Canvas

Jon Sporring

August 2022

1 Introduction

Canvas is a small F# library by which you can open a window and manipulates its pixels. In the following, we will describe how to write a simple graphics application. Consider the following example:

```
#r "nuget:DIKU.Canvas, 1.0"

open Canvas

let w = 600;
let h = 400;
let C = create w h
do setLine C black (0,0) (w-1,h-1)
do show C "My First Canvas"
```

Saving the above in a file `myFirstCanvas.fsx` and running it with `dotnet fsi myFirst` in the terminal, should create a window on the screen with a black line going diagonally from the top left to the bottom right shown in Figure 1. The program terminates, when you close the window or press `<ctrl-c>` in the terminal. The program first creates a canvas with the `create`, then it draws a line on the canvas with `setLine`, and finally, it shows the canvas on the screen, i.e., it tells the operating system to open a window and display the canvas.

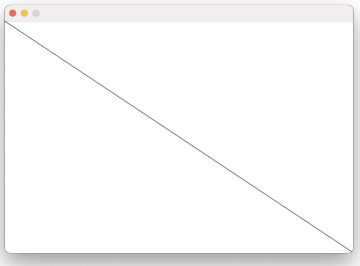

Figure 1: A demonstration of the output of Canvas.

2 Drawing in canvas

A canvas is a 2-dimensional buffer that stores pixels, and it is created using the function,

```
create : int -> int -> canvas
```

It takes a width and a height and returns a canvas. It has an integer coordinate system, whose origin is in the top left corner, the first coordinate increases to the right, and the second coordinate increases downwards I.e., for a canvas with width and height (w, h) then a canvas coordinate (i, j) has $0 \leq i < w$ and $0 \leq j < h$, such that $(0, 0)$ is the top-left corner and $(w - 1, h - 1)$ is the bottom-right corner. For a given canvas, its width and height can be obtained with the functions,

```
width       : canvas -> int
height      : canvas -> int
```

Each pixel is a combination of 4 values: red, green, blue, and alpha, also known as RGBA. Each value is an unsigned, 8-bit integer, the red-green-blue in combination gives represents all the colors, the screen can display, and alpha is the degree of transparency, such that $(r, g, b, a) = (255, 0, 0, 255)$ is a red and non-transparent pixel, and $(r, g, b, a) = (0, 255, 0, 128)$ is a semi-transparent green pixel. You can manipulate colors with

```
fromRgb   : int * int * int -> color
fromArgb  : int * int * int * int -> color
fromColor : color -> int * int * int * int
```

and there are a number of builtin colors: `red`, `blue`, `green`, `yellow`, `lightgrey`, `white`, and `black`.

The value of a pixel can be read using

```
getPixel    : canvas -> int * int -> color
```

which is a pixel that takes an existing canvas and a coordinate pair (2-tuple) and returns the pixel's color. To write a pixel, use

```
setPixel    : canvas -> color -> int * int -> unit
```

Note that this function has a side effect! It takes a canvas, a color, and a coordinate pair and updates the canvas. Extensions of `setPixel` are available for drawing lines and boxes:

```
setLine     : canvas -> color -> int * int -> int * int -> unit
setBox      : canvas -> color -> int * int -> int * int -> unit
setFillBox  : canvas -> color -> int * int -> int * int -> unit
```

Again, these functions have the side-effect that the canvas is updated. They all take a canvas, a color, and two coordinate pairs. Assuming the coordinate pairs are `p1` and `p2`, then `setLine` draws a line from `p1` to `p2`, `setBox` draws the contour of a box, where `p1` is the top left corner and `p2` is the bottom right corner. The function `setFillBox` draws a filled box with `p1` and `p2` being its corners.

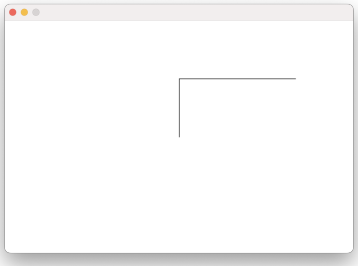

Figure 2: A demonstration of turtle graphics in Canvas.

Manipulating pixels only manipulates an internal representation of the canvas but does not change anything on the screen. When the canvas has been drawn upon, it can be shown with

```
show      : canvas -> string -> unit
```

which takes a canvas and a title string, and opens a window that displays the canvas. The function returns when the user closes the window. It can also be stopped by pressing `<ctrl-c>`.

3 Turtle graphics

Turtle graphics is an alternative method of drawing line drawings in canvas. The turtle is a metaphor for a small animal, which can do simple actions, and in Canvas the turtle can turn and move with `Turn`, `Move`, and left the pen up and down with `PenUp` and `PenDown`. The turtle starts in the center of the screen pointing upwards. An example of a turtle program is as follows:

```
#r "nuget:DIKU.Canvas, 1.0"

open Canvas

let cmd = [Move 100; Turn 90; Move 200]
let w = 600;
let h = 400;
do turtleDraw (w,h) "My First Canvas" cmd
```

This should produce a window as shown in Figure 2. The commands are given as a list of `turtleCmd`, which can be one of either

```
SetColor of color
Turn of int
Move of int
PenUp
PenDown
```

The SetColor is used to set a color, e.g., `SetColor green` will make the next line, the turtle draws, green. Turning the turtle is done in degrees, e.g., `Turn 30`

will make the turtle point in a new direction which is 30 degrees clockwise turn compared to the present direction. `Move 15` will make the turtle move 15 pixels in the direction, it is pointing. If the pen is down, as it is per default, then it will also draw a line on its path. `PenUp` and `PenDown` lowers or raises the pen. The turtle's starting point is the center of the Canvas pointing up, with a black pen down. The commands are given as a list, i.e., elements are separated by semicolons, and the commands are executed from left to right. The function

```
turtleDraw : int*int -> string -> turtleCmd list -> unit
```

makes a canvas with a list of commands. `turtleDraw` takes a width-height pair, the name of a window, and the list of commands, and opens a window on the screen. The function returns, once the window is closed by the user.

4 Interaction with the user

Canvas can be programmed to respond to user input by listining to the keyboard. To do this, we use `runApp` instead of `show`. The function `runApp` has the following interface:

```
// start an app that can listen to key-events
runApp    : string -> int -> int
             -> (int -> int -> 's -> canvas)
             -> ('s -> key -> 's option)
             -> 's -> unit
```

It takes a window title as a string, a width and a height, a function that can draw a create a canvas with content, and a function that can listen and react to keyboard input and a state. Consider the following example:

```
#r "nuget:DIKU.Canvas, 1.0"

open Canvas

type state = color

let draw w h (s:state) =
  let C = Canvas.create w h
  setFillBox C s (0,0) (w-1,h-1)
  C

let react (s:state) (k:Canvas.key) : state option =
    match getKey k with
        | LeftArrow ->
            printfn "Going red!"
            Some red
        | RightArrow ->
            printfn "Going blue!"
            Some blue
        | _ -> None

do runApp "ColorTest" 600 600 draw react red
```

When the program is run, it opens a window and paints everything inside it red. When the user presses the right arrow key, the screen is painted blue, and "Going blue!" is printed on the terminal, and when the left arrow key is pressed, then the screen is painted red, and "Going red!" is printed in the terminal. The program is stopped by closing the window or pressing `<ctrl-c>`.

The program works as follows: `runApp` maintains an internal state, whose type is defined by the user. Here `state` can be defined to be anything but here it is a color. Let us call this `s`. The initial value of this state is set by `runApp`'s last argument, here `red`. After the state has been initialized, `runApp` calls `draw w h s`, i.e., `draw 600 600 red`, a window is created, and the canvas is shown on the screen. Then it waits for the user to press any key. The result is put in internal `key`-value, let us call it `k`, and `runApp` calls the user-specified `react` function with `react s k`. This function must return an `option` type, i.e., either `Some t`, where `t` is any state, or `None`. If `react` returns `None`, then nothing happens. However, if `react` returns `Some t`, then `runApp`'s internal state is changed to `t`, `draw` is called again but now with `t` as the state, and the window's content is updated with the new canvas. To help all of this, the function

```
getKey : key -> ImgUtilKey
```

converts the internal key-value to one of `Unknown`, `DownArrow`, `UpArrow`, `LeftArrow`, `RightArrow`, and `Space`.