My Little Photoshop

1 Instructions

At the end of this practical work, you will have to hand in an archive file following these instructions:

```
rendu-tpcs2-login_x.zip
|-- rendu-tpcs2-login_x/
    |-- AUTHORS
    |-- MyLittlePhotoshop.sln
    |-- MyLittlePhotoshop
        |-- Everything except bin/ et obj/
```

Of course you have to replace "`login_x`" with your own login.

Do not forget to check the following items before handing the file in:

- The `AUTHORS` must follow the usual format (a *, a space, your login and a new line).
- No `bin` or `obj` directories in the project.
- **The code must compile!**

2 Introduction

2.1 Objectives

During this practical work you will study the following topics:

- The use of `Bitmap` in C#
- Image Processing
- A bit of maths...

3 Course

3.1 Image Processing

> In imaging science, image processing is any form of signal processing for which the input is an image, such as a photograph or video frame; the output of image processing may be either an image or a set of characteristics or parameters related to the image.
>
> — Wikipedia

Image processing has been an active field since the 1920s with at first compression methods to make image transmission possible despite the very limited network capabilities at that time. It then developed for the second world war with radar but it is mostly since the 1960s that the

modern techniques were developed with the discovery of links with signal processing and the increasing power of computers.

Nowadays this field is everywhere in popular software or embedded boards to allow robots to determine what is around them for instance.

This processing is usually done by applying operators on an image which we can separate into three major families:

Point by point operators
: These are the simplest and consist in applying a color changing function on each of the pixels without worrying about their neighbours.

Local operators
: They can achieve more complicated processing by taking into account the pixels near the one being processed. They are necessary for filters like blurs.

Morphological operators
: Based on mathematical morphology[1], they are mainly used to extract elements from an image in order to interpret them and extract information.

By applying and combining these kinds of operators, one can retrieve a lot of information from an image for a lot of applications like text recognition, mapping or video tracking for instance.

This week's practical work will not deal much with information extraction, but rather with filters to enhance the image or prepare it for further steps.

3.2 The `Bitmap` class

To represent an image and be able to modify its pixels in C# , one can use the `Bitmap` class that provides the `GetPixel(x, y)` and `SetPixel(x, y, color)` methods.

To discover all its functionnalities, you can visit its MSDN page[2] or instanciate the class and let Visual Studio autocomplete the list of its attributes and methods after puting a dot at the end of the created object's name.

In order to use the `Bitmap` class add this line at the beginning of the files:

```
using System.Drawing;
```

4 Exercises

4.1 Before starting

During this practical work you will have to search for documentation on the internet. Here are some interesting links to help you:

- `http://en.wikipedia.org/wiki/Image_processing`
- `http://docs.gimp.org/en/plug-in-convmatrix.html`
- `http://en.wikipedia.org/wiki/Gaussian_blur`
- `http://en.wikipedia.org/wiki/Edge_detection`

You can download the base code on `http://perso.epita.fr/~acdc`.

To start, create a `Filters` class where you will implement everything.

[1] `http://en.wikipedia.org/wiki/Mathematical_morphology`
[2] `http://msdn.microsoft.com/en-us/library/system.drawing.bitmap%28v=vs.110%29.aspx`

4.2 Exercise 0: Point by point

Write the `MapPixels` function that browses the given `Bitmap` and replaces each of its pixels with the result of the application of the `f` function given as the second parameter (just as we did in OCaml – you will hear about it more in detail later).

Prototype:

```
public static Bitmap MapPixels(Bitmap img, Func<Color, Color> f)
```

4.3 Exercise 1: First filters

Write the functions described below representing point by point filters to be given to the `MapPixels` function.

Greyscale
Turns the image to grey scales. Remember to search for the appropriate coefficients for each component to get good results.

Pinkify
Makes the image more pink while keeping it recognizable. You can calculate a "pink scale" the way you did for the previous filter and return the average of this value and the base color for example.

Binarize
Turns the image into a binary one (black and white only) with a threshold method (other methods are significantly better but can not be applied point by point).

Invert
Turns the image into its negative (black becomes white and vice-versa).

Prototypes:

```
public static Color Greyscale(Color c)
public static Color Pinkify(Color c)
public static Color Binarize(Color c)
public static Color Invert(Color c)
```

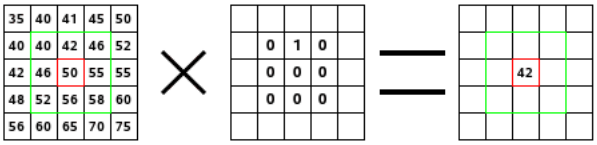
4.4 Exercise 2: Mirror, beautiful mirror

Write the `MirrorH` and `MirrorV` that apply respectively a vertical and horizontal symmetry to the image.
Prototypes:

```
public static Bitmap MirrorH(Bitmap img)
public static Bitmap MirrorV(Bitmap img)
```

4.5 Exercise 3: Convolution

Write the `Convolution` function that applies a square convolution matrix (same width and height) to an image.
To do that you have to browse the image and replace each component of the current pixel's color with the sum of the products of the equivalent component of a nearby pixel and its coefficient in the matrix.
Here is a schematic to demonstrate the algorithm:



You can consider the pixels outside the image as being black (0 everywhere).
PROTIP: You can write an `IsValid(int x, int y, Size s)` function to test if the coordinates are inside the image.
PROTIP 2: To avoid overflows due to matrix errors or floating point approximation it is advised to check if the components stay in the $[0; 255]$ range.
PROTIP 3: You can retrieve the width of the matrix with `mat.getLength(0)` (or 1 for the height but that should not be needed here).
PROTIP 4: Don't forget to work on a copy of the image to avoid interfering with your own processing.

Prototype:

```
public static Bitmap Convolution(Bitmap img, float[,] mat)
```

4.6 Exercise 4: Enter the Matrix

Define the `AverageMat`, `GaussMat` and `EdgeMat` matrices that will be used by `Convolution`, a size of 3x3 is enough here and should not take too long to be applied:

AverageMat
Average blur matrix.

GaussMat
Gaussian blur matrix. Do not forget to normalize (sum of the fields = 1) to avoid overflows.

EdgeMat
Edge detection matrix. Search for Roberts, Prewitt, Sobel and Canny methods (sorted by difficulty).

To define a matrix in C# you can do it like this:

```
public static float[,] MyLittleMatrix = new float[,]
                                                {
                                                    {0, 0, 0},
                                                    {0, 1, 0},
                                                    {0, 0, 0}
                                                };
```

4.7 Bonus

Here are some bonus ideas sorted by difficulty:

Easy

- Brightening filter (log)
- Darkening filter (exp)

Medium

- Other convolution matrices
- Rotations

Hard

- Morphology filters (erosion, dilation, opening, closing)

Over 9000

- Blob detection (segmentation and union-find)
- Use Fourrier transforms (FFT) to speed up a lot
- Use `ClGlInterop`[3] to speed up even more

5 Expected results

Figure 1: Base image

Figure 2: Greyscale

Figure 3: Pinkify

Figure 4: Binarize

[3] `http://www.cmsoft.com.br/index.php?option=com_content&view=category&layout=blog&id=137&Itemid=197`

Figure 5: Invert

Figure 6: MirrorV

Figure 7: MirrorH

Figure 8: AverageMat

Figure 9: GaussMat

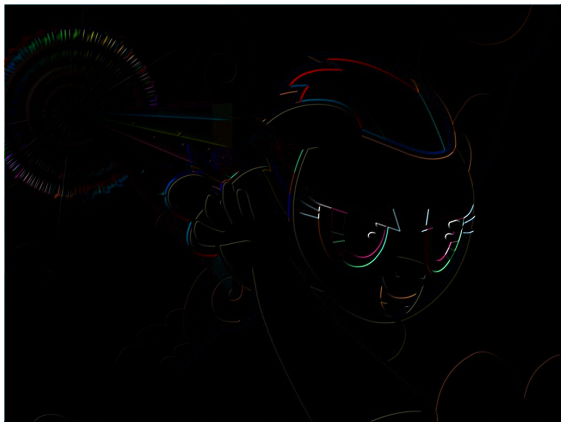


Figure 10: EdgeMat

It's dangerous to code alone !