Courses IPCx, C3: Filter, Code Comments

Copyright © by V. Miszalok, last update: 24-03-2002

In `filter1Doc.h` in front of `class CFilter1Doc : public CDocument`

`#include < vector >` //declares the dynamic arrays of the Standard Template Library STL.
`#include < math.h >` //declares the function `_hypot` which is used for the gradient filter.
The blanks inside the < > clauses may be suppressed. They are just necessary in a HTML-Document otherwise HTML treats the words `vector` and `math.h` as HTML-Tag.

`BITMAPFILEHEADER FH;` //structure of overall length = 14 bytes containing 5 variables (see MSDN and C6_Bitmap_FAQs)

`BYTE IBytes[1200];` //untyped space for 1200 bytes for BitmapInfoHeader+Palette. You don't know the types of BitmapInfoHeaders you will read and you don't know if there will be palettes or not. You just reserve 1200 bytes for the worst case: BITMAPV5HEADER (=136 byte) plus 256 palette entries (=1024 bytes). 1160 bytes are wasted in case of traditional 24-Bit-Bitmaps but you can probably afford that.

`BITMAPINFOHEADER* pIH;` //typed pointer allows the access to biSize, biWidth etc.

`BITMAPINFO* pI;` //This typed pointer is needed by function `StretchDIBits`. Structure BITMAPINFO (see MSDN and C6_Bitmap_FAQs) is longer than BITMAPINFOHEADER but it contains in its first part the structure BITMAPINFOHEADER completely . It allows access to both the BitmapInfoHeader and (if `biClrUsed > 0`) the palette. If `biClrUsed > 0` then `StretchDIBits` automatically accesses and uses the palette with the help of this pointer.

`std::vector< BYTE > Original ;` //name of a dyn. array of bytes aimed to store the original pixels
`std::vector< BYTE > Noise ;` //name of a dyn. array of bytes aimed to store the noisy image
`std::vector< BYTE > Lowpass ;` //name of a dyn. array of bytes aimed to store the blurred image
`std::vector< BYTE > HighVertical;` //name of a dyn. array of bytes aimed to store the image with the vertical edges
`std::vector< BYTE > HighHorizont;` //name of a dyn. array of bytes aimed to store the image with the horizontal edges
`std::vector< BYTE > HighGradient;` //name of a dyn. array of bytes aimed to store the length of the gradient of vertical and horizontal edges

---

In `filter1Doc.cpp` inside the constructor `CFilter1Doc()` you have to initialize 3 variables:

`memset( IBytes, 0, sizeof(IBytes) );` // clear the 1200 untyped bytes in order to mark the absence of an image. Function `CFilter1View::OnDraw()` reads from here if you already loaded an image or not.

`pIH = (BITMAPINFOHEADER*) IBytes;` //The typed pointer points now to the the head of IBytes[1200]

`pI = (BITMAPINFO*) IBytes;` //The typed pointer points now to the the head of IBytes[1200] as `pIH` does.

In `filter1Doc.cpp` inside `Serialize(CArchive& ar)` inside the `else` clause: Code for reading an image and for computing the noisy and the filtered images

---

```
ar.Read( & FH, sizeof(BITMAPFILEHEADER) );
```
//Read 14 bytes from the harddisk.

```
if ( FH.bfType != 'MB') { forget_it(); return;
```
//The first 2 bytes form the reversed string of BM.

```
if ( FH.bfSize <= 54 ) { forget_it(); return;
```
//A bitmap file contains at least 55 bytes.

```
if ( FH.bfOffBits < 54 ) { forget_it(); return;
```
//The shortest possible headers need 54 bytes.

```
int nBytesInfo = FH.bfOffBits - sizeof(BITMAPFILEHEADER);
```
//That is the space left for the BitmapInfoHeader and palette.

```
int nBytesPixel = FH.bfSize - FH.bfOffBits;
```
//That is the space for all the pixels.

```
ar.Read( IBytes, nBytesInfo );
```
//Read BitmapInfoHeader+palette from harddisk

```
if ( !(pIH->biBitCount == 8 ) { forget_it(); return; }
```
//Ignore all 1-, 4-, 16-, 24- and 32-bit bitmaps.

```
Original .resize( nBytesPixel );
```
//Reserve memory space for the original image.
```
Noise .resize( nBytesPixel );
```
//Reserve memory space for the noisy image.
```
Lowpass .resize( nBytesPixel );
```
//Reserve memory space for the blurred image.
```
HighVertical.resize( nBytesPixel );
```
//Reserve memory space for the vertical edge image.
```
HighHorizont.resize( nBytesPixel );
```
//Reserve memory space for the horizontal edge image.
```
HighGradient.resize( nBytesPixel );
```
//Reserve memory space for the gradient image.
Do not confound the array names `Noise` and `Lowpass` with the integers `noise` and `lowpass`, that will be introduced below. C++ carefully distinguishes between these names.

```
std::vector< BYTE >::iterator p, p1, p2, p3, p4;
```
//declaration of 5 special pointers for dyn. byte arrays. Such special pointers are called iterators.

```
ar.Read( &Original.front(), nBytesPixel );
```
//Read all original pixels from the harddisk into the main memory starting at the first adress of the dynamic byte array named `Original`.

Destroy the original image by artificial NOISE

`#define AMPLITUDE 128;` //precompiler statement that replaces in the following code lines the string `AMPLITUDE` by the string `128`. Try out other values between 2 and 1000.

`for ( p1=Original.begin(), p2=Noise.begin(); p1 < Original.end(); p1++, p2++ )` //for loop. It begins by setting two pointers `p1` and `p2` to their initial values at the head of two arrays. It checks after each loop if the first pointer reached the end of its array. If not, it increments both pointers. This loop will take each original pixel from position p1, extract the grey value, randomize it and write the value back to position p2. In the project instructions this line was broken down into three lines because it is too long to be printed wthout a line feed. Logically it remains one line. Line feeds and blanks have no significance in C.

`int noise = *p1 + AMPLITUDE/2 - rand()%AMPLITUDE;` //rand() is a random integer between 0 to the hithest possible integer. rand() modulo AMPLITUDE limits it to the range between 0 and 127. Subtracting the result from 64 will result in a range from -63 to +63. The grey values of the original pixels are in the range between 0 and 255. Together with the random offset the resulting noise value is somewhere in the range between - 63 and 318.

`if ( noise < 0 ) *p2 = 0;` //When it is negative, we clip it to 0.

`else if ( noise > 255 ) *p2 = 255;` //When it is over 255, we clip it to 255.

`else *p2 = BYTE(noise);` //Otherwise we take it as it is, shorten the integer `noise` to a byte and write it to the output image `Noise`.

Destroy the original image by BLURRING

`#define LOWPASSSIZE 11` //precompiler statement that replaces in the following code lines the string `LOWPASSSIZE` by the string `11`. Try out other values between 3 and 49. LOWPASSSIZE defines the square size of the convolution kernel or in other words the lateral length of the quadratic filter matrix. The blurring effect and the computing time increases rapidly with this value.

`#define ADDMIDWEIGHT 0` //precompiler statement that replaces in the following code lines the string `ADDMIDWEIGHT` by the string `0`. Try out other values between 1 and 1000. ADDMIDWEIGHT is the abbreviation of "Adding a Middle Weight to the normal weight of 1 to the central element of the filter matrix". It changes the standard average filter to a mid-weighted lowpass filter. This allows a subtle control of the blurring effect. When this value increases, the blurring effect decreases. When you want to see heavy changes, `ADDMIDWEIGHT` must have a value in the range of the no of elements of the filer matrix, i.e. `LOWPASSSIZE*LOWPASSSIZE.` It does not influence the computing time.

`memset( &Lowpass.front(), 0, nBytesPixel );` //Preset zero into the whole memory space of Lowpass. The Lowpass image will be smaller than the original image. It looses on left, right, top and bottom LOWPASSIZE/2 columns and rows. This border space should be black and it is easier to black out the whole image than just its borders.

`int xSize = pIH->biWidth;` // `xSize` = image width = no. of columns
`int ySize = pIH->biHeight;`// `ySize` = image height = no. of rows
//The following 4 code loops are easier to read and understand when we use `xSize` and `ySize` instead of the BITMAPINFOHEADER variables.

`int x, y, xx, yy, sum, d = LOWPASSSIZE/2;` //several integer variables with no initial values except `d` which is set to the half size of the filter matris.

`int norm = (2*d+1)*(2*d+1)+ADDMIDWEIGHT;` //This is the normalisation divisor. The lateral length of the quadratic filter matrix must always be set to an odd integer regardless if LOWPASSSIZE is even. The normalization divisor must contain all weights i.e. the no of elements of weight 1 plus the additional middle weight.

`for ( y = d; y < ySize - d; y++ )` //Take all rows except the first d and the last d lines.

`p1 = Original.begin() + y * xSize;` //pointer to the first pixel in the row
`p = Lowpass .begin() + y * xSize;` //pointer to the first pixel in the corresponding row of the output image.

`for ( x = d; x < xSize - d; x++ )` //Take alle pixels in the row ecept the first `d` and the last `d` pixels.

`p2 = p1 + x;` //Move the pointer to pixel no. x. This is the pointer to the middle of the filter kernel. The quadratic kernel matrix extends from here symmetrically in all 4 directions. From here starts the convolution formula.

`sum = ADDMIDWEIGHT * *p2;` //Initialize the sum to the middle weight and multiply by the grayvalue of the current original pixel.

`for ( yy = -d; yy <= d; yy++ )` //Take all rows between -d above and +d below the current row. Theses are the rows that the filter kernel covers in this moment.

`p3 = p2 + yy * xSize;` //set a pointer to the the column -d left of the current column. In the moment the left border of the kernel matrix is in this column.

`for ( xx = -d; xx < d; xx++ )` //Take all columns between -d to the left and +d to the right of the current column.

`sum += *p4;` //Add the greyvalues and keep in mind that this implies that all these pixels including the central one are here assumed to have weight 1.0.

`*(p+x) = (BYTE)( sum / norm );` //After the complete summation of all pixels under the filter kernel, divide the sum by the norm (=sum of weights) and store the result to the corresponding place into the lowpass image. The inner two for-loops and the convolution formula end here.

Three slightly different highpass filters

```
memset( &HighVertical.front(), 0, nBytesPixel );
memset( &HighHorizont.front(), 0, nBytesPixel );
memset( &HighGradient.front(), 0, nBytesPixel );
```
//These three statements clean up the three output images. This prevents any rubbish from being displayed in the outmost rows and colums of the three images. Nothing important will happen when you delete these statements .

```
for ( y = 1; y < ySize - 1; y++ )
```
//Take alle rows except the first and the last one.

```
p1 = Original.begin() + y * xSize;
p2 = HighVertical.begin() + y * xSize;
p3 = HighHorizont.begin() + y * xSize;
p4 = HighGradient.begin() + y * xSize;
```
//4 pointers are set to the first pixels of 4 corresponding rows.

```
for ( x = 1; x < xSize - 1; x++ );
```
//Take all pixels in the row except the first and the last one.

```
p = p1 + x;
```
//pointer to the current original pixel.

```
int sumleft = *(p-1) + *(p-1-xSize) + *(p-1+xSize);
```
//Sum up the grayvalues of the left, the upper left and the lower left neighbours.

```
int sumright = *(p+1) + *(p+1-xSize) + *(p+1+xSize);
```
//Sum up the grayvalues of the right, the upper right and the lower right neighbours.

```
int sumtop = *(p-xSize) + *(p-xSize-1) + *(p-xSize+1);
```
//Sum up the grayvalues of the upper, the upper left and the upper right neighbours.

```
int sumbottom = *(p+xSize) + *(p+xSize-1) + *(p+xSize+1);
```
//Sum up the grayvalues of the lower, the lower left and the lower right neighbours.

```
int diff = sumleft - sumright;
if ( diff < 0 ) diff *= -1;
if ( diff > 255 ) *(p2+x) = 255; else *(p2+x) = BYTE(diff);
```
//Compute the absolute vertical difference, limit it to max 255 and write it to the first output image.

```
diff = sumtop - sumbottom;
if ( diff < 0 ) diff *= -1;
if ( diff > 255 ) *(p3+x) = 255; else *(p3+x) = BYTE(diff);
```
//Compute the absolute horizontal difference, limit it to max 255 and write it to the second output image.

```
double diff_v = sumleft - sumright;
double diff_h = sumtop - sumbottom;
diff = int( 0.5 + _hypot( diff_v, diff_h ) );
```
//The compiler transforms `double` and `float` variables to `integer` by a brutal cut behind the decimal point. Example: `1.99` comes out to be `1.` In order to obtain a rounded result you have to add `0.5` to positive figures (or subtract `0.5` from negative figures). Example: `int(0.5 + 1.99) == int(2.49) == 2.`

```
if ( diff > 255 ) *(p4+x) = 255; else *(p4+x) = BYTE(diff);
```
//The 4 statements do the following: They compute the combined length with Pythagoras' theorem, round the result (which is always positive) to an integer, limit it to max 255 and write it to the third output image.