Learning to program with F#

Jon Sporring

November 13, 2016

Contents

Chapter 13

Graphical User Interfaces

A *command-line interface* (*CLI*) is a method for communicating with the user through text. In contrast, a *graphical user interface* (*GUI*) extends the ways of communicating with the user to also include organising the screen space in windows, icons, and other visual elements, and a typical way to activate these elements are through a pointing device such as the mouse or by touch. Some of these elements may themselves be textual, and thus most operating systems offers access to a command-line interface in a window alongside other interface types.

- command-line interface
- CLI
- graphical user interface
- GUI

Fsharp includes a number of implementations of graphical user interfaces, but at time of writing only *WinForms* is supported on both the Microsoft .Net and the Mono platform, and hence, WinForms will be the subject of the following chapter.

- WinForms

WinForms is designed for *event driven programs*, which spends most time waiting for the user to perform an action, called and event, and for each event has a set of predefined responses to be performed by the program. For example, Figure 13.1 shows the program Safari, which is a graphical user interface for accessing web-servers. The program present information to the user in terms of text and images, and has areas that when activated by clicking with a mouse or similar allows the user to, e.g., go to other web-pages by type URL, to follow hyperlinks, and to generate new pages by entering search queries.

- event driven programs

13.1 Drawing primitives in Windows

WinForms is based on two namespaces: `System.Windows.Forms` and `System.Drawing`. To start making a graphical display on the screen, the first thing to do is open a window, which acts as a reserved screen space for our output. With WinForms, this may be done as shown in Listing 13.1, and the result is shown in Figure 13.3.

Listing 13.1, winforms/openWindow.fsx:
Create the window and turn over control to the operating system. Use `win.Show()` on Microsoft Windows instead.

```
// Create a window
let win = new System.Windows.Forms.Form ()
// Start the event-loop. Use "win.Show()" on Microsoft Windows instead.
System.Windows.Forms.Application.Run win
```

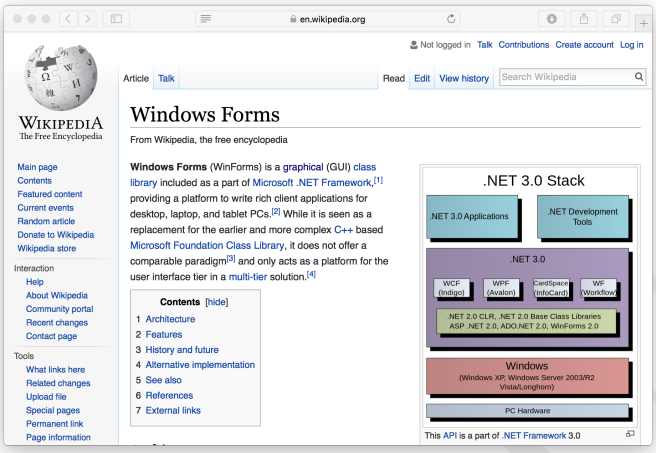

Figure 13.1: A web-browser is a graphical user interface for accessing a web-server and interacting with its services. Here the browser is showing the page `https://en.wikipedia.org/wiki/Windows_Forms` at time of writing.

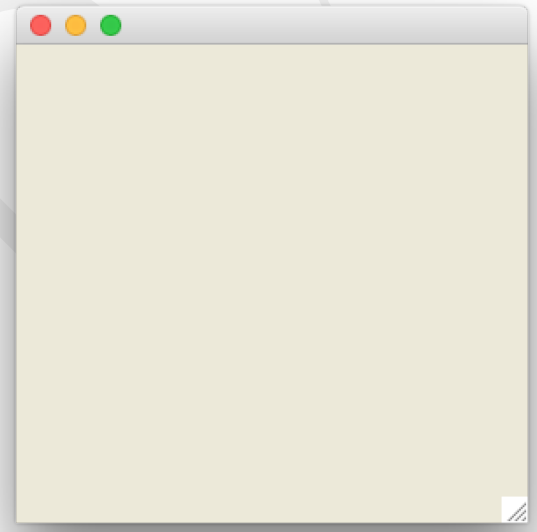

Figure 13.2: Result of running listing Listing 13.1.

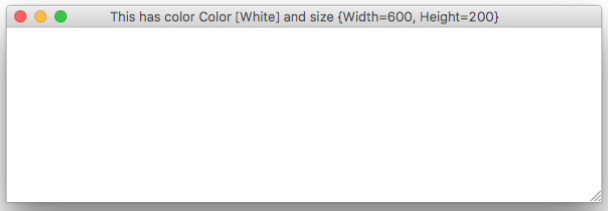

Figure 13.3: Result of running listing Listing 13.2.

The `new System.Windows.Forms.Form ()` creates an object (See Chapter 20), but does not display the window on the screen. When the function `System.Windows.Forms.Application.Run` is applied to the object, then the control is handed over to the WinForm's *event-loop*, which continues until the window is closed by, e.g., pressing the icon designated by the operating system. On the mac OSX that is the red button in the top left corner of the window frame, and on Window it is the cross on the top right corner of the window frame.

· event-loop

The window, which WinForms calls a form, has a long list of *methods* and *properties*. E.g., the background color may be set by `BackColor`, the title of the window may be set by `Text`, and you may get and set the size of the window with the `Size`. This is demonstrated in Listing

· methods
· properties

Listing 13.2, winforms/windowAttributes.fsx:
Create the window and changing its properties.

```
// Create a window
let win = new System.Windows.Forms.Form ()
// Set some properties
win.BackColor <- System.Drawing.Color.White
win.Size <- System.Drawing.Size (600, 200)
win.Text <- sprintf "This has color %A and size %A" win.BackColor win.Size
// Start the event-loop. Use "win.Show()" on Microsoft Windows instead.
System.Windows.Forms.Application.Run win
```

These properties have been programmed as *accessors* implying that they may used as mutable variables. The *`System.Drawing.Color`* is a general structure for specifying colors as 4 channels: alpha, red, green, blue, where each channel is an 8 bit unsigned integer, where the alpha channel specifies the transparency of a color, where values 0–255 denotes the range of fully transparent to fully opaque, and the remaining channels denote the amount of red, green, and blue where 0 is none and 255 is full intensity. Any color may be created using the `FromArgb` method, e.g., an opaque red is given by `System.Drawing.Color.FromArgb (255, 255, 0, 0)`. There are also many build-in colors, e.g., the same red color is also a known color and may be obtained as `System.Drawing.Color.Red`. For a given color, then the 4 alpha, red, green, and blue channel's values may be obtained as the `A`, `R`, `G`, `B`, see Listing 13.3

· accessors
· `System.Drawing.Color`

Listing 13.3, drawingColors.fsx:
Defining colors and accessing their values.

```
// open namespace for brevity
open System.Drawing
// Define a color from ARGB
let c = Color.FromArgb (0xFF, 0x7F, 0xFF, 0xD4) //Aquamarine
printfn "The color  %A is (%x, %x, %x, %x)" c c.A c.R c.G c.B
// Define a list of named colors
let colors = [Color.Red; Color.Green; Color.Blue; Color.Black; Color.Gray;
    Color.White]
for col in colors do
  printfn "The color %A is (%x, %x, %x, %x)" col col.A col.R col.G col.B
```

```
The color  Color [A=255, R=127, G=255, B=212] is (ff, 7f, ff, d4)
The color Color [Red] is (ff, ff, 0, 0)
The color Color [Green] is (ff, 0, 80, 0)
The color Color [Blue] is (ff, 0, 0, ff)
The color Color [Black] is (ff, 0, 0, 0)
The color Color [Gray] is (ff, 80, 80, 80)
The color Color [White] is (ff, ff, ff, ff)
```

The `System.Drawing.Size` is a general structure for specifying sizes as height and width pair of integers.

WinForms supports drawing of simple graphics primitives. Simple examples are `System.Drawing.Pen` to specify the color to be drawn, `System.Drawing.Point` to specify a pair of coordinates, and `System.Drawing.Graphics.DrawLine`. `DrawLine` is different than the previous examples since it must be related to a specific device, and it is typically accessed as an event. Displaying graphics in WinForms is performed as the reaction to an event. E.g., windows are created by the program, moved, minimized, occluded by other windows, resized, etc., by the user or the program, and each action may require that the content of the window is refreshed. Thus, we must create a function that WinForms can call, when it determines that the content needs to be redrawn. This is known as a *call-back function*, and it is added to an existing form using the `Paint.Add` function. As an example, consider the problem of draw a triangle in a window. For this we need to make a function that can draw a triangle not once, but any time WinForms determines it necessary to draw and redraw the triangle. This is done in Listing 13.4.

· call-back function

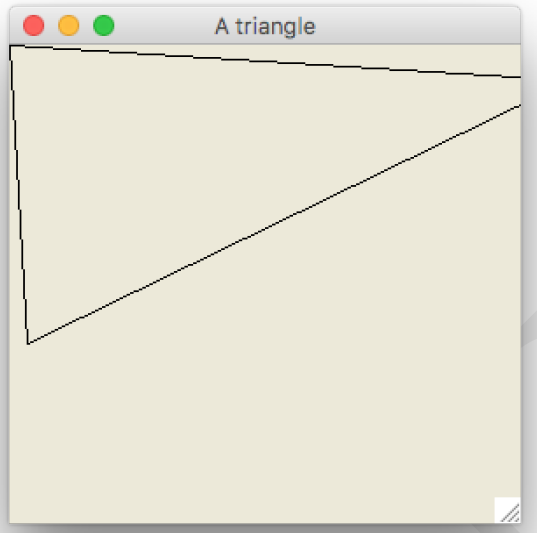

Figure 13.4: Result of running listing Listing 13.4.

Listing 13.4, winforms/triangle.fsx:
Create the window and changing its properties.

```
// Choose some points and a color
let Points =
  [|System.Drawing.Point (0,0);
   System.Drawing.Point (10,170);
   System.Drawing.Point (320,20);
   System.Drawing.Point (0,0)|]
let penColor = System.Drawing.Color.Black
// Create window and setup drawing function
let pen = new System.Drawing.Pen (penColor)
let win = new System.Windows.Forms.Form ()
win.Text <- "A triangle"
win.Paint.Add (fun e -> e.Graphics.DrawLines (pen, Points))
// Start the event-loop. Use "win.Show()" on Microsoft Windows instead.
System.Windows.Forms.Application.Run win
```

A walk-through of the code is as follows: First we create an array of points and a pen color, then we create a pen and a window. The method for drawing the triangle is added as an anonymous function using the created window's `Paint.Add` method. This function is to be called as a response to a paint event, and takes a `PaintEventArgs` object, which includes the System.Drawing.Graphics object. Since this object will be related to a specific device, when `Paint` is called then we may call the `DrawLine` function to sequentially draw lines between our array of points. Finally, we hand the form to the event-loop, which as one of the earliest events will open the window and call the `Paint` function we have associated with the form.

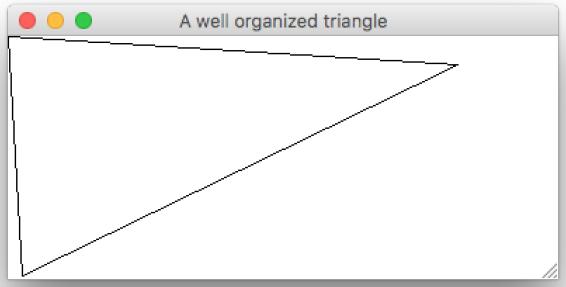

Figure 13.5: Result of running listing Listing 13.5.

Listing 13.5, winforms/triangleOrganized.fsx:
Create the window and changing its properties.

```
open System.Windows.Forms
open System.Drawing

type coordinates = (float * float) list
type pen = Color * float

/// Create a form and add a paint function
let createForm backgroundColor (width, height) title draw =
  let win = new Form ()
  win.Text <- title
  win.BackColor <- backgroundColor
  win.Size <- Size (width, height)
  win.Paint.Add draw
  win

/// Draw a polygon with a specific color
let drawPoints (coords : coordinates) (pen : pen) (e : PaintEventArgs) =
  let pairToPoint (x : float, y : float) =
    Point (int (round x), int (round y))
  let color, width = pen
  let Pen = new Pen (color, single width)
  let Points = Array.map pairToPoint (List.toArray coords)
  e.Graphics.DrawLines (Pen, Points)

// Setup drawing details
let title = "A well organized triangle"
let backgroundColor = Color.White
let size = (400, 200)
let coords = [(0.0, 0.0); (10.0, 170.0); (320.0, 20.0); (0.0, 0.0)]
let pen = (Color.Black, 1.0)

// Create form and start the event-loop.
let win = createForm backgroundColor size title (drawPoints coords pen)
Application.Run win // win.Show()
```

12

Listing 13.6, winforms/transformWindows.fsx:
Create the window and changing its properties: top part.

```
open System.Windows.Forms
open System.Drawing

type coordinates = (float * float) list
type pen = Color * float
type polygon = coordinates * pen

/// Create a form and add a paint function
let createForm backgroundColor (width, height) title draw =
  let win = new Form ()
  win.Text <- title
  win.BackColor <- backgroundColor
  win.Size <- Size (width, height)
  win.Paint.Add draw
  win

/// Draw a polygon with a specific color
let drawPoints (polygLst : polygon list) (e : PaintEventArgs) =
  let pairToPoint (x : float, y : float) =
    Point (int (round x), int (round y))

  for polyg in polygLst do
    let coords, (color, width) = polyg
    let pen = new Pen (color, single width)
    let Points = Array.map pairToPoint (List.toArray coords)
    e.Graphics.DrawLines (pen, Points)

/// Translate point array
let translatePoints (dx, dy) arr =
  let translatePoint (dx, dy) (x, y) =
    (x + dx, y + dy)
  List.map (translatePoint (dx, dy)) arr

/// Rotate point array
let rotatePoints theta arr =
  let rotatePoint theta (x, y) =
    (x * cos theta - y * sin theta, x * sin theta + y * cos theta)
  List.map (rotatePoint theta) arr
```

[1]Todo: **requires the introduction of type declarations.**
[2]Todo: **Remember to talk about pen width.**

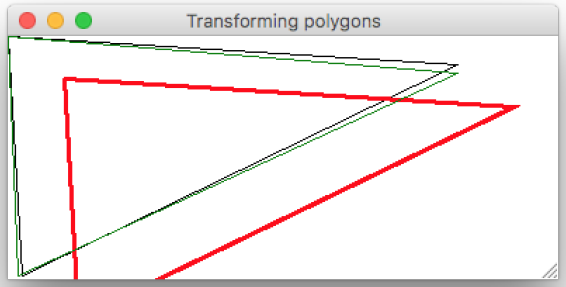

Figure 13.6: Result of running listing Listing 13.7.

Listing 13.7, winforms/transformWindows.fsx:
Create the window and changing its properties: bottom part.

```
// Setup drawing details
let title = "Transforming polygons"
let backgroundColor = Color.White
let size = (400, 200)
let points = [(0.0, 0.0); (10.0, 170.0); (320.0, 20.0); (0.0, 0.0)]
let polygLst =
  [(points, (Color.Black, 1.0));
   (translatePoints (40.0, 30.0) points, (Color.Red, 2.0));
   (rotatePoints (1.0 *System.Math.PI / 180.0) points, (Color.Green, 1.0))
   ]

// Create form and start the event-loop.
let win = createForm backgroundColor size title (drawPoints polygLst)
System.Windows.Forms.Application.Run win // win.Show()
```

Problem 13.1:

Given a triangle produce a Mandela drawing, where n rotated versions of the triangle is drawn around its center of mass.

Listing 13.8, winforms/rotationalSymmetry.fsx:
Create the window and changing its properties.

```
/// Calculate the mass center of a list of points
let centerOfPoints (points : (float * float) list) =
  let addToAccumulator acc elm = (fst acc + fst elm, snd acc + snd elm)
  let sum = List.fold addToAccumulator (0.0,  0.0) points
  (fst sum / (float points.Length), snd sum / (float points.Length))

/// Generate repeated rotated point-color pairs
let rec rotatedLst points color width src dest nth n =
  if n > 0 then
    let newPoints =
      points
      |> translatePoints (- fst src, - snd src)
      |> rotatePoints ((float n) * nth)
      |> translatePoints dest
    (newPoints, (color, width))
      :: (rotatedLst points color width src dest nth (n - 1))
  else
    []

// Setup drawing details
let title = "Rotational Symmetry"
let backgroundColor = Color.White
let size = (600, 600)
let points = [(0.0, 0.0); (10.0, 170.0); (320.0, 20.0); (0.0, 0.0)]
let src = centerOfPoints points
let dest = ((float (fst size)) / 2.0, (float (snd size)) / 2.0)
let n = 36;
let nth = (360.0 / (float n)) * (System.Math.PI / 180.0)
let orgPoints =
  points
  |> translatePoints (fst dest - fst src, snd dest - snd src)
let polygLst =
  rotatedLst points Color.Blue 1.0 src dest nth n
    @ [(orgPoints, (Color.Red, 3.0))]

// Create form and start the event-loop.
let win = createForm backgroundColor size title (drawPoints polygLst)
Application.Run win // win.Show()
```

[3]

13.2 Buttons and stuff

...

[3] Todo: **Add other things to draw: filled stuff, clearing, circles, text**

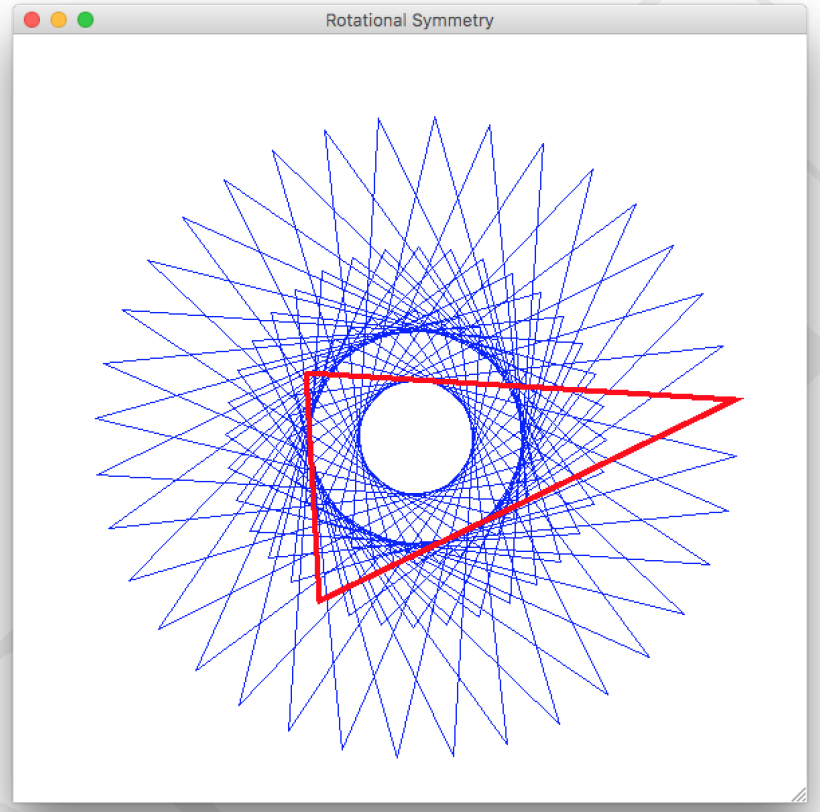

Figure 13.7: Result of running listing Listing 13.8.

Bibliography

[1] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[2] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.

[3] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[4] X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. `http://worldpowersystems.com/projects/codes/X3.4-1963/`.

[5] George Pólya. *How to solve it*. Princeton University Press, 1945.

Index