Course 2DCis: 2D-Computer Graphics with C#
Chapter C2: Comments to the Draw Project

---

using namespaces

//The .NET Framework Class Library FCL contains thousands of classes.
For better orientation it is subdivided into "namespaces" each containing a subset of related classes.
Any class and its members have to be called by writing the full tree of its namespace which forces the programmer to write spaghetti-long identifiers.
With the "`using`" directive you can shorten such long identifiers and the compiler will complete the missing namespaces for You.

```
using System; //The System namespace contains fundamental classes and base classes that define all commonly-used data types, events and exceptions. It is the home of the base class of all classes System.Object and of all primitive data types such as Int32, Int16, Double, String.
```

```
using System.Drawing; //Home of the Graphics class and its drawing methods such as Point, Rectangle, Pen, Brush, DrawString, DrawLine etc.
```

```
using System.Windows.Forms; //Home of the Form class (base class of our main window Form1) and its method Application.Run.
```

Entry to start our .NET Windows program: `public class Form1 : Form`

//We derive our window `Form1` from the class `Form`, which the compiler automatically finds in the `System.Windows.Forms` namespace.

```
[STAThread] static void Main() { Application.Run( new Form1() ); }
```
//Create a single thread instance of `Form1` and ask the operating system to start it.
`[STAThread]` is important to assure the communication between the sole main thread and secondary dialog threads.
`Application.Run( new Form1() )` is a static method starting and showing an visual object derived from `System.Windows.Forms.Form` = a standard Windows-window.

```
Graphics g; //Class to store and access the current Device Context.
```

```
Point p0, p1, mid_of_p, mid_of_r; //Coordinates of the starting point and end point of a line, center of gravity and center of the bounding box.
```

```
Double perimeter, area; //Space for two double precision floating point values.
```

```
Brush redbrush = new SolidBrush(Color.Red);
```
//Create a new `Brush` object once that can be used at any call of `...OnPaint(...)`.

```
Brush graybrush = SystemBrushes.Control;
```
//Create a new `Brush` object once that can be used at any call of `...OnPaint(...)`.

```
Brush blackbrush = SystemBrushes.ControlText; //Create a pointer to an already existing Brush object (just a programming shortcut).
```

```
Pen blackpen = SystemPens.ControlText;
```
//Create a pointer to an already existing `Pen` object (just a programming shortcut).

```
Pen redpen = new Pen(Color.Red);
```
//Create a new `Pen` object once that can be used at any call of `...OnPaint(...)`.

```
Pen greenpen = new Pen(Color.Green);
```
//Create a new `Pen` object once that can be used at any call of `...OnPaint(...)`.

```
String myline; //A string containing a line of text.
```

```
UInt16 i, n; //Some variables for counting vertices. UInt16 is a primitive data type of the
.NET Framework Class Library FCL which denotes "unsigned short (16 bit) integer".
```

```
const UInt16 nMax = 100;
//Constant length of the array named polygon. Our polygon is restricted to maximal nMax vertices.
Recommended experiment: Alter this constant to 10 and observe the behaviour during drawing.
```

```
Point[] polygon = new Point[nMax];
//Definition and space for a fixed-length array of nMax vertices, each containing 2 integers: X and Y.
Sample: You can access the y-coordinate of vertex i by writing polygon[i].Y.
```

Constructor `public Form1()` inside `public class Form1` inside `public class Form1`

```
Width = 800;
//Starting width of Form1. This statement is not obligatory but the default width is rather narrow for drawing.
```

```
Height = 600;
//Starting height of Form1. This statement is not obligatory but the default height is rather narrow.
```

```
g = this.CreateGraphics();
//The current Device Context is loaded once and will be maintained for the rest of the life time of Form1.
This is the simplest method to get a Device Context once and forever, but this method will ignore any change of
the window size during run time. Professional code must reload the Device Context much more often e.g. at any
OnMouseMove and OnPaint events in order to keep in pace with possible changes of Form1 during run time.
```

Overridden event handler `OnMouseDown(MouseEventArgs e)` inside `public class Form1`

```
p0 = e.Location; //Read the current mouse position from the structure MouseEventArgs.
```

```
polygon[0] = p0; //Store the current mouse position as the first vertex of polygon.
```

```
n = 1; //Remeber that one vertex has been stored in the array polygon.
```

```
Invalidate(); //Ask the operating system to send a Paint-Message to Form1.
After receiving this message Form1 will execute our method (described below):
 protected override void OnPaint( PaintEventArgs e ).
```

Overridden event handler `OnMouseMove(MouseEventArgs e)` inside `public class Form1`

```
if ( e.Button == MouseButtons.None ) return;
//Do nothing if no mouse button has been pressed during moving.
```

```
p1 = e.Location; //Read the current mouse position from the structure MouseEventArgs.
```

```
Int32 dx = p1.X - p0.X; //dx is the horizontal distance between the new point and the last vertex.
```

```
Int32 dy = p1.Y - p0.Y; //dy is the vertical distance between the new point and the last vertex.
```

```
if ( dx*dx + dy*dy < 100 ) return;
//Pythagoras' theorem. If the distance is less than 10, ignore the new point.
```

```
if ( n >= nMax-1 ) return; //Stop anything if there are nMax-1 vertices already.
We leave one vertex space unused in order to close the polygon, when the mouse button goes up
(see below in OnMouseUp(MouseEventArgs e)).
```

```
g.DrawLine( blackpen, p0, p1 ); //Draw a straight line from the last point to the new one.
```

`polygon[n++] = p0 = p1;` //Remember the current mouse position `p1` as `p0` and write it at the next position of `polygon` and Increment the vertex counter n to the next free position.

`g.DrawString( myline, Font, graybrush , 0, Font.Height );`
//Erase the old text (if any) in the second line from the screen.
We override the old `myline` with the same text and font as before but using the background brush.

`myline = String.Format( "{0}, {1}", p1.X, p1.Y );`
//Compose a string containing two integers (they are here converted to two strings) separated by a comma and a blank. The `String.Format` method replaces `{0}` by a textual equivalent of `p1.X` and `{1}` by a textual equivalent of `p1.Y`.

`g.DrawString( myline, Font, blackbrush, 0, Font.Height );`
//Write `myline` into the second line at the left upper corner of the client area of Form1 using a black brush.

`g.DrawRectangle( blackpen, p1.X-3, p1.Y-3, 7, 7 );`
//Draw a small 7x7-rectangle around the new point with the usual standard pen and color.

Overridden event handler `OnMouseUp(MouseEventArgs e)` inside `public class Form1`
`if ( n < 2 ) return;` //Do nothing if there is one single vertex only.

`p0 = polygon[n++] = polygon[0];` //Close the polygon by copying the first vertex at the end of the array and increment the vertex counter after copying.

`perimeter = area = 0;` //Clear the accumulators to add up intermediate values.
At the end of the accumulation, these variables will contain the true values of perimeter and area.

`mid_of_p.X = mid_of_p.Y = 0;` //Clear the accumulators of the x and y values. At the end of the accumulation, these variables will contain the sum of all x- and the sum of all y-coordinates.

`Int32 xmin, xmax, ymin, ymax;` //The borders of the bounding box of the polygon:
.`xmin` = left border, `xmax` = right border, `ymin` = upper border, `ymax` = lower border.

`xmin = xmax = p0.X;` //Let us start the bounding box with left border = right border = first `x`.
`ymin = ymax = p0.Y;` //Let us start the bounding box with upper border = lower border = first `y`.

`for ( int i = 1; i < n; i++ )`
//Start at vertex no. 1 and run through all higher vertices stored in the array `polygon`.

`p1 = polygon[i];` //`p1` is an intermediary variable, just to avoid to write "`polygon[i]`".

`Double dx = p1.X - p0.X;` //`dx` is the horizontal distance between vertex `i` and its predecessor `i-1`.
`Double dy = p1.Y - p0.Y;` //`dy` is the vertical distance between vertex `i` and its predecessor `i-1`.

`Double my = (p0.Y + p1.Y) / 2.0;`
//`my` is the mean of the both `y`-values, i.e. the `y`-value of the mid point between vertex `i` and vertex `i-1`. (`dx * my` is the area of the trapezoid between the line from `i-1` to `i` and the x-axis.)

`perimeter += Math.Sqrt( dx*dx + dy*dy );`
//Pythagoras' theorem. Length of the line `i-1` to `i`. (`Math.Sqrt` returns the square root of its argument.)

`area += dx * my;` //Sum up the area of the trapezoid extending above the line from `i-1` to `i` to the x-axis.
`mid_of_p.X += p1.X;` //Sum up the x.
`mid_of_p.Y += p1.Y;` //Sum up the y.

`if ( p1.X < xmin ) xmin = p1.X;`
//If the current x is outside left of the current bounding box then expand the rectangle to the left.
`if ( p1.X > xmax ) xmax = p1.X;`
//If the current x is outside right of the current bounding box then expand the rectangle to the right.
`if ( p1.Y < ymin ) ymin = p1.Y;`
//If the current y is outside above the current bounding box then expand the rectangle upwards.
`if ( p1.Y > ymax ) ymax = p1.Y;`
//If the current y is outside below the current bounding box then expand the rectangle downwards.

```
p0 = p1; //Replace the old point by the new one.
```

```
mid_of_r.X = ( xmax + xmin ) / 2; //Horizontal mid of the bounding box.
mid_of_r.Y = ( ymax + ymin ) / 2; //Vertical mid of the bounding box.
mid_of_p.X /= n-1; //Compute the mean of the sum of the all x.
We did not sum up the first vertex, because it is identical with the last one.
mid_of_p.Y /= n-1; //Compute the mean of the sum of the all y.
We did not sum up the first vertex, because it is identical with the last one.
```

```
minmax.X = xmin-1; minmax.Width = xmax - xmin + 2; //Store xmin and xmax-xmin into the minmax rectangle structure. It looks better when we give 1 pixel more space to the left and the right.
```

```
minmax.Y = ymin-1; minmax.Height = ymax - ymin + 2; //Store ymin and ymax-ymin into the minmax rectangle structure. It looks better when we give 1 pixel more space above and below.
```

```
Invalidate(); //Ask the operating system to send a Paint-Message to Form1.
After receiving this message Form1 will execute our method (described below):
 protected override void OnPaint( PaintEventArgs e ).
```

Overridden event handler `OnPaint(PaintEventArgs e)` inside `public class Form1`

```
g.DrawString( "Press the left mouse button and move!", Font, redbrush, 0, 0 );
//This is the first line in the left upper corner of Form1.
```

```
if ( n < 2 ) return; //There is nothing to do in case of no or only one vertex.
```

```
myline = String.Format( "Perimeter= {0}, Area= {1}", (Int32)perimeter, (Int32)area );
//Prepare the text for the third line.
```

```
g.DrawString( myline, Font, blackbrush, 0, 2*Font.Height );
//Display the third line using the usual standard color.
```

```
g.DrawRectangle(greenpen, minmax ); //Draw the bounding box minmax.
```

```
Point midRect = new Point( minmax.X + minmax.Width /2, minmax.Y + minmax.Height/2);
//Compute the mid of the bounding box.
```

```
g.DrawLine( greenpen, mid_of_r.X - 4, mid_of_r.Y , mid_of_r.X + 4, mid_of_r.Y );
//Draw a small (lenght: 9 pixels) horizontal green line at he mid of the bounding box.
g.DrawLine( greenpen, mid_of_r.X , mid_of_r.Y - 4, mid_of_r.X , mid_of_r.Y + 4 );
//Draw a small (lenght: 9 pixels) vertical green line at he mid of the bounding box.
```

```
g.FillEllipse( blackbrush, mid_of_p.X-5, mid_of_p.Y-5, 11, 11 );
//Draw a small 11x11 black circle around the center of gravity.
```

```
for ( i=0; i < n-1; i++ ) g.DrawLine( blackpen, polygon[i], polygon[i+1] );
//Draw the complete closed polygon with the usual standard color.
```

```
for ( int i=0; i < n-3; i+=3 ) g.DrawBezier( redpen, polygon[i], polygon[i+1],
polygon[i+2], polygon[i+3] ); //Draw a sequence of red Bézier curves covering 4 vertices each.
When there are vertices left at the end of the polygon, they are ignored and the red Bézier curve is left open.
```

```
g.DrawRectangle( greenpen, minmax ); //Draw a green bounding box.
```

```
g.DrawLine( greenpen, mid_of_r.X - 4, mid_of_r.Y , mid_of_r.X + 4, mid_of_r.Y );
g.DrawLine( greenpen, mid_of_r.X , mid_of_r.Y - 4, mid_of_r.X , mid_of_r.Y + 4 );
//Draw a green 9-pixel-cross to mark the center of the bounding box.
```

```
g.FillEllipse( blackbrush, mid_of_p.X-5, mid_of_p.Y-5, 11, 11 ); //Draw a 11-pixel-circle to mark the center of gravity.
```