Course 2DCis: 2D-Computer Graphics with C#
Chapter C4: Comments to the Animation Project

`using` namespaces

`using System;` //Namespace of the base class of all classes "`System.Object`" and of all primitive data types such as `Int32, Int16, Single, Double, String`.

`using System.Drawing;` //Namespace of the "`Graphics`" class and its drawing methods such as `DrawString, DrawLine, DrawRectangle, FillClosedCurve` etc.

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

`using System.Collections;` //Namespace of the "`ArrayList`" class (see below the dynamic array `polygon`).

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

All objects that are used inside the function `OnTimer(...)` (see below) must be declared "`static`", because the function `OnTimer(...)` is `static`. There is only one `Timer` object for all instances of `Form1`. No matter how often You start `Form1`, any instance uses the same `Timer`. Any"`static`" object resides only once inside its class and not inside each instance of its class as nonstatic objects do. See the help text in Visual Studio .NET: `Help -> Index -> Filtered by: .NET Framework SDK -> Look for: static keyword, C#`.

`static void Main() { Application.Run(new Form1()); }` //Create an instance of `Form1` and ask the operating system to start it as main window of our program.

`static Graphics g, bitmap_g;` //Twice a Device Context = foreground and background for flicker free animation with double buffering. `g` will manage the client area of `Form1` and `bitmap_g` will manage a background mirror image of it. See: Last statements of `OnResize(...)`.

`static Bitmap bitmap;` //Background image. Will be cloned from the client area of `Form1`. See: Last statements of `OnResize(...)`.

`static Single zoom = 1.01f;` //Class to store and access the current Device Context.

`static Single cosinus = (Single)Math.Cos( Math.PI/180.0 );` //Class to store and access the current Device Context.

`static Single sinus = (Single)Math.Sin( Math.PI/180.0 );` //Class to store and access the current Device Context.

`static Brush redbrush = new SolidBrush( Color.Red );` //Create a red `Brush`.

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

`static Font arial10 = new Font( "Arial", 10 );` //Create a font.

`static Single myWidth, myHeight;` //two variables aimed to contain the current size of the client area of `Form1`.

`static PointF[] pf;` //Pointer to an array of `PointF` with unknown length. `PointF` is a structure similar to `Point` but with x and y as float.

`ArrayList polygon = new ArrayList();` //This dynamic array of variable length is aimed to contain all vertices (vertices are `Point`-Objects: see below in the functions `OnMouseDown` and `OnMouseMove`).

`Point p0 = new Point();` //Coordinates of the starting point of a line.

`Point p1 = new Point();` //Coordinates of the end point of a line (which serves as starting point of the next line).

`Timer myTimer = new Timer();` //This `Timer` sends messages at fixed time intervals to `Form1`, that trigger `Form1` to execute its `OnTimer(..)` method.

Constructor `public Form1()`

`Text = "Anim1: Draw an Endless Animation";` //Title in the blue header line of `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 for drawing.

`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 i.e. at any `OnMouseMove` and `OnPaint` events in order to keep in pace with possible changes of `Form1` during run time.

`SetStyle(ControlStyles.ResizeRedraw,true);` //Urges Form1 to redraw itself at run time by calling `protected override void OnPaint(PaintEventArgs e)` on any change of size (i.e. when the user drags a window border or edge). `OnResize(...)` calls automatically `OnPaint(...)`.

`myTimer.Tick += new EventHandler( OnTimer );` //Redirects the `Timer` messages to our `Form1`-method `OnTimer(...)`.

`myTimer.Interval = 1;` //Send the Timer messages at the shortest possible time interval (1 millisecond means as fast as possible).

Overridden event handler `OnMouseDown(MouseEventArgs e)`

`myTimer.Stop();` //Stops the Timer from sending messages. We do not want any message during drawing.

`polygon.Clear(); Invalidate();` //Delete all former vertices from the dynamic array `polygon`. Its length is now 0. `Invalidate` asks 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 )` which redraws the complete polygon.
The current effect is just to erase the client area of `Form1` because the currently empty array `polygon` is displayed. This action removes the old polygon from the screen.

`p0.X = e.X;` //Remember the current mouse position.

`p0.Y = e.Y;` //Remember the current mouse position.

`polygon.Add( p0 );` //Write the first vertex at position 0 into the dynamic array `polygon`.

Overridden event handler `OnMouseMove(MouseEventArgs e)`

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

`p1.X = e.X;` //Take the current mouse position.

`p1.Y = e.Y;` //Take the current mouse position.

`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 then `10`, ignore the new point.

`g.DrawLine( blackpen, p0, p1 );` //Draw a line from the old vertex to the current vertex.

`polygon.Add( p1 );` //Elongate the dynamic array `polygon` by adding space for an object of type `Point` and store the current vertex into this empty space.

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

Overridden event handler `OnMouseUp(MouseEventArgs e)`

`if ( polygon.Count < 2 ) return;` //Do nothing if there is one vertex only inside the polygon.

`pf = new PointF[polygon.Count];` //Allocate memory for the array `pf`. The lenght of `pf` is identical with the length of the dynamic array `polygon`.

`for ( Int32 i=0; i < polygon.Count; i++ ) pf[i] = (Point)polygon[i];` //Go through all vertices which a currently stored in the dynamic array `polygon`, and copy them into the new array `pf`. The explicit type cast operator `(Point)` is necessary to convert the type `object` to the type `Point`. The conversion from type `Point` to type `PointF` needs no explicit type cast operator. Writing the complete casting results in a confusing (but correct) code: `pf[i] = (PointF)((Point)polygon[i]);`

`myTimer.Start();` //The `Timer` is now allowed to send its messages. The animation starts now immediately after drawing. This statement corresponds to the `myTimer.Stop();`-statement inside the `OnMouseDown(...)` function.

Overridden event handler `OnPaint(PaintEventArgs e)`

`e.Graphics.DrawString("Press the left mouse button and move!", arial10, redbrush, 300, 0 );` //Display the text somewhere in the middle of the first line of `Form1`.

Overridden event handler `OnResize(PaintEventArgs e)`

This method is called whenever the user drags one of the corners or borders of `Form1`. After its execution the OnPaint(...) method is invoked because of the `SetStyle(ControlStyles.ResizeRedraw,true);` -statement in the constructor `public Form1()`.

`g = this.CreateGraphics();` //Update the current `Graphics`-object in order to inform it that there are new dimensions of `Form1`.

`g.Clear(SystemColors.Control);` //Erase everything (using the normal background color of a `Form`) from the client area of `Form1` .

`myWidth = ClientRectangle.Width;` //Remember the current width of the client area of `Form1` for futher use inside the OnTimer(...) function.

`myHeight = ClientRectangle.Height;` //Remember the current height of the client area of `Form1` for futher use inside the OnTimer(...) function.

`if ( bitmap != null ) bitmap.Dispose();` //Throw away any old background image.

`bitmap = new Bitmap( myWidth, myHeight );` //Background image: Clone it from the `Form1` client area. See: Last statements of `OnTimer(...)`.

`if ( bitmap_g != null ) bitmap_g.Dispose();` //Throw away any old background `Graphics`-object.

`bitmap_g = Graphics.FromImage( bitmap );` //Create a `Graphics`-object around the background image. See: Last statements of `OnTimer(...)`.

Event handler `protected static void OnTimer( Object myObject, EventArgs myEventArgs )`

`Single x, y, xmin, ymin, xmax, ymax, xmid, ymid;` //Some local variables.

`xmin = xmax = pf[0].X;` //Set the surrounding rectangle to the first vertex.

`ymin = ymax = pf[0].Y;` /Set the surrounding rectangle to the first vertex.

`for ( Int32 i=0; i < pf.Length; i++ )` //Go through all vertices.

`x = pf[i].X;` //Take the x-value from the vertex no. `i` .

`y = pf[i].Y;` //Take the y-value from the vertex no. `i` .

`if ( x < xmin ) xmin = x;` //If x is left oft the current left border, shift the left border to the left.

`if ( x > xmax ) xmax = x;` //If x is right oft the current right border, shift the right border to the right.

`if ( y < ymin ) ymin = y;` //If y is obove oft the current upper border, shift the upper border up.

`if ( y > ymax ) ymax = y;` //If y is below oft the current lower border, shift the lower border down.

`xmid = (xmin+xmax) / 2;` //Mid point of the surrounding rectangle.

`ymid = (ymin+ymax) / 2;` //Mid point of the surrounding rectangle.

`if ( xmin < 0 || ymin < 0 || xmax > myWidth || ymax > myHeight )` //Does the surrounding rectangle cross one of the borders of `Form1` ?

`//g.Clear(SystemColors.Control);` //If it crosses, erase every thing using the normal background color of a `Form`. This statement is not necessary when we use double buffering.

`g.DrawString("Press the left mouse button and move!", arial10, redbrush, 320, 0 );` //Redraw the first line of text.

`zoom = 0.99f;` //Set the zoom factor to 99 %. From now on the animation shrinks (zooms down).

`if ( xmax - xmin < 50 || ymax - ymin < 50 )` //Is the surrounding rectangle smaller than 50 pixels width or 50 pixels height ?

`//g.Clear(SystemColors.Control);` //If it is too small, erase eyerything using the normal background color of a `Form`. This statement is not necessary when we use double buffering.

`zoom = 1.01f;` //Set the zoom factor to101 %. From now on the animation pumps up (zooms up).

`for ( Int32 i=0; i < pf.Length; i++ )` //Go through all vertices.

`x = pf[i].X - xmid;` //Shift the mid point of the polygon to coordinate origin 0/0 at the upper left corner of `Form1`.

`y = pf[i].Y - ymid;` //Shift the mid point of the polygon to coordinate origin 0/0 at the upper left corner of `Form1`.

`x *= zoom;` //Multipy x by the zoom factor.

`y *= zoom;` //Multipy y by the zoom factor.

`Single xx = x*cosinus - y*sinus;` //Rotate around the origin.

`Single yy = x*sinus + y*cosinus;` //Rotate around the origin.

`pf[i].X = xx + xmid;` //Shift the polygon back from the origin to its normal position inside `Form1` and store the result back to the array `pf`.

`pf[i].Y = yy + ymid;` //Shift the polygon back from the origin to its normal position inside `Form1` and store the result back to the array `pf`.

`bitmap_g.FillRectangle( SystemBrushes.Control, 0, 0, myWidth, myHeight );` //Invisible erasure of the background image.

`bitmap_g.DrawLines( blackpen, pf );` //Invisible drawing of the complete polygon into the background image using a black pen . The Graphics-function `DrawLines` determines automatically the no. of vertices and connects them with lines. It does not close a polyline and does not fill it with color even when the polyline is a closed polgon.

`g.DrawImage( bitmap, 0, 0 );` //En bloc copy of the background image to the foreground = last step of double buffering.