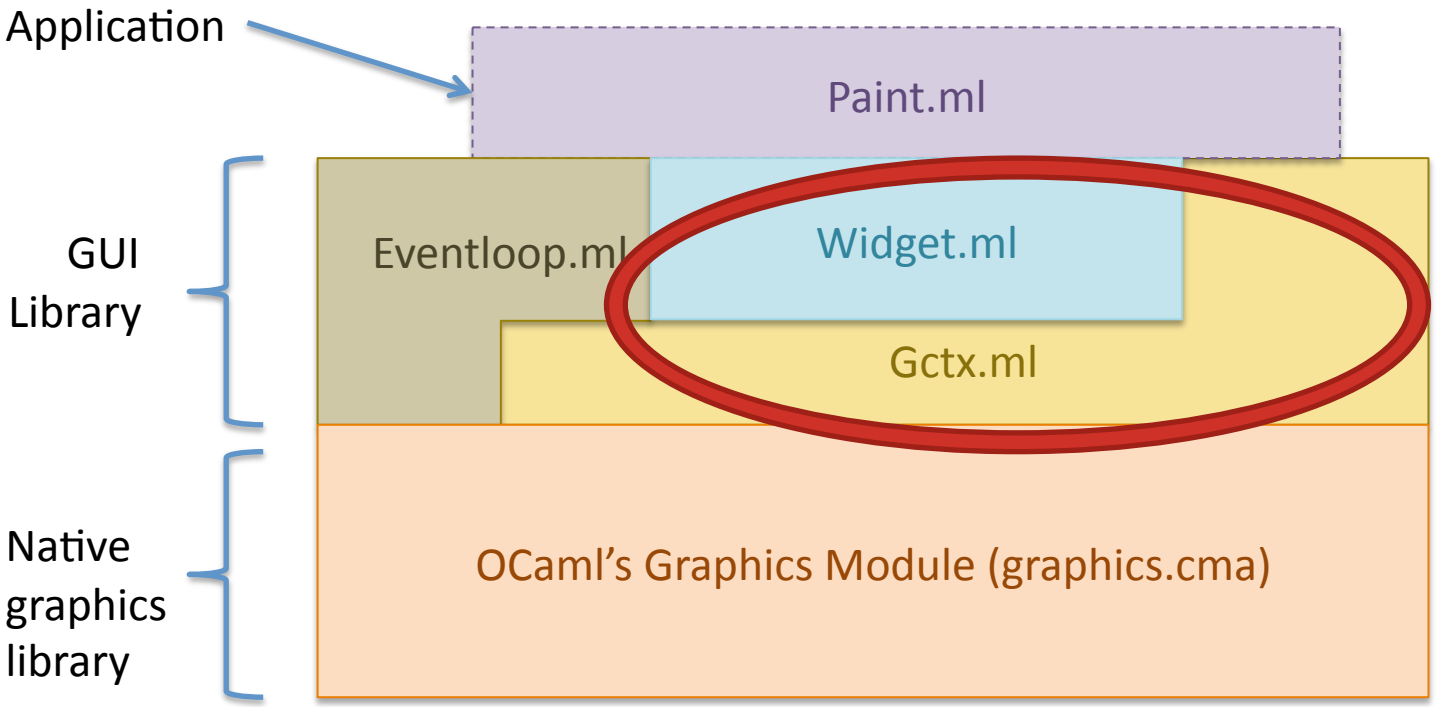


# Project Architecture



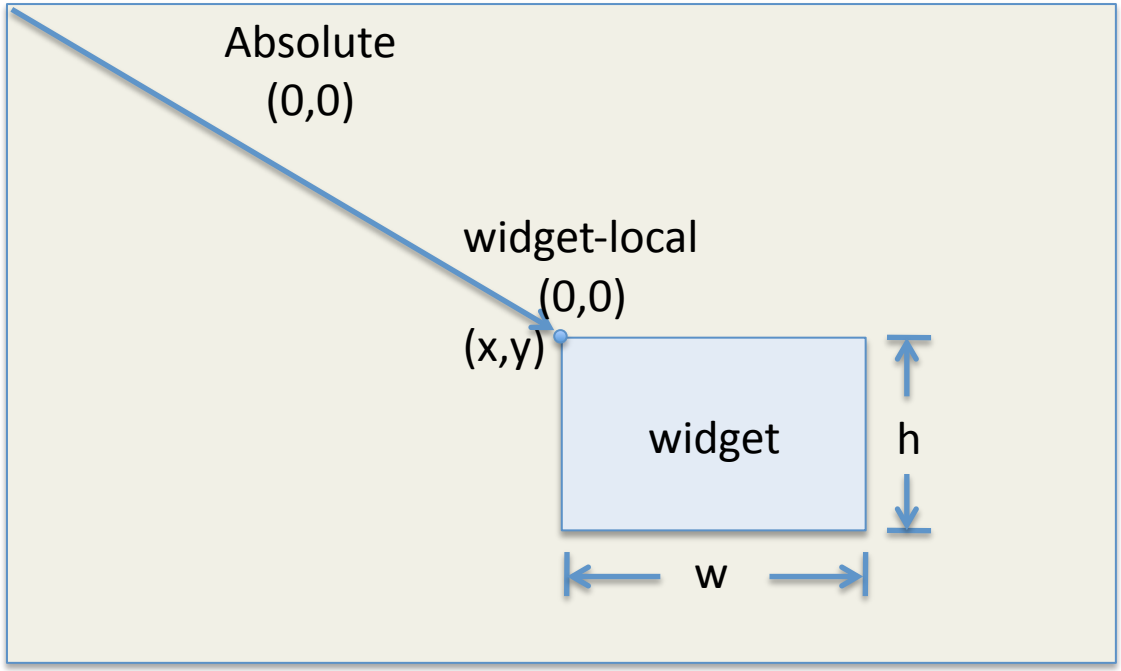
# Challenge 1: Widget Layout

- 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.



# Graphics Contexts

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).




All drawing that is done by the widget should be done using the graphics context operations (i.e. Gctx.draw_rect), instead of using the native graphics library (i.e. Graphics.draw_rect) so that the widget is location independent.

# Gctx.mli (excerpt)

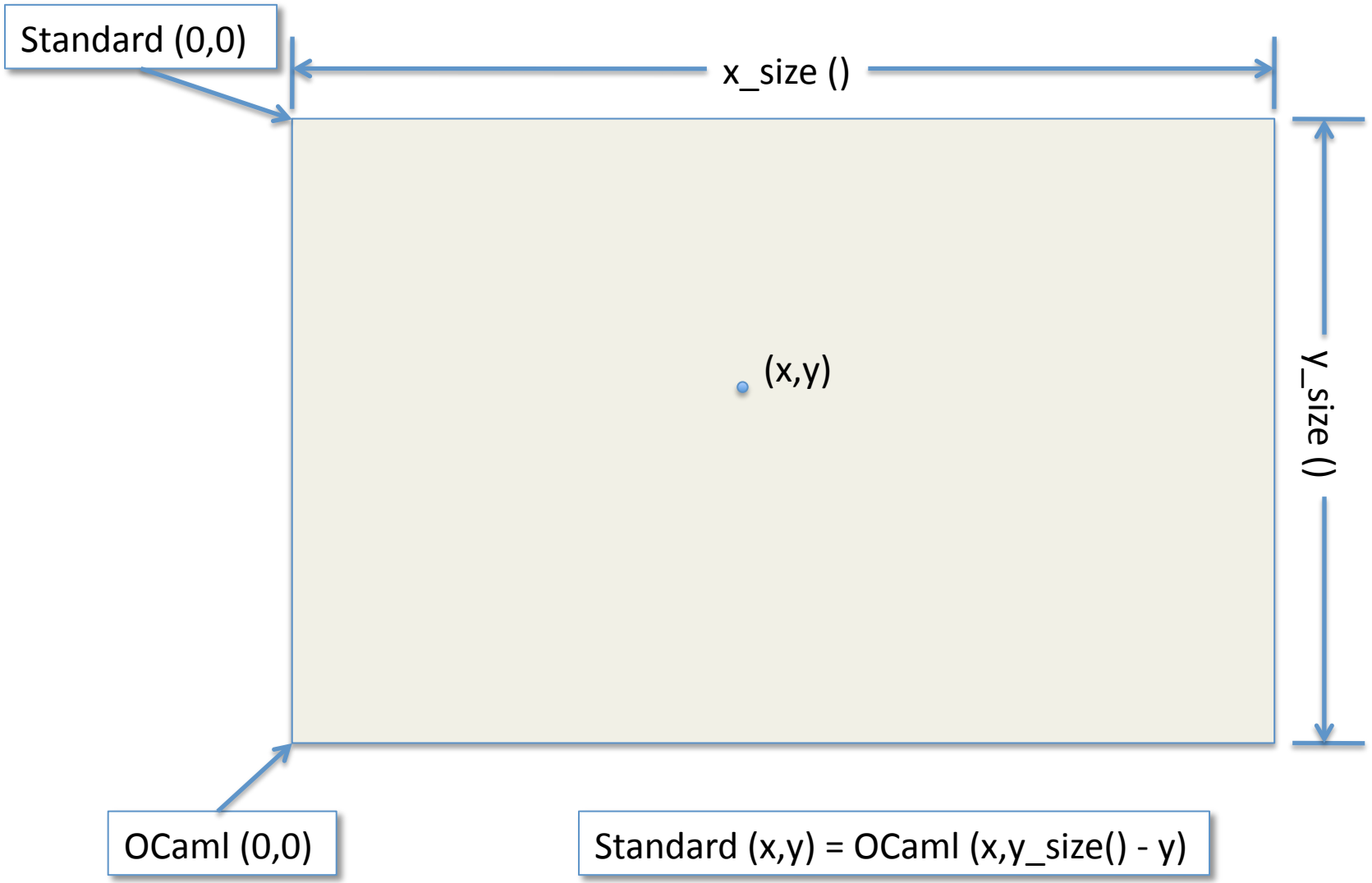
```
type t  (** The main (abstract) type of graphics contexts *)

(** Creates a fresh Gctx.t *)
val create : unit -> t
(** Produce a new Gctx.t shifted  by (dx,dy) *)
val translate : t -> int * int -> t

(** A widget-relative position *)
type position = int * int
(** A width and height paired together. *)
type dimension = int * int

(** Display text at the given (widget-local) position *)
val draw_string : t -> position -> string -> unit
(** Calculates the size of a text when rendered. *)
val text_size : t -> string -> dimension
```

# OCaml vs. Standard Coordinates



# Simple Widgets

Building Widgets up from scratch

# Simple Widgets

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

- 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

Note: don't confuse this type with the type named 't' in the Gctx module. In the paint application, we'll call the type of (simple) widgets "(Simple)Widget.t" and the type of graphics contexts "Gctx.t." In their respective modules, they are the type of interest, so there they are just t.

# Simple widgets (SimpleWidget.mli)

```
(* Some text on the screen *)
val label  : string -> t

(* Empty space *)
val space  : Gctx.dimension -> t

(* Some space parameterized by a draw fcn *)
val canvas : Gctx.dimension -> (Gctx.t -> unit) -> t

(* Adds a border around another widget *)
val border : t -> t

(* Put two widgets next to each other *)
val hpair : t -> t -> t
```

Demo: swdemo.ml

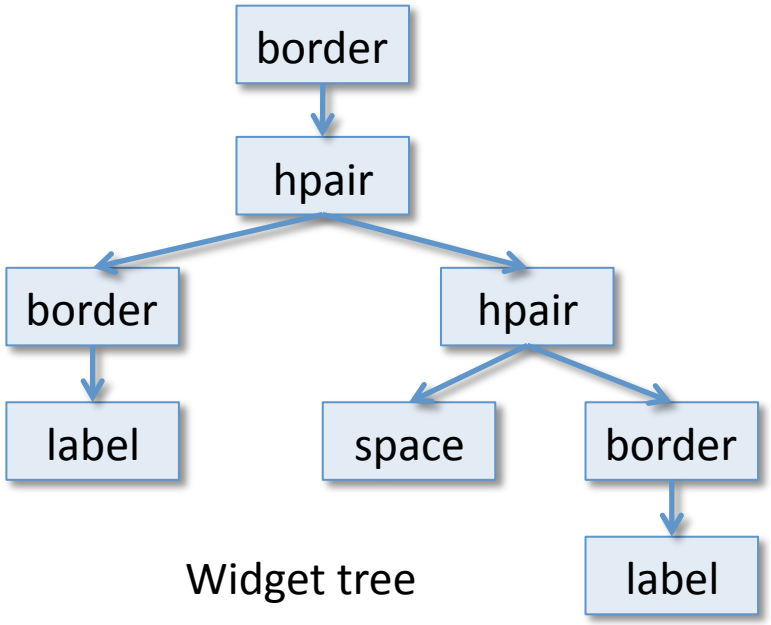
# The Widget Hierarchy

- Widgets 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

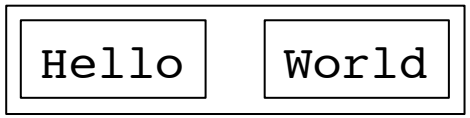
*If you draw the state of the abstract machine for a widget program, the tree will be visible in the heap – the saved stack of the "repaint" function for a container widget will contain references to its children.

# Widget Hierarchy Pictorially

```
(* 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

# Implementing the Widgets

# Implementing the Widgets

simpleWidget.ml

```
(* Display a string on the screen. *)
let label (s:string) : t =
{
  repaint = (fun (g:Gctx.t) ->
                          Gctx.draw_string g (0,0) s);
  size    = (fun (g:Gctx.t) -> Gctx.text_size g s)
}
```

simpleWidget.ml

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

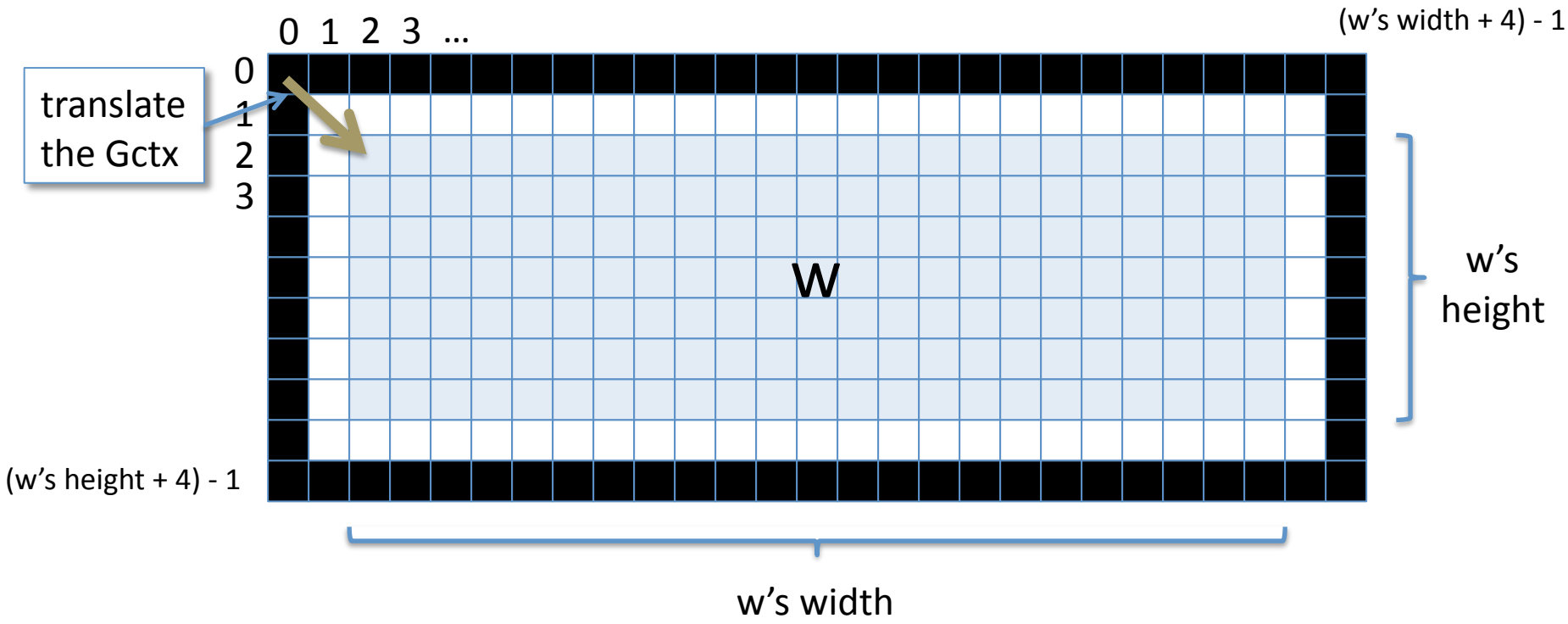
# The canvas Widget

- Region of the screen that can be drawn upon
- Has a fixed width and height
- Parameterized by a repaint method

simpleWidget.ml

```
(* some space with a draw fcn *)
let canvas ((w,h): Gctx.dimension)
           (repaint_fun: Gctx.t -> unit) : t =
  {
    repaint = repaint_fun;
    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

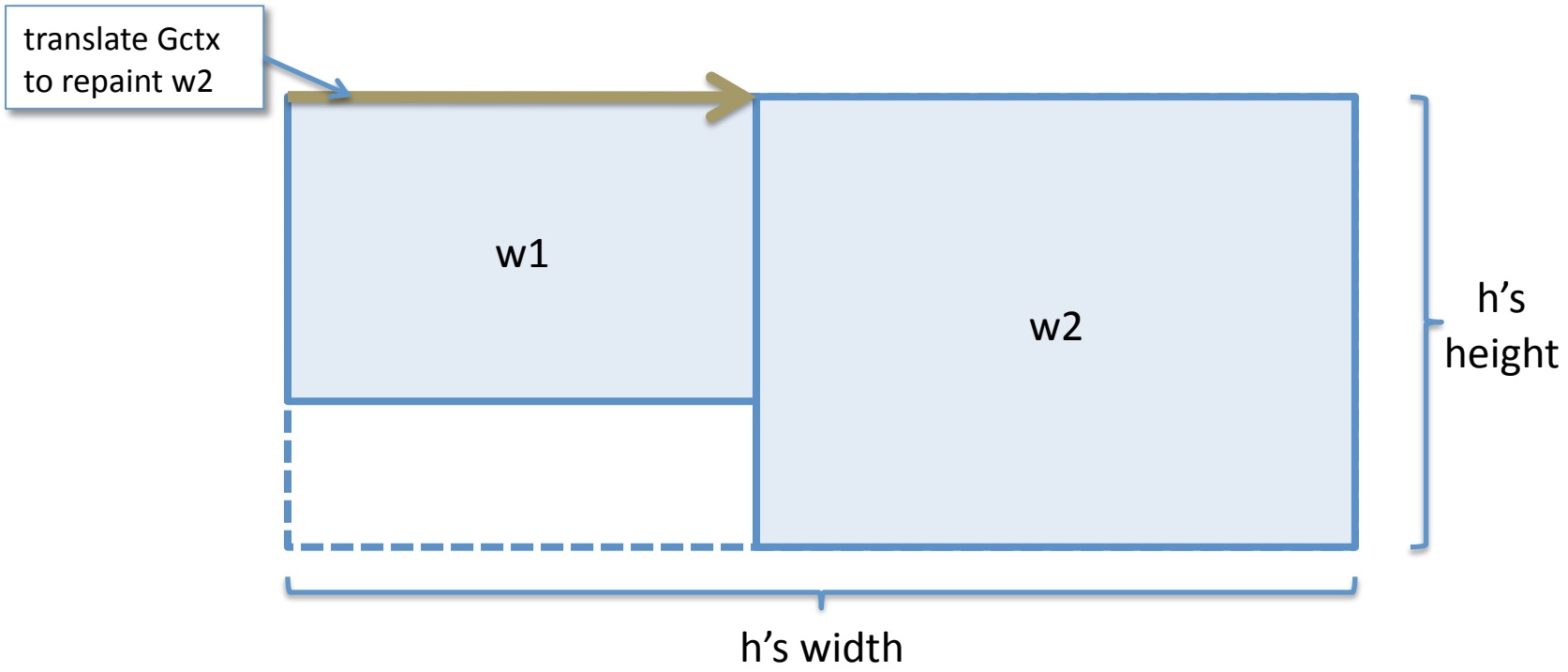
```
let border (w:t) :t =
{
  repaint = (fun (g:Gctx.t) ->
    let (width,height) = w.size g in
    let x = width + 3 in
    let y = height + 3 in
    Gctx.draw_line g (0,0) (x,0);
    Gctx.draw_line g (0,0) (0,y);
    Gctx.draw_line g (x,0) (x,y);
    Gctx.draw_line g (0,y) (x,y);
    let g = Gctx.translate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