Visual Computing and Multimedia

Abel J. P. Gomes

LAB. 2

IMAGE PROCESSING

1. Objectives
2. PGM files
3. Exercises
4. Homework

Departamento de Informática
Universidade da Beira Interior
Portugal
2011

Lab. 2

IMAGE PROCESSING

1. Objectives

1.1. General Objectives

In general terms, the idea of this labwork is to lead students to write C++ code to read, write, and manipulate images. The general objectives are then the following:

- Familiarize student himself/herself with reading/writing images from/to a file.
- Introduce and learn about some simple image processing algorithms.
- Improve student's skills with mastering object-oriented programming in C++, namely manipulating arrays and implementing constructors, destructors, copy-constructors, as well as overloading various operators.
- To have a first contact with C++ STL (Standard Template Library).
- Learn to document and describe programs.

1.2. Specific Objectives

More specifically, student has to write a package to implement the image data type using C arrays and C++ arrays (STL). The image data type should allow you to:

- Read an image from a file.
- Save an image to a file.
- Get the info of an image.
- Set the value of a pixel.
- Get the value of a pixel.
- Extract a sub-image from an image.
- Compute the average gray-level value of an image.
- Enlarge an image by some factor *s*.
- Shrink an image by some factor *s*.
- Reflect an image in the horizontal or vertical directions.
- Translate an image by some amount *t*.
- Rotate an image by some angle *theta*.
- Compute the sum of two images.
- Compute the difference of two images.
- Compute the negative of an image.

2. Specification of Image Class and its Methods/Functions

You will have to write a program that interacts with the above image data type and the user. The user should have the option to choose anyone of the above options *in any order*. Your program should be able to handle various user input errors.

Image type specification:

```
class Image
{
   public:
     constructor // default - no parameters
     constructor // with parameters
     destructor
     copy_constructor
     operator= //overload assignment
     getImageInfo
     getPixelVal
     setPixelVal
     inBounds
     getSubImage
     meanGray
     enlargeImage
     shrinkImage
     reflectImage
     translateImage
     rotateImage
     operator+ //overload addition for images
     operator- //overload subtraction for images
     reflect
     translate
     rotate
     negateImage

   private:
     int N; // no of rows
     int M; // no columns
     int Q; // no gray-level values
     int **pixelVal;
};
```

3. Exercises

Let us then design and implement the following class functions for PGM images:

1) **readImage(fileName, image)** Reads in an image from a file. The pixel values are stored in an array and the image width, height, and number of gray-levels is recorded in the appropriate fields. A NOT-PGM exception should be raised if the image file to be read is not in PGM format (*image* is an object of type *ImageType*).

2) **writeImage(fileName, image):** Writes out an image to a file in the appropriate format (*image* is

an object of type *ImageType*).

3) **getImageInfo(noRows, noCols, maxVal)**: It returns the height (no. of rows) of the image, the width (no. of columns) of the image, and the max pixel value (should be returned using "call by reference").

4) **int getPixelVal(r, c)**: Returns the pixel value at (r, c) location. An OUT-OF-BOUNDS exception should be raised if (r, c) falls outside the image.

5) **setPixelVal(r, c, value)**: Sets the pixel value at location (r, c) to *value*. An *OUT-OF-BOUNDS* exception should be raised if (r, c) falls outside the image.

6) **bool inBounds(r, c)**: Returns true if the pixel (r, c) is inside the image.

7) **getSubImage(ULr, ULc, LRr, LRc, oldImage)**: It crops a rectangular area within *oldImage*. Often, for image analysis, we want to investigate more closely a specific area within the image, called a Region of Interest (ROI). They are used to limit the extent of image processing operations to some small part of the image. The ROI is a rectangular area within the image, defined either by the coordinates of its upper-left (UL) and lower-right (LR) corners or by the coordinates of its upper-left corner and its dimensions. You can obtain the pixel coordinates of the UL and LR corners using *xv* by moving the cursor on the desired positions and by pressing the middle button (*Warning:* the first number displayed corresponds to *c* and the second to *r*). An *OUT- OF-BOUNDS* exception should be raised if UL or LR fall outside the image.

8) **int meanGray()**: Computes the average gray level value of an image (returns the results as an integer by truncating it).

9) **enlargeImage(s, oldImage)**: Enlarges the input image *oldImage* by some *integer* factor *s*. Enlarging an image is useful for magnifying small details in an image. There are various ways to enlarge a given image. Here, we will use a simple method: to enlarge an image by a given factor *s*, we must replicate pixels such that each pixel in the input image becomes an sxs block of identical pixels in the output image. This technique is most easily implemented by iterating over pixels of the output image and computing the coordinates of the corresponding input image pixel.

10) **shrinkImage(s, oldImage)**: Shrinks the input image *oldImage* by some *integer* factor *s*. Shrinking an image is useful, for example, to reduce a large image in size so that it fits on the screen. There are various ways to shrink a given image. Here, we will use a simple method: to shrink an image by a scale factor *s*, we must sample every *s*th pixel in the horizontal and vertical dimensions and ignore the others. Again, this technique is most easily implemented by iterating over pixels of the output image and computing the coordinates of the corresponding input image pixel.

11) **reflectImage(flag, oldImage)**: Reflects the input image *oldImage* along the horizontal or vertical directions (determined by the boolean "flag"). Reflection along either direction can be performed by simply reversing the order of pixels in the rows or columns of the image.

12) **translateImage(t, oldImage)**: Translates the input image *oldImage* by some amount *t*. The translation process can be performed with the following equations:

$$r' = r + t \quad (1)$$

$$c' = c + t \quad (2)$$

where t is an integer (note: the translation amounts in the horizontal and vertical directions can be different in general). There are some practical difficulties implementing translation using the above equations (see question 3 below).

13) **rotateImage(theta, oldImage)**: Rotates the input image *oldImage* by some angle *theta*. The rotation process requires the use of the following equations:

$$r' = r\,cos(theta) - c\,sin(theta) \quad (3)$$

$$c' = r\,sin(theta) + c\,cos(theta) \quad (4)$$

where *theta* is the angle of rotation (positive values correspond to counterclockwise rotation). Although the above formula is the basis of rotation, it only gets you halfway there because it will rotate an image about point (0,0). In most cases, what we really want is to rotate an image about its center. The following equations rotate an image about its center:

$$r' = r_0 + (r - r_0)\,cos(theta) - (c - c_0)\,sin(theta) \quad (5)$$

$$c' = c_0 + (r - r_0)\,sin(theta) + (c - c_0)\,cos(theta) \quad (6)$$

There are some practical difficulties implementing rotation using (3)-(4) or (5)-(6). Let us consider what happens to pixel (0,100) after a 90 degrees rotation using equations (3)-(4):

$$r' = r\,cos(90) - c\,sin(90) = -100\,sin(90) = -100$$

$$c' = r\,sin(90) + c\,cos(90) = 0\,sin(90) = 0$$

In this case, the pixel moves to coordinates (-100,0). This is clearly a problem since pixels cannot have negative coordinates. Let's now consider what happens to pixel (50,0) after a 35 degrees rotation:

$$r' = r\,cos(35) - c\,sin(35) = 50\,cos(35) = 40.96$$

$$c' = r\,sin(35) + c\,cos(35) = 50\,sin(35) = 28.68$$

The coordinates calculated by the transformation equations are not integers, and therefore do not index a pixel in the output image.

The first problem can be resolved by testing coordinates to check that they lie within the bounds of the output image before attempting to copy pixels. A simple solution to the second problem is to find the nearest integers to r' and c' and use these as the coordinates of the transformed pixel.

Please note that the C++ math functions *cos()* and *sin()* require that the angle is given in *radians*

(enter "man cos" or "man sin" from the command line to get a description of these functions). To convert degrees to radians, use the following formula:

$$\theta_{rad} = \theta_{deg} \times \pi / 180.0 \quad (7)$$

To use these functions successfully, you need to include the following header file in your program:

```
#include <math.h>
```

Also, you need to "link" your program to the math library (this can be done during compilation by appending *-lm* at the end of the command you are using to compile your program). There are some practical difficulties implementing rotation using the above equations (see question 4 below).

14) **operator+**: Computes the sum of two images. Addition is used to combine the information in two images. For example, you can implement simple "image morphing" using image addition (e.g., try adding together the following images from the image gallery: *person1.pgm*, *person2.pgm*, *person3.pgm*). If we add two 8-bit images, then pixels in the resulting image can have values in the range 0-510. One way to deal with this problem is by choosing a 16-bit representation (i.e., set Q=510) for the output image. Another way is to use the formula shown below:

$$O(r, c) = aI_1(r, c) + (1 - a)I_2(r, c) \quad (8)$$

where a is a constant in the interval [0,1] (addition is a special case when $a = 0.5$)

15) **operator-**: Computes the difference of two images. The main application of image subtraction is in *change detection*. If we make two observations of a scene and compute their difference, then changes will be indicated by pixels in the difference image that have non-zero values. If we subtract two 8-bit images, then pixels in the resulting image can have values between -255 and +255. This necessitates the use of 16 bit signed integers in the output image. However, the sign is usually unimportant and we just consider the absolute difference in which case we just need 8 bit integers:

$$O(r, c) = |I_1(r, c) - I_2(r, c)| \quad (9)$$

16) **negateImage**: Computes the negative of an image. This can be done by the following transformation:

$$O(r, c) = -I(r, c) + 255 \quad (10)$$

where *I(r,c)* is the input image (i.e., a grayscale image with 255 possible gray levels) and $O(r, c)$ is the output image.

4. Homework

Instructions

You should have three source code files: one for the application program containing the main function (*main.cpp*), one header file *image.h* that specifies the image data type, and a file *image.cpp* that actually

implements the image data type.

Describe the implementation of each function in detail. Each function should be discussed into a separate section with the title of each section being the same as the name of the function. The sections should be clearly separated from each other.

Questions:

Answering the following questions does not require that you have prior knowledge of image processing. Just spend some time thinking about them and give us your best possible answers along with some justification.

You do not have to do any extra coding regarding these questions but you are encouraged to do so. Interesting ideas which are implemented and demonstrated will get *extra credit* !!

Document in your report any extensions you might have made and let the TA know during the demo.

1. (2pts extra) The algorithm you have to implement in this assignment to enlarge a given image is actually very simple. Can you think of a better approach? Compare your approach with the approach suggested in this assignment (i.e., list possible advantages/disadvantages).

2. (2pts extra) The algorithm you have to implement in this assignment to shrink a given image is actually very simple. Can you think of a better approach? Compare your approach with the approach suggested in this assignment (i.e., list possible advantages/disadvantages).

3. (2pts extra) There is a practical difficulty with the direct application of the translation equations (1) and (2). When translating, what is done with the "leftover" space? For example, if we move everything one row down, what do we put in the top row? One solution is to fill the top row with a constant value value (typically black (0) or white (255)). Can you think of a better solution?

4. (4pts extra) We have already discussed two practical difficulties associated with rotation: (a) the case where the transformed pixel coordinates fall outside the image and (b) the case where the transformed pixel coordinates are not integers. In the first case, we suggested simply ignoring the transformed pixel coordinates that fall outside the range. This is probably enough in most cases. In the second case, we suggested finding the nearest integer neighbors to r' and c'. This approach, however, will not produce a value for every pixel in the output image. In other words, it will produce numerous "holes" in the rotated image where no value was computed. Can you suggest a way for dealing with this problem?

5. (2pts extra) This question is regarding image subtraction. By examining the difference image, you will notice some rather small non-zero values in image areas where there is not really any change (e.g., subtract "backg1.pgm" from "backg2.pgm" and look at the values of the difference image). These differences are due to sensor noise, slight changes in illumination and various other factors. Can you suggest of a way to deal with this problem?