Chapter 1

Graphical User Interfaces

A *graphical user interface* (*GUI*) uses graphical elements such as windows, icons, and sound to communicate with the user, and a typical way to activate these elements is through a pointing device such as the mouse or by touch. Some of these elements may themselves be textual, and thus most operating systems offer access to a *command-line interface* (*CLI*) in a window alongside other interface types.

An example of a graphical user interface is a web-browser, shown in Figure 1.1. The

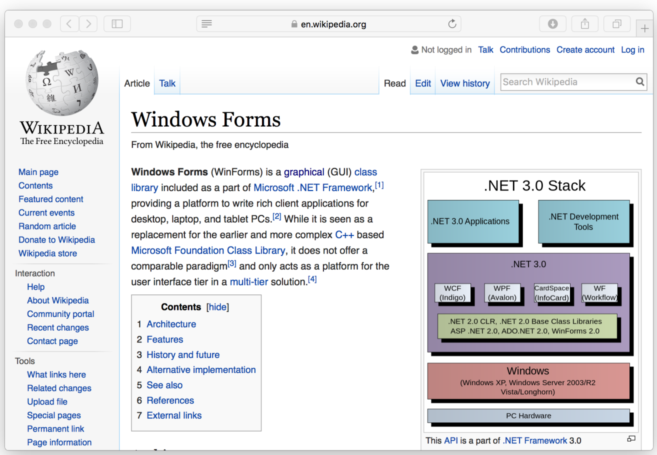


Fig. 1.1 A web-browser is a graphical user interface for accessing a web-server and interacting with its services. Here the browser is showing the page `https://en.wikipedia.org/wiki/Windows_Forms` at time of writing.

program presents information to the user in terms of text and images, has active areas that may be activated by clicking, allows the user to go other web-pages by typing a URL or following hyperlinks, and can generate new pages through search queries.

F# includes a number of implementations of graphical user interfaces, and at time of writing, both *GTK+* and *WinForms 2.0* are supported on both the Microsoft .Net and the Mono platform. WinForms can be used without extra libraries during compilation, and therefore will be the subject of the following chapter.

WinForms is a set of libraries that simplifies many common tasks for applications, and in this chapter, we will focus on the graphical user interface part of WinForms. A *form* is a visual interface used to communicate information with the user, typically a window. Communication is done through *controls*, which are elements that display information or accept input. Examples of controls are a box with text, a button, and a menu. When the user gives input to a control element, this generates an *event* which you can write code to react to. WinForms is designed for *event-driven programming*, meaning that at runtime, most time is spent on waiting for the user to give input. See **??** for more on event-driven programming.

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.

1.1 Opening a Window

The namespaces *`System.Windows.Forms`* and *`System.Drawing`* are central for programming graphical user interfaces with WinForms. `System.Windows.Forms` includes code for generating forms, controls, and handling events. `System.Drawing` is used for low-level drawing, and it gives 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. All controls in `System.Windows.Forms` in Mono are drawn using `System.Drawing`.

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 1.1, and the result is shown in Figure 1.2. Note that the present version of WinForms on MacOs only works with the 32-bit implementation of mono, *`mono32`*, as demonstrated in the example. The `new System.Windows.Forms.Form ()` creates an object (See **??**), but does not display the window on the screen. We use the optional `new` keyword, since the form is an `IDisposable` object and may be implicitly disposed of. I.e., it ⋆ is recommended to **instantiate `IDisposable` objects using `new` to contrast them with other object types.** Executing `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

Listing 1.1 Create the window and turn over control to the operating system. See Figure 1.2.

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

```
$ fsharpc --nologo openWindow.fsx && mono32 openWindow.exe
```

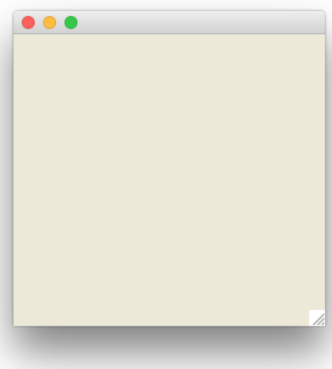


Fig. 1.2 A window opened by Listing 1.1.

of the window frame, and on Window it is the cross on the top right corner of the window frame.

The window 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 `Size`. This is demonstrated in Listing 1.2. These properties are *accessors*, implying that they act as mutable variables.

Listing 1.2 Create the window and change its properties. See Figure 1.3

```
// Prepare window form
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 "Color '%A' and Size '%A'" win.BackColor
   win.Size

// Start the event-loop.
System.Windows.Forms.Application.Run win
```

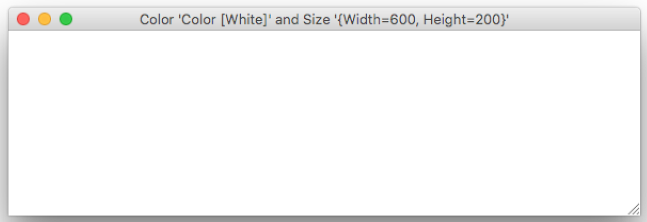


Fig. 1.3 A window with user-specified size and background color, see Listing 1.2.

1.2 Drawing Geometric Primitives

The *System.Drawing.Color* is a structure for specifying colors as 4 channels: alpha, red, green, and blue. Some methods and properties for the Color structure is shown in Table 1.1. Each channel is an 8-bit unsigned integer. The alpha channel

Method/Property	Description
Properties of an existing color structure	
`A : byte`	The value of the alpha channel.
`R : byte`	The value of the red channel.
`G : byte`	The value of the green channel.
`B : byte`	The value of the blue channel.
`ToArgb : unit -> int`	The 32-bit integer value of the color.
Static properties returning a color structure by its name.	
`Black : Color`	The ARGB value `0xFF000000`.
`Blue : Color`	The ARGB value `0xFF0000FF`.
`Brown : Color`	The ARGB value `0xFFA52A2A`.
`Gray : Color`	The ARGB value `0xFF808080`.
`Green : Color`	The ARGB value `0xFF00FF00`.
`Orange : Color`	The ARGB value `0xFFFFA500`.
`Purple : Color`	The ARGB value `0xFF800080`.
`Red : Color`	The ARGB value `0xFFFF0000`.
`White : Color`	The ARGB value `0xFFFFFFFF`.
`Yellow : Color`	The ARGB value `0xFFFFFF00`.
Static methods for converting between color structures and integers representations.	
`FromArgb :` `r:int * g:int * b:int -> Color`	Create a color structure from red, green, and blue values.
`FromArgb :` `a:int * r:int * g:int * b:int -> Color`	Create a color structure from alpha, red, green, and blue values.
`FromArgb : argb:int -> Color`	Create a color structure from a single integer.

Table 1.1 Some methods and properties of the `System.Drawing.Color` structure.

specifies the transparency of a color, where values 0–255 denote 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. As a shorthand, colors are often referred to as a single 32-bit unsigned integer, whose bits are organized in groups of 8 bits as 0xAARRGGBB, where AA is the alpha channel's values 0x00–0xFF etc. 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, the 4 alpha, red, green, and blue channels' values may be obtained as the `A`, `R`, `G`, and `B` members, see Listing 1.3

Listing 1.3 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
```

```
$ fsharpc --nologo drawingColors.fsx && mono drawingColors.exe
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 namespace `System.Drawing` contains many useful functions and values. Listing 1.2 used *`System.Drawing.Size`* to specify a size by a pair of integers. Other important values and functions are *`Point`*, which specifies a coordinate as a pair of points; *`Pen`*, which specifies how to draw lines and curves; *`Font`*, which specifies the font of a string; *`SolidBrush`* and *`TextureBrush`*, used for filling geometric primitives, and *`Bitmap`*, which is a type of *`Image`*. These are summarized in Table 1.2.

The *`System.Drawing.Graphics`* is a class for drawing geometric primitives to a display device, and some of its methods are summarized in Table 1.3.

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*. Both coordinate systems have their origin in the top-left corner, with

Constructor	Description
`Bitmap(int, int)`	Create a new empty `Image` of specified size.
`Bitmap(Stream)` `Bitmap(string)`	Create a `Image` from a `System.IO.Stream` or from a file specified by a filename.
`Font(string, single)`	Create a new font from the font's name and em-size.

`Pen(Brush)`
`Pen(Brush), single)`
`Pen(Color)`
`Pen(Color, single)`
	Create a pen to paint either with a brush or solid color and possibly with specified width.

`Point(int, int)`
`Point(Size)`
`PointF(single, single)`
	Create an ordered pair of integers or singles specifying x- and y-coordinates in the plane.

`Size(int, int)`
`Size(Point)`
`SizeF(single, single)`
`SizeF(PointF)`
	Create an ordered pair of integers or singles specifying height and width in the plane.

`SolidBrush(Color)` `TextureBrush(Image)`	Create a `Brush` as a solid color or from an image to fill the interior of geometric shapes.

Table 1.2 Basic geometrical structures in WinForms. `Brush` and `Image` are abstract classes.

Constructor	Description
`DrawImage : Image * (Point []) -> unit` `DrawImage : Image * (PointF []) -> unit`	Draw an image at a specific point and size.
`DrawImage : Image * Point -> unit` `DrawImage : Image * PointF -> unit`	Draw an image at a specific point.
`DrawLines : Pen * (Point []) -> unit` `DrawLines : Pen * (PointF []) -> unit`	Draw a series of lines between the n'th and $n + 1$'th points.
`DrawString :` `string * Font * Brush * PointF -> uni`	Draw a string at the specified point.

Table 1.3 Basic geometrical structures in WinForms.

the first coordinate, x, increasing to the right, and the second, y, increasing down, as illustrated in Figure 1.4. The Screen coordinate system has its origin in the top-left

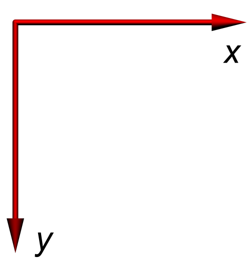


Fig. 1.4 Coordinate systems in Winforms have the y axis pointing down.

corner of the screen, while the client coordinate system has its origin in the top-left corner of 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, will be discussed later. Conversion between client and screen coordinates is done with *System.Drawing.PointToClient* and *System.Drawing.PointToScreen*.

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 form's *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.*

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 at any amount of times as deemed necessary by the operating system. An example of such a program is shown in Listing 1.4. A walk-through of the code is as follows: First, we open the

Listing 1.4 Adding line graphics to a window. See Figure 1.5

```
// Open often used libraries, beware of namespace polution!
open System.Windows.Forms
open System.Drawing

// Prepare window form
let win = new Form ()
win.Size <- Size (320, 170)

// Set paint call-back function
let paint (e : PaintEventArgs) : unit =
  let pen = new Pen (Color.Black)
  let points =
    [|Point (0,0); Point (10,170); Point (320,20); Point (0,0)|]
  e.Graphics.DrawLines (pen, points)
win.Paint.Add paint

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

two libraries that we will use heavily. This will save us some typing, but also pollute our namespace. E.g., now `Point` and `Color` are existing types, and we cannot define our own identifiers with these names. Then we create the form with size 320×170, we add a paint call-back function, and we start the event-loop. The event-loop will call the paint function, whenever the system determines that the window's content needs to be refreshed. 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. The function `paint` chooses a pen and a set of points and draws a set of lines connecting the points.

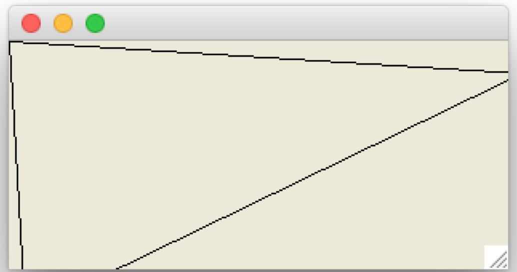


Fig. 1.5 Drawing a triangle using Listing 1.4.

The code in Listing 1.4 is not optimal. Despite the fact that the triangle spans the rectangle $(0, 0)$ to $(320, 170)$ and the window's size is set to $(320, 170)$, our window is too small and the triangle is clipped at the window border. The error is that we set the window's `Size` property, which determines the size of the window including top bar and borders. Alternatively, we may set the `ClientSize`, which determines the size of the drawable area, and this is demonstrated in Listing 1.5 and Figure 1.6. Thus, **prefer the `ClientSize` over the `Size` property for internal consistency.**

Listing 1.5 Adding line graphics to a window. See Figure 1.6.

```
// Open often used libraries, beware of namespace polution!
open System.Windows.Forms
open System.Drawing

// Prepare window form
let win = new Form ()
win.ClientSize <- Size (320, 170)

// Set paint call-back function
let paint (e : PaintEventArgs) : unit =
  let pen = new Pen (Color.Black)
  let points =
    [|Point (0,0); Point (10,170); Point (320,20); Point
   (0,0)|]
  e.Graphics.DrawLines (pen, points)
win.Paint.Add paint

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

Considering the program in Listing 1.4, we may identify a part that concerns the specification of the triangle, or more generally the graphical model, and some which concern system specific details. For future maintenance, it is often a good idea to

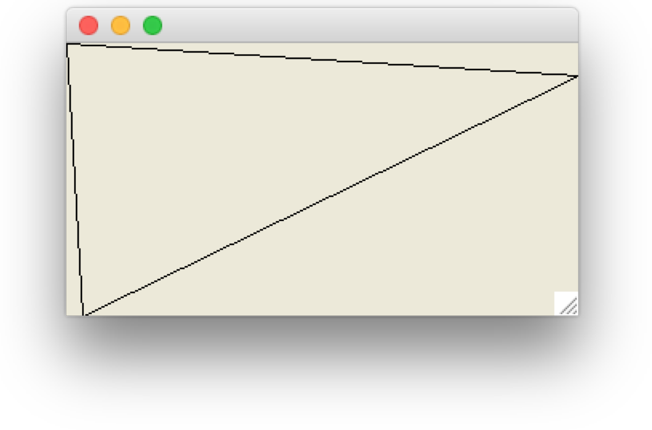


Fig. 1.6 Setting the ClientSize property gives a predictable drawing area, see Listing 1.5 for code.

separate the model from how it is viewed on a specific system. 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. We think of the model and the viewing part of the code as top and bottom layers, respectively, and these are often connected with a connection layer. This *Model-View paradigm* is shown in Figure 1.7. While it is not easy to completely separate the general from the specific, ★

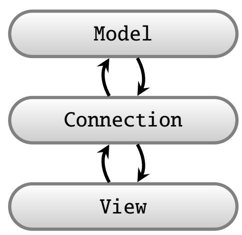


Fig. 1.7 Separating model from view gives flexibility later.

it is often a good idea to strive for some degree of separation.

In Listing 1.6, the program has been redesigned to follow the Model-View paradigm, where `view` contains most of the WinForms-specific code, and `model` contains most of the geometry, which could be reused with other graphical user interfaces. The model still uses the geometric primitives from WinForms for brevity, since a general implementation of geometric primitives avoiding WinForms would have a very similar interface. This program is longer, but there is a much better separation of *what* is to be displayed (model) from *how* it is to be done (view).

To further our development of a general program for displaying graphics, consider the case where we are to draw another two triangles, that are a translation and rotations

Listing 1.6 Improved organization of code for drawing a triangle. See Figure 1.8.

```
// Open often used libraries, beware of namespace polution!
open System.Windows.Forms
open System.Drawing

///////////// WinForm specifics /////////////
/// Setup a window form and return function which can activate it
let view (sz : Size) (pen : Pen) (pts : Point []) : (unit -> unit) =
  let win = new System.Windows.Forms.Form ()
  win.ClientSize <- sz
  win.Paint.Add (fun e -> e.Graphics.DrawLines (pen, pts))
  fun () -> Application.Run win // function as return value

///////////// Model /////////////
// A black triangle, using winform primitives for brevity
let model () : Size * Pen * (Point []) =
  let size = Size (320, 170)
  let pen = new Pen (Color.FromArgb (0, 0, 0))
  let lines =
    [|Point (0,0); Point (10,170); Point (320,20); Point (0,0)|]
  (size, pen, lines)

///////////// Connection /////////////
// Tie view and model together and enter main event loop
let (size, pen, lines) = model ()
let run = view size pen lines
run ()
```

of the original, and where we would like to specify the color of each triangle individually. A simple extension of `model` in Listing 1.6 for generating many shapes of different colors is `model : unit -> Size * ((Point []) * Pen) list`, i.e., semantically augment each point array with a pen and return a list of such pairs. For this example, we also program translation and rotation transformations. See Listing 1.7 for the result. We update `view` accordingly to iterate through this list as shown in Listing 1.8. Since we are using WinForms primitives in the model, the connection layer is trivial, as shown in Listing 1.9.

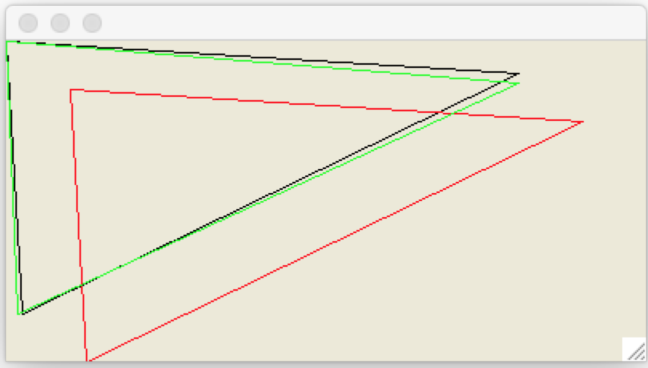


Fig. 1.8 Better organization of the code for drawing a triangle, see Listing 1.6.

Listing 1.7 Model of a triangle and simple transformations of it. See also Listing 1.8 and 1.9.

```
///////////// Model /////////////
// A black triangle, using WinForm primitives for brevity
let model () : Size * ((Pen * (Point [])) list) =
  /// Translate a primitive
  let translate (d : Point) (arr : Point []) : Point [] =
    let add (d : Point) (p : Point) : Point =
      Point (d.X + p.X, d.Y + p.Y)
    Array.map (add d) arr

  /// Rotate a primitive
  let rotate (theta : float) (arr : Point []) : Point [] =
    let toInt = int << round
    let rot (t : float) (p : Point) : Point =
      let (x, y) = (float p.X, float p.Y)
      let (a, b) = (x * cos t - y * sin t, x * sin t + y * cos t)
      Point (toInt a, toInt b)
    Array.map (rot theta) arr

  let size = Size (400, 200)
  let lines =
    [|Point (0,0); Point (10,170); Point (320,20); Point (0,0)|]
  let black = new Pen (Color.FromArgb (0, 0, 0))
  let red = new Pen (Color.FromArgb (255, 0, 0))
  let green = new Pen (Color.FromArgb (0, 255, 0))
  let shapes =
    [(black, lines);
     (red, translate (Point (40, 30)) lines);
     (green, rotate (1.0 *System.Math.PI / 180.0) lines)]
  (size, shapes)
```

Listing 1.8 A view for lists of pairs of pen and point arrays. See also Listing 1.7 and 1.9.

```
// Open often used libraries, beware of namespace polution!
open System.Windows.Forms
open System.Drawing

///////////// WinForm specifics /////////////
/// Setup a window form and return function to activate
let view (sz : Size) (shapes : (Pen * (Point [])) list) :
    (unit -> unit) =
  let win = new Form ()
  win.ClientSize <- sz
  let paint (e : PaintEventArgs) ((p, pts) : (Pen * (Point
    []))) : unit =
    e.Graphics.DrawLines (p, pts)
  win.Paint.Add (fun e -> List.iter (paint e) shapes)
  fun () -> Application.Run win // function as return value
```

Listing 1.9 Model of a triangle and simple transformations of it. See also Listing 1.7 and 1.8.

```
///////////// Connection //////////////
// Tie view and model together and enter main event loop
let (size, shapes) = model ()
let run = view size shapes
run ()
```

1.3 Programming Intermezzo: Hilbert Curve

A curve in 2 dimensions has a length but no width, and we can only visualize it by giving it a width. Thus, it came as a surprise to many when Giuseppe Peano in 1890 demonstrated that there exist curves which fill every point in a square. The method he used to achieve this was recursion:

Problem 1.1

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 horizontal axis. To draw this curve, we need 3 basic operations: Move forward (F), turn right (R), and turn left (L). The turning is w.r.t. the present direction. A Hilbert Curve is a space-filling curve which can be expressed recursively as:

$$A \rightarrow LBFRAFARFBL, \tag{1.1}$$
$$B \rightarrow RAFLBFBLFAR, \tag{1.2}$$

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. To keep track of the level of recursion, we introduce an index as:

$$A_{n+1} \rightarrow LB_nFRA_nFA_nRFB_nL,$$
$$B_{n+1} \rightarrow RA_nFLB_nFB_nLFA_nR,$$

for $n > 0$ and $A_0 \rightarrow \emptyset$ and $B_0 \rightarrow \emptyset$. Thus, the first-order curve is

$$A_1 \rightarrow LB_0FRA_0FA_0RFB_0L \rightarrow LFRFRFL,$$

and the second order curve is

$$\begin{aligned} A_2 \rightarrow & LB_1FRA_1FA_1RFB_1L \\ \rightarrow & LRFLFLFRFRLFRFRFLFLFRFRFLRFRFLFLFRL. \end{aligned}$$

Since $LR = RL = \emptyset$ the above simplifies to

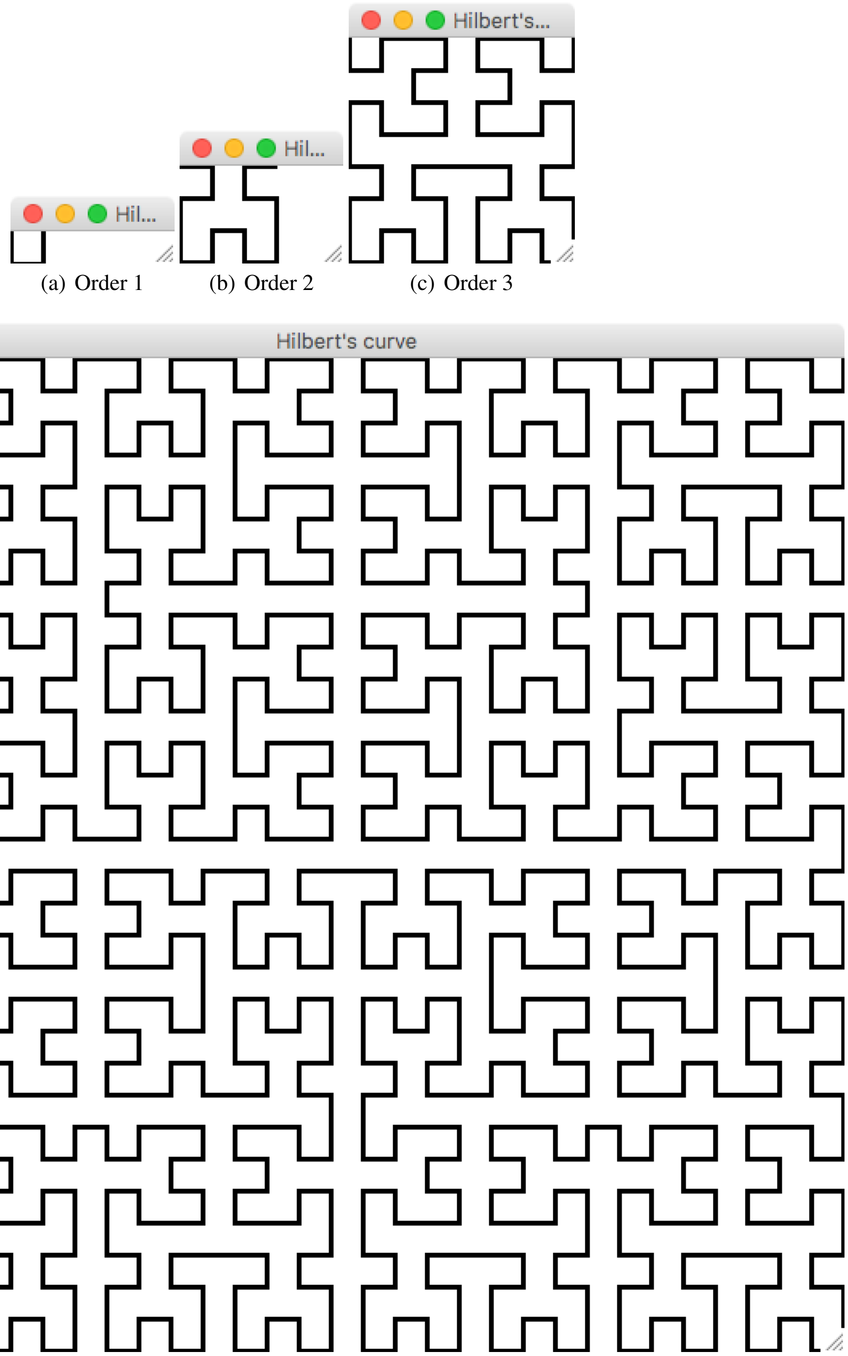
$$A_2 \rightarrow FLFLFRFFRFRFLFLFRFRFFRFLFLF$$

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

Our strategy to solve this problem will be first to define the curves in terms of movement commands $LRFL\ldots$. For this, we will define a discriminated union `type Command = F | L | R`. The movement commands can then be defined as a `Command list` type. The list for a specific order is a simple set of recursive functions in F# which we will call `A` and `B`.

To produce a graphical drawing of a command list, we must transform it into coordinates, and during the conversion, we need to keep track of both the present position and the present heading, since not all commands draw. This is a concept similar to Turtle Graphics, which is often associated with the Logo programming language from the 1960's. In Turtle graphics, we command a little robot - a turtle - which moves in 2 dimensions and can turn on the spot or move forward, and its track is the line being drawn. Thus we introduce a `type Turtle = {x : float; y : float; d : float}` record. Conversion of command lists to turtle lists is a fold programming structure, where the command list is read from left-to-right, building up an accumulator by adding each new element. For efficiency, we choose to prepend the new element to the accumulator. This we have implemented as the `addRev` function. Once the full list of turtles has been produced, then it is reversed.

Finally, the turtle list is converted to WinForms `Point` array, and a window of appropriate size is chosen. The resulting model part is shown in Listing 1.10. The view and connection parts are identical to Listing 1.8 and 1.9, and Figure 1.9 shows the result of using the program to draw Hilbert curves of orders 1, 2, 3, and 5.



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

(d) Order 5

Fig. 1.9 Hilbert curves of orders 1, 2, 3, and 5 by code in Listing 1.10.

Listing 1.10 Using simple turtle graphics to produce a list of points on a polygon. The code continues in Listing 1.11. The view and connection parts are identical to Listing 1.8 and 1.9.

```
// Turtle commands, type definitions must be in outermost scope
type Command = F | L | R
type Turtle = {x : float; y : float; d : float}

// A black Hilbert curve using WinForm primitives for brevity
let model () : Size * ((Pen * (Point [])) list) =
  /// Hilbert recursion production rules
  let rec A n : Command list =
    if n > 0 then
      [L]@B (n-1)@[F; R]@A (n-1)@[F]@A (n-1)@[R; F]@B (n-1)@[L]
    else
      []
  and B n : Command list =
    if n > 0 then
      [R]@A (n-1)@[F; L]@B (n-1)@[F]@B (n-1)@[L; F]@A (n-1)@[R]
    else
      []

  /// Convert a command to turtle record and prepend to list
  let addRev (lst : Turtle list) (cmd : Command) (len : float) : Turtle list =
    let toInt = int << round
    match lst with
      | t::rest ->
        match cmd with
          | L -> {t with d = t.d + 3.141592/2.0}::rest // left
          | R -> {t with d = t.d - 3.141592/2.0}::rest // right
          | F -> {t with x = t.x + len * cos t.d; // forward
                         y = t.y + len * sin t.d}::lst
      | _ -> failwith "Turtle list must be non-empty."

  let maxPoint (p1 : Point) (p2 : Point) : Point =
    Point (max p1.X p2.X, max p1.Y p2.Y)
```

Listing 1.11 Continued from Listing 1.10.

```
  // Calculate commands for a specific order
  let curve = A 5
  // Convert commands to point array
  let initTrtl = {x = 0.0; y = 0.0; d = 0.0}
  let len = 20.0
  let line =
    List.fold (fun acc elm -> addRev acc elm len) [initTrtl] curve // Convert command list to reverse turtle list
    |> List.rev // Reverse list
    |> List.map (fun t -> Point (int (round t.x), int (round t.y))) // Convert turtle list to point list
    |> List.toArray // Convert point list to point array
  let black = new Pen (Color.FromArgb (0, 0, 0))
  // Set size to as large as shape
  let minVal = System.Int32.MinValue
  let maxPoint = Array.fold maxPoint (Point (minVal, minVal)) line
  let size = Size (maxPoint.X + 1, maxPoint.Y + 1)
  (size, [(black, line)]) // return shapes as singleton list
```

1.4 Handling Events

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 handlers that we may use to interact with the user. Listing 1.12 demonstrates event handlers for moving and resizing a window, for clicking in a window, and for typing on the keyboard.
Listing 1.12 shows the output from an interaction with the program which is the

Listing 1.12 Catching window, mouse, and keyboard events.

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

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

// 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; // records when button down
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

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

result of the following actions: moving the window, resizing the window, clicking

Listing 1.13: Output from an interaction with the program in Listing 1.12.

```
Move: {X=22,Y=22}
Move: {X=22,Y=22}
Move: {X=50,Y=71}
Resize: {X=0,Y=0,Width=307,Height=290}
MouseDown: {X=144,Y=118}
MouseClick: {X=144,Y=118}
MouseUp: {X=144,Y=118}
MouseDown: {X=144,Y=118}
MouseUp: {X=144,Y=118}
MouseDown: {X=96,Y=66}
MouseMove: {X=96,Y=67}
MouseMove: {X=97,Y=69}
MouseMove: {X=99,Y=71}
MouseMove: {X=103,Y=74}
MouseMove: {X=107,Y=77}
MouseMove: {X=109,Y=79}
MouseMove: {X=112,Y=81}
MouseMove: {X=114,Y=82}
MouseMove: {X=116,Y=84}
MouseMove: {X=117,Y=85}
MouseMove: {X=118,Y=85}
MouseClick: {X=118,Y=85}
MouseUp: {X=118,Y=85}
KeyPress: "a"
KeyPress: "b"
KeyPress: "c"
```

the left mouse key, pressing and holding the down the left mouse key while moving the mouse, releasing the left mouse key, and typing “abc”. As demonstrated, some actions, like moving the mouse, result in a lot of events, and some, like the initial window moves results, are surprising. Thus, event-driven programming should take care to interpret the events robustly and carefully.

Common for all event-handlers is that they listen for an event, and when the event occurs, the functions that have been added using the `Add` method are called. This is also known as sending a message. Thus, a single event can give rise to calling zero or more functions.

Graphical user interfaces and other systems often need to perform actions that depend on specific lengths of time or a certain point in time. To measure length of time F# has the *`System.Windows.Forms.Timer`* class, which technically is an optimized of `System.Timers.Timer` for graphical user interfaces. The Timer class can be used to create an event after a specified duration of time. F# also has the *`System.DateTime`* class to specify points in time. An often used property is `System.DateTime.Now`, which returns a `DateTime` object for the date and time when the property is accessed. The use of these two classes is demonstrated in Listing 1.14 and Figure 1.10. In the code, a label has been created to show the present date and time. The label is a type

Listing 1.14 Using `System.Windows.Forms.Timer` and `System.DateTime.Now` to update the display of the present date and time. See Figure 1.10 for the result.

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

let win = new Form () // make a window form
win.ClientSize <- Size (200, 50)

// make a label to show time
let label = new Label()
win.Controls.Add label
label.Width <- 200
label.Text <- string System.DateTime.Now // get present time and date

// make a timer and link to label
let timer = new Timer()
timer.Interval <- 1000 // create an event every 1000 millisecond
timer.Enabled <- true // activate the timer
timer.Tick.Add (fun e -> label.Text <- string System.DateTime.Now)

Application.Run win // start event-loop
```

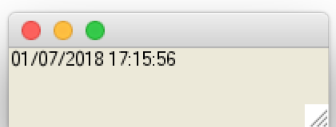


Fig. 1.10 See Listing 1.14.

of control, and it is displayed using the default font which is rather small. How to change this and other details on controls will be discussed in the next section.

In the example, the label is redrawn everytime the text is changed, such that the current value is correctly displayed on the screen. Sometimes it is necessary to force a control to redraw which can be done with the `Refresh()` method. Since a `Form` is also a type of control, it is common to trigger a redraw event for the top form, which in Listing 1.14 would be `win.Refresh()`. Thus, `Refresh()` and a `Timer` object can be used to produce animations.

1.5 Labels, Buttons, and Pop-up Windows

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. Listing 1.15 shows a small program that displays a label and a button in a window, and when the button is pressed, then the label is updated. As the listing demonstrates, the button is created using the *`System.Windows.Forms.Button`* constructor, and it is added to the window's form's control list. The `Location` property controls its position w.r.t. the enclosing form. Other accessors are `Width`, `Text`, and `Size`.

Listing 1.15 Create the button and an event, see also Figure 1.11.

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

let win = new Form () // make a window form
win.ClientSize <- Size (140, 120)

// Create a label
let label = new Label()
win.Controls.Add label
label.Location <- new Point (20, 20)
label.Width <- 120
let mutable clicked = 0
let setLabel clicked =
  label.Text <- sprintf "Clicked %d times" clicked
setLabel clicked

// Create a button
let button = new Button ()
win.Controls.Add button
button.Size <- new Size (100, 40)
button.Location <- new Point (20, 60)
button.Text <- "Click me"
button.Click.Add (fun e -> clicked <- clicked + 1; setLabel
   clicked)

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

`System.Windows.Forms` includes a long list of controls, some of which are summarized in Table 1.4. Examples are given in `controls`, shown in Listing 1.16 and Figure 1.12.

Some controls open separate windows for more involved dialogue with the user. Some examples are `MessageBox`, `OpenFileDialog`, and `SaveFileDialog`.

Fig. 1.11 After pressing the button 3 times. See Listing 1.15.

Method/Property	Description
`Button`	A clickable button.
`CheckBox`	A clickable check box.
`DateTimePicker`	A box showing a date with a drop-down menu for choosing another.
`Label`	A displayable text.
`ProgressBar`	A box showing a progress bar.
`RadioButton`	A single clickable radio button. Can be paired with other radio buttons.
`TextBox`	A text area, which can accept input from the user.

Table 1.4 Some types of `System.Windows.Forms.Control`.

`System.Windows.Forms.MessageBox` is used to have a simple but restrictive dialogue with the user, which is demonstrated in Listing 1.17 and Figure 1.13. As an alternative to the `YesNo` response button, the message box also offers `AbortRetryIgnore`, `OK`, `OKCancel`, `RetryCancel`, and `YesNoCancel`. Note that all other windows of the process are blocked until the user closes the dialogue window.

With *`System.Windows.Forms.OpenFileDialog`*, you can ask the user to select an existing filename, as demonstrated in Listing 1.18 and Figure 1.14. Similarly to `OpenFileDialog`, *`System.Windows.Forms.SaveFileDialog`* asks for a file name, but if an existing file is selected, then the user will be asked to confirm the choice.

Listing 1.16 Examples of control elements added to a window form, see also Figure 1.12.

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

let win = new Form () // Create a window
win.ClientSize <- Size (300, 300)

let button = new Button () // Make a button
win.Controls.Add button
button.Location <- new Point (20, 20)
button.Text <- "Click me"

let lbl = new Label () // Add a label
win.Controls.Add lbl
lbl.Location <- new Point (20, 60)
lbl.Text <- "A text label"

let chkbox = new CheckBox () // Add a check box
win.Controls.Add chkbox
chkbox.Location <- new Point (20, 100)

let pick = new DateTimePicker () // Add a date and time picker
win.Controls.Add pick
pick.Location <- new Point (20, 140)

let prgrss = new ProgressBar () // Show a progress bar
win.Controls.Add prgrss
prgrss.Location <- new Point (20, 180)
prgrss.Minimum <- 0
prgrss.Maximum <- 9
prgrss.Value <- 3

let rdbttn = new RadioButton () // Add a radio button
win.Controls.Add rdbttn
rdbttn.Location <- new Point (20, 220)

let txtbox = new TextBox () // Add a text input field
win.Controls.Add txtbox
txtbox.Location <- new Point (20, 260)
txtbox.Text <- "Type something"

Application.Run win // Show everything and start event-loop
```

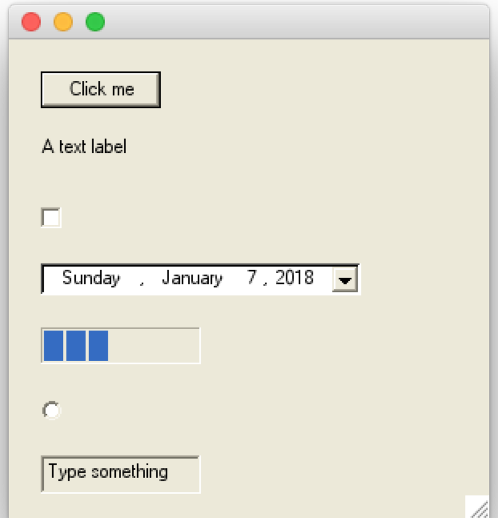


Fig. 1.12 Examples of control elements. See Listing 1.16.

Listing 1.17 Create the MessageBox, see also Figure 1.13.

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

let win = new Form ()
win.ClientSize <- Size (140, 80)

let button = new Button ()
win.Controls.Add button
button.Size <- new Size (100, 40)
button.Location <- new Point (20, 20)
button.Text <- "Click me"
// Open a message box when button clicked
let buttonClicked (e : EventArgs) =
  let question = "Is this statement false?"
  let caption = "Window caption"
  let boxType = MessageBoxButtons.YesNo
  let response = MessageBox.Show (question, caption, boxType)
  printfn "The user pressed %A" response
button.Click.Add buttonClicked

Application.Run win
```

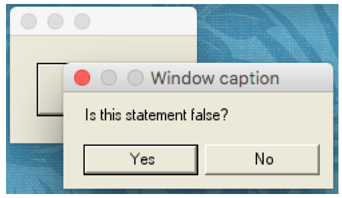


Fig. 1.13 After pressing the “Click-me” button. See Listing 1.17.

Listing 1.18 Create the `OpenFileDialog`, see also Figure 1.14.

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

let win = new Form ()
win.ClientSize <- Size (140, 80)

let button = new Button ()
win.Controls.Add button
button.Size <- new Size (100, 40)
button.Location <- new Point (20, 20)
button.Text <- "Click me"
// Open a message box when button clicked
let buttonClicked (e : EventArgs) =
  let opendlg = new OpenFileDialog()
  let okOrCancel = opendlg.ShowDialog()
  printfn "The user pressed %A and selected %A" okOrCancel
   opendlg.FileName
button.Click.Add buttonClicked

Application.Run win
```

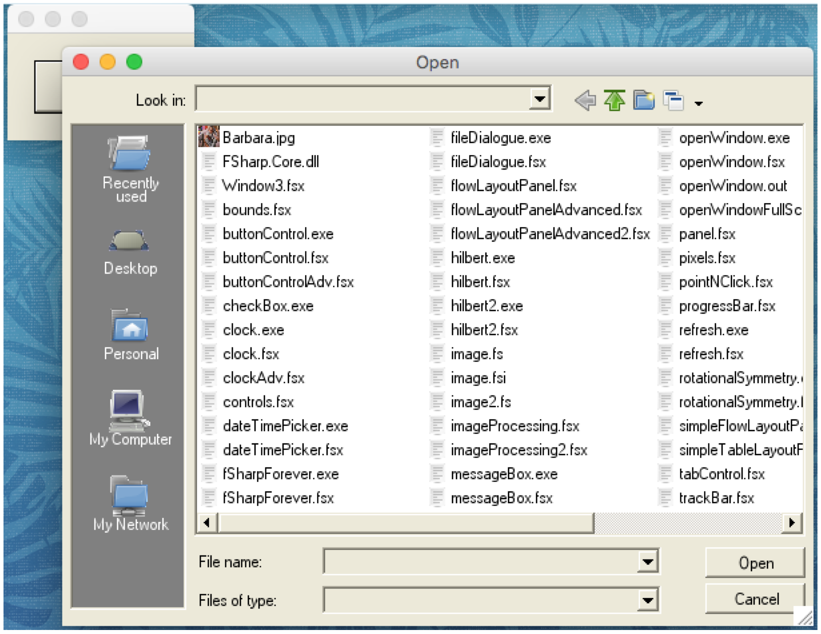


Fig. 1.14 Ask the user for a filename to read from. See Listing 1.18.

1.6 Organizing Controls

It is often useful to organize the controls in groups, and such groups are called *Panels* in WinForms. 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 1.19 and Figure 1.15. The label and textbox

Listing 1.19 Create a panel, label, and text input controls.

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

let win = new Form () // Create a window form
win.ClientSize <- new Size (200, 100)

// Customize the Panel control
let panel = new Panel ()
panel.ClientSize <- new Size (160, 60)
panel.Location <- new Point (20,20)
panel.BorderStyle <- BorderStyle.Fixed3D
win.Controls.Add panel // Add panel to window

// Customize the Label and TextBox controls
let label = new Label ()
label.ClientSize <- new Size (120, 20)
label.Location <- new Point (15,5)
label.Text <- "Input"
panel.Controls.Add label // add label to panel

let textBox = new TextBox ()
textBox.ClientSize <- new Size (120, 20)
textBox.Location <- new Point (20,25)
textBox.Text <- "Initial text"
panel.Controls.Add textBox // add textbox to panel

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

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.

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 1.20–1.21 and in Figure 1.16. The program illustrates how the button elements flow under four possible flow directions with `System.Windows.FlowDirection`, and how `System.Windows.WrapContents` influences what happens to content that

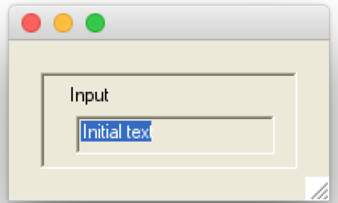


Fig. 1.15 A panel including a label and a text input field, see Listing 1.19.

flows outside the panel's region. A walkthrough of the program is as follows. The

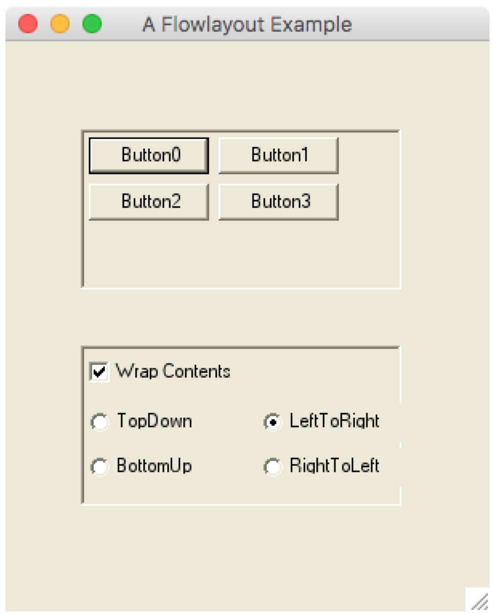


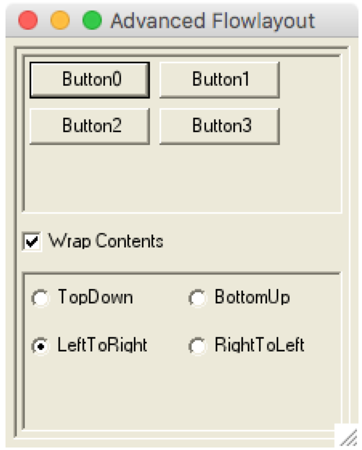
Fig. 1.16 Demonstration of the `FlowLayoutPanel` panel, `CheckBox`, and `RadioButton` controls, see Listing 1.20–1.21.

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 `Button`s 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 the `buttonLst.[0]` in the top left corner, and `buttonLst.[1]` to its right. Other flow directions do it differently, and the reader is encouraged to experiment with the program.

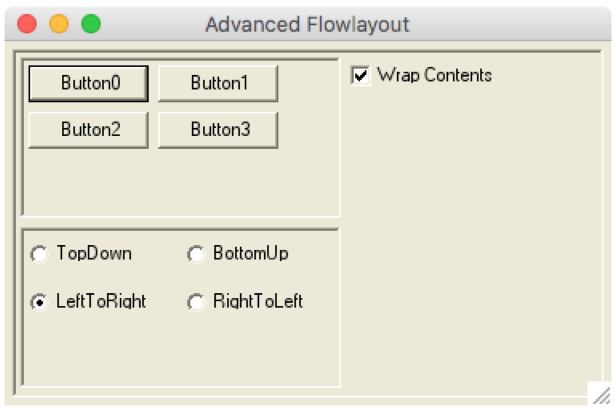
Listing 1.20 Create a FlowLayoutPanel with checkbox and radio buttons.

```
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)
```

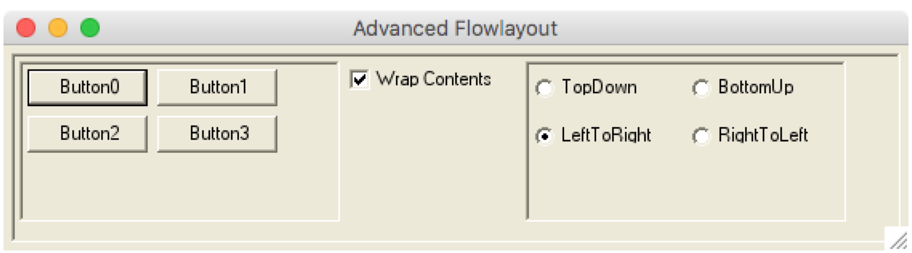
The program in Listing 1.21 has not completely separated the semantic blocks of the interface and relies on explicit setting of coordinates of controls. This can be avoided by using nested panels. E.g., in Listing 1.22–1.23, the program has been rewritten as a nested set of `FloatLayoutPanel` in three 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. This results in three different views, all shown in Figure 1.17.



(a)



(b)



(c)

Fig. 1.17 Nested `FlowLayoutPanel`, see Listing 1.22–1.23, allows for dynamic arrangement of content. Content flows when the window is resized.

Listing 1.21 Create a FlowLayoutPanel with checkbox and radio buttons. Continued from Listing 1.20.

```
// 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 controls, 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
```

Listing 1.22 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 1.23 Create nested FlowLayoutPanel. Continued from Listing 1.22.

```
// 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

// customize window, add controls, 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
```