Programming Languages and Techniques (CIS120)

Lecture 18
March 13, 2019

Objects, GUI Library Design
Chapters 17 & 18
Objects and GUls
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*
  - 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
Have you ever used a GUI library (such as Java's Swing) to construct a user interface?

- Never
- Just a little
- Seriously
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 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.
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.

• 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
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

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

*Simplified!

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

*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

(* 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))))

Hello World

On the screen

Widget tree
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
Main loop for all GUI applications (simplified)

- “run” function takes top-level widget w as argument, containing all other widgets in the application.

```ocaml
let run (w: widget) : unit =
    Graphics.loop
    (fun e ->
      clear_graph ();
      w.handle e;
      w.repaint ()
    )
```

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

GUI terminology: “event loop”
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

```
Gctx.ml
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

Paint.ml

Native graphics library

Application

GUI Library

Eventloop.ml

Widget.ml

OCaml’s Graphics Module (graphics.cma)

Gctx.ml

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 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
let top = Gctx.top_level in
(* move origin and change the color *)
let nctx = Gctx.with_color Color.red
            (Gctx.translate top (dx,dy)) in
Gctx.draw_string top (0,10) "CIS 120";
Gctx.draw_string nctx (0,10) "CIS 120"
OCaml vs. “Standard” Coordinates

Standard (0,0) = OCaml (0,0)

Standard (x,y) = OCaml (x, size_y - y)
Module Gctx

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

```ocaml
type gctx
```

(** The top-level graphics context *)

```ocaml
val top_level : gctx
```

(** A widget-relative position *)

```ocaml
type position = int * int
```

(** Display text at the given (relative) position *)

```ocaml
val draw_string : gctx -> position -> string -> unit
```

(** Draw a line between the two specified positions *)

```ocaml
val draw_line : gctx -> position -> position -> unit
```

(** Produce a new gctx shifted by (dx,dy) *)

```ocaml
val translate : gctx -> int * int -> gctx
```

(** Produce a new gctx with a different pen color *)

```ocaml
val with_color : gctx -> color -> gctx
```
Recall: A \textit{widget} is an object with three methods...

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

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

let rec loop (f: event -> unit): unit =
  let e = wait_next_event () in
  f e;
  loop f
Container widgets propagate repaint commands to their children, *with appropriately modified graphics contexts*:
<table>
<thead>
<tr>
<th>Have you done Task 0 yet?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not yet</td>
</tr>
<tr>
<td>I've started</td>
</tr>
<tr>
<td>Completed and moving on!</td>
</tr>
</tbody>
</table>
Building a GUI library & application
Review: A “Hello World” application

(* 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)))

=? swdemo.ml

On the screen
Widget Layout

Building blocks of GUI applications
see simpleWidget.ml
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.
Simple Widgets

(*) An interface for simple GUI widgets *)

```ocaml
module simpleWidget = 

(* An interface for simple GUI widgets *)

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
- Repainting is relative to a graphics context
(* 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) 
}

(* A "blank" area widget -- it just takes up space *)
let space ((w,h):int*int) : widget =
{ 
  repaint = (fun (_:Gctx.gctx) -> ()); 
  size = (fun () -> (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
  – ...which will directly use the Gctx drawing routines to draw on the canvas

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

simpleWidget.ml