Course IPCis: Image Processing with C#
Chapter C2: The Histo Project

An empty window

Guidance for **Visual Studio 2008**:

1) Main menu after start of VS 2008: File → New Project... →
Visual Studio installed templates: Windows Forms Application
Name: histo1 → Location: C:\temp → Create directory for solution: switch off → OK
Form1.cs[Design] appears.
2) Two superfluous files must be deleted: Form1.Designer.cs and Program.cs.
You reach these files via the Solution Explorer - histo1-window:
Click the plus-sign in front of branch histo1 and the plus-sign in front of branch Form1.cs.
Right-click the branch Program.cs. A context menu opens. Click Delete. A message box appears:
'Program.cs' will be deleted permanently. Quit with OK.
Right-click the branch Form1.Designer.cs and delete this file too.
3) Right-click the gray window Form1. A small context menu opens. Click View Code.
You see now the pre programmed code of Form1.cs. Erase this code completely.
4) Write the following three lines into the empty Form1.cs:

```
public class Form1 : System.Windows.Forms.Form
{ static void Main() { System.Windows.Forms.Application.Run( new Form1() ); }
}
```

5) Click Debug in the main menu of VS 20085.
A sub menu opens. Click Start Without Debugging Ctrl F5.

Read and display an image

Erase the three lines of code for the empty window in Form1.cs and replace them by:

```
using System;
using System.Drawing;
using System.Drawing.Imaging;
using System.Windows.Forms;

public class Form1 : Form
{ static void Main() { Application.Run( new Form1() );  }
  Brush bbrush = SystemBrushes.ControlText;
  Brush wbrush = new SolidBrush( Color.White );
  Pen   bpen   = SystemPens.ControlText;
  Pen   rpen   = new Pen( Color.Red );
  Bitmap bmp, bmp_binary, bmp_histo;
  BitmapData binaryData; //for Versions 2 and 3
  Byte[,] grayarray;     //2D-Byte-Array
  Int32[] Histogram = new Int32[256];
  Rectangle histo_r = new Rectangle( 0,0,257,101 );
  Graphics g, g_histo;
  public Form1()
  { MenuItem miRead = new MenuItem( "&Read", new EventHandler( MenuFileRead ) );
    MenuItem miExit = new MenuItem( "&Exit", new EventHandler( MenuFileExit ) );
    MenuItem miFile = new MenuItem( "&File", new MenuItem[] { miRead, miExit } );
    Menu = new System.Windows.Forms.MainMenu( new MenuItem[] { miFile } );
    Text = "Histo1";
    SetStyle( ControlStyles.ResizeRedraw, true );
    Width  = 800;
    Height = 600;
  }
  void MenuFileRead( object obj, EventArgs ea )
  { OpenFileDialog dlg = new OpenFileDialog();
    if ( dlg.ShowDialog() != DialogResult.OK ) return;
    try
    { Cursor.Current = Cursors.WaitCursor;
      bmp = (Bitmap)Image.FromFile( dlg.FileName );
      Cursor.Current = Cursors.Arrow;
      Invalidate();
    } catch {}
  }
  void MenuFileExit( object obj, EventArgs ea )
  { Application.Exit(); }
  protected override void OnPaint( PaintEventArgs e )
  { if ( bmp == null )
    { e.Graphics.DrawString( "Open an Image File !", Font, bbrush, 0, 0 ); return; }
    e.Graphics.DrawImage( bmp, ClientRectangle );
  }
}
```

Click Debug → `Start Without Debugging Ctrl F5`. Try to read different image formats: `BMP, ICO, GIF, JPG, PNG, TIFF`.

Code for histogram

Finish `histo1`.

Write a new line of code to `void MenuFileRead( object obj, EventArgs ea )` below the line `bmp = (Bitmap)Image.FromFile( dlg.FileName );` but above the line `Cursor.Current = Cursors.Arrow;`:

```
    GenerateTheHistogram();
```

Write a new function between the existing functions `void MenuFileRead(...)` and `void MenuFileExit(...)`:

```
  void GenerateTheHistogram()
  { if ( bmp == null ) return;
    bmp_binary = new Bitmap( bmp.Width, bmp.Height, PixelFormat.Format32bppRgb  ); //Version 1
    //bmp_binary = new Bitmap( bmp.Width, bmp.Height, PixelFormat.Format32bppRgb  ); //Version 2
    //bmp_binary = new Bitmap( bmp.Width, bmp.Height, PixelFormat.Format1bppIndexed ); //Version 3
    grayarray = new Byte[bmp.Height, bmp.Width];
    Color color;
    for ( Int32 y=0; y < bmp.Height; y++ )
      for ( Int32 x=0; x < bmp.Width; x++ )
      { color = bmp.GetPixel( x, y );
        Int32 gray = ( color.R + color.G + color.B ) / 3;
        grayarray[y, x] = (Byte)gray;
        Histogram[gray]++;
      }
    Int32 hmax = 0;
    for ( Int32 i=0; i < 256; i++ )
      if ( Histogram[i] > hmax ) hmax = Histogram[i];
    for ( Int32 i=0; i < 256; i++ )
      Histogram[i] = (100*Histogram[i]) / hmax;
    bmp_histo = new Bitmap( histo_r.Width, histo_r.Height, PixelFormat.Format32bppRgb );
    g_histo = Graphics.FromImage( bmp_histo );
    g_histo.FillRectangle( wbrush, 0,0,256,100 );
    g_histo.DrawString( "click here and move !", Font, bbrush, 1, 1 );
    for ( Int32 i=0; i < 256; i++ ) g_histo.DrawLine( bpen, i, 100, i, 100 - Histogram[i]);
    g_histo.DrawRectangle( rpen, 0,0,256,100 );
  }
```

Write the following additional lines into the function `protected override void OnPaint( PaintEventArgs e )` below the existing line `e.Graphics.DrawImage( bmp, ClientRectangle );`:

```
    histo_r.X  = ClientRectangle.Width  - histo_r.Width  - 10;
    histo_r.Y  = ClientRectangle.Height - histo_r.Height - 10;
    e.Graphics.DrawImage( bmp_histo, histo_r );
```

Click Debug → `Start Without Debugging Ctrl F5`. Try out the histogram.

Mouse events and threshold

Finish `histo1`.

Write two new event handler functions for mouse events below the existing `void MenuFileExit(...)` but above `protected override void OnPaint(...)`:

```
  protected override void OnMouseMove( MouseEventArgs e )
  { if ( e.Button == MouseButtons.None ) return;
    if ( !histo_r.Contains( e.X, e.Y ) ) return;
    if ( bmp == null ) return;
    Byte threshold = (Byte)(e.X - histo_r.X);
    //Version 1 (no pointers but slow)***************************************
    for ( Int32 y=0; y < bmp.Height; y++ )
      for ( Int32 x=0; x < bmp.Width; x++ )
      { if ( grayarray[y ,x] > threshold )
               bmp_binary.SetPixel( x, y, Color.White );
        else bmp_binary.SetPixel( x, y, Color.Black );
      }
    //End of Version 1 ******************************************************
    g = CreateGraphics();
    g.DrawImage( bmp_binary, ClientRectangle );
    g.DrawImage( bmp_histo, histo_r );
    g.DrawLine( rpen, histo_r.X+threshold, histo_r.Y,
                      histo_r.X+threshold, histo_r.Y+histo_r.Height-1 );
  }

  protected override void OnMouseUp(MouseEventArgs e)
  { Invalidate(); }
```

Click `Debug` → `Start Without Debugging Ctrl F5`. Try out thresholding + binarisation.

Fast pixel access with pointers

Finish `histo1`.

The binary images that appear on mouse movements appear jerky and with some delay. It's possible to make it nearly as fast as C++ code using pointers for pixel access, instead of using the slow commands "`SetPixel( x, y, Color.White );`" and "`SetPixel( x, y, Color.Black );`"

At first we have to ask the compiler to allow pointers which are normally forbidden in C# for safety reasons.
In the main menu of VisualStudio 2005 click `Project` → `histo1 Properties` → `Build` and set the 3. flag "`Allow unsafe code`" to `True`. → Quit the `histo1 Property Pages` by clicking the tab `Form1.cs`.
In the function `protected override void OnMouseMove(...)` comment out everything between the comment lines with the comment clauses "`/*`" and "`*/`". It should look like this:

```
/*
//Version 1 (no pointers but slow)***************************************
```
   Two `for`-loops consisting of 6 lines incl. the braces.
```
//End of Version 1 ******************************************************
*/
```

Replace the crippled code by:

```
    //Version 2 (fast pointers creating a memory wasting 32-bit binary image)*
    unsafe
    { //lock bmp_binary from being shifted in memory by the garbage collector
      binaryData = bmp_binary.LockBits( new Rectangle( 0,0,bmp.Width,bmp.Height ),
                              ImageLockMode.WriteOnly, PixelFormat.Format32bppRgb );
      //Byte* p1fix, p1run =  fixed + running pointers to grayarray  = input image
      UInt32* p2fix, p2run; //fixed + running pointers to binaryData = output image
      fixed ( Byte* p1fix = grayarray )    // lock grayarray in memory
      { Byte* p1run = p1fix;               // running pointer to grayarray
        p2fix = (UInt32*)binaryData.Scan0; // pointer to output image
        for ( int y=0; y < bmp.Height; y++ )
        { p2run = p2fix + y * bmp.Width; //p2run points to first byte in row y
          for ( int x=0; x < bmp.Width; x++ )
          { if ( *p1run++ > threshold ) *p2run++ = 0xFFFFFF; // white
            else                        *p2run++ = 0;        // black
          } // end of for x
        }   // end of for y
      }     // end of fixed, end of p1fix, unlock grayarray
      bmp_binary.UnlockBits( binaryData ); //end of p2fix, unlock bmp_binary
    } // end of unsafe
    //End of Version 2 ******************************************************
```

Click `Debug` → `Start Without Debugging Ctrl F5`. Try out the new binarisation. It is much faster than the old one. When the images are small (as Madonna.bmp) the binarisation follows the mouse without delay. You can further speed it up by taking out the first and the last command from the `unsafe`-block and by transferring both together at the end of `void GenerateTheHistogram()`. These commands are now executed only once (instead of being executed at any `MouseMove`). Simply write at end of `void GenerateTheHistogram()`:

```
      //lock bmp_binary from being shifted in memory by the garbage collector
      binaryData = bmp_binary.LockBits( new Rectangle( 0,0,bmp.Width,bmp.Height ),
                              ImageLockMode.WriteOnly, PixelFormat.Format32bppRgb );
      bmp_binary.UnlockBits( binaryData ); // end of p2fix, unlock bmp_binary
```

It is completely illogical, but it works until the garbage collector re-arranges the memory.

Slim 1-bit-pixel binary images

Finish `histo1`.
It's a enormous waste of memory to code a binary black or white pixel with 32 bits.
The class `Bitmap` provides an economic 1-bit image format `Format1bppIndexed`.
The problem is, that the class doesn't have a `GetPixel`- neither a `SetPixel`-access to those 1-bit pixels.
There is no other way as to read 8 pixels at once with the help of a `Byte`-pointer and to mask out the aimed bit with a suitable `Byte`-mask.

In the head of `Form1` below the line `Graphics g, g_histo;` complement the following two `Byte`-arrays:

```
  Byte[]  ORmask = { 128,  64,  32,  16,   8,   4,   2,   1 };// 1 bit  each //for Version 3
  Byte[] ANDmask = { 127, 191, 223, 239, 247, 251, 253, 254 };// 7 bits each //for Version 3
```

Comment out in `void GenerateTheHistogram()`) both `bmp_binary = ...`-commands `//Version 1` and `//Version 2`. Activate command `//Version 3`. It should look like this:

```
    //bmp_binary = new Bitmap( bmp.Width, bmp.Height, PixelFormat.Format32bppRgb  ); //Version 1
    //bmp_binary = new Bitmap( bmp.Width, bmp.Height, PixelFormat.Format32bppRgb  ); //Version 2
    bmp_binary = new Bitmap( bmp.Width, bmp.Height, PixelFormat.Format1bppIndexed ); //Version 3
```

Comment out in `protected override void OnMouseMove(...)` all lines between the comment lines using two comment clauses "`/*`" and "`*/`". It should look like this:

```
/*
//Version 2 (fast pointers creating a memory wasting 32-bit binary image)*
```
   20 lines
```
//End of Version 2 ************************************************************
*/
```

Replace the crippled code by:

```
    //Version 3 (fast pointers creating a 1-bit binary image)****************
    unsafe
    { //lock bmp_binary from being shifted in memory by the garbage collector
      binaryData = bmp_binary.LockBits( new Rectangle( 0,0,bmp.Width,bmp.Height ),
                              ImageLockMode.WriteOnly, PixelFormat.Format1bppIndexed );
      Byte* p2fix, p2row, p2run;         // pointers to binaryData = output image
      fixed ( Byte* p1fix = grayarray )  // lock grayarray = input image in memory
      { Byte* p1run = p1fix;             // running pointer to grayarray
        p2fix = (Byte*)binaryData.Scan0; // pointer to output image
        for ( int y=0; y < bmp.Height; y++ )
        { p2row = p2fix + y * binaryData.Stride; //p2row points to first byte in row y
          for ( int x=0; x < bmp.Width; x++ )
          { p2run = p2row + x / 8;
            if ( *p1run++ > threshold ) *p2run |=  ORmask[ x % 8 ];//set    1 bit
            else                        *p2run &= ANDmask[ x % 8 ];//remove 1 bit
          } // end of for x
        }   // end of for y
      }     // end of fixed, end of p1fix, unlock grayarray
      bmp_binary.UnlockBits( binaryData ); // end of p2fix, unlock bmp_binary
    } // end of unsafe
    //End of Version 3 ******************************************************
```

Click `Debug` → `Start Without Debugging Ctrl F5`.
As in Version 2 both `BinaryData`-commands can be executed once for ever outside the `unsafe`-block. See the end of the previous paragraph.

Faster with a DualCore CPU and two threads

Programming images with threads makes little sense when there is just one CPU. Two threads have to be executed one after the other and there will be nearly no acceleration. But using the new DualCore-CPUs from Intel **www.intel.com** and AMD **www.amd.com** two threads can run in parallel. When You slice the image horizontally in an upper and a lower part and start a thread per slice You will obtain double speed.
Of course the same code still runs with old SingleCore-CPUs but in the old slow mode.
Try out the threading-version (95% identical with version 3 = pointers and output of 1-bit-images): **CIPCisHisto_Code_Thread.htm**.

Sample images

The program should read and display a broad range of image formats: `BMP, ICO, GIF, JPG, PNG, TIFF`.
If You use an old 8-bit graphics board or if You set Your desktop to 256 colors, the colors may look strange.
If You find no sample images on Your hard disk, use the following ones:

Download: Butterfly.bmp 217 kB 24bit-true-color image
Download: Madonna.bmp 18 kB 8bit-gray-value image
Download: Lena256.bmp 66 kB 8bit-gray-value image
Download: Lena512.bmp 258 kB 8bit-gray-value image
Download: Angiography.bmp 66 kB 8bit-gray-value image

Exercises

Click `Help` in the main menu of Visual Studio. Click the sub-menu `Index`.
Choose `Filtered by: .NET Framework`. Type into the `Look for:`-field the following key words:
`PixelFormat enumeration`
`Bitmap → Bitmap class → constructor`, try to find the `C#`-constructor
`public Bitmap(int, int, PixelFormat);`
`arrays, multidimensional → C# Programmer's Reference`
`Color structure → all members`, try to find the `Public Properties: Color.R, Color.G, Color.B`.

Change position and height of `histo_r`. Adapt the height norm of the histogram.
Program three histogram images separately for the three colors red, green and blue.
Change the program to see the original image below the threshold and just change the pixels above the threshold to white. Use the method `Color.FromArgb(int, int, int);`
Program the inverse case, where the pixels below the thresholds appear black,
the rest conserving their original values.
Invent and try out new variants of the program in form of new projects `histo2, histo3`.