5

The Graphics Interface

This chapter presents the `Graphics` library, which is included in the distribution of the Objective Caml-language. This library is designed in such a way that it works identically under the main graphical interfaces of the most commonly used operating systems: Windows, MacOS, Unix with X-Windows. `Graphics` permits the realization of drawings which may contain text and images, and it handles basic events like mouse clicks or pressed keys.

The model of programming graphics applied is the "painter's model:" the last touch of color erases the preceding one. This is an imperative model where the graphics window is a table of points which is physically modified by each graphics primitive. The interactions with the mouse and the keyboard are a model of event-driven programming: the primary function of the program is an infinite loop waiting for user interaction. An event starts execution of a special handler, which then returns to the main loop to wait for the next event.

Although the `Graphics` library is very simple, it is sufficient for introducing basic concepts of graphical interfaces, and it also contains basic elements for developing graphical interfaces that are rich and easy to use by the programmer.

Chapter overview

The first section explains how to make use of this library on different systems. The second section introduces the basic notions of graphics programming: reference point, plotting, filling, colors, bitmaps. The third section illustrates these concepts by describing and implementing functions for creating and drawing "boxes." The fourth section demonstrates the animation of graphical objects and their interaction with the background of the screen or other animated objects. The fifth section presents event-driven programming, in other terms the skeleton of all graphical interfaces. Finally, the last

section uses the library `Graphics` to construct a graphical interface for a calculator (see page 86).

Using the `Graphics` *Module*

Utilization of the library `Graphics` differs depending on the system and the compilation mode used. We will not cover applications other than usable under the interactive toplevel of Objective Caml. Under the Windows and MacOS systems the interactive working environment already preloads this library. To make it available under Unix, it is necessary to create a new toplevel. This depends on the location of the X11 library. If this library is placed in one of the usual search paths for C language libraries, the command line is the following:

```
ocamlmktop -custom -o mytoplevel graphics.cma -cclib -lX11
```

It generates a new executable`mytoplevel` into which the library `Graphics` is integrated. Starting the executable works as follows:

```
./mytoplevel
```

If, however, as under Linux, the library X11 is placed in another directory, this has to be indicated to the command `ocamlmktop`:

```
ocamlmktop -custom -o mytoplevel graphics.cma -cclib \
           -L/usr/X11/lib -cclib -lX11
```

In this example, the file `libX11.a` is searched in the directory `/usr/X11/lib`.

A complete description of the command `ocamlmktop` can be found in chapter 7.

Basic notions

Graphics programming is tightly bound to the technological evolution of hardware, in particular to that of screens and graphics cards. In order to render images in sufficient quality, it is necessary that the drawing be refreshed (redrawn) at regular and short intervals, somewhat like in a cinema. There are basically two techniques for drawing on the screen: the first makes use of a list of visible segments where only the useful part of the drawing is drawn, the second displays all points of the screen (bitmap screen). It is the last technique which is used on ordinary computers.

Bitmap screens can be seen as rectangles of accessible, in other terms, displayable points. These points are called *pixels*, a word derived from *picture element*. They are the basic elements for constructing images. The height and width of the main bitmap

is the resolution of the screen. The size of this bitmap therefore depends on the size of each pixel. In monochrome (black/white) displays, a pixel can be encoded in one bit. For screens that allow gray scales or for color displays, the size of a pixel depends on the number of different colors and shades that a pixel may take. In a bitmap of 320x640 pixels with 256 colors per pixel, it is therefore necessary to encode a pixel in 8 bits, which requires video memory of: $480 * 640$ bytes $= 307200$ bytes $\simeq$ 300KB. This resolution is still used by certain MS-DOS programs.

The basic operations on bitmaps which one can find in the `Graphics` library are:

- coloration of pixels,
- drawing of pixels,
- drawing of forms: rectangles, ellipses,
- filling of closed forms: rectangles, ellipses, polygons,
- displaying text: as bitmap or as vector,
- manipulation or displacement of parts of the image.

All these operations take place at a *reference point*, the one of the bitmap. A certain number of characteristics of these graphical operations like the width of strokes, the joints of lines, the choice of the character font, the style and the motive of filling define what we call a *graphical context*. A graphical operation always happens in a particular graphical context, and its result depends on it. The graphical context of the `Graphics` library does not contain anything except for the current point, the current color, the current font and the size of the image.

Graphical display

The elements of the graphical display are: the reference point and the graphical context, the colors, the drawings, the filling pattern of closed forms, the texts and the bitmaps.

Reference point and graphical context

The `Graphics` library manages a unique main window. The coordinates of the reference point of the window range from point $(0, 0)$ at the bottom left to the upper right corner of the window. The main functions on this window are:

- `open_graph`, of type *`string -> unit`*, which opens a window;
- `close_graph`, of type *`unit -> unit`*, which closes it;
- `clear_graph`, of type *`unit -> unit`*, which clears it.

The dimensions of the graphical window are given by the functions `size_x` and `size_y`.

The string argument of the function `open_graph` depends on the window system of the machine on which the program is executed and is therefore not platform independent. The empty string, however, opens a window with default settings. It is possible to

specify the size of the window: under X-Windows, `" 200x300"` yields a window which is 200 pixels wide and 300 pixels high. Beware, the space at the beginning of the string `" 200x300"` is required!

The graphical context contains a certain number of readable and/or modifiable parameters:

the current point:	`current_point` : *unit -> int * int*
	`moveto` : *int -> int -> unit*
the current color:	`set_color` : *color -> unit*
the width of lines:	`set_line_width` : *int -> unit*
the current character font:	`set_font` : *string -> unit*
the size of characters:	`set_text_size` : *int -> unit*

Colors

Colors are represented by three bytes: each stands for the intensity value of a main color in the RGB-model (red, green, blue), ranging from a minimum of 0 to a maximum of 255. The function `rgb` (of type *int -> int -> int -> color*) allows the generation of a new color from these three components. If the three components are identical, the resulting color is a gray which is more or less intense depending on the intensity value. Black corresponds to the minimum intensity of each component (`0 0 0`) and white is the maximum (`255 255 255`). Certain colors are predefined: `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan` and `magenta`.

The variables `foreground` and `background` correspond to the color of the fore- and the background respectively. Clearing the screen is equivalent to filling the screen with the background color.

A color (a value of type *color*) is in fact an integer which can be manipulated to, for example, decompose the color into its three components (`from_rgb`) or to apply a function to it that inverts it (`inv_color`).

```
(* color == R * 256 * 256  +  G * 256  +  B  *)
# let from_rgb (c : Graphics.color)  =
    let r = c / 65536  and  g = c / 256 mod 256  and  b = c mod 256
    in (r,g,b);;
val from_rgb : Graphics.color -> int * int * int = <fun>
# let inv_color (c : Graphics.color) =
     let (r,g,b) = from_rgb c
     in Graphics.rgb (255-r) (255-g) (255-b);;
val inv_color : Graphics.color -> Graphics.color = <fun>
```

The function `point_color`, of type *int -> int -> color*, returns the color of a point when given its coordinates.

Drawing and filling

A drawing function draws a line on the screen. The line is of the current width and color. A filling function fills a closed form with the current color. The various line- and filling functions are presented in figure 5.1.

drawing	filling	type
`plot`		*int -> int -> unit*
`lineto`		*int -> int -> unit*
	`fill_rect`	*int -> int -> int -> int -> unit*
	`fill_poly`	*(int * int) array -> unit*
`draw_arc`	`fill_arc`	*int -> int -> int -> int -> int -> unit*
`draw_ellipse`	`fill_ellipse`	*int -> int -> int -> int -> unit*
`draw_circle`	`fill_circle`	*int -> int -> int -> unit*

Figure 5.1: Drawing- and filling functions.

Beware, the function `lineto` changes the position of the current point to make drawing of vertices more convenient.

Drawing polygons To give an example, we add drawing primitives which are not predefined. A polygon is described by a table of its vertices.

```
# let draw_rect x0 y0 w h =
    let (a,b) = Graphics.current_point()
    and x1 = x0+w and y1 = y0+h
    in
      Graphics.moveto x0 y0;
      Graphics.lineto x0 y1; Graphics.lineto x1 y1;
      Graphics.lineto x1 y0; Graphics.lineto x0 y0;
      Graphics.moveto a b;;
val draw_rect : int -> int -> int -> int -> unit = <fun>

# let draw_poly r =
    let (a,b) = Graphics.current_point () in
    let (x0,y0) = r.(0) in Graphics.moveto x0 y0;
      for i = 1 to (Array.length r)-1 do
        let (x,y) = r.(i) in Graphics.lineto x y
      done;
      Graphics.lineto x0 y0;
      Graphics.moveto a b;;
val draw_poly : (int * int) array -> unit = <fun>
```

Please note that these functions take the same arguments as the predefined ones for filling forms. Like the other functions for drawing forms, they do not change the current point.

Illustrations in the painter's model This example generates an illustration of a token ring network (figure 5.2). Each machine is represented by a small circle. We place the set of machines on a big circle and draw a line between the connected machines. The current position of the token in the network is indicated by a small black disk.

The function `net_points` generates the coordinates of the machines in the network. The resulting data is stored in a table.

```
# let pi = 3.1415927;;
val pi : float = 3.1415927
# let net_points (x,y) l n =
    let a = 2. *. pi /. (float n) in
    let rec aux (xa,ya) i =
      if i > n then []
      else
        let na = (float i) *. a in
        let x1 = xa + (int_of_float ( cos(na) *. l))
        and y1 = ya + (int_of_float ( sin(na) *. l))  in
        let np = (x1,y1) in
          np :: (aux np (i+1))
    in Array.of_list (aux (x,y) 1);;
val net_points : int * int -> float -> int -> (int * int) array = <fun>
```

The function `draw_net` displays the connections, the machines and the token.

```
# let draw_net (x,y) l n sc st =
   let r = net_points (x,y) l n in
     draw_poly r;
     let draw_machine (x,y) =
       Graphics.set_color Graphics.background;
       Graphics.fill_circle x y sc;
       Graphics.set_color Graphics.foreground;
       Graphics.draw_circle x y sc
     in
       Array.iter draw_machine r;
       Graphics.fill_circle x y st;;
val draw_net : int * int -> float -> int -> int -> int -> unit = <fun>
```

The following function call corresponds to the left drawing in figure 5.2.

```
# draw_net (140,20) 60.0 10 10 3;;
- : unit = ()

# save_screen "IMAGES/tokenring.caa";;
```

```
- : unit = ()
```

We note that the order of drawing objects is important. We first plot the connections

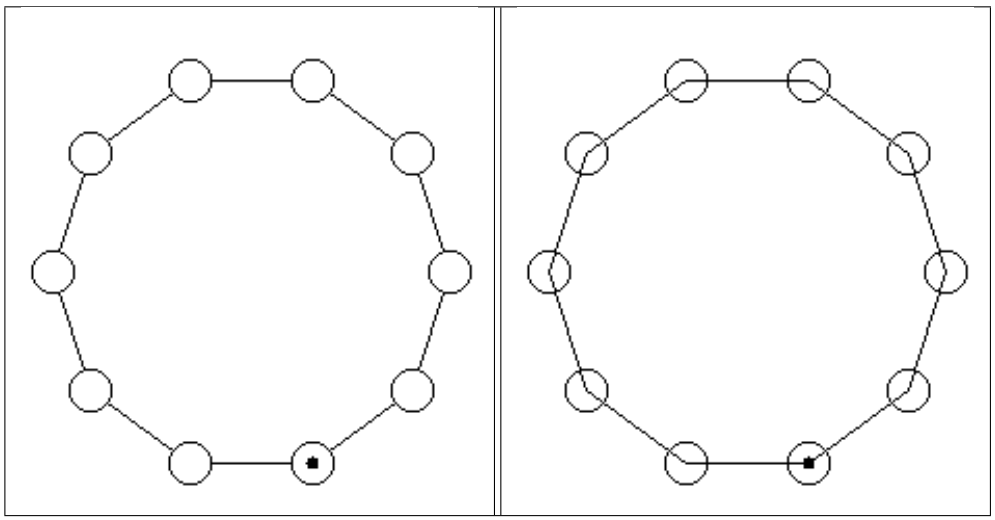


Figure 5.2: Tokenring network.

then the nodes. The drawing of network nodes erases some part of the connecting lines. Therefore, there is no need to calculate the point of intersection between the connection segments and the circles of the vertices. The right illustration of figure 5.2 inverts the order in which the objects are displayed. We see that the segments appear inside of the circles representing the nodes.

Text

The functions for displaying texts are rather simple. The two functions `draw_char` (of type *`char -> unit`*) and `draw_string` (of type *`string -> unit`*) display a character and a character string respectively at the current point. After displaying, the latter is modified. These functions do not change the current font and its current size.

Note

The displaying of strings may differ depending on the graphical interface.

The function `text_size` takes a string as input and returns a pair of integers that correspond to the dimensions of this string when it is displayed in the current font and size.

Displaying strings vertically This example describes the function `draw_string_v`, which displays a character string vertically at the current point. It is used in figure 5.3. Each letter is displayed separately by changing the vertical coordinate.

```
# let draw_string_v  s =
```

```
    let (xi,yi) = Graphics.current_point()
    and l = String.length s
    and (_,h) = Graphics.text_size s
    in
       Graphics.draw_char s.[0];
       for i=1 to l-1 do
         let (_,b) = Graphics.current_point()
         in Graphics.moveto xi (b-h);
            Graphics.draw_char s.[i]
        done;
      let (a,_) = Graphics.current_point() in Graphics.moveto a yi;;
val draw_string_v : string -> unit = <fun>
```

This function modifies the current point. After displaying, the point is placed at the initial position offset by the width of one character.

The following program permits displaying a legend around the axes (figure 5.3)

```
#
  Graphics.moveto 0 150; Graphics.lineto 300 150;
  Graphics.moveto 2 130; Graphics.draw_string "abscissa";
  Graphics.moveto 150 0; Graphics.lineto 150 300;
  Graphics.moveto 135 280; draw_string_v "ordinate";;
- : unit = ()
```

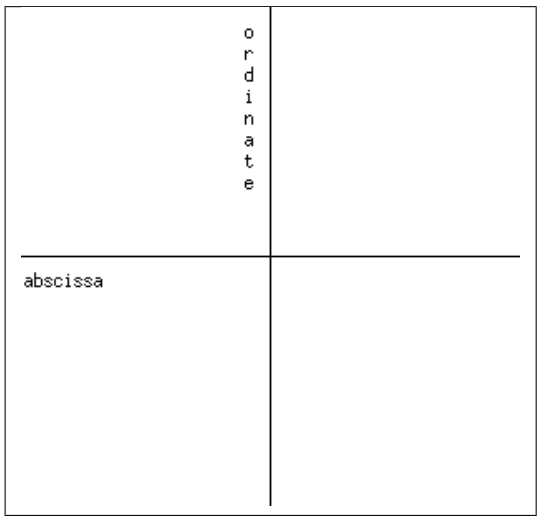


Figure 5.3: Legend around axes.

If we wish to realize vertical displaying of text, it is necessary to account for the fact that the current point is modified by the function `draw_string_v`. To do this, we define the function `draw_text_v`, which accepts the spacing between columns and a list of words as parameters.

```
# let draw_text_v n l =
    let f s = let (a,b) = Graphics.current_point()
              in draw_string_v s;
                 Graphics.moveto (a+n) b
    in List.iter f l;;
val draw_text_v : int -> string list -> unit = <fun>
```

If we need further text transformations like, for example, rotation, we will have to take the *bitmap* of each letter and perform the rotation on this set of pixels.

Bitmaps

A bitmap may be represented by either a color matrix (*color array array*) or a value of abstract type [1] *image*, which is declared in library `Graphics`. The names and types of the functions for manipulating bitmaps are given in figure 5.4.

function	type
make_image	*color array array -> image*
dump_image	*image -> color array array*
draw_image	*image -> int -> int -> unit*
get_image	*int -> int -> int -> int -> image*
blit_image	*image -> int -> int -> unit*
create_image	*int -> int -> image*

Figure 5.4: Functions for manipulating bitmaps.

The functions `make_image` and `dump_image` are conversion functions between types *image* and *color array array*. The function `draw_image` displays a bitmap starting at the coordinates of its bottom left corner.

The other way round, one can capture a rectangular part of the screen to create an image using the function `get_image` and by indicating the bottom left corner and the upper right one of the area to be captured. The function `blit_image` modifies its first parameter (of type *image*) and captures the region of the screen where the lower left corner is given by the point passed as parameter. The size of the captured region is the one of the image argument. The function `create_image` allows initializing images by specifying their size to use them with `blit_image`.

The predefined color `transp` can be used to create transparent points in an image. This makes it possible to display an image within a rectangular area only; the transparent points do not modify the initial screen.

1. Abstract types hide the internal representation of their values. The declaration of such types will be presented in chapter 14.

Polarization of Jussieu This example inverts the color of points of a bitmap. To do this, we use the function for color inversion presented on page 120, applying it to each pixel of a bitmap.

```
# let inv_image i =
    let inv_vec = Array.map (fun c → inv_color c) in
    let inv_mat = Array.map inv_vec in
    let inverted_matrix = inv_mat (Graphics.dump_image i) in
    Graphics.make_image inverted_matrix;;
val inv_image : Graphics.image -> Graphics.image = <fun>
```

Given the bitmap `jussieu`, which is displayed in the left half of figure 5.5, we use the function `inv_image` and obtain a new "solarized" bitmap, which is displayed in the right half of the same figure.

```
# let f_jussieu2 () = inv_image jussieu1;;
val f_jussieu2 : unit -> Graphics.image = <fun>
```

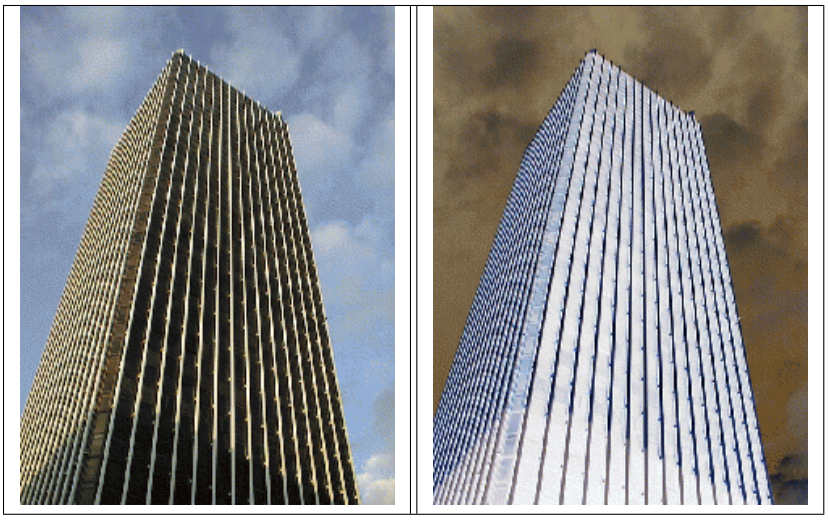
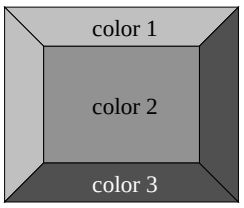


Figure 5.5: Inversion of Jussieu.

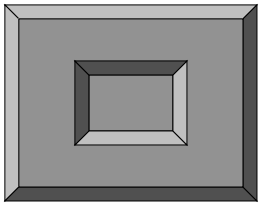
Example: drawing of boxes with relief patterns

In this example we will define a few utility functions for drawing boxes that carry relief patterns. A box is a generic object that is useful in many cases. It is inscribed in a rectangle which is characterized by a point of origin, a height and a width.

To give an impression of a box with a relief pattern, it is sufficient to surround it with two trapezoids in a light color and two others in a somewhat darker shade.



Inverting the colors, one can give the impression that the boxes are on top or at the bottom.



Implementation We add the border width, the display mode (top, bottom, flat) and the colors of its edges and of its bottom. This information is collected in a record.

```
# type relief = Top | Bot | Flat;;
# type box_config =
    { x:int; y:int; w:int; h:int; bw:int; mutable r:relief;
      b1_col : Graphics.color;
      b2_col : Graphics.color;
      b_col : Graphics.color};;
```

Only field `r` can be modified. We use the function `draw_rect` defined at page 121, which draws a rectangle.

For convenience, we define a function for drawing the outline of a box.

```
# let draw_box_outline bcf col =
    Graphics.set_color col;
    draw_rect bcf.x bcf.y bcf.w  bcf.h;;
val draw_box_outline : box_config -> Graphics.color -> unit = <fun>
```

The function of displaying a box consists of three parts: drawing the first edge, drawing the second edge and drawing the interior of the box.

```
# let draw_box bcf =
     let x1 = bcf.x and y1 = bcf.y in
     let x2 = x1+bcf.w and y2 = y1+bcf.h in
     let ix1 = x1+bcf.bw and ix2 = x2-bcf.bw
     and iy1 = y1+bcf.bw and iy2 = y2-bcf.bw in
     let border1 g =
       Graphics.set_color g;
       Graphics.fill_poly
         [| (x1,y1);(ix1,iy1);(ix2,iy1);(ix2,iy2);(x2,y2);(x2,y1) |]
```

```
    in
    let border2 g =
      Graphics.set_color g;
      Graphics.fill_poly
        [| (x1,y1);(ix1,iy1);(ix1,iy2);(ix2,iy2);(x2,y2);(x1,y2) |]
    in
    Graphics.set_color bcf.b_col;
    ( match bcf.r with
          Top  →
            Graphics.fill_rect ix1 iy1 (ix2-ix1) (iy2-iy1);
            border1 bcf.b1_col;
            border2 bcf.b2_col
        | Bot  →
            Graphics.fill_rect ix1 iy1 (ix2-ix1) (iy2-iy1);
            border1 bcf.b2_col;
            border2 bcf.b1_col
        | Flat →
            Graphics.fill_rect x1 y1 bcf.w bcf.h );
    draw_box_outline bcf Graphics.black;;
val draw_box : box_config -> unit = <fun>
```

The outline of boxes is highlighted in black. Erasing a box fills the area it covers with the background color.

```
# let erase_box bcf =
     Graphics.set_color bcf.b_col;
     Graphics.fill_rect (bcf.x+bcf.bw) (bcf.y+bcf.bw)
                        (bcf.w-(2*bcf.bw)) (bcf.h-(2*bcf.bw));;
val erase_box : box_config -> unit = <fun>
```

Finally, we define a function for displaying a character string at the left, right or in the middle of the box. We use the type *position* to describe the placement of the string.

```
# type position = Left | Center | Right;;
type position = | Left | Center | Right
# let draw_string_in_box pos str bcf col =
      let (w, h) = Graphics.text_size str in
      let ty = bcf.y + (bcf.h-h)/2 in
      ( match pos with
            Center → Graphics.moveto (bcf.x + (bcf.w-w)/2) ty
          | Right  → let tx = bcf.x + bcf.w - w - bcf.bw - 1 in
                     Graphics.moveto tx ty
          | Left   → let tx = bcf.x + bcf.bw + 1 in Graphics.moveto tx ty  );
      Graphics.set_color col;
      Graphics.draw_string str;;
val draw_string_in_box :
  position -> string -> box_config -> Graphics.color -> unit = <fun>
```

Example: drawing of a game We illustrate the use of boxes by displaying the position of a game of type "tic-tac-toe" as shown in figure 5.6. To simplify the creation of boxes, we predefine colors.

```
# let set_gray x =  (Graphics.rgb x x x);;
val set_gray : int -> Graphics.color = <fun>
# let gray1= set_gray 100 and gray2= set_gray 170 and gray3= set_gray 240;;
val gray1 : Graphics.color = 6579300
val gray2 : Graphics.color = 11184810
val gray3 : Graphics.color = 15790320
```

We define a function for creating a grid of boxes of same size.

```
# let rec create_grid nb_col n sep b  =
    if n < 0 then []
    else
      let px = n mod nb_col and py = n / nb_col in
      let nx = b.x +sep + px*(b.w+sep)
      and ny = b.y +sep + py*(b.h+sep) in
      let b1 = {b with x=nx; y=ny} in
      b1 :: (create_grid nb_col (n-1) sep b);;
val create_grid : int -> int -> int -> box_config -> box_config list = <fun>
```

And we create the vector of boxes:

```
# let vb =
    let b =  {x=0; y=0; w=20;h=20; bw=2;
              b1_col=gray1; b2_col=gray3; b_col=gray2; r=Top} in
    Array.of_list (create_grid 5 24 2  b);;
val vb : box_config array =
  [|{x=90; y=90; w=20; h=20; bw=2; r=Top; b1_col=6579300; b2_col=15790320;
     b_col=11184810};
    {x=68; y=90; w=20; h=20; bw=2; r=Top; b1_col=6579300; b2_col=15790320;
     b_col=...};
    ...|]
```

Figure 5.6 corresponds to the following function calls:

```
# Array.iter draw_box vb;
  draw_string_in_box Center "X" vb.(5) Graphics.black;
  draw_string_in_box Center "X" vb.(8) Graphics.black;
  draw_string_in_box Center "O" vb.(12) Graphics.yellow;
  draw_string_in_box Center "O" vb.(11) Graphics.yellow;;
- : unit = ()
```

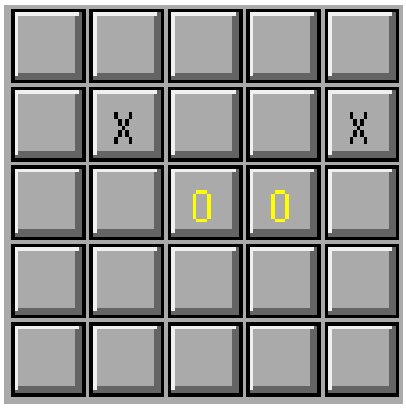


Figure 5.6: Displaying of boxes with text.

Animation

The animation of graphics on a screen reuses techniques of animated drawings. The major part of a drawing does not change, only the animated part must modify the color of its constituent pixels. One of the immediate problems we meet is the speed of animation. It can vary depending on the computational complexity and on the execution speed of the processor. Therefore, to be portable, an application containing animated graphics must take into account the speed of the processor. To get smooth rendering, it is advisable to display the animated object at the new position, followed by the erasure of the old one and taking special care with the intersection of the old and new regions.

Moving an object We simplify the problem of moving an object by choosing objects of a simple shape, namely rectangles. The remaining difficulty is knowing how to redisplay the background of the screen once the object has been moved.

We try to make a rectangle move around in a closed space. The object moves at a certain speed in directions X and Y. When it encounters a border of the graphical window, it bounces back depending on the angle of impact. We assume a situation without overlapping of the new and old positions of the object. The function `calc_pv` computes the new position and the new velocity from an old position `(x,y)`, the size of the object `(sx,sy)` and from the old speed `(dx,dy)`, taking into account the borders of the window.

```
# let calc_pv (x,y) (sx,sy) (dx,dy) =
    let nx1 = x+dx     and  ny1 = y + dy
    and nx2 = x+sx+dx  and  ny2 = y+sy+dy
    and ndx = ref dx   and  ndy = ref dy
    in
      ( if (nx1 < 0) || (nx2 >= Graphics.size_x()) then ndx := -dx );
```

```
      ( if (ny1 < 0) || (ny2 >= Graphics.size_y()) then ndy := -dy );
      ((x+ !ndx, y+ !ndy), (!ndx, !ndy));;
val calc_pv :
  int * int -> int * int -> int * int -> (int * int) * (int * int) = <fun>
```

The function `move_rect` moves the rectangle given by `pos` and `size n` times, the trajectory being indicated by its `speed` and by taking into account the borders of the space. The trace of movement which one can see in figure 5.7 is obtained by inversion of the corresponding bitmap of the displaced rectangle.

```
# let move_rect pos size speed n =
    let (x, y) = pos and (sx,sy) = size in
    let mem = ref (Graphics.get_image x y sx sy) in
    let rec move_aux x y speed n =
     if n = 0 then Graphics.moveto x y
     else
      let ((nx,ny),n_speed) = calc_pv (x,y) (sx,sy) speed
      and old_mem = !mem in
       mem := Graphics.get_image nx ny sx sy;
       Graphics.set_color Graphics.blue;
       Graphics.fill_rect nx ny sx sy;
       Graphics.draw_image (inv_image old_mem) x y;
       move_aux nx ny n_speed (n-1)
    in move_aux x y speed n;;
val move_rect : int * int -> int * int -> int * int -> int -> unit = <fun>
```

The following code corresponds to the drawings in figure 5.7. The first is obtained on a uniformly red background, the second by moving the rectangle across the image of Jussieu.

```
# let anim_rect () =
    Graphics.moveto 105 120;
    Graphics.set_color Graphics.white;
    Graphics.draw_string "Start";
    move_rect (140,120) (8,8) (8,4) 150;
    let (x,y) = Graphics.current_point() in
     Graphics.moveto (x+13) y;
    Graphics.set_color Graphics.white;
    Graphics.draw_string "End";;
val anim_rect : unit -> unit = <fun>
# anim_rect();;
- : unit = ()
```

The problem was simplified, because there was no intersection between two successive positions of the moved object. If this is not the case, it is necessary to write a function that computes this intersection, which can be more or less complicated depending on

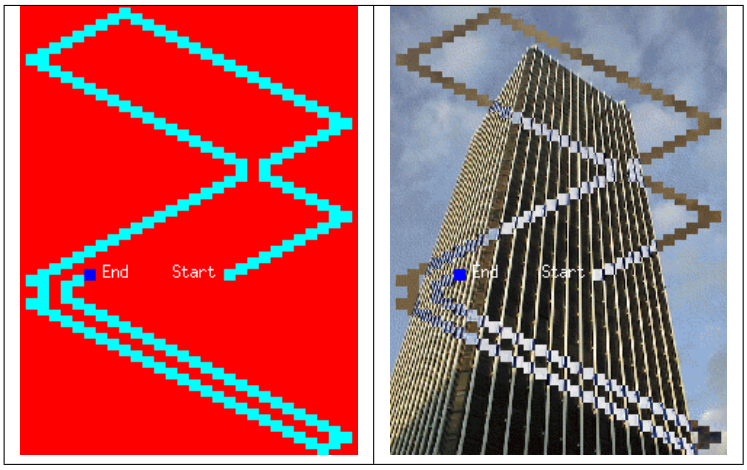


Figure 5.7: Moving an object.

the form of the object. In the case of a square, the intersection of two squares yields a rectangle. This intersection has to be removed.

Events

The handling of events produced in the graphical window allows interaction between the user and the program. `Graphics` supports the treating of events like keystrokes, mouse clicks and movements of the mouse.

The programming style therefore changes the organization of the program. It becomes an infinite loop waiting for events. After handling each newly triggered event, the program returns to the infinite loop except for events that indicate program termination.

Types and functions for events

The main function for waiting for events is `wait_next_event` of type *`event list -> status`*.

The different events are given by the sum type *`event`*.

```
type event = Button_down | Button_up | Key_pressed | Mouse_motion | Poll;;
```

The four main values correspond to pressing and to releasing a mouse button, to movement of the mouse and to keystrokes. Waiting for an event is a blocking operation except if the constructor `Poll` is passed in the event list. This function returns a value of type *`status`*:

```
type status =
  { mouse_x : int;
    mouse_y : int;
```

```
    button : bool;
    keypressed : bool;
    key : char};;
```

This is a record containing the position of the mouse, a Boolean which indicates whether a mouse button is being pressed, another Boolean for the keyboard and a character which corresponds to the pressed key. The following functions exploit the data contained in the event record:

- `mouse_pos`: *`unit -> int * int`*: returns the position of the mouse with respect to the window. If the mouse is placed elsewhere, the coordinates are outside the borders of the window.
- `button_down`: *`unit -> bool`*: indicates pressing of a mouse button.
- `read_key`: *`unit -> char`*: fetches a character typed on the keyboard; this operation blocks.
- `key_pressed`: *`unit -> bool`*: indicates whether a key is being pressed on the keyboard; this operation does not block.

The handling of events supported by `Graphics` is indeed minimal for developing interactive interfaces. Nevertheless, the code is portable across various graphical systems like Windows, MacOS or X-Windows. This is the reason why this library does not take into account different mouse buttons. In fact, the Mac does not even possess more than one. Other events, such as exposing a window or changing its size are not accessible and are left to the control of the library.

Program skeleton

All programs implementing a graphical user interface make use of a potentially infinite loop waiting for user interaction. As soon as an action arrives, the program executes the job associated with this action. The following function possesses five parameters of functionals. The first two serve for starting and closing the application. The next two arguments handle keyboard and mouse events. The last one permits handling of exceptions that escape out of the different functions of the application. We assume that the events associated with terminating the application raise the exception `End`.

```
# exception End;;
exception End
# let skel f_init f_end f_key f_mouse f_except =
    f_init ();
    try
        while true do
          try
            let s = Graphics.wait_next_event
                      [Graphics.Button_down; Graphics.Key_pressed]
            in if s.Graphics.keypressed then f_key s.Graphics.key
               else if s.Graphics.button
                    then f_mouse s.Graphics.mouse_x s.Graphics.mouse_y
```

```
          with
               End → raise End
             | e   → f_except e
        done
    with
        End  → f_end ();;
val skel :
  (unit -> 'a) ->
  (unit -> unit) ->
  (char -> unit) -> (int -> int -> unit) -> (exn -> unit) -> unit = <fun>
```

Here, we use the skeleton to implement a mini-editor. Touching a key displays the typed character. A mouse click changes the current point. The character '&' exits the program. The only difficulty in this program is line breaking. We assume as simplification that the height of characters does not exceed twelve pixels.

```
# let next_line () =
    let (x,y) = Graphics.current_point()
    in if y>12 then Graphics.moveto 0 (y-12)
       else Graphics.moveto 0 y;;
val next_line : unit -> unit = <fun>
# let handle_char c = match c with
      '&'  → raise End
    | '\n' → next_line ()
    | '\r' → next_line ()
    |  _   → Graphics.draw_char c;;
val handle_char : char -> unit = <fun>
# let go () = skel
    (fun () → Graphics.clear_graph ();
              Graphics.moveto 0 (Graphics.size_y() -12) )
    (fun () → Graphics.clear_graph())
    handle_char
    (fun x y → Graphics.moveto x y)
    (fun e → ());;
val go : unit -> unit = <fun>
```

This program does not handle deletion of characters by pressing the key `DEL`.

Example: telecran

Telecran is a little drawing game for training coordination of movements. A point appears on a slate. This point can be moved in directions X and Y by using two control buttons for these axes without ever releasing the pencil. We try to simulate this behavior to illustrate the interaction between a program and a user. To do this we reuse the previously described skeleton. We will use certain keys of the keyboard to indicate movement along the axes.

We first define the type *state*, which is a record describing the size of the slate in terms of the number of positions in X and Y, the current position of the point and the scaling factor for visualization, the color of the trace, the background color and the color of the current point.

```
# type state = {maxx:int; maxy:int; mutable x : int; mutable y :int;
                scale:int;
                bc : Graphics.color;
                fc: Graphics.color; pc : Graphics.color};;
```

The function `draw_point` displays a point given its coordinates, the scaling factor and its color.

```
# let draw_point x y s c =
    Graphics.set_color c;
    Graphics.fill_rect (s*x) (s*y) s s;;
val draw_point : int -> int -> int -> Graphics.color -> unit = <fun>
```

All these functions for initialization, handling of user interaction and exiting the program receive a parameter corresponding to the state. The first four functions are defined as follows:

```
# let t_init s () =
    Graphics.open_graph (" " ^ (string_of_int (s.scale*s.maxx)) ^
      "x" ^ (string_of_int (s.scale*s.maxy)));
    Graphics.set_color s.bc;
    Graphics.fill_rect 0 0 (s.scale*s.maxx+1) (s.scale*s.maxy+1);
    draw_point s.x s.y s.scale s.pc;;
val t_init : state -> unit -> unit = <fun>
# let t_end s () =
    Graphics.close_graph();
    print_string "Good bye..."; print_newline();;
val t_end : 'a -> unit -> unit = <fun>
# let t_mouse s x y = ();;
val t_mouse : 'a -> 'b -> 'c -> unit = <fun>
# let t_except s ex = ();;
val t_except : 'a -> 'b -> unit = <fun>
```

The function `t_init` opens the graphical window and displays the current point, `t_end` closes this window and displays a message, `t_mouse` and `t_except` do not do anything. The program handles neither mouse events nor exceptions which may accidentally arise during program execution. The important function is the one for handling the keyboard `t_key`:

```
# let t_key s c =
   draw_point s.x s.y s.scale s.fc;
    (match c with
    '8' → if s.y < s.maxy then  s.y <- s.y + 1;
  | '2' → if s.y > 0 then s.y <- s.y - 1
```

```
  | '4' → if s.x > 0 then s.x <- s.x - 1
  | '6' → if s.x < s.maxx then s.x <- s.x + 1
  | 'c' →   Graphics.set_color s.bc;
            Graphics.fill_rect 0 0 (s.scale*s.maxx+1) (s.scale*s.maxy+1);
            Graphics.clear_graph()
  | 'e' → raise End
  | _ → ());
   draw_point s.x s.y s.scale s.pc;;
val t_key : state -> char -> unit = <fun>
```

It displays the current point in the color of the trace. Depending on the character passed, it modifies, if possible, the coordinates of the current point (characters: '2', '4', '6', '8'), clears the screen (character: 'c') or raises the exception `End` (character: 'e'), then it displays the new current point. Other characters are ignored. The choice of characters for moving the cursor comes from the layout of the numeric keyboard: the chosen keys correspond to the indicated digits and to the direction arrows. It is therefore useful to activate the numeric keyboard for the ergonomics of the program.

We finally define a state and apply the skeleton function in the following way:

```
# let stel = {maxx=120; maxy=120; x=60; y=60;
              scale=4; bc=Graphics.rgb 130 130 130;
              fc=Graphics.black; pc=Graphics.red};;
val stel : state =
  {maxx=120; maxy=120; x=60; y=60; scale=4; bc=8553090; fc=0; pc=16711680}
# let slate () =
    skel (t_init stel)  (t_end stel) (t_key stel)
     (t_mouse stel) (t_except stel);;
val slate : unit -> unit = <fun>
```

Calling function `slate` displays the graphical window, then it waits for user interaction on the keyboard. Figure 5.8 shows a drawing created with this program.

A Graphical Calculator

Let's consider the calculator example as described in the preceding chapter on imperative programming (see page 86). We will give it a graphical interface to make it more usable as a desktop calculator.

The graphical interface materializes the set of keys (digits and functions) and an area for displaying results. Keys can be activated using the graphical interface (and the mouse) or by typing on the keyboard. Figure 5.9 shows the interface we are about to construct.

We reuse the functions for drawing boxes as described on page 126. We define the following type:

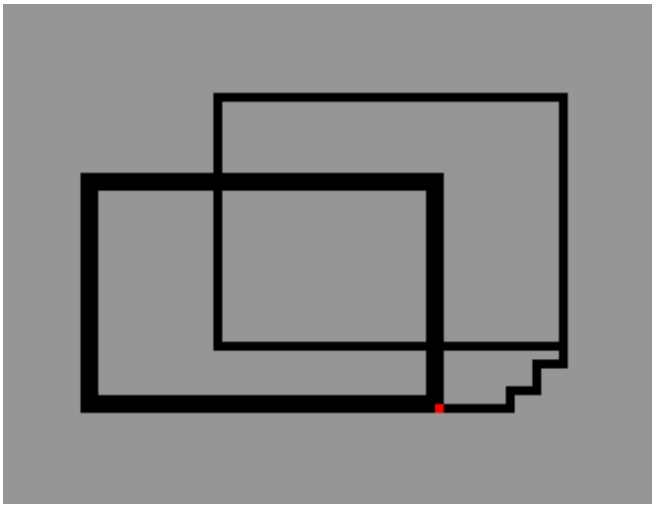


Figure 5.8: Telecran.

Figure 5.9: Graphical calculator.

```
# type calc_state =
    { s : state; k : (box_config * key * string ) list; v : box_config } ;;
```

It contains the state of the calculator, the list of boxes corresponding to the keys and the visualization box. We plan to construct a calculator that is easily modifiable. Therefore, we parameterize the construction of the interface with an association list:

```
# let descr_calc =
    [  (Digit 0,"0");  (Digit 1,"1");  (Digit 2,"2");  (Equals, "=");
       (Digit 3,"3");  (Digit 4,"4");  (Digit 5,"5");  (Plus, "+");
```

```
    (Digit 6,"6");  (Digit 7,"7");  (Digit 8,"8");  (Minus, "-");
    (Digit 9,"9");  (Recall,"RCL"); (Div, "/");     (Times, "*");
    (Off,"AC");     (Store, "STO"); (Clear,"CE/C")
  ] ;;
```

Generation of key boxes At the beginning of this description we construct a list of key boxes. The function `gen_boxes` takes as parameters the description (`descr`), the number of the column (`n`), the separation between boxes (`wsep`), the separation between the text and the borders of the box (`wsepint`) and the size of the board (`wbord`). This function returns the list of key boxes as well as the visualization box. To calculate these placements, we define the auxiliary functions `max_xy` for calculating the maximal size of a list of complete pairs and `max_lbox` for calculating the maximal positions of a list of boxes.

```
# let gen_xy vals comp o  =
    List.fold_left (fun  a (x,y) → comp (fst a) x,comp (snd a) y) o vals ;;
val gen_xy : ('a * 'a) list -> ('b -> 'a -> 'b) -> 'b * 'b -> 'b * 'b = <fun>
# let max_xy vals = gen_xy vals max (min_int,min_int);;
val max_xy : (int * int) list -> int * int = <fun>
# let max_boxl l =
    let bmax (mx,my) b = max mx b.x, max my b.y
    in List.fold_left bmax  (min_int,min_int) l ;;
val max_boxl : box_config list -> int * int = <fun>
```

Here is the principal function `gen_boxes` for creating the interface.

```
# let gen_boxes descr n wsep wsepint wbord  =
    let l_l = List.length descr in
    let nb_lig = if l_l mod n = 0 then l_l / n else l_l / n + 1 in
    let ls = List.map (fun (x,y) → Graphics.text_size y) descr in
    let sx,sy = max_xy  ls in
    let sx,sy= sx+wsepint ,sy+wsepint  in
      let r = ref [] in
        for i=0 to l_l-1 do
          let px = i mod n and py = i / n in
            let b = { x = wsep * (px+1) + (sx+2*wbord) * px  ;
                      y = wsep * (py+1) + (sy+2*wbord) * py ;
                      w = sx; h = sy ; bw = wbord;
                      r=Top;
                      b1_col = gray1;  b2_col = gray3; b_col =gray2}
            in r:= b::!r
        done;
        let mpx,mpy = max_boxl !r in
        let upx,upy = mpx+sx+wbord+wsep,mpy+sy+wbord+wsep in
        let (wa,ha) = Graphics.text_size "        0" in
          let v = { x=(upx-(wa+wsepint +wbord))/2 ; y= upy+ wsep;
                    w=wa+wsepint; h = ha +wsepint; bw = wbord *2; r=Flat ;
```

```
                    b1_col = gray1;  b2_col = gray3; b_col =Graphics.black}
          in
            upx,(upy+wsep+ha+wsepint+wsep+2*wbord),v,
            List.map2 (fun b (x,y) → b,x,y ) (List.rev !r) descr;;
val gen_boxes :
  ('a * string) list ->
  int ->
  int ->
  int -> int -> int * int * box_config * (box_config * 'a * string) list =
  <fun>
```

Interaction Since we would also like to reuse the skeleton proposed on page 133 for interaction, we define the functions for keyboard and mouse control, which are integrated in this skeleton. The function for controlling the keyboard is very simple. It passes the translation of a character value of type ***key*** to the function `transition` of the calculator and then displays the text associated with the calculator state.

```
# let f_key cs c =
    transition cs.s (translation c);
    erase_box cs.v;
    draw_string_in_box Right (string_of_int cs.s.vpr) cs.v Graphics.white ;;
val f_key : calc_state -> char -> unit = <fun>
```

The control of the mouse is a bit more complex. It requires verification that the position of the mouse click is actually in one of the key boxes. For this we first define the auxiliary function `mem`, which verifies membership of a position within a rectangle.

```
# let mem (x,y) (x0,y0,w,h) =
      (x >= x0) && (x< x0+w) && (y>=y0) && ( y<y0+h);;
val mem : int * int -> int * int * int * int -> bool = <fun>
# let f_mouse cs x y =
    try
      let b,t,s =
        List.find (fun (b,_,_) →
                     mem (x,y) (b.x+b.bw,b.y+b.bw,b.w,b.h)) cs.k
      in
        transition cs.s t;
        erase_box cs.v;
        draw_string_in_box Right (string_of_int cs.s.vpr ) cs.v Graphics.white
    with Not_found → ();;
val f_mouse : calc_state -> int -> int -> unit = <fun>
```

The function `f_mouse` looks whether the position of the mouse during the click is really-dwell within one of the boxes corresponding to a key. If it is, it passes the corresponding key to the transition function and displays the result, otherwise it will not do anything.

The function `f_exc` handles the exceptions which can arise during program execution.

```
# let f_exc cs ex =
    match ex with
     Division_by_zero →
       transition cs.s Clear;
       erase_box cs.v;
       draw_string_in_box Right "Div 0" cs.v (Graphics.red)
  | Invalid_key → ()
  | Key_off → raise End
  | _  → raise ex;;
val f_exc : calc_state -> exn -> unit = <fun>
```

In the case of a division by zero, it restarts in the initial state of the calculator and displays an error message on its screen. Invalid keys are simply ignored. Finally, the exception `Key_off` raises the exception `End` to terminate the loop of the skeleton.

Initialization and termination The initialization of the calculator requires calculation of the window size. The following function creates the graphical information of the boxes from a key/text association and returns the size of the principal window.

```
# let create_e k =
       Graphics.close_graph ();
       Graphics.open_graph " 10x10";
       let mx,my,v,lb = gen_boxes k 4 4 5 2 in
       let s = {lcd=0; lka = false; loa = Equals; vpr = 0; mem = 0}   in
         mx,my,{s=s; k=lb;v=v};;
val create_e : (key * string) list -> int * int * calc_state = <fun>
```

The initialization function makes use of the result of the preceding function.

```
# let f_init mx my cs () =
         Graphics.close_graph();
         Graphics.open_graph (" "^(string_of_int mx)^"x"^(string_of_int my));
         Graphics.set_color gray2;
         Graphics.fill_rect 0 0 (mx+1) (my+1);
         List.iter (fun (b,_,_) → draw_box b) cs.k;
         List.iter
           (fun (b,_,s) → draw_string_in_box Center s b Graphics.black) cs.k ;
         draw_box cs.v;
         erase_box cs.v;
         draw_string_in_box Right "hello" cs.v (Graphics.white);;
val f_init : int -> int -> calc_state -> unit -> unit = <fun>
```

Finally the termination function closes the graphical window.

```
# let f_end e () = Graphics.close_graph();;
val f_end : 'a -> unit -> unit = <fun>
```

The function `go` is parameterized by a description and starts the interactive loop.

```
# let go descr =
    let mx,my,e = create_e descr in
      skel (f_init mx my e) (f_end e) (f_key e) (f_mouse e) (f_exc e);;
val go : (key * string) list -> unit = <fun>
```

The call to `go descr_calc` corresponds to the figure 5.9.

Exercises

Polar coordinates

Coordinates as used in the library `Graphics` are Cartesian. There a line segment is represented by its starting point `(x0,y0)` and its end point `(x1,y1)`. It can be useful to use polar coordinates instead. Here a line segment is described by its point of origin `(x0,y0)`, a length (radius) `(r)` and an angle `(a)`. The relation between Cartesian and Polar coordinates is defined by the following equations:

$$\begin{cases} x_1 &= x_0 + r * \cos(a) \\ y_1 &= y_0 + r * \sin(a) \end{cases}$$

The following type defines the polar coordinates of a line segment:

```
# type seg_pol = {x:float; y:float; r:float; a:float};;
type seg_pol = { x: float; y: float; r: float; a: float }
```

1. Write the function `to_cart` that converts polar coordinates to Cartesian ones.
2. Write the function `draw_seg` which displays a line segment defined by polar coordinates in the reference point of `Graphics`.
3. One of the motivations behind polar coordinates is to be able to easily apply transformations to line segments. A translation only modifies the point of origin, a rotation only affects the angle field and modifying the scale only changes the length field. Generally, one can represent a transformation as a triple of floats: the first represents the translation (we do not consider the case of translating the second point of the line segment here), the second the rotation and the third the scaling factor. Define the function `app_trans` which takes a line segment in polar coordinates and a triple of transformations and returns the new segment.
4. One can construct recursive drawings by iterating transformations. Write the function `draw_r` which takes as arguments a line segment `s`, a number of iterations `n`, a list of transformations and displays all the segments resulting from the transformations on `s` iterated up to `n`.
5. Verify that the following program does produce the images in figure 5.10.

   ```
   let pi = 3.1415927 ;;
   ```

```
let s = {x=100.; y= 0.; a= pi /. 2.; r = 100.} ;;
draw_r s 6 [ (-.pi/.2.),0.6,1.; (pi/.2.), 0.6,1.0] ;;
Graphics.clear_graph();;
draw_r s 6 [(-.pi /. 6.), 0.6, 0.766;
            (-.pi /. 4.), 0.55, 0.333;
            (pi /. 3.), 0.4, 0.5 ] ;;
```

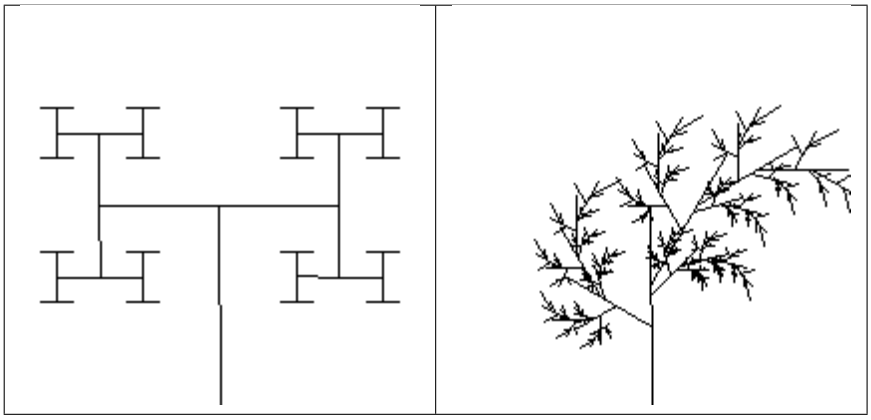


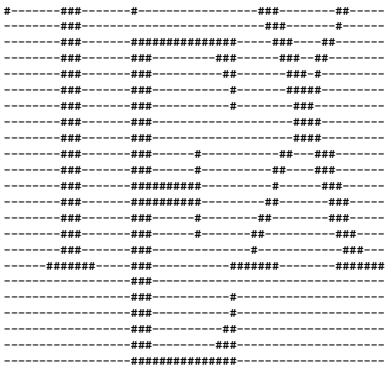
Figure 5.10: Recursive drawings.

Bitmap editor

We will attempt to write a small bitmap editor (similar to the command `bitmap` in X-window). For this we represent a bitmap by its dimensions (width and height), the pixel size and a two-dimensional table of booleans.

1. Define a type *bitmap_state* describing the information necessary for containing the values of the pixels, the size of the bitmap and the colors of displayed and erased points.
2. Write a function for creating bitmaps (`create_bitmap`) and for displaying bitmaps (`draw_bitmap`) .
3. Write the functions `read_bitmap` and `write_bitmap` which respectively read and write in a file passed as parameter following the ASCII format of X-window. If the file does not exist, the function for reading creates a new bitmap using the function `create_bitmap`. A displayed pixel is represented by the character `#`, the absence of a pixel by the character `-`. Each line of characters represents a line of the bitmap. One can test the program using the functions `atobm` and `bmtoa` of X-window, which convert between this ASCII format and the format of bitmaps created by the command `bitmap`. Here is an example.

```
##################------------#######--------######
##################--------------###------------##--
###-----###-----###--------------###------------#---
##------###------##--------------###-----------##---
#-------###------#---------------###---------##----
```



4. We reuse the skeleton for interactive loops on page 133 to construct the graphical interface of the editor. The human-computer interface is very simple. The bitmap is permanently displayed in the graphical window. A mouse click in one of the slots of the bitmap inverts its color. This change is reflected on the screen. Pressing the key 'S' saves the bitmap in a file. The key 'Q' terminates the program.
   - Write a function `start` of type *`bitmap_state -> unit -> unit`* which opens a graphical window and displays the bitmap passed as parameter.
   - Write a function `stop` that closes the graphical window and exits the program.
   - Write a function `mouse` of type *`bitmap_state -> int -> int -> unit`* which modifies the pixel state corresponding to the mouse click and displays the change.
   - Write a function `key` of type *`string -> bitmap_state -> char -> unit`* which takes as arguments the name of a file, a bitmap and the char of the pressed key and executes the associated actions: saving to a file for the key 'S' and raising of the exception `End` for the key 'Q'.
5. Write a function `go` which takes the name of a file as parameter, loads the bitmap, displays it and starts the interactive loop.

Earth worm

The earth worm is a small, longish organism of a certain size which grows over time while eating objects in a world. The earth worm moves constantly in one direction. The only actions allowing a player to control it are changes in direction. The earth worm vanishes if it touches a border of the world or if it passes over a part of its body. It is most often represented by a vector of coordinates with two principal indices: its head and its tail. A move will therefore be computed from the new coordinates of its head, will display it and erase the tail. A growth step only modifies its head without affecting the tail of the earth worm.

1. Write the Objective Caml type or types for representing an earth worm and the world where it evolves. One can represent an earth worm by a queue of its coordinates.

2. Write a function for initialization and displaying an earth worm in a world.
3. Modify the function `skel` of the skeleton of the program which causes an action at each execution of the interactive loop, parameterized by a function. The treatment of keyboard events must not block.
4. Write a function `run` which advances the earth worm in the game. This function raises the exception `Victory` (if the worm reaches a certain size) and `Loss` if it hits a full slot or a border of the world.
5. Write a function for keyboard interaction which modifies the direction of the earth worm.
6. Write the other utility functions for handling interaction and pass them to the new skeleton of the program.
7. Write the initiating function which starts the application.

Summary

This chapter has presented the basic notions of graphics programming and event-driven programming using the `Graphics` library in the distribution of Objective Caml. After having explained the basic graphical elements (colors, drawing, filling, text and bitmaps) we have approached the problem of animating them. The mechanism of handling events in `Graphics` was then described in a way that allowed the introduction of a general method of handling user interaction. This was accomplished by taking a game as model for event-driven programming. To improve user interactions and to provide interactive graphical components to the programmer, we have developed a new library called `Awi`, which facilitates the construction of graphical interfaces. This library was used for writing the interface to the imperative calculator.

To learn more

Although graphics programming is naturally event-driven, the associated style of programming being imperative, it is not only possible but also often useful to introduce more functional operators to manipulate graphical objects. A good example comes from the use of the `MLgraph` library,

Link: http://www.pps.jussieu.fr/~cousinea/MLgraph/mlgraph.html

which implements the graphical model of PostScript and proposes functional operators to manipulate images. It is described in [CC92, CS94] and used later in [CM98] for the optimized placement of trees to construct drawings in the style of Escher.

One interesting characteristic of the `Graphics` library is that it is portable to the graphical interfaces of Windows, MacOS and Unix. The notion of virtual bitmaps can be found in several languages like Le_Lisp and more recently in Java. Unfortunately, the `Graphics` library in Objective Caml does not possess interactive components for

the construction of interfaces. One of the applications described in part II of this book contains the first bricks of the `Awi` library. It is inspired by the *Abstract Windowing Toolkit* of the first versions of Java. One can perceive that it is relatively easy to extend the functionality of this library thanks to the existence of functional values in the language. Therefore chapter 16 compares the adaptation of object oriented programming and functional and modular programming for the construction of graphical interfaces. The example of `Awi` is functional and imperative, but it is also possible to only use the functional style. This is typically the case for purely functional languages. We cite the systems `Fran` and `Fudget` developed in Haskell and derivatives. The system `Fran` permits construction of interactive animations in 2D and 3D, which means with events between animated objects and the user.

Link: http://www.research.microsoft.com/~conal/fran/

The `Fudget` library is intended for the construction of graphical interfaces.

Link: http://www.cs.chalmers.se/ComputingScience/Research/Functional/Fudgets/

One of the difficulties when one wants to program a graphical interface for ones application is to know which of the numerous existing libraries to choose. It is not sufficient to determine the language and the system to fix the choice of the tool. For Objective Caml there exist several more or less complete ones:

- the encapsulation of `libX`, for X-Windows;
- the `librt` library, also for X-Windows;
- `ocamltk`, an adaptation of `Tcl/Tk`, portable;
- `mlgtk`, an adaptation of `Gtk`, portable.

We find the links to these developments in the "Caml Hump":

Link: http://caml.inria.fr/hump.html

Finally, we have only discussed programming in 2D. The tendency is to add one dimension. Functional languages must also respond to this necessity, perhaps in the model of VRML or the Java 3D-extension. In purely functional languages the system `Fran` offers interesting possibilities of interaction between *sprites*. More closely to Objective Caml one can use the `VRcaML` library or the development environment SCOL.

The `VRcaML` library was developed in the manner of `MLgraph` and integrates a part of the graphical model of VRML in Objective Caml.

Link: http://www.pps.jussieu.fr/~emmanuel/Public/enseignement/VRcaML

One can therefore construct animated scenes in 3D. The result is a VRML-file that can be directly visualized.

Still in the line of Caml, the language SCOL is a functional communication language with important libraries for 2D and 3D manipulations, which is intended as environment for people with little knowledge in computer science.

Link: http://www.cryo-networks.com

The interest in the language SCOL and its development environment is to be able to create distributed applications, e.g. client-server, thus facilitating the creation of Internet sites. We present distributed programming in Objective Caml in chapter 20.