Learning to program with F#

Jon Sporring

November 16, 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.

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

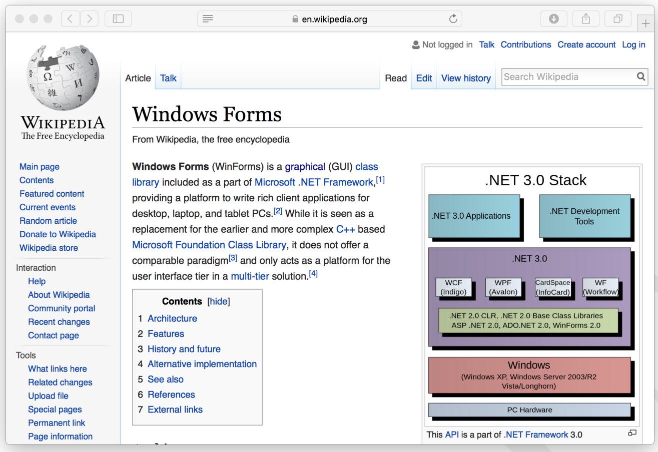


Figure 13.1: A web-browser is a graphical user interface for accessing a web-server and interacting with its services. Here the browser is showing the page `https://en.wikipedia.org/wiki/Windows_Forms` at time of writing.

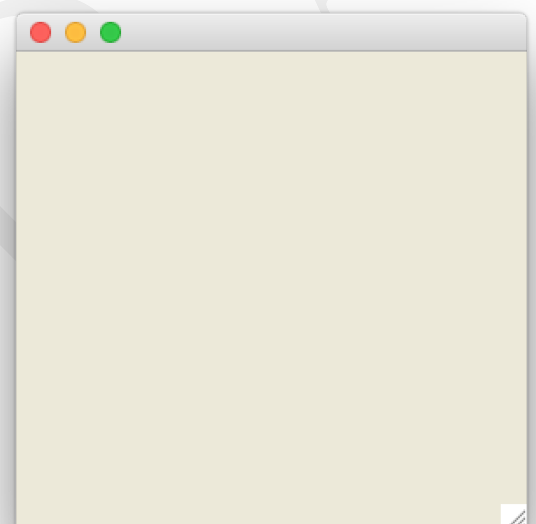


Figure 13.2: A window opened by Listing 13.1.

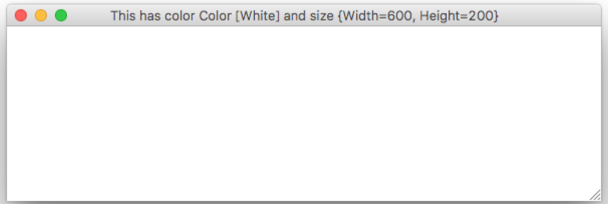


Figure 13.3: A window with user-specified size and background color, see Listing 13.2.

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.
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. Drawing is done with reference to a coordinate system. WinForms operates with several coordinate systems, the most important is the *screen coordinates*. Screen coordinate (x, y) have their origin in the top-left corner, and x increases to the right, while y increases down.[1] Thus, we may draw a triangle as demonstrated in Listing 13.4.

· call-back function

· screen coordinates

[1] Todo: **Possibly something about client coordinates and world coordinates.**

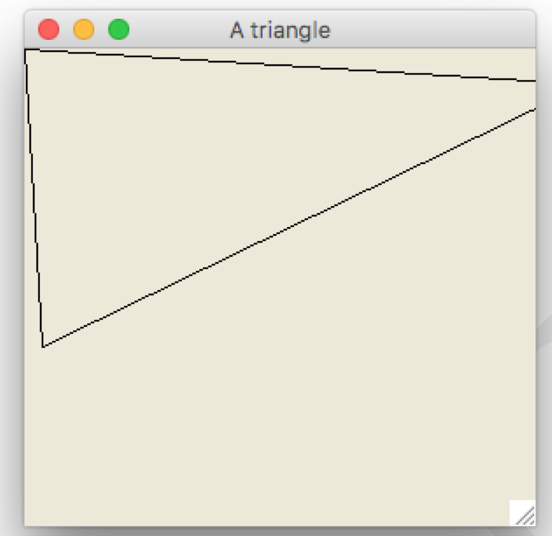


Figure 13.4: Drawing a triangle using Listing 13.4.

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 `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.

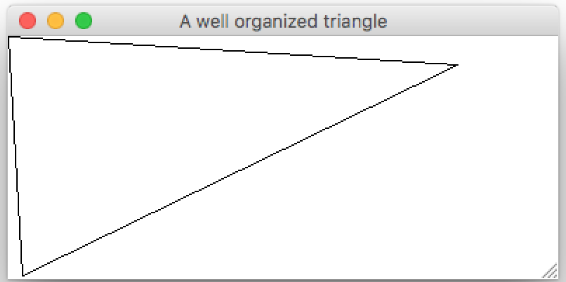


Figure 13.5: Better organization of the code for drawing a triangle, see Listing 13.5.

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
```

[2][3]

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
```

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

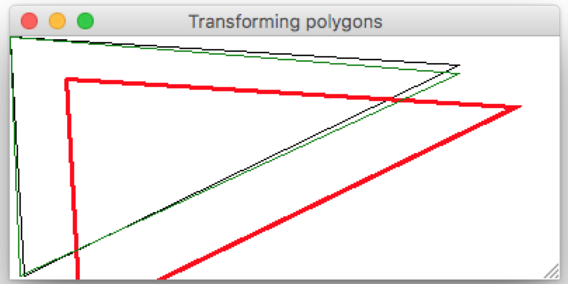


Figure 13.6: Transformed versions of the same triangle resulting from running the code in Listing 13.7.

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

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
```

[4]

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

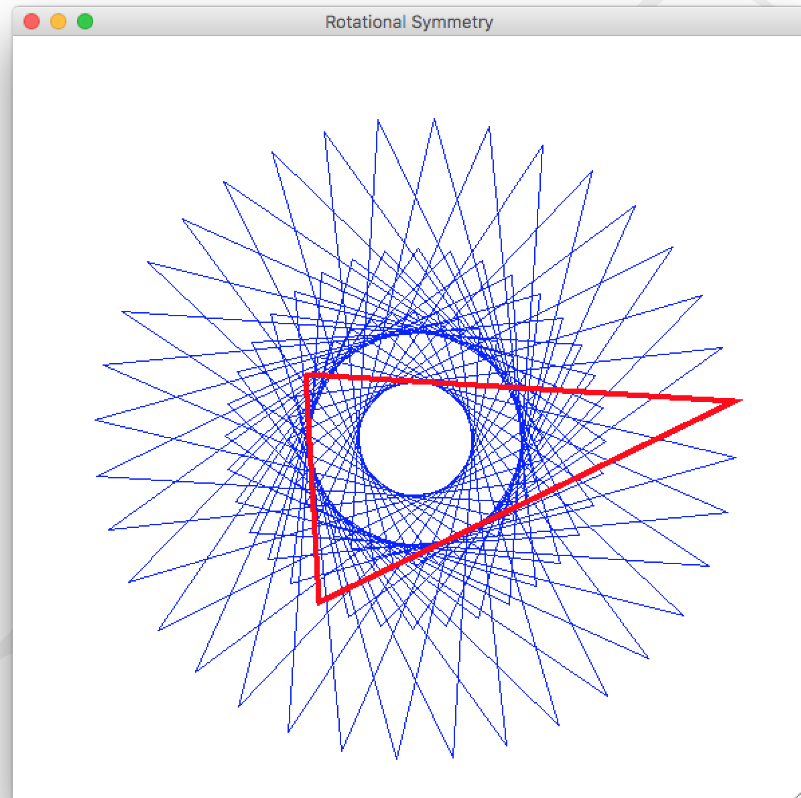


Figure 13.7: A symmetric figure resulting from Listing 13.8.

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.
`DrawClosedCurve : Pen * Point[]`	Draws a closed cardinal spline defined by an array of 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.
`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.
`FillRegion : Brush * Region`	Fills the interior of a Region.

Table 13.1: Some methods of the `System.IO.Path` class.

13.2 Programming intermezzo

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: Draw (F), turn right ($+$), and turn left ($-$). The turning is w.r.t. the present diretion. A Hilbert Curve is a spacefilling curve, which be expressed recursively as:

$$A \to -BF + AFA + FB- \tag{13.1}$$
$$B \to +AF - BFB - FA+ \tag{13.2}$$

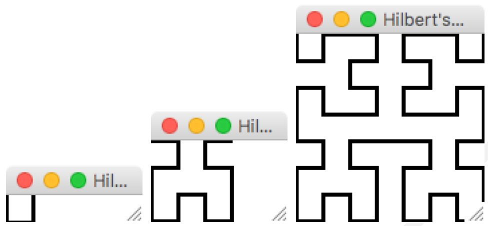
starting with A. The order of the curve is the depth of the recursion, and to draw a 0'th order curve, we don't recurse at all, i.e., ignore all occurrences of the symbols A and B on the right-hand-side of (13.1), and get $-F + F + F-$. For the 1'st order curve, we recurse once, i.e.,

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

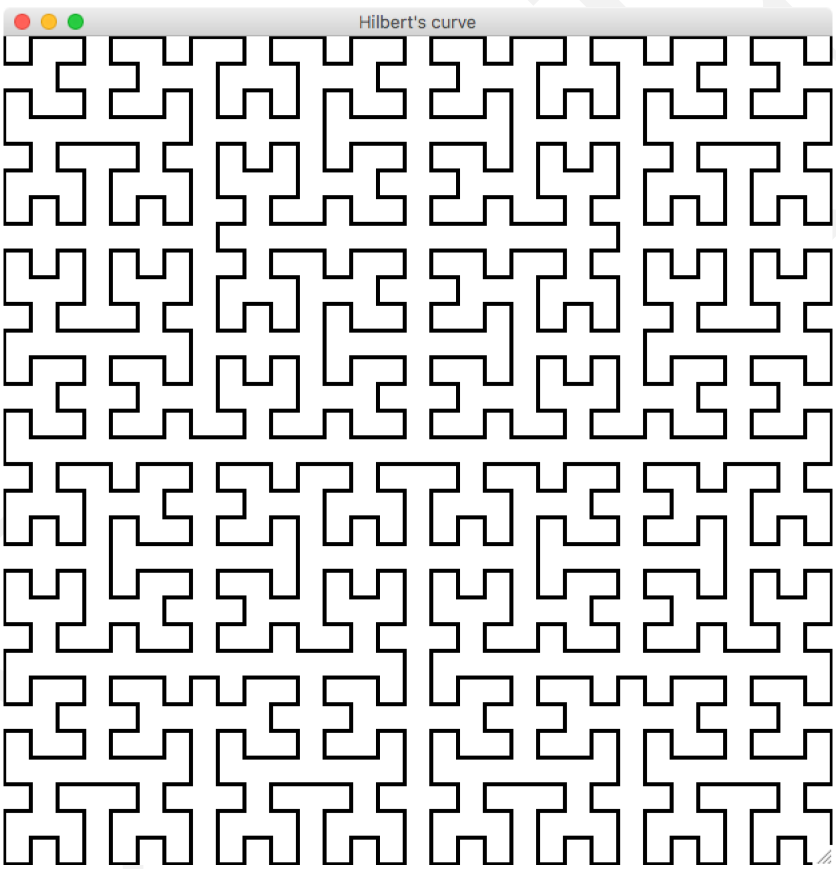
Make a program, that given an order produces an image of the Hilbert curve.

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

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

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

/// Add a line to the curve of present direction
let draw (l, dir, (c : coordinates)) =
  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 =
  let maxPoint (p1 : float*float) (p2 : float*float) =
    (max (fst p1) (fst p2), max (snd p1) (snd p2))
  List.fold maxPoint (-infinity, -infinity) c

/// Hilbert recursion production rules
let rec hilbertA n (l, dir, c) =
  if n > 0 then
    ((l, dir, c) |> turnLeft |> hilbertB (n-1) |> draw |> turnRight |>
   hilbertA (n-1) |> draw |> hilbertA (n-1) |> turnRight |> draw |>
   hilbertB (n-1) |> turnLeft)
  else
    (l, dir, c)
and hilbertB n (l, dir, c) =
  if n > 0 then
    ((l, dir, c) |> turnRight |> hilbertA (n-1) |> draw |> turnLeft |>
   hilbertB (n-1) |> draw |> hilbertB (n-1) |> turnLeft |> draw |>
   hilbertA (n-1) |> turnRight)
  else
    (l, dir, c)

// Calculate curve
let order = 5
let l = 20.0
let (_, dir, C) = hilbertA 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)
System.Windows.Forms.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.

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

```
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)
  // Paint event
  win.Paint.Add draw
  // Window event
  win.Move.Add (fun e -> printfn "Move: %A" win.Location)
  win.Resize.Add (fun _ -> printfn "Resize: %A" win.DisplayRectangle)
  // Mouse event
  let mutable record = false;
  win.MouseMove.Add (fun e -> if record then printfn "MouseMove: %A" e.Location)
  win.MouseDown.Add (fun e -> printfn "MouseDown: %A" e.Location; (record <- true))
  win.MouseUp.Add (fun e -> printfn "MouseUp: %A" e.Location; (record <- false))
  win.MouseClick.Add (fun e -> printfn "MouseClick: %A" e.Location)
  // Keyboard event
  win.KeyPreview <- true
  win.KeyPress.Add (fun e -> printfn "KeyPress: %A" (e.KeyChar.ToString ()))
  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)

let backgroundColor = System.Drawing.Color.White
let title = "Window events"
let size = (200, 200)
let polygLst = []

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

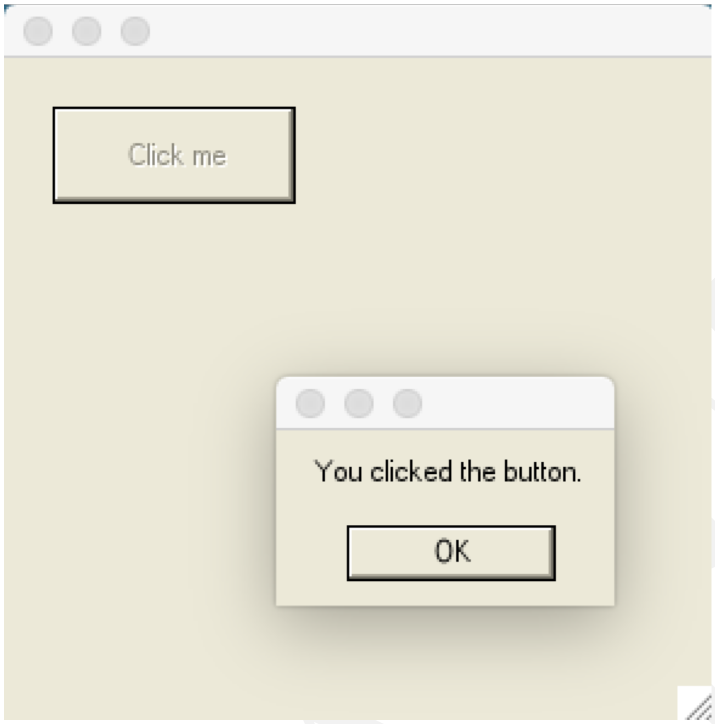


Figure 13.9: A button is pressed and the event handler calls the `MessageBox.Show` dialogue window by the code in Listing 13.11.

13.3 Buttons and stuff

Listing 13.11, winforms/buttonControl.fsx:
Create the button and an event.

```
/// A button event
let buttonClicked (e : System.EventArgs) =
  ignore (System.Windows.Forms.MessageBox.Show "You clicked the button.")

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

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

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

Listing 13.12, winforms/panel.fsx:
Create a panel, label, text input controls.

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

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

// Customize the Form.
form.Text <- "A panel";
form.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 <- "label1";
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 form and label and textBox to panel.
form.Controls.Add(panel);
panel.Controls.Add(label);
panel.Controls.Add(textBox);

// Give control to WinForms' event loop
Application.Run form
```

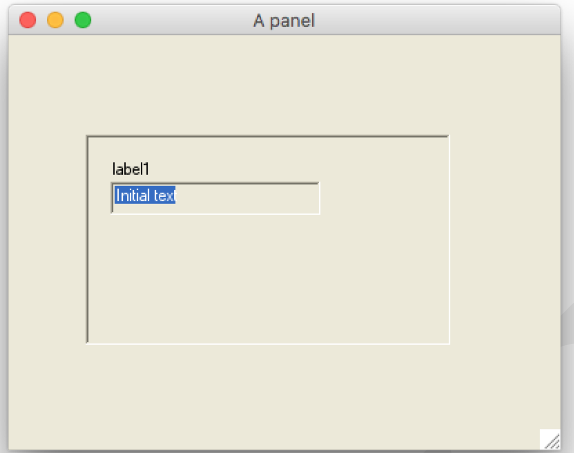


Figure 13.10: A panel including a label and a text input field, see Listing 13.12.

Listing 13.13, winforms/flowLayoutPanel.fsx:
Create a flowLayoutPanel, with checkbox and radiobuttons.

```
open System
open System.Windows.Forms

let flowLayoutPanel = new System.Windows.Forms.FlowLayoutPanel ();
let button1 = new System.Windows.Forms.Button ();
let button2 = new System.Windows.Forms.Button ();
let button3 = new System.Windows.Forms.Button ();
let button4 = new System.Windows.Forms.Button ();
let wrapContentsCheckBox = new System.Windows.Forms.CheckBox ();
let flowTopDownBtn = new System.Windows.Forms.RadioButton ();
let flowBottomUpBtn = new System.Windows.Forms.RadioButton ();
let flowLeftToRight = new System.Windows.Forms.RadioButton ();
let flowRightToLeftBtn = new System.Windows.Forms.RadioButton ();

//
// button1
//
button1.Location <- new System.Drawing.Point (3, 3);
button1.Name <- "button1";
button1.TabIndex <- 0;
button1.Text <- "button1";
//
// button2
//
button2.Location <- new System.Drawing.Point (84, 3);
button2.Name <- "button2";
button2.TabIndex <- 1;
button2.Text <- "button2";
//
// button3
//
button3.Location <- new System.Drawing.Point (3, 32);
button3.Name <- "button3";
button3.TabIndex <- 2;
button3.Text <- "button3";
//
// button4
//
```

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

```
//
// wrapContentsCheckBox
//
wrapContentsCheckBox.Location <- new System.Drawing.Point (46, 162);
wrapContentsCheckBox.Name <- "wrapContentsCheckBox";
wrapContentsCheckBox.TabIndex <- 1;
wrapContentsCheckBox.Text <- "Wrap Contents";
wrapContentsCheckBox.Checked <- true
wrapContentsCheckBox.CheckedChanged.Add (fun _ -> flowLayoutPanel.
    WrapContents <- wrapContentsCheckBox.Checked)
//
// flowTopDownBtn
//
flowTopDownBtn.Location <- new System.Drawing.Point (45, 193);
flowTopDownBtn.Name <- "flowTopDownBtn";
flowTopDownBtn.TabIndex <- 2;
flowTopDownBtn.Text <- "Flow TopDown";
flowTopDownBtn.Checked <- flowLayoutPanel.FlowDirection = FlowDirection.
    TopDown;
flowTopDownBtn.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection
    <- FlowDirection.TopDown);
//
// flowBottomUpBtn
//
flowBottomUpBtn.Location <- new System.Drawing.Point (44, 224);
flowBottomUpBtn.Name <- "flowBottomUpBtn";
flowBottomUpBtn.TabIndex <- 3;
flowBottomUpBtn.Text <- "Flow BottomUp";
flowBottomUpBtn.Checked <- flowLayoutPanel.FlowDirection = FlowDirection.
    BottomUp
flowBottomUpBtn.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection
     <- FlowDirection.BottomUp);
//
// flowLeftToRight
//
flowLeftToRight.Location <- new System.Drawing.Point (156, 193);
flowLeftToRight.Name <- "flowLeftToRight";
flowLeftToRight.TabIndex <- 4;
flowLeftToRight.Text <- "Flow LeftToRight";
flowLeftToRight.Checked <- flowLayoutPanel.FlowDirection = FlowDirection.
    LeftToRight;
flowLeftToRight.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection
     <- FlowDirection.LeftToRight);
//
// flowRightToLeftBtn
//
flowRightToLeftBtn.Location <- new System.Drawing.Point (155, 224);
flowRightToLeftBtn.Name <- "flowRightToLeftBtn";
flowRightToLeftBtn.TabIndex <- 5;
flowRightToLeftBtn.Text <- "Flow RightToLeft";
flowRightToLeftBtn.Checked <- flowLayoutPanel.FlowDirection =
    FlowDirection.RightToLeft;
flowRightToLeftBtn.CheckedChanged.Add (fun _ -> flowLayoutPanel.
    FlowDirection <- FlowDirection.RightToLeft);
```

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

```
//
// flowLayoutPanel
//
flowLayoutPanel.Controls.Add (button1);
flowLayoutPanel.Controls.Add (button2);
flowLayoutPanel.Controls.Add (button3);
flowLayoutPanel.Controls.Add (button4);
flowLayoutPanel.Location <- new System.Drawing.Point (47, 55);
flowLayoutPanel.BorderStyle <- BorderStyle.Fixed3D;
flowLayoutPanel.Name <- "flowLayoutPanel";
flowLayoutPanel.TabIndex <- 0;
flowLayoutPanel.WrapContents <- wrapContentsCheckBox.Checked
//
// Form1
//
let Form1 = new Form ()
Form1.ClientSize <- new System.Drawing.Size (292, 266);
Form1.Controls.Add (flowRightToLeftBtn);
Form1.Controls.Add (flowLeftToRight);
Form1.Controls.Add (flowBottomUpBtn);
Form1.Controls.Add (flowTopDownBtn);
Form1.Controls.Add (wrapContentsCheckBox);
Form1.Controls.Add (flowLayoutPanel);
Form1.Name <- "Form1";
Form1.Text <- "Form1";
Application.Run Form1
```

[5]

...

[5] Todo: `Click.Add` **expects a function** `System.EventArgs -> unit` **therfore the** `ignore` **function.**

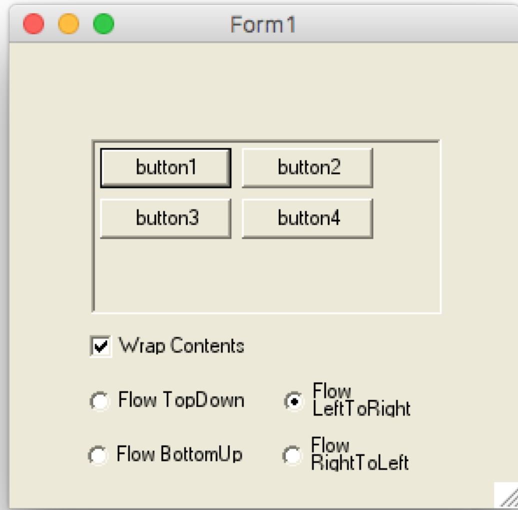


Figure 13.11: Demonstration of the `FlowLayoutPanel` panel, `CheckBox`, and `RadioButton` controls, see Listing 13.15.

Function	Description
`DataGridView`	Display data on a table.
`TextBox`	Display editable text.
`Label`	Display text.
`LinkLabel`	Display clickable text.
`ProgressBar`	Display the current progress as a bar.
`WebBrwoser`	Enable navigation of the web.
`CheckedListBox`	Display a scrollable check box list.
`ComboBox`	Display a drop-down list.
`ListBox`	Display a list of text and icons.
`PictureBox`	Display a bitmap image
`CheckBox`	Display a checkbox and a label of text.
`RadioButton`	Display an on-off radio button
`TrackBar`	Enable the user to input value by moving a cursor on a slider bar
`DateTimePicker`	Enable the user to select a date from a graphical calendar
`ColorDialogue`	Enable the user to pick a color
`FontDialog`	Enable the user to pick a font and its attributes
`OpenFileDialog`	Enable the user to navigate the file system and select a file..
`PrintDialog`	Enable the user to select a printer and its attributes.
`SaveDialog`	Enable the user to navigate the file system and specify a filename.
`MenuStrip`	Allow the user to choose from a custom menu
`Button`	Display a clickable button with text
`Tooltip`	Briefly display a pop-up window, when the user rests the pointer on the control
`SoundPlayer`	Play sounds in the `.wav` format.

Table 13.2: Some controls available in WinForms.

Function	Description
`Panel`	Groups a set of controls in a scrollable frame.
`GroupBox`	Group a set of controls in a non-scrollable frame.
`TabControl`	Group controls in tabpages, A tabpage is selected by clicking on its tab.
`SplitContainer`	Group controls into two resizable panels.
`TableLayoutPanel`	Group controls into a grid.
`FlowLayoutPanel`	Group controls into a set of flowable panels. The panels may flow horizontally or vertically as a response to window resizing.

Table 13.3: Some controls for grouping other controls.

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