2. The color model. Color ⇒ grayscale and grayscale ⇒ 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).

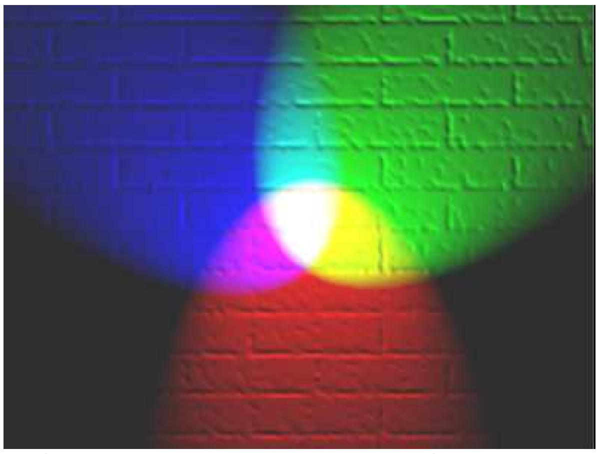


Fig. 2.1. A representation of additive color mixing. Projection of primary color lights on a screen shows secondary colors where two overlap; the combination of all three of red, green, and blue in appropriate intensities makes white [1].

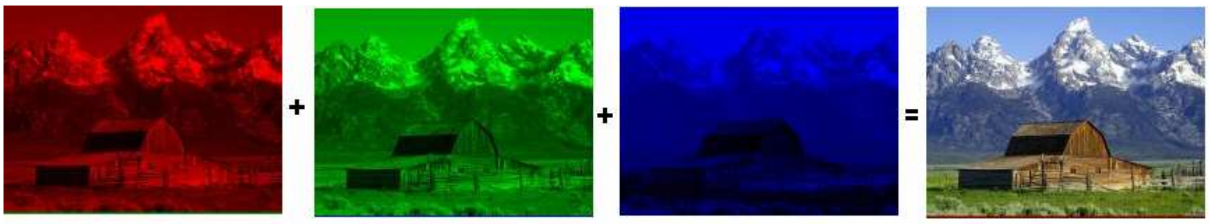


Fig. 2.2. The color of an image is obtained by combining the tree elementary colors for each pixel (three elementary color images).

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).

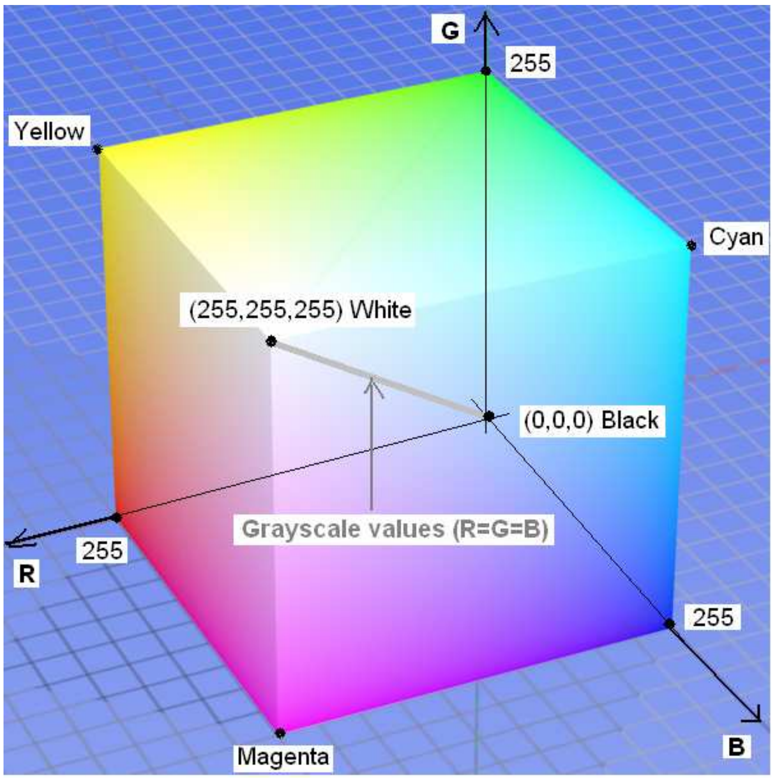


Fig. 2.3. The RGB color model mapped to a cube. In this example (RGB24 bitmap image) each color is represented on 8 bits (256 colors). The total number of colors is 2^8x2^8x2^8 = 2^{34} = 16.777.216.

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 figure 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] = paleta[k].rgbRed;
      // „sort” the LUT
      paleta[k].rgbRed = paleta[k].rgbGreen = paleta[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: $k \to g(k)$ (fig. 2.5):

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

2.4. Accessing the LUT's contents and other relevant information from the 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!

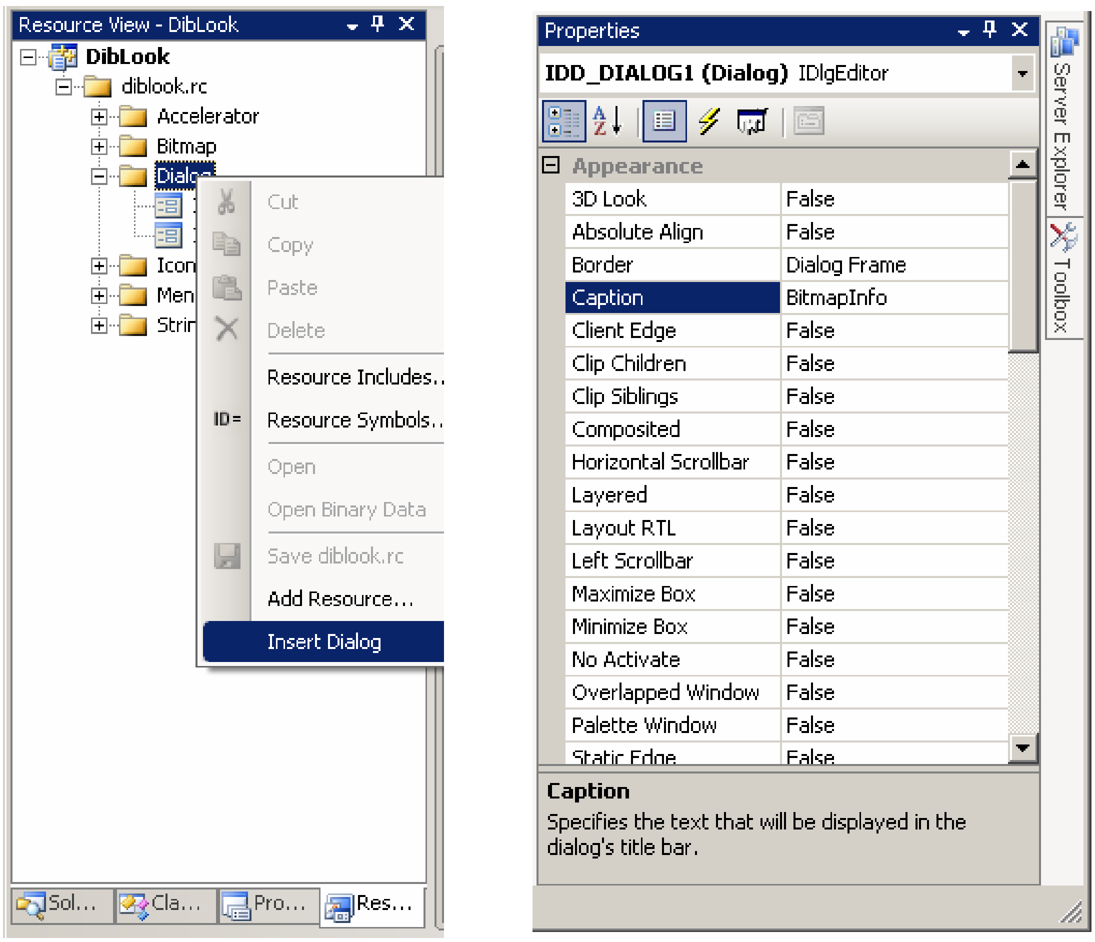
2.5. Guide to display information in a dialog box

2.5.1. Creating a new *Dialog Box* resource

1. Switch in the *Workspace* Window on the *ResourceView* 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).



IDD_DIALOG1 (Dialog) IDlgEditor

Appearance

3D Look	False
Absolute Align	False
Border	Dialog Frame
Caption	BitmapInfo
Client Edge	False
Clip Children	False
Clip Siblings	False
Composited	False
Horizontal Scrollbar	False
Layered	False
Layout RTL	False
Left Scrollbar	False
Maximize Box	False
Minimize Box	False
No Activate	False
Overlapped Window	False
Palette Window	False
Static Edge	False

Caption
Specifies the text that will be displayed in the dialog's title bar.

a. b.

Fig. 2.6.

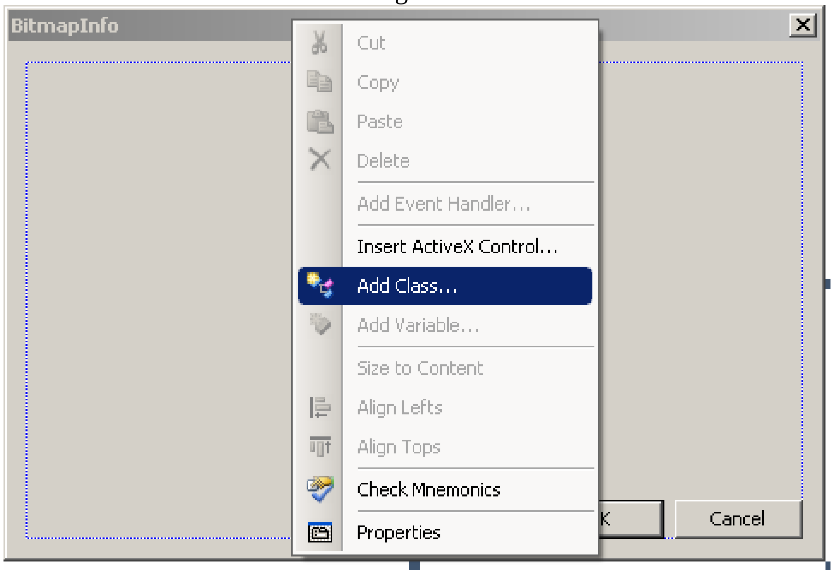
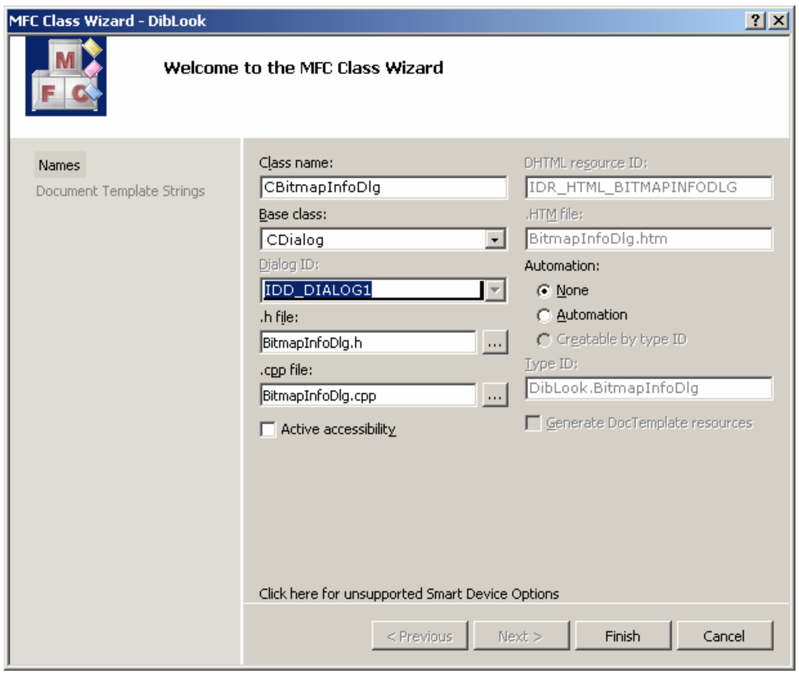


Fig. 2.7.

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).



MFC Class Wizard - DibLook

Welcome to the MFC Class Wizard

Names
Document Template Strings

Class name: CBitmapInfoDlg
Base class: CDialog
Dialog ID: IDD_DIALOG1
.h file: BitmapInfoDlg.h
.cpp file: BitmapInfoDlg.cpp
[ ] Active accessibility

DHTML resource ID: IDR_HTML_BITMAPINFODLG
.HTM file: BitmapInfoDlg.htm
Automation: [✓ None] Automation Creatable by type ID
Type ID: DibLook.BitmapInfoDlg
[ ] Generate DocTemplate resources

Click here for unsupported Smart Device Options

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;
}
```

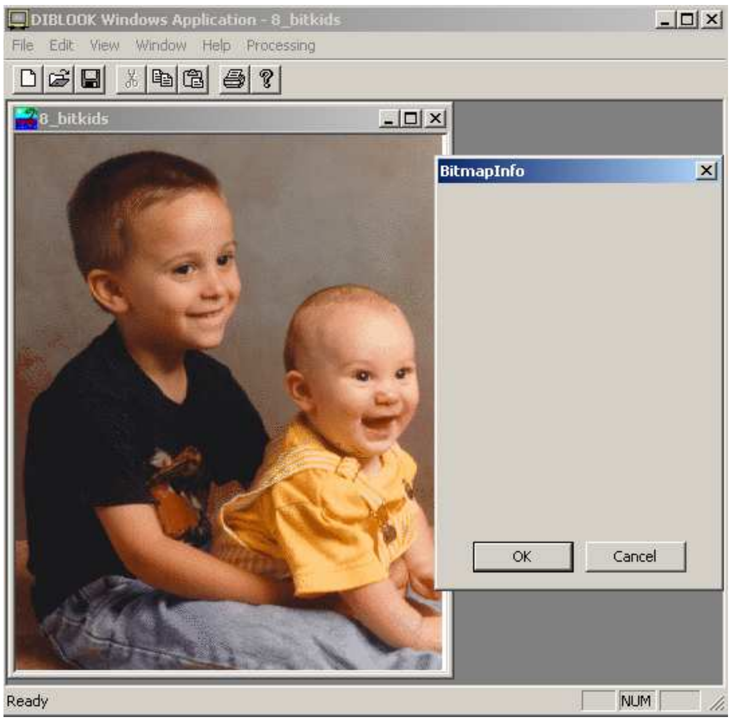

DIBLOOK Windows Application - 8_bitkids
File Edit View Window Help Processing
8_bitkids
BitmapInfo
OK
Cancel
Ready
NUM

Fig. 2.9.

2.5.2. Designing the dialog box

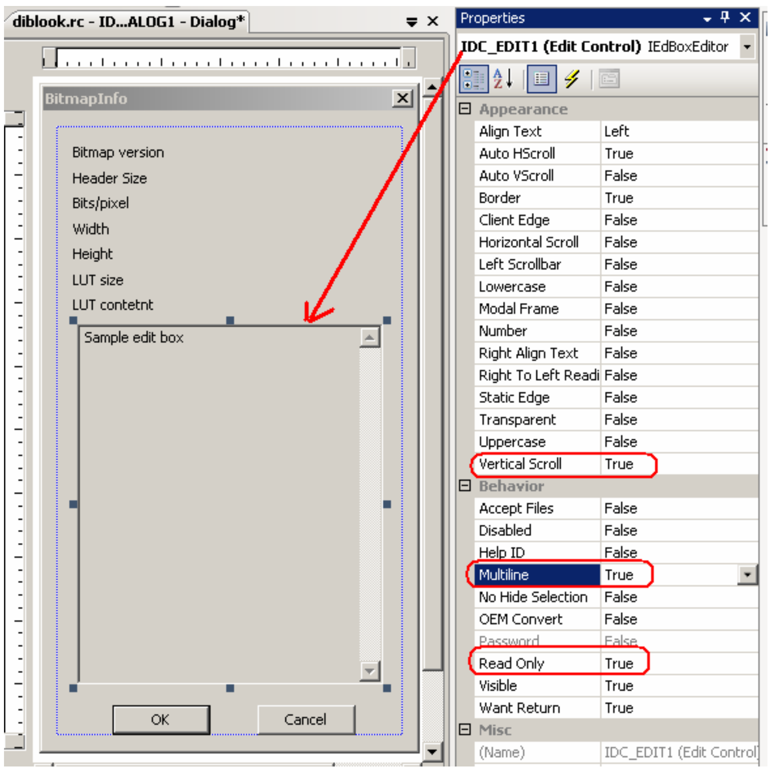
In order to show or to get some data from the dialog box, *Controls* should be added to the already created dialog resource. The most common controls are the *Static Text* (for output) and the *Edit Box* (for output/input). In the current case only output is required. In order to add a control to the dialog, select the *Toolbox* window 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 Box*. Edit box controls have allocated an individual ID by default (there is no need to change it). For the *Edit Box* controls 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 *CSting*), category (select *Value*)) etc.). The member variables associated to the controls using the *Add Member Variable Wizzard* are added automatically to the dialog class (fig. 2.10.d).

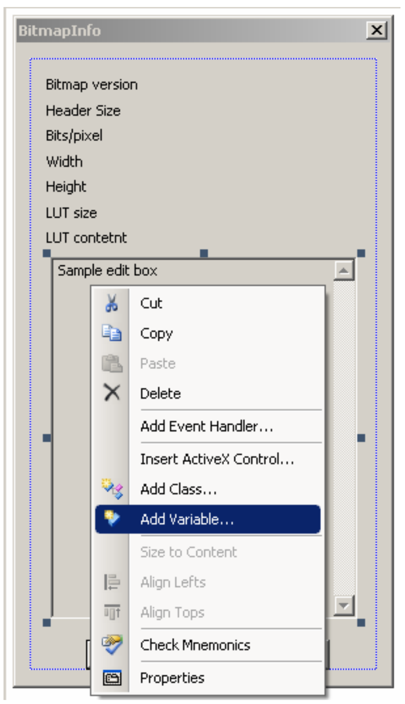


Properties

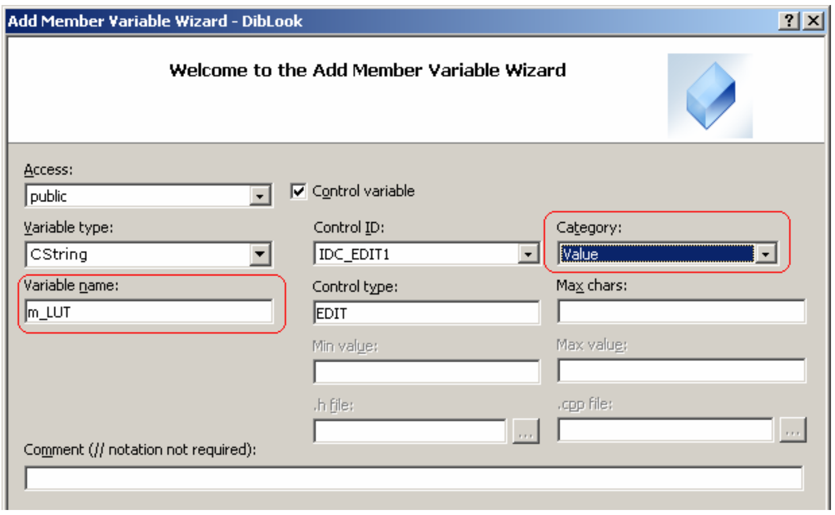
IDC_EDIT1 (Edit Control) IEdBoxEditor

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.



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:

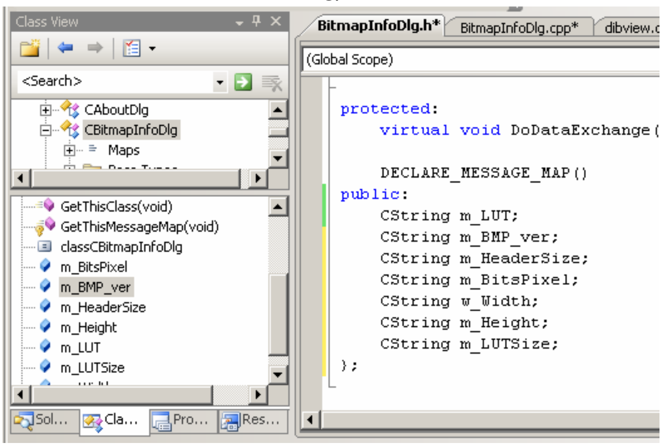
Max value:

.h file:

.cpp file:

Comment (// notation not required):

c.



```
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("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 box for displaying the LUT)
      CString buffer;
      for (int i=0;i<iColors;i++)
      {
            buffer.Format("%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).

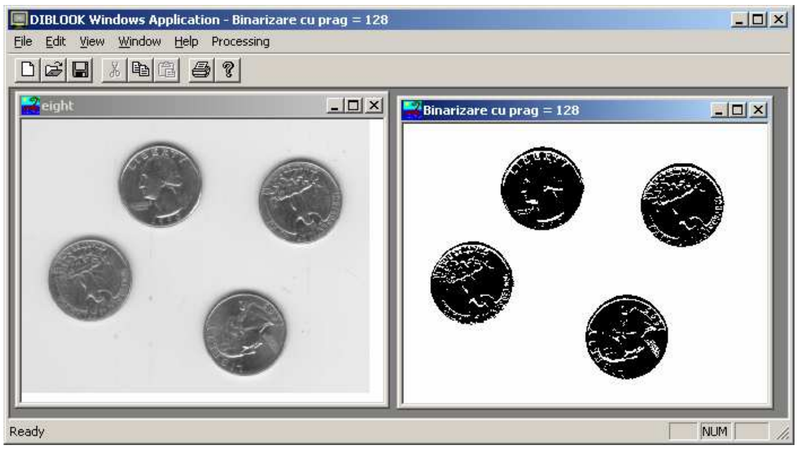
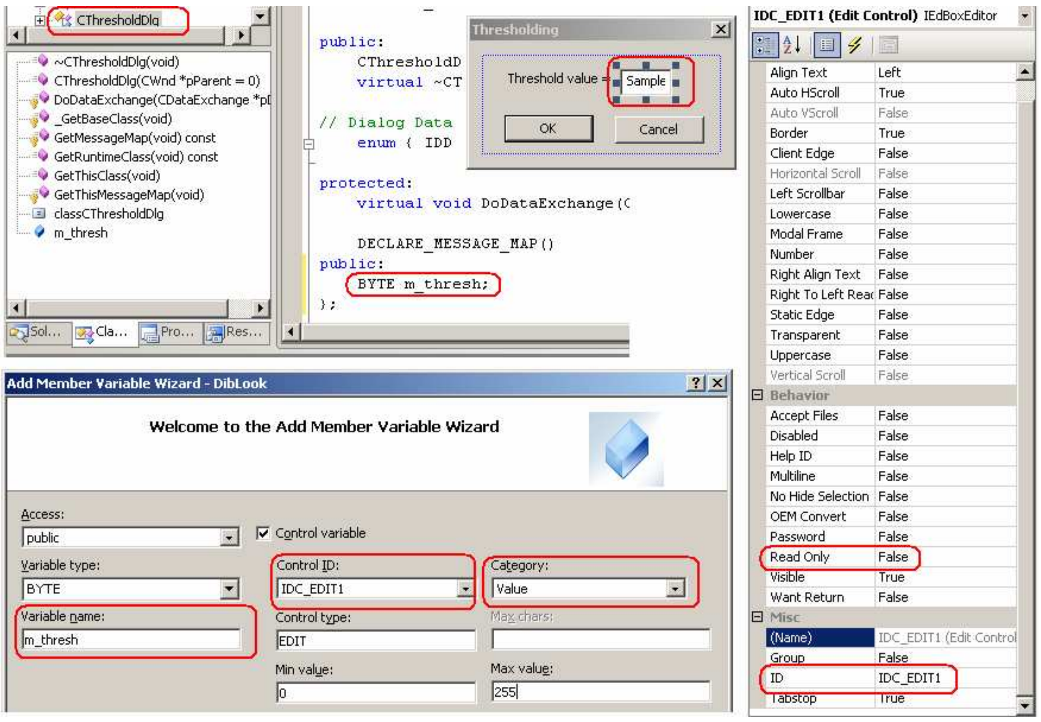

DIBLOOK Windows Application - Binarizare cu prag = 128
File Edit View Window Help Processing
eight
Binarizare cu prag = 128
Ready
NUM

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:

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

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 box control is created and used to get a value is similar as presented in section 2.5. The edit box should allow editing its content (not to be read-only (default), as shown in figure 2.12. The type of the variable used to get/store the value typed in the edit box can be a numerical one (BYTE – fig. 1.12).



```
CThresholdDlg
    ~CThresholdDlg(void)
    CThresholdDlg(CWnd *pParent = 0)
    DoDataExchange(CDataExchange *pI
    _GetBaseClass(void)
    GetMessageMap(void) const
    GetRuntimeClass(void) const
    GetThisClass(void)
    GetThisMessageMap(void)
    classCThresholdDlg
    m_thresh
```

```
public:
    CThresholdD
    virtual ~CT

// Dialog Data
    enum { IDD

protected:
    virtual void DoDataExchange(C

    DECLARE_MESSAGE_MAP()
public:
    BYTE m_thresh;
};
```

Thresholding — Threshold value: Sample — OK — Cancel

Add Member Variable Wizard - DibLook

Welcome to the Add Member Variable Wizard

- Access: public
- ✓ Control variable
- Variable type: BYTE
- Control ID: IDC_EDIT1
- Category: Value
- Variable name: m_thresh
- Control type: EDIT
- Max chars:
- Min value: 0
- Max value: 255

IDC_EDIT1 (Edit Control) IEdBoxEditor

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 Reac	False
Static Edge	False
Transparent	False
Uppercase	False
Vertical Scroll	False
Behavior	
Accept Files	False
Disabled	False
Help ID	False
Multiline	False
No Hide Selection	False
OEM Convert	False
Password	False
Read Only	False
Visible	True
Want Return	False
Misc	
(Name)	IDC_EDIT1 (Edit Control
Group	False
ID	IDC_EDIT1
Tabstop	True

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("Threshold = %d", threshold);
            END_PROCESSING (buf);
      }
}
```

2.7. Practical work

1. 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.
2. Add to the DIBLook framework a processing function for the *color ⇒ grayscale* conversion of a RGB24 images (24 bits/pixel), using (2.1).
3. Add to the DIBLook framework a processing function for the *color ⇒ grayscale* conversion of an indexed images (8 bits/pixel), using (2.1).
4. Add to the DIBLook framework a processing function which sorts the LUT of an indexed image, as described in section 2.3.2.
5. Compare the content of an unsorted LUT with a sorted one (using the grayscale image obtained from *Kids.bmp* color image).
6. Integrate functions from points 3 and 4 in a single one.
7. 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.
8. **Save your work. Use the same application in the next laboratories. At the end of the image processing laboratory you should present your own application with the implemented algorithms!!!**

References

[1] http://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