Learning to program with F#

Jon Sporring

November 20, 2016

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*) also includes graphical elements such as windows, icons, and sound, 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 programming*, meaning that at run-time, most time is spend on waiting for the user to perform an action, called and *event*, and each possible event has a predefined response to be performed.

· event driven programming
· event

An example of a graphical user interface is a web-browser, as shown in Figure 13.1. The program present information to the user in terms of text and images and has active areas that may be activated by clicking and which allows the user to go to other web-pages by type URL, to follow hyperlinks, and to generate new pages by entering search queries. Designing easy to use graphical user interfaces is a challenging task. This chapter will focus on examples of basic graphical elements and how to program these in WinForms.

13.1 Drawing primitives in Windows

The main workhorse of WinForms are the functions and classes defined in the namespaces: ***`System.Windows.Forms`*** and *`System.Drawing`*. These give access to the *Windows Graphics Device Interface* (*GDI+*), which allows you to create and manipulate graphics objects targeting several platforms such as screens and paper.

· `System.Windows.Forms`
· `System.Drawing`
· Windows Graphics Device Interface
· GDI+
· forms

To display a graphical user interface on the screen, the first thing to do is open a window, which acts as a reserved screen-space for our output. In WinForms windows are called *forms*. Code for opening a window is shown in Listing 13.1, and the result is shown in Figure 13.3.

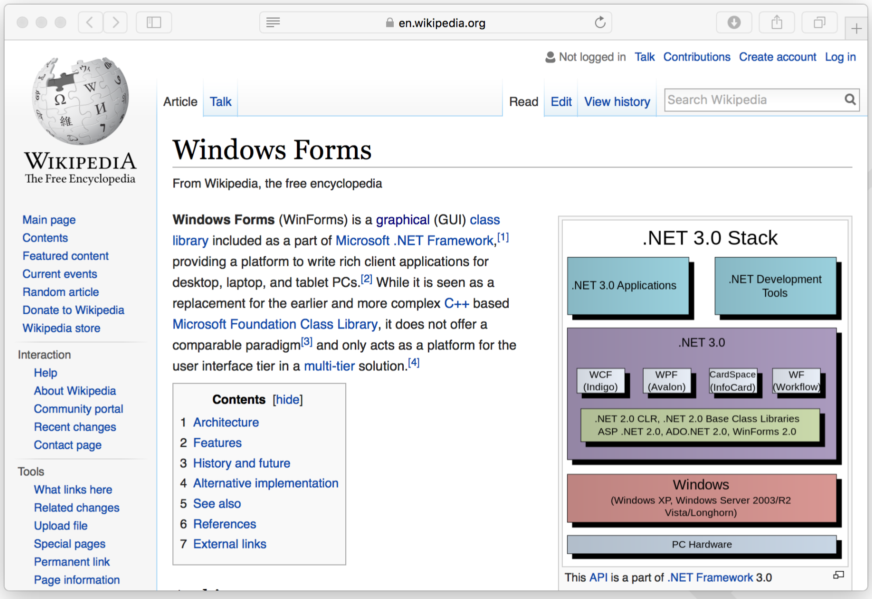


Figure 13.1: A web-browser is a graphical user interface for accessing a web-server and interacting with its services. Here the browser is showing the page `https://en.wikipedia.org/wiki/Windows_Forms` at time of writing.

Listing 13.1, winforms/openWindow.fsx:
Create the window and turn over control to the operating system.

```
// Create a window
let win = new System.Windows.Forms.Form ()
// Start the event-loop.
System.Windows.Forms.Application.Run win
```

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 WinForms' *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 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 13.2. · methods · properties

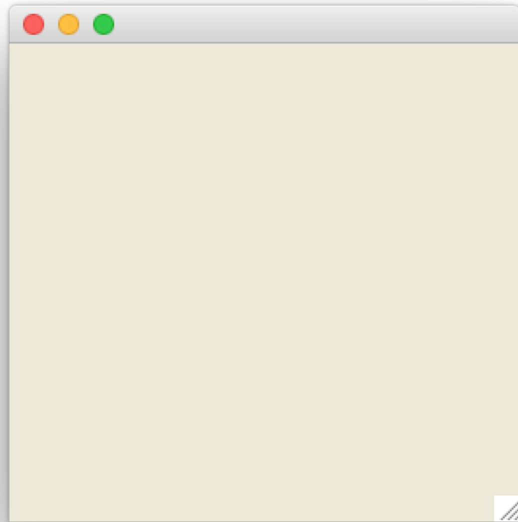


Figure 13.2: A window opened by Listing 13.1.

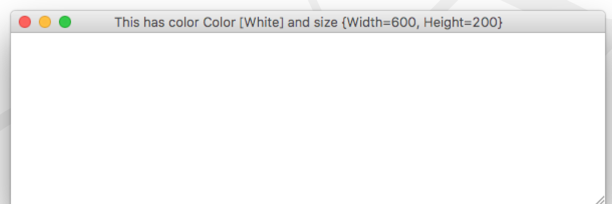


Figure 13.3: A window with user-specified size and background color, see Listing 13.2.

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.
System.Windows.Forms.Application.Run win
```

These properties have been programmed as *accessors* implying that they may used as mutable variables. · accessors

The `System.Drawing.Color` is a general structure for specifying colors as 4 channels: alpha, red, green, · `System.Drawing.Color`

Method/Property	Description
`A`	Get the value of the alpha channel of a color.
`B`	Get the value of the blue channel of a color.
`Black`	Get a predefined color with ARGB value of `0 xFF000000`.
`Blue`	Get a predefined color with ARGB value of `0 xFF0000FF`.
`FromArgb : int -> Color` `FromArgb : int*int*int*int -> Color`	Create a color structure..
`G`	Get the value of the green channel of a color.
`Green`	Get a predefined color with ARGB value of `0 xFF00FF00`.
`R`	Get the value of the red channel of a color.
`Red`	Get a predefined color with ARGB value of `0 xFFFF0000`.
`ToArgb : Color -> int`	Get the 32 bit integer representation of a color.
`White`	Get a predefined color with ARGB value of `0 xFFFFFFFF`.

Table 13.1: Some methods and properties of the `System.Drawing.Color` structure.

blue. Some methods and properties for the Color structure is shown in Table 13.1. Each channel is an 8 bit unsigned integer, but often referred as the 32 bit unsigned integer by concatenating the channels. 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

Constructor	Description
`Point(int, int)` `Point(Size)`	An ordered pair of integers specifying x- and y-coordinates in the plane.
`Size(int, int)` `Size(Point)`	An ordered pair of integers specifying height and width in the plane.
`Rectangle(int, int, int, int)` `Rectangle(Point, Size)`	A structure specifying a rectangular region by its upper left corner and its size.

Table 13.2: Basic geometrical structures in WinForms.

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 uses a number of types for specifying various objects some of which are shown in Table 13.2. [1][2]

WinForms supports drawing of geometric primitives such as lines, rectangles, and ellipses, but not points. To draw an element similar to a point, you must draw a small rectangle or ellipse. The location and shape of geometrical primitives are specified in a coordinate system, and WinForms operates with 2 coordinate systems: *screen coordinates* and *client coordinates*. Screen coordinate (x, y) have their origin in the top-left corner of the screen, and x increases to the right, while y increases down. Client coordinates refers to the drawable area of a form or a control, i.e., for a window this will be the area without the window borders, scroll and title bars. A control is a graphical object such as a clickable button, which will be discussed later. Conversion between client and screen coordinates is done with *`System.Drawing.PointToClient`* and *`System.Drawing.PointToScreen`*. To draw geometric primitives, we must also specify the pen using for line like primitives and the brush for filled regions.

· screen coordinates
· client coordinates
· `System.Drawing.PointToClient`
· `System.Drawing.PointToScreen`

[1]Todo: **Note on difference between Size and ClientSize.**
[2]Todo: **Do something about the vertical alignment of minpage.**

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 any time. This is known as a *call-back function*, and it is added to an existing form using the ***`Paint.Add`*** method. Due to the event-driven nature of WinForms, functions for drawing graphics primitives are only available when responding to an event, e.g., ***`System.Drawing.Graphics.DrawLines`*** draws a line in a window, and it is only possible to call this function, as part of an event handling.

· call-back function
· `Paint.Add`
· `System.Drawing.Graphics.DrawLines`

As an example, consider the problem of drawing a triangle in a window. For this we need to make a function that can draw a triangle not once, but any time. An example of such a program is shown in Listing 13.4.

Listing 13.4, winforms/triangle.fsx:
Adding line graphics to a window.

```
// 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.
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 `System.Windows.Forms.PaintEventArgs` object, which includes the System.Drawing.Graphics object. Since this object will be related to a specific device, when the window's `Paint` method is called, then we may call the `System.Drawing.Graphics.DrawLines` 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.

Considering the program in Listing 13.4, we may identify a part that concerns the specification of the triangle, or more generally the graphical model, some parts which handling events, and some which concerns system specific details on initialization of the interface. For future maintenance, it is often a good idea to **separate the model from the implementation**. E.g., it may be that at some point, you decide that you would rather use a different library than WinForms. In this case, the general graphical model will be the same but the specific details on initialization and event handling will be different. While it is not easy to completely separate the general from the specific, it is often a good idea to strive some degree of separation. E.g., in Listing 13.4, the program has been redesigned to make use of an initialization function and a paint function.

Advice

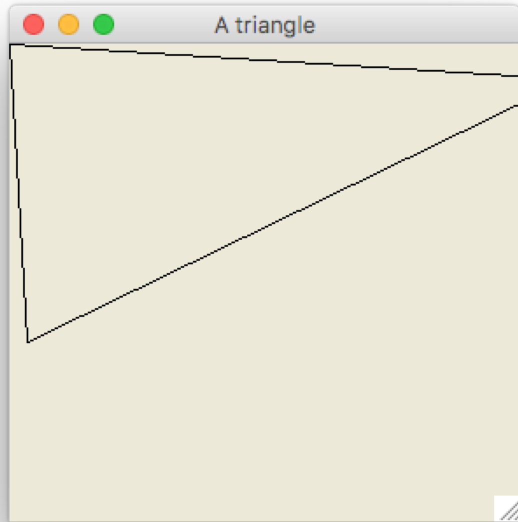


Figure 13.4: Drawing a triangle using Listing 13.4.

Listing 13.5, winforms/triangleOrganized.fsx:
Improved organization of code for drawing a triangle. Compare with Listing 13.4.

```
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.ClientSize <- 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
```

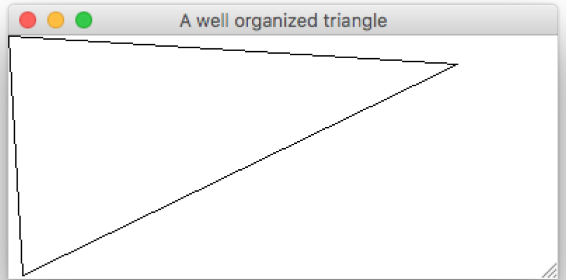


Figure 13.5: Better organization of the code for drawing a triangle, see Listing 13.5.

While this program is longer, to this author there is a much better separation of *what* is to be displayed from the *how* it is to be done, since the *how* is not contained in the functions `createForm` and `drawPoints`. The user-defined types `coordinates` and `pen` further emphasizes the semantic content of the data structures in use and makes use of Fsharp's type checker to reduce run-time errors. [3][4]

Organizing code into functions that operate on data structures as Listing 13.5 is the first step in *Structured programming* to be discussed in Part IV. Consider the case, where are to draw two new triangles, that are a translation and a rotations of the original. A simple extension of Listing 13.5 is to make a list of lists of Points and to extend `drawPoints` with a loop for drawing all shapes in the list. This structure should include the ability to draw shapes in different styles, hence we arrive at a structure of type `(coordinates * pen) list`. Furthermore, since the problem is to draw the same shape at different locations and orientations, instead of calculating the new coordinates by hand, it is useful to add functions to translate and rotation a given shape. Thus we arrive at the program shown in Listing 13.7, and which results in the output shown in Figure 13.6.

· Structured programming

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

Listing 13.6, winforms/transformWindows.fsx:
Reusable code for drawing in windows.

```
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.ClientSize <- 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 a point
let translatePoint (dx, dy) (x, y) =
  (x + dx, y + dy)

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

/// Rotate a point
let rotatePoint theta (x, y) =
  (x * cos theta - y * sin theta, x * sin theta + y * cos theta)

/// Rotate point array
let rotatePoints theta arr =
  List.map (rotatePoint theta) arr
```

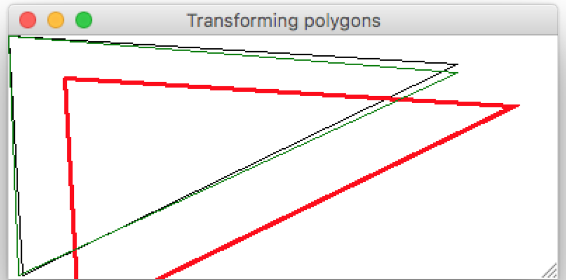


Figure 13.6: Transformed versions of the same triangle resulting from running the code in Listing 13.7.

Listing 13.7, winforms/transformWindows.fsx:
Code for drawing triangles using the reusable part shown in Listing 13.7.

```
// 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)
Application.Run win
```

We now have a basis for solving the following problem:

Problem 13.1:

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

Reusing the top part of Listing 13.7 and replacing the bottom part with the code shown in Listing 13.8, we arrive a the solution illustrated in Figure 13.7.

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
```

[5]

The `System.Drawing.Graphics` class contains many other algorithms for drawing graphics primitives, some of which are listing in Table 13.3 [6]

13.2 Programming intermezzo

Reusing the top part of Listing 13.7 we are now able to tackle complicated problems such as space-filling curves. A curve in 2 dimensions has a length but no width, and we can only visualize it by

[5] Todo: **Remember to add something on piping.**
[6] Todo: **Give examples of more methods**

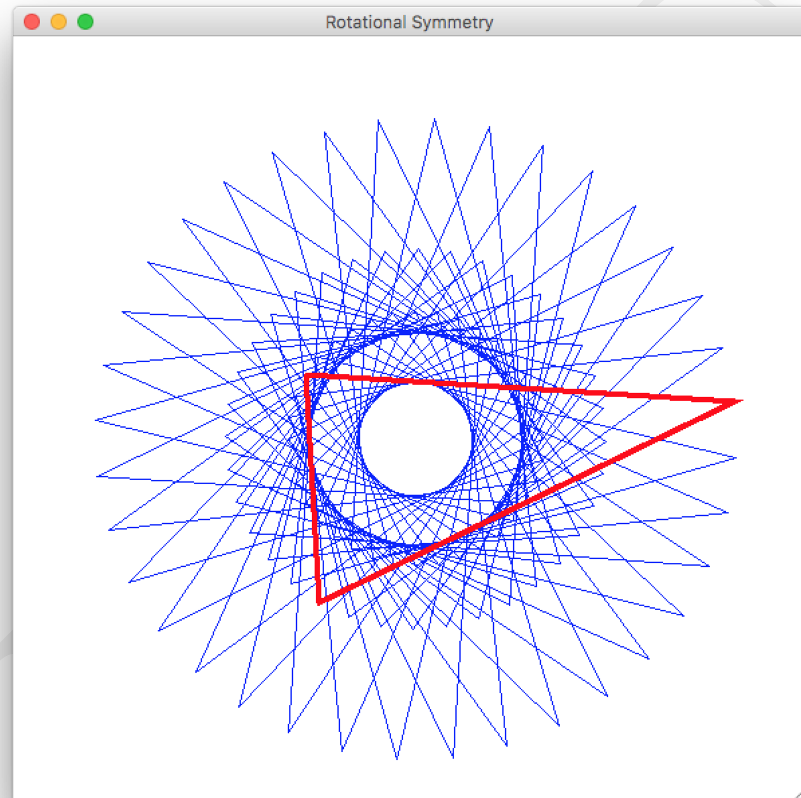


Figure 13.7: A symmetric figure resulting from Listing 13.8.

Function	Description
`DrawArc : Pen * Rectangle * Single * Single`	Draws an arc representing a portion of an ellipse specified by a Rectangle structure.
`DrawBezier : Pen * Point * Point * Point * Point`	Draws a Bézier spline defined by four Point structures.
`DrawCurve : Pen * Point[]`	Draws a cardinal spline through a specified array of Point structures.
`DrawEllipse : Pen * Rectangle`	Draws an ellipse specified by a bounding Rectangle structure.
`DrawImage : Image * Point[]`	Draws the specified Image at the specified location and with the specified shape and size.
`DrawLine : Pen * Point * Point`	Draws a series of line segments that connect an array of Point structures.
`DrawLines : Pen * Point[]`	Draws a series of line segments that connect an array of Point structures.
`DrawPie : Pen * Rectangle * Single * Single`	Draws a pie shape defined by an ellipse specified by a Rectangle structure and two radial lines.
`DrawPolygon : Pen * Point[]`	Draws a polygon defined by an array of Point structures.
`DrawRectangles : Pen * Rectangle[]`	Draws a series of rectangles specified by Rectangle structures.
`DrawString : String * Font * Brush * PointF`	Draws the specified text string at the specified location with the specified Brush and Font objects.
`FillClosedCurve : Brush * Point[]`	Fills the interior of a closed cardinal spline curve defined by an array of Point structures.
`FillEllipse : Brush * Rectangle`	Fills the interior of an ellipse defined by a bounding rectangle specified by a Rectangle structure.
`FillPie : Brush * Rectangle * Single * Single`	Fills the interior of a pie section defined by an ellipse specified by a RectangleF structure and two radial lines.
`FillPolygon : Brush * Point[]`	Fills the interior of a polygon defined by an array of points specified by Point structures.
`FillRectangle : Brush * Rectangle`	Fills the interior of a rectangle specified by a Rectangle structure.

Table 13.3: Some methods of the `System.Drawing.Graphics` class.

giving it a width. It is thus came as a surprise to many when Giuseppe Peano in 1890 demonstrated that there exists curves, which fill every point in a square. The method he used to achieve this was recursion.

Problem 13.2:

Consider a curve consisting of piecewise straight lines all with the same length but with varying angles 0°, 90°, 180°, or 270° w.r.t. the horisontal axis. To draw this curve we need 3 basic operations: Move forward (F), turn right ($+$), and turn left ($-$). The turning is w.r.t. the present direction. A Hilbert Curve is a space-filling curve, which be expressed recursively as:

$$A \to -BF + AFA + FB- \tag{13.1}$$

$$B \to +AF - BFB - FA+ \tag{13.2}$$

starting with A. For practical illustrations, we typically only draw space filling curves to a specified depth of recursion, which is called the order of the curve. Hence, to draw a first order curve, we don't recurse at all, i.e., ignore all occurrences of the symbols A and B on the right-hand-side of (13.1), and get

$$A \to -F + F + F - .$$

For the second order curve, we recurse once, i.e.,

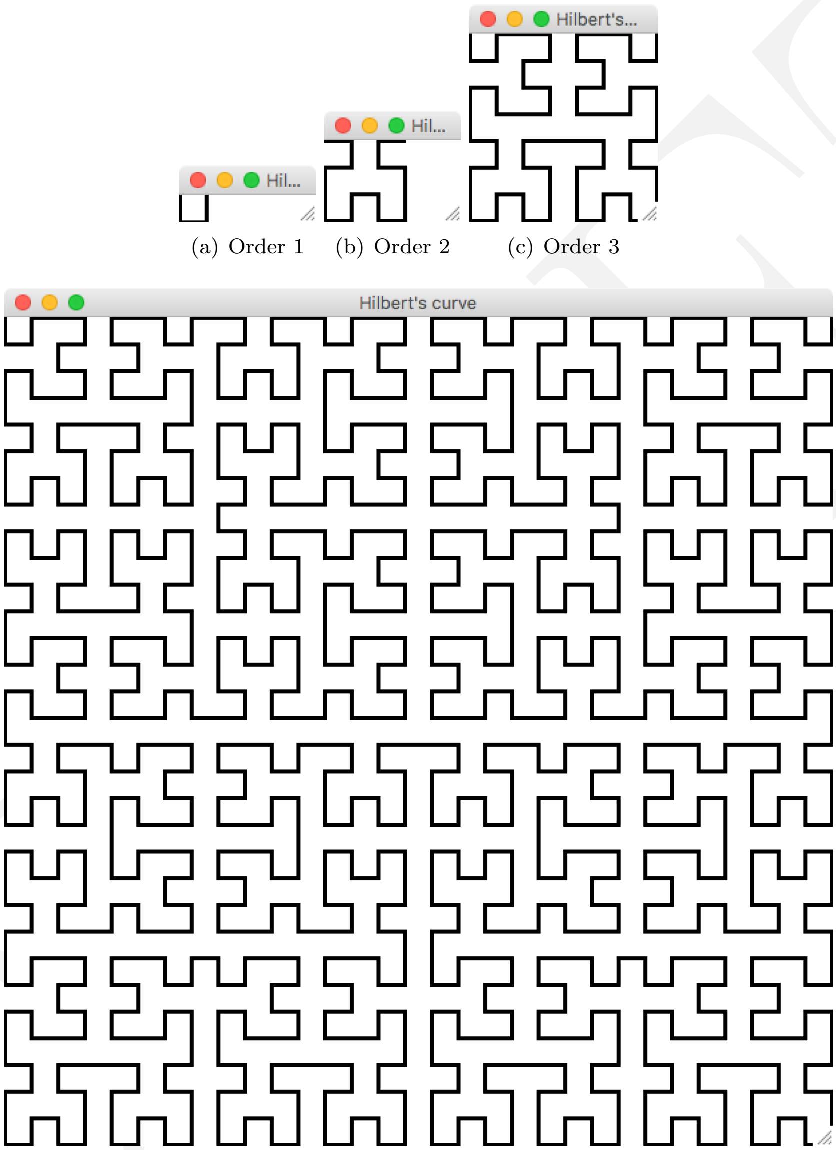
$$\begin{aligned}
A \to& - BF + AFA + FB- \\
&\to - (+AF - BFB - FA+)F \\
&\qquad + (-BF + AFA + FB-)F(-BF + AFA + FB-) \\
&\qquad\quad + F(+AF - BFB - FA+)- \\
&\to AF - BFB - FA + FBF + AFA + FB - F - BF + AFA + FBF + AF - BFB - FA \\
&\to F - F - F + FF + F + F - F - F + F + FF + F - F - F.
\end{aligned}$$

Make a program, that given an order produces an image of the Hilbert curve.

Our strategy will be to draw the curve sequentially from the origin to the end point. For this we will make a data structure `type curve = float * float * coordinates)` length and the orientation of the next forward move and a partial list straight line pieces. The initial point would thus be `1.0, 0.0,` [0.0, 0.0)]). We will also make 2 mutually recursive functions `ruleA : float curve -> curve` and `ruleB : float curve -> curve`, which takes n order and our data structure as input and for positive orders adds its rule to the end of the curve. We also need the three basic operations. To move forward, we will use the `rotatePoint` and `translatePoint` functions from Listing 13.7, that is, we may take line piece of specified length starting in the origin, rotate it according to the present orientation of the curve, and translate it to the present end-point of the curve. The two turn right and left only need to add or subtract 90° to the present orientation. Thus, reusing the top part of Listing 13.7 we arrive at the solutions shown in Listing 13.9. In Figure 13.8 are shown the result of using the program to draw Hilbert curves of orders 1, 2, 3, and 5.

Listing 13.9, winforms/hilbert.fsx:
Create the window and changing its properties.

```
type curve = float * float * coordinates

/// Turn 90 degrees left
let left (l, dir, c) : curve = (l, dir + 3.141592/2.0, c)

/// Turn 90 degrees right
let right (l, dir, c) : curve = (l, dir - 3.141592/2.0, c)

/// Add a line to the curve of present direction
let forward (l, dir, c) : curve =
  let nextPoint = rotatePoint dir (l, 0.0)
  (l, dir, c @ [translatePoint c.[c.Length-1] nextPoint])

/// Find the maximum value of each coordinate element in a list
let maximum (c : coordinates) =
  let maxPoint p1 p2 =
    (max (fst p1) (fst p2), max (snd p1) (snd p2))
  List.fold maxPoint (-infinity, -infinity) c

/// Hilbert recursion production rules
let rec ruleA n C : curve =
  if n > 0 then
    (C |> left |> ruleB (n-1) |> forward |> right |> ruleA (n-1) |>
   forward |> ruleA (n-1) |> right |> forward |> ruleB (n-1) |> left)
  else
    C
and ruleB n C : curve =
  if n > 0 then
    (C |> right |> ruleA (n-1) |> forward |> left |> ruleB (n-1) |>
   forward |> ruleB (n-1) |> left |> forward |> ruleA (n-1) |> right)
  else
    C

// Calculate curve
let order = 5
let l = 20.0
let (_, dir, c) = ruleA order (l, 0.0, [(0.0, 0.0)])

// Setup drawing details
let title = "Hilbert's curve"
let backgroundColor = Color.White
let cMax = maximum c
let size = (int (fst cMax)+1, int (snd cMax)+1)
let polygLst = [(c, (Color.Black, 3.0))]

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



(a) Order 1 (b) Order 2 (c) Order 3

(d) Order 5

Figure 13.8: Hilbert curves of order 1, 2, 3, and 5 by code in Listing 13.9.

13.3 Events, Controls, and Panels

In the previous section, we have looked at how to draw graphics using the `Paint` method of an existing form object. Forms have many other event listeners, that we may use to interact with the user. Listing 13.11 demonstrates event handlers for moving and resizing a window, for clicking in a window, and for typing on the keyboard.

**Listing 13.10, winforms/windowEvents.fsx:
Catching window, mouse, and keyboard events.**

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

// create a form
let win = new Form ()

// Customize the window
win.Text <- "Window events"
win.BackColor <- Color.White
win.ClientSize <- Size (200, 200)
// Window event
let windowMove (e : EventArgs) =
  printfn "Move: %A" win.Location
win.Move.Add windowMove
let windowResize (e : EventArgs) =
  printfn "Resize: %A" win.DisplayRectangle
win.Resize.Add windowResize
// Mouse event
let mutable record = false;
let mouseMove (e : MouseEventArgs) =
  if record then printfn "MouseMove: %A" e.Location
win.MouseMove.Add mouseMove
let mouseDown (e : MouseEventArgs) =
  printfn "MouseDown: %A" e.Location; (record <- true)
win.MouseDown.Add mouseDown
let mouseUp (e : MouseEventArgs) =
  printfn "MouseUp: %A" e.Location; (record <- false)
win.MouseUp.Add mouseUp
let mouseClick (e : MouseEventArgs) =
  printfn "MouseClick: %A" e.Location
win.MouseClick.Add mouseClick
// Keyboard event
win.KeyPreview <- true
let keyPress (e : KeyPressEventArgs) =
  printfn "KeyPress: %A" (e.KeyChar.ToString ())
win.KeyPress.Add keyPress

// Start the event-loop.
Application.Run win
```

Listing 13.11: Output from an interaction with the program in Listing 13.11.

```
Move: {X=22,Y=22}
Move: {X=22,Y=22}
Move: {X=83,Y=161}
Resize: {X=0,Y=0,Width=275,Height=211}
MouseDown: {X=179,Y=123}
MouseClick: {X=179,Y=123}
MouseUp: {X=179,Y=123}
MouseDown: {X=179,Y=123}
MouseMove: {X=178,Y=123}
MouseMove: {X=174,Y=123}
MouseMove: {X=169,Y=123}
MouseMove: {X=164,Y=123}
MouseMove: {X=159,Y=123}
MouseMove: {X=154,Y=123}
MouseMove: {X=150,Y=123}
MouseMove: {X=146,Y=123}
MouseMove: {X=137,Y=123}
MouseMove: {X=130,Y=123}
MouseMove: {X=124,Y=122}
MouseMove: {X=119,Y=120}
MouseMove: {X=113,Y=118}
MouseMove: {X=109,Y=115}
MouseMove: {X=105,Y=112}
MouseMove: {X=101,Y=110}
MouseMove: {X=100,Y=109}
MouseMove: {X=97,Y=108}
MouseMove: {X=95,Y=107}
MouseClick: {X=95,Y=107}
MouseUp: {X=95,Y=107}
KeyPress: "f"
KeyPress: "s"
KeyPress: "h"
KeyPress: "a"
KeyPress: "r"
KeyPress: "p"
```

In Listing 13.11 is shown the output from an interaction with the program which is the result of the following actions: moving the window, resizing the window, clicking the left mouse key, pressing and holding the down the left mouse key while moving the mouse, and releasing the left mouse key, and type “fsharp”. As demonstrated, some actions like moving the mouse results in a lot of events, and some like the initial window moves results are surprising. Thus, event drivent programming should take care to interpret the events robustly and carefully.

In WinForms buttons, menus and other interactive elements are called *Controls*. A form is a type of control, and thus, programming controls are very similar to programming windows. In Listing 13.12 is shown a small program that displays a button in a window, and when the button is pressed, then a dialogue is opened in another window. Figure 13.9 shows an example dialogue.

· Controls

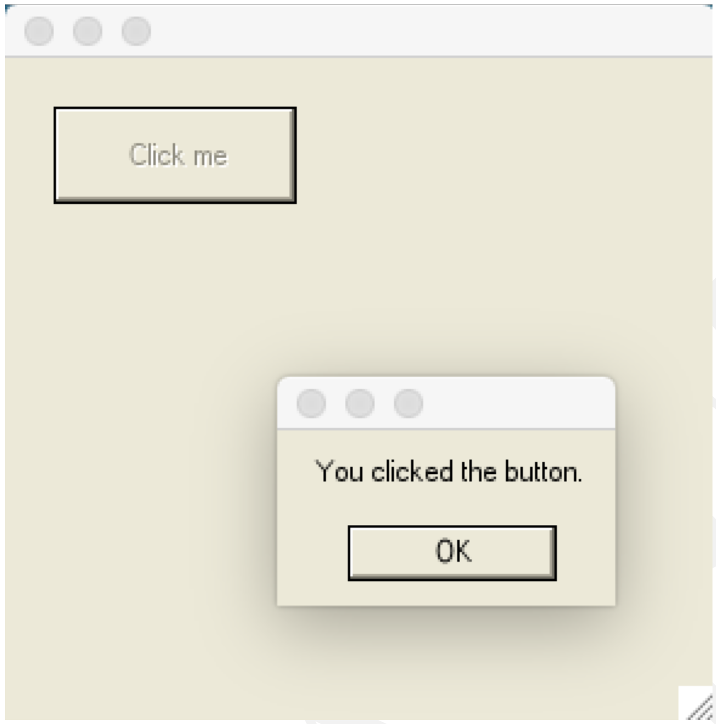


Figure 13.9: A button is pressed and the event handler calls the `MessageBox.Show` dialogue window by the code in Listing 13.12.

Listing 13.12, winforms/buttonControl.fsx:
Create the button and an event.

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

/// A button event
let buttonClicked (e : EventArgs) =
  MessageBox.Show "You clicked the button." |> ignore

// Create a button
let button = new Button ()
button.Size <- new Size (100, 40)
button.Location <- new Point (20, 20)
button.Text <- "Click me"
button.Click.Add buttonClicked

// Create a window and add button
let win = new Form ()
win.Controls.Add button

// Start the event-loop.
Application.Run win
```

As the listing demonstrates, the button is created using the *`System.Windows.Forms.Button`* constructor, and it is added to the form's control list. It is possible to have many controls on each form, but

· `System.Windows.Forms.Button`

at times it is useful to organize the controls in groups. Such groups are called *Panels* in WinForms, and an example of creating a ***System.Windows.Forms.Panel*** that includes a ***System.Windows.Forms.TextBox*** and ***System.Windows.Forms.Label*** for getting user input is shown in Listing 13.13 and Figure 13.10.

· Panels
· System.Windows.Forms.Panel
· System.Windows.Forms.TextBox
· System.Windows.Forms.Label

Listing 13.13, winforms/panel.fsx:
Create a panel, label, text input controls.

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

// Initialize a form containing a panel, textbox, and a label
let win = new Form ()
let panel = new Panel ()
let textBox = new TextBox ()
let label = new Label ()

// Customize the window
win.Text <- "Window with a panel"
win.ClientSize <- new Size (400, 300)

// Customize the Panel control
panel.Location <- new Point (56,72)
panel.Size <- new Size (264, 152)
panel.BorderStyle <- BorderStyle.Fixed3D

// Customize the Label and TextBox controls
label.Location <- new Point (16,16)
label.Text <- "label"
label.Size <- new Size (104, 16)
textBox.Location <- new Point (16,32)
textBox.Text <- "Initial text"
textBox.Size <- new Size (152, 20)

// Add panel to window and label and textBox to panel
win.Controls.Add panel
panel.Controls.Add label
panel.Controls.Add textBox

// Start the event-loop
Application.Run win
```

The label and textbox are children of the panel, and the main advantage of using panels is that the coordinates of the children are relative to the top left corner of the panel. I.e., moving the panel will move the label and the textbox at the same time.

Several types of panels exists in WinForms. A very flexible panel is the ***System.Windows.Forms.FlowLayoutPanel***, which arranges its objects according to the space available. This is useful for graphical user interfaces targeting varying device sizes, such as a computer monitor and a tablet, and it also allows the program to gracefully adapt, when the user changes window size. A demonstration of `System.Windows.Forms.FlowLayoutPanel` together with `System.Windows.Forms.CheckBox` and `System.Windows.Forms.RadioButton` is given in Listing 13.15 and in Figure 13.11. The program illustrates how the button elements flow under four possible `System.Windows.FlowDirections`, and how `System.Windows.WrapContents` influences, what happens to content that flows outside the panels region.

· System.Windows.Forms.FlowLayoutPanel

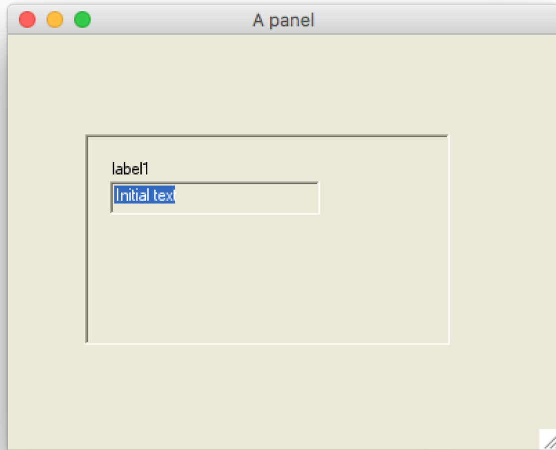


Figure 13.10: A panel including a label and a text input field, see Listing 13.13.

Listing 13.14, winforms/flowLayoutPanel.fsx:
Create a FlowLayoutPanel, with checkbox and radiobuttons.

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

let flowLayoutPanel = new FlowLayoutPanel ()
let buttonLst =
  [(new Button (), "Button0");
   (new Button (), "Button1");
   (new Button (), "Button2");
   (new Button (), "Button3")]
let panel = new Panel ()
let wrapContentsCheckBox = new CheckBox ()
let initiallyWrapped = true
let radioButtonLst =
  [(new RadioButton (), (3, 34), "TopDown", FlowDirection.TopDown);
   (new RadioButton (), (3, 62), "BottomUp", FlowDirection.BottomUp);
   (new RadioButton (), (111, 34), "LeftToRight", FlowDirection.
    LeftToRight);
   (new RadioButton (), (111, 62), "RightToLeft", FlowDirection.
    RightToLeft)]

// customize buttons
for (btn, txt) in buttonLst do
  btn.Text <- txt

// customize wrapContentsCheckBox
wrapContentsCheckBox.Location <- new Point (3, 3)
wrapContentsCheckBox.Text <- "Wrap Contents"
wrapContentsCheckBox.Checked <- initiallyWrapped
wrapContentsCheckBox.CheckedChanged.Add (fun _ -> flowLayoutPanel.
    WrapContents <- wrapContentsCheckBox.Checked)

// customize radio buttons
for (btn, loc, txt, dir) in radioButtonLst do
  btn.Location <- new Point (fst loc, snd loc)
  btn.Text <- txt
  btn.Checked <- flowLayoutPanel.FlowDirection = dir
  btn.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection <- dir)
```

Listing 13.15, winforms/flowLayoutPanel.fsx:
Create a FlowLayoutPanel, with checkbox and radiobuttons.

```
// customize flowLayoutPanel
for (btn, txt) in buttonLst do
  flowLayoutPanel.Controls.Add btn
flowLayoutPanel.Location <- new Point (47, 55)
flowLayoutPanel.BorderStyle <- BorderStyle.Fixed3D
flowLayoutPanel.WrapContents <- initiallyWrapped

// customize panel
panel.Controls.Add (wrapContentsCheckBox)
for (btn, loc, txt, dir) in radioButtonLst do
  panel.Controls.Add (btn)
panel.Location <- new Point (47, 190)
panel.BorderStyle <- BorderStyle.Fixed3D

// Create a window, add controlse, and start event-loop
let win = new Form ()
win.ClientSize <- new Size (302, 356)
win.Controls.Add flowLayoutPanel
win.Controls.Add panel
win.Text <- "A Flowlayout Example"
Application.Run win
```

A walkthrough of the program is as follows. The goal is to make 2 areas, one giving the user control over display parameters, and another displaying the result of the user's choices. These are `FlowLayoutPanel` and `Panel`. In the `FloatLayoutPanel` there are four `Buttons`, to be displayed in a region, that is not tall enough for the buttons to be shown in vertical sequence and not wide enough to be shown in horizontal sequence. Thus the `FlowDirection` rules come into play, i.e., the buttons are added in sequence as they are named, and the default `FlowDirection.LeftToRight` arranges places the `buttonLst.[0]` in the top left corner, and `buttonLst.[1]` to its right. Other flow directions does it differently, and the reader is encourage to experiment with the program.

The program in Listing 13.15 has not completely separated the semantic blocks of the interface and relies on explicitly setting of coordinates of controls. This can be avoided by using nested panels. E.g., in Listing 13.17, the program has been rewritten as a nested set of `FloatLayoutPanel` in three groups: The button panel, the checkbox, and the radio button panel. Adding a `Resize` event handler for the window to resize the outermost panel according to the outer window, allows for the three groups to change position relative to each other, which results in three different views all shown in Figure 13.12.

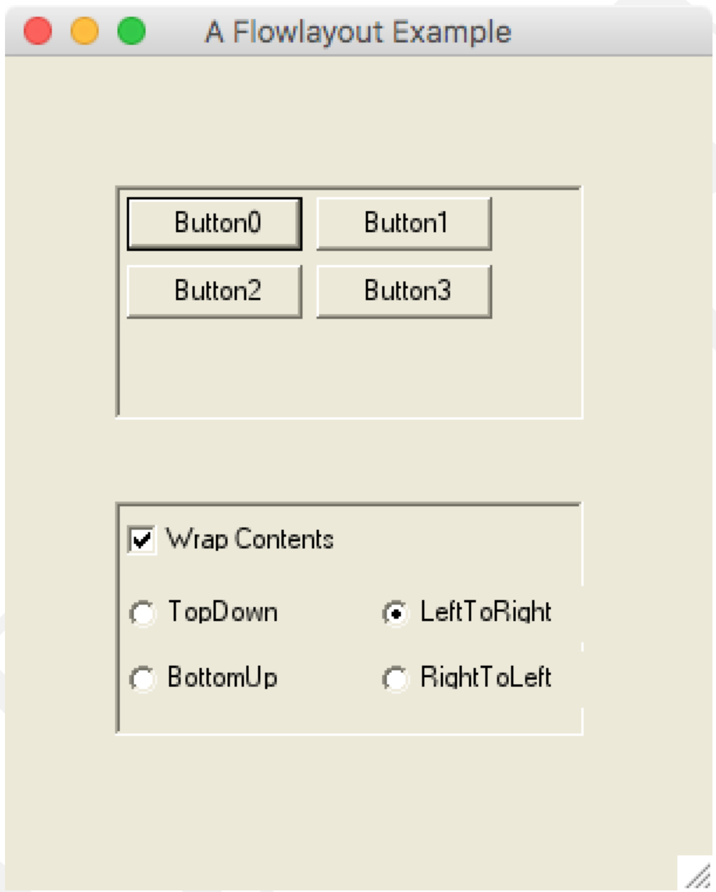


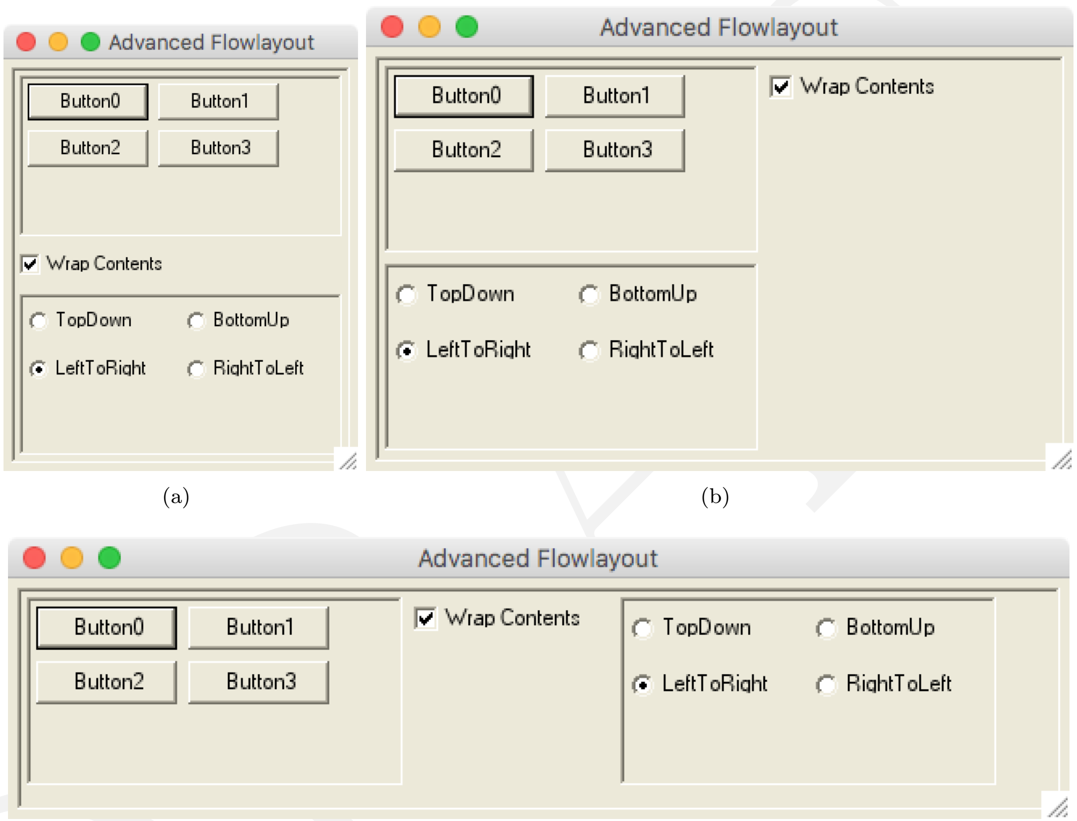
Figure 13.11: Demonstration of the `FlowLayoutPanel` panel, `CheckBox`, and `RadioButton` controls, see Listing 13.15.

Listing 13.16, winforms/flowLayoutPanelAdvanced.fsx: Create nested FlowLayoutPanel.

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

let win = new Form ()
let mainPanel = new FlowLayoutPanel ()
let mainPanelBorder = 5
let flowLayoutPanel = new FlowLayoutPanel ()
let buttonLst =
  [(new Button (), "Button0");
   (new Button (), "Button1");
   (new Button (), "Button2");
   (new Button (), "Button3")]
let wrapContentsCheckBox = new CheckBox ()
let panel = new FlowLayoutPanel ()
let initiallyWrapped = true
let radioButtonLst =
  [(new RadioButton (), "TopDown", FlowDirection.TopDown);
   (new RadioButton (), "BottomUp", FlowDirection.BottomUp);
   (new RadioButton (), "LeftToRight", FlowDirection.LeftToRight);
   (new RadioButton (), "RightToLeft", FlowDirection.RightToLeft)]

// customize buttons
for (btn, txt) in buttonLst do
  btn.Text <- txt

// customize radio buttons
for (btn, txt, dir) in radioButtonLst do
  btn.Text <- txt
  btn.Checked <- flowLayoutPanel.FlowDirection = dir
  btn.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection <- dir)

// customize flowLayoutPanel
for (btn, txt) in buttonLst do
  flowLayoutPanel.Controls.Add btn
flowLayoutPanel.BorderStyle <- BorderStyle.Fixed3D
flowLayoutPanel.WrapContents <- initiallyWrapped

// customize wrapContentsCheckBox
wrapContentsCheckBox.Text <- "Wrap Contents"
wrapContentsCheckBox.Checked <- initiallyWrapped
wrapContentsCheckBox.CheckedChanged.Add (fun _ -> flowLayoutPanel.
    WrapContents <- wrapContentsCheckBox.Checked)
```

Listing 13.17, winforms/flowLayoutPanelAdvanced.fsx: Create nested FlowLayoutPanel, see Listing 13.17.

```
// customize panel
// changing border style changes ClientSize
panel.BorderStyle <- BorderStyle.Fixed3D
let width = panel.ClientSize.Width / 2 - panel.Margin.Left - panel.Margin.
    Right
for (btn, txt, dir) in radioButtonLst do
  btn.Width <- width
  panel.Controls.Add (btn)

mainPanel.Location <- new Point (mainPanelBorder, mainPanelBorder)
mainPanel.BorderStyle <- BorderStyle.Fixed3D
mainPanel.Controls.Add flowLayoutPanel
mainPanel.Controls.Add wrapContentsCheckBox
mainPanel.Controls.Add panel

// Create a window, add controlse, and start event-loop
win.ClientSize <- new Size (220, 256)
let windowResize _ =
  let size = win.DisplayRectangle.Size
  mainPanel.Size <- new Size (size.Width - 2 * mainPanelBorder, size.
    Height - 2 * mainPanelBorder)
windowResize ()
win.Resize.Add windowResize
win.Controls.Add mainPanel
win.Text <- "Advanced Flowlayout"
Application.Run win
```



(a)

(b)

(c)

Figure 13.12: Nested `FlowLayoutPanel`, see Listing 13.17, allows for dynamic arrangement of content. Content flows, when the window is resized.