1. Getting started with the DIBLook framework

1.1. Introduction

The purpose of this first laboratory is to acquaint the students with the framework application which will be used in the practical works related to the Image Processing lecture.

The background knowledge necessary to successfully complete the image processing laboratory are:

- **Compulsory**: *C, Computer Programming, Data Structures and Algorithms.*
- **Optional (recommended)**: *C++, Visual C++ 9.0 (Visual Studio 2008), Object Oriented Methods, Fundamental Algorithms, Programming Techniques, Linear Algebra and Geometry, Discrete Mathematics, Numerical Calculus, Special Mathematics*

1.2. Overview of the DIBLook framework

The framework which will be used for the implementation and testing of the learned image processing algorithms is based on the *DIBLook* sample application available in MSDN. A modified version of this application (for easier usage) is available on the Image Processing Laboratory's web page.

DIBLook is a *MDI (Multiple Document Interface)* application [1] complying the Document-View Architecture [2], [3] (Fig. 1.1) of the *MFC* (*Microsoft Foundation Class Library*) [4].

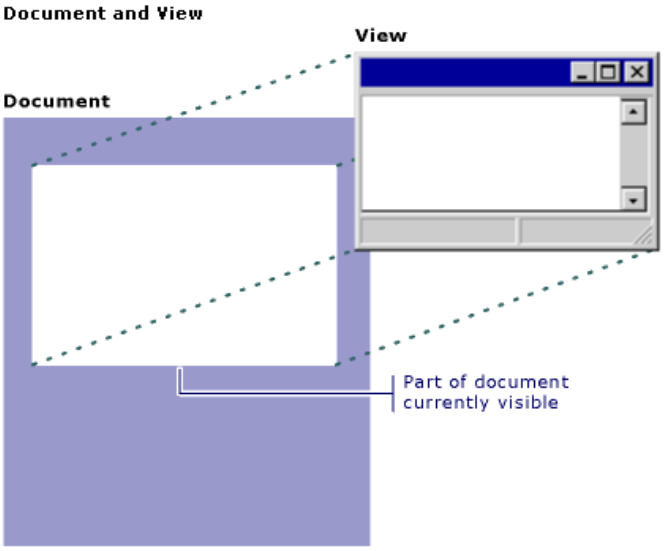

Fig. 1.1. The Document View architecture [3]

The original *DIBLook* allows the user to open, view, and save bitmap images (*.bmp, *.dib) (Fig. 1.2). Each Image is opened in a different window and has associated its own View-Object (instantiated from the *CDibView* class) and Document Object (instantiated from the *CDibDoc* class) (Fig. 1.6). The *View Object* is used to interact with the data associated to the bitmap which is stored in the *Document Object*.

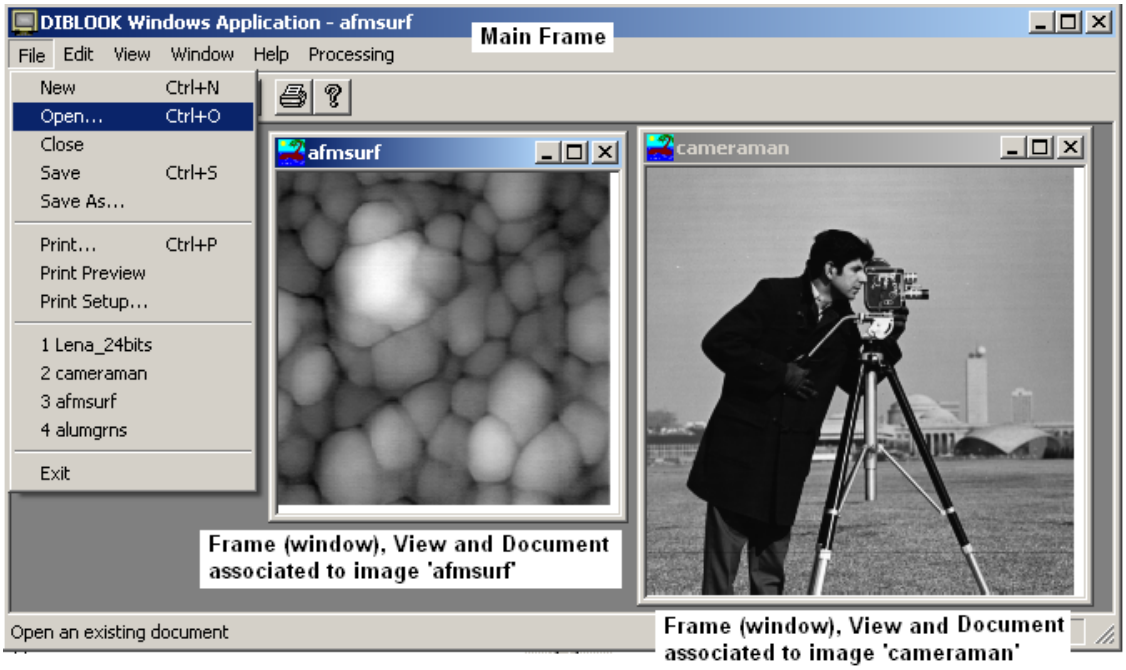

Fig. 1.2. Each opened image is displayed in a different frame/window and as its own *View Object* and *Document Object*.

1.3. Adding a processing function to the DIBLook application

In order to perform any sort of processing to an opened image the following steps should are required:

1. Switch on the *Resource View* tab (if it doesn't appear open it form the menu *View->Other Windows->Resource View*) and open the *IDR_DIBTYPE* menu (Fig. 1.3):

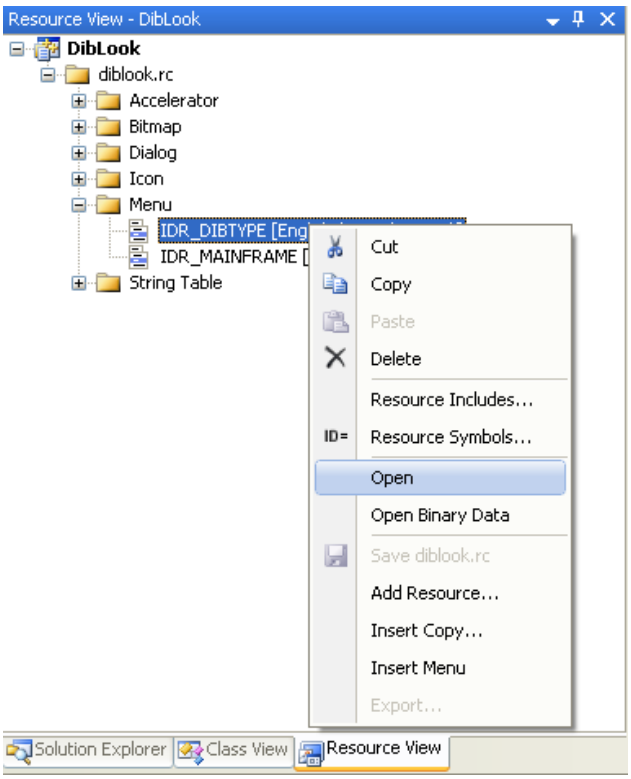

Fig. 1.3. Application resource window

2. Add a new menu entry by typing the name below the existing entry (Fig.1.4):

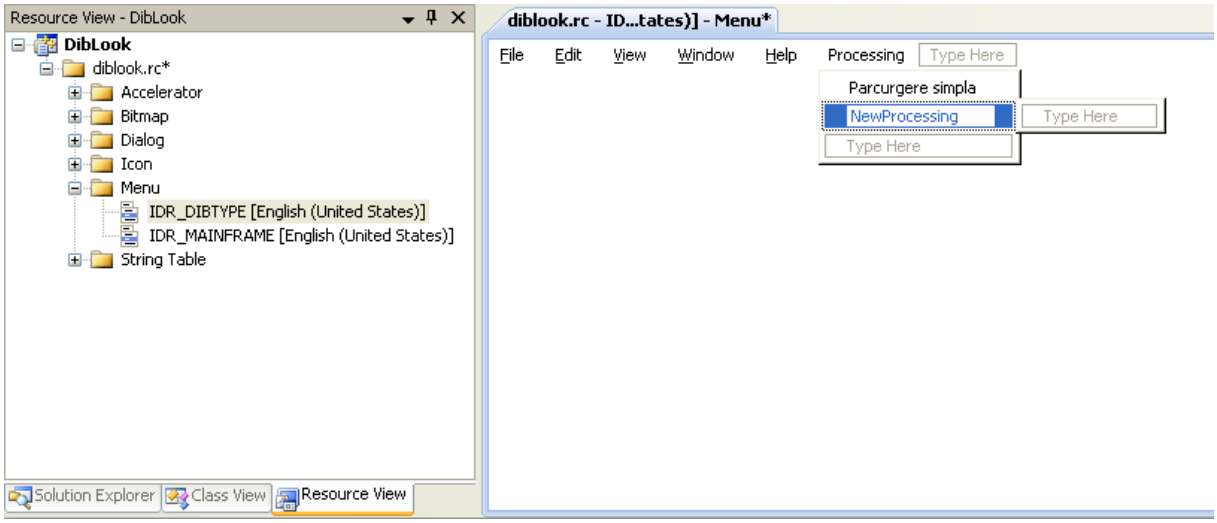

Fig. 1.4. Adding the *NewProcessing* entry to the menu

3. Associate a function to be executed when the menu is clicked using *Add Event Handler…* (right click on the menu):

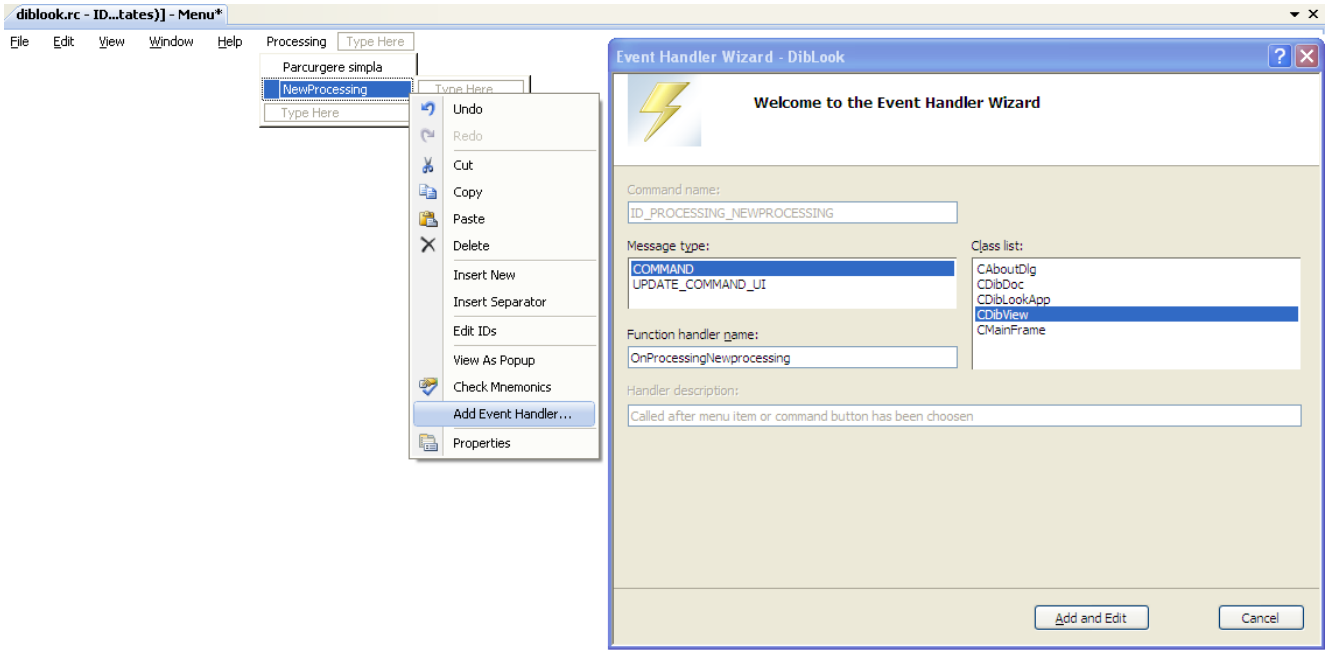

Fig. 1.5. Adding the function associated to the mouse click event on the *NewProcessing* entry

Important things to notice:
a. The new function should be a member of *CDibView* Class !
b. The function should be called by the *COMMAND* massage (generated by the ‘click’ on the menu).

4. Add and access the code for the new function through the *Add and Edit* button.

```
void CDibView::OnProcessingNewprocessing()
{
	// TODO: Add your command handler code here
}
```

1.4. A sample image processing function

A sample of a simple image processing function is given in the provided *DIBLook* application source code. The function *OnProcessingParcurgereSimpla* is a member of the *CDibView* Class (compulsory) and was created following the steps presented in the previous chapter (1.3). It shows how to access the pixels of an 8 bits/pixel source bitmap image, performs some simple operations (equals the entries from the LUT (grayscale) and the negatives each pixel of the image) and shows the results in a new/destination window (associated with its corresponding new/destination *Document* and *View* objects).

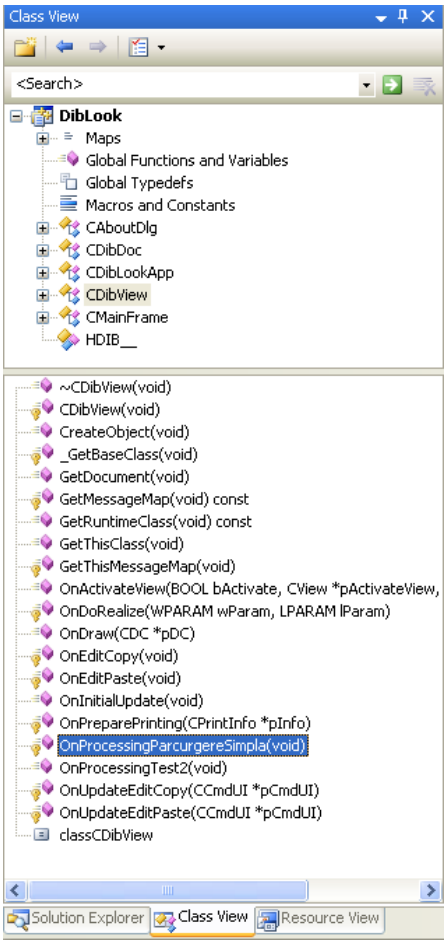

Fig. 1.6. The *Class-View* window and the *CDibView* class methods

```
void CDibView::OnProcessingParcurgereSimpla()
{
      BEGIN_PROCESSING();

      // Makes a grayscale image by equalizing the R, G, B components from the LUT
      for (int k=0;  k < iColors ; k++)
            bmiColorsDst[k].rgbRed=bmiColorsDst[k].rgbGreen=
                  bmiColorsDst[k].rgbBlue=k;

      // Goes through the bitmap pixels and performs their negative
      for (int i=0;i<dwHeight;i++)
            for (int j=0;j<dwWidth;j++)
                  lpDst[i*w+j]= 255 - lpSrc[i*w+j]; //makes image negative

      END_PROCESSING("Operation name");
}
```

1.4.1. The macro definition: `BEGIN_PROCESSING()`

It provides all the necessary initializations definitions and allocations. It is defined at the beginning of *dibview.cpp* file (is not provided with the original *DIBLook* sample from the MSDN. Be aware if you want to edit it (each line should be ended by '\' + <ENTER>, comments are not allowed etc.).

```
#define BEGIN_PROCESSING()                                                  \
     CDibDoc* pDocSrc=GetDocument();                                        \
     CDocTemplate* pDocTemplate=pDocSrc->GetDocTemplate();                  \
     CDibDoc* pDocDest=(CDibDoc*) pDocTemplate->CreateNewDocument();        \
     BeginWaitCursor();                                                     \
     HDIB hBmpSrc=pDocSrc->GetHDIB();                                       \
     HDIB hBmpDest = (HDIB)::CopyHandle((HGLOBAL)hBmpSrc);                  \
     if ( hBmpDest==0 ) {                                                   \
          pDocTemplate->RemoveDocument(pDocDest);                           \
          return;                                                           \
     }                                                                      \
     BYTE* lpD = (BYTE*)::GlobalLock((HGLOBAL)hBmpDest);                    \
     BYTE* lpS = (BYTE*)::GlobalLock((HGLOBAL)hBmpSrc);                     \
     int iColors = DIBNumColors((char *)&(((LPBITMAPINFO)lpD)->bmiHeader));  \
     RGBQUAD *bmiColorsDst = ((LPBITMAPINFO)lpD)->bmiColors;                \
     RGBQUAD *bmiColorsSrc = ((LPBITMAPINFO)lpS)->bmiColors;                \
     BYTE * lpDst = (BYTE*)::FindDIBBits((LPSTR)lpD);                       \
     BYTE * lpSrc = (BYTE*)::FindDIBBits((LPSTR)lpS);                       \
     DWORD dwWidth  = ::DIBWidth((LPSTR)lpS);                               \
     DWORD dwHeight = ::DIBHeight((LPSTR)lpS);                              \
     DWORD w= WIDTHBYTES(dwWidth*((LPBITMAPINFOHEADER)lpS)->biBitCount);    \
```

Comments:

//Access to the document object of the current view (associated to the image opened in the active window/frame
CDibDoc* pDocSrc=GetDocument();

//Access to its template
CDibDoc* pDocDest=(CDibDoc*) pDocTemplate->CreateNewDocument();

//Creates the destination object with the same template as the source document
CDibDoc* pDocDest=(CDibDoc*) pDocTemplate->CreateNewDocument();

//Gets the handle to the Source Image
HDIB hBmpSrc=pDocSrc->GetHDIB();

//Creates a copy of the source image handle in the destination one
HDIB hBmpDest = (HDIB)::CopyHandle((HGLOBAL)hBmpSrc);

//Gets the pointer to the beginning of the Destination and Source images in the memory
BYTE* lpD = (BYTE*)::GlobalLock((HGLOBAL)hBmpDest);
BYTE* lpS = (BYTE*)::GlobalLock((HGLOBAL)hBmpSrc);

//Gets the number of entries from the LUT (for an indexed image): iColors = 2^n-1
// n = 1, 4 or 8 (no. of bits/pixel)
// For a RGB image (n = 16, 24 or 32 bits/pixel): iColors = 0
int iColors = DIBNumColors((char *)&(((LPBITMAPINFO)lpD)->bmiHeader));

```
//Gets the pointer to the beginning of the LUT
RGBQUAD *bmiColorsDst = ((LPBITMAPINFO)lpD)->bmiColors;
RGBQUAD *bmiColorsSrc = ((LPBITMAPINFO)lpS)->bmiColors;

//Gets the pointers to the beginning of the bitmap data (pixels) of the destination/source images
BYTE * lpDst = (BYTE*)::FindDIBBits((LPSTR)lpD);
BYTE * lpSrc = (BYTE*)::FindDIBBits((LPSTR)lpS);

// Gets the width and the height and the bitmap (image data) [pixels]
DWORD dwWidth  = ::DIBWidth((LPSTR)lpS);
DWORD dwHeight = ::DIBHeight((LPSTR)lpS);

//Gets the width of an image line from the memory in number of double-words for a bitmap (1 double word = 4 bytes = 32 bits = memory alignment in a 32 bit Windows OS); biBitCount represents the number of bits/pixel
DWORD w=WIDTHBYTES(dwWidth*((LPBITMAPINFOHEADER)lpS)->biBitCount);
```

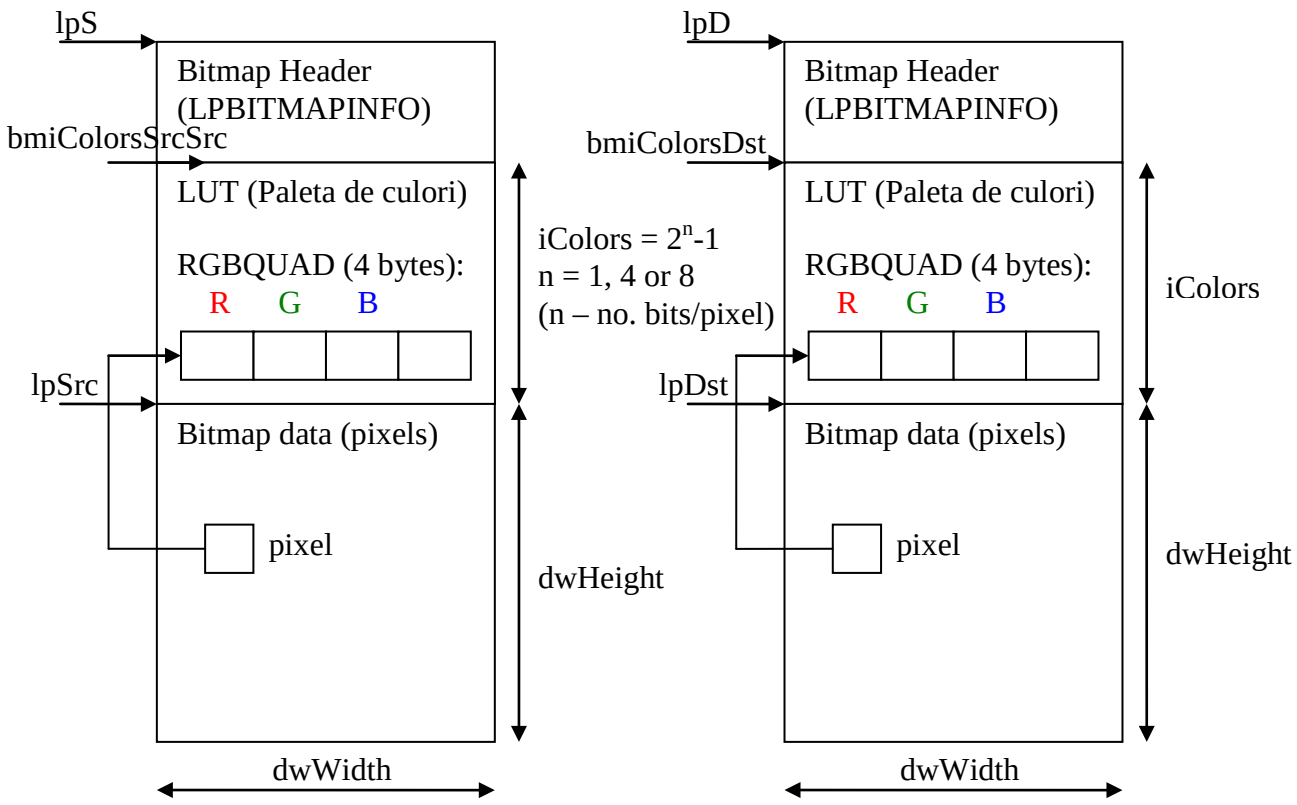

Fig. 1.7. Structure of a bitmap (with LUT – 1, 4 or 8 bits / pixel) in the memory (source image and destination image).

1.4.2. The macro definition: `END_PROCESSING("Operation name");`

```
#define END_PROCESSING(Title)                                          \
     ::GlobalUnlock((HGLOBAL)hBmpDest);                                \
     ::GlobalUnlock((HGLOBAL)hBmpSrc);                                 \
     EndWaitCursor();                                                  \
     pDocDest->SetHDIB(hBmpDest);                                      \
```

```
pDocDest->InitDIBData();                                                    \
pDocDest->SetTitle((LPCSTR)Titlu);                                          \
CFrameWnd* pFrame=pDocTemplate->CreateNewFrame(pDocDest,NULL);              \
pDocTemplate->InitialUpdateFrame(pFrame,pDocDest);
```

Comments:

//Releasing the handles of the bitmaps
::GlobalUnlock((HGLOBAL)hBmpDest);
::GlobalUnlock((HGLOBAL)hBmpSrc);

//Setting the handle of the destination image and initializing other data in the associated Document object
pDocDest->SetHDIB(hBmpDest);
pDocDest->InitDIBData();
pDocDest->SetTitle((LPCSTR)Titlu);

//Creating a frame for the destination image (results) and updating its content with the processed image
CFrameWnd* pFrame=pDocTemplate->CreateNewFrame(pDocDest,NULL);
pDocTemplate->InitialUpdateFrame(pFrame,pDocDest);

1.4.3. Accessing the LUT

The LookUp Table (the colors palette) can be accessed through the *bmiColorsSrc/ bmiColorsDst* pointer. It is a table of 4 bytes entries (RGBQUAD structure) containing a byte for each color (R,G,B) and a reserved one.

In the given example the LUT entries of the destination image are equalized with their index, obtaining a grayscale image.

```
// Makes a grayscale image by equalizing the R, G, B components from the LUT
for (int k=0;  k < iColors ; k++)
     bmiColorsDst[k].rgbRed=bmiColorsDst[k].rgbGreen=
          bmiColorsDst[k].rgbBlue=k;
```

1.4.4. Accessing the image pixels from the bitmap data for an indexed image (with LUT)

The pixels of an 8 bits/pixel bitmap image can be accessed as in the example bellow:

```
// Goes through the bitmap pixels and performs their negative
for (int i=0;i<dwHeight;i++)
     for (int j=0;j<dwWidth;j++)
      {
          lpDst[i*w+j]= 255 - lpSrc[i*w+j]; //makes image negative
     }
```

The location of the current pixel (*i*,*j*) of the bitmap is at address *i***w*+*j* relative to the beginning of the bitmap data. *w* is the width of a line in number of double words (1 double word = 4 bytes = 32 bits = memory alignment in a 32 bit Windows OS):

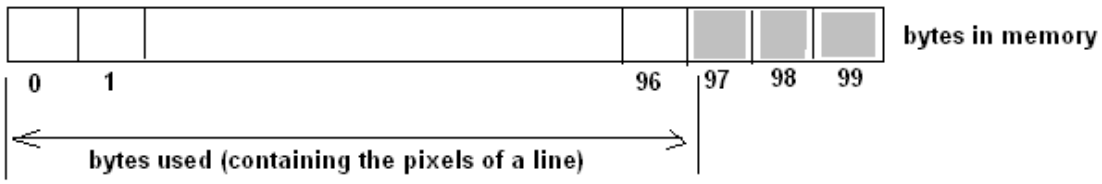

Fig. 1.8. Example of how a line of 97 pixels is stored in the memory.

1.4.5. Accessing the image pixels from the bitmap data for an RGB image

Images with 16, 24, or 32 bits/pixel don't have a LUT. Instead, each pixel from the bitmap data contains the color information (the values of the 3 components R, G, B) in the bitmap data. In the following example the most common RGB image will be considered: 24 bits/pixel image (also called RGB24). In Fig. 1.9 the structure of such an image in the memory is shown:

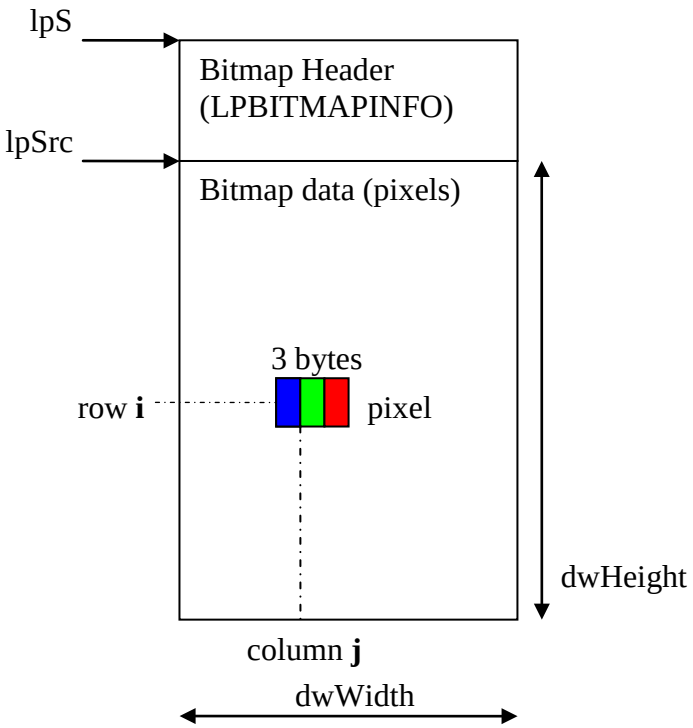

Fig. 1.9. Structure of a 24 bits/pixel (RGB24) bitmap image (without LUT) in the memory.

The pixels (the color components) of a RGB24 bitmap image can be accessed as in the example bellow:

```
INCEPUT_PRELUCRARI();
BYTE red, green, blue;

for (int i=0;i<dwHeight;i++)
     for (int j=0;j<dwWidth;j++)
     {
          red = lpSrc[i*w+3*j+2];
          green = lpSrc[i*w+3*j+1];
          blue = lpSrc[i*w+3*j];
     }
```

1.5. Practical work

1. Make a copy of the *DIBLook* application in your local (working) folder.
2. Open the *diblook.sln* (the solution file) in Visual C++ 9.0.
3. Build and run the application.
4. Test the provided sample function: *Processing->Parcurgere simpla*
5. Add a new menu and associated processing function (using the hints from chapter 1.3 and the example from chapter 1.4).
6. Apply some simple arithmetic operations on the pixels of the input image (adding/subtracting/multiplying with a constant) and put the results in the corresponding pixels of the destination image. Add some supplementary conditions in order to normalize the results (the values of the output/destination pixels) in the BYTE range (0 … 255).
7. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms!!!**

References

[1] http://msdn2.microsoft.com/en-us/library/ms632591(VS.85).aspx
[2] http://www.functionx.com/visualc/Lesson05.htm
[3] http://msdn2.microsoft.com/en-us/library/4x1xy43a(VS.80).aspx
[4] http://msdn2.microsoft.com/en-us/library/d06h2x6e(VS.71).aspx