Programming Languages and Techniques (CIS120)

Lecture 18

October 15th , 2015

Designing a GUI Library

Announcements

- HW5: GUI & Paint
  - Available on the web site
  - Due Thursday, October 22 at 11:59pm

Where we're going…

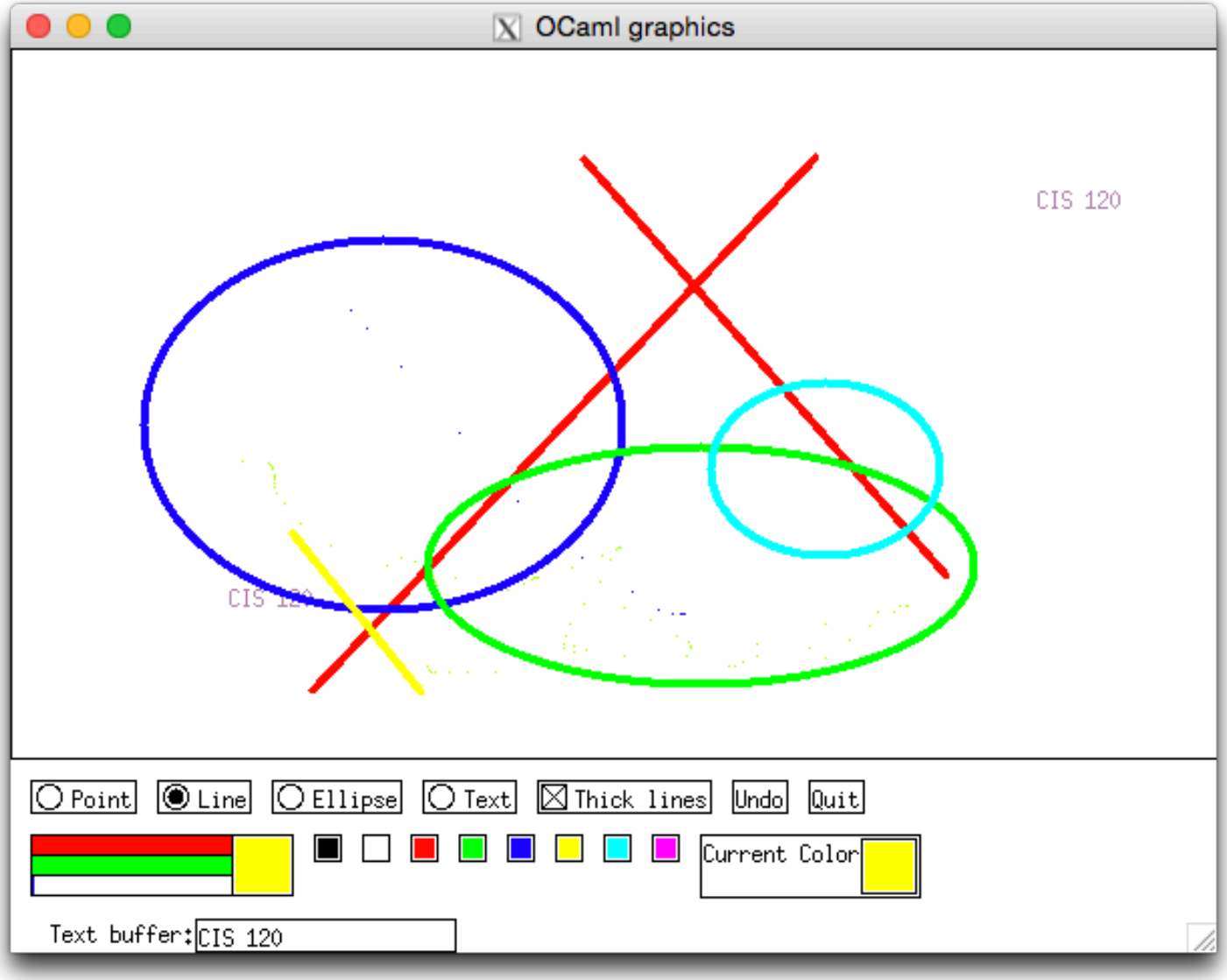
- HW 5: Build a GUI library and client application *from scratch* in OCaml

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

GUI Design

putting objects to work

Building a GUI and GUI Applications



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

1. Yes
2. No

Step #1: Understand the Problem

- We don't want to build just one graphical application: we want to make sure that our code is *reusable.*

- 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?

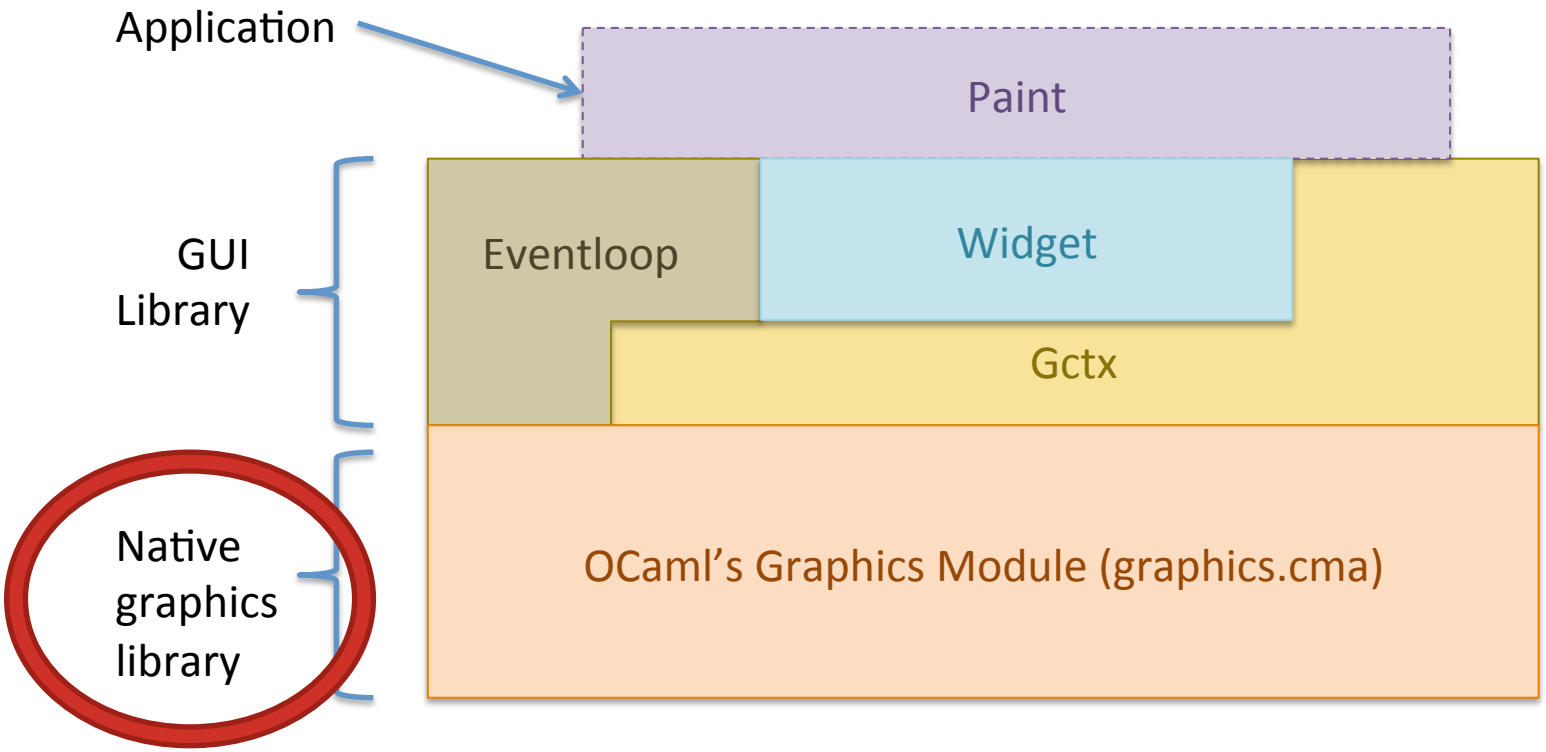
Designing a GUI library

- OCaml's Graphics library provides very *simple* 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: http://caml.inria.fr/pub/docs/manual-ocaml/libref/Graphics.html

- How do we go from that to a functioning, reusable GUI library?

Step 2, Interfaces: Project Architecture*

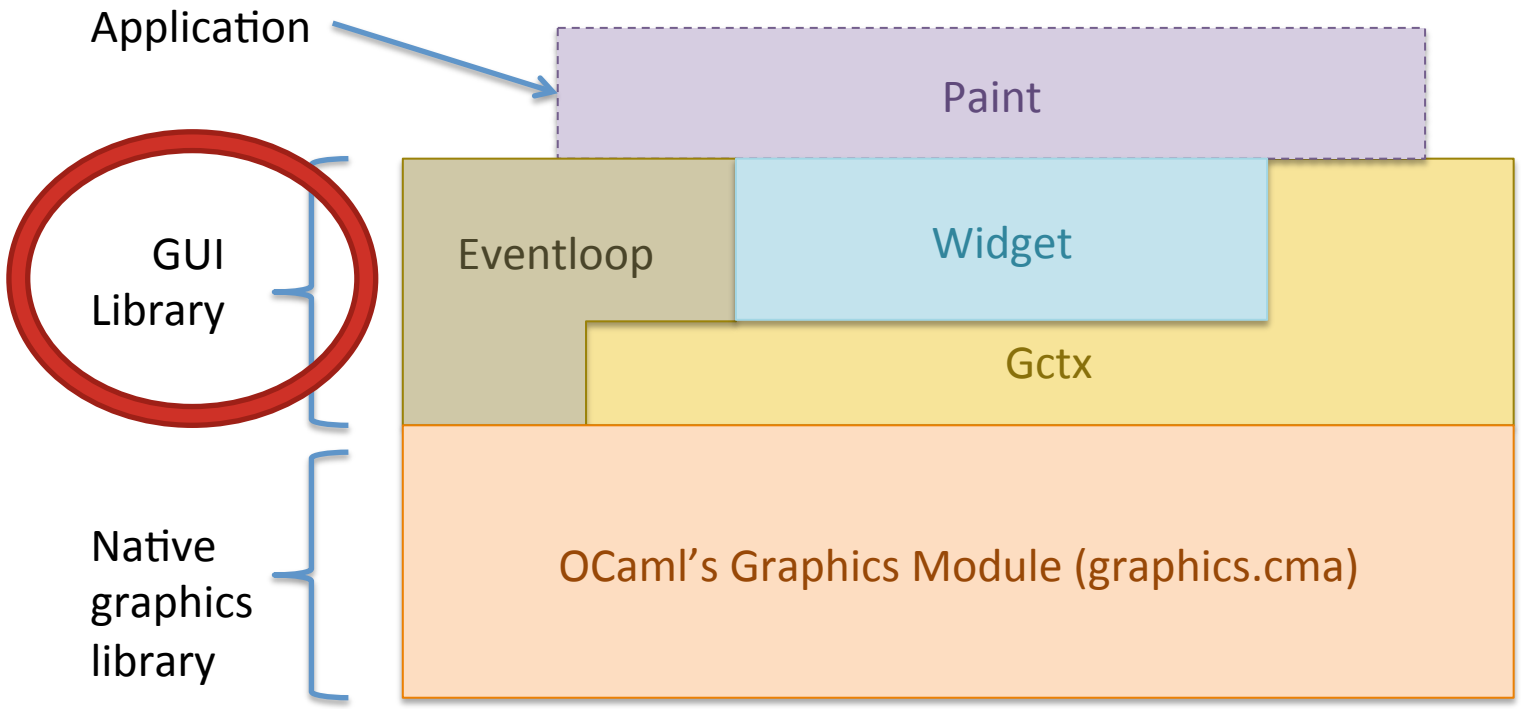
*Note: Subsequent program snippets are color-coded according to this diagram.



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

Step 2, 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)

- 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….

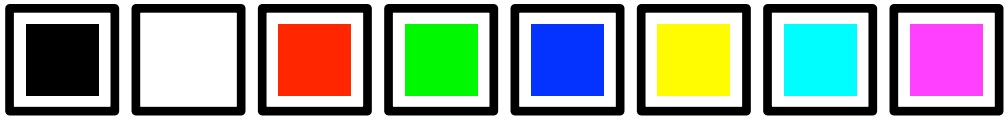
GUI terminology - Eventloop

- Main loop of any GUI application

```
let run (w:widget) : unit =
  Graphics.open_graph "";              (* open a new window *)
  Graphics.auto_synchronize false;

  let rec loop () : unit =
    Graphics.clear_graph ();

    repaint w;

    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.

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]
```

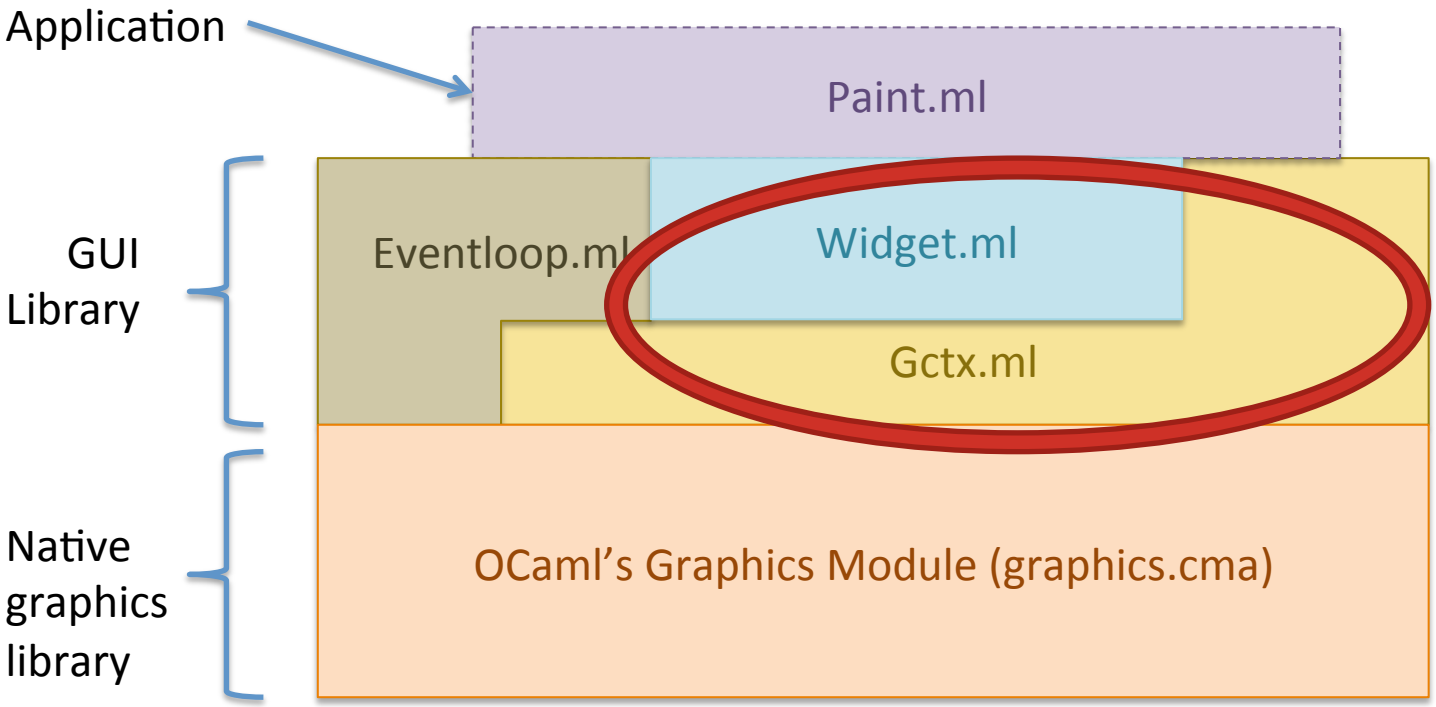
hlist is a container widget. It takes a list of widgets and turns them into a single one by laying them out horizontally.

paint.ml

- 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

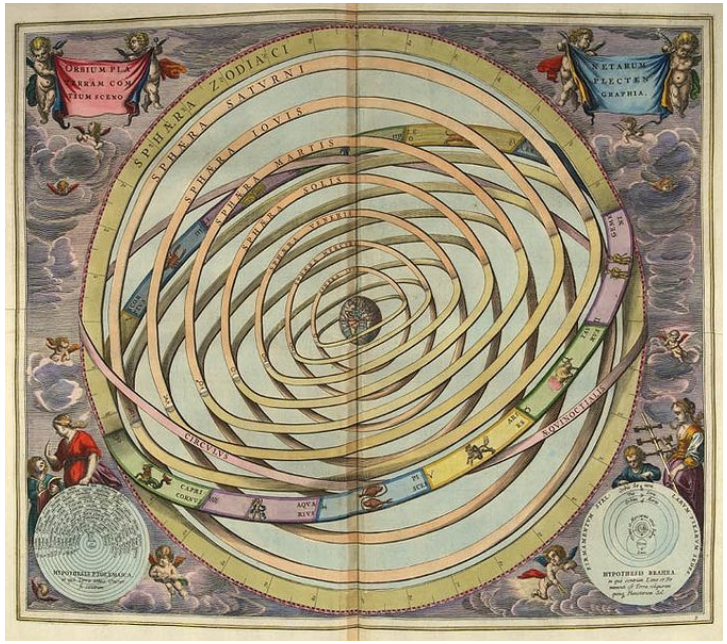
- Widgets are “things drawn on the screen”. How to make them location independent?
- Idea: Use a graphics context to make drawing primitives *relative* to the widget’s local coordinates.



The graphics context isolates the widgets from the Graphics module.

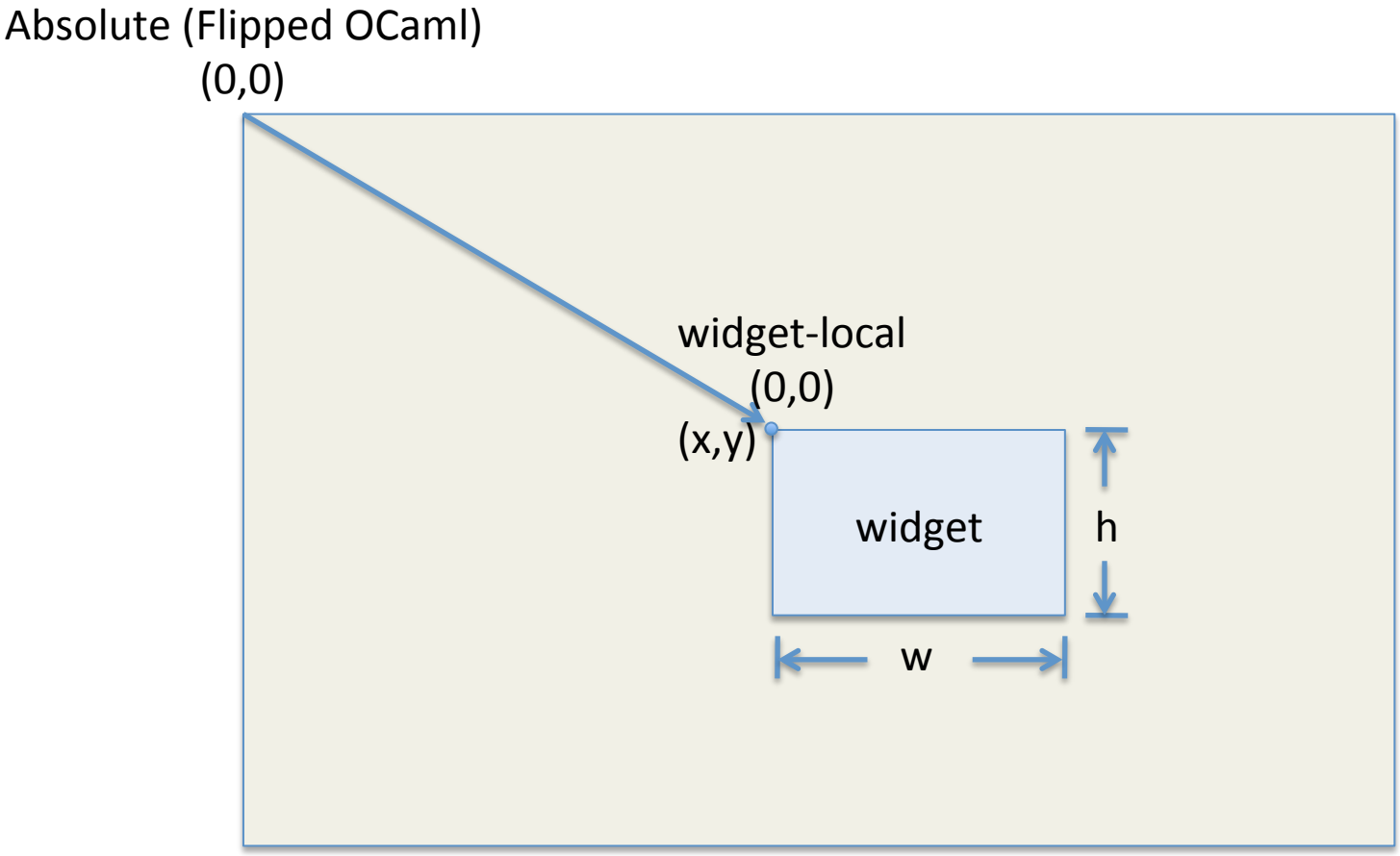
GUI terminology – Graphics Context

- 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



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

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).

Module: Gctx

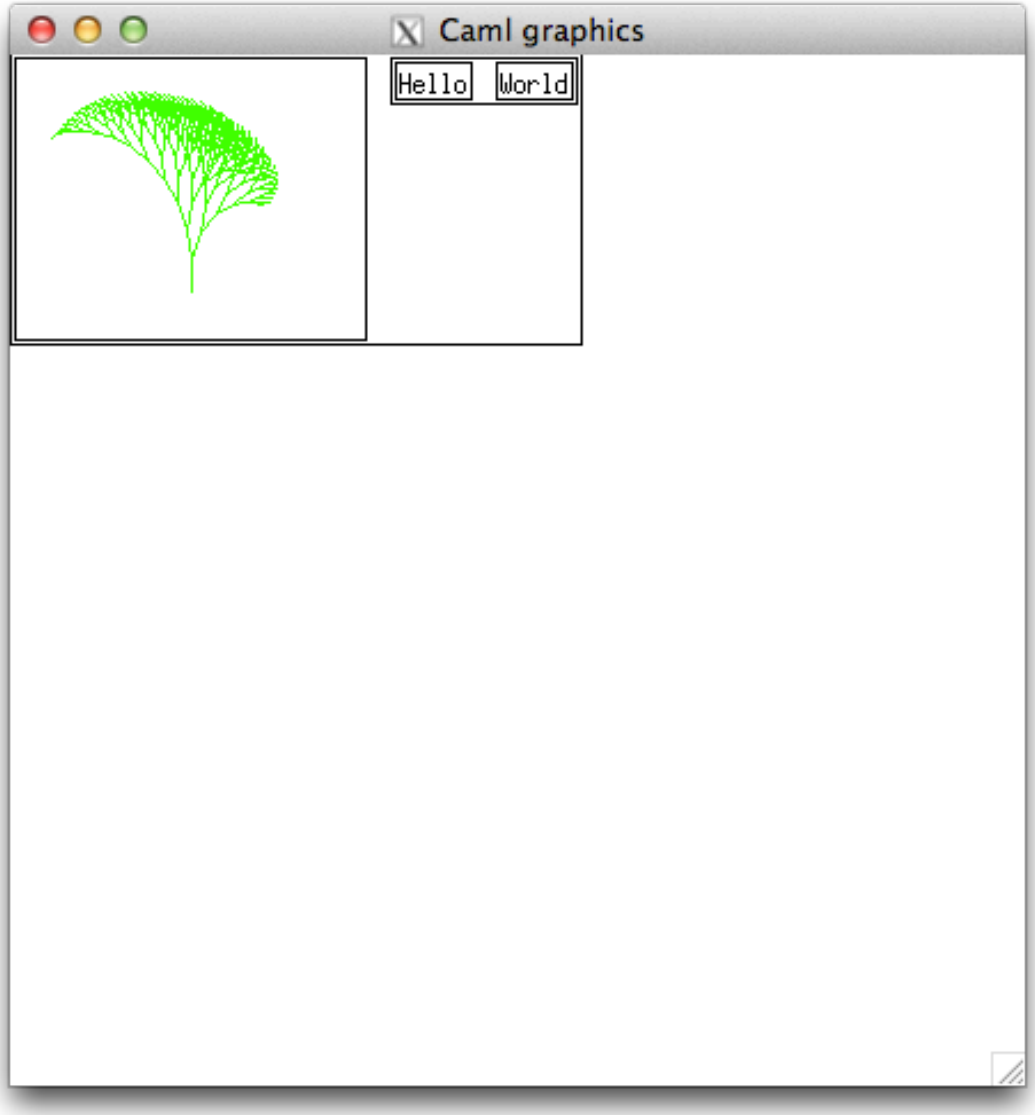
Contextualizes graphics drawing operations

Module: Widgets

Building blocks of GUI applications

see simpleWidget.ml

swdemo.ml



Simple Widgets

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

- 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

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

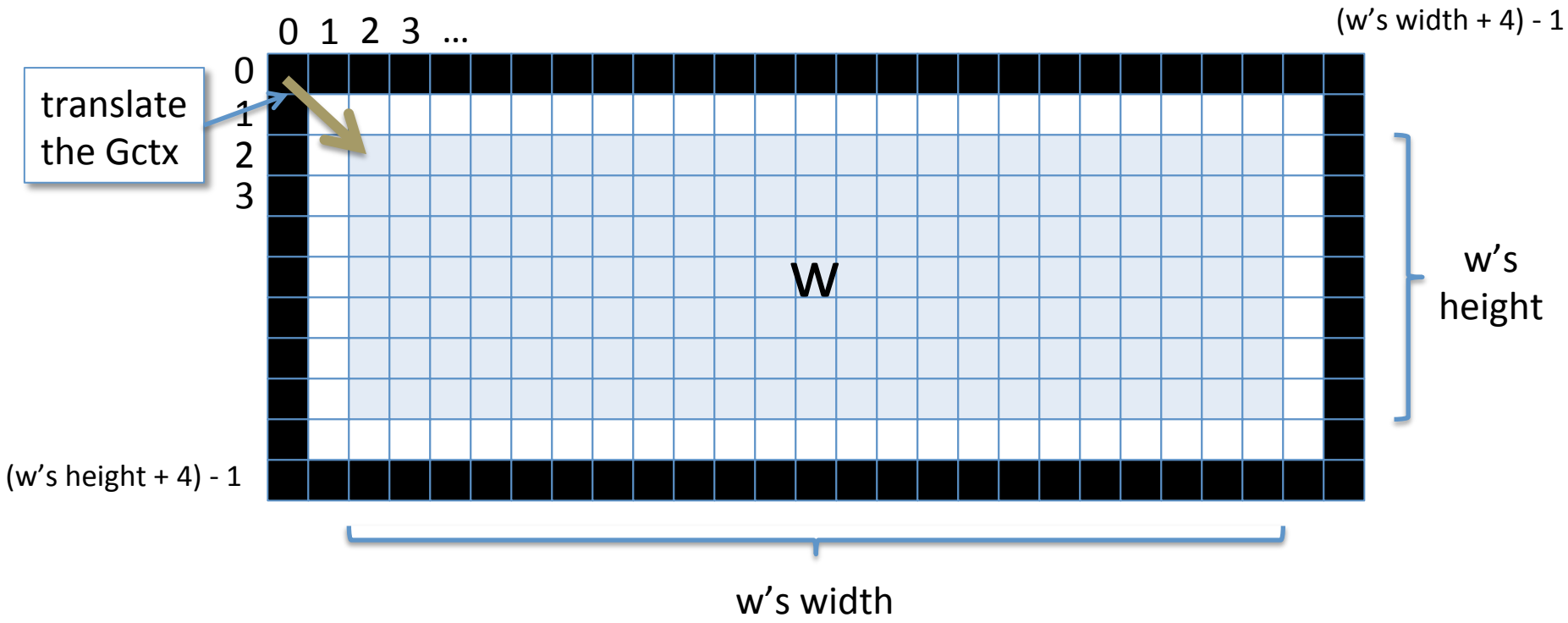
simpleWidget.ml

```
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

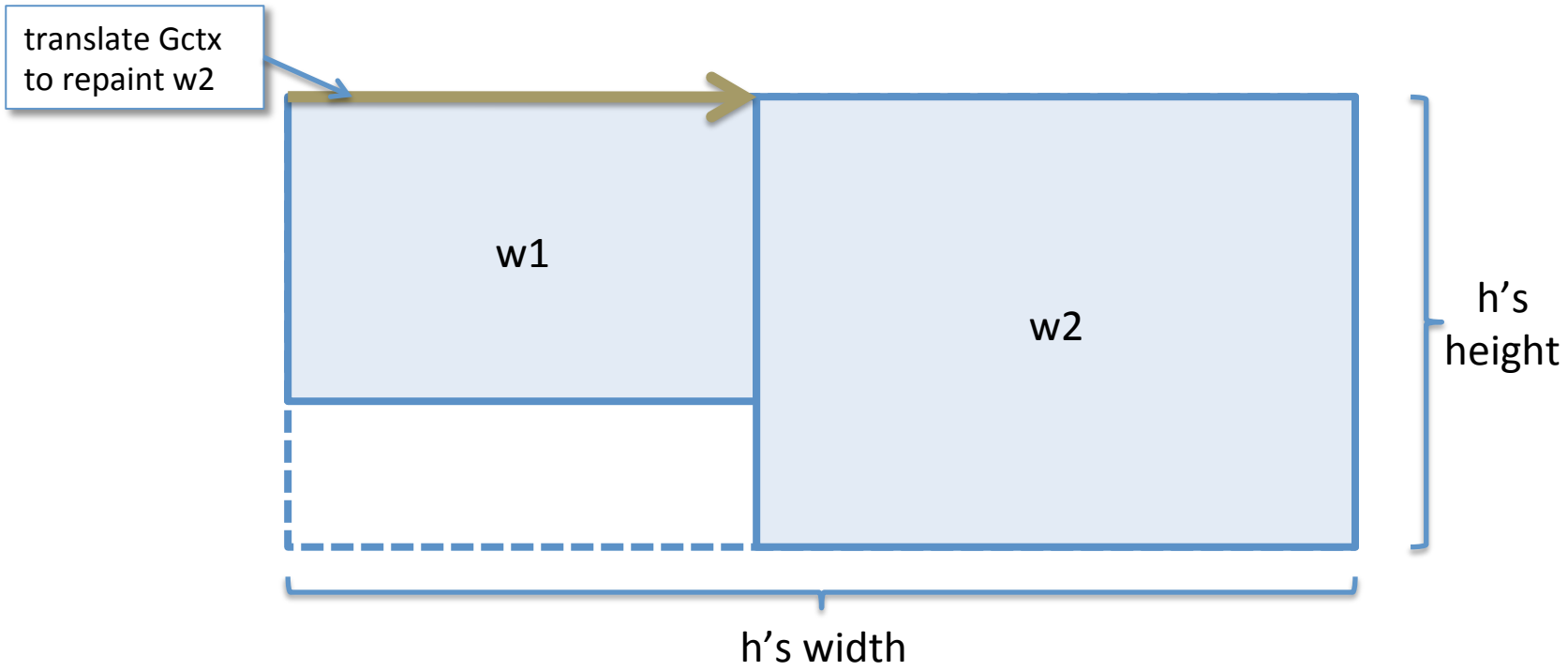
simpleWidget.ml

```
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 g = Gctx.translate g (2,2) in
  w.repaint g);

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.

Did you attend lecture today?

1. Yes