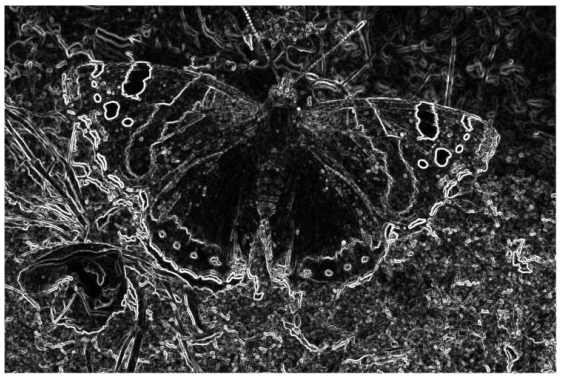
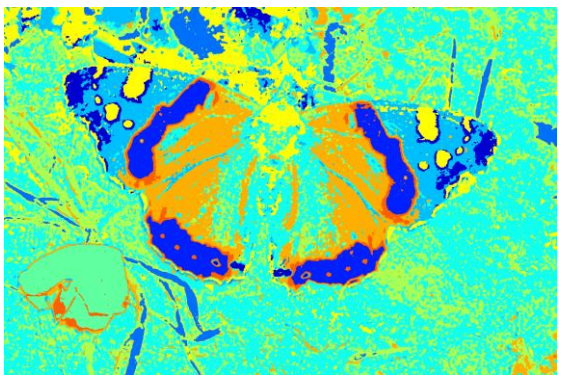




Sergiu Nedevschi, Tiberiu Marița,
Radu Dănescu, Florin Oniga,
Raluca Brehar, Ion Giosan, Silviu Bota,
Anca Ciurte, Andrei Vatavu

Image Processing

Laboratory Guide

UTPRESS

Cluj-Napoca, 2016
ISBN 978-606-737-137-6

Editura U.T.PRESS
Str.Observatorului nr. 34
C.P.42, O.P. 2, 400775 Cluj-Napoca
Tel.:0264-401.999 / Fax: 0264 - 430.408
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Director: Ing. Călin D. Câmpean

Recenzia: Prof.dr.ing. Ioan Salomie
Prof.dr.ing. Gheorghe Sebestyen-Pal

ISBN 978-606-737-137-6
Bun de tipar: 07.01.2016

Contents

Preface

This guide targets the students in the 3rd year of bachelor degree at Computer Science Department of Technical University of Cluj-Napoca, but not only. This guide can be usefull to anyone who is interested in learning image processing.

The structure of the guide was didacticaly designed, each chapter featuring a practical aspect. The reader is advised to go through the chapters in the order of presentation, as each chapter may contain aspects that have been studied in previous chapters.

The current edition of this guide is the result of collective research work of Image Processing and Pattern Recognition Group (IPPRG), Computer Science Department. Based on the group experience in the field, the selected topics to be addressed in this guide are those that allow an easy approach while covering the fundamental image processing aspects.

The students of the Technical University of Cluj-Napoca will use this guide to pursuit the laboratory activities. Each chapter begins with a brief overview of theoretical concepts followed by the practical aspects required to accomplish the implementation. The practical activities are stated at the end of each chapter. Students are advised to read the entire chapter before attending the laboratory, to familiarize themselves with the current theme.

The team of authors wishes you a pleasant reading !

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++ 12.0 (Visual Studio 2013), 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].

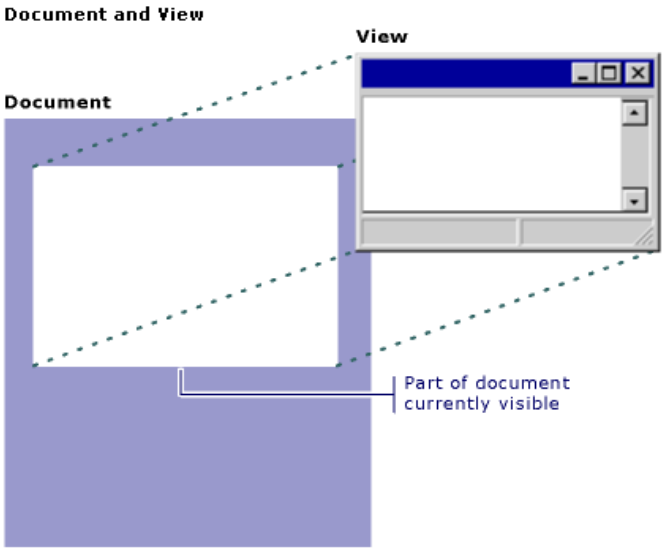


Fig. 1.1. The Document View architecture [3]

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*.

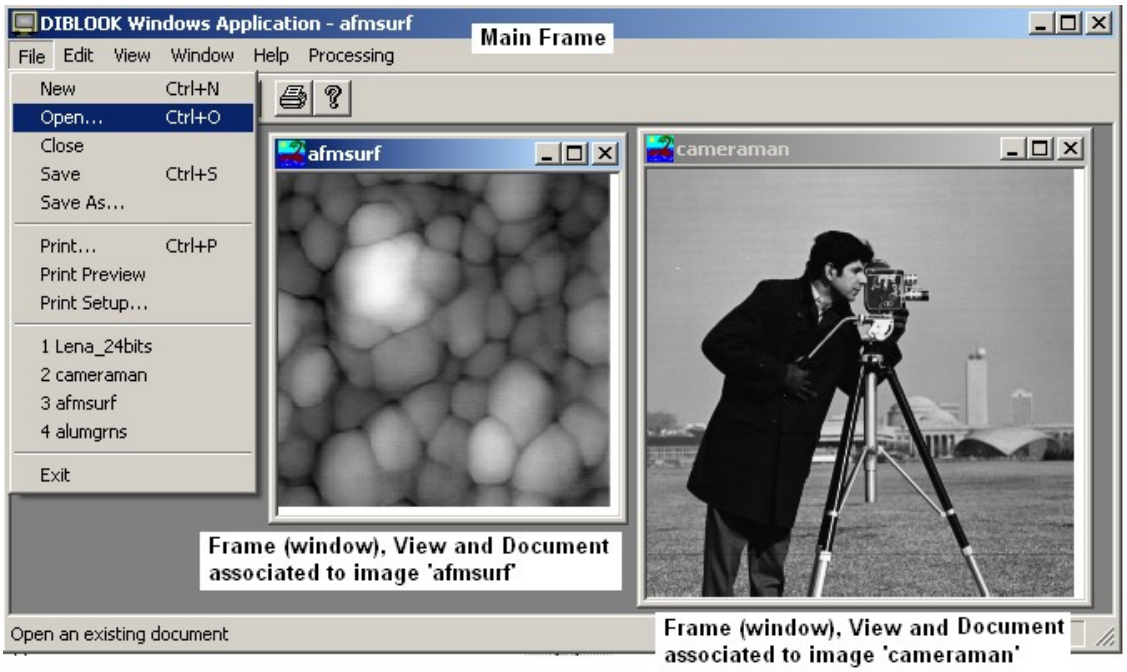


Fig. 1.2. Each opened image is displayed in a different frame/window and as its own *View Object* and *Document Object*.

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 from the menu *View -> [Other Windows->] Resource View*) and open the *IDR_DIBTYPE* menu (Fig. 1.3):

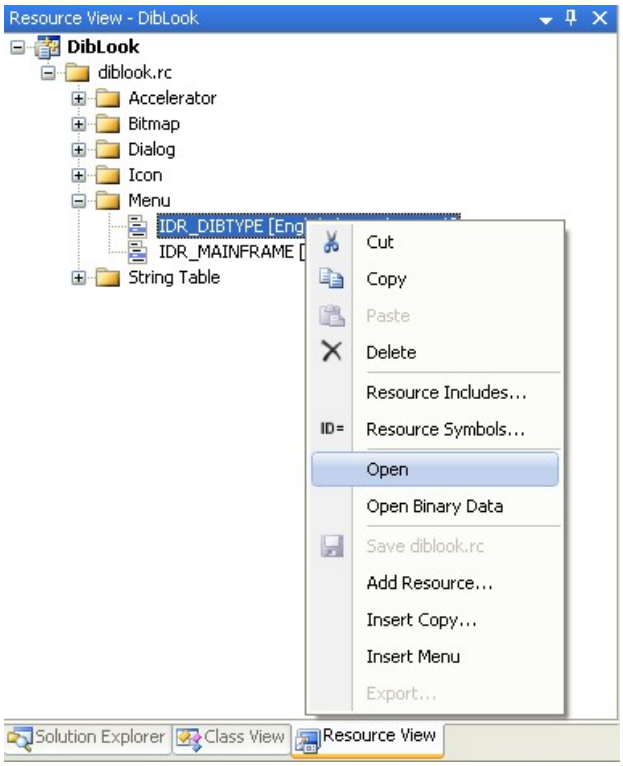


Fig. 1.3. Application resource window

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

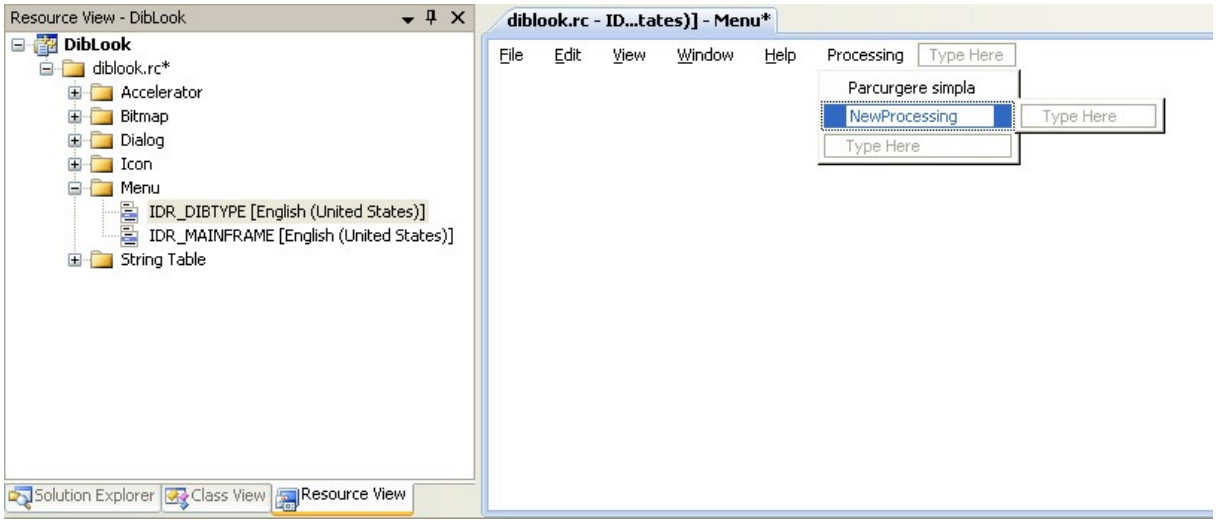


Fig. 1.4. Adding the *NewProcessing* entry to the menu

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

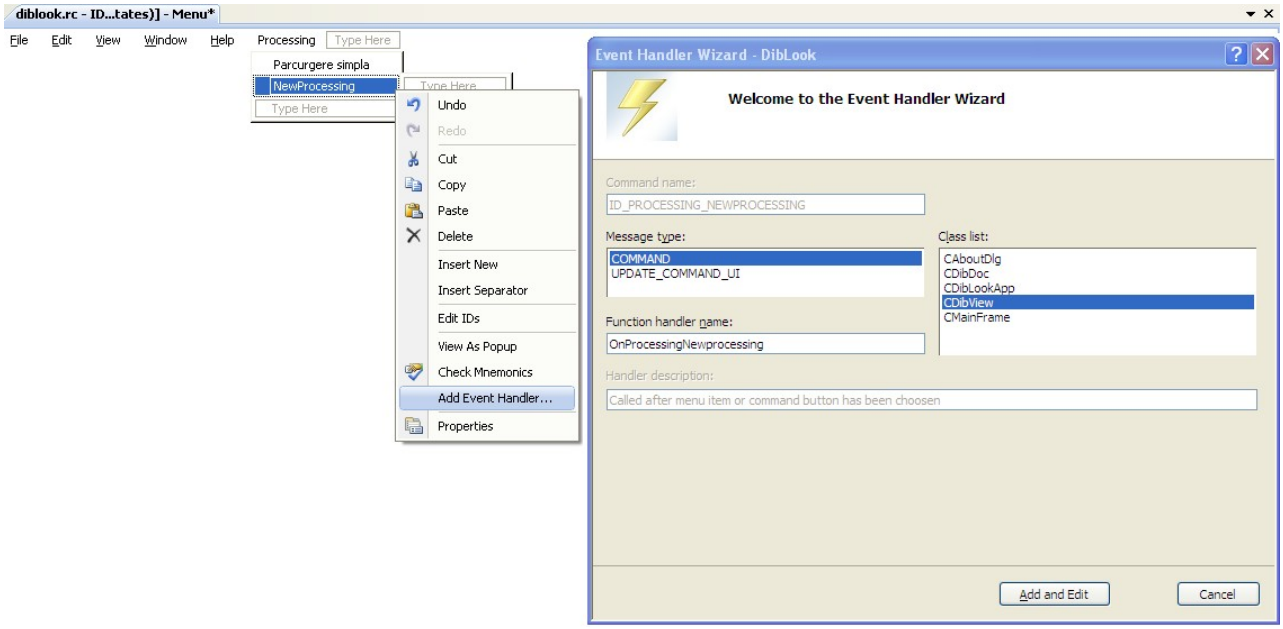


Fig. 1.5. Adding the function associated to the mouse click event on the *NewProcessing* entry

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).

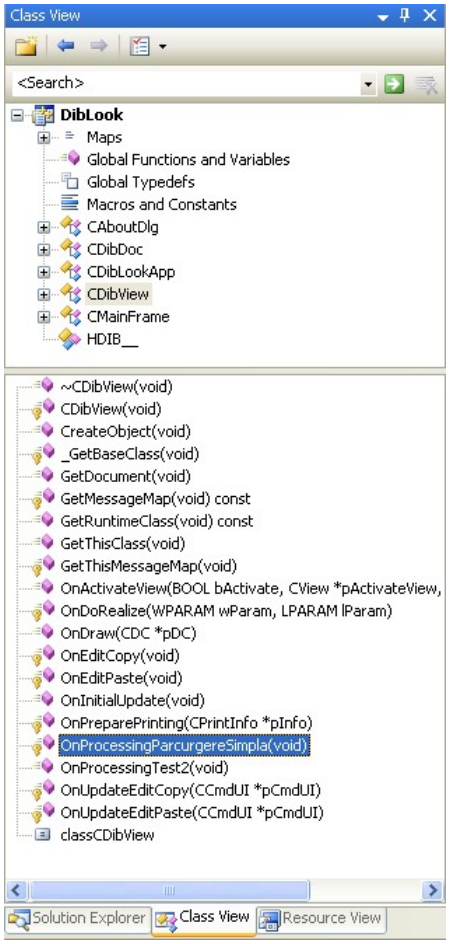


Fig. 1.6. The *Class-View* window and the *CDibView* class methods

```
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);
```

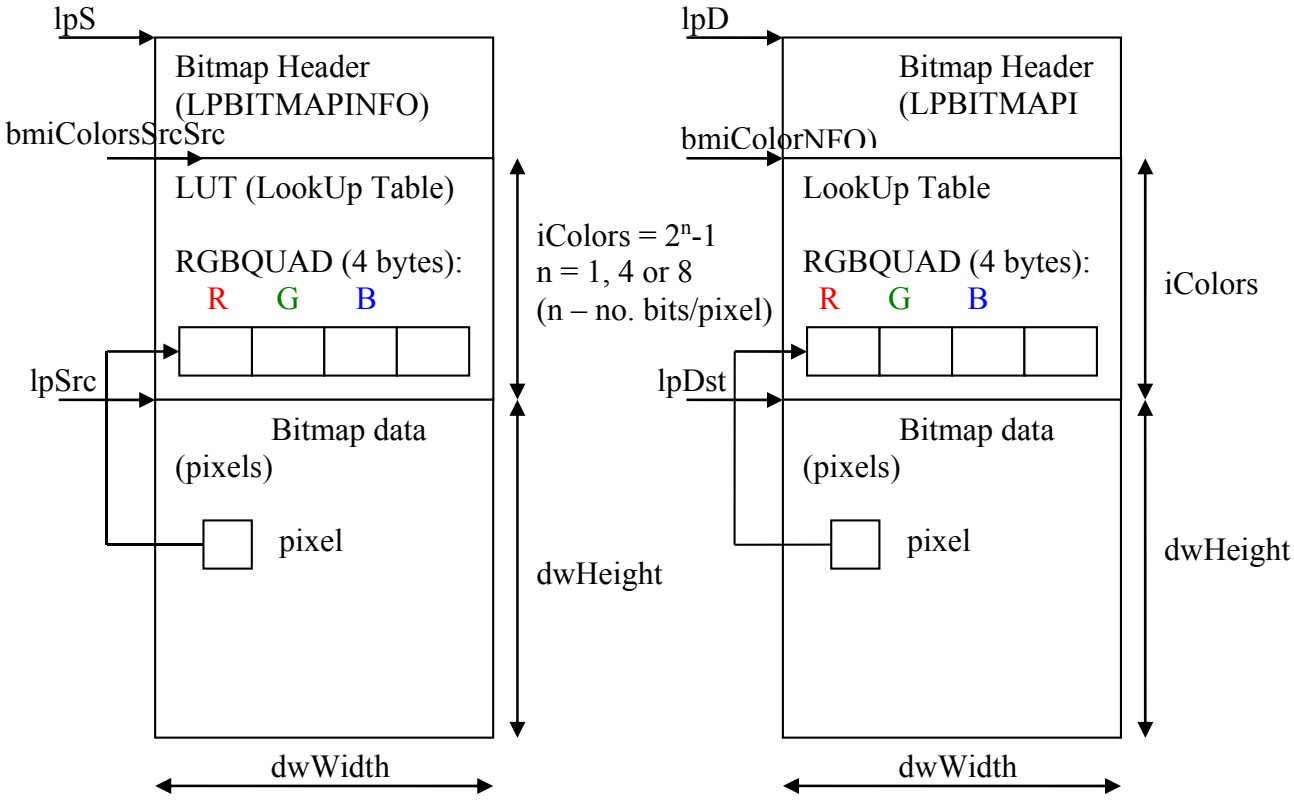


Fig. 1.7. Structure of a bitmap (with LUT – 1, 4 or 8 bits / pixel) in the memory (source image and destination image).

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):

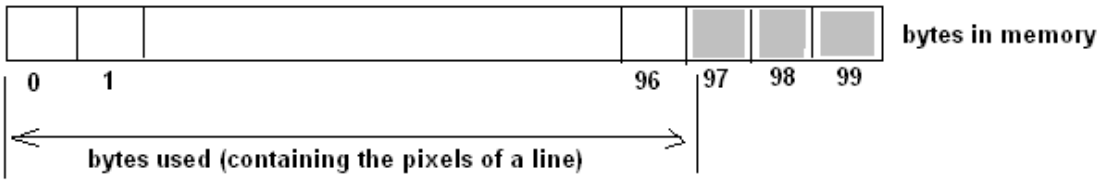


Fig. 1.8. Example of how a line of 97 pixels is stored in the memory.

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:

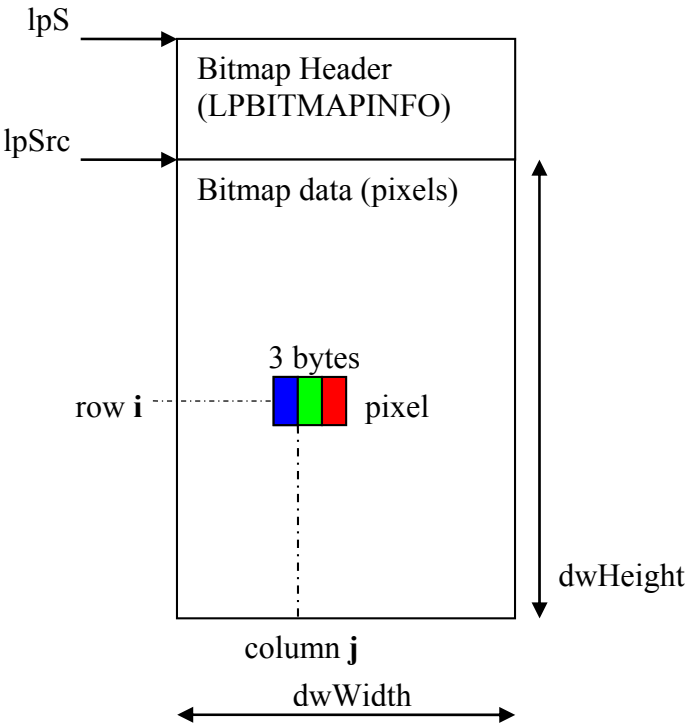


Fig. 1.9. Structure of a 24 bits/pixel (RGB24) bitmap image (without LUT) in the memory.

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

```
BEGIN_PROCESSING();
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++ 10.0.
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. Close the project (Visual Studio environment) and then execute the file *clean.bat*, located in the project's folder, in order to clean it from the files resulted at the build time.
8. Implement the above operations using OpenCV.
9. **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!!!**

1.6 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

2 The RGB color model. Grayscale and black&white conversions

2.1 Introduction

The purpose of this second laboratory is to learn the basic color handling procedures related to the digital bitmap images.

2.2 The RGB color model

The color of each pixel (both for the acquisition device (camera) and for displays (TV, CRT, LCD)) is obtained through the combination of the tree elementary colors: **R**ed, **G**reen and **B**lue (additive color model – fig. 2.1 and 2.2).

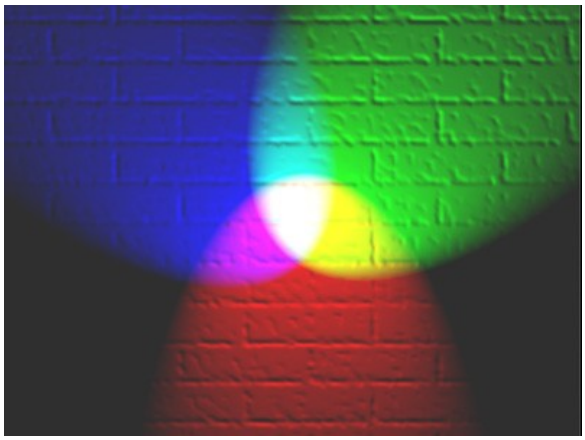


Fig. 2.1. A representation of additive color mixing. Projection of primary color lights on a screen shows secondary colors where two overlap; the combination of all three of red, green, and blue in appropriate intensities makes white [1].

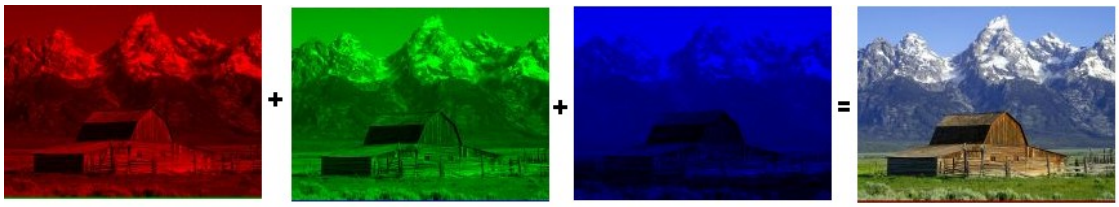


Fig. 2.2. The color of an image is obtained by combining the tree elementary colors for each pixel (three elementary color images).

Therefore, each pixel of a bitmap image will be characterized by a value for each of the tree primary colors. Its color is a point from the 3D space of the RGB color model (fig. 2.3). In this color cube, the origin of the R, G and B axes corresponds to the *black color* (0,0,0). The opposite vertex of the cube corresponds to the *white color* (255,255,255). The diagonal between the black and the white colors corresponds to the *grayscale* values (R=G=B). Three vertexes correspond to the primary colors **R**ed, **G**reen and **B**lue. The other 3 vertexes are corresponding to the complementary colors: **C**yan, **M**agenta and **Y**ellow. If the origin of the color model is translated into the '*white'* point and the tree axes of the coordinate system are considered the C, M and Y axes, the complementary CMY color model is obtained (which is used in the color printing devices).

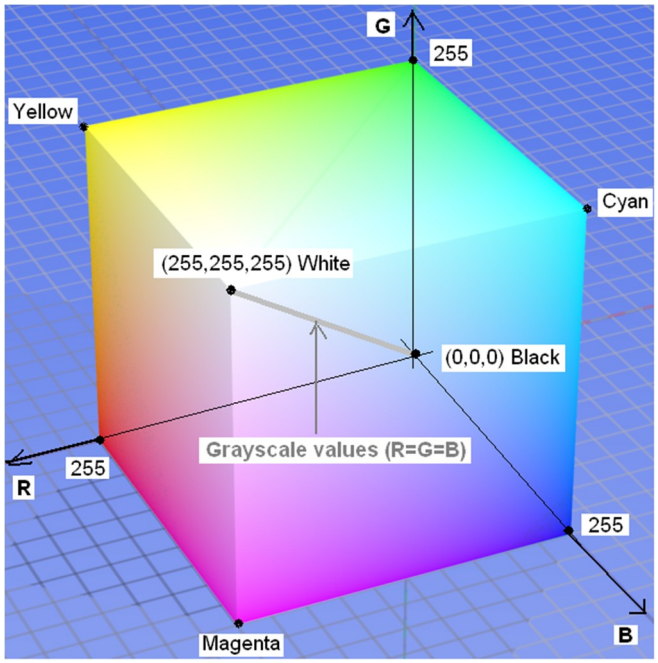


Fig. 2.3. The RGB color model mapped to a cube. In this example (RGB24 bitmap image) each color is represented on 8 bits (256 colors). The total number of colors is 2^8x2^8x2^8 = 2^{34} = 16.777.216.

For an RGB24 (24 bits/pixels) image the whole color space can be represented (true color image). In an indexed image (with LUT) only a subspace of the color space from Fig. 2.3 can be represented. In this context the number of bits/pixel (the number of bits used to encode each color) is called 'color *depth'* (table 2.1):

Table 2.1. Color depths vs. image type

Color Depth	No. of. Colors	Color Mode	Palette (LUT)
1 bit color	2	Indexed Color	Yes
4 bit color	16	Indexed Color	Yes
8 bit color	256	Indexed Color	Yes
16 bit color	65536	True Color	No
24 bit color	16.777.216	True Color	No
32 bit color	16.777.216	True Color	No

There are also other color models [2] used to represent the color but they will not be discussed here.

2.3 Conversion of a color image into a grayscale one

In order to convert a color image into a grayscale one, the 3 color components of each pixel must be equalized. A common procedure is to make the average of the three color components:

$$R_{Dst} = G_{Dst} = B_{Dst} = \frac{R_{Src} + G_{Src} + B_{Src}}{3} \qquad (2.1)$$

2.3.1 The case of the RGB24 (24 bits/pixel) images

In this case the formula from (2.1) can be applied by accessing the tree color components of each pixel from the source/destination image as shown in Laboratory 1.

2.3.2 The case of the indexed images (with LUT).

In this case the entries of the destination's image LUT should be iterated (see example from Laboratory 1) and the color components of each entry should be converted using (2.1).After this simple operation, a common situation which can occur is the following: the entries of the LUT are not any more ordered in ascending direction upon their grayscale values (Fig. 2.4):

Old index	R	G	B	X
0	100	100	100	-
1	20	20	20	-
2	32	32	32	-
.				
.				
.				
255	78	78	78	-

Fig. 2.4. Un-sorted LUT after color-to-grayscale conversion of an 8 bits/pixel indexed image.

Some further processing on the grayscale image would require a sorted LUT. Therefore this operation should be done after the conversion.

A simple method to sort the LUT:

1. Create a BYTE vector of size 256:

```
ex: BYTE g[256];
```

2. Go through the LUT and initialize the values of *g* with one of the color components of the entry *k*. of the unsorted LUT (Fig. 2.4) followed by the "sorting" of the LUT by assigning the value of the index '*k*' to each of the tree color components from the entry '*k*' (in this order):

```
for (k = 0 … iColors) {
        // initialize vector g
        g[k] = palette[k].rgbRed;
        // „sort" the LUT
        palette[k].rgbRed = palette[k].rgbGreen = palette[k].rgbBlue = k;
}
```

New Index	R	G	B	X	g
0	0	0	0	-	5
1	1	1	1	-	23
2	2	2	2	-	14
.					
.					
.					
255	255	255	255	-	243

Fig. 2.5. Sorted LUT after step 2.

3. Finally the '*Bitmap data*' (image pixels) should be iterated and the old values (indexes) of each pixel should be replaced with the new ones, according to the established correspondence: $k \rightarrow g(k)$ (Fig. 2.5):

```
k = lpDst[i*w+j];
lpDst[i*w+j] = g[k];
```

2.4 Accessing the LUT's contents in bitmap header

The following example shows how the information from the bitmap header can be accessed:

```
//Gets the pointer to the beginning of the Bitmap Header in memory in a
//as BITMAPINFO STRUCTURE pointer
LPBITMAPINFO pBitmapInfoSrc = (LPBITMAPINFO)lpS;
```

or

```
BITMAPINFO *pBitmapInfoSrc = (BITMAPINFO*) lpS;

// gets the size of the bitmap
pBitmapInfoSrc->bmiHeader.biSize;
//gets the number of bits/pixel
pBitmapInfoSrc->bmiHeader.biBitCount; //the number of bits/pixel (1, 4, 8,
//16, 24, 32)
.........
```

where the BITMAPINFO and BITMAPINFOHEADER structures [3] are defined as bellow:

```
typedef struct tagBITMAPINFO {
  BITMAPINFOHEADER bmiHeader;
  RGBQUAD          bmiColors[1];
} BITMAPINFO, *LPBITMAPINFO;

typedef struct tagBITMAPINFOHEADER{
  DWORD  biSize;
  LONG   biWidth;
  LONG   biHeight;
  WORD   biPlanes;
  WORD   biBitCount;
  DWORD  biCompression;
  DWORD  biSizeImage;
  LONG   biXPelsPerMeter;
  LONG   biYPelsPerMeter;
  DWORD  biClrUsed;
  DWORD  biClrImportant;
} BITMAPINFOHEADER, *LPBITMAPINFOHEADER;
```

The way in which the LUT entries can be accessed was presented in Laboratory 1!

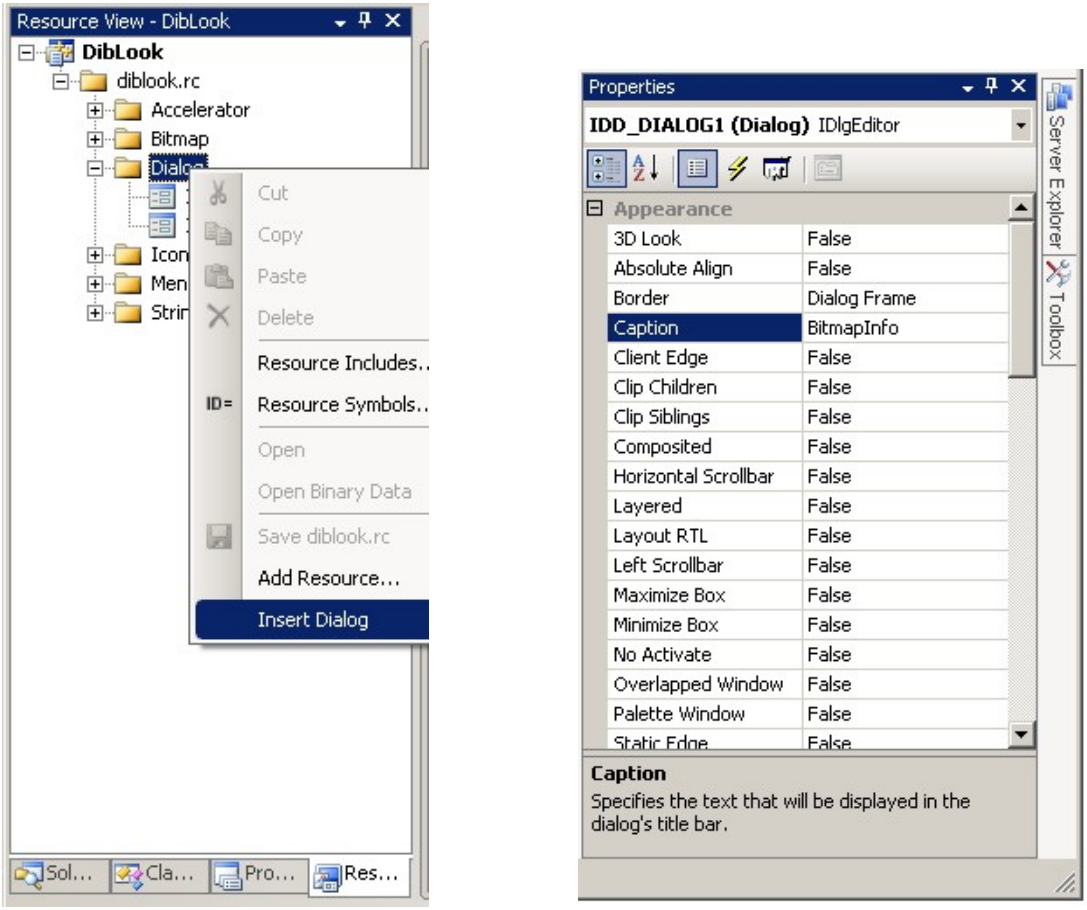
2.5 Guide to display information in a dialog box

2.5.1 Creating a new *Dialog Box* resource

1. Switch in the *Workspace* Window on the *Resource View* tab, expand the *Dialog* element, right-click on it and then left-click on *Insert Dialog* (Fig. 2.6.a).

2. Right-click on the newly created dialog resource. In the right part of the VC environment *Properties* window associated to the dialog will be activated(Fig. 2.6.b) you can change the name (recommended), the style, the resource ID (not recommended) and so on.

3. Right-click on the newly created dialog resource and select the *Add class*' option (Fig. 2.7). The „*MFC Class Wizard*" dialog will be opened (Fig.2.8).



a. b.

Fig. 2.6.

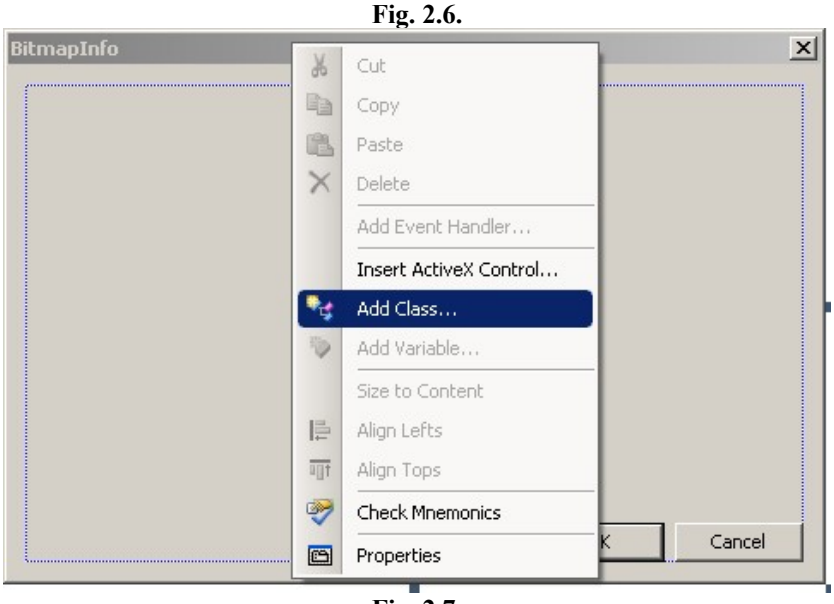


Fig. 2.7.

4. Give a relevant name for the class associated to the dialog (example CBitmapInfoDlg. The wizard automatically creates the appropriate *.h and *.cpp files for the class (the name of the file is usually similar with the name of the dialog – you don't have to change it). The new class can be easily accessed through the *ClassView* tab of the *Workspace* window (where is added automatically – Fig. 2.10.d).

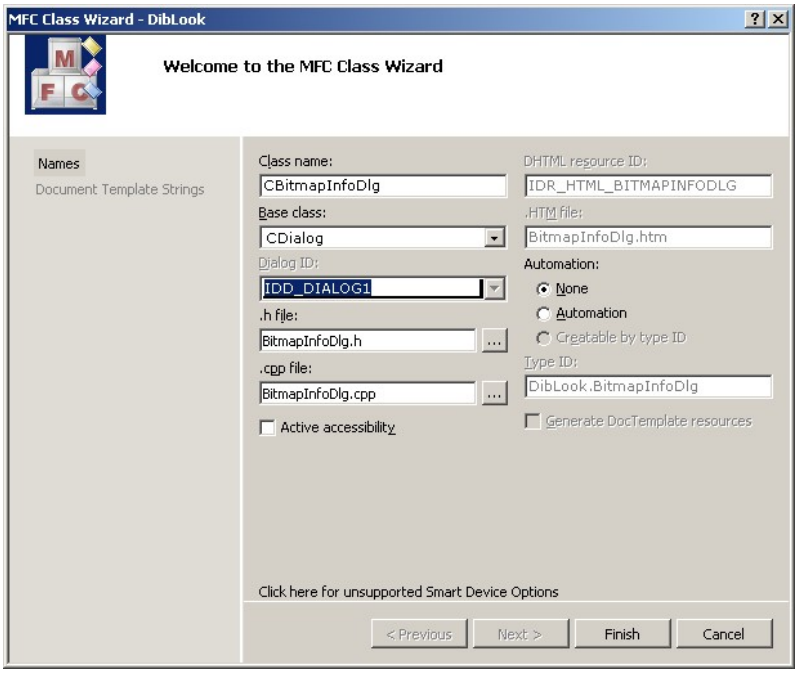


Fig. 2.8.

5. Include the header of the new dialog class in the *#include* section of the *dibview.cpp* file:

```
#include "BitmapInfoDlg.h"
```

6. Create an instance (object) of the new class and display the dialog in your processing function. The code below only displays the new dialog resource in *modal* way (the code following the *DoModal()* call will be executed only after the dialog is closed). There are also ways to display a dialog in a non-modal way (homework if you want).

```
void CDibView::OnProcessingAfisarebmpheader()
{
	// You can use the call to the macro bellow when
	// you don't need to display a destination image
	BEGIN_SOURCE_PROCESSING;

	//creates an instance (object) of the dialog class
	CBitmapInfoDlg dlgBmpHeader;

	// TODO: Add here the code for reading the bitmap header content and for
	// writing it in the dialog

	//displays dialog in 'modal' mode
	dlgBmpHeader.DoModal();

	// You can use the call to the macro bellow when
	// you don't need to display a destination image
	END_SOURCE_PROCESSING;
}
```

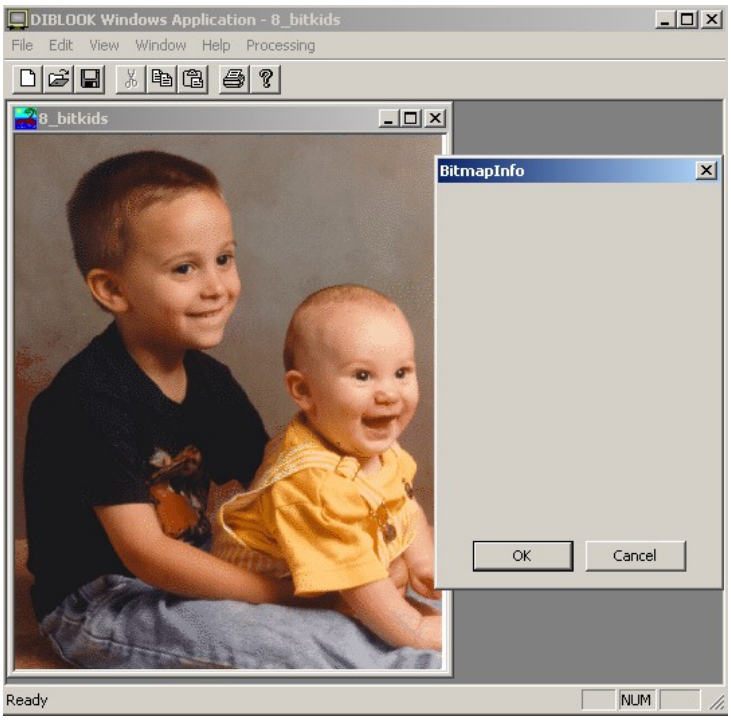


Fig. 2.9.

2.5.2 Designing the dialog box

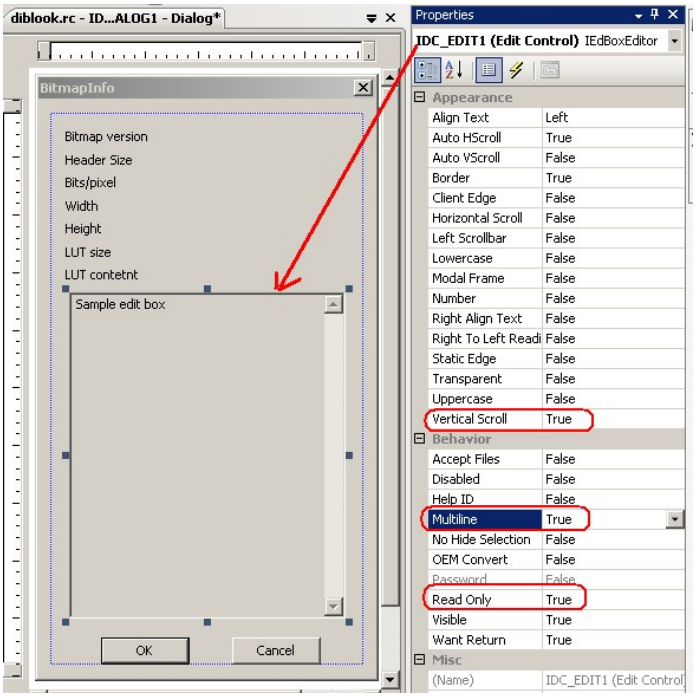
In order to show or to get some data from the dialog box, *Controls* should be added to the already created dialog resource. The most common controls are the *Static Text* (for output) and the *Edit Control* (for output/input). In the current case only output is required. In order to add a control to the dialog, select the *Toolbox* window (open it from menu *View->Toolbox*) and within select a control and the click in the dialog to drop the control in the desired location.

Individual fields (as the bitmap height, width etc.) can be easily shown in *Static Text* controls. In order to write something in a static text, an individual ID should be given explicitly to each static text control (the default/generic ID is ID_STATIC for all static text controls).

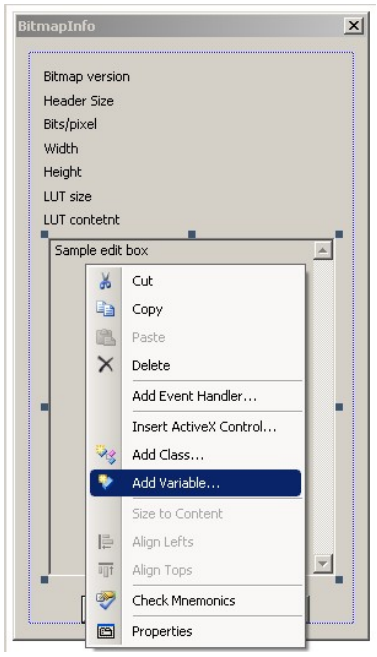
Tables (as the content of the LUT) can be shown in an *Edit Control*. *Edit Control*s have allocated an individual ID by default (there is no need to change it). For the *Edit Control*s the styles can be edited for the desired appearance (Fig. 2.10.a).

Once the controls are added to the dialog, a set of variables should be associated with them. This can be done using the *Add Member Variable Wizard* dialog ((right-click on the control resource and select the *Add Variable* option – Fig.2.10.b).

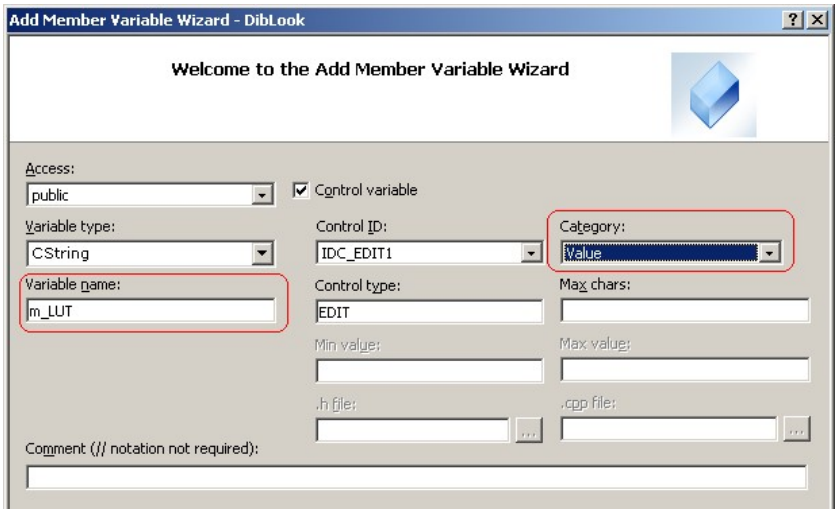
In the *Add Member Variable Wizard* dialog (Fig. 2.10.c) to each control (identified by each ID) a member variable should be added (specified by name, type (use *CString*), category (select *Value*) etc.). The member variables associated to the controls using the *Add Member Variable Wizard* are added automatically to the dialog class (Fig. 2.10.d).


diblook.rc - ID...ALOG1 - Dialog*
BitmapInfo
Bitmap version
Header Size
Bits/pixel
Width
Height
LUT size
LUT contetnt
Sample edit box
OK
Cancel
Properties
IDC_EDIT1 (Edit Control) IEdBoxEditor
Appearance
Align Text Left
Auto HScroll True
Auto VScroll False
Border True
Client Edge False
Horizontal Scroll False
Left Scrollbar False
Lowercase False
Modal Frame False
Number False
Right Align Text False
Right To Left Readi False
Static Edge False
Transparent False
Uppercase False
Vertical Scroll True
Behavior
Accept Files False
Disabled False
Help ID False
Multiline True
No Hide Selection False
OEM Convert False
Password False
Read Only True
Visible True
Want Return True
Misc
(Name) IDC_EDIT1 (Edit Control)

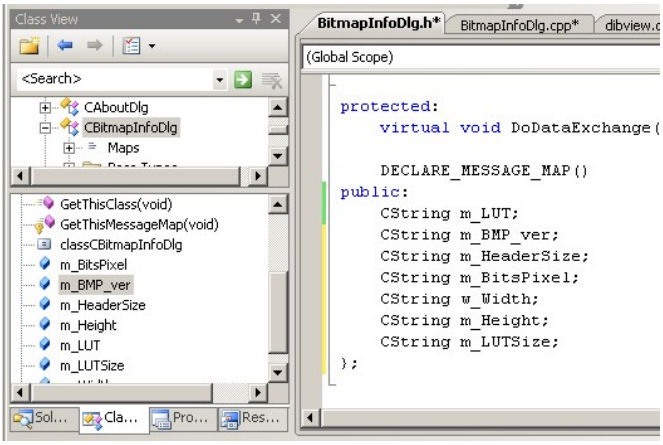
a.


BitmapInfo
Bitmap version
Header Size
Bits/pixel
Width
Height
LUT size
LUT contetnt
Sample edit box
Cut
Copy
Paste
Delete
Add Event Handler...
Insert ActiveX Control...
Add Class...
Add Variable...
Size to Content
Align Lefts
Align Tops
Check Mnemonics
Properties

b.


Add Member Variable Wizard - DibLook
Welcome to the Add Member Variable Wizard
Access:
public
Control variable
Variable type:
CString
Control ID:
IDC_EDIT1
Category:
Value
Variable name:
m_LUT
Control type:
EDIT
Max chars:
Min value:
Max value:
.h file:
.cpp file:
Comment (// notation not required):

c.


Class View
<Search>
CAboutDlg
CBitmapInfoDlg
Maps
GetThisClass(void)
GetThisMessageMap(void)
classCBitmapInfoDlg
m_BitsPixel
m_BMP_ver
m_HeaderSize
m_Height
m_LUT
m_LUTSize
BitmapInfoDlg.h*
BitmapInfoDlg.cpp*
(Global Scope)
protected:
virtual void DoDataExchange(
DECLARE_MESSAGE_MAP()
public:
CString m_LUT;
CString m_BMP_ver;
CString m_HeaderSize;
CString m_BitsPixel;
CString w_Width;
CString m_Height;
CString m_LUTSize;
};

d

Fig. 2.10.

2.5.3 Writing into the dialog box

In order to write the desired data in a dialog box, the data should be written into the variables associated with the controls of the dialog. This should be done in the processing function (before calling the DoModal() method which shows the dialog):

```
void CDibView::OnProcessingAfisarebmpheader()
{
     BEGIN_SOURCE_PROCESSING;

 //creates an instance (object) of the dialog class
 CBitmapInfoDlg dlgBmpHeader;

 LPBITMAPINFO pBitmapInfoSrc = (LPBITMAPINFO)lpS;

 dlgBmpHeader.m_Width.Format(_TEXT("Image width [pixels]: %d"),
                             pBitmapInfoSrc->bmiHeader.biWidth);
             // and the other info ......

 // Stores the entries of the LUT in the CString variable m_LUT
 // (associated to the edit control for displaying the LUT)
 CString buffer;
 for (int i=0;i<iColors;i++)
 {
       buffer.Format(_TEXT("%3d.\t%3d\t%3d\t%3d\r\n"),i,
                             bmiColorsSrc[i].rgbRed,
                             bmiColorsSrc[i].rgbGreen,
                             bmiColorsSrc[i].rgbBlue);
       dlgBmpHeader.m_LUT+=buffer;
 }

 //displays the dialog in 'modal' mode
 dlgBmpHeader.DoModal();

 END_SOURCE_PROCESSING;
}
```

2.6 Conversion of a grayscale image in a binary (black & white) image

A binary (black & white image) is an image which contains only 2 colors: black and white. A Binary image can be obtained from a grayscale image through a simple operation called thresholding. Thresholding is the most trivial image segmentation technique which allows separation of objects from the background (Fig. 2.11).

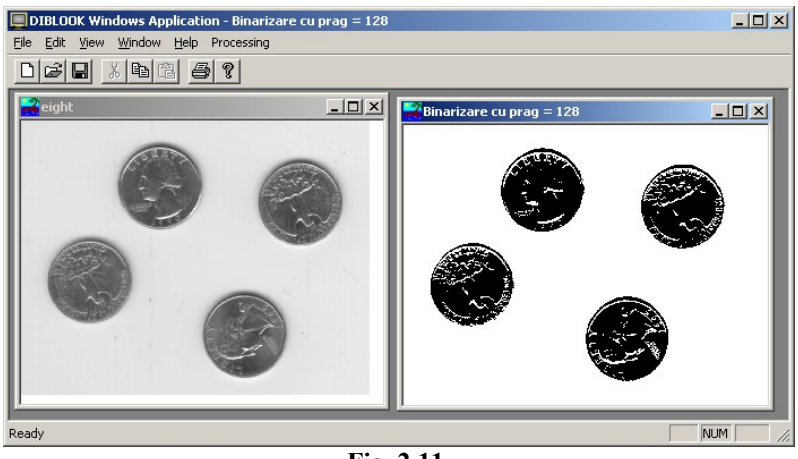


Fig. 2.11.

In this laboratory the thresholding with a fixed (arbitrary chosen) threshold value of an indexed (8 bits/pixel) grayscale image will be discussed. The thresholding can be performed by scanning the values of each pixel from the input image and replacing the corresponding pixel in the destination image using the following condition:

$$lpDst[i*w+j] = \begin{cases} 0 \quad (black) & , \quad if \quad lpSrc[i*w+j] < threshold \\ 255 \quad (white) & , \quad if \quad lpSrc[i*w+j] \geq threshold \end{cases} \tag{2.2}$$

The value of the threshold can be established inline the code (not recommended) or through a dialog box (recommended). The way in which a dialog resource and an *Edit Control* is created and used to get a value is similar as presented in section 2.5. The *Edit Control* should allow editing its content (not to be read-only (default), as shown in Fig. 2.12. The type of the variable used to get/store the value typed in the *Edit Control* can be a numerical one (BYTE) (Fig. 2.12).

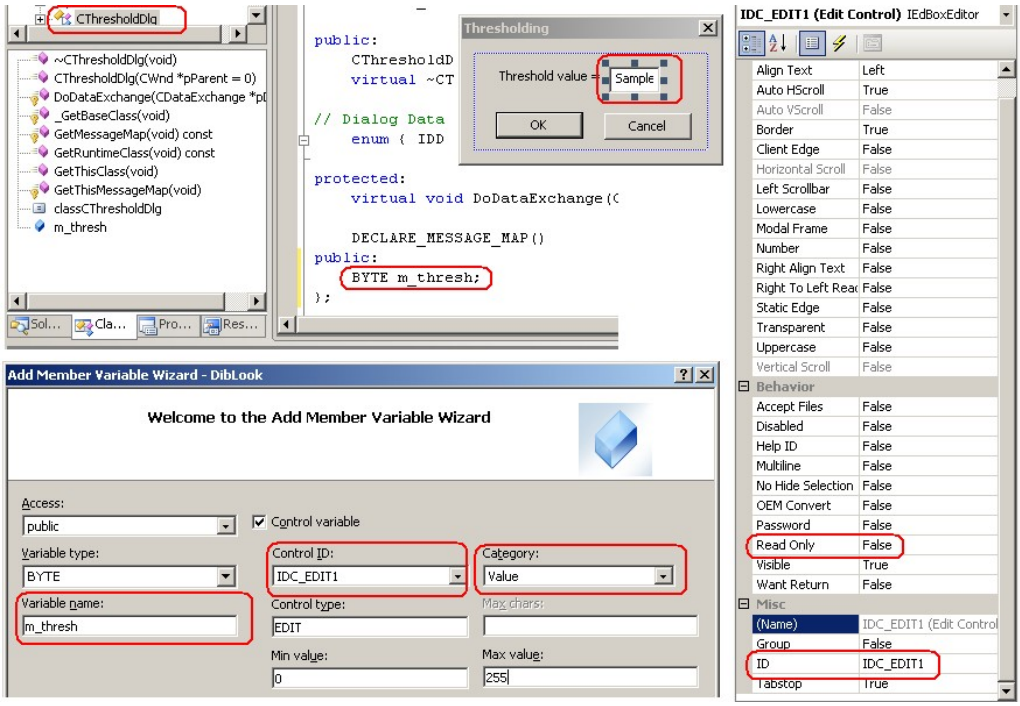


Fig. 2.12.

Sample code for getting the threshold value from the dialog:

```
void CDibView::OnProcessingBinarizarecupragarbitrar()
{
	BYTE threshold;
	//creates an instance (object) of the dialog class
	CThresholdDlg dlgThresh;

	if (dlgThresh.DoModal() == IDOK) {
		threshold=dlgThresh.m_thresh;
		BEGIN_PROCESSING();
		//  Go through the bitmap pixels and performs thresholding
		// ...
		CString buf;
		buf.Format(_TEXT("Threshold = %d"), threshold);
		END_PROCESSING (buf);
	}
}
```

2.7 Practical work

10. Add to the DIBLook framework a function for displaying (in a dialog box) the information from the bitmap header and the content of the LUT.
11. Add to the DIBLook framework a processing function for the *color* ⇒ *grayscale* conversion of a RGB24 images (24 bits/pixel), using (2.1).
12. Add to the DIBLook framework a processing function for the *color* ⇒ *grayscale* conversion of an indexed images (8 bits/pixel), using (2.1).
13. Add to the DIBLook framework a processing function which sorts the LUT of an indexed image, as described in section 2.3.2.
14. Compare the content of an unsorted LUT with a sorted one (using the grayscale image obtained from *Kids.bmp* color image).
15. Integrate functions from points 3 and 4 in a single one.
16. Add to the DIBLook framework a processing function for the *grayscale* ⇒ *black&white* conversion for indexed images (8 bits/pixel), using (2.2). Read the value of the threshold from an edit control of a dialog box. Test the thresholding operation with several/different threshold values on various grayscale images.
17. **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!!!**

2.8 References

[1] http://en.wikipedia.org/wiki/RGB_color_model
[2] http://en.wikipedia.org/wiki/Color_models
[3] http://msdn2.microsoft.com/en-us/library/ms779712(VS.85).aspx

3 Camera Calibration and Image Undistortion

3.1 Introduction

The purpose of this laboratory is to learn about the camera lens distortions, how to find the distortion parameters through the camera calibration process and as well as how to use these parameters to undistort images.

3.2 Perspective Projection

By using a *pinhole* camera model, a 3D point from the scene is projected into the image plane according to the following perspective transformation:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = A \begin{bmatrix} X_N \\ Y_N \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}}_{\text{Intrinsic Parameters}} \begin{bmatrix} X/Z \\ Y/Z \\ 1 \end{bmatrix} \tag{3.1}$$

Where:

- (u, v) are the coordinates of the point in the image plane.
- A is the *camera matrix* with the intrinsic parameters:
  - (f_x, f_y) – the focal distances along x and y axes,
  - (c_x, c_y) – the optical centers.
- (X, Y, Z) are the coordinates of a 3D point in the scene.
- (X_N, Y_N) are the normalized coordinates: $X_N = \frac{X}{Z}, Y_N = \frac{Y}{Z}$.

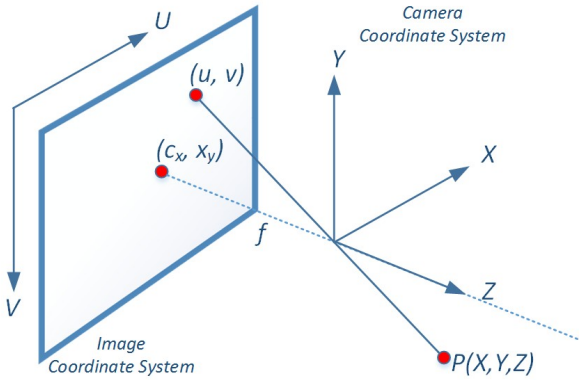


Fig. 3.1 The projection of a 3D point (X,Y,Z) onto the image plane at position (u,v).

For the sake of the simplicity the projection model will not include the matrix of extrinsic parameters (rotation and translation components that are used to bring the 3D points form the world coordinate system to the camera coordinate system). Therefore, we consider that all points from the scene are in the camera coordinate systems and there is no need for an extra rotation and translation of 3D points from the world coordinate system to the camera coordinate system (see lecture notes).

3.3 Lens Distortions

In reality, the camera lenses suffer from non-linear lens distortions, so that the captured points are slightly distorted. In other words, the projections of the rectilinear lines from the scene do not remain rectilinear in the image plane.

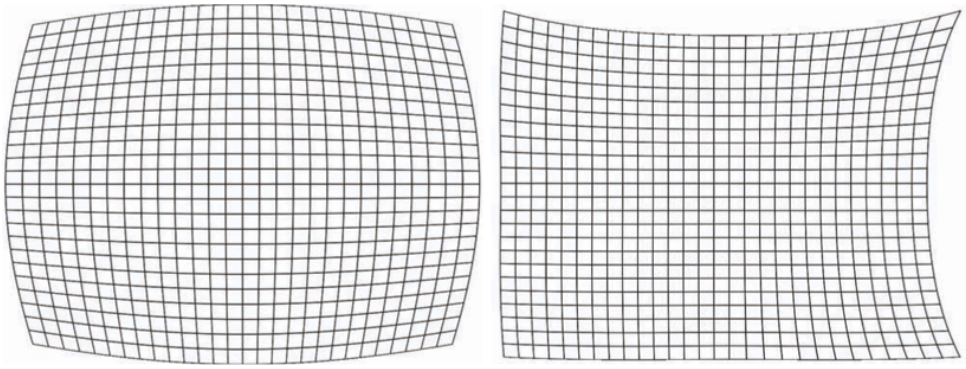


Fig. 3.2 Left: the radial distortion (barrel distortion). Right: a combination of a radial (pincushion distortion) and a tangential distortion [1].

The lens distortions can be divided into two main types:

- ***Radial distortions*:** In the case of the radial distortion, straight lines form the scene will appear curved in the image. This "bending" effect increases at once with the distance from the image center. Usually, the radial distortions can be classified as *barrel* or *pincushion* distortions (see Fig. 3.2). Having the coordinates of the normalized real points (X_N, Y_N), the radially distorted pixels (x_d, y_d) can be described by the following equations:

$$\begin{aligned} x_d &= X_N \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4 + ...) \\ y_d &= Y_N \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4 + ...) \end{aligned} \tag{3.2}$$

  where:

$$r^2 = X_N^2 + Y_N^2 \tag{3.3}$$

  $k_1, k_2, \ldots$ are the radial distortion coefficients. In practice, we can consider only the first two coefficients (k_1 and k_2).

- ***Tangential distortions*:** In the tangential distortion some points appear closer while other points may look farther than expected. This deviation is caused by the fact that the image plane is not perfectly parallel to the lens. The tangential distortion will deviate a point (X_N, Y_N) from its true position according to the following equations:

$$\begin{aligned} x_d &= X_N + 2p_1 \cdot X_N Y_N + p_2 \cdot (r^2 + 2X_N^2) \\ y_d &= Y_N + p_1 \cdot (r^2 + 2Y_N^2) + 2p_2 \cdot X_N Y_N \end{aligned} \tag{3.4}$$

  where p_1 and p_2 are the tangential distortion coefficients.

After including both types of lens distortions, a point from the scene will be affected according to:

$$\begin{aligned} x_d &= X_N \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4) + 2p_1 \cdot X_N Y_N + p_2 \cdot (r^2 + 2X_N^2) \\ y_d &= Y_N \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4) + Y_N + p_1 \cdot (r^2 + 2Y_N^2) + 2p_2 \cdot X_N Y_N \end{aligned} \tag{3.5}$$

The distorted point (x_d, y_d) from the scene will be projected onto the image as:

$$\begin{bmatrix} u_d \\ v_d \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_d \\ z_d \\ 1 \end{bmatrix} \tag{3.6}$$

The four distortion parameters and the camera matrix are calculated through the calibration process.

3.4 Camera calibration

The aim of the calibration step here is to find the four distortion coefficients (k_1, k_2, p_1, p_2), and the four intrinsic parameters f_x, f_y, c_x, c_y that describe the camera matrix. It must be noted that, the extrinsic parameters can also be estimated through the camera calibration process. The most common calibration method is to use some objects with a known geometry, such as a chessboard pattern. The algorithm finds a set of specific points on the pattern and their projection on the image. The camera matrix and the distortion parameters are estimated by using these two correspondent point sets. In order to get a higher precision several snapshots with the pattern in different positions and orientations should be taken (see Fig. 3.3).

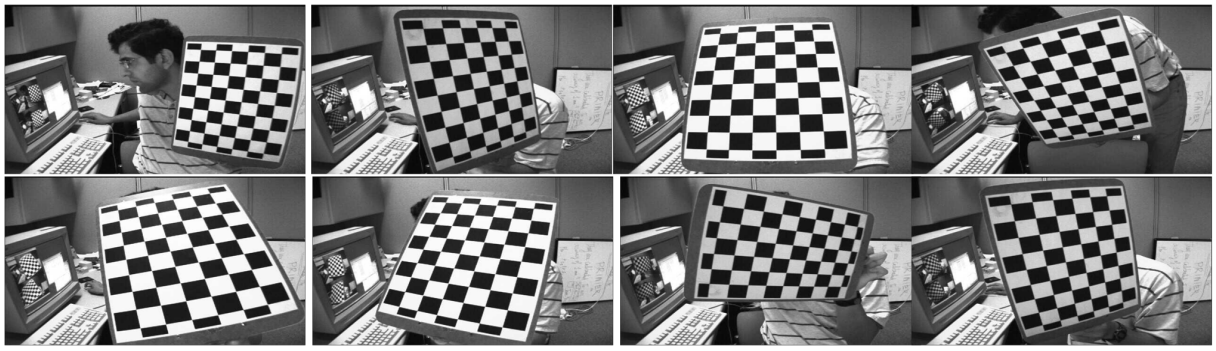


Fig. 3.3 The calibration process. Using several snapshots with a calibration chessboard with known geometry.

In order to simplify the process of getting the distortion parameters and the intrinsic parameters, we will use the *OpenCV* library and a modified version of an existing calibration code from the sample examples.

For integrating the calibration process, the following steps should be applied:

1. Download the Visual studio and OpenCV starter project `OpenCVApplication-VS2013.zip` (or use an existing one if you have one with the already integrated OpenCV libraries). Another solution would be to manually integrate the OpenCV libs and headers and to change the corresponding settings in the visual studio in order to get it to work properly.
2. Extract the .zip archive of the project into a local folder.
3. Download the `calib.zip`, extract it and copy its content directly into the project root folder. Therefore the project will be extended with the following files:
    a. `Images/CameraCalibration/` folder including 14 images, `left01.jpg – left14.jpg`, as in the Fig. 3.3.
    b. `calib_settings.xml` – the settings loaded by the calibration module such as the type of pattern used in the calibration images, the number of corners to be extracted, the path to the sequence.xml file containing the list of the calibration images, the square size (in our case we use a chessboard) etc.
    c. `sequence.xml` – an xml file enumerating the path to all the images to be used in the calibration.
    d. `camera_calibration.cpp` and `.h` files – the calibration code.
4. Add the new .cpp and .h files into the project: right click the project in the workspace window. Choose `Add→Existing Item` and select the `camera_calib.cpp` and `camera_calib.h` files.

5. Include the `"camera_calibration.h"` in the `#include` section in the `OpenCVApplication.cpp`.
6. In the file OpenCVApplication.cpp, main() function, create a new menu entry and a subsequent switch clause calling the new calibration and undistort function, similar to the following example:

```
int main(){
		...
		printf(" 8 - Image Calibration and Undistort\n");
		...
	switch (op){
		...
	  case 8:
		testCalibUndistort();
		break;
	  }
	...
}
```

The starting implementation for the `testCalibUndistort()`function is:

```
void testCalibUndistort(){

	Mat cameraMatrix, distCoeffs;  //The camera and distortion matrices –
	//to be estimated by the calib() function
	Settings s;
	bool isCalibrated = calib(cameraMatrix, distCoeffs, s);   //Estimate the
	//camera matrix and the distortion parameters

	const char ESC_KEY = 27;
	if (s.inputType == Settings::IMAGE_LIST && s.showUndistorsed)
	{
		Mat src, dst, rview, map1, map2;
		src = imread(s.imageList[0], 1);
		Size imageSize = src.size();

		for (int i = 0; i < (int)s.imageList.size(); i++)
		{
			src = imread(s.imageList[i], 1);
			if (src.empty())
				continue;

			dst = src.clone();

			// -----------------------TO DO - Your Code Here----------
			// ---------Implement the undistort algorithm------------
			 //my_undistort(src, dst, cameraMatrix, distCoeffs);
			//- your function
			// ------------------------------------------------------------

			imshow("Image View", dst);
			char c = (char)waitKey();
			if (c == ESC_KEY || c == 'q' || c == 'Q')
				break;
		}
	}
}
```

One of the key instructions above is the `calib(cameraMatrix, distCoeffs, s);` function which performs the calibration with the specified settings in the xml files and returns a vector of distortion coefficients `distCoeffs` and the camera matrix `cameraMatrix` with the

intrinsic parameters. These two structures will be used later in the Undistortion algorithm.

7. Compile. If there are compiling errors at this point, then it is more likely that the new cpp files were not added properly into the project or the header file `"camera_calibration.h"` was not included.

8. After running the application, the result with the detected corners should be similar to the one from the Fig. 3.4.

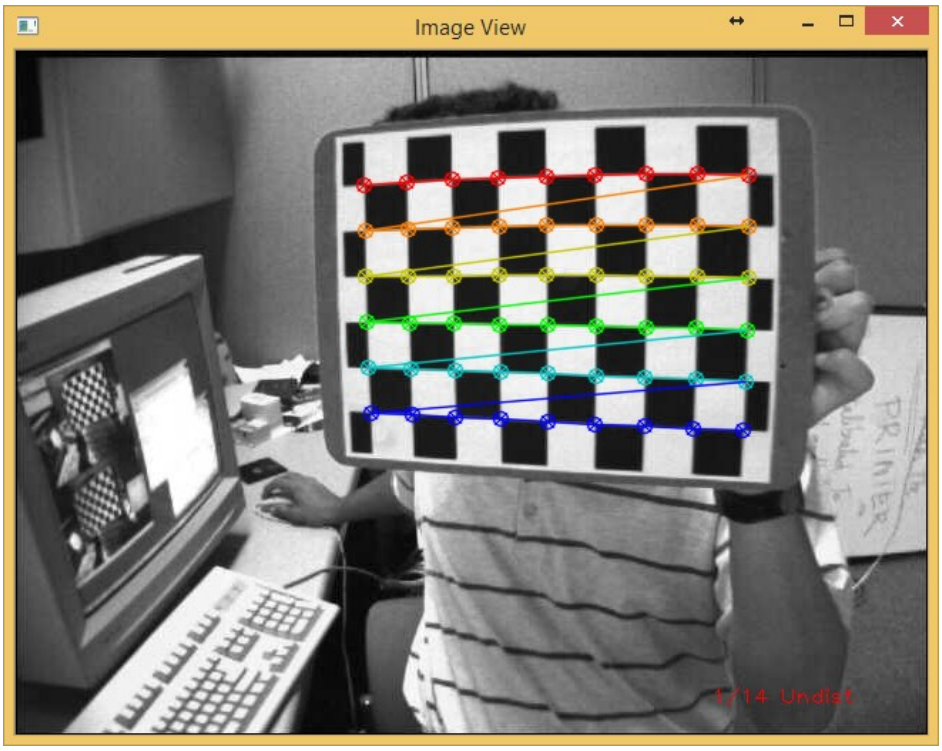


Fig. 3.4 The detected corners by the calibration step.

9. Once we obtained the camera matrix `cameraMatrix` and the distortion parameters `distCoeffs` then we can proceed with the next step – image undistortion

3.5 Image Undistortion

The undistortion process aims to correct the distorted image according to the obtained radial and tangential lens distortion parameters.

3.5.1 The Inverse Mapping Algorithm

Instead of finding the corrected position (u,v) of a given distorted point (u_d,v_d), an inverse mapping is used. Thus, for each destination point (from the undistorted image) the corresponding position (u_d,v_d), in the distorted image is calculated with the model described by the equation (3.5). The intensity value in the (u_d,v_d) is copied into the (u,v) position.

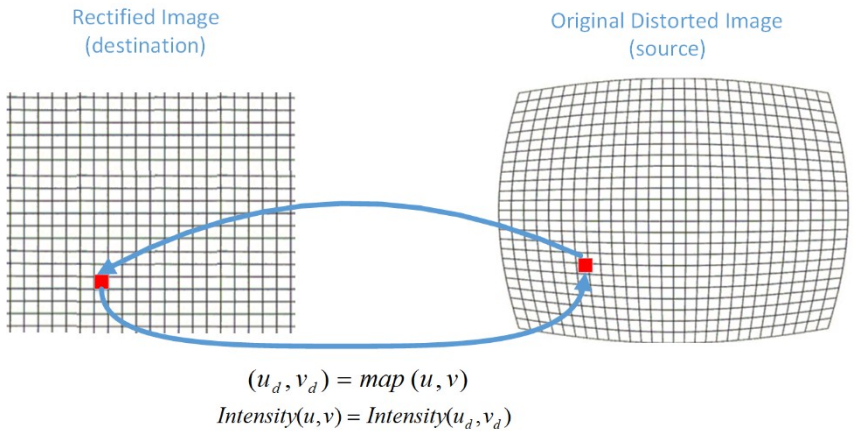


Fig. 3.5 Inverse mapping approach: Obtaining the intensity in the undistorted image (destination) by selecting the corresponding value from the distorted image (source).

The algorithm can be summarized into the following main steps:

For each pixel (u,v) from the allocated undistorted image space (which is going to be determined):

1. Compute the corresponding normalized coordinates (X_N,Y_N) in the camera coordinate system:
$$X_N = (u - c_x)/f_x$$
$$Y_N = (u - c_y)/f_y \tag{3.7}$$
2. Compute the distorted coordinates (x_d, y_d):
$$x_d = X_N \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4) + 2p_1 \cdot X_N Y_N + p_2 \cdot (r^2 + 2X_N^2)$$
$$y_d = Y_N \cdot (1 + k_1 \cdot r^2 + k_2 \cdot r^4) + Y_N + p_1 \cdot (r^2 + 2Y_N^2) + 2p_2 \cdot X_N Y_N \tag{3.8}$$
3. Project the obtained values into the distorted image space:
$$u_d = map_x(u,v) = x_d f_x + c_x$$
$$v_d = map_y(u,v) = y_d f_y + c_y \tag{3.9}$$
4. Copy the obtained intensity from the (u_d, v_d) position in the distorted image to the corresponding undistorted location (u,v).

Implement the above steps as a function:
`void my_undistort(Mat& src, Mat& dst, Mat& cameraMatrix, Mat& distCoeffs)` and call this function from the indicated row, in the `testCalibUndistort()`.

The four distortion coefficients and the intrinsic parameters may be recovered as in the following example:

```
const double* coef = distCoeffs.ptr<double>();

double k1 = coef[0];
double k2 = coef[1];
double p1 = coef[2];
double p2 = coef[3];

double cx = cameraMatrix.at<double>(0, 2);
double cy = cameraMatrix.at<double>(1, 2);
double fx = cameraMatrix.at<double>(0, 0);
double fy = cameraMatrix.at<double>(1, 1);
```

3.5.2 Bilinear Interpolation

Because usually the mapped distorted values are floating-points, a pixel intensity value at a fractional position needs be computed. A direct method would be to round the coordinates to the nearest integer values and to use the intensity from that location (nearest-neighbor interpolation). A better solution is to use a bilinear interpolation. Therefore the interpolated value can be estimated as:

$$\begin{aligned} u_0 &= \text{int}(u_d) \\ v_0 &= \text{int}(v_d) \\ u_1 &= u_0 + 1 \\ v_1 &= v_0 + 1 \\ I_0 &= S(u_0, v_0) \cdot (u_1 - u_d) + S(u_0, v_1) \cdot (u_d - u_0) \\ I_1 &= S(u_0, v_1) \cdot (u_1 - u_d) + S(u_1, v_1) \cdot (u_d - u_0) \\ D(u,v) &= I_0 \cdot (v_1 - v_d) + I_1 \cdot (v_d - v_0) \end{aligned} \tag{3.10}$$

where $S(u,v)$ represent the intensity in the original (distorted) image, while the $D(u,v)$ is the interpolated value copied in the undistorted image.

An example of the image undistortion is presented in Fig. 3.6

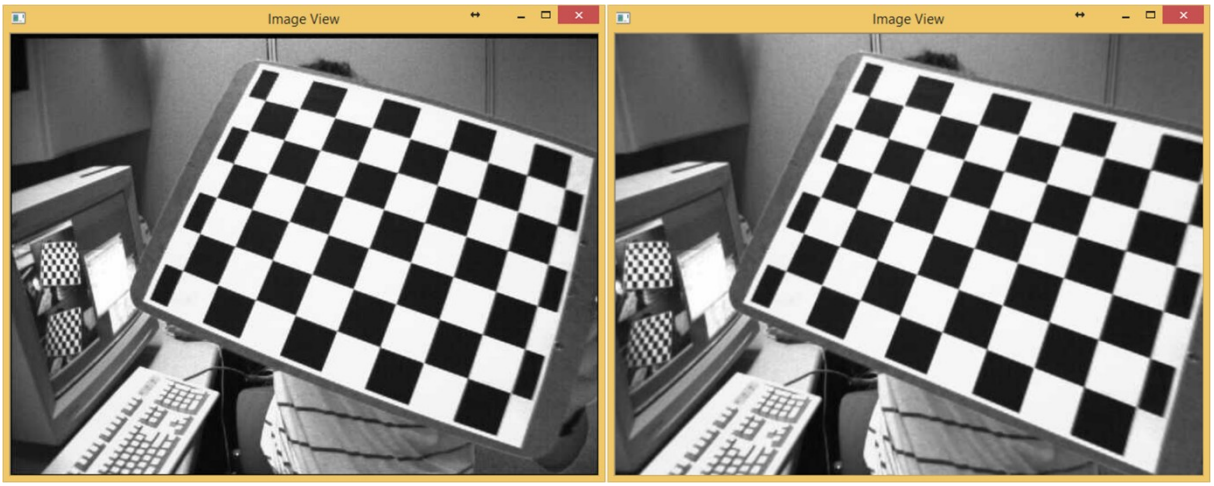


Fig. 3.6 Left: the image affected by lens distortions. Right: the undistorted result.

After testing the implemented image correction and interpolation methods described above, we can compare the obtained result with the existing undistortion algorithm provided by the OpenCV library using the cv:undistort() function:

```
cv::undistort(src, dst, cameraMatrix, distCoeffs);
```

3.6 Practical work

1. Integrate the OpenCV starter application and the provided sourse code files.
2. Follow the steps from section 3.4 to perform the camera callibration for finding the four distortion coefficients (k_1, k_2, p_1, p_2), and the four intrinsic parameters f_x, f_y, c_x, c_y.
3. Use the estimated parameters to implement the image undistortion algoritm, as decribed in section 3.5.1.
4. Implement the bilinear interpolation (section 3.5.2).

5. **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!!!**

3.7 References

[1] B. Sajadi, and M. Aditim *Markerless view-independent registration of multiple distorted projectors on extruded surfaces using an uncalibrated camera*, in IEEE Transactions on Visualization and Computer Graphics, Vol. 15, no. 6, pp. 1307-1316, 2009.
[2] R.C.Gonzales, R.E.Woods, *Digital Image Processing, 2-nd Edition,* Prentice Hall, 2002.
[3] OpenCV Documentation, *Camera Calibration and 3D Reconstruction – calib3d module*. http://docs.opencv.org/2.4/doc/tutorials/calib3d/camera_calibration/camera_calibration.html
[4] OpenCV source code on GITHUB: https://github.com/Itseez/opencv/find/master

4 The histogram of image intensity levels

4.1 Introduction

This laboratory work presents the concept of image histogram together with an algorithm for dividing the image histogram into multiple bins and reducing the image gray levels (gray levels quantization).

4.2 The histogram of intensity levels

Being given a grayscale image with the highest intensity value L (for an image with 8 bits/pixel $L=255$), the intensity (gray) level histogram is defined by a function $h(g)$ that has as value, for each intensity level $g \in [0 \ldots L]$, the number of pixels in the image or in the region of interest that have intensity equal to g.

$$h(g) = N_g \tag{4.1}$$

N_g – the number of pixels in the image or in the region of interest that have the intensity equal to g.

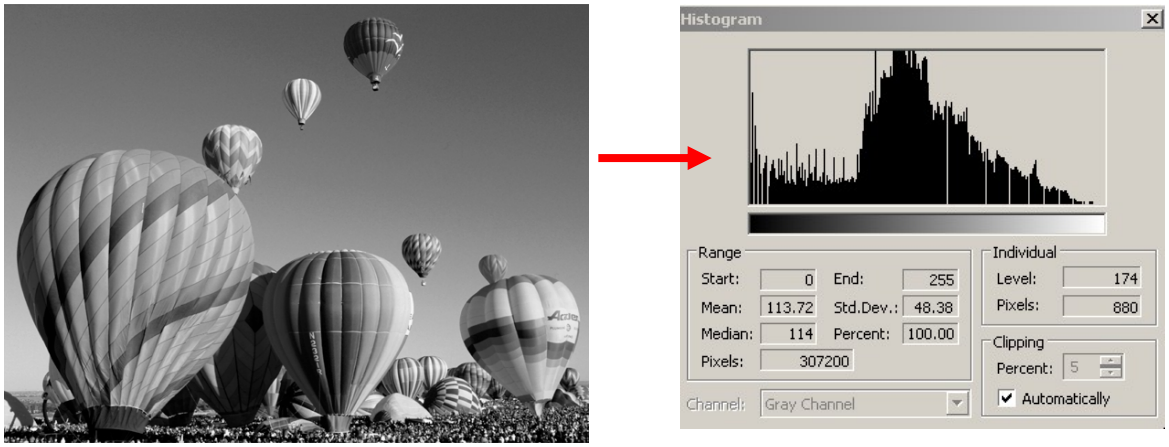


Fig. 4.1 Example: the histogram of a grayscale image

The function obtained by normalizing the histogram with the number of pixels in the image (in the ROI) is called the probability density function (PDF) of the intensity levels.

$$p(g) = \frac{h(g)}{M} \tag{4.2}$$

where:
$M = image\_height \times image\_width.$

PDF has the following properties:

$$\begin{cases} p(g) >= 0 \\ \int_{-\infty}^{\infty} p(g)dg = 1, \quad \sum_{g=0}^{L} \frac{h(g)}{M} = \frac{M}{M} = 1 \end{cases} \tag{4.3}$$

4.3 Application: Multilevel thresholding

This algorithm determines multiple thresholds for reducing the number of image intensity (gray) levels. Its first step is to determine the histogram maxima. Then, each gray level is assigned to the closest maximum.

The following steps must be performed in order to determine the histogram maxima:

1. Normalize the histogram (transform it into a PDF)
2. Choose a window width *2*WH+1* (a good value for WH is 5)
3. Choose a threshold *TH* (a good value is 0.0003)
4. For each position (middle of the window) *k* from *0+WH* to *255-WH*
   - Compute the average *v* of normalized histogram values in the interval [*k-WH*, *k+WH*]. Remark: the value *v* is the average of *2*WH+1* values
   - If *PDF*[*k*]>*v+TH* and *PDF*[*k*] is greater or equal than all *PDF* values in the interval [*k-WH*, *k+WH*] then *k* corresponds to a histogram maximum. Store it and then continue from the next position.
5. Insert 0 at the beginning of the maxima position list and 255 at the end (this allows the colors black and white to be represented exactly).

The second step is thresholding. Thresholds are located at equal distances between the maxima. Therefore the algorithm for thresholding is simply: assign to each pixel the color value of the nearest histogram maximum.

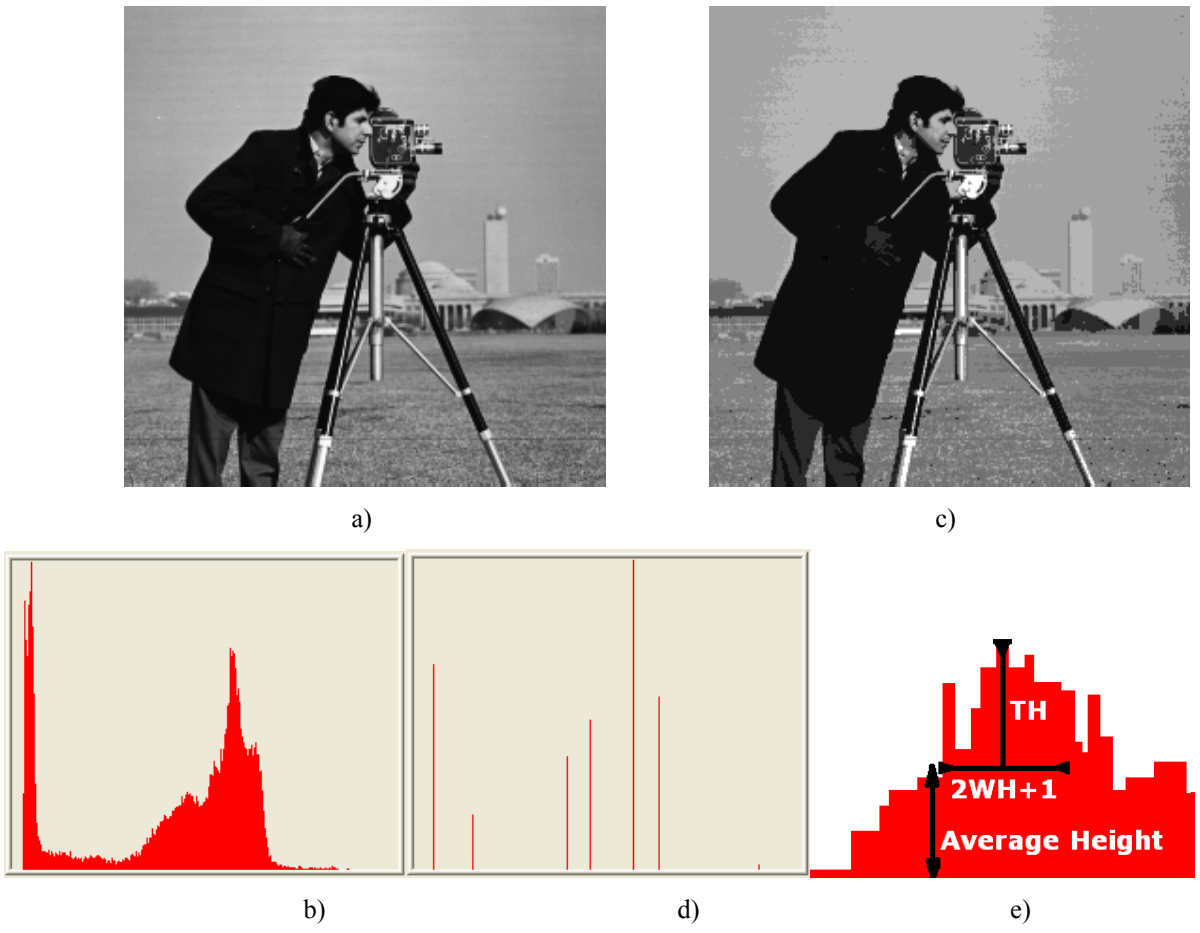


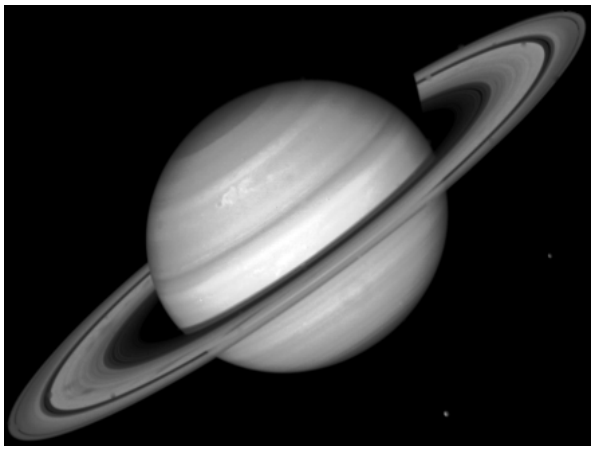
Fig. 4.2 a) The initial image; b) The histogram of the intitial image; c) The obtained multilevel thresholded image; d) The histogram of the multilevel thresholded image; e) The histogram maxima computation algorithm

As seen in the Fig. 4.3 b, the results are visually unacceptable when the number of gray levels is small. To obtain more visually acceptable a dithering algorithm can be applied. Such an algorithm spreads the quantization error to multiple pixels. An example of a dithering algorithm is the Floyd-Steinberg algorithm:

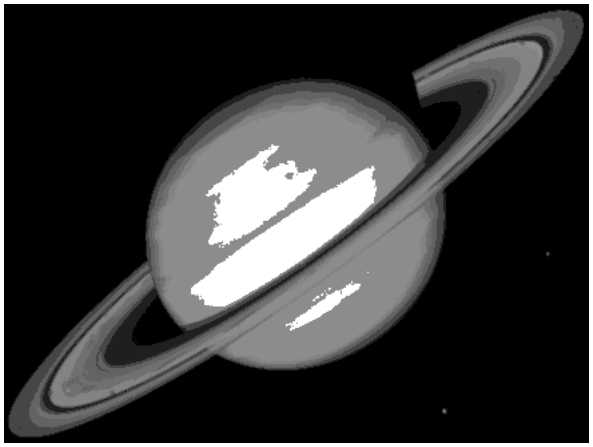
```
for each y from bottom to top
   for each x from left to right
      oldpixel := pixel(x,y)
      newpixel := find_closest_histogram_maximum(oldpixel)
      pixel(x,y) := newpixel
      error := oldpixel - newpixel
      pixel(x+1,y) := pixel(x+1,y) + 7*error /16
      pixel(x-1,y+1) := pixel(x-1,y+1) + 3*error/16
      pixel(x,y+1) := pixel(x,y+1) + 5*error/16
      pixel(x+1,y+1) := pixel(x+1,y+1) + error/16
```

This algorithm computes the quantization error and spreads it to the neighboring pixels according to the following fractions matrix (**X** = current pixel's location):

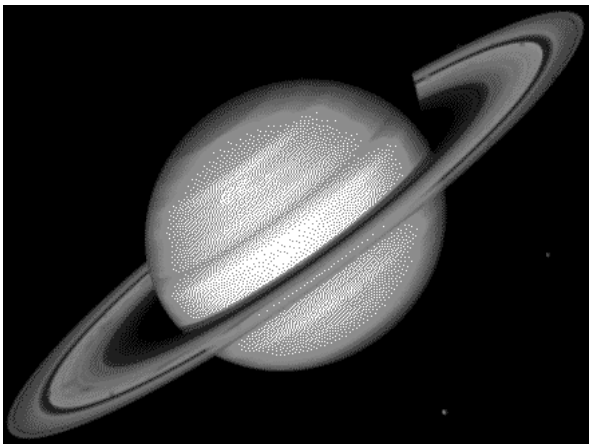
3/16	3/16	3/16
0	**X**	7/16
0	0	0



a)



b)



c)

Fig. 4.3 a) The initial image; b) The obtained multilevel thresholded image; c) Dithering on the initial image using the Floyd-Steinberg algorithm

4.4 Implementation details: histogram display in a dialog box

4.4.1 Option 1: Displaying the histogram in a Picture control

The histogram will be displayed using a dialog window (*Dialog Box*). It is used a control of type *Picture* inside the dialog box for displaying the histogram.
The control of type *Picture* will have a rectangular shape and implicitly a certain width and height. Its width must be at least $L+1$ pixels, where L equals the highest intensity value in the image for which the histogram is computed (L=255 for an 8 bits/pixel grayscale image). The components of the histogram will be depicted in the form of vertical bars of height equal to the number of pixels corresponding to each intensity value. The vertical bars corresponding to the levels of intensity $0...L$ will be displayed in order, from left to right.

Remark: in some cases the number of pixels having certain intensity in the histogram may be greater than the height of the *Picture* control component. For avoiding those cases, the displayed histogram will be scaled with a value (each value in the histogram array will be divided by the maximum value in the histogram and then it will be multiplied with the height of the *Picture* control component). The scaling step will be performed only if the maximum value in the histogram array is greater than the height of the display control.

1. Insert a new dialog box (check the laboratory work 2!).
2. In the dialog box, add a control of type *Picture* in which the histogram will be displayed. Modify the properties of the *Picture* type control (right click and *Properties*) Fig. 4.4):
    a. Modify its ID to IDC_HISTOGRAM
    b. In the *Properties* section, for appearance, set "*True"* the values for *Client Edge* and *Modal Frame*.

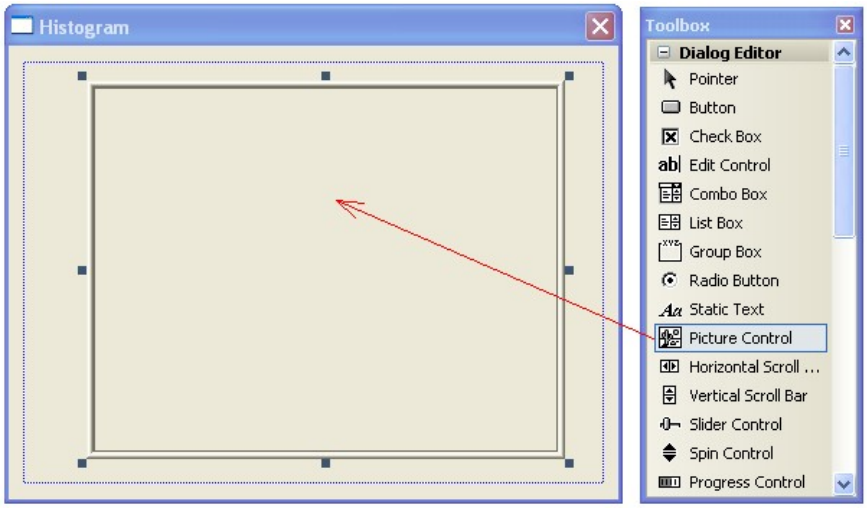


Fig. 4.4 The design of the dialog box for displaying the histogram

3. Create a new class for the previously created *Dialog Box* (right click and *Add Class…*). Name the class *CDlgHistogram* and then close the *Wizard* and the created dialog window.
4. Create a new class for managing the control in which the histogram is displayed. This is done by accessing the main menu and following the steps bellow (Fig. 4.5):
    a. *Project -> Add class…*
    b. Choose the *Visual C++ -> MFC* category and then the *MFC Class* template
    c. Name the class *CHistogram* and choose its base class to be *CStatic*.

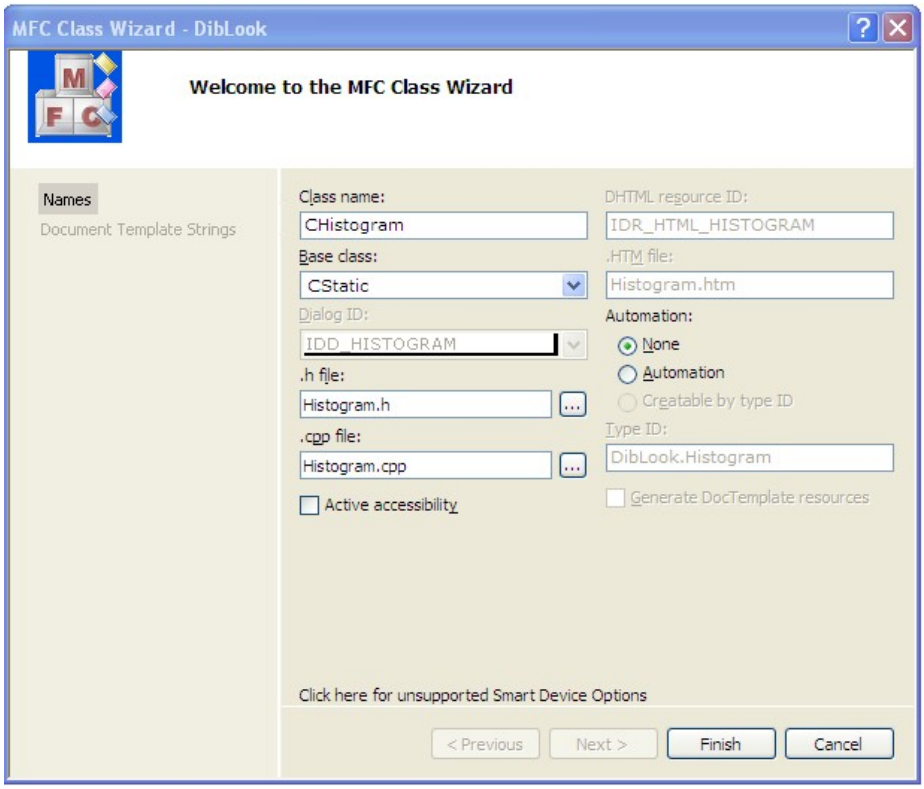


Fig. 4.5 The creation of the class *CHistogram* for the control used to display the histogram

5. For IDC_HISTOGRAM attach the variable *m_Histogram* of category *Control* and type *CHistogram* (Pay attention: the file *CDlgHistogram.h* must include the header *Histogram.h*) (Fig. 4.6)

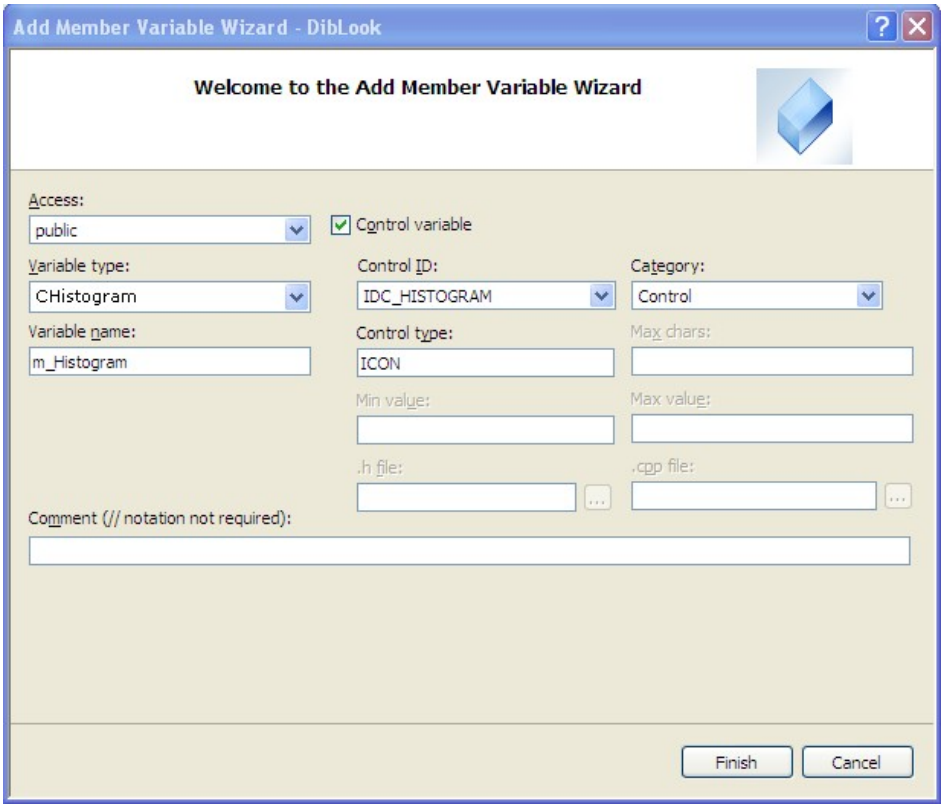


Fig.4.6 Adding a member variable *m_Histogram* for the control in which the histogram will be displayed

6. Attach a handler for displaying the histogram:
   a. In the tabulator *Class View* perform a right click on the class *CHistogram* and then *Properties…*
   b. In the *Properties* window choose *Messages* section
   c. On the message *WM_PAINT* add the method *OnPaint* (*<Add> OnPaint*) using the combo-box

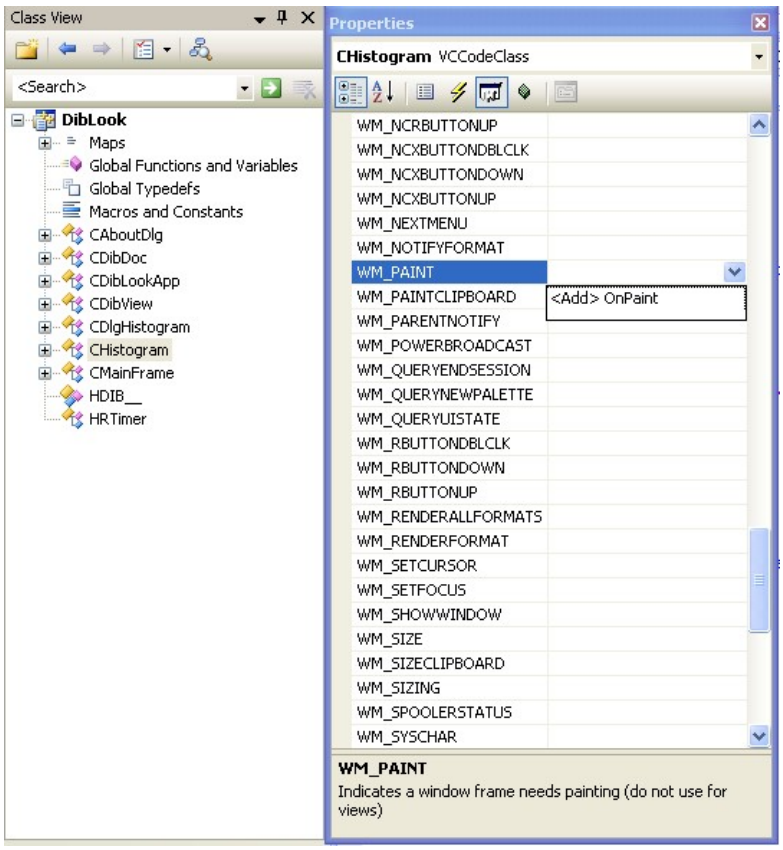


Fig.4.7 Adding the method *OnPaint* attached to the message *WM_PAINT* for the class *CHistogram*

7. In the header file *Histogram.h* at the *public* section define the integer array *values*[*256*] that represents the histogram to be displayed (possibly scaled before being displayed)

```
class CHistogram : public CStatic
{
// Construction
public:
CHistogram();

// Attributes
public:
	int values[256];

	// There are no changes in the following lines
	.
	.
	.
	}
```

8. In the source file *Histogram.cpp*, rewrite the *OnPaint()* method for displaying (and possibly scaling) the histogram defined by the input array *values*[*256*]:

```
void CHistogram::OnPaint()
```

```
{
    CPaintDC dc(this); // device context for display
    CPen pen(PS_SOLID, 1, RGB(255,0,0)); // define the display pen-
                                                    for red color
    CPen *pTempPen=dc.SelectObject(&pen); // select the display pen
    CRect rect;
    GetClientRect(rect); // get the available display rectangular area
    int height=rect.Height(); // height of the display area
    int width=rect.Width(); // width of the display area

    // find the maximum in the array values[256]
    int i;
    int maxValue=0;
    for (i=0;i<256;i++)
        if (values[i]>maxValue)
              maxValue=values[i];

    // check if scaling is necessary
    double scaleFactor=1.0;
    if (maxValue>=height)
    {
          // scaling is necessary
          scaleFactor=(double)height/maxValue;
    }

    // display the histogram in the form of vertical bars
    for (i=0;i<256;i++)
    {
          // find the length of the line
          int lengthLine=(int)(scaleFactor*values[i]);
          //display the line
          dc.MoveTo(i,height);
          dc.LineTo(i,height-lengthLine);
    }

    dc.SelectObject(pTempPen); // restore the display pen
}
```

9. Display the computed histogram in the created *Dialog-Box*:
    a. Add a new processing menu for computing the histogram and add a method associated to it : *OnDisplayHistogram()*
    b. Include the file *DlgHistogram.h* in the file *DibView.cpp*
    c. The method *OnDisplayHistogram()* attached to the click event on the processing menu will be defined as follows:

```
void CDibView::OnDisplayHistogram()
{
    BEGIN_SOURCE_PROCESSING;

    int histValues[256];
    float FDPValues[256];

    // write the code for computing the histogram and store it in the array
    of int, histValues[256]
    // write the code for computing the PDF and store it in the array of
    double FDPValues[256]

    // instantiate a dialog box for display and associate the histogram
    CDlgHistogram dlg;
    memcpy(dlg.m_Histogram.values,histValues,sizeof(histValues));
    // display the dialog box
    dlg.DoModal();

    END_SOURCE_PROCESSING;
 }
```

4.4.2 Option 2: Displaying the histogram directly in the user area of a dialog box

Create a new dialog box and attach it the class *CDlgGrayStatistics.* Displaying the histogram directly in the user area of a dialog box as in Fif. 4.8 can be done by defining an *OnPaint* function corresponding to the WM_PAINT message associated to the dialog box/window (Fig. 4.9) as in the example bellow:

```
#define LEFT 10
#define HEIGHT 100
#define BOTTOM 200

void CDlgGrayStatistics::OnPaint()
{
	CPaintDC dc(this); // device context for painting
	POINT pct;
	for (int g=0; g<256; g++) {
		pct.x=LEFT + g;
		int N=(float)(m_hist[g]*HEIGHT)/(float)m_maxhist;
		for (int n=0; n <N ; n++) {
			pct.y=BOTTOM-n;
			dc.SetPixel(pct,RGB(0,0,0));
		}
	}
}
```

Where: `int *m_hist; int m_maxhist;` are public members of the dialog class and are initialized in your CDibView:: processing function …

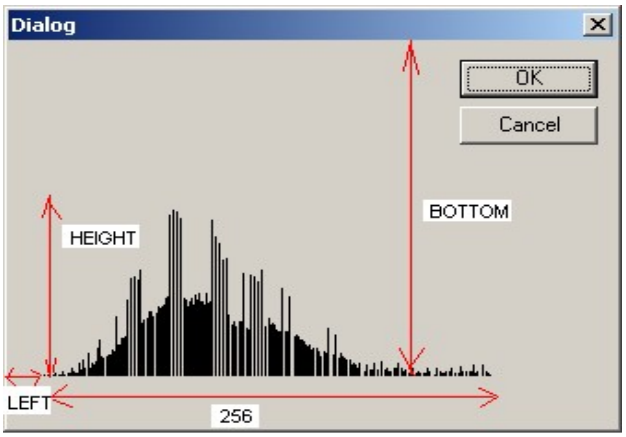


Fig. 4.8 Example of displaying the histogram directly in the user area of a dialog box

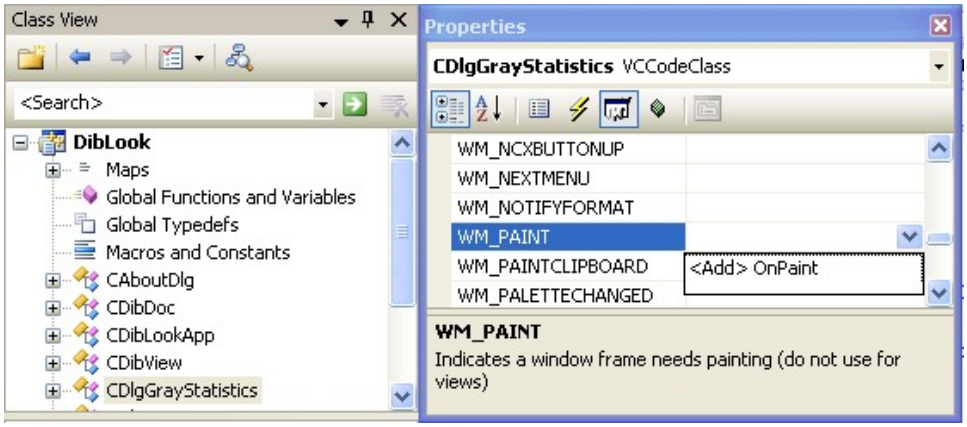


Fig. 4.9 Associating the *OnPaint* function to the WM_PAINT message of the dialog box

4.5 Practical work

1. Compute the histogram for a given grayscale image with 8 bits/pixel (in an array of integers having dimension 256) and the PDF (in a vector of float of dimension 256). Display the computed histogram by choosing a method presented in the section 4.4.
2. Implement the multilevel thresholding algorithm.
3. Enhance the multilevel thresholding algorithm using the Floyd-Steinberg dithering.
4. **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.**

4.6 References

[1]. R.C.Gonzales, R.E.Woods, *Digital Image Processing. 2-nd Edition*, Prentice Hall, 2002.
[2]. Floyd-Steinberg algorithm, http://en.wikipedia.org/wiki/Floyd-Steinberg_dithering

5 Geometrical features of binary objects

5.1 Introduction

This lab work presents some important geometric properties of binary images and the algorithms used for computing them. The properties described are: the area, the center of mass, the elongation axis, the perimeter, the thinness ratio, the aspect ratio and the projections of the binary image.

5.2 Theoretical considerations

After applying segmentation and labeling algorithms to images we obtain a new image in which each object can be referenced separately.
An object 'i' is described in the image by the function:

$$I_i(r,c) = \begin{cases} 1, & \text{if } I(r,c) \in \text{object labeled 'i'} \\ 0 & \text{otherwise} \end{cases}$$
$$\text{where } r \in [0...Height-1] \text{ and } c \in [0...Width-1]$$

The geometric properties of objects can be classified into two categories:

- position and orientation properties: the center of mass, the area, the perimeter, the elongation axis
- shape properties: aspect ratio, thinness ratio, Euler's number, the projections, the Feret diameters of the objects

5.2.1 Area

$$A_i = \sum_{r=0}^{H-1}\sum_{c=0}^{W-1} I_i(r,c) \tag{5.1}$$

The area A_i is measured in pixels and it indicates the relative size of the object.

5.2.2 The center of mass

$$\overline{r_i} = \frac{1}{A_i}\sum_{r=0}^{H-1}\sum_{c=0}^{W-1} rI_i(r,c) \tag{5.2}$$

$$\overline{c_i} = \frac{1}{A_i}\sum_{r=0}^{H-1}\sum_{c=0}^{W-1} cI_i(r,c) \tag{5.3}$$

The equations above correspond to the row and column where the center of mass is located. This attribute helps us locate the object in a bi-dimensional image.

5.2.3 The axis of elongation (the axis of least second order moment)

$$\tan(2\varphi_i) = \frac{2\sum_{r=0}^{H-1}\sum_{c=0}^{W-1}(r-\overline{r_i})(c-\overline{c_i})I_i(r,c)}{\sum_{r=0}^{H-1}\sum_{c=0}^{W-1}(c-\overline{c_i})^2 I_i(r,c) - \sum_{r=0}^{H-1}\sum_{c=0}^{W-1}(r-\overline{r_i})^2 I_i(r,c)} \tag{5.4}$$

If both the nominator and the denominator of the above equation are equal to zero, than the object has a circular symmetry, and any line that passes through the center of mass is a symmetry axis.

For finding the direction of the line (the angle) one must apply the arctangent function. The arctangent is defined on the interval $(-\infty, +\infty)$ and it takes values in the interval $(-\pi/2, \pi/2)$. The evaluation of the arctangent becomes unstable when the denominator of the fraction tends to zero.

The signs of the numerator and of the denominator are important for determining the right quadrant in which the result lays. The arctangent function does not make the difference between directions that are opposed. For this reason the usage of the function "atan2" is suggested. The "atan2" function has as arguments the numerator and the denominator of such fraction, and it returns a result in the interval $(-\pi, \pi)$.

The axis of elongation gives information about how the object is positioned in the field of view, that is its orientation. The axis corresponds to the direction in which the object (seen as a plane surface of constant width) can rotate most easily (has a minimum kinetic moment).

5.2.4 The perimeter

The perimeter of the object helps us determine the position of the object in space and it also gives information about the shape of the object. The perimeter can be computed by counting the number of pixels on the contour (pixels of value 1 and having at least one neighbor pixel of value 0).

A first approach to contour detection is the scanning of the image, line by line and counting the number of pixels in the object that satisfy the condition mentioned above. A main disadvantage of this method is that we cannot distinguish the exterior contour from the interior contours (if they exist they are generated by the holes in the object). As the pixels of digital images represent distributions on a rectangular raster, the length of curves and oblique lines in the image cannot be correctly estimated by counting the pixels. A first correction is given by the multiplication by $\pi/4$ of the perimeter that resulted in the previous algorithm. There are other methods for length correction. These methods take into account the type of neighborhood used (4 neighbors, 8 neighbors etc).

Another method for detecting the contour of an object involves the usage of an existing algorithm for edge detection, the thinning of the edges until they become 1 pixel thick and in the end the counting of the resulted edge pixels.

Methods of type "chain-codes" represent complex methods for contour detection and offer a high accuracy.

5.2.5 The thinness ratio (circularity)

$$T = 4\pi\left(\frac{A}{P^2}\right) \tag{5.5}$$

The function above has the maximum value equal to 1, and for this value we obtain a circle. The thinness ratio is used for determining how "round" an object is. If the value of T is close to 1, the object tends to be round.
The value of the thinness ratio also offers information on how regular an object is. The objects that have a regular contour have a greater value of T than the objects of irregular contours. The value 1/T is called irregularity factor of the object (or compactness factor).

5.2.6 The aspect ratio

This property is found by scanning the image and keeping the minimum and maximum values of the lines and columns that form the rectangle circumscribed to the object.

$$R = \frac{c_{\max} - c_{\min} + 1}{r_{\max} - r_{\min} + 1} \tag{5.6}$$

5.2.7 The projections of the binary object

The projections give information about the shape of the object. The horizontal projection equals the sum of pixels computed on each line of the image, and the vertical projection is given by the sum of the pixels on the columns.

$$h_i(r) = \sum_{c=0}^{W-1} I_i(r,c) \tag{5.7}$$

$$v_i(c) = \sum_{r=0}^{H-1} I_i(r,c) \tag{5.8}$$

The projections are used in applications of text recognition in which the interest object can be normalized.

5.3 Implementation details

In order to distinguish between the various objects present in an image, we will suppose that each one of them is painted using a different color. These colors may be the result of a previous labeling step, or may be generated manually (see Fig. 5.1).

There are various approaches for implementing the geometrical properties extractors. A simple approach is to compute them for all objects in an image at once. There are at most 255 objects plus background in the image format described above, each one having a different color index.

The second approach is to use the mouse to select an object. The user should position the mouse pointer over a pixel belonging to the desired object and double-click on it. In response to this action, you should display a message box containing the values of the desired features.

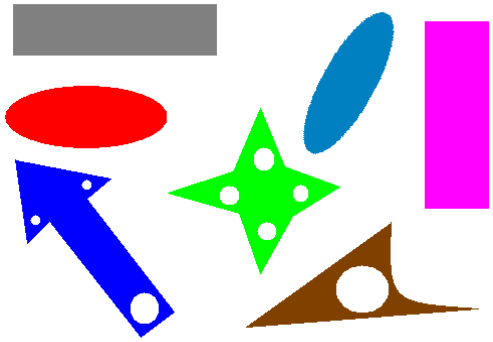


Fig. 5.1 Example of a labeled image on which the algorithms described here could be tested

In order to add a handler for the double click event you must follow these steps (see Fig. 5.2):

1. On the workspace window select the Class View tab;
2. Right click on the CDibView class and choose Properties…;
3. In the Properties window choose the Messages section;
4. On the WM_LBUTTONDBLCLK add the method OnLButtonDblClk (<Add> OnLButtonDblClk) by using the combo-box. The wizard will generate a message handler method, receiving as a parameter the coordinates of the mouse pointer, relative to the view's at the time when the double-click occurred.

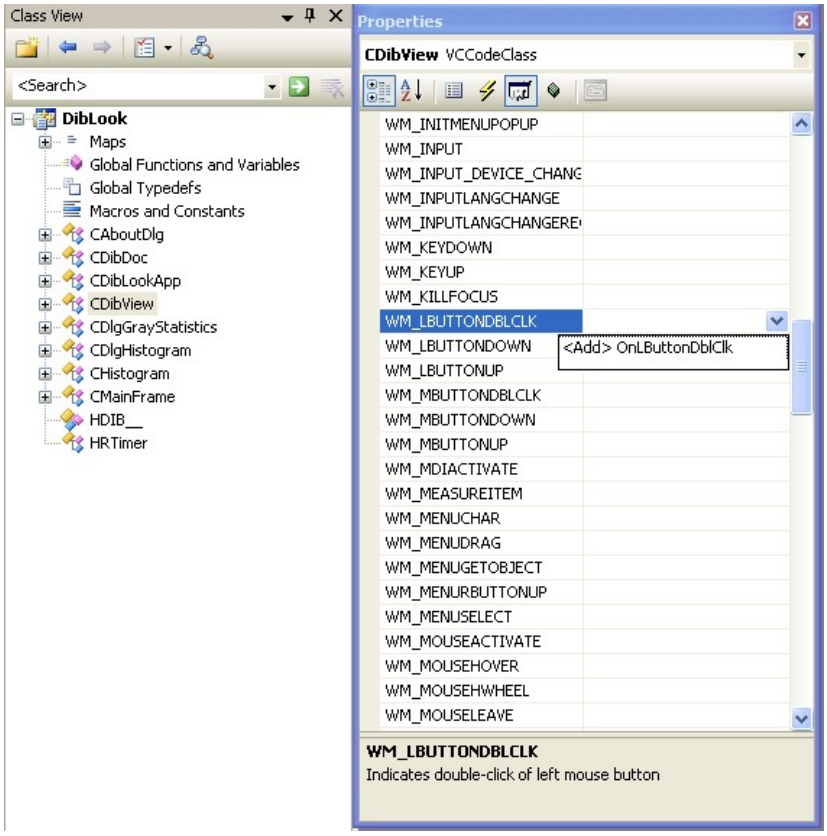


Fig. 5.2 Adding a new message handler for the double-click using the left mouse button in CDibView class

The following code presents a stub mouse event handler that computes the coordinates of the mouse click in the image, and displays a message box with the coordinates and the color index of the pixel over which the click occurred.

```
void CDibView::OnLButtonDblClk(UINT nFlags, CPoint point)
{
	BEGIN_SOURCE_PROCESSING;

	//obtain the scroll position (because of scroll bars' positions
	//the coordinates may be shifted) and adjust the position
	CPoint pos = GetScrollPosition()+point;

	//point contains the window's client area coordinates
	//the y axis is inverted because of the way bitmaps
	//are represented in memory
	int x = pos.x;
	int y = dwHeight-pos.y-1;

     //test if the position is inside the image
	if (x>0 && x<dwWidth && y>0 && y<dwHeight)
	{
		//prepare a CString for formating the output message
		CString info;
		info.Format(_TEXT("x=%d, y=%d, color=%d"), x, y, lpSrc[y*w+x]);
		AfxMessageBox(info);
	}

	END_SOURCE_PROCESSING;

	//call the superclass' method
	CScrollView::OnLButtonDblClk(nFlags, point);
}
```

A third approach is to either select an individual object or compute the features for all objects and display the results directly on the destination image. An easy way to display text and graphics on an image is to use the Windows GDI (Graphics Device Interface) functions.

Each graphical operation in Windows must be accomplished through a DC (Device Context) object. This object holds such data as the device driver that performs the drawing (the driver for the graphics card, printer, memory device), the surface on which the drawing is performed (the main display surface, a back surface, a surface located in main memory), the current drawing pen (color, line width) the current brush (for filling surfaces) and so on.

In order to draw on a bitmap, the bitmap must be “selected” in the DC, so that all subsequent drawing will be done over the image. The device independent bitmaps (DIBs) used in DIBView cannot be selected directly in a DC. In order to cope with this problem, a device dependent bitmap (DDB) must be created, and the data from the source image copied to it. Next, the DDB is selected in a memory DC and drawing is performed. Finally, the pixels in the DDB are copied back to the destination DIB. The following code performs all the above steps. It also displays a line and a text at the coordinates where a mouse double click occurred.

```
void CDibView::OnLButtonDblClk(UINT nFlags, CPoint point)
{
	BEGIN_PROCESSING();
```

```
        CDC dc; //memory DC
        dc.CreateCompatibleDC(0); //create it compatible with the screen

        CBitmap ddBitmap;//to hold a device dependent bitmap compatible with
                         //the screen

        //create a DDB, compatible with the screen
        //and initialize it with the data from the source DIB
        HBITMAP hDDBitmap =
       CreateDIBitmap(::GetDC(0), &((LPBITMAPINFO)lpS)->bmiHeader, CBM_INIT,
       lpSrc, (LPBITMAPINFO)lpS, DIB_RGB_COLORS);

        //attach the handle to the CBitmap object
        ddBitmap.Attach(hDDBitmap);

        //select the DDB into the memory DC
        //so that all drawing will be performed on the DDB
        CBitmap* pTempBmp = dc.SelectObject(&ddBitmap);

        //from this point onward, all drawing done using the DC object
        //will be made on the DDB
        //obtain the scroll position (because of scroll bars' positions
        //the coordinates may be shifted) and adjust the position
        CPoint pos = GetScrollPosition()+point;

        //create a green pen for drawing
        CPen pen(PS_SOLID, 1, RGB(0,255,0));

        //select the pen on the device context
        CPen *pTempPen = dc.SelectObject(&pen);
       //draw a text
        dc.TextOut(pos.x,pos.y, "test");
        //and a line
        dc.MoveTo(pos.x,pos.y);
        dc.LineTo(pos.x, pos.y-20);

        //select back the old pen
        dc.SelectObject(pTempPen);
        //and the old bitmap
        dc.SelectObject(pTempBmp);

        //copy the pixel data from the device dependent bitmap
        //to the destination DIB
        GetDIBits(dc.m_hDC, ddBitmap, 0, dwHeight, lpDst, (LPBITMAPINFO)lpD,
        DIB_RGB_COLORS);

        END_PROCESSING("line");
}
```

5.4 Practical work

1. For each individual object in a labeled image compute the object's area, center of mass, axis of elongation, perimeter, thinness ratio and aspect ratio. For displaying the results, you can chose among the following two options (presented in section 5.3):
    a. display the geometrical features of all objects in a dialog box;
    b. display the geometrical features of each object in a MessageBox when the area of the object is double-clicked.

2. Display objects' axes of elongation on the destination image using the GDI drawing functions.
3. Compute and display the objects' projections.
4. Display objects' mass centers and areas on the destination image using the GDI drawing functions.
5. **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!!!**

5.5 References

[1]. Umbaugh Scot E, *Computer Vision and Image Processing*, Prentice Hall, NJ, 1998, ISBN 0-13-264599-8

6 Binary objects labeling

6.1 Introduction

In this laboratory an object labeling algorithm which allows you to label distinct objects from a binary (black&white) image is presented. This algorithm is useful for the separation of distinct objects for further analyses/measurements applied on each individual object.

6.2 Theoretical considerations

In order to extract specific features of the objects from a digital image, it is necessary to perform a segmentation process to the original image. As a result of the segmentation operation, the obtained image will contain well-differentiated objects. The purpose of the labeling process is to assign to each distinct object a unique label (integer number). In the following, a fast labeling algorithm (which scans the image pixels of a binary image only once) will be presented.

Algorithm steps

1. Labeling the pixels from the source image, using a single image scan, and establishing the equivalent label pairs.
2. Establishing equivalence labeling classes.
3. Updating operation needed to replace each pixel's label with the label of its equivalence class.

Step 1. Labeling the pixels from the source image and establishing the equivalent label pairs.

The labeling process needs first to define the type of connectivity used. Then the image is scanned (line by line, in top-down and left-right order) and connected pixels are labeled with the same label. The presented algorithm uses a 5-conectivity (Fig. 6.1), in which the current pixel is denoted with 'X':

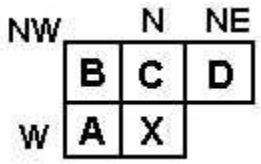


Fig. 6.1 The 5-conectivity used.

The algorithm for the general case is presented in Fig. 6.2. We consider pixels having a value different from "0" belonging to the background (in 8 bits/pixels bitmap images, object points are black – intensity value =0). In Fig. 6.2 the following notations are used:

- X_Label, A_Label, B_Label, C_Label, D_Label are the labels assigned to pixels X, A, B, C and D respectively (see Fig. 6.1);
- NewPair(Label1,Label2) is a function which appends pair (Label1,Label2) in the list of the equivalent label pairs.
- NewLabel is a function used to generate a new label by incrementing the last assigned label with 1 (the first generated new label is 1).

Observation: The pixels labels of the source image are kept in an integer matrix having the same dimensions as the source image. In the labels' matrix, the background pixels are labeled with "0".

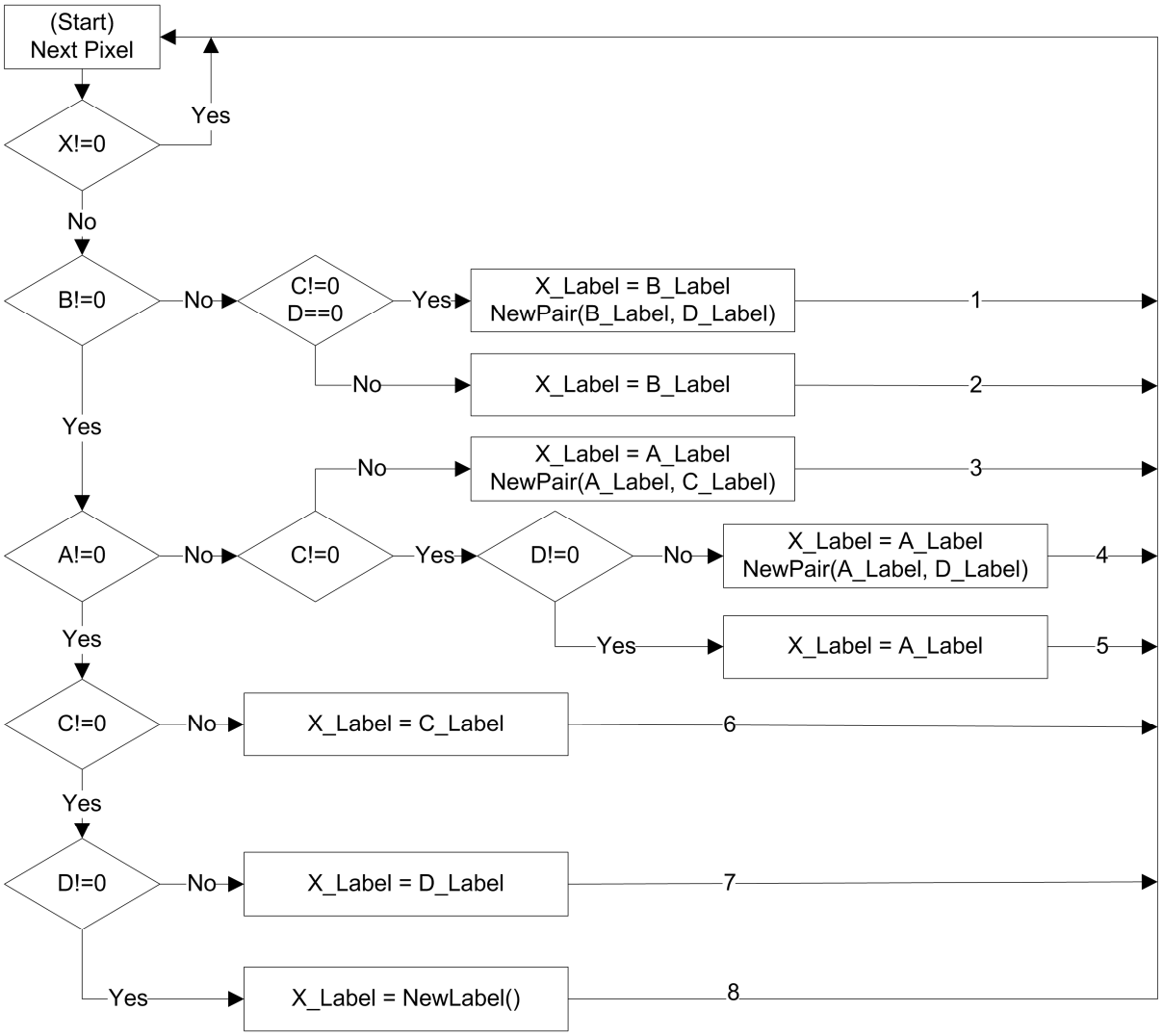


Fig. 6.2 Block diagram of the labeling algorithm

The labeling algorithm is applied for each individual pixel by scanning the pixels in a top-down left-right manner. **Observation:** in order to use the 5 pixel connectivity (Fig. 6.1) the image scanning must start on the second line from the top and second column from the left and will finish on the last line and the column before the last one. This is because the four neighbors of the labeled pixel W, NW, N, NE (pixels A, B, C, D, Fig. 6.1) must be located inside the image; otherwise the memory buffer in which the image is located may be exceeded.

The pixel labeling and equivalent label generation algorithm is described next for the following cases:

a) If the current pixel belongs to the background then we skip to the next pixel;
b) If the current pixel belongs to an object, then it must be labeled, and the following cases may occur:

- If the neighbors from NW and NE are object pixels and the neighbor from the N direction is not an object pixel, then the label of the current pixel will be the same as NW neighbor's label and an equivalent pair will be added for the NW and NE directions (branch 1, Fig. 6.2);

- If the neighbor from NW is an object pixel and either the neighbor from the N is an object pixel or the neighbor from NE is not an object pixel, then the label of the current pixel will be the same as the label of the pixel from NW (branch 2, Fig. 6.2);
- If the neighbor from NW is not a pixel object and the neighbors from W and N are object pixels then the label of the pixel will be the same as the label of the W pixel and the equivalent pair will be added for the W and N neighbors (branch 3, Fig. 6.2);
- If the neighbors from the NW and N are not object pixels and the neighbors from W and NE are object pixels, then the label of the pixel will be made the same as the label of the W pixel and an equivalent pair is added for the W and NE pixel labels (branch 4, Fig. 6.2);
- If the neighbor from the W direction is an object pixel and the neighbors from NW, N and NE are not object pixels, then the label of the current pixel will be equal to the label of the pixel from the W direction (branch 5, Fig. 6.2);
- If the neighbors from the NW and W directions are not object pixels and the neighbor from the N direction is an object pixel, then the label of the current pixel will be the same as the label of the N pixel (branch 6, Fig. 6.2);
- If the neighbors from NW, W and N are not object pixels and the neighbor from the NE direction is an object pixel, then the label of the current pixel will be the same as the label of the NE pixel (branch 7, Fig. 6.2);
- If none of the pixel's neighbors (NW, W, N and NE) are object pixels, then a new label will be generated for the current pixel (branch 8, fig. Fig. 6.2).

Multiple labeling of an object appears in the case of sequential scanning, in the case of "J" shape objects, as you can see in Fig. 6.3. The labeling process begins to label two "different" objects with labels "1" and "2", until the pixel "X" is reached, where we found that object 1 and 2 are connected. At this stage the two labels are considered equivalent and are appended to the list of equivalent pairs (NewPair(1,2)).

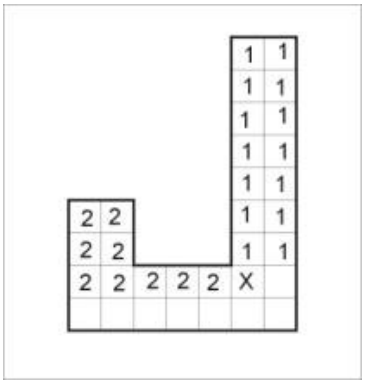


Fig. 6.3 The case of multiple labeled objects

Step 2. Establishing the labeling equivalence classes.

After all the pixels were labeled and the equivalence pairs were established, the equivalence classes must be found. The equivalence pairs are binary relations stored in the list. To find the equivalence classes a graph can be used in which the nodes are the labels, and the arches are the equivalence binary relations. The problem of finding the equivalence classes is to find the connected sub graphs (the transitive closure of the equivalence relationships).

To find a connected sub-graph a breadth first search can be used, starting from any graph node. All the neighbors of the starting node are put in a queue and are marked as visited (in order to

track the nodes that were already considered as part of the current connected sub-graph); when a node is extracted from the queue, its neighbors are added (if they were not previously visited) and are marked as visited. The process is repeated for each node in the queue, until the queue becomes empty. All the nodes that went through the queue will bear the same label which will be associated to the equivalence class. Next, an unvisited node is found and its equivalence class is built. This process continues until all the graph's nodes have been visited. An isolated node represents an equivalent class containing a single element.

For relabeling, an integer array may be used. The new label for the old label *i* will be located at index *i*. All array's elements are initialized with 0. This array may also be used for keeping track of the labels that were not previously visited: a 0 value marks that the label (graph node) was not previously visited.

One must go through the equivalent pair list in order to search for one node's neighbors (the node is the first element of the equivalence pair and the neighbor is the other). This adjacency list graph representation has the disadvantage of long search times. An adjacency matrix can be used, but it would require a lot of memory. A sparse matrix representation can be used instead.

Step 3. Re-labeling all the labels with the values corresponding to the equivalence classes.

The label image is scanned, and each label is replaced by the new label corresponding to its equivalence class. This operation can be accomplished by using the array described above.

6.3 Labeling examples

Fig. 6.4 Labeling examples

6.4 Implementation hints

For labeling, the source image must be scanned in a top-down, left-right manner. Because of the five pixels connectivity used (Fig. 6.1), the scanning will start from the second line and second column from the top and will end at the last line and the column before the last one (see Step 1, section 6.2).

```
for (int i=dwHeight-2;i>=0;i--)
      for (int j=1;j<dwWidth-1;j++)
            if (lpSrc[i*w+j]==0)
            {
                  // if the current pixel is black
                  // the labeling algorithm is applied
                  // the value is stored in the label image
                  // the equivalence pairs are also stored
            }
```

Observations:
1. The bitmap image is vertically flipped in memory (line 0 from the memory corresponds to the bottom line of the image).
2. For the labels image (matrix) an *int* array having the same size as the image will be allocated.

After the object pixels were labeled, the equivalence classes are determined based on the pairs from the set of equivalent labels stored previously (see Step 2, section 6.2).

The last step is to re-label the pixels with the label of the equivalence class to which its old label belongs. Displaying the labeled objects is made with different colors (highly contrasting colors) and can be done by modifying the 1…254 LUT entries (the 0 and 255 are not modified because the 0 value is a marker of an inexistent equivalence class and the value 255 is reserved for the white background). This works if the resulting equivalence classes have labels from 1 to 254.

```
// modifying the LUT for displaying with different colors
for (int k=1;k<=254;k++)
{
      // generate a random color for index k, 1≤k≤254
      bmiColorsDst[k].rgbRed = randomValueRed;
      bmiColorsDst[k].rgbGreen = randomValueGreen;
      bmiColorsDst[k].rgbBlue = randomValueBlue;
}
```

Observations:
1. The binary image used for labeling must be an 8 bits/pixel image with a sorted LUT which contains only black pixels (index 0 – for objects) and white pixels (value 255 – for background)
2. Using the previously presented coloring method with different colors for all labeled objects allows only 254 different colors for displaying the labeled objects.

6.5 Practical Work

1. Implement the steps for the binary objects labeling algorithm.
2. Add a processing function to the DIBLook framework for executing this algorithm. Display the labeled objects using different colors.
3. **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!!!**

6.6 References

[1]. Umbaugh Scot E, *Computer Vision and Image Processing*, Prentice Hall, NJ, 1998, ISBN 0-13-264599-8
[2]. Robert M. Haralick, Linda G. Shapiro, *Computer and Robot Vision*, Addison-Wesley Publishing Company, 1993.

7 Border Tracing Algorithm

7.1 Objectives:

The purposes of this laboratory session are:

- to extract the objects' contours using a border tracing algorithm;
- to represent efficiently each extracted contour using chain codes;
- to take advantage of using chain codes in representing the objects' contours (border reconstruction, matching, merging etc.).

7.2 Theoretical Background

7.2.1 Border Tracing Algorithm

The border tracing algorithm is used to extract the contours of the objects (regions) from an image. When applying this algorithm it is assumed that the image with regions is either binary or those regions have been previously labeled.

Algorithm's steps:

1. Search the image from top left until a pixel of a new region is found; this pixel P_0 is the starting pixel of the region border. Define a variable *dir* which stores the direction of the previous move along the border from the previous border element to the current border element. Assign
   (a) *dir* = 0 if the border is detected in 4-connectivity (Fig. 7.1 a)
   (b) *dir* = 7 if the border is detected in 8-connectivity (Fig. 7.1 b)
2. Search the 3x3 neighborhood of the current pixel in an anti-clockwise direction, beginning the neighborhood search at the pixel positioned in the direction
   (a) (*dir* + 3) *mod* 4 (Fig. 7.1 c)
   (b) (*dir* + 7) *mod* 8 if *dir* is even (Fig. 7.1 d)
   (*dir* + 6) *mod* 8 if *dir* is odd (Fig. 7.1 e)
   The first pixel found with the same value as the current pixel is a new boundary element P_n. Update the *dir* value.
3. If the current boundary element P_n is equal to the second border element P_1 and if the previous border element P_{n-1} is equal to P_0, stop. Otherwise repeat step (2).
4. The detected border is represented by pixels $P_0 \ldots P_{n-2}$.

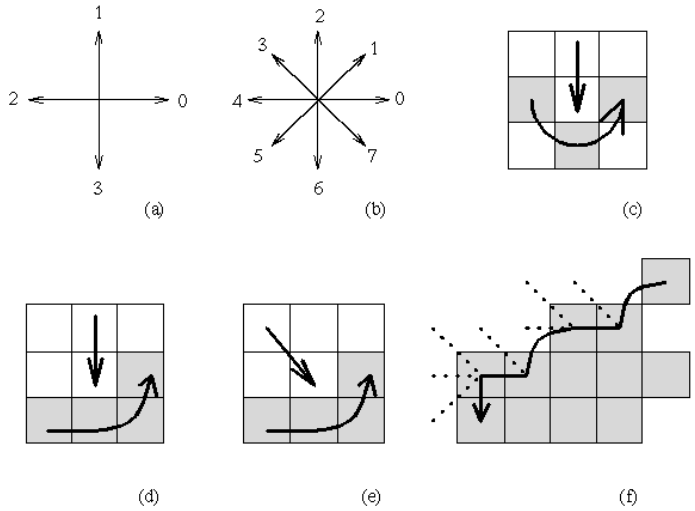


Fig.7.1 (a) Direction notation, 4-connectivity, (b) 8-connectivity, (c) pixel neighborhood search sequence is 4-connectivity, (d),(e) search sequence in 8-connectivity, (f) boundary tracing in 8-connectivity (dashed lines show pixels tested during the border tracing).

Remarks:

- The above algorithm works for all regions larger than one pixel.
- Looking for the border of a single-pixel region is a trivial problem.
- This algorithm is able to find region borders but does not find borders of region holes.
- To search for the object's holes' borders as well, the border must be traced starting in each region or hole border element if this element has never been a member of any border previously traced.
- Note that if objects are of unit width, more conditions must be added.

7.2.2 Chain Codes Extraction

The *chain code* provides a storage-efficient representation for the boundary of an object in a binary image. The chain code representation incorporates such pertinent information as the length of the boundary of the encoded object, its area, and moments. Chain codes lend to efficient calculation of certain curve parameters. Additionally, chain codes are invertible in that an object can be reconstructed from its chain code representation.

The basic idea behind the chain code is that each boundary pixel of an object has an adjacent boundary pixel neighbor whose direction from the given boundary pixel can be specified by a unique number between 0 and 7 (8-connectivity neighborhood). Chain codes could also be defined using a 4-connectivity neighborhood. A 4-connecivity neighborhood chain codes example it is presented in Fig. 7.2.

In the following we discussion we use the 8-connectivity neighborhood. Given a pixel, consider its eight neighboring pixels. Each 8-neighbor can be assigned a number from 0 to 7 representing one of eight possible directions from the given pixel (see Fig. 7.2). This is done with the same orientation throughout the entire image.

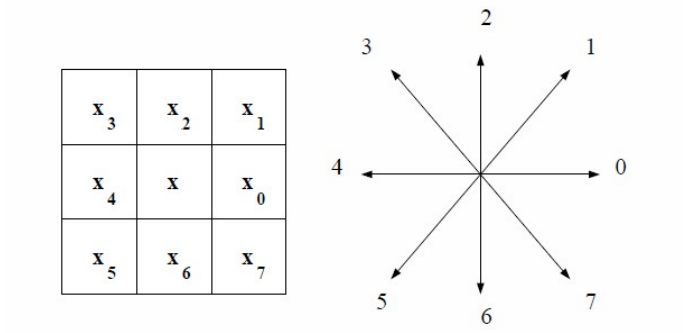


Fig.7.2 The 8-neighborhood and the associated eight directions

The chain code for the boundary of a binary image is a sequence of integers $c=\{c_0, c_1, \ldots, c_{n-1}\}$, having each c_i from the set $\{0,1, \ldots ,7\}$ for $i=0, 1, \ldots, n-1$. The number of elements in the sequence c is called the length of the chain code. The elements c_0 and c_{n-1} are called the *initial* and *terminal point* of the code, respectively. Starting at a given base point, the boundary of an object in a binary image can be traced out using the head-to-tail directions that the chain code provides.

Fig. 7.3 illustrates the process of tracing out the boundary of an airplane by following direction vectors. The information in Fig. 7.3 is then "flattened" to derive the chain code for its boundary. Suppose we choose the topmost left feature pixel in Fig. 7.3 as the base point for the boundary encoding. The chain code for the boundary of the airplane is the sequence:
7, 6, 7, 7, 0, … , 1, 1, 1.

Given the base point and the chain code, the boundary of the airplane can be completely reconstructed. The chain code is an efficient way of storing boundary information because it requires only three bits ($2^3 = 8$) to determine any one of the eight directions.

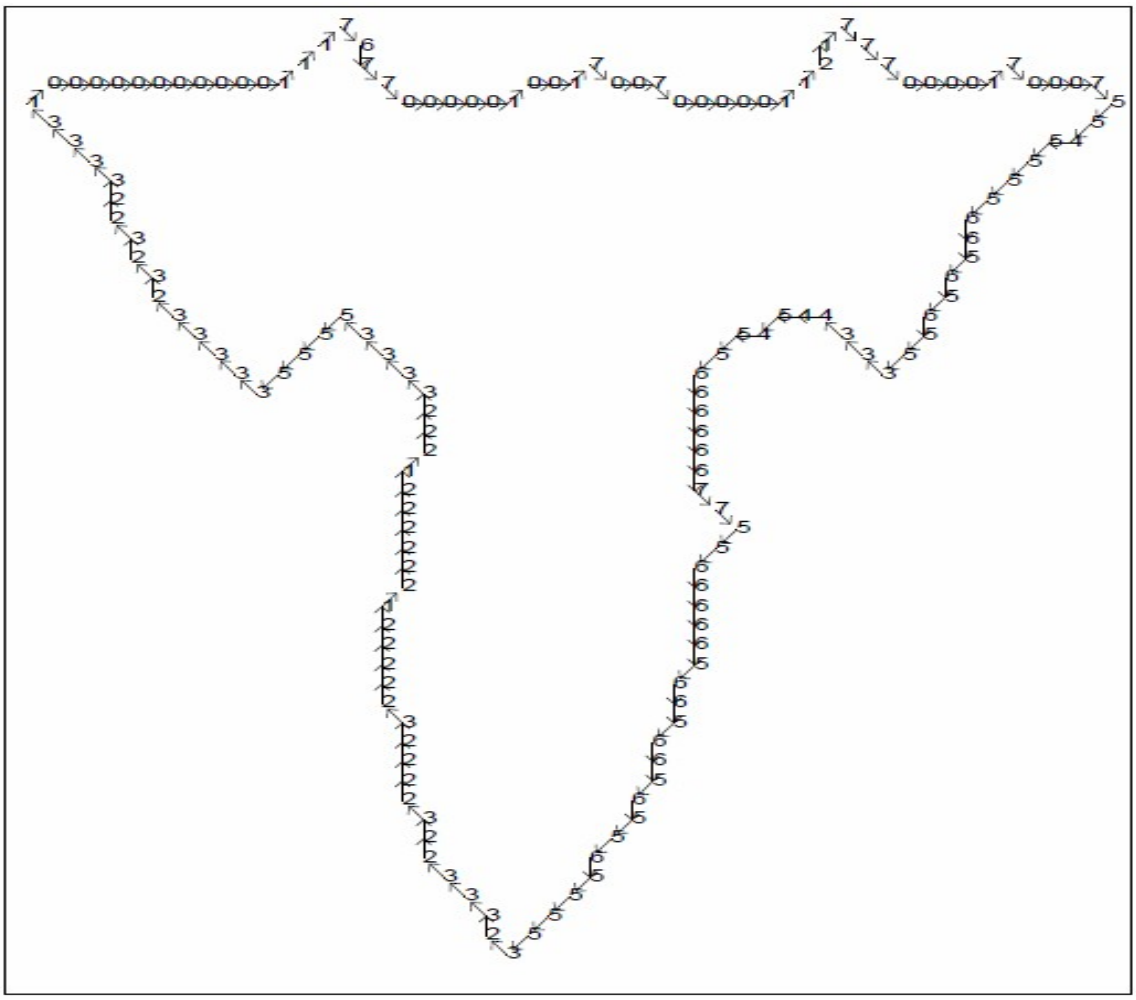


Fig.7.3 Chain code directions with associated direction numbers

Chain codes may be made position-independent by ignoring the "start point". If they represent closed boundaries they may be "start point normalized" by choosing the start point so that the resulting sequence of direction codes forms an integer of minimum magnitude.

The "derivative" of the chain code is useful because it is invariant under boundary rotation. The derivative (really a first difference mod 4 or 8) is simply another sequence of numbers indicating the relative direction of chain code segments; the number of left hand turns of $\pi/2$ or $\pi/4$ needed to achieve the direction of the next chain segment. A *mod* 4 or *mod* 8 difference is called a chain code *derivative* (see Fig. 7.4).

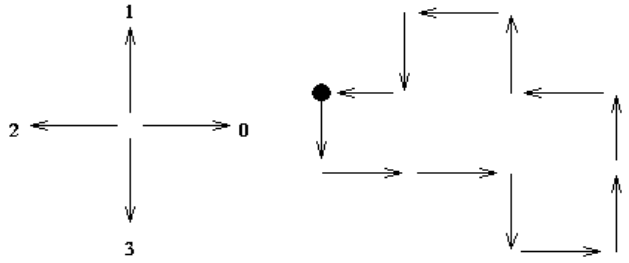


Fig.7.4 Chain code in 4-connectivity and its derivative.

Code:	3, 0, 0, 3, 0, 1, 1, 2, 1, 2, 3, 2
Derivative:	1, 0, 3, 1, 1, 0, 1, 3, 1, 1, 3, 1

Chain codes properties:

1. Chain codes describe an object by a sequence of unit-size (4-connectivity) line segments with a given orientation.
2. The first element of such a sequence must bear information about its position to allow reconstruction of the region.
3. Even codes {0, 2, 4, 6} correspond to horizontal and vertical directions; odd codes {1, 3, 5, 7} correspond to the diagonal directions.
4. Each code can be considered as the angular direction, in multiples of 45 degrees that we must move to go from one contour pixel to the next.
5. The absolute coordinates of the first contour pixel (e.g. top, leftmost) together with the chain code of the contour represent a complete description of the discrete region contour.
6. When there is a change between two consecutive chain codes, then the contour has changed direction. This point is defined as a *corner*.

7.3 Practical Work

Using the Diblook framework and the laboratory's additional images and files:

1. Implement the border tracing algorithm and draw the object contour on an image having a single object.
2. Starting from the border tracing algorithm write the algorithm that builds the chain code and derivative chain code for an object. Compute and display both codes (chain code and derivative chain code) for an image with a single object.
3. Implement a function that reconstructs (draws) the border of an object over an image having as inputs the start point coordinates and the chain code in 8-neighborhood *(reconstruct.txt*). Load the image *gray_background.bmp* and apply the function that reconstructs the border. You should obtain the contour of the word "EXCELLENT" (having all the letters connected).
4. **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!!!**

Additional info:

The test images with a single object have:

- 8 bits/pixel
- index 0 for object's pixels (black pixels)
- other index value for background pixels (white pixels)

The file *reconstruct.txt* is a text file having:

- on the first line the start point coordinates (row column) separated with a space;
- on the second line the number of chain codes;
- on the third line the chain codes (sequence of directions in 8-connectivity) separated with a space.

7.4 Refernces

[1] Border Tracing – Digital Image Processing lectures, The University of Iowa, http://www.icaen.uiowa.edu/~dip/LECTURE/Segmentation2.html#tracing
[2] Contour Representations – Quantitative Imaging Group, Delft University http://www.ph.tn.tudelft.nl/Courses/FIP/noframes/fip-Contour.html#Heading27
[3] G.X. Ritter, J.N. Wilson – Handbook of Computer Vision Algorithms in Image Algebra Second Edition – Chapter 10.4 Chain Code Extraction and Correlation, CRC Press, New York 2001
[4] Chain Codes – Digital Image Processing lectures, The University of Iowa http://www.icaen.uiowa.edu/~dip/LECTURE/Shape2.html#chaincodes
[5] Representation of Two-Dimensional Geometric Structures, http://homepages.inf.ed.ac.uk/rbf/BOOKS/BANDB/LIB/bandb8_12.pdf

8 Morphological operations on binary images

8.1 Introduction

Morphological operations are affecting the form, structure or shape of an object. Applied on binary images (black & white images – Images with only 2 colors: *black* and *white*). They are used in pre or post processing (filtering, thinning, and pruning) or for getting a representation or description of the shape of objects/regions (boundaries, skeletons convex hulls).

8.2 Theoretical considerations

The two principal morphological operations are *dilation* and *erosion* [1]. Dilation allows objects to expand, thus potentially filling in small holes and connecting disjoint objects. Erosion shrinks objects by etching away (eroding) their boundaries. These operations can be customized for an application by the proper selection of the structuring element, which determines exactly how the objects will be dilated or eroded.

Notations:

black pixel: in grayscale values for a 8 bits/pixel indexed image its value will be 0
white pixel: in grayscale values for a 8 bits/pixel indexed image its value will be 255

8.2.1 The dilation

The *dilation* process is performed by laying the structuring element **B** on the image **A** and sliding it across the image in a manner similar to convolution (will be presented in a next laboratory). The difference is in the operation performed. It is best described in a sequence of steps:

1. If the origin of the structuring element coincides with a 'white' pixel in the image, there is no change; move to the next pixel.
2. If the origin of the structuring element coincides with a 'black' in the image, make black all pixels from the image covered by the structuring element.

Notation:

$A \oplus B$

The structuring element can have any shape. Typical shapes are presented below:

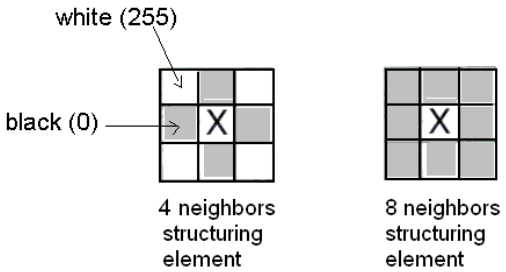


Fig. 8.1 Typical shapes of the structuring elements (B)

An example is shown in Fig. 8.2. Note that with a dilation operation, all the 'black' pixels in the original image will be retained, any boundaries will be expanded, and small holes will be filled.

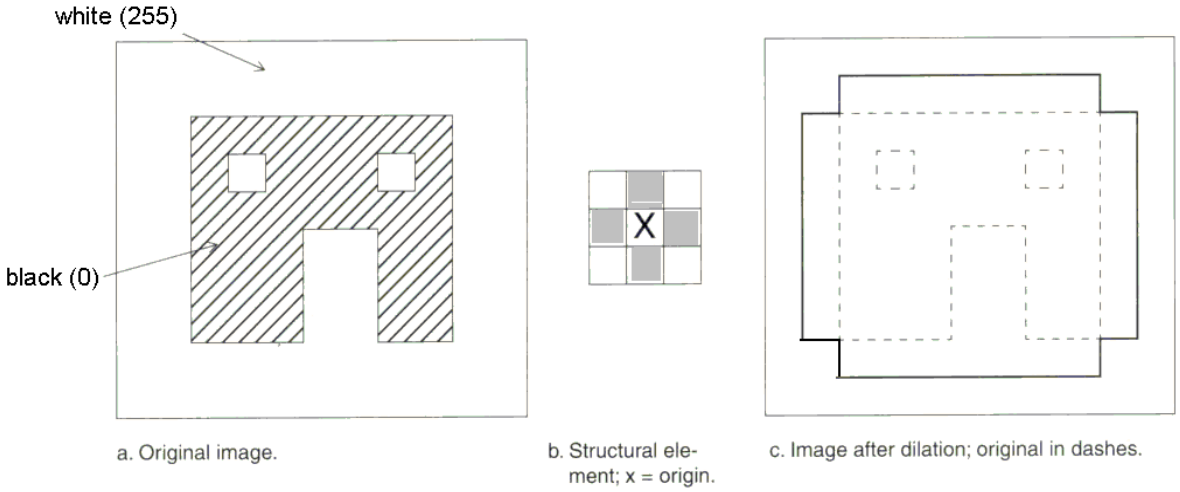


Fig. 8.2 Illustration of the dilatation process

a. b.

Fig. 8.3 Example of the dilation: a. Original image **A**; b. The result image: $\mathbf{A} \oplus \mathbf{B}$.

8.2.2 The erosion

The *erosion* process is similar to dilation, but we turn pixels to 'white', not 'black'. As before, slide the structuring element across the image and then follow these steps:

1. If the origin of the structuring element coincides with a 'white' pixel in the image, there is no change; move to the next pixel.

2. If the origin of the structuring element coincides with a 'black' pixel in the image, and any of the 'black' pixels in the structuring element extend beyond the object (with 'black' pixels) in the image, then change the 'black' pixel in the image to a 'white' pixel.

Notation:

$$A \ominus B$$

In Fig. 8.4, the only remaining pixels are those that coincide to the origin of the structuring element where the entire structuring element was contained in the existing object. Because the structuring element is 3 pixels wide, the 2-pixel-wide right leg of the image object was eroded away, but the 3-pixel-wide left leg retained some of its center pixels.

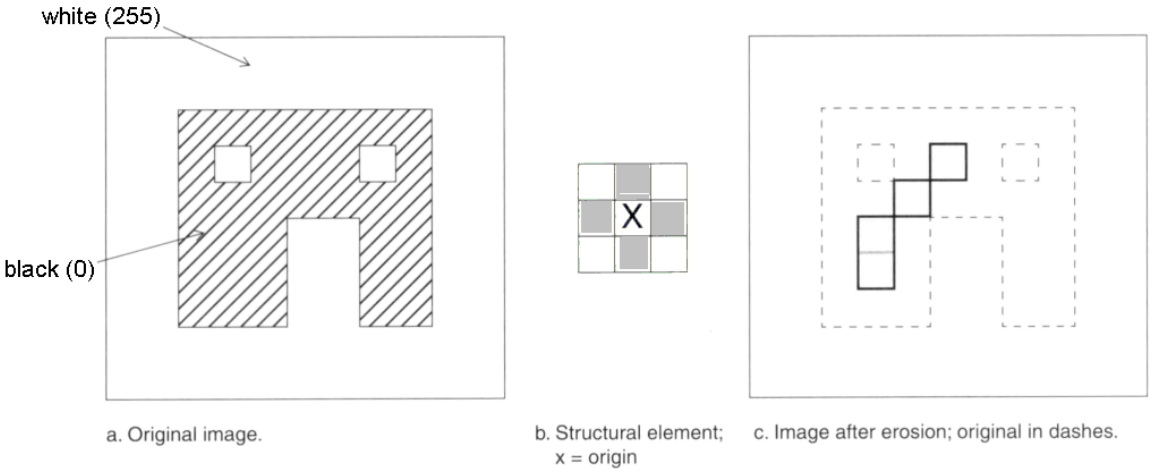


Fig. 8.4 Illustration of the erosion process

a. b.

Fig. 8.5 Example of the erosion: a. Original image **A**; b. The result image: **A** Θ **B**.

8.2.3 Opening and closing

These two basic operations, dilation and erosion, can be combined into more complex sequences. The most useful of these for morphological filtering are called opening and closing [1]. *Opening* consists of an erosion followed by a dilation and can be used to eliminate all pixels in regions that are too small to contain the structuring element. In this case the structuring element is often called a probe, because it is probing the image looking for small objects to filter out of the image. See Fig. 8.6 for the illustration of the opening process.

Notation:

$$A \circ B = (A \Theta B) \oplus B$$

Closing consists of a dilation followed by erosion and can be used to fill in holes and small gaps. In Fig. 8.7 we see that the closing operation has the effect of filling in holes and closing gaps. Comparing the left and right images from Fig. 8.8, we see that the order of operation is important. Closing and opening will generate different results even though both consist of erosion and dilation.

Notation:

$$A \bullet B = (A \oplus B) \Theta B$$

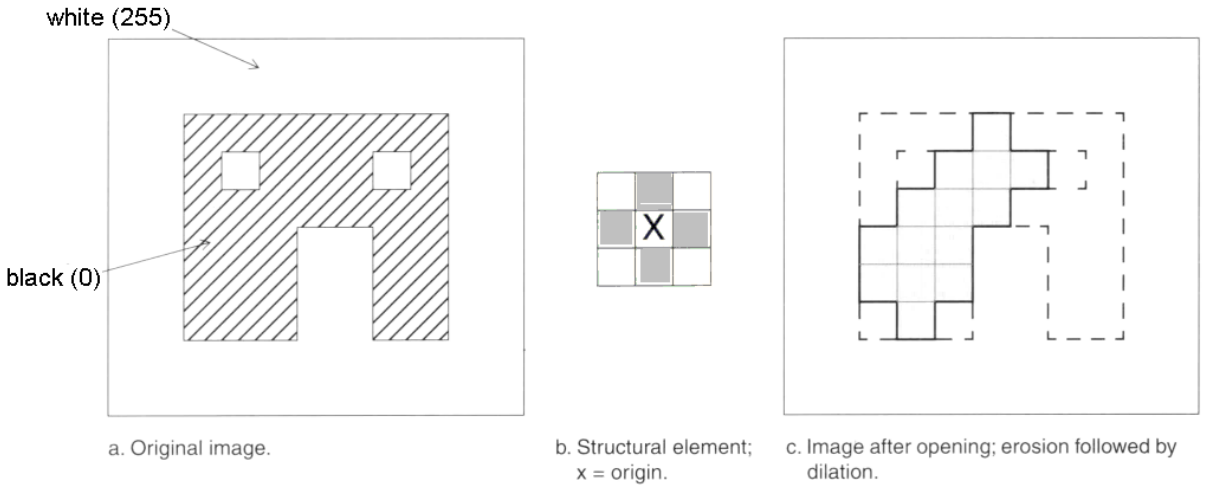


Fig. 8.6 Illustration of the opening process

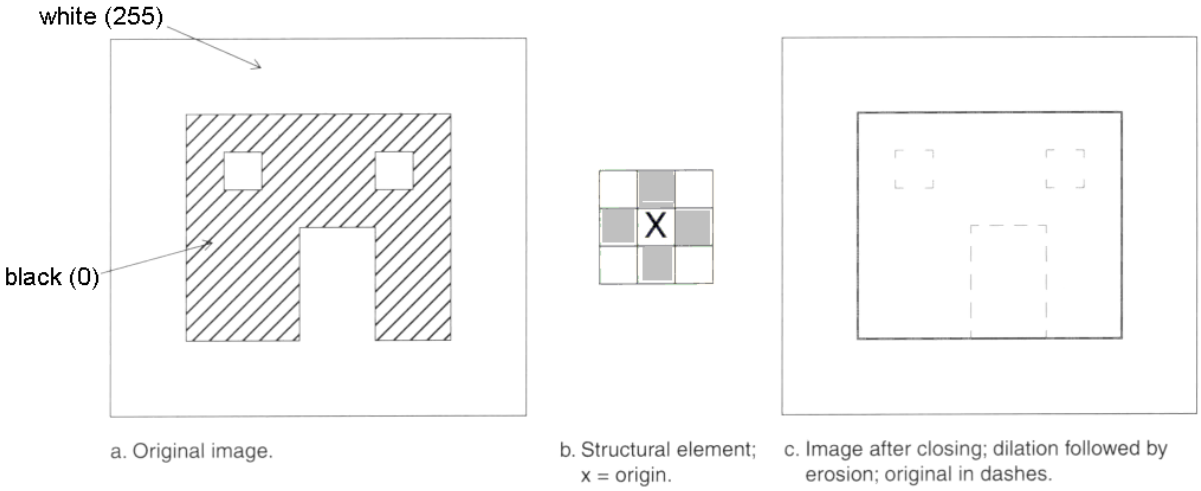


Fig. 8.7 Illustration of the closing process

a. b.

Fig. 8.8 Results of the opening (a) and closing (b) operations applied on the original image from Fig. 8.5a.

8.2.4 Some basic morphological algorithms [2]

8.2.4.1 Boundary extraction

The boundary of a set A, denoted by $\beta(A)$, can be obtained by first eroding A by B and then performing the set differences between A and its erosion. That is,

$$\beta(A)=A-(A\Theta B)$$

where

B is a suitable structuring element.

'–' is the difference operation on sets (illustrated in Fig. 8.10)

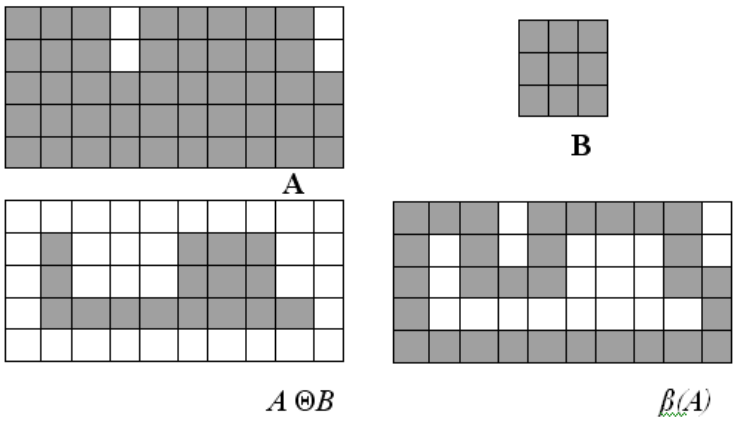


Fig. 8.9 Illustration of the boundary extraction algorithm

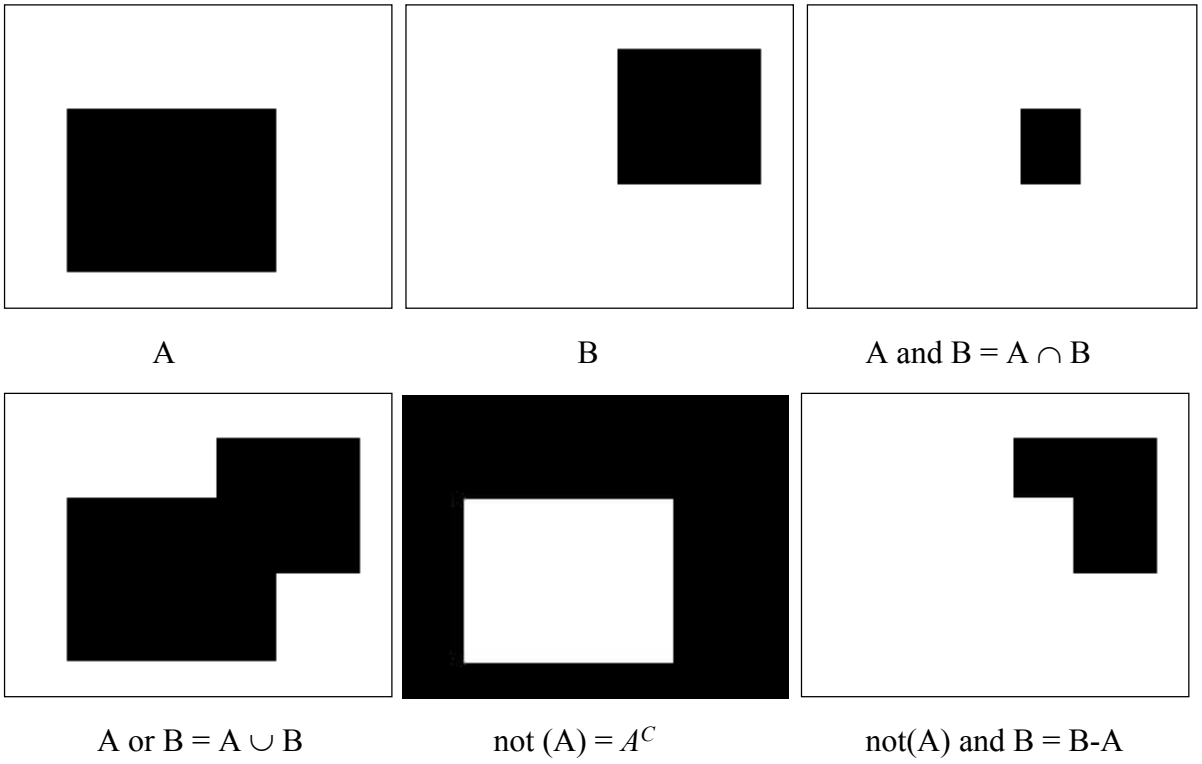


Fig. 8.10 Illustration of the main operations on sets

8.2.4.2 Region filling

Next we develop a simple algorithm for region filling based on set dilations, complementation, and intersections.

Beginning with a point p inside the boundary, the objective is to fill the entire region with 'black'. If we adopt the convention that all non-boundary (background) points are labeled 'white', then we assign a value of 'black' to p to begin. The following procedure then fills the region with 'black':

$$X_k = (X_{k-1} \oplus B) \cap A^C \qquad k=1,2,3,\ldots$$

where

$X_0=p$,
B is the symmetric structuring element
$\cap$ - is the intersection operator (see Fig. 8.10)
A^C – is the complement of set A (see Fig. 8.10)

The algorithm terminates at iteration step k if $X_k=X_{k-1}$. The set union of X_k and A contains the filled set and its boundary.

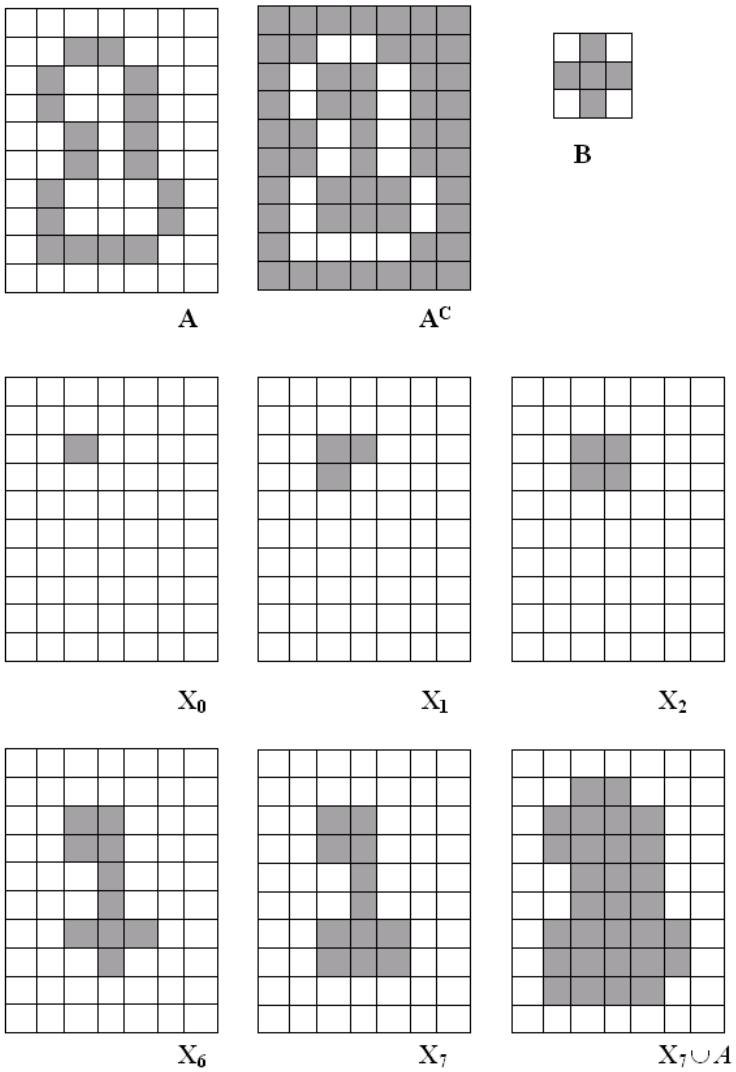


Fig. 8.11 Illustration of the region filling algorithm

8.3 Implementation hints

8.3.1 Using a supplementary image buffer for chain processing

The results of the basic morphological operations (dilation and erosion) should be applied in the following manner:

Destination image = *Source image* (operator) *Structuring element*

The source image shouldn't be affected in any way!

For the implementation of the combined morphological operations (opening and closing) or of the repeated operations (for example: *n* consecutive erosions) in a single processing function a supplementary image buffer should be created used. This can be done as follows (the code exemplifies how to perform two dilations, using an auxiliary temporary buffer):

```
//allocate a temporary buffer.
//its dimensions should be w and dwHeight (same as
//the pixel array in the source and destination bitmap
unsigned char *lpTemp = new unsigned char [w*dwHeight];
//perform 1st dilation, and write the result
//in the temporary buffer
//perform 2nd dilation, and write the result
//in the destination image
//free the buffer! C++ doesn't have garbage collection
//use the delete[] operator, not delete!!
delete[] lpTemp;
```

8.3.2 Additional hints for designing an input dialog box

If you want select the type of the morphological operation which you want to apply and the number of its repetitions, you can use a dialog box as input. Creating a dialog box and using edit controls as inputs was presented in Laboratory 2.

The following example illustrates the implementation of a single selection from multiple options (for example the type the morphological operation, dilation, erosion, opening, closing, contour extraction, region filling) using radio buttons. A radio button is similar to a check box, with the difference that, in a group of radio buttons, only a single radio button may be selected at any time. In order to create a group of radio buttons, allow user interaction with it and obtain the selected button you must perform the following steps (see Fig. 8.12):

1. The first step is to create a new Dialog box and to add the corresponding class *CDlgSelectMorphologicalOperation* as explained in Laboratory 2.
2. In order to visually group the various radio buttons add a Group Box control on the dialog. Give the group box a suggestive name, such as "Operation Type"
3. Add the first radio button on the group box. Give it suggestive ID, such as IDC_OPERATION_TYPE. Make sure you select the "Group" option!! Also, for this and all subsequent radio buttons, make sure that the option "Auto" in the "Appearance" section is selected!!
4. Add the other radio buttons. It is not necessary to change their IDs. Make sure that their Group checkbox is NOT selected!!
5. Associate an integer member variable with the first control (the one with the group box selected.
6. The associated member variable holds the index of the selected control. The index is 0-based, i.e. if the first button is selected then the index is 0, if the second radio button is selected then the index is 1 and so on.

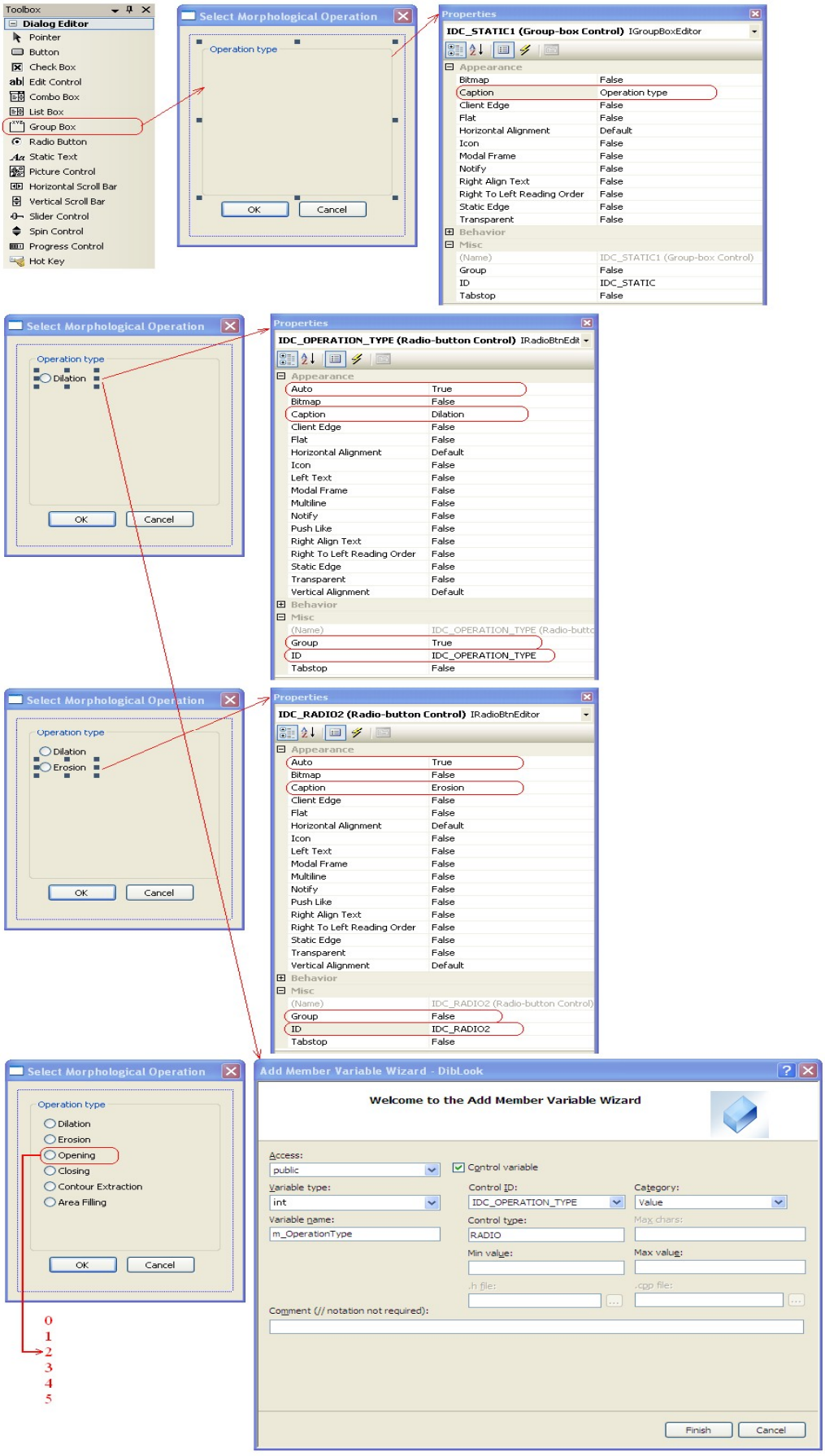


Fig. 8.12 Adding a group box, radio buttons, and associating the first radio button control with an index integer member variable

A clean way to obtain the desired operation selection from a group of radio buttons is:

1. Define an enumeration in your dialog class (*CDlgSelectMorphologicalOperation* in this example) holding the desired options. You should add this in the public section of your class, in the corresponding header file (*DlgSelectMorphologicalOperation.h* in this example):

```
class CDlgSelectMorphologicalOperation : public CDialog {
public:
//enumeration holding the operation type
//(enumerations are by default zero based)
enum EOperationType {
        Dilation,
        Erosion,
        Opening,
        Closing,
        ContourExtraction,
        AreaFilling,
  };
  ……
};
```

2. In your processing method (in *dibview.cpp*) add the following code to instantiate the dialog and obtain the selected operation:

```
//instantiate the dialog
CDlgSelectMorphologicalOperation dlgSelect;
//set the default selection to the first operation
dlgSelect.m_OperationType = 0;
//show the dialog in modal mode
dlgSelect.DoModal();
//obtain the selection
switch(dlgSelect.m_OperationType) {
        case CDlgSelectMorphologicalOperation::Dilation:
                …… //code for dilation
                break; //do not forget to put break after each
case
        case CDlgSelectMorphologicalOperation::Erosion:
                …… //code for erosion
                break;
        case CDlgSelectMorphologicalOperation::Opening:
                …… //code for opening
                break;
        case CDlgSelectMorphologicalOperation::Closing:
                …… //code for closing
                break;
        case CDlgSelectMorphologicalOperation::ContourExtraction:
                …… //code for contour extraction
                break;
        case CDlgSelectMorphologicalOperation::AreaFilling:
                …… //code for region filling
                break;
}
```

8.4 Practical work

1. Add to the DIBLook framework processing functions which implement the basic morphological operations.
2. Add the facility to apply the morphological operations repeatedly (*n* times). For that purpose use a dialog box to input the number of repetitions *n* (through an Edit control) and to select the type of the morphological operation (through radio buttons).
3. Implement the boundary extraction algorithm.
4. Implement the region filling algorithm.

5. **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!!!**

8.5 References

[1]. Umbaugh Scot E, *Computer Vision and Image Processing*, Prentice Hall, NJ, 1998, ISBN 0-13-264599-8
[2] R.C.Gonzales, R.E.Woods, *Digital Image Processing. 2-nd Edition*, Prentice Hall, 2002.

9 Statistical properties of grayscale images

9.1 Introduction

This laboratory work presents the main statistic features that characterize the distribution of intensity levels in a grayscale image or in an area / region of interest (ROI) of the image. These statistic features can be applied similarly to color images, on each color component.

The following notation will be used throughout this lab:

- *L=255* highest intensity level
- *h(g)* histogram function, counts the number of pixels with gray level *g*
- *M=H*W*, number of pixels in the image
- *p(g)=h(g)/M* gray level probability distribution function (PDF).

9.2 The mean value of intensity levels

The mean value of intensity levels is a measure of the mean intensity of the given image or of the region of interest. A dark image has a low mean value (Fig. 9.1 a), and a bright image has a high mean value (Fig. 9.1 b).

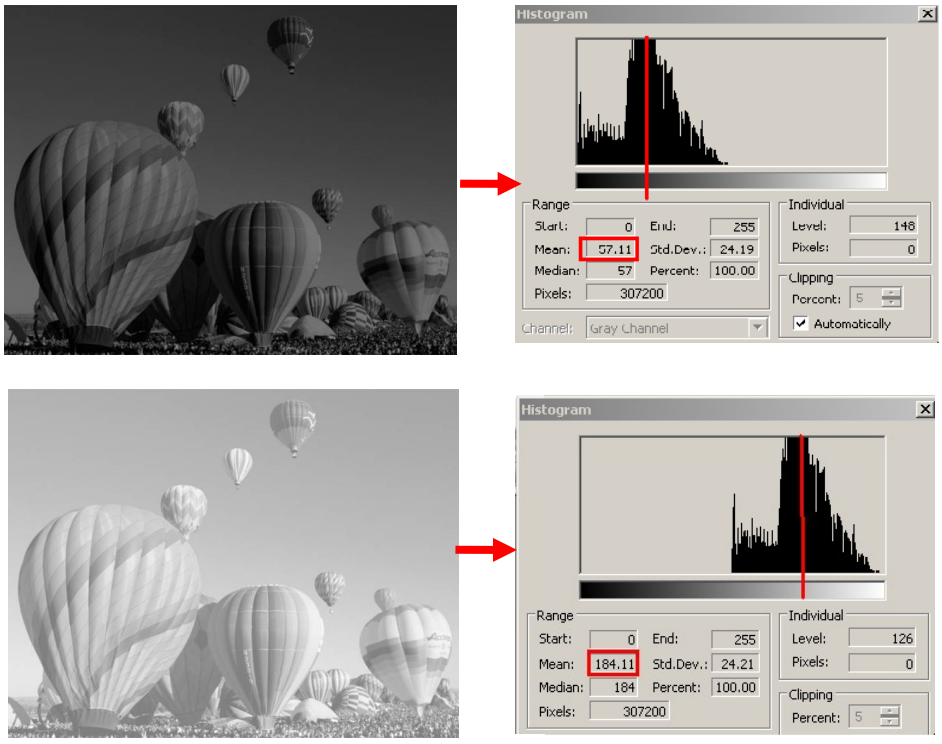


Fig. 9.1 The position of the histogram and the mean value of the intensity levels for a dark image (a) and a bright image (b)

The mean intensity value is computed as follows:

$$\bar{g} = \mu = \int_{-\infty}^{+\infty} g \cdot p(g) dg = \sum_{g=0}^{L} g \cdot p(g) = \frac{1}{M} \sum_{g=0}^{L} g \cdot h(g) \tag{9.1}$$

$$\bar{g} = \mu = \frac{1}{M} \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} I(i,j) \tag{9.2}$$

9.3 The standard deviation of the intensity levels

The standard deviation of the intensity levels represents a measure of the contrast of an image (region of interest). It characterizes the dispersion (spreading) of the intensity levels with respect to the mean value. An image having a high contrast will have a large standard deviation (Fig. 9.2 a – the histogram is spread on the entire range of intensity levels), and an image having a low contrast will be characterized by a small standard deviation (Fig. 9.2 b – the histogram is restricted to some intensity levels located around the mean value).

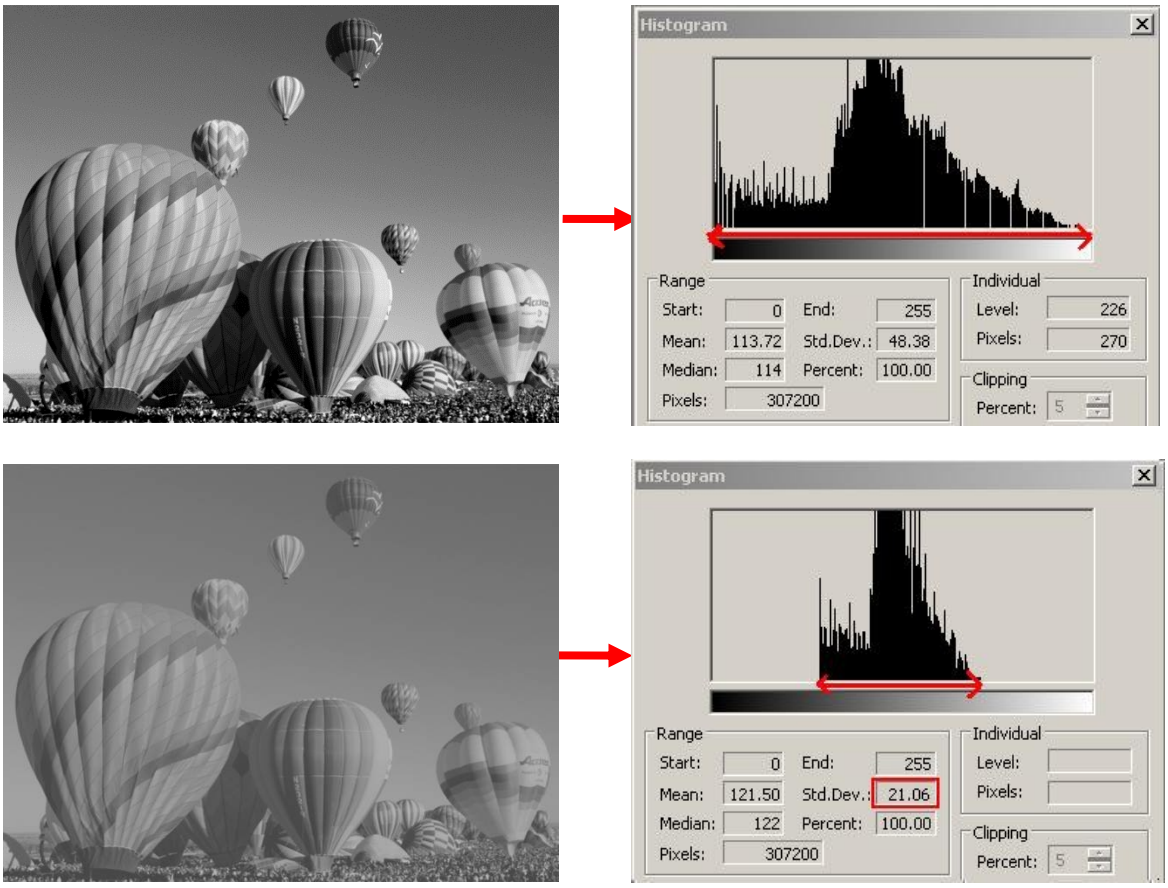


Fig. 9.2 The position of the histogram and of the standard deviation (2σ) of the intensity levels for an image of high contrast (a) and an image of low contrast (b).

The standard deviation of the intensity levels is given by:

$$\sigma = \sqrt{\sum_{g=0}^{L} (g-\mu)^2 \cdot p(g)} \tag{9.3}$$

$$\sigma = \sqrt{\frac{1}{M} \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} \left(I(i,j)-\mu\right)^2} \tag{9.4}$$

9.4 Threshold selection by optimal image approximation

This algorithm determines an optimal threshold, in the sense that the approximation error between the original image and the resulting binary image is minimal. The error can either be

computed as the sum of absolute differences (P1, city block or Manhattan norm) or as the squared sum of differences (the square of a P2 or Euclidean norm). A further improvement is to determine two gray level values for the approximating the image, instead of using black and white.

9.4.1 Minimizing the city block distance

The threshold that minimizes this distance is (see course notes):

$$T = \underset{t}{\operatorname{argmax}}\, t\sum_{g=t}^{L} h(g) = \underset{t}{\operatorname{argmax}}\, tA(t) \tag{9.5}$$

The function A(t) can be computed incrementally by using the recurrence:

$$\begin{aligned} A(t+1) &= A(t) - h(t), \\ A(0) &= \sum_{g=0}^{L} h(g) = M \end{aligned} \tag{9.6}$$

9.4.2 Minimizing the Euclidean distance

The threshold that minimizes this distance is (see course notes):

$$T = \underset{t}{\operatorname{argmax}}\, t\sum_{g=t}^{L} (2g-t)h(g) = \underset{t}{\operatorname{argmax}}\, tW(t) \tag{9.7}$$

The function W(t) can be computed incrementally by observing that:

$$\begin{aligned} W(t+1) &= \sum_{g=t+1}^{L} (2g-t-1)h(g) = \\ &= \sum_{g=t}^{L} (2g-t-1)h(g) - (2t-t-1)h(t) = \\ &= \sum_{g=t}^{L} (2g-t)h(g) - \sum_{g=t}^{L} h(g) - (t-1)h(t) = \\ &= W(t) - A(t) - (t-1)h(t) \end{aligned} \tag{9.8}$$

where A(t) is computed according to (9.6). We also have that:

$$W(0) = \sum_{g=0}^{L} 2gh(g) = 2\mu M \tag{9.9}$$

9.4.3 Improving the image approximation by using two non-black&white levels

The optimal threshold T is obtained by maximizing:

$$T = \underset{t}{\operatorname{argmax}} \left(\frac{\left(\sum_{g=0}^{t-1} gh(g)\right)^2}{\sum_{g=0}^{t-1} h(g)} + \frac{\left(\sum_{g=t}^{L} gh(g)\right)^2}{\sum_{g=t}^{L} h(g)} \right) = \underset{t}{\operatorname{argmax}} \left(\frac{\left(\mu M - M_{high}(t)\right)^2}{M - A(t)} + \frac{\left(M_{high}(t)\right)^2}{A(t)} \right), \tag{9.10}$$

$$t \geq 1 \text{ and } \sum_{g=0}^{t-1} h(g) \neq 0 \text{ and } \sum_{g=t}^{L} h(g) \neq 0 \ \left(M - A(t) \neq 0 \text{ and } A(t) \neq 0\right)$$

where A was computed in (9.6), and

$$M_{high}(t+1) = M_{high}(t) - th(t),$$
$$M_{high}(0) = \sum_{g=0}^{L} gh(g) = \mu M \tag{9.11}$$

Using the determined threshold T, the gray level values that minimize the approximation error are the two medians:

$$g_{low}(T) = \frac{\sum_{g=0}^{g=T-1} gh(g)}{\sum_{g=0}^{g=T-1} h(g)} \text{ and } g_{high}(T) = \frac{\sum_{g=T}^{g=L} gh(g)}{\sum_{g=T}^{g=L} h(g)} \tag{9.12}$$

9.5 Histogram analytical transformation functions

In Fig. 9.3 are shown some typical transformation functions of the intensity values, which can be expressed in an analytical form:

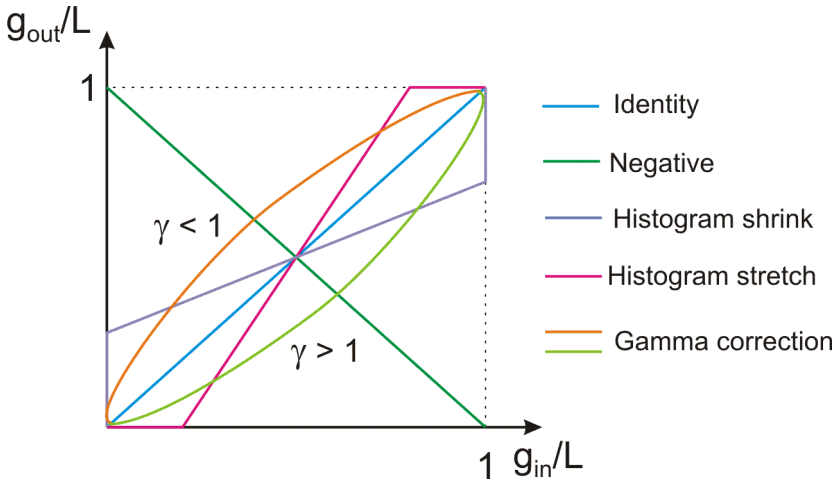


Fig. 9.3 Typical gray levels transformation functions

9.5.1 Identity function (no effect):

$$g_{out} = g_{in} \tag{9.13}$$

9.5.2 Image negative:

$$g_{out} = L - g_{in} = 255 - g_{in} \tag{9.14}$$

9.5.3 Histogram stretching / shrinking:

$$g_{out} = g_{out}^{MIN} + (g_{in} - g_{in}^{MIN}) \frac{g_{out}^{MAX} - g_{out}^{MIN}}{g_{in}^{MAX} - g_{in}^{MIN}} \tag{9.15}$$

where:

$$\frac{g_{out}^{MAX} - g_{out}^{MIN}}{g_{in}^{MAX} - g_{in}^{MIN}} = \begin{cases} > 1 & \Rightarrow \quad stretch \\ < 1 & \Rightarrow \quad shrink \end{cases} \tag{9.16}$$

9.5.4 Gamma correction:

$$g_{out} = L\left(\frac{g_{in}}{L}\right)^{\gamma} \tag{9.17}$$

Where:

γ is a positive coefficient: < 1 (gamma encoding/compression) or > 1 (gamma decoding / decompression)

Attention: always check that: 0 <= g_{out} <=255. If outside the domain, values should be saturated!!!

Fig. 9.4 Results of gamma correction operations

9.5.5 Brightness changing (histogram slide)

$$g_{out} = g_{in} + offset \tag{9.18}$$

Attention: always the following checking will be done: 0 <= g_{out} <=255. If an overflow beyond these limits appears, output values will be truncated or scaled!!!

9.6 Histogram equalization

Histogram equalization is a transform which allows us to obtain an output image with a quasi-uniform histogram/PDF, regardless the shape of the histogram/PDF of the input image. For that purpose, the following transform will be used (see lecture notes for more details):

$$s_k = T(r_k) = \sum_{j=0}^{k} p_r(r_j) = \sum_{j=0}^{k} \frac{n_j}{n} \quad , \quad k = 0...L \tag{9.19}$$

where:

r_k – normalized intensity level of the input image corresponding to the (un-normalized) intensity level k: $r_k = \frac{k}{L}$, ($0 \le r_k \le 1$ and $0 \le k \le L$)

s_k – corresponding normalized intensity level of the output image;

$p_C(r_k)$ – cumulative probability density function (CPDF) of the input image

$$p_C(r_k) = \sum_{j=0}^{k} p_r(r_j) = \sum_{g=0}^{k} \frac{h_r(g)}{M} \tag{9.20}$$

r_j – normalized intensity level of the input image corresponding to the (un-normalized) intensity level j: $r_j = \frac{j}{L}$.

9.6.1 Histogram equalization algorithm

1. Compute the histogram or the PDF of the input image (as a 256 elements vector)
2. Compute the CPDF of the input image (9.20), as a vector of 256 elements.
3. Compute the transformation for the histogram equalization according to (9.20). Because the s_k values obtained from (9.19) are normalized intensity values, it is necessary to transform the normalized intensity values s_k back to un-normalized ones by multiplication with L (the highest intensity value: 255 for 8 bits/pixel images):

$$g_{out} = Ls_k = \frac{L}{M} \sum_{g=0}^{g_{in}} h(g) \quad , \quad k = g_{in} \tag{9.21}$$

This transformation function can be written as an equivalence table (vector):

$$g_{out} = tab(g_{in}) = 255 \cdot p_C(g_{in}) \tag{9.22}$$

4. The intensity values of the output (equalized) image are computed using the equivalence table:

$$lpDst[i*w+j] = tab[lpSrc[i*w+j]] \tag{9.23}$$

9.7 Practical work:

1. Compute and display the mean and standard deviation of image intensity levels.
2. Implement the three functions for automatic threshold computation (section 9.4) and threshold the images according to these values.
3. Implement the histogram transformation functions (section 9.5) for image negative, histogram stretching/shrinking, gamma correction, histogram slide. Input the limits $g_{out}^{MIN}, g_{out}^{MAX}$, the gamma coefficient and the brightness increase value from a dialog box.
4. Implement the histogram equalization algorithm (section 9.6).

5. **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.**

9.8 References

[1]. R.C.Gonzales, R.E.Woods, *Digital Image Processing. 2-nd Edition*, Prentice Hall, 2002.

10 Image filtering in the spatial and frequency domains

10.1 Introduction

In this laboratory the convolution operator will be presented. This operator is used in the linear image filtering process applied in the spatial domain (in the image plane by directly manipulating the pixels) or in the frequency domain (applying a Fourier transform, filtering and then applying the inverse Fourier transform. Examples of such filters are: low pass filters (for smoothing) and high pass filters (for edge enhancement).

10.2 The convolution process in the spatial domain

The convolution process implies the usage of a convolution mask/kernel H (usually with symmetric shape and size w*w, with w=2k+1) which is applied on the source image according to (10.2).

$$I_D = H * I_S \tag{10.1}$$

$$I_D(x,y) = \sum_{i=-k}^{k} \sum_{j=-k}^{k} H(i,j) \cdot I_S(x+i, y+j), \quad x = 0...Height-1, \quad y = 0...Width-1 \tag{10.2}$$

This implies the scanning of the source image I_S, pixel by pixel, ***ignoring the first and last k rows and columns*** (Fig. 10.1) and the computation of the intensity value in the current position *(x, y)* of the destination image I_D using (9.2). The convolution mask is positioned spatially with its central element over the current position *(x, y)*.

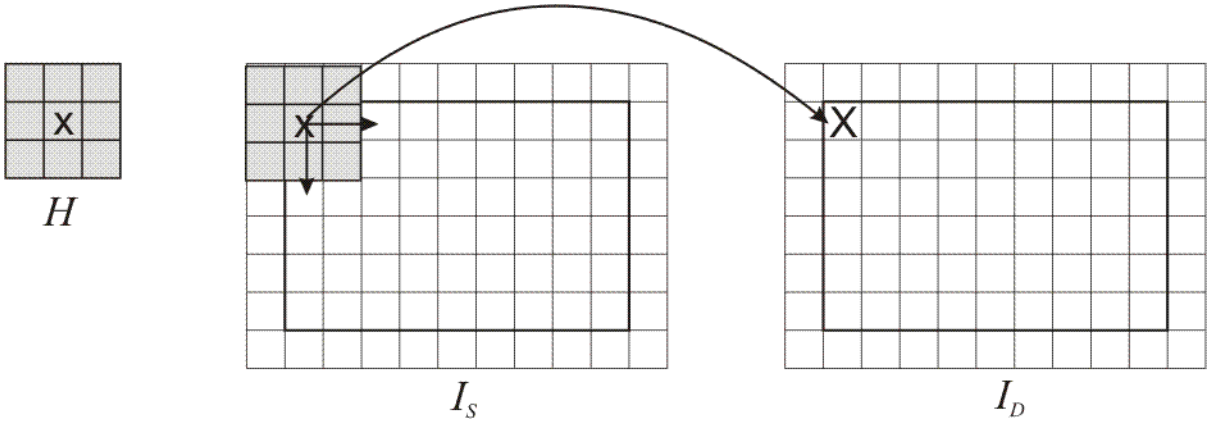


Fig. 10.1 Illustration of the convolution process.

The convolution kernels can have also non-symmetrical shapes (the central/reference element is not positioned in the center of symmetry). Convolution with such kernels is applied in a similar way, but such examples will not be presented in the current laboratory.

10.2.1 Low-pass filters

Low-pass filters are used for image smoothing and noise reduction (see the lecture material). Their effect is an averaging of the current pixel with the values of its neighbors, observable as a "blurring" of the output image (they allow to pass only the low frequencies of the image).

All elements of the kernels used for low-pass filtering have positive values. Therefore, a common practice used to scale the result in the intensity domain of the output image is to divide the result of the convolution with the sum of the elements of the kernel:

$$I_D(x,y) = \frac{1}{c} \cdot \sum_{i=-k}^{k} \sum_{j=-k}^{k} H(i,j) \cdot I_S(x+i, y+j) \tag{10.3}$$

where:

$$c = \sum_{i=-k}^{k} \sum_{j=-k}^{k} H(i,j) \tag{10.4}$$

Examples:

Mean filter (3x3):

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{10.5}$$

Gaussian filter (3x3):

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{10.6}$$

a. **b.** **c.**

Fig. 10.2 a. Original image; b. Result obtained by applying a 3x3 mean filter. c. Result obtained by applying a 5x5 mean filter.

10.2.2 High-pass filters

These filters will highlight regions with step intensity variations, such as edges (will allow to pass the high frequencies).

The kernels used for edge detection have the sum of their elements equal to 0:

Laplace filters (edge detection) (3x3):

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{10.7}$$

or

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{10.8}$$

High-pass filters (3x3):

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{10.9}$$

or

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9 & -1 \\ -1 & -1 & -1 \end{bmatrix} \tag{10.10}$$

a. b. c.

Fig. 10.3 a. The result of applying the Laplace edge detection filter (9.8) on the original image (Fig. 10.2 a); b. The result of applying the Laplace edge detection filter (9.8) on the blurred image from Fig. 10.2 b (previously filtered with the 3x3 mean filter); c. The result obtained by filtering the original image with the high-pass filter (9.10)

10.3 Image filtering in the frequency domain

The 1D discrete Fourier transform (DFT) of an array of N real or complex numbers is an array of N complex numbers, given by:

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi jkn}{N}}, \quad k = \overline{0...N-1} \tag{10.11}$$

The inverse discrete Fourier transform (IDFT) is given by:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k e^{\frac{2\pi jkn}{N}}, \quad n = \overline{0...N-1} \tag{10.12}$$

The 2D DFT is performed by applying the 1D DFT on each row of the input image and then on each column of the previous result. The 2D IDTF is performed by applying the 1D IDFT on each column of the DFT "image" and then on each row of the previous result. The set of

complex numbers which are the result of the DFT may also be represented in polar coordinates (magnitude, phase). The set of (real) magnitudes represent the frequency power spectrum of the original array.

The DFT and its inverse are usually performed using the Fast Fourier Transform recursive approach, which reduces the computation time from $O(n^2)$ to $O(n \ln n)$, which represents a significant speed increase, especially in the case of 2D image processing, where a $O(n^2 m^2)$ complexity would be intractable for large images as opposed to the almost linear in number of pixels $O(nm \ln(nm))$ complexity.

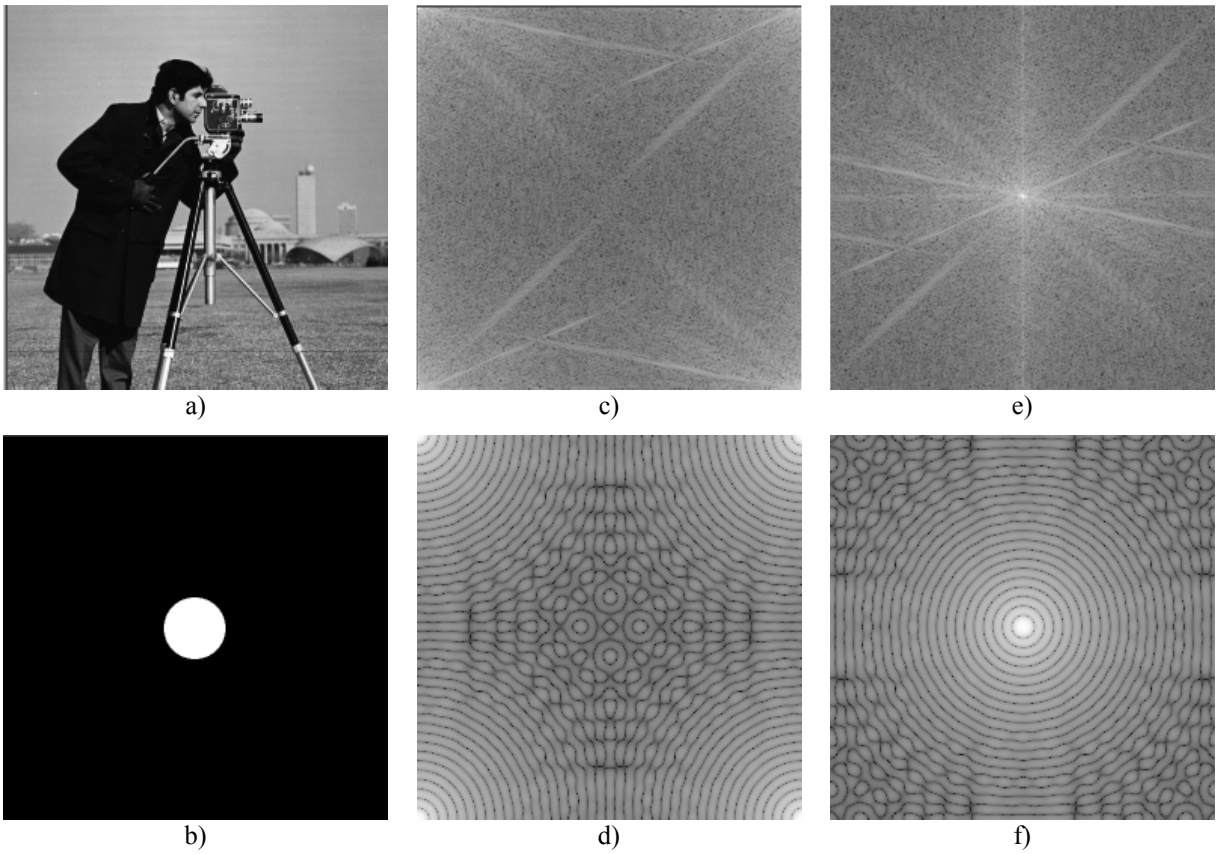


Fig.10.4 a) and b) original images; c) and d) logarithm of magnitude spectra; e) and f) centered logarithm of magnitude spectra

10.3.1 Aliasing

The aliasing phenomenon is a consequence of the Nyquist frequency limit (a sampled signal cannot represent frequencies higher than half the sampling frequency). This means that the higher half of the frequency domain representation is redundant. This fact can also be seen from the identity:

$$X_k = X^*_{N-k} \tag{10.13}$$

(where the asterisk denotes complex conjugation) which is true if the input numbers x_k are real. Therefore, the typical 1D Fourier spectrum will contain the low frequency components in both the lower and upper part, with high frequency located symmetrically about the middle. In 2D, the low frequency components will be located near the image corners and the high frequency

components in the middle (see Fig. 10.4c, d). This makes the spectrum hard to read and interpret. In order to center the low frequency components spectrum about the middle of the spectrum, one should first perform the transformation on the input data:

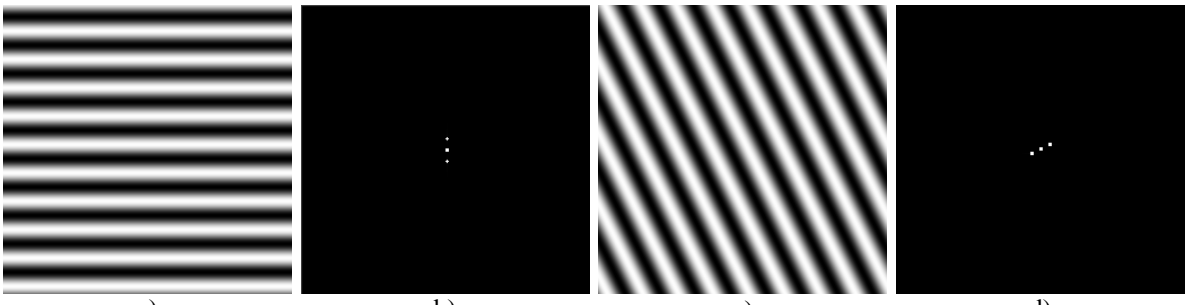
$$x_k \leftarrow (-1)^k x_k \tag{10.14}$$

In 2D the centering transformation becomes:

$$x_{uv} = (-1)^{u+v} x_{uv} \tag{10.15}$$

After applying this centering transform, in 1D the spectrum will contain the low frequency components in the center, and the high frequency components will be located symmetrically toward the left and right ends of the spectrum. In 2D, the low frequency components will be located in the middle of the image, while various high frequency components will be located toward the edges.

The magnitudes located on any line passing through the DFT image center represent the 1D frequency spectrum components of the original image, along the direction of the line. Every such line is therefore symmetrical about its middle (the image center).



a) b) c) d)

Fig. 10.5 Fourier transforms of sine image waves a) and c). The center point in b) and d) represent the DC component, the other two symmetrical points are due to the sine wave frequency.

10.3.2 Ideal low-pass and high-pass filters in frequency domain

The convolution in spatial domain is equivalent to scalar multiplication in frequency domain. Therefore, especially for large convolution kernels, it is computationally convenient to perform convolution in the frequency domain.

The algorithm for filtering in the frequency domain is:

a) Perform the image centering transform on the original image (9.15)
b) Perform the DFT transform
c) Alter the Fourier coefficients according to the required filtering
d) Perform the IDFT transform
e) Perform the image centering transform again (this undoes the first centering transform).

An ideal low pass filter will alter all the Fourier coefficients that are further away from the image center *(W/2, H/2)* than a given distance *R*, by turning them to zero (*W* is the image width and *H* is the image height):

$$X'_{uv} = \begin{cases} X_{uv}\ , & \left(\frac{H}{2}-u\right)^2+\left(\frac{W}{2}-v\right)^2 \le R^2 \\ 0\ , & \left(\frac{H}{2}-u\right)^2+\left(\frac{W}{2}-v\right)^2 > R^2 \end{cases} \tag{10.16}$$

An ideal high-pass filter will alter all Fourier coefficients that are at a distance less than R from the image center *(W/2, H/2)*, by turning them to 0.

$$X'_{uv} = \begin{cases} X_{uv}\ , & \left(\frac{H}{2}-u\right)^2+\left(\frac{W}{2}-v\right)^2 > R^2 \\ 0\ , & \left(\frac{H}{2}-u\right)^2+\left(\frac{W}{2}-v\right)^2 \le R^2 \end{cases} \tag{10.17}$$

The results of filtering with ideal low- and high-pass filtering are presented in Fig. 10.6 b) and c). Unfortunately, the corresponding spatial filters Fig. 10.6 e) and d) are not FIR (they have an infinite support) and keep oscillating away from their centers. Because of this, the low-pass and high-pass filtered images have a disturbing ringing wavy aspect. In order to correct this, the cutoff in the frequency domain must be smoother, as presented in the next section.

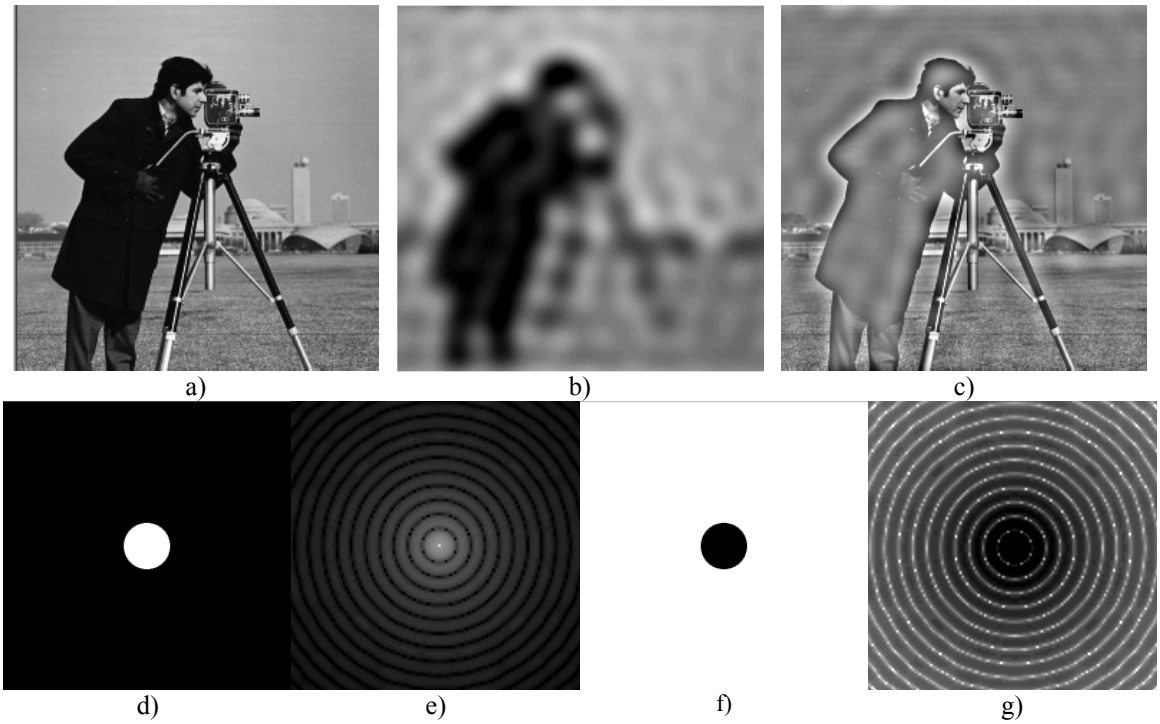


Fig.10.6 a) original image; b) result of ideal low-pass filtering; c) result of ideal high-pass filtering; d) ideal low-pass filter in the frequency domain; e) corresponding ideal low-pass filter in the spatial domain; f) ideal high-pass filter in the frequency domain; g) corresponding ideal high-pass filter in the spatial domain

10.3.3 Gaussian low-pass and high-pass filtering in the frequency domain

In the case of Gaussian filtering, the frequency coefficients are not cut abruptly, but smoother cutoff process is used instead. This also takes advantage of the fact that the DFT of a Gaussian function is also a Gaussian function (Fig. 10.7 d -g).

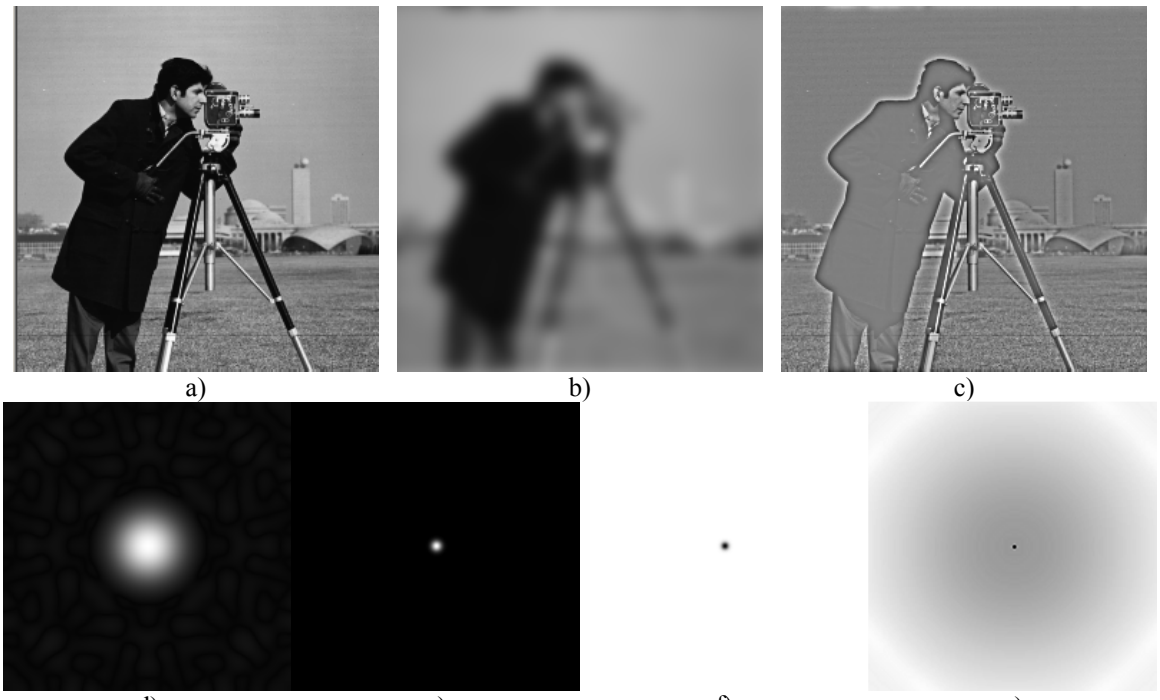
The Gaussian low-pass filter attenuates frequency components that are further away from the image center *(W/2, H/2)*. $A \sim \frac{1}{\sigma}$ where σ is the standard deviation of the equivalent spatial domain Gaussian filter.

$$X'_{uv} = X_{uv} e^{-\frac{\left(\frac{H}{2}-u\right)^2+\left(\frac{W}{2}-v\right)^2}{A^2}} \tag{10.18}$$

The Gaussian high-pass filter attenuates frequency components that are near to the image center *(W/2, H/2)*:

$$X'_{uv} = X_{uv}\left(1-e^{-\frac{\left(\frac{H}{2}-u\right)^2+\left(\frac{W}{2}-v\right)^2}{A^2}}\right) \tag{10.19}$$

Fig. 10.7**Error! Reference source not found.** shows the results of Gaussian filter. Notice that the ringing (wavy) effect visible in Fig. 10.6 disappeared.



a) b) c) d) e) f) g)

Fig.10.7 a) original image; b) result of Gaussian low-pass filtering; c) result of Gaussian high-pass Filtering; d) Gaussian low-pass filter in the frequency domain; e) corresponding Gaussian low-pass filter in the spatial domain; f) Gaussian high-pass filter in the frequency domain; g) corresponding Gaussian high-pass filter in the spatial domain

10.4 Implementation details

10.4.1 Spatial domain filters

Low-pass filters will always have positive coefficients, and therefore, the resulting filtered image will have positive values. You must ensure that the resulting image fits in the desired

range (0-255 in our case). In order to ensure this, you must ensure that the coefficients of a low-pass filter sum to 1. If you are using integer operations pay attention to the order of operations! Usually, the division should be the last operation performed in order to minimize the rounding errors!

High-pass filters will have both positive and negative coefficients. You must ensure that the final result is an integer between 0 and 255! There are three possibilities to ensure that the resulting image fits the destination range. The first one is to compute:

$$S_+ = \sum_{F_k>0} F_k, \quad S_- = \sum_{F_k<0} -F_k,$$
$$S = \frac{1}{2\max\{S_+, S_-\}} \tag{10.20}$$
$$I_D(u,v) = S(F * I_S)(u,v) + \left\lfloor \frac{L}{2} \right\rfloor$$

In the formula above S_+ represents the sum of positive filter coefficients and S_- the sum of negative filter coefficients magnitudes. This result of applying the high-pass filter always lies in the interval $[-LS_-, LS_+]$ where L is the maximum image gray level (255). The result of this transform will place scale the result to [*-L/2, L/2*] and then move the 0 level to *L/2*.

Another approach is to perform all operations using signed integers determine the minimum and maximum and then linearly transform the resulting values according to:

$$D = \frac{L(S - \min)}{\max - \min} \tag{10.21}$$

The third approach is to compute the magnitude of the result and saturate everything that exceeds the maximum level L.

10.4.2 Frequency domain filters

A library and a header file is supplied for performing the fast Fourier transform. The library is called “dibfft.lib” and the header file is called “dibfft.h”. In order to use the library file you should first copy the “dibfft.lib” and “dibfft.h” files to the “Diblook” folder.

Then right click on the “Diblook” project entry in the workspace window, select “Add > Existing Item…”. It will automatically open the window “Add Existing Item - DibLook”. You should select and add the file “dibfft.h” to the project.

Then the library “dibfft.lib” should be included in the project linker section. Perform right click on the “DibLook” project in the workspace window and select “Properties”. It will automatically open the window “DibLook Property Pages”. In the “Configuration” section choose “All Configurations” and then add the library “dibfft.lib” in the “Linker” section (see Fig. 10.8).

Finally you should add the header include “dibfft.h” in the “#include” section in the “dibview.cpp” file.

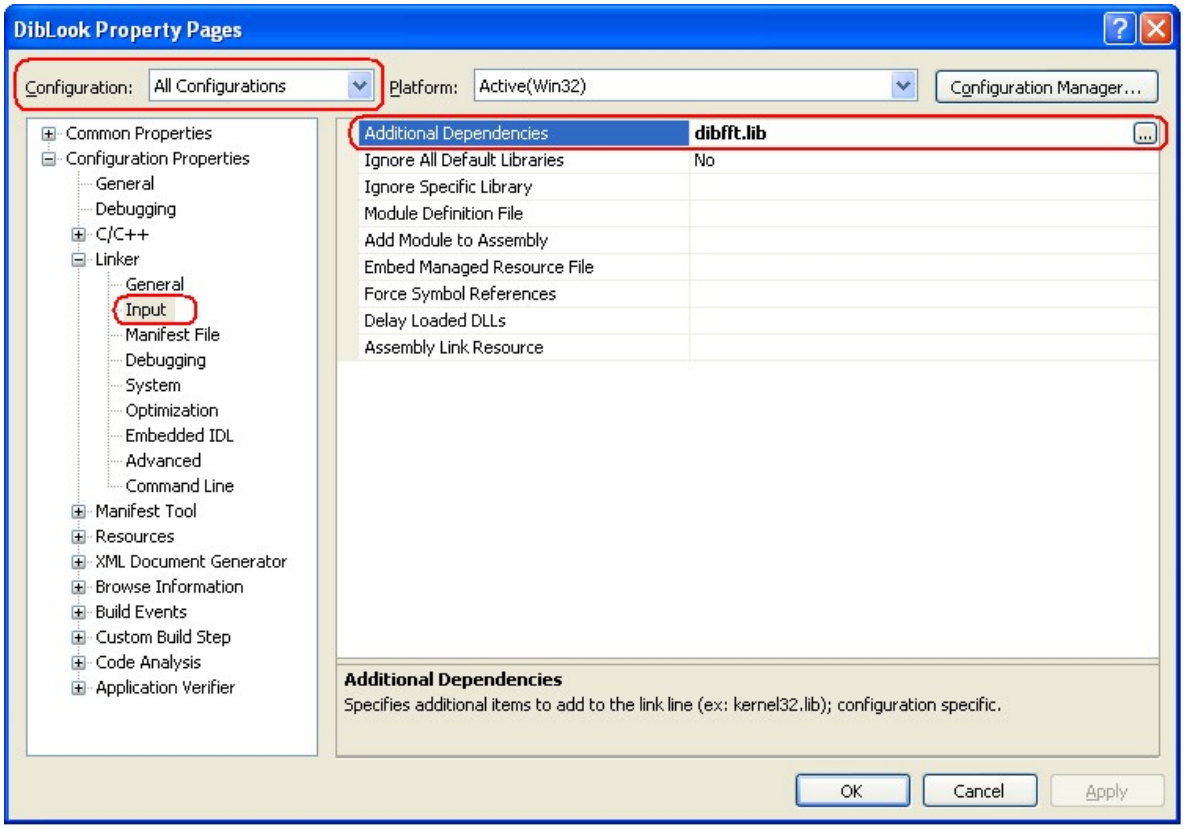


Fig. 10.8 Adding a library file to the project

The library provides the following functions:

```
/*The first two parameters are the image width & height
These functions only work correctly for width, height powers of 2 and >=4.
Parameters 3 and 4 are the input real and imaginary parts arrays
of width*height values. Acceptable value types are unsigned char (BYTE),
float and double. The imaginary part is optional (T*)0 can be provided  instead,
and the input will be assumed as consisting of values of 0.
Parameters 5 and 6 are the output real and imaginary parts. Acceptable value types
are unsigned char (BYTE), float and double. The imaginary part is optional (T*)0 can
be provided instead. The imaginary part of the output will be discarded in this
case.*/

/*perform FFT on image rows*/
template<class T> void fftrows(int width, int height, const T *ix, const T *iy, double
*ox, double *oy);

/*perform IFFT on image rows*/
template<class T> void ifftrows(int width, int height, const double *ix, const double
*iy, T *ox, T *oy);

/*perform FFT on image cols*/
template<class T> void fftcols(int width, int height, const T *ix, const T *iy, double
*ox, double *oy);



/*perform IFFT on image cols*/
template<class T> void ifftcols(int width, int height, const double *ix, const double
*iy, T *ox, T *oy);

/*perform FFT on image*/
template<class T> void fftimage(int width, int height, const T *inpx, const T* inpy,
double *ox, double *oy);
```

```
/*perform IFFT on image*/
template<class T> void ifftimage(int width, int height, const double *ix, const double
*iy, T *outpx, T *outpy);
```

The functions provided are template based and work for BYTE, float and double inputs/outputs. Imaginary parts are optional for both input and output. Use NULL pointers to specify missing inputs/outputs.

The following code gives an example of FFT followed by an IFFT. The original image should be recovered:

```
BEGIN_PROCESSING();
double *real= new double[dwWidth*dwHeight];
double *imag= new double[dwWidth*dwHeight];
fftimage(dwWidth, dwHeight, lpSrc, (BYTE*)0, real, imag);
ifftimage(dwWidth, dwHeight, real, imag, lpDst, (BYTE*)0);
END_PROCESSING("FFT");
```

A few important aspects of working with frequency domain values:

1. Always use for FFT only grayscale images having both width and height powers of two (example: image “cameraman.bmp” having width=height=256=2^8 pixels)

2. Always perform the centering transform (9.15) before performing the FFT and after performing the IFFT:

```
for(int i=0; i<dwHeight; i++)
       for(int j=0; j<dwWidth; j++)
              D[i*w+j] = ((i+j)&1)?-S[i*w+j]:S[i*w+j];
```

3. The DC (0,0) Fourier coefficient highly dominates the other ones. When displaying the magnitude of Fourier coefficients it is better to use the logarithm of the module+1! You should determine the maximum value of the logarithm and scale the remaining values to fit the range (0-255).

4. The Fourier transform of an image is a complex array of floating point values! Store both real and imaginary values as floating points! When converting back to the spatial domain the imaginary values may be discarded (for usual filters they should be 0 anyway).

10.5 Practical work

1. Implement the convolutions with the kernels from equations (10.5) .... (10.10).
2. Implement a customized convolution operator of size 3x3 using values specified by the user in a dialog box. The scaling coefficient should be computed automatically as either the reciprocal of filter coefficient sum for low pass filters or according to equation for high-pass filters.
3. Import the *dibfft* library into Diblook. Add a processing function that performs the FFT transform of an input image and the transforms the result back to the spatial domain using IFFT. Check if the destination is the same as the source!
4. Add a processing function that computes and displays the logarithm of the magnitude of the Fourier transform of an input image.

5. Add processing functions that perform low- and high-pass filtering in the frequency domain using the ideal and Gaussian filters from equations (10.16) ... (10.19).
6. **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.**

10.6 References

[1]. Umbaugh Scot E, *Computer Vision and Image Processing*, Prentice Hall, NJ, 1998, ISBN 0-13-264599-8
[2] R.C.Gonzales, R.E.Woods, *Digital Image Processing, 2-nd Edition,* Prentice Hall, 2002

11 Noise modeling and digital image filtering

11.1 Introduction

Noise represents unwanted information which deteriorates image quality. Noise is defined as a process (n) which affects the acquired image (f) and is not part of the scene (initial signal – s). Using the additive noise model, this process can be written as:

$$f(i,j) = s(i,j) + n(i,j) \tag{11.1}$$

Digital image noise may come from various sources. The acquisition process for digital images converts optical signals into electrical signals and then into digital signals and is one processes by which the noise is introduced in digital images. Each step in the conversion process experiences fluctuations, caused by natural phenomena, and each of these steps adds a random value to the resulting intensity of a given pixel.

11.2 Noise modeling

Noise (n) may be modeled either by a histogram or a probability density function which is superimposed on the probability density function of the original image (s). In the following, the models for the most common types of noise will be presented: salt and pepper noise and Gaussian noise. Other types of noise, such as negative exponential model, gamma/Erlang model, Rayleigh model are also presented in the literature (see the course notes!).

11.2.1 The salt & pepper noise

In the salt & pepper noise model only two possible values are possible, a and b, and the probability of obtaining each of them is less than 0.1 (otherwise, the noise would vastly dominate the image). For an 8 bit/pixel image, the typical intensity value for pepper noise is close to 0 and for salt noise is close to 255.

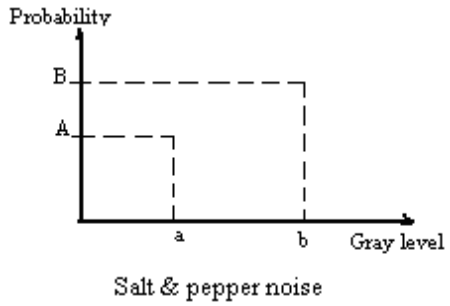


Fig. 111.1 Probability density function for the salt & pepper noise model.

$$PDF_{Salt\&pepper} = \begin{cases} A & for\ g = a\ (\text{"}pepper\text{"}) \\ B & for\ g = b\ (\text{"}salt\text{"}) \end{cases} \tag{11.2}$$

The salt & pepper noise is generally caused by malfunctioning of camera's sensor cells, by memory cell failure or by synchronization errors in the image digitizing or transmission.

11.2.2 Gaussian noise

The Gaussian noise has a normal (Gaussian) probability density function:

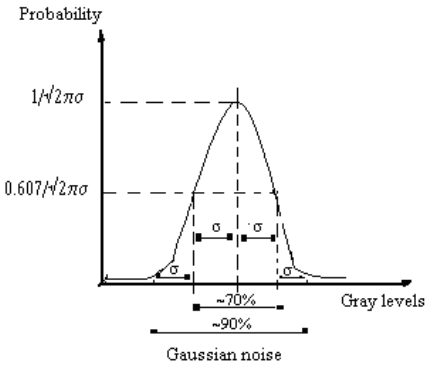


Fig. 111.2 Probability density function for the Gaussian noise model

$$PDF_{Gaussian} = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(g-\mu)^2}{2\sigma^2}} \tag{11.3}$$

where:

g = gray level;
μ = mean;
σ = standard deviation;

Approximately 70% of the values are contained between $\mu \pm \sigma$ and 90% of the values are contained between $\mu \pm 2\sigma$. Although, theoretically speaking, the PDF is non-zero everywhere between -∞ and +∞, it is customary to consider the function 0 beyond $\mu \pm 3\sigma$.

Gaussian noise is useful for modeling natural processes which introduce noise (e.g. noise caused by the discrete nature of radiation and the conversion of the optical signal into an electrical one – detector/shot noise, the electrical noise during acquisition – sensor electrical signal amplification, etc.).

11.3 Noise removal using spatial filters

11.3.1 Ordered filters (non-linear)

Ordered filters are based on a specific image statistic, called ordered statistic. They are called non-linear, because they cannot be applied as a linear operator (such as a convolution kernel). These filters operate on small windows, and replace the value of the central pixel (similarly to convolution). The ordered statistic is a technique which arranges all the pixels in sequential order, based on their gray-level value. The position of an element in this ordered set can be characterized by its rank. Given a NxN window W, the pixel values can be sorted in ascending order:

$$I_1 \le I_2 \le I_3 \le \ldots \le I_{N^2} \tag{11.4}$$

where:

$\{ I_1, I_2, I_3, \ldots, I_{N^2} \}$ represent the intensity values of the pixels located within the NxN window W.

For example: given a 3x3 window:

$$\begin{bmatrix} 110 & 110 & 114 \\ 100 & 106 & 104 \\ 95 & 88 & 85 \end{bmatrix}$$

The result of applying the ordered statistic will be:

{85, 88, 95, 100, 104, 106, 110, 110, 114}

The median filter: selects the middle value from the ordered statistic and replaces the destination pixel with it. In the example above, the selected value would be 104. The median filter allows the elimination of *salt & pepper* noise.

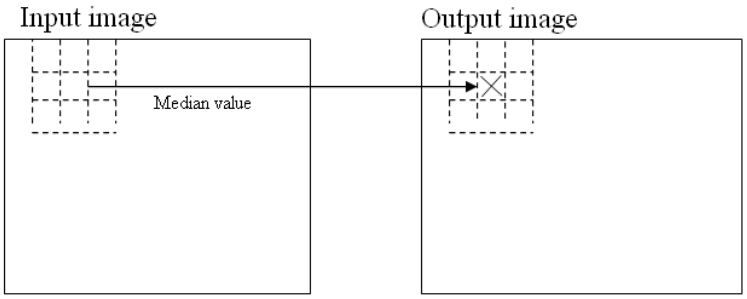


Fig. 111.3 Applying the median filter

The maximum filter: selects the largest value amongst the ordered values of pixels from the window. In the above example, the value selected is 114. This filter can be used to eliminate the *pepper noise*, but it amplifies the *salt* noise if applied to a salt & pepper noise image.

The minimum filter: selects the smallest value amongst the ordered values of pixels from the window. In the above example, the value selected is 85. This filter can be used to eliminate the *salt noise*, but it amplifies the *pepper* noise if applied to a salt & pepper noise image.

11.3.2 Linear filters

These filters are applied by convolution (a linear operation) with a low-pass filter convolution kernel. In the following, the computation of the elements of a convolution kernel for Gaussian noise elimination will be presented.

11.3.3 Designing a variable size Gaussian convolution kernel

Gaussian noise removal must be performed using a filter with adequate shape and size, correlated to the amount of the Gaussian noise that corrupts the image (see Fig. 111.2). The filter size *w* of such a filter is usually 6σ (for example, for a Gaussian noise with σ=0.8 $\Rightarrow$ w = 4.8 $\approx$ 5).

Constructing the elements of such a kernel/Gaussian filter G will be performed using the following equations:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}} \tag{11.5}$$

where:
(x0,y0) – are the coordinates of the central row and column of the kernel (see Fig. 111.4).

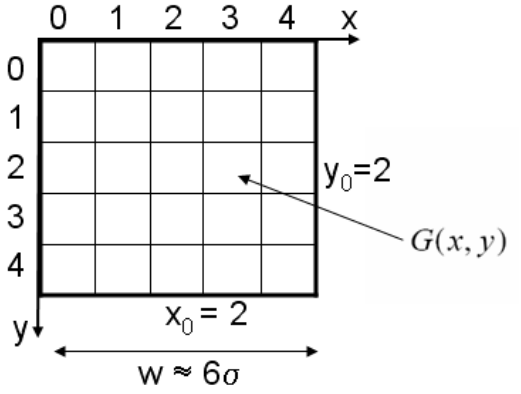


Fig. 111.4 Design example of a Gaussian kernel/filter G having a 5x5 size.

11.3.4 Image filtering/restoration

It is accomplished by the convolution of the source image with a Gaussian kernel/filter computed previously:

$$I_D = G * I_S \tag{11.6}$$

When the filter size *w* is large, the convolution may be time consuming (*w* x *w* multiplications for each pixel). In this case, the Gaussian decomposition may be used:

$$G(x,y) = G(x)G(y) \tag{11.7}$$

and replacing the convolution of a 2D nucleus *G* with two convolutions of a 1D nucleus G_x and G_y:

$$I_D = \left(G_x G_y\right) * I_S = G_x * \left(G_y * I_S\right) \tag{11.8}$$

where:
Gx and Gy are the central line and column vectors of the 2D nucleus (Fig. 111.5):

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-x_0)^2}{2\sigma^2}} \tag{11.9}$$

$$G(y) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y-y_0)^2}{2\sigma^2}} \tag{11.10}$$

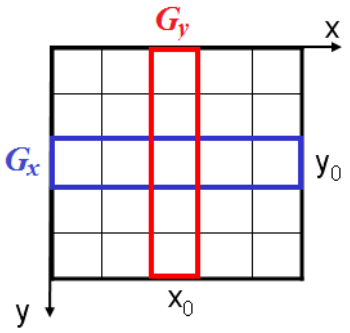


Fig. 111.5 The two vectors Gx and Gy into which a 2D Gaussian kernel may be separated

In this case, the number of multiplications needed for each pixel is w for each of the two convolutions.

11.4 Practical work

1. Implement a median filter with a variable dimension (w = 3, 5 or 7) specified by the user.
2. Implement the filtering operation with a 2D Gaussian filter, with standard deviation σ and variable size with w (w = 3, 5 or 7), specified by the user. The values of the kernel's components will be automatically computed as a function of σ, as in equation (11.5).
3. Implement Gaussian filtering by using a Gaussian kernel separated into 2 vector components G_x and G_y having a standard deviation of σ and a variable size w (w = 3, 5 or 7), specified by the user. The vector components values G_x and G_y will be computed automatically as a function of σ, as in equations (11.9) and (11.10).
4. **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.**

11.5 References

[1]. R.C.Gonzales, R.E.Woods, *Digital Image Processing. 2-nd Edition*, Prentice Hall, 2002.

12 Edge detection

12.1 Introduction

This laboratory presents the edge detection problem in digital images. Edge points are found where the image intensity encounters a steep variation along a specific direction 'x' (Fig.12.1). This intensity variation can be detected and quantified by finding the local maxima of the first order derivative of the image intensity (the gradient: $\nabla f=f'$) or by finding the zero crossings of the second order derivative of the image intensity (the laplacian: $\nabla^2 f=f''$).

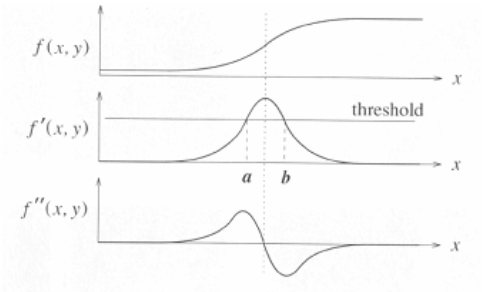


Fig.12.1 Detection methods of the edge points (points were the image intensity suffers a steep variation).

Further on only the gradient based methods will be approached.

12.2 Computing the image gradient

The gradient in an image point is a vector heading the direction of the intensity variation around this point (Fig. 12.2). Its module is proportional with the speed of this variation (12.1). If the edge points are part of a contour (as in Fig. 12.2) the gradient will be perpendicular on the tangent to the contour at that point.

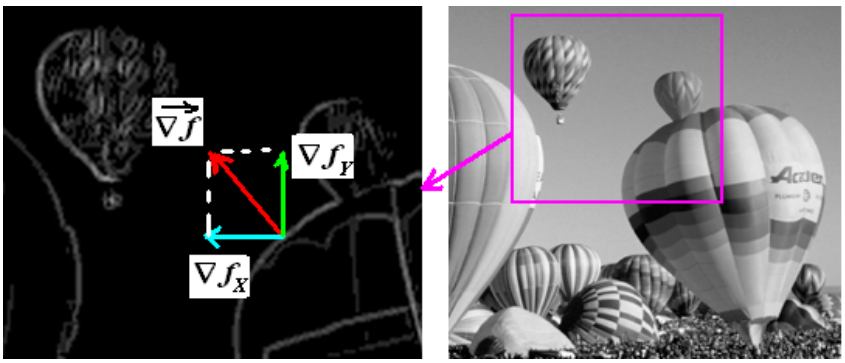


Fig. 12.2 Left: illustration of the image gradient (in an edge point) on the image of the gradient module.

The gradient of a two variables continuous function f is defined as:

$$\nabla f(x,y)=\begin{bmatrix}\nabla f_x\\ \nabla f_y\end{bmatrix}=\begin{bmatrix}\frac{\partial f}{\partial x}\\ \frac{\partial f}{\partial y}\end{bmatrix}=\begin{bmatrix}\lim\limits_{\Delta x\to 0}\frac{f(x+\Delta x,y)-f(x,y)}{\Delta x}\\ \lim\limits_{\Delta y\to 0}\frac{f(x,y+\Delta y)-f(x,y)}{\Delta y}\end{bmatrix} \tag{12.1}$$

For digital images, the gradient can be approximated by making Δx and Δy equal to 1 in (12.1):

$$\nabla f(x,y) = \begin{bmatrix} \nabla f_x \\ \nabla f_y \end{bmatrix} = \begin{bmatrix} f[x+1,y] - f[x,y] \\ f[x,y+1] - f[x,y] \end{bmatrix} \tag{12.2}$$

Other approximations of the two components of the gradient can be computed through the convolution of the image with the following kernels:

Prewitt:

$$\begin{aligned} \nabla f_x &= f(x,y) * \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \\ \nabla f_y &= f(x,y) * \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \end{aligned} \tag{12.3}$$

Sobel:

$$\begin{aligned} \nabla f_x &= f(x,y) * \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \\ \nabla f_y &= f(x,y) * \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \end{aligned} \tag{12.4}$$

Roberts (cross):

$$\begin{aligned} \nabla f_x &= f(x,y) * \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \\ \nabla f_y &= f(x,y) * \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \end{aligned} \tag{12.5}$$

As a vector, the gradient can be quantified by a magnitude (12.6) and a direction (12.7):

$$\left|\nabla f(x,y)\right| = \sqrt{\left(\nabla f_x(x,y)\right)^2 + \left(\nabla f_y(x,y)\right)^2} \tag{12.6}$$

$$\theta(x,y) = arctg\left(\frac{\nabla f_y(x,y)}{\nabla f_x(x,y)}\right) \tag{12.7}$$

12.3 Practical work

1. The horizontal ∇f_x and vertical ∇f_y components of the gradient through convolution with the kernels given in (12.3) ... (12.5) will be computed and the results will be shown in the destination image (the convolution operation was already implemented in Laboratory 9).
2. The gradient magnitude (12.6) and direction (12.7) will be computed using the three operators (Sobel, Prewitt and Roberts) and the results of the gradient magnitude will be shown in the destination image.
3. The thresholding with an arbitrary and fixed threshold of the results obtained at point 2 will be shown in the destination image.
4. **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.**

12.4 References

[1] E.Trucco, A.Verri, *Introductory Techniques for 3-D Computer Vision,* Prentice Hall, 2001
[2] R.C.Gonzales, R.E.Woods, *Digital Image Processing, 2-nd Edition,* Prentice Hall, 2002

13 The Canny edge detection method

13.1 Introduction

The edge detection method proposed by Canny is based on the image gradient computation but in addition tries to:

- maximize the signal-to-noise ratio for a proper detection;
- find a good localization of the edge points;
- minimize the number of positive responses around a single edge (suppression of the gradient module non-maxima)

13.2 The steps of the Canny edge detection method:

1. Noise filtering through a Gaussian kernel
2. Computing the gradient's module and direction
3. Non-maxima suppression of the gradient's module
4. Edge linking through adaptive hysteresis thresholding.

13.2.1 Noise filtering through a Gaussian kernel

The noise in the image is high frequency information which overlaps the original image signal. This introduces false edge points. The noise intrinsic to the image acquisition process can be modeled by a Gaussian distribution and can be suppressed by a Gaussian filter (see Laboratory 10).

13.2.2 Computing the gradient's magnitude and direction

Computing the gradient's module and direction requires the allocation of two temporary image buffers (with the same size as the image) and the initialization of their elements according to equations (12.6) and (12.7) respectively, where the horizontal $\nabla f_x(x,y)$ and the vertical $\nabla f_y(x,y)$ components of the image gradient can be computed using the Prewitt operator (12.3) or the Sobel operator (12.4).

13.2.3 Non-maxima suppression of the gradient's module

Its purpose is the thinning of the edges by retaining only the edge points with the highest gradient module along the direction of the image intensity variation (along the direction of the gradient vector).

The first step consists in the quantization of the gradient directions, computed using (12.7), in 4 regions shown in Fig. 13.1:

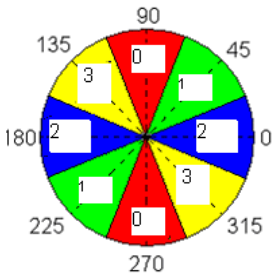


Fig. 13.1 Quantization of the gradient directions in the non-maxima suppression step

Supposing that, for example, the direction of the gradient in an image point is "1" (Fig. 13.2), the module of the gradient in point P is a local maximum if: $|\nabla P| > |\nabla I_6|$ and

$|\nabla P| > |\nabla I_2|$. If these conditions are fulfilled, the point P is retained as an edge point, otherwise is rejected.

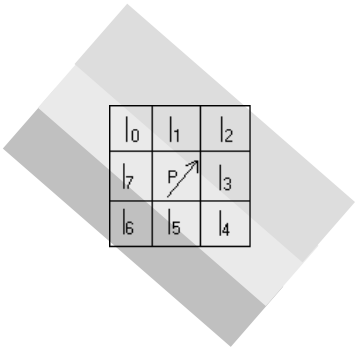


Fig. 13.2 Example for the non-maxima suppression.

13.2.4 Edge linking through adaptive hysteresis thresholding

After computing the image gradient and performing the non-maxima suppression procedure, an "image" is obtained in which the pixel values are equal with the gradient's modules in that pixel. Moreover, the thickness of the edge pixels (with non-zero module) has an ideal value of one pixel. In the following, the steps required to obtain the final edges are described:

13.3 Adaptive thresholding

Adaptive thresholding tries to extract a quite constant number edge points for a given image size. In this way, lighting and contrast variations are compensated (fixed threshold would extract either too much or too few edge points).

The parameter which is given to the threshold detection procedure is the ratio between the number of edge points and the number of points with non-zero gradient module:

$$NoEdgePixels = p * (NoPixels - ZeroGradientModulePixels) \quad (13.1)$$

Parameter *p* has usually values between 0.01 and 0.1.

The algorithm is the following:

1. The histogram of the gradient's magnitude image (values obtained after non-maxima suppression) is computed. These values will be scaled to fit within [0..255] range (by division with $4\sqrt{2}$ if the gradient was computed using the Sobel operator). The result is a histogram Hist[0..255]:

$$Hist[i] = No\ of\ pixels\ having\ the\ scaled\ gradient\ magnitude\ value\ i \quad (13.2)$$

2. The number of pixels with non-zero values which would not be edge points is computed:

$$NoNonEdge = (1\text{-}p) * (Height*Width - Hist[0]) \quad (13.3)$$

3. Starting with position 1 the values of the histogram are summed. When the sum exceeds the value *NoNonEdge*, then the index *i* reached in the counting process is the searched threshold.

This technique, intuitively, will find the gradient magnitude value (*AdaptiveThresholding*) bellow which *NoNonEdge* pixels are found.

Pay attention to the pixels located at the image margins (where the image gradient was not computed)! Their values should be zero or should not be taken into account, because they can modify the value of the threshold.

13.4 Edge extension through hysteresis

Adaptive thresholding does not guarantees the completeness of the edges (shadowed parts of the objects or presence of noise can affect the edge detection process). The result will be an image with many fragmented edges.

Therefore an edge extension technique is needed. The edges obtained by adaptive thresholding are considered STRONG EDGES and we try to extend them with weaker edge points, which have not passed the thresholding with the initial value, but could be detected with a lower threshold.

Formally, two thresholds are defined:

$$Threshold\_high = AdaptiveThresholding \quad (13.4)$$

$$Threshold\_low = k * Threshold\_high \quad (13.5)$$

where k<1 (for example, $k = 0.4$).

The image of the gradient module is scanned pixel by pixel. In the destination image the pixels with the gradient magnitude higher then *Threshold_high* are labeled as STRONG_EDGES (e.g. with the value 255). The pixels with the gradient magnitude between *Threshold_low* and *Threshold_high* are labeled as WEAK_EDGES (e.g. with the value 128).

The pixels with the gradient magnitude bellow *Threshold_low* are considered NON-EDGES and are rejected. The inverted result (negative) of this labeling is shown in Fig. 13.3-left.

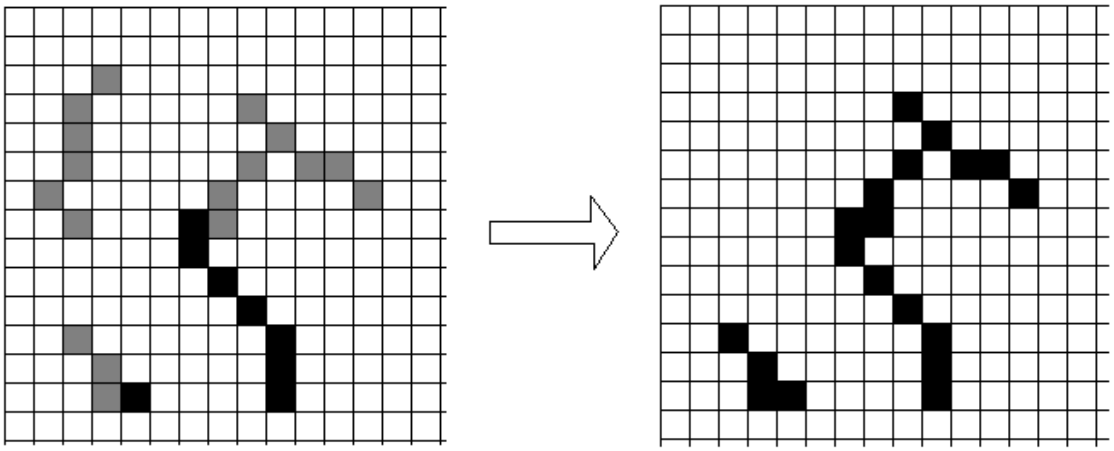


Fig. 13.3 Left: the image of the labeled strong and weak edges; Right: the result of the extension of the strong edges with connected WEAK edges.

Next step consists in the extension of the STRONG_EDGE points with neighboring WEAK_EDGE points, if they are parts of a connected component (see laboratory and lecture related to “Labeling”) – as in Fig. 13.3. If a STRONG_EDGE point has WEAK_EDGE

neighbor, the WEAK_EDGE neighbor is labeled as a STRONG_EDGE point. This STRONG_EDGE becomes a new source of edge extension. The process is repeated until the STRONG_EDGE points cannot be extended further by joining them with WEAK_EDGE points.

An efficient implementation of this step uses a queue to perform a breadth first search through WEAK_EDGE points connected to STRONG_EDGE points and mark them as STRONG_EDGE points. The algorithm would look like this:

1. Scan the image, top left to bottom right, pick the first STRONG_EDGE point encountered and push its coordinates in the queue.
2. While (queue is not empty)
    a) Extracts the first point from the queue
    b) Find all the WEAK_EDGE neighbors of the current point
    c) Label in the image all these neighbors as STRONG_EDGE points
    d) Push the image coordinates of these neighbors into the queue
    e) Continue to the next STRONG_EDGE point
3. Go to step 1 considering the next STRONG_EDGE point.
4. Eliminate the remaining WEAK_EDGE points from the image by turning them to NON_EDGE (0)

Final consideration: regarding the definition of the neighborhood used in the above algorithm, the common 4-type or 8-type neighborhood can be used, or a tolerance of 1 to 2 pixels can be considered. The reason is that, due to noise, the edges may be interrupted by small gaps.

13.5 Practical work

1. The steps 1 – 3 of the Canny edge detection algorithm will be implemented
    a. step 1 – was already implemented in laboratory 10;
    b. step 2 – is the implementation from point 2 using the Sobel filters;
    c. step 3 – implement the non-maxima suppression operation.
    The results obtained after step 3 will be shown in the destination image. The results will be compared with the one obtained at point 2 after the simple use of the Sobel operator.
2. Edge linking through adaptive hysteresis thresholding algorithm (step 4 of the Canny method) will be implemented. The intermediate results of the STRONG_EDGE and WEAK_EDGE points (after the hysteresis thresholding and before the edge extension step) and the final results (with the final edges) will be shown in the destination image. The implementation will be experimented for different values of the parameters *p, k* and neighborhood types.
3. The final results of the implemented Canny edge detection method will be tested/experimented on different image types.
4. **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.**

13.6 References

[1] E.Trucco, A.Verri, *Introductory Techniques for 3-D Computer Vision,* Prentice Hall, 2001
[2] R.C.Gonzales, R.E.Woods, *Digital Image Processing, 2-nd Edition,* Prentice Hall, 2002

14 Color image processing

14.1 Introduction

This laboratory makes an introduction to color image processing and presents some basic processing in RGB and HSI color spaces. First, the RGB to HSI and HSI to RGB transforms are defined. Then, the following color image processing are presented:

- contrast enhancement,
- edge detection,
- and color segmentation.

14.2 HSI Colorspace

The HSI color space (see Fig. 14.1) separates the color information from the intensity component in a color image, being an ideal image representation for color based processing that are intuitive to human eyes. HSI color space describes a color object in terms of hue (H), saturation (S) and brightness or intensity (I):

- Hue is a color attribute that describes a pure color (and not a combination of 3 colors like in RGB color space)
- Saturation is a measure of the degree to which a pure color is diluted by white light
- Brightness is a subjective descriptor for color sensation that is hard to measure. The most suitable and easiest solution is to define it as the gray level intensity since is a useful descriptor and easily interpretable.

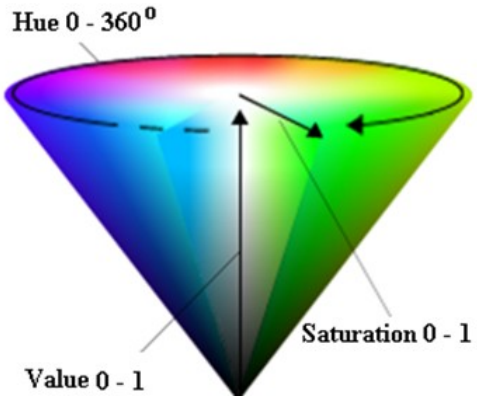


Fig.14.1. HSI Colorspace [2]

14.2.1 RGB to HSI Transform

Given an RGB image (with $\boldsymbol{R}, \boldsymbol{G}, \boldsymbol{B} \in [\mathbf{0}, \ldots, \mathbf{1}]$), the H, S and I components are computed as follows:

$$H = \begin{cases} \theta, & if\ B \le G \\ 360 - \theta, & if\ B > G \end{cases}, \quad \text{where } \theta = \operatorname{acos}\left\{ \frac{\frac{1}{2}[(R-G)+(R-B)]}{[(R-G)^2+(R-B)(G-B)]^{1/2}} \right\} \tag{14.1}$$

$$S = 1 - \frac{3}{(R+G+B)}[\min(R,G,B)] \tag{14.2}$$

$$I = \frac{1}{3}(R+G+B) \tag{14.3}$$

The domain values are:
- $\boldsymbol{H} \in [\boldsymbol{0}, \boldsymbol{2\pi})$
- $\boldsymbol{S} \in [\boldsymbol{0}, \boldsymbol{1})$
- $\boldsymbol{V} \in [\boldsymbol{0}, \boldsymbol{1})$

Observation: Hue component might present singularities when all components have small values R=G=B≈0, case when H is undefined. Consequently, computing H in areas with low intensity will lead to numerical errors. The solution is to ignore those areas and set them to 0.
Ex: If (R<3 && G<3 && B<3) then H=0

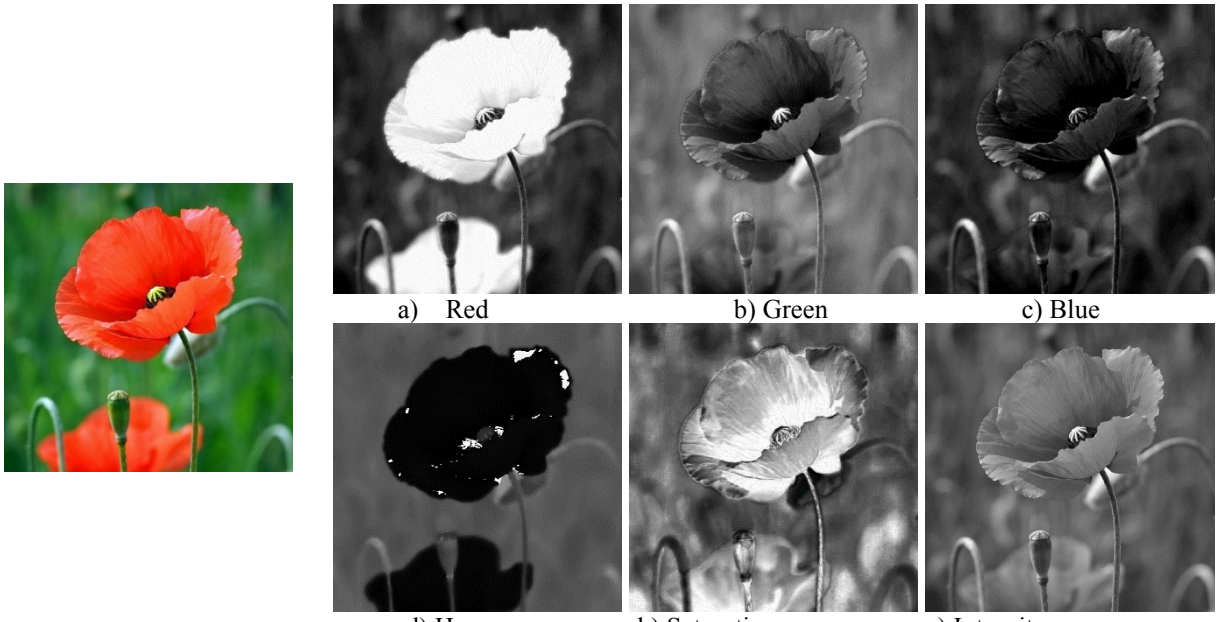


Fig. 14.2. **RGB to HSI**: [left] Original image. [right] RGB channels in the first row and HSI channels in the second row

Observation: The H, S, I values need to be normalized to [0, 255] in order to display the results.

14.2.2 HSI to RGB Transform

The reverse transform from HSI to RGB if performed as follows.
Given an image in HSI color space (with $H \in [0, 2\pi)$ and $S, I \in [0,1]$), the R,G and B components are defined as follows:

If $\boldsymbol{0^o \leq H < 120^o * 180/\pi}$ (RG sector):

$$B = I(1 - S) \tag{14.4}$$

$$R = I\left[1 + \frac{S \cos H}{\cos(60^o - H)}\right] \tag{14.5}$$

$$G = 3I - (R + B) \tag{14.6}$$

If $\boldsymbol{120^o * 180/\pi \leq H < 240^o * 180/\pi}$ (GB sector):
If H is in this sector, we first have to perform: $H = H - 120^o$

$$R = I(1 - S) \tag{14.7}$$

$$G = I\left[1 + \frac{S\cos H}{\cos(60^o - H)}\right] \tag{14.8}$$

$$B = 3I - (R + G) \tag{14.9}$$

If $240^o * 180/\pi \leq H < 2\pi$ (BR sector):
If H is in this sector, we first have to perform: $H = H - 240^o$

$$G = I(1 - S) \tag{14.10}$$

$$B = I\left[1 + \frac{S\cos H}{\cos(60^o - H)}\right] \tag{14.11}$$

$$R = 3I - (G + B) \tag{14.12}$$

14.3 Color image processing

14.3.1 Modifying Contrast in color images

As presented in chapter 9, some approaches for contrast modification are: histogram stretching/ shrinking and histogram equalization. In color images, these operations imply the modification in image brightness and contrast, without influencing the hue and saturation. Since only the intensity component should be affected, the HSI color space seem the right choice in changing contrast in color images.

Therefore, in order to modify the contrast in color image, the following steps need to be performed:
- Convert the source image from RGB to HSI
- Compute histogram stretching/ shrinking or histogram equalization operation on the I channel,
- Convert the image from HSI (with the modified I channel) to RGB

We recall the formulation for contrast modification based on histogram processing. See Laboratory 8 for more details.

- **Histogram stretching / shrinking:**

$$g_{out} = g_{out}^{MIN} + (g_{in} - g_{in}^{MIN})\frac{g_{out}^{MAX} - g_{out}^{MIN}}{g_{in}^{MAX} - g_{in}^{MIN}}, \tag{14.13}$$

where: $\frac{g_{out}^{MAX} - g_{out}^{MIN}}{g_{in}^{MAX} - g_{in}^{MIN}} = \begin{cases} > 1 & \Rightarrow \quad stretch \\ < 1 & \Rightarrow \quad shrink \end{cases}$

- **Histogram equalization**

$$g_{out} = 255\, p_C(g_{in}), \tag{14.14}$$

where

g_{in}, g_{out} – are the gray values in the input and output respectively images (**in this case, they correspond to the input/output values of I channel**).

p_C – is the cumulative histogram

Fig. 14.3 show the visual result for contrast modification using histogram shrinking (middle image) and histogram equalization (right image). The corresponding histograms of I channel are also shown in the second row of the figure.

Fig. 14.3. Modifying contrast in color images: a) Original image; b) The result of histogram shrinking ($g_{out}^{MIN} = 100$, $g_{out}^{MAX} = 200$); c) The result of histogram equalization.

14.3.2 Color edge detection

Generally, in color images the gradients are computed as a composite gradient image that lead to a more complete edge detail than the gradients computed in single RGB channels.

The direction and magnitude of the composite gradient in RGB images is computed as follows:

$$u = \frac{\partial R}{\partial x} r + \frac{\partial G}{\partial x} g + \frac{\partial B}{\partial x} b$$

$$v = \frac{\partial R}{\partial y} r + \frac{\partial G}{\partial y} g + \frac{\partial B}{\partial y} b$$

$$g_{xx} = u \cdot u = \left|\frac{\partial R}{\partial x}\right|^2 + \left|\frac{\partial G}{\partial x}\right|^2 + \left|\frac{\partial B}{\partial x}\right|^2$$

$$g_{yy} = v \cdot v = \left|\frac{\partial R}{\partial y}\right|^2 + \left|\frac{\partial G}{\partial y}\right|^2 + \left|\frac{\partial B}{\partial y}\right|^2$$

$$g_{xy} = u \cdot v = \left|\frac{\partial R}{\partial x}\right|\left|\frac{\partial R}{\partial y}\right| + \left|\frac{\partial G}{\partial x}\right|\left|\frac{\partial G}{\partial y}\right| + \left|\frac{\partial B}{\partial x}\right|\left|\frac{\partial B}{\partial y}\right|$$

$$\textbf{Direction}: \theta = \frac{1}{2}\tan^{-1}\left[\frac{2g_{xy}}{(g_{xx}-g_{yy})}\right] \quad (14.15)$$

$$\textbf{Magnitude}: F(\theta) = \left\{\frac{1}{2}\left[\left(g_{xx}+g_{yy}\right)+\left(g_{xx}-g_{yy}\right)\cos 2\theta + 2g_{xy}sin2\theta\right]\right\}^{\frac{1}{2}} \quad (14.16)$$

Implementation details:

- Compute the derivatives on x and y axes for each R,G,B channel by filtering the image with the following convolution kernels:
  $D_x = [-1\ 0\ 1]$ and $D_y = \begin{bmatrix} 1 \\ 0 \\ -1 \end{bmatrix}$
- Use function **atan2** $\in [-\pi, \pi]$ to compute $\tan^{-1}$.

The edge detection result is shown in Fig. 14.4.

Fig. 14.4. Edge detection in color images: a) Original image; b) Magnitude $F(\theta)$

14.3.3 Color segmentation

Segmentation is an important topic in color image processing. The HSI color space is more suitable for color segmentation than RGB due to the fact that color separated in one channel (hue).

Assuming that the target object have a specific color that can be modeled and learned a priori (e.g. road signs), a fast segmentation method is as follows.

Considering a Gaussian model for the target color with parameters μ_H and σ_H, the segmentation is performed by classifying each pixel in the image in two categories: **object** and **non-object** by verifying the condition:

- if $\boldsymbol{H} \in [\boldsymbol{\mu_H} - \boldsymbol{k}\cdot\boldsymbol{\sigma_H},\ \boldsymbol{\mu_H} + \boldsymbol{k}\cdot\boldsymbol{\sigma_H}]$ then the pixel is classified as object (R=G=B=0), else the pixel is classified as non-object (R=G=B=255).
  Choose values for k in the interval [2,…,3].

Fig. 14.5 show the visual representation of the segmentation result.

Fig. 14.5. The segmentation result for a given model of parameters: $\mu_H = 4.2 * \frac{2\pi}{255}$ and $\sigma_H = 1.3 * \frac{2\pi}{255}$, and $k = 3$

14.4 Practical work

1. Convert an RGB image to HSI color space. Display each channel in different processing function.
2. Add a processing function for contrast modification (ex. Histogram stretching/ shrinking or histogram equalization) in HSI color space.
3. Add a processing function for color edge detection in RGB color space.
4. Implement the color segmentation method presented in section 14.3.3 for the following color model $\mu_H = 4.2 * \frac{2\pi}{255}$ and $\sigma_H = 1.3 * \frac{2\pi}{255}$.
5. **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.**

14.5 References

[1] R.C.Gonzales, R.E.Woods, *Digital Image Processing, 2-nd Edition,* Prentice Hall, 2002.
[2]. Microsoft – Developer Technologies: https://msdn.microsoft.com/en-us/library/windows/desktop/dn742482.aspx

Appendix I. Image processing in MATLAB

1. Introduction

MATLAB environment offers an easy way to prototype applications that are based on complex mathematical computations. This annex presents some basic image processing operations that can run in MATLAB. MATLAB has the great advantage that is a matrix oriented environment. All variables in MATLAB are actually arrays. Scalar values are 1x1 matrices. Common operations on matrices include addition, subtraction, logical operations, multiplication, division, matrix transpose, determinant, inverse matrix, eigenvalues and eigenvectors, etc. A very important and powerful aspect in MATLAB is that most operations can be performed either element by element or directly on the whole matrix. For example, multiplication can be applied in the mathematical sense of matrix multiplication (a non-commutative operation) and also in the sense of scalar multiplication, where the multiplication is performed element by element. The latter option is particularly useful in the image processing field where most operations are performed at pixel or pixel neighbor's level.

A very important aspect is that, unlike C language, indexing starts at value 1 and matrix elements are organized on lines and then on columns. The element (i, j) of the matrix A, the element located at row *i* and column *j*.

2. Read an image

Imread MATLAB function reads images in different formats, the result being a matrix. The function call can be performed as following:

```
I = imread ('nume.bmp');
```

or

```
I = imread ('nume.jpg','jpg');
```

The first string can contain also the image path.

3. Display an image

Imshow function displays an image into a graphic handler. Function call:

```
imshow (I);
```

In case when an image is already displayed, then, by calling the ***imshow*** function, the image will be displayed in the opened figure. In case the user wants to display in a new figure, the function call must be preceded by the call of ***figure*** function:

```
figure, imshow (I);
```

To close all the opened figures, user must call function the ***close all*** function:

```
close all
```

4. Conversion from RGB to Grayscale

Most common image processing operations require to reduce the quantity of information in image. The representation that preserve the relevant image information while reducing the complexity is a grayscale representation, having values between 0 and 255. A color image will be converted to grayscale by calling the ***rgb2gray*** function:

```
Ig = rgb2gray (I);
```

5. Thresholding

Thresholding process is used to convert a greyscale image into a black and white image (also referred as binary image), where the pixels intensities are reduced to only two values: 0 and 1. The thresholding process require the definition of a threshold *P* with value between 0 and 255. Then each pixels is compared with the threshold and the one that are greater will be assigned to 1, while the lowest ones will be assigned to 0.

```
P = 100;
IB = (Ig > P);
```

An easy way to define the threshold (but not always the optimal way) is to choose the P value to be equal to the mean intensity value of the image.

```
P = mean (Ig(:));
IB = (Ig>P);
```

6. Morphologic Operations

The basic morphological operation are erosion and dilation. Both operations have as parameters the original image and a structuring element. The structuring element indicate the neighbors that will be duplicated or eliminated after applying morphological operations.

Dilation has an effect of object „thickening”.

```
ID = imdilate (IB, S);
```

S is the structuring element. It is a binary matrix that can be defined by the user, or they can be generated by calling ***strel*** function and create structuring elements with predefined shapes such as: *disk*, *square*, *rectangle*, *diamond* etc.

```
S = strel ('disk', 5);
```

The call of this particular function will generate a circular structuring element with radius 5.

Erosion has the opposite effect of dilation, reducing the dimension of the objects.

```
IE = imerode (IB, S);
```

Based on the basic morphological operations, other derivate operations can be defined, as for example opening and closing. Opening operation is defined as a dilation followed by an erosion.

Its role is to eliminate small objects (or noise) remained as a result of thresholding operation. The closing, on the other hand, has the role of eliminating small holes in the objects. Both operations will keep the real dimensions of the objects.

```
IO = imopen (IB, S);
IC = imclose (IB, S);
```

7. Image labeling

Labeling process is performed on binary images and assigns a unique label to each individual object, by exploring the neighborhood relations between object pixels. Each pixel in the output image will have a value that corresponds to the object to which it belongs. MATLAB dispose of the predefined ***bwlabel*** function that performs the image labeling. An example of ***bwlabel*** function call is:

```
[L N] = bwlabel (IO, 8);
```

The first parameter corresponds to the binary image, and the second one to the neighborhood type (4 – for 4 neighborhood and 8 – for 8 neighborhood).

The output **L** is the image containing the labeled objects, along with the number of distinct objects **N**.

Display the labeling results:
Image **L** contains unique labels for each detected object. Though, the display of the result by just calling the ***imshow*** function is not relevant, since visually the labels are not properly differentiate. For a better visualization, MATLAB dispose of the ***label2rgb*** function that emphasize the color of each label:

```
IRGB = label2rgb (L, @jet, 'k', 'shuffle');
imshow (IRGB);
```

The first parameter is the labeled image **L**, the second one is a predefined color palette '**@jet'**, the third one is the background color '**k**' (- black), and the last one '**shuffle**' indicates the random choice of colors. The output of this call is a color image contains objects with different random colors.

8. Labeling applications

Having as input a labeled image, it can be generated a binary image having non-zero pixels only in the position corresponding to an object. For example, we can create the binary image corresponding to the object having the label 1:

```
O1 = (L==1);
```

In this image, we can extract the pixel coordinates contained in the current object using the function ***find***.

```
[row column] = find (O1);
```

The mass center of object one can be computed as the mean value of the row and column coordinates.

```
r_centermass = mean (row);
c_centermass = mean (column);
```

The min and max coordinates of the object can be found as follows:

```
r_minim = min (row);
r_maxim = max (row);
c_minim = min (column);
c_maxim = max (column);
```

The object area, or the number of object pixels is directly computed as:

```
mass = sum (O1(:));
```

9. Brightness and contrast adjustment

The MATLAB function ***imadjust*** automatically computes the optimal mean contrast and brightness for an image that is specified as a parameter.

```
IA = imadjust( Ig )
```

The image brightness can be changed by simply adding a constant value to all pixels in the image.

```
IBright = IG + 50;  // brighter
IDark = IG – 50;    // darker
```

10. Saving image on disk

To write an image on disk, Matlab use the function ***imwrite***:

```
imwrite (I, 'file_name', 'type');
```

The parameters of this function are:
I – the image to be saved;
file_name – the name of the image including the path on disk where the user wants to save the image and the image extension;
type – specifies the type of the image (e.g. 'bmp', 'jpg', etc.).

If I is the result of thresholding, it will contain only the values 0 and 1. In this case, the image should be multiplied with 255 before saving it, so the 1 value to correspond to white. Also, the type of output images provided by several processing is double, and sometimes it is required a conversion to byte before saving.

```
imwrite (255*uint8(IB), 'binary.bmp', 'bmp');
imwrite (uint8(L), 'labels.bmp', 'bmp');
```

11. Practical work

1. Read and display an image.
2. Read the rgb image ‘stars.jpg’. Convert it to grayscale and display the result. Save the resulted image on disk in a ‘.bmp’ image format.
3. Read the image ‘eight.bmp’ and convert it to binary image. Post process the output image with morphological operations in order to eliminate the imperfections.
4. Starting from the resulted image resulted at point 3., apply the labeling operation. Save the output image. Display the number of labeled objects, their area and mass center.
5. Read the images ‘hawkes_bay_nz.bmp’ and ‘wheel.bmp’. Apply some operations to adjust the brightness and the contrast in the image. Display and save the results.
6. Write MATLAB function that performs the tasks 1-5, having as parameter either the name of the image, or directly the matrix of pixels.

Appendix II. Image processing using the OpenCV library

1. Introduction

OpenCV (Open Source Computer Vision Library: http://opencv.org) is an open-source BSD-licensed library that includes several hundreds of computer vision algorithms of image processing at pixel level to complex methods such as camera calibration, object detection, optical flow, stereovision etc. It also includes tools for data storage and manipulation and graphical user interface functions. The OpenCV library package is freely available and can be downloaded (http://opencv.org/). The online documentation for the various editions of the library is available here: http://docs.opencv.org/ [1].

A framework based on the OpenCV library, called *OpenCVApplication* is available (you can download it from the personal web pages of the teaching staff). The framework is personalized for several versions of the Visual Studio (C++) development environments and several editions of the OpenCV library. The application includes some basic examples for opening and processing images and video streams. It can be used independently (without installing the OpenCV kit).

In the following, some basic data structures and image processing functions of the library are presented, as described in the 2.4.x API documentation (2.x API is essentially a C++ API).

In the OpenCV API, all the classes and functions are placed into the `cv` namespace. To access them from your code, use the `cv::` specifier or `using namespace cv;` directive:

```
...
cv::Mat src_gray, dst;
cv::equalizeHist(src_gray, dst);
...
```

or

```
using namespace cv;
...
Mat src_gray, dst;
equalizeHist(src_gray, dst);
...
```

Some of the current or future OpenCV external names may conflict with STL or other libraries. In this case, use explicit namespace specifiers to resolve the name conflicts:

2. Basic data structures in OpenCV

The Point_ class

`Point_` is a template class that specifies a 2D point by coordinates x and y. You can perform most of the unary and binary operations between `Point` type operators:

```
pt1 = pt2 + pt3;
pt1 = pt2 - pt3;
pt1 = pt2 * a;
pt1 = a * pt2;
pt1 += pt2;
pt1 -= pt2;
```

```
pt1 *= a;
double value = norm(pt); // L2 norm
pt1 == pt2;
pt1 != pt2;
```

You can use specific types for the coordinates and there is a cast operator to convert point coordinates to the specified type. The following type aliases are defined:

```
typedef Point_<int> Point2i;
typedef Point2i Point;
typedef Point_<float> Point2f;
typedef Point_<double> Point2d;
```

The Point3_ class

`Point3_` is a template class that specifies a 3D point by coordinates x, y and z. It supports all the vector arithmetic and comparison operations (same as for the `Point_` class). You can use specific types for the coordinates and there is a cast operator to convert point coordinates to the specified type. The following type aliases are defined:

```
typedef Point3_<int> Point3i;
typedef Point3_<float> Point3f;
typedef Point3_<double> Point3d;
```

The Size_ class

`Size_` is a template class that specifies the size of an image or rectangle. The class includes two members called `width` and `height`. The same arithmetic and comparison operations as for `Point_` class are available.

The Rect_ class

`Rect_` is the template class for 2D rectangles, described by the following parameters:

- Coordinates of the top-left corner: `Rect_::x` and `Rect_::y`
- Rectangle width (`Rect_::width`) and height (`Rect_::height`).

The following type allias is defined:

```
typedef Rect_<int> Rect;
```

The following operations on rectangles are implemented:

- shift: `rect = rect ± point`
- expand/shrink: `rect = rect ± size`
- augmenting operations: `rect += pont, rect -= pont, rect += size, rect -= size`
- intersection: `rect = rect1 & rect2, rect &= rect1`
- minimum area rectangle containing 2 rectangles: `rect = rect1 | rect2, rect |= rect1`
- rectangle comparission: `rect == rect1, rect != rect1`

The Vec class

The `Vec` class is commonly used to describe pixel types of multi-channel arrays. For example to describe a RGB 24 image pixel the following type can be used:

```
typedef Vec<uchar, 3> Vec3b;
```

The following vector operations are implemented:

```
v1 = v2 + v3
v1 = v2 - v3
v1 = v2 * scale
v1 = scale * v2
v1 = -v2
```
`v1 += v2` and other augmenting operations
```
v1 == v2, v1 != v2
```
`norm(v1)` (Euclidean norm)

The Scalar_ class

`Scalar_` is a template class for a 4-element vector derived from Vec.

The Mat class

The **`Mat`** class represents an n-dimensional single-channel or multi-channel array. It can be used to store real or complex vectors and matrices, grayscale or color images, histograms etc. 2-dimensional matrices are stored row-by-row, 3-dimensional matrices are stored plane-by-plane, and so on.

```
class CV_EXPORTS Mat
{
public:
    // ... a lot of methods ...
    ...
    //! the array dimensionality, >= 2
    int dims;
    //! the number of rows and columns or
    // (-1, -1) when the array has more than 2 dimensions
    int rows, cols;
    //! pointer to the data
    uchar* data;
    // other members
    ...
};
```

The most common ways to create a Mat object are
`create(nrows, ncols, type)` or
using one of the constructors: `Mat(nrows, ncols, type[, fillValue])`

Examples:
```
Mat m1,m2;
// create a 100x100 3 channel byte matrix
m1.create(100,100,CV_8UC(3));
// create a 5x5 complex matrix filled with (1-2j)
Mat m2(5,5,CV_32FC2,Scalar(1,-2));
```

```
// create a 640x480 1 channel byte matrix filled with 0
Mat src(480,640,CV_8UC1, 0);
```

Accessing the elements of a matrix can be done in several ways. Supposing that `src` is a gray-scale image with 480 rows and 640 columns (initialized by opening an image from the disk), the examples below are presenting how a simple processing like the image negative can be implemented:

```
int height = src.rows;
int width = src.cols;
Mat dst = src.(clone);

// the "easy/slow" approach
for (int i=0; i<height; i++)
{
	for (int j=0; j<width; j++)
	{
		uchar val = src.at<uchar>(i,j);
		uchar neg = 255-val;
		dst.at<uchar>(i,j) = neg;
	}
}
```

or

```
// the fast approach
for (int i = 0; i < height; i++)
{
	// get the pointer to row i
	const uchar* SrcRowi = src.ptr<uchar>(i);
	uchar* DstRowi = dst.ptr<uchar>(i);
	//iterate through each row
	for (int j = 0; j < width; j++)
	{
		uchar val = SrcRowi[j];
		uchar neg = 255 - val;
		DstRowi[j] = neg;
	}
}
```

or

```
// the fastest approach using the "diblook style"
uchar *lpSrc = src.data;
uchar *lpDst = dst.data;
int w = src.step; // no DWORD alignment is done !!!
for (int i = 0; i<height; i++)
	for (int j = 0; j < width; j++)
	{
		uchar val = lpSrc[i*w + j];
		lpDst[i*w + j] = 255 - val;
	}
```

The operations implemented on matrices that can be combined in arbitrary complex expressions. Few examples are presented bellow (see the documentation for more examples). In the expressions bellow `A`, `B` stand for matrices (`Mat`), `s` for a scalar (`Scalar`), `alpha` for a scalar (`double`)).

- Addition, subtraction, negation: `A+B, A-B, A+s, A-s, s+A, s-A, -A`
- Scaling: `A*alpha`

- Matrix multiplication: `A*B`
- Transposition: `A.t()` (means A^T)
- Matrix inversion and pseudo-inversion, solving linear systems and least-squares problems: `A.inv([method])` (~ A^{-1}) , `A.inv([method])*B` (~ `X: AX=B`)
- Comparison: `A cmpop B, A cmpop alpha, alpha cmpop A`, where `cmpop` is one of: `>, >=, ==, !=, <=, <`. The result of comparison is an 8-bit single channel mask whose elements are set to 255 (if the particular element or pair of elements satisfy the condition) or 0.
- Bitwise logical operations: `A logicop B, A logicop s, s logicop A, ~A`, where `logicop` is one of : `&, |, ^`.
- Element-wise minimum and maximum: `min(A, B), min(A, alpha), max(A, B), max(A, alpha)`
- Element-wise absolute value: `abs(A)`
- Cross-product, dot-product: `A.cross(B)` `A.dot(B)`
- Matrix initializers: `Mat::eye(), Mat::zeros(), Mat::ones()`

In order to get a region of interest (ROI) from a matrix defined by a Rect Structure use the following statements:

```
Mat image;
Rect ROI_rect;
Mat roi=image(ROI_rect);
```

3. Reading, writing and displaying images and videos

To open an image file stored on the disk the `imread` function can be used. It can handle/decode the most common image formats (bmp, jpg, gif, png etc.):

```
Mat imread(const string& filename, int flags=1 )
```

The input parameters are the `filename` and an optional `flags` parameter specifying the color type of a loaded image:

- CV_LOAD_IMAGE_ANYDEPTH - returns a 16-bit/32-bit image when the input has the corresponding depth, otherwise converts it to 8-bit.
- CV_LOAD_IMAGE_COLOR - always convert the image to a color one.
- CV_LOAD_IMAGE_GRAYSCALE - always converts the image to a grayscale one
- **>0** - returns a 3-channel color image.
- **=0** - returns a grayscale image.
- **<0** - returns the loaded image as is (with alpha channel).

The output is a `Mat` object containing the image for successful completion of the operation. Otherwise the function returns an empty matrix (`Mat::data==NULL`).

To display an image in a specified window, the `imshow` function can be used:

```
void imshow(const string& winname, InputArray mat)
```

In order to control the size and position of the display window the following functions can be used:

```
void namedWindow(const string& winname, int flags=WINDOW_AUTOSIZE )
void moveWindow(const string& winname, int x, int y)
```

To save the image on to the disk the `imwrite` function can be used. The image format is chosen based on the `filename` extension.

```
bool imwrite(const string& filename, InputArray img,
                            const vector<int>& params=vector<int>())
```

The input parameter `params` contains format-specific save parameters encoded as pairs `paramId_1, paramValue_1, paramId_2, paramValue_2, ...` . The following parameters are currently supported:

- For JPEG, it can be a quality (`CV_IMWRITE_JPEG_QUALITY`) from 0 to 100 (the higher is the better). Default value is 95.
- For PNG, it can be the compression level (`CV_IMWRITE_PNG_COMPRESSION`) from 0 to 9. A higher value means a smaller size and longer compression time. Default value is 3.
- For PPM, PGM, or PBM, it can be a binary format flag (`CV_IMWRITE_PXM_BINARY`), 0 or 1. Default value is 1.

The following example illustrates the above mentioned image handling functions by opening a color image, converting it to grayscale and saving it to the disk and displaying the source image and the destination/result image in a separate windows:

```
void testImageOpenAndSave()
{
	Mat src, dst;
	src = imread("Images/Lena_24bits.bmp", CV_LOAD_IMAGE_COLOR); //Read the image

	if (!src.data) //Check for invalid input
	{
		printf("Could not open or find the image\n");
		return;
	}

	//Get the image resolution
	Size src_size = Size(src.cols, src.rows);

	//Display window
	const char* WIN_SRC = "Src"; //window for the source image
	namedWindow(WIN_SRC, CV_WINDOW_AUTOSIZE);
	cvMoveWindow(WIN_SRC, 0, 0);

	const char* WIN_DST = "Dst"; //window for the destination (processed) image
	namedWindow(WIN_DST, CV_WINDOW_AUTOSIZE);
```

```
	cvMoveWindow(WIN_DST, src_size.width + 10, 0);

	cvtColor(src, dst, CV_BGR2GRAY); //converts the source image to grayscale
	imwrite("Images/Lena_24bits_gray.bmp", dst); //writes the destination to file

	imshow(WIN_SRC, src);
	imshow(WIN_DST, dst);

	printf("Press any key to continue ...\n");
	waitKey(0);
}
```

The `VideoCapture` class provides the C++ API for capturing video from cameras or for reading video files. In the following example is shown how you can use it (i.e. performing canny edge detection on every frame and displaying the result in a destination window; the example also shows how you can compute the processing time).

```
void testVideoSequence()
{
	VideoCapture cap("Videos/rubic.avi"); // open a video file from disk
	//VideoCapture cap(0); // open the default camera (i.e. the built in web cam)
	if (!cap.isOpened()) // opening the video device failed
	{
		printf("Cannot open video capture device\n");
		return;
	}

	Mat frame, grayFrame, dst;

	// video resolution
	Size capS = Size((int)cap.get(CV_CAP_PROP_FRAME_WIDTH),
		(int)cap.get(CV_CAP_PROP_FRAME_HEIGHT));

	// Init. display windows
	const char* WIN_SRC = "Src"; //window for the source frame
	namedWindow(WIN_SRC, CV_WINDOW_AUTOSIZE);
	cvMoveWindow(WIN_SRC, 0, 0);

	const char* WIN_DST = "Dst"; //window for the destination (processed) frame
	namedWindow(WIN_DST, CV_WINDOW_AUTOSIZE);
	cvMoveWindow(WIN_DST, capS.width + 10, 0);

	char c;
	int frameNum = -1;

	for (;;)
	{
		cap >> frame; // get a new frame from camera
		if (frame.empty())
		{
			printf("End of the video file\n");
			break;
		}

		++frameNum;

		double t = (double)getTickCount(); // Get the current time [s]

		// Insert your processing here ....
		cvtColor(frame, grayFrame, CV_BGR2GRAY);
		// Performs canny edge detection on the current frame
		Canny(grayFrame, dst, 40, 100, 3);
		// ....
		// End of processing

		// Get the current time again and compute the time difference [s]
```

```
            t = ((double)getTickCount() - t) / getTickFrequency();

            // Print (in the console window) the processing time in [ms]
            printf("Time = %.3f [ms]\n", t * 1000);

            // output written in the WIN_SRC window (upper left corner)
            char msg[100];
            sprintf(msg, "%.2f[ms]", t * 1000);
            putText(frame, msg, Point(5, 20), FONT_HERSHEY_SIMPLEX,
                                        0.5, CV_RGB(255, 0, 0), 1, 8);
            imshow(WIN_SRC, frame);
            imshow(WIN_DST, dst);

            c = cvWaitKey(0);  // waits a key press to advance to the next frame
            if (c == 27) {
                  // press ESC to exit
                  printf("ESC pressed - capture finished");
                  break;  //ESC pressed
            }
      }
}
```

4. Basic operations applied on images

The operations are generic for array type objects (i.e. `Mat`) and if the array is initialized with an image, the result is a pixel level operation. In the case of multi-channel arrays, each channel is processed independently. Some functions allow the specification of an optional mask used to select a sub-array.

- `void` **`add`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`, InputArray` **`mask`**`=noArray(), int` **`dtype`**`=-1)` - Calculates the per-element sum of two arrays or an array and a scalar: *src1* is added to *src2* and the result is stored in dst. The function can be replaced with matrix expressions: `dst = src1 + src2;`

- `void` **`addWeighted`**`(InputArray` **`src1`**`, double` **`alpha`**`, InputArray` **`src2`**`, double` **`beta`**`, double` **`gamma`**`, OutputArray` **`dst`**`, int` **`dtype`**`=-1)` – performs the weighted sum of two arrays, and is equivalent with the following matrix expression: `dst = src1*alpha + src2*beta + gamma;`

- `void` **`absdiff`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`)` - performs the per-element absolute difference between two arrays or between an array and a scalar.

- `void` **`bitwise_and`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`, InputArray` **`mask`**`=noArray()` – computes the per-element bit-wise conjunction of two arrays (*src1* and *src2*) or an array and a scalar.

- `oid` **`divide`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`, double` **`scale`**`=1, int` **`dtype`**`=-1)` – performs the division operation between the elements of two arrays; the result is multiplied with the scaling parameter.

- `Scalar` **`mean`**`(InputArray` **`src`**`, InputArray` **`mask`**`=noArray())` – calculates the mean value of the array elements, independently for each channel of the array.

- `void` **`max`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`))` – computes the per-element maximum of two arrays: *src1* and *src2*.

`void` **`min`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`)` – computes the per-element minimum of two arrays: *src1* and *src2*.

- `void` **`minMaxLoc`**`(InputArray` **`src`**`, double*` **`minVal`**`, double*` **`maxVal`**`=0, Point*` **`minLoc`**`=0, Point*` **`maxLoc`**`=0, InputArray` **`mask`**`=noArray())` – returns the minimum and maximum value in the *src* array and also their coordinates (the function does not work with multi-channel arrays.).

- `void` **`multiply`**`(InputArray` **`src1`**`, InputArray` **`src2`**`, OutputArray` **`dst`**`, double` **`scale`**`=1, int` **`dtype`**`=-1 )` - performs the per-element scaled product of two arrays.

5. Morphological operations

In OpenCV the morphological operations work on both binary and grayscale images. Each morphological operation requires a structuring element. This can be created using `getStructuringElement` function:

```
Mat getStructuringElement(int shape, Size ksize, Point anchor=Point(-1,-1)
```

his function creates a structuring element of given shape and dimension. The *shape* parameter controls its shape and can take the following constant values:

- `CV_SHAPE_RECT`
- `CV_SHAPE_CROSS`
- `CV_SHAPE_ELLIPSE`

The *anchor* parameter specifies the anchor position within the element. The default value (-1, -1) means that the anchor is at the center. Note that only the shape of a cross-shaped element depends on the anchor position. In other cases the anchor just regulates how much the result of the morphological operation is shifted.

Morphological dilation can be performed using the *dilate* function:

```
void dilate (InputArray src, OutputArray dst, InputArray kernel, Point
anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const
Scalar& borderValue=morphologyDefaultBorderValue() )
```

Morphological erosion can be performed using the *erode* function:

```
void erode (InputArray src, OutputArray dst, InputArray kernel, Point
anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const
Scalar& borderValue=morphologyDefaultBorderValue() )
```

An example that performs a 2 iterations erosion followed by a 2 iterations dilation using a 3x3 cross-shape structuring element is presented below:

```
//structuring element for morpho operations
Mat element = getStructuringElement(MORPH_CROSS, Size(3, 3));
erode(src, temp, element, Point(-1, -1), 2);
```

```
dilate(temp, dst, element, Point(-1, -1), 2);
```

unction `morphologyEx` can be used to perform more complex morphological operations:

```
void morphologyEx(InputArray src, OutputArray dst, int op, InputArray kernel, Point
anchor=Point(-1,-1), int iterations=1, int borderType=BORDER_CONSTANT, const Scalar&
borderValue=morphologyDefaultBorderValue() )
```

The `op` parameter specifies the type of the operation which is performed:

- **MORPH_OPEN** - an opening operation
- **MORPH_CLOSE** - a closing operation
- **MORPH_GRADIENT** - a morphological gradient
- **MORPH_TOPHAT** - "top hat"
- **MORPH_BLACKHAT** - "black hat"

6. Thresholding

The thresholding operation in OpenCV are performed with *threshold* function:

```
double threshold(InputArray src, OutputArray dst, double thresh,
                 double maxval, int type)
```

The operation is applied on the *src* image and the resulted binary image is stored in *dst* array. The threshold value is specified by *thresh* parameter and the thresholding method is specified by *type* parameter. The *type* parameter can take one of the following constant values:

- THRESH_BINARY $dst(x,y) = \begin{cases} maxval, if\ src(x,y) > TH \\ 0, otherwise \end{cases}$
- THRESH_BINARY_INV $dst(x,y) = \begin{cases} 0, if\ src(x,y) > TH \\ maxval, otherwise \end{cases}$
- THRESH_TRUNC $dst(x,y) = \begin{cases} TH, if\ src(x,y) > TH \\ src(x,y), otherwise \end{cases}$
- THRESH_TOZERO $dst(x,y) = \begin{cases} src(x,y), if\ src(x,y) > TH \\ 0, otherwise \end{cases}$
- THRESH_TOZERO_INV $dst(x,y) = \begin{cases} 0, if\ src(x,y) > TH \\ src(x,y), otherwise \end{cases}$

The above values can be combined with `THRESH_OTSU`. In this case, the function computes the optimal threshold value using the Otsu's algorithm (the implementation works only for 8-bit images) which is used instead of the specified *thresh* parameter. The function returns the computed optimal threshold value.

7. Filters

In OpenCV there are some optimized function designed to perform the filtering operations. These optimizations depend however on the hardware and software architecture of the system.

Some example of functions that perform filtering operations are as follows:

`void medianBlur(InputArray` **`src`**`, OutputArray` **`dst`**`, int` **`ksize`**`)` – implements the median filter

`void blur(InputArray` **`src`**`, OutputArray` **`dst`**`, Size` **`ksize`**`, Point` **`anchor`**`=Point(-1,-1), int` **`borderType`**`=BORDER_DEFAULT )` – implements the normalized box filter (mean filter).

`void GaussianBlur (InputArray` **`src`**`, OutputArray` **`dst`**`, Size` **`ksize`**`, double` **`sigmaX`**`, double` **`sigmaY`**`=0, int` **`borderType`**`=BORDER_DEFAULT )` - implements the gaussian filter.

`oid Laplacian(InputArray` **`src`**`, OutputArray` **`dst`**`, int` **`ddepth`**`, int` **`ksize`**`=1, double` **`scale`**`=1, double` **`delta`**`=0, int` **`borderType`**`=BORDER_DEFAULT ) - p`erforms the filtering with a Laplacian filter using the specified aperture size if `ksize >1`. If `ksize ==1` the 3x3 Laplacian filter have the following elements is applied:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

`void Sobel(InputArray` **`src`**`, OutputArray` **`dst`**`, int` **`ddepth`**`, int` **`dx`**`, int` **`dy`**`, int` **`ksize`**`=3, double` **`scale`**`=1, double` **`delta`**`=0, int` **`borderType`**`=BORDER_DEFAULT )-` computes the first, second, third, or mixed image derivatives (`dx` and `dy` parameters) using an extended Sobel operator. Most often, the function is called with (`xorder` = 1, `yorder` = 0, `ksize` = 3) or (`xorder` = 0, `yorder` = 1, `ksize` = 3) to calculate the first x- or y- image derivative F (see chapter 12 for more details).

`void filter2D(InputArray` **`src`**`, OutputArray` **`dst`**`, int` **`ddepth`**`, InputArray` **`kernel`**`, Point` **`anchor`**`=Point(-1,-1), double` **`delta`**`=0, int` **`borderType`**`=BORDER_DEFAULT )-` this is a generalized function designed to apply the filtering operation with a custom convolution kernel. The kernel elements are defined in *kernel* parameter as a matrix. Default value (-1,-1) for the *anchor* parameter means that the anchor is at the kernel center.

8. References

[1] OpenCV on-line documentation, http://docs.opencv.org/ , cited Dec. 2015.