GUI en Caml

Pierre Weis
Pierre.Weis@inria.fr

9 mai 2001

PLAN

1. Notion d'interface graphique.

2. Exemples illustrés en Caml/Tk (Tcl/Tk).

Sigles

GUI : *Graphical User Interface*, c'est une librairie qui offre au programmeur l'accès aux fonctionnalités graphiques de la machine.

IHM : *Interface Homme Machine*, c'est l'ensemble des moyens d'interaction qu'un programme met à la disposition de son utilisateur pour l'aider à diriger le programme.

RAD : *Rapid Application Development* Toolkit, ensemble d'outils qui permettent de créer rapidement une application, c'est-à-dire un programme *et* son IHM.

Une GUI pour l'IHM des RAD

La GUI offre des services graphiques qui permettent la définition d'interactions à la souris et la présentation des résultats dans des fenêtres (dessins, textes).

La GUI est toujours complexe: boutons, ascenseurs, échelles, menus, etc ... tous munis de nombreux gadgets.

Elle offre une interface de programmation *relativement* indépendante de l'OS.

GUIs disponibles en Caml

En Caml, on dispose de plusieurs bibliothèques

- Caml/Tk tourne sous Unix (avec X-Windows), Windows (avec CygWin), et *susceptible* de tourner sur Mac.
- LablTk, uniquement sur Objective Caml (utilise les variantes polymorphes et les labels).
- LablGTK, uniquement sur Objective Caml (interface avec GTK+ la boîte à outils graphique de "The GIMP" GNU Image Manipulation Program).

Architecture globale d'une GUI

Une GUI comporte

- un ensemble d'objets graphiques: les *widgets*,
- un mécanisme de placement de ces objets dans la fenêtre graphique,
- une boucle d'interaction avec l'utilisateur (programmation par événements).

Architecture globale d'une GUI

Les widgets se comptent par dizaines, pour chacune desquelles on compte parfois une dizaine de fonctions.

Il faut impérativement structurer ces bibliothèques. On distingue:

- La structuration modulaire: chaque widget est un module qui définit le type du widget et comporte toutes les fonctions définies sur ce widget.
- La structuration objet: chaque widget est un objet qui comprend toutes les fonctions définies sur lui, en tant que *méthodes*.

Programmation par événements

La bibliothèque graphique comporte une boucle d'interaction qui observe les actions de l'utilisateur.

À chaque événement, la bibliothèque réagit en notifiant le widget concerné, en appelant un bout de code contenu dans le widget.

Ce bout de code est appelé *call-back*: selon l'approche, c'est une méthode de l'objet ou une fonction.

En général, un call-back ne doit pas prendre trop de temps . . .

Bibliothèque utilisée dans ces exemples

Pour des raisons de compatibilité avec Caml Light, on utilise une bibliothèque orientée modules, Caml/Tk (la seule disponible pour Caml Light).

Pour des questions de réutilisation de code (TeX :) les exemples sont donnés en Objective Caml.

La traduction en Caml Light n'est pas difficile . . .

Structure d'un programme Caml/Tk

- Ouverture de l'espace de nom de la GUI

```
open Tk;;
```

- Création des objets graphiques

```
let main_window = openTk ();;

let button =
  Button.create main_window [Text "Hello World!"];;
```

- Placement (*packing*),

```
pack [button] [];;
```

- Lancement de la boucle d'événements

```
mainLoop ();;
```

Exemple: start

Le code:

```
open Tk;;

let main_window = openTk ();;
val main_window : Widget.widget = <abstr>

let button =
  Button.create main_window [Text "Hello World!"];;
val button : Widget.widget = <abstr>

let start () =
  pack [button] [];
  mainLoop ();;
val start : unit -> unit = <fun>

start ();;
```

Compilation de programmes Caml/Tk

La compilation de ce programme

```
ocamlc -I +camltk -c start.ml
ocamlc -I +camltk -custom camltk.cma start.cmo
```

Utilisation d'un système interactif avec la bibliothèque Tk

```
ocamlmktop -o ocamltk -custom -I +camltk camltk.cma

ocamltk -I +camltk
        Objective Caml version 3.01

# open Tk;;
# pack;;
- : Widget.widget list -> Tk.options list -> unit = <fun>
```

Exemple: stop

On crée un autre bouton pour quitter l'application, avec un call-back

```
open Tk;;
let main_window = openTk ();;
let button =
  Button.create main_window [Text "Hello World!"];;

let quit_callback () = closeTk ();;
let quit =
  Button.create main_window
    [Text "Quit"; Command quit_callback];;

let start () =
  pack [button; quit] [];
  mainLoop ();;
start ();;
```

```
ocamlc -I +camltk -c stop.ml
ocamlc -I +camltk -custom camltk.cma stop.cmo
```

Placements des boutons

Pour avoir les boutons côte à côte, il suffit d'indiquer une option de la géometrie:

```
pack [button; quit] [Side Side_Left];
```

L'un au-dessous de l'autre

```
pack [button; quit] [Side Side_Bottom];
```

L'un au-dessus de l'autre (défaut)

```
pack [button; quit] [Side Side_Top];
```

Calculette une opération

Le problème graphique:

$$x + y = \text{somme de } x \text{ et } y$$

Labels: boutons sans actions ni effets associé.
Pour $+$, $=$, et le résultat.

Entrées interactive de texte: pour x et y.

```
open Tk;;
let main_window = openTk ();;

let operation = Label.create main_window [Text "+"];;
let result_sign = Label.create main_window [Text "="];;
let result = Label.create main_window [];;

let x_entry =
  Entry.create main_window [TextWidth 6; Relief Sunken];;
let y_entry =
  Entry.create main_window [TextWidth 6 ; Relief Sunken];;

Entry.create:
 Widget.widget -> Tk.options list -> Widget.widget
```

Calculette une opération

```
let quit_button =
  Button.create main_window
    [Text "Quit"; Command closeTk];;

pack
 [x_entry; operation; y_entry;
  result_sign; result]
 [Side Side_Left];
pack [quit_button] [];

(* Make the window resizable *)
Wm.minsize_set main_window 1 1;

mainLoop ();;
```

Et l'on peut entrer des nombres:

Calculette

Définition et mise en place des call-backs:

```
let x = ref 0
and y = ref 0;;

let refresh_result () =
  Label.configure result
    [Text (string_of_int (!x + !y))];;

let get_numbers_and_refresh (entry, r) _ _ =
  try
    r := int_of_string (Entry.get entry);
    refresh_result ()
  with
  | Failure "int_of_string" ->
      Label.configure result [Text "error"];;
```

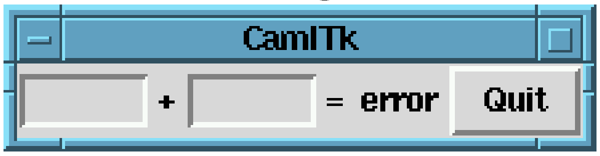
Calculette

Rq: application partielle ...

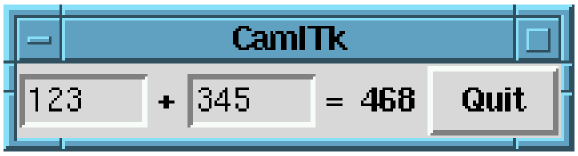
```
XScrollCommand: (float -> float -> unit) -> Tk.options
Entry.configure: Widget.widget -> Tk.options list -> unit
```

```
Entry.configure x_entry
 [XScrollCommand (get_numbers_and_refresh (x_entry, x))];
Entry.configure y_entry
 [XScrollCommand (get_numbers_and_refresh (y_entry, y))];;
```

Au lancement, `Entry.get` n'est pas défini:



Ça marche!



Sélecteur de couleurs

Les incontournables (omis dorénavant)

```
open Tk;;

let main_window = openTk ();;

let quit =
  Button.create main_window
    [Text "Quit"; Command closeTk];;
```

Pour le choix r, g, b: trois échelles ou *scales*,

```
let make_scale_255 wid name =
  Scale.create wid
    [Label name; From 0.0; To 255.0;
     Length (Centimeters 10.0); Orient Horizontal];;

let red, green, blue =
 make_scale_255 main_window "Red",
 make_scale_255 main_window "Green",
 make_scale_255 main_window "Blue";;
```

Sélecteur de couleurs

Pour avoir un échantillon: un cadre ou *frame*

```
let pattern =
  Frame.create main_window
   [Height (Centimeters 1.5);
    Width (Centimeters 6.0)];;
```

On en fixe le fond avec la couleur que définissent les échelles

```
let get_color () =
  let r = int_of_float (Scale.get red)
  and g = int_of_float (Scale.get green)
  and b = int_of_float (Scale.get blue) in
  Printf.sprintf "#%02x%02x%02x" r g b;;

let set_color _ =
  let color = get_color () in
  Frame.configure pattern
   [Background (NamedColor color)];;
```

Sélecteur de couleurs

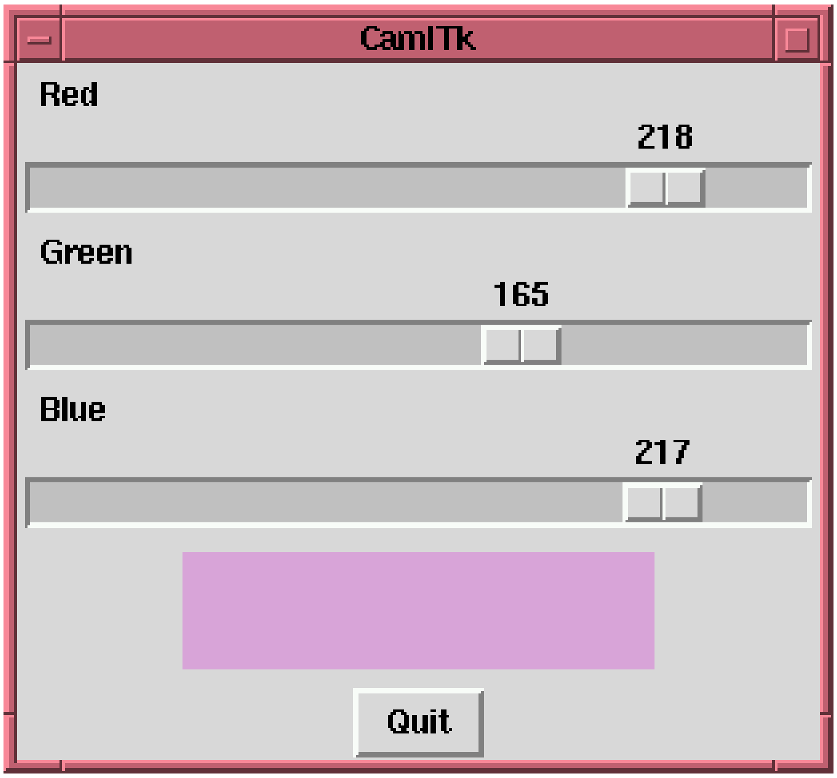
On configure les échelles:

```
let scales = [red; green; blue];;

let configure_scale sc =
  Scale.configure sc [ScaleCommand set_color];;

List.iter configure_scale scales;
```

On range tout:

```
pack scales [Side Side_Top];
pack [quit] [Side Side_Bottom];
pack [pattern] [Side Side_Bottom;
                PadY (Millimeters 2.0)];
mainLoop ();;
```

Sélecteur de couleurs



Sélecteur de couleurs II

On peaufine: titre de la fenêtre

```
Wm.title_set main_window "Choose a color";;
```

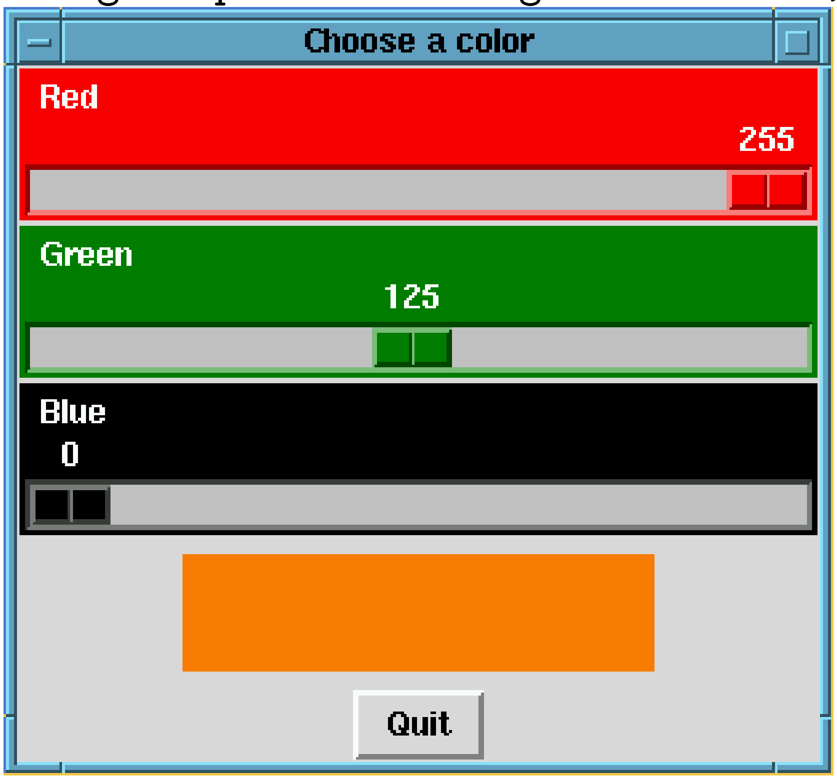
Changement dynamique de la couleur des échelles:

```
let tk_color_of_rgb r g b =
  NamedColor (Printf.sprintf "#%02x%02x%02x" r g b);;

let configure_scales r g b =
  let configure_scale sc r g b =
    Scale.configure sc
      [Background (tk_color_of_rgb r g b)] in
  configure_scale red r 0 0;
  configure_scale green 0 g 0;
  configure_scale blue 0 0 b;;
```

Sélecteur de couleurs II

On modifie le call-back en conséquence:

```
let set_color _ =
  let r = int_of_float (Scale.get red)
  and g = int_of_float (Scale.get green)
  and b = int_of_float (Scale.get blue) in
  configure_scales r g b;
  let color = tk_color_of_rgb r g b in
  Frame.configure pattern [Background color];;
```



Édition: textes

Pour créer un texte éditable: on crée un widget *text* qui renvoie le texte édité au call-back argument. On ajoute un ascenseur (ou *scroll-bar*).

```
let make_text_widget top topcb =
  let f = Frame.create top [] in
  let text = Text.create f [] in
  let scrollbar = Scrollbar.create f [] in
  scroll_link scrollbar text;
  pack [text; scrollbar]
       [Side Side_Left; Fill Fill_Y];
  pack [f][];
  let cb s =
    destroy f;
    topcb s in
  make_kill_buffer text cb;;
```

Liaison du texte à la scroll-bar

```
let scroll_link sb txt =
  Text.configure txt [YScrollCommand (Scrollbar.set sb)];
  Scrollbar.configure sb [ScrollCommand (Text.yview txt)];;
```

Édition: kill buffer

```
let make_kill_buffer text cb =
  let kill_ring = ref [] in
  let add_to_kill_ring s =
    kill_ring := s :: !kill_ring in
  let get_killed_text () =
    match !kill_ring with
    | [] -> ""
    | s :: l -> s in

  let insertMark = TextIndex (Mark "insert", []) in
  let eol_insertMark =
    TextIndex (Mark "insert", [LineEnd]) in

  let kill () =
    let s = Text.get text insertMark eol_insertMark in
    add_to_kill_ring (s ^ "\n");

  let yank () =
    let s = get_killed_text () in
    Text.insert text insertMark s [] in

  let yank_more () =
  ...
```

Édition: liaison des touches

CTRL-K, CTRL-Y, ESC-Y, CTRL-C quitte et renvoie le résultat

```
bind text [[Control], KeyPressDetail "k"]
  (BindSet ([], fun _ -> kill ()));

bind text [[Control], KeyPressDetail "y"]
  (BindSet ([], fun _ -> yank ()));

bind text [[Alt], KeyPressDetail "y"]
  (BindSet ([], fun _ -> yank_more ()));

bind text [[Control], KeyPressDetail "c"]
  (BindSet ([],
    fun _ ->
     let s = Text.get text
       (TextIndex (LineChar (0,0), []))
       (TextIndex (End, [])) in
     cb s));;
```

Édition: fabrication du widget texte

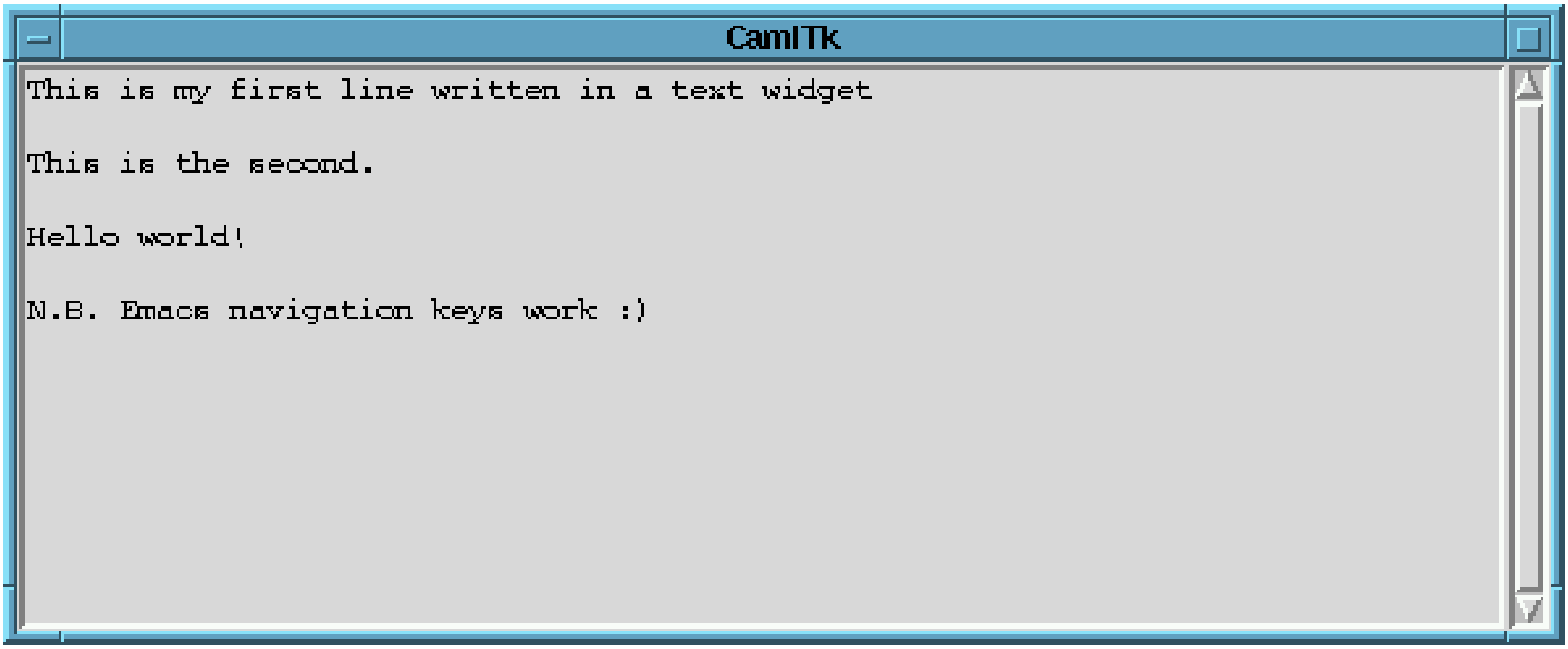
On crée un cadre, le texte et la scroll-bar. On fabrique un nouveau call-back avec celui de la fenêtre principale qui reçoit le résultat final.

```
let make_text_widget top cbtop =
  let f = Frame.create top [] in
  let text = Text.create f [] in
  let scrollbar = Scrollbar.create f [] in
  scroll_link scrollbar text;
  pack [text; scrollbar]
       [Side Side_Left; Fill Fill_X; Fill Fill_Y];
  pack [f][];
  let cb s =
    destroy f;
    cbtop s in
  make_kill_buffer text cb;;
```

Édition: enchaînement de call-backs

Le call-back de la fenêtre principale d'appel détruit la fenêtre et imprime.

```
let top = openTk ();;
make_text_widget top
 (fun s ->
    closeTk ();
    print_string s;
    print_newline ());;
```



Aide générique

On obtient des fonctionnalités génériques par abstraction. Cas de l'aide par boutons contextuels:

```
let give_help parent lines () =
 let help_window = Toplevel.create parent [] in
 Wm.title_set help_window "Help";

 let help_frame = Frame.create help_window [] in
 let help_txtw = create_filled_text help_frame lines in

 let quit_help () = destroy help_window in
 let ok_button =
   Button.create help_frame [Text "Ok"; Command quit_help] in

 pack [help_txtw; ok_button ] [Side Side_Bottom];
 pack [help_frame] [];;
```

Text suffisant pour afficher un message donné

La fenêtre d'aide s'adapte automatiquement à la taille du message.

```
let create_filled_text parent lines =
  let lnum = List.length lines
  and lwidth =
    List.fold_right
     (fun line max ->
       let l = String.length line in
       if l > max then l else max)
     lines 1 in
  let txtw =
    Text.create parent
      [TextWidth lwidth; TextHeight lnum] in
  List.iter
   (fun line ->
      Text.insert txtw (TextIndex (End, [])) line [];
      Text.insert txtw (TextIndex (End, [])) "\n" [])
   lines;
  txtw;;
```

Utilisation de l'aide générique

```
let quit =
  Button.create fp [Text "Quit"; Command closeTk] in
let help_lines =
 ["Pour jouer, cliquer sur une des pièces";
  "entourant le trou";
  "";
  "To play, click on a part next to the hole"] in
let help =
  Button.create fp
    [Text "Help"; Command (give_help fp help_lines)] in
pack [quit; help] [Side Side_Left; Fill Fill_X];
```

Manipulation d'images

On utilise des *canevas* et des *photos*.

Fonction de chargement:

```
let load_image filename =
  let img = Imagephoto.create [File filename] in
  let w = Imagephoto.width img
  and h = Imagephoto.height img in
  img, w, h;;
```

Impression sur canevas:

```
let display_image wid filename =
  let img, w, h = load_image filename in
  let c =
    Canvas.create wid
     [Width (Pixels w); Height (Pixels h)] in
  let img =
    Canvas.create_image c (Pixels w) (Pixels h)
     [ImagePhoto img; Anchor SE] in
  c;;
```

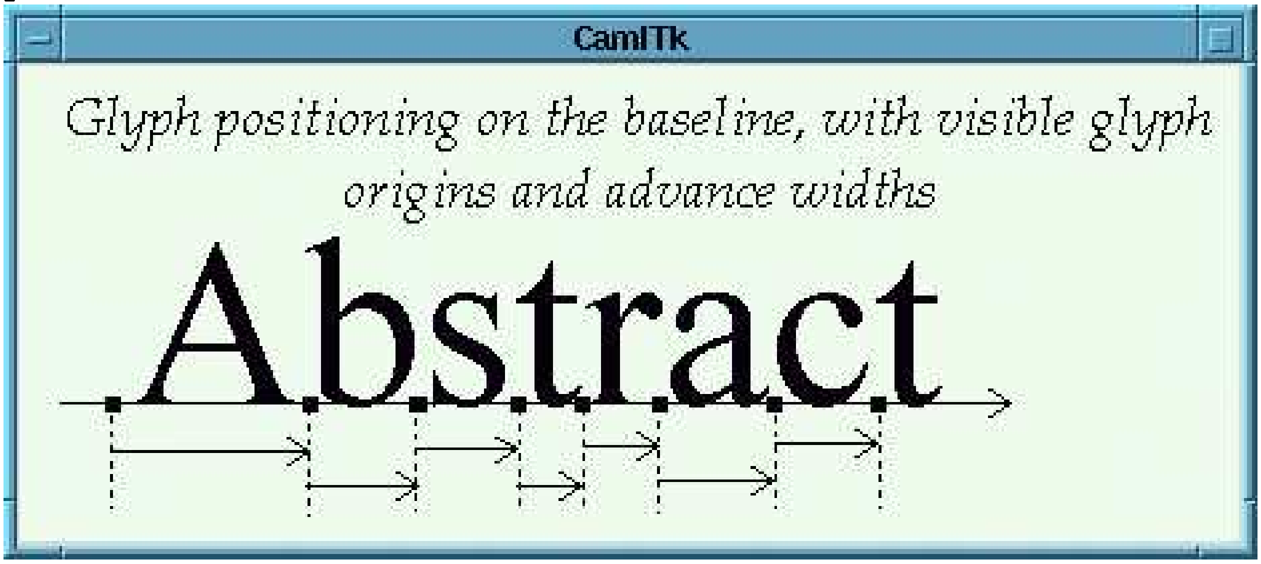
Manipulation d'images

Coquetterie: la fenêtre principale quitte avec ESC.

```
let top = openTk ();;

bind top [[], KeyPressDetail "Escape"]
  (BindSet ([], fun _ -> closeTk ()));
```

Chargement d'un gif et rangement du canevas.

```
let c = display_image top "baseline.gif" in
pack [c] [];;
```



Usage de la souris

On crée un canevas dans lequel on charge des images (marquées *img*):

```
let c =
  Canvas.create top
    [Width (Pixels w); Height (Pixels h)];;

let load_image filename =
  let photo = Imagephoto.create [File filename] in
  let w = Imagephoto.width photo
  and h = Imagephoto.height photo in
  let img =
    Canvas.create_image c (Pixels w) (Pixels h)
     [ImagePhoto photo; Anchor NW; Tags [Tag "img"]] in
  Canvas.coords_set c img [Pixels 0; Pixels 0];
  img, w, h;;

let imgs =
  List.map load_image ["CamlBook.gif"; "dojoji.gif"];;
```

Localisation de la souris

```
let get_coords c img =
  match Canvas.coords_get c img with
  | [x0; y0] -> int_of_float x0, int_of_float y0
  | _ -> assert false;;

let on_img x y c (img, w, h) =
  let x0, y0 = get_coords c img in
  x0 <= x && x <= x0 +  h &&
  y0 <= y && y <= y0 + w;;

let img_of c x y = List.find (on_img x y c) imgs;;
```

Déplacement d'une image:

```
let move img (tx, ty) ei =
  Canvas.coords_set c img
   [Pixels (ei.ev_MouseX + tx);
    Pixels (ei.ev_MouseY + ty)];;
```

Définition des call-back d'événements

```
let rec get ei =
  let x, y = ei.ev_MouseX, ei.ev_MouseY in
  let img, _, _ = img_of c x y in
  let img_x, img_y = get_coords c img in
  let t = img_x - x, img_y - y in
  Canvas.bind c (Tag "img") [[], Motion]
    (BindSet
      ([Ev_MouseX; Ev_MouseY],
       move img t))

and release ei =
  Canvas.bind c (Tag "img") [[], Motion]
    (BindSet ([Ev_MouseX; Ev_MouseY], ignore));;
```

Fin du programme

```
Canvas.bind c (Tag "img") [[], ButtonPress]
 (BindSet ([Ev_MouseX; Ev_MouseY], get));
Canvas.bind c (Tag "img") [[], ButtonRelease]
 (BindSet ([Ev_MouseX; Ev_MouseY], release));;
pack [c] [];;

mainLoop ();;
```

Autres applications

Camlbrowser: browser de sources pour Caml Light

Ocamlbrowser: . . .

MMM: brouteur en Caml (Caml/Tk) avec applets ...

Mine d'exemples: `http://pauillac.inria.fr/caml/Examples/oc/camltk/`

Taquin, Tetris, Camleyes, Gif animation, ...

Conclusions

Vraiment grosses bibliothèques: il faut ériger en dogme la *programmation par crassuçage* ...

Question: et les objets dans tout ça ?

Réponse: voir X Windows, Inside Macintosh, la bible du PC (avec les MFC), puis se faire une philosophie et la communiquer à tous ...