1. Introduction to the OpenCV library

1.1. Introduction

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

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. The bitmap image format

The bmp format is used to store images in uncompressed form. It uses raster graphics to store digital images independently of the display device. It is capable of storing monochrome and color images with different encoding depth. The depth determines the number of possible colors and determines the image size. The file itself has the following structure:

- a bitmap file header - which contains a signature field, the file size and the offset to the pixel array;
- DIB header - which stores various information such as image dimensions, bits per pixel;
- color table (or look-up table) - for images with a color palette;
- the pixel array - contains the actual image information stored in a linearized manner and padded.

The following image illustrates the bitmap format for a 24bit color image. The image height and width are denoted dwHeight and dwWidth respectively.

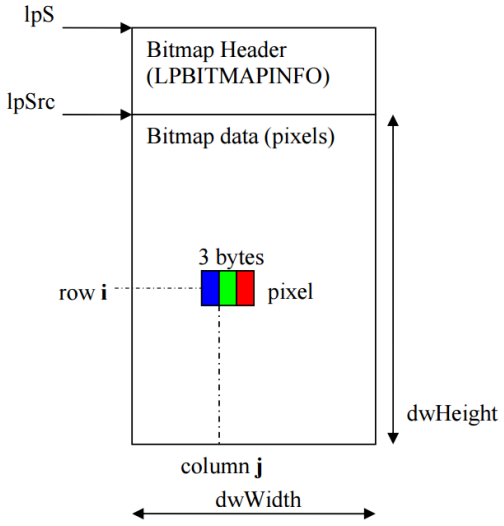

Figure 1. Bitmap image format

1.3. Overview of the OpenCV framework

The framework on which you will be working on contains the OpenCV libraray 2.4.13 bundled together with a Visual Studio 2013 solution. Include settings have been preconfigured, all static and dynamic libraries are included with the solution.

Your task is to create new functions and call them from the main function. You should group your work according to laboratory sessions and give suggestive names to functions. All code examples assume that you have included the cv namespace (**`using namespace cv`**), otherwise prepend **`cv::`** to all OpenCV classes and methods. A guideline for introducing new functions is given in the following code snippet (gray text indicates what you need to introduce):

```
void negative_image(){
   //implement function
}
int main(){
   int op;
   do{
        printf("Menu:\n");
        //...
        printf(" 7 - L1 Negative Image \n");
        //...
        printf(" 0 - Exit\n\n");
        printf("Option: ");
        scanf("%d",&op);
        switch (op)
        {
            //...
            case 7:
                negative_image();
                break;
        }
   }
   while (op!=0);
   return 0;
}
```

You should save your work from each session. The project can be cleaned with the clean.bat executable which deletes all build outputs and reduces the project size considerably. *Alternatively, to save space, just backup the main cpp file since the project solutions should not change.*

1.4. The Mat class

Images are stored as Mat objects in OpenCV. It is class for a generic matrix that can be used to hold other data as well, such as a normal 2x2 matrix or higher dimensional matrices.

Important fields of the Mat class are:

- rows - the number of rows of the matrix = the height of the image;
- cols - the number of columns of the matrix = the width of the image;

- data - pointer to the memory location of the actual image; it is of type **`unsigned char *`**, so it must be cast to the correct type for accessing operations

The simplest and cleanest way to create a Mat object called img is to use the 3 parameter constructor:

```
Mat img(rows, cols, type);
```

The last parameter encodes the type of data that is stored in the matrix. An example type would be CV_8UC1, which it represents: 8 bit, unsigned char, single channel. In general the first number after CV_ represents the number of bits required; the letter indicates the data type; and Cx shows the number of channels.

type code	data type	used for
CV_8UC1	unsigned char	grayscale image (8bits/pixel)
CV_8UC3	Vec3b	color image (3x8bits/pixel)
CV_16SC1	short	data storage
CV_32FC1	float	data storage
CV_64FC1	double	data storage

Table 1. Common OpenCV data type codes

Example 1 - create a grayscale matrix of size 256x256:

```
Mat img(256,256,CV_8UC1);
```

Example 2 - create a color image of dimension with 720 rows and 1280 columns:

```
Mat img(720,1280,CV_8UC3);
```

Example 3 - create a 2x2 real matrix with values [1 2; 3 4], and print it:

```
float vals[4] = {1, 2, 3, 4};
Mat M(2,2,CV_32FC1,vals); //4 parameter constructor
std::cout << M << std::endl;
```

Notice, you can use the standard output stream with a Mat object.

For a detailed decription of the Mat class, see the official documentation at:
http://docs.opencv.org/2.4.13/modules/core/doc/basic_structures.html#mat

1.5. Opening/reading an image

To open an image and to store it as a Mat object use the **`imread`** function:

```
Mat img = imread("path_to_image", flag);
```

The first parameter contains the relative or absolute path to the image file; the second flag parameter can be:

- CV_LOAD_IMAGE_UNCHANGED (-1) - load the image in the same format as it was saved;
- CV_LOAD_IMAGE_GRAYSCALE (0) - load the image as a grayscale image; loading converts it to 8UC1 (1 channel unsigned char) image and performs grayscale conversion if required;

- CV_LOAD_IMAGE_COLOR (1) - load the image and convert it to a 8UC3 (3 channel unsigned char) image; it copies the grayscale channel to all color channels if required.

Example 1 - open an image in the current folder in the format it was saved:

```
Mat img = imread("cameraman.bmp", -1);
```

1.6. Accessing the data from an image

Matrix elements are indexed according to standard mathematical matrix notation. This means that the origin will be positioned at the top left corner of the image. The first index will indicate the row (increasing downwards) and the second index will indicate the column (increasing to the right). The following figure illustrates the indexing scheme:

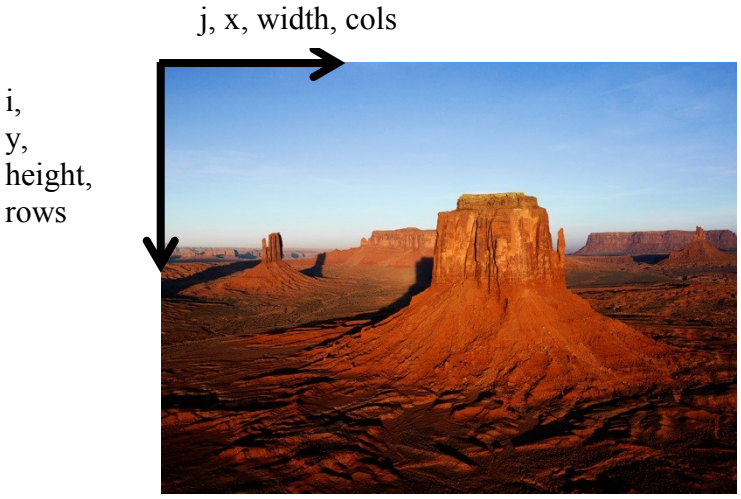

Figure 2. Indexing scheme for images

Always follow this convention to avoid indexing mistakes. When processing an image, first loop over the rows then over the columns.

To access the data from a grayscale image at row *i* and column *j* use the **`at`** method:

```
unsigned char pixel = img.at<unsigned char>(i,j);
```

Notice, that you need to provide the data type which is stored in the matrix (**`unsigned char`**).

For faster access, we can use the **`data`** pointer and the **`step`** field directly:

```
unsigned char pixel = img.data[i*img.step[0] + j];
```

All data is stored in a linearized manner, row after row and from left to right, starting from the **`data`** pointer. Padding may be introduced so ***avoid*** accessing via **`i*img.cols+j`** because it might give wrong results for padded images.

You can also use a pointer to the data from the *i*-th row:

```
unsigned char pixel = img.ptr(i)[j];
```

To access the 3 component color at row *i* and column *j* from a color image, use the proper type:

```
Vec3b pixel = img.at< Vec3b>(i,j);
unsigned char B = pixel[0];
unsigned char G = pixel[1];
unsigned char R = pixel[2];
```

Vec3b is a vector with 3 byte (**unsigned char**) components. |It is recommended for manipulating color images.

Code can be simplified by using the **Mat_<T>** templated subclass of the Mat class, which enables omitting the type for access operations. At the creation of a **Mat_<T>** object you must provide the underlying type that is stored in the matrix.

```
Mat_<uchar> img = imread("fname",CV_LOAD_IMAGE_GRAYSCALE);
uchar pixel = img(i,j);
```

Here we have also used the type definition **uchar** which stands for **unsigned char**. Accessing a value from a certain position permits both reading and writing operations.

1.7. Viewing an image

To view a loaded image use the **imshow** function followed by a **waitKey** call:

```
imshow("image", img);
waitKey(0);
```

This shows the image in a new window called *image* and waits for the user to input a key indefinitely. The **waitKey** function has only one parameter: how long it waits for a user input (measured in milliseconds). Zero means to wait forever.

Always follow each **imshow** operation with a **waitKey** command. Image windows can be moved and resized, which is desirable if you want to illustrate input and output side by side in the same configuration many times.

1.8. Saving/writing an image

To save an image to the disk use the imwrite function:

```
imwrite("fname", img);
```

The file name contains the path, the name and the extension, which determines the format of the image. You can save in multiple formats such as: *bmp, jpg, png*.

1.9. Sample function

The following sample code loads a grayscale image and transforms it into its negative image:

```
void negative_image(){
    Mat img = imread("Images/cameraman.bmp",
                     CV_LOAD_IMAGE_GRAYSCALE);
    for(int i=0; i<img.rows; i++){
        for(int j=0; j<img.cols; j++){
            img.at<uchar>(i,j) = 255 - img.at<uchar>(i,j);
        }
    }
    imshow("negative image",img);
    waitKey(0);
}
```

The image file must reside in the Images folder next to the project solution file.

1.10. Practical work

1. Download and build the OpenCV application.
2. Test the negative_image function.
3. Implement a function which changes the gray levels of an image by an additive factor
4. Implement a function which changes the gray levels of an image by a multiplicative factor. Save the resulting image.
5. Create a color image of dimension 256 x 256. Divide it into 4 squares and color the squares from top to bottom, left to right as: white, red, green, yellow.
6. Create a 3x3 float matrix, determine its inverse and print it.

References

[1] http://docs.opencv.org/2.4.13/modules/core/doc/basic_structures.html