Introduction to Programming

— Practical #6 —

1 Images

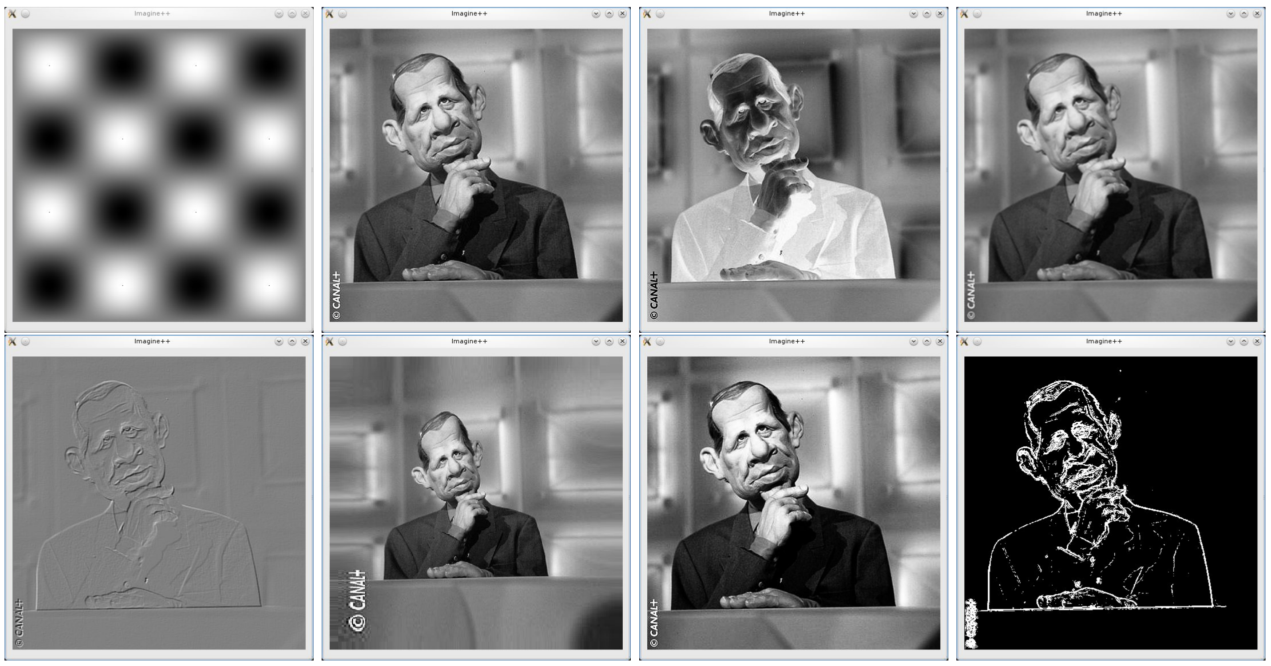

Figure 1 – Two images and various processing on the second one (negative, blur, relief, deformation, contrast and edges).

In this practical, we will play with bidimensional arrays (but stored in 1D arrays), first static then dynamic. To change from matrices, however fascinating they can be, we will work with images (figure 1).

1.1 Allocation

1. *Get the project :*
   Download the file `Tp6_Initial.zip`, decompress it and launch your favorite development environment, Qt Creator.
2. *Fill the memory :*
   Nothing to do with what we will do after, but it is nice to have seen it at least once... Do, in an infinite loop, allocations of 1000000 integers without deallocation, each followed by a pause of 0.5 seconds, and look at the process size growing. (Use `Ctrl+Shift+Echap` to get the task manager under Windows, use command `top` in a terminal in Linux or MacOS).[1]

1.2 Static arrays

3. *Gray levels :*
   A black and white image is represented by an array of pixels of constant dimentions `W=300` and `H=200`. Each pixel `(i,j)` is a `byte` (integers between 0 and 255) with value 0 for black and 255 for white. The

1. Your program should crash at a certain point, when no more memory is available. If not, try in mode Release to have a true handling of the heap (The mode Debug may behave differently...)

origin of coordinates is the top left, `i` is horizontal and `j` is vertical. In a mono-dimensional array of `byte` `t` of size `W*H` store the pixel `(i,j)` in `t[i+W*j]` :

— Create a black image and display it with `putGreyImage(0,0,t,W,H)`.
— Same for a white image.
— Same for a gradation from black to white (be careful with risks of Euclidean division with integers, conversion to double may be necessary).
— Same with $t(i,j) = 128 + 127\sin(4\pi i/W)\sin(4\pi j/H)$ (see figure 1). Use

```
#include <cmath>
```

to get mathematical functions and constants : `M_PI` vaut π.[2]

4. *Colors :*
Display with `putColorImage(0,0,r,g,b,W,H)`, an image in color stored in three arrays `r`, `g` and `b` (red, green, blue). Use function `click()` to wait a user mouse click after each new display.

1.3 Dynamic arrays

5. *Dimensions from the keyboard :*
Modify the program so that `W` and `H` are not constant anymore but values input from the user with the keyboard. Do not forget deallocation.

1.4 Load a file

6. *Color image :*
The call `loadColorImage(srcPath("ppd.jpg"),r,g,b,W,H);` loads the file `"ppd.jpg"` that is located in the source folder (`srcPath`), allocates by itself the arrays `r,g,b`, fills them with image pixel values, and assigns also `W` and `H`.[3] Warning : do not forget to deallocate arrays `r,g,b` with delete [] after use.
— Load this image and display it. Do not forget deallocation, again.

7. *Black and white image :*
The function `loadGreyImage(srcPath("ppd.jpg"),t,W,H)` does the same but converts the image to grayscale. Display this image...

1.5 Functions

8. *Split the work :*
We will only work with the graylevel image from now on. Write functions to allocate, destroy, display and load images :

```
byte* allocImage(int W,int H);
void deallocImage(byte *I);
void displayImage(byte* I,int W,int H);
byte* loadImage(const char* name,int &W,int &H);
```

9. *Files :*
Create files `image.cpp` and `image.h` for the functions above...

1.6 Structure

10. *Principle :*
Modify the preceding program to use a structure :

```
struct Image {
  byte* t;
  int w,h;
};
```

`AllocImage()` and `LoadImage()` can return some `Image`.

11. *Independence :*
To avoid having to remember how pixels are stored, add :

2. To be exact, `M_PI` is not standard, some compilers may not define it. Don't worry, your compiler has it.
3. The size of the image is stored in a header of the file, and read by the function. This is fortunate, otherwise the program would have to know in advance the size of the images it uses.

```
byte get(Image I,int i,int j);
void set(Image I,int i,int j,byte g);
```

12. *Processing :*
    Add in `main.cpp` different functions to process images

    ```
    Image negative(Image I);
    Image blur(Image I);
    Image relief(Image I);
    Image edges(Image I,double threshold);
    Image deform(Image I);
    ```

    and use them :

    (a) `negative` : invert black and white by an affine transform.

    (b) `blur` : each pixel becomes the average of itself and its 8 neighbors. Beware of pixels at image border that have fewer than 8 neighbors (leave them unchanged and use instruction `continue`!). In general, when you do image processing, you must always be careful not to get outside the image.

    (c) `relief` : the derivative along a diagonal gives the impression of cast shadows from an oblique lighting.
    — Approximate this derivative by finite difference : it is proportional to $I(i+1,j+1)-I(i-1,j-1)$.
    — Rescale the gray levels by an affine function to get back in range 0 to 255.

    (d) `edges` : compute by finite differences the horizontal $d_x = (I(i+1,j) - I(i-1,j))/2$ and the vertical d_y derivatives, then the norm of gradient $|\nabla I| = \sqrt{d_x^2 + d_y^2}$ and display in white the points where this is above some threshold.

    (e) `deform` : Build a new image with the principle $J(i,j) = I(f(i,j))$ with f smartly chosen. You can use a sine to to from `0` to `W-1` and from 0 to `H-1` in a nonlinear fashion. [4]

1.7 Final clean-up

13. Clean up everything : it is likely you have put in comments the code for first questions in order to save time during further development. Put them back uncommented, and check that everything still compiles and runs as expected.

14. If you have time left, you can try :
    — Make a reduced image.
    — Instead of negative, you can change the contrast, for example by multiplying (by a small factor λ) the difference between a pixel and the average of its neighbors (this is the negative of Laplacian) : [5]

$$J(i,j) = I(i,j) + \lambda \left(I(i,j) - \frac{I(i-1,j) + I(i+1,j) + I(i,j-1) + I(i,j+1)}{4} \right)$$

4. Whatever your choice, if function f goes outside the image I, just replace with a white pixel.
5. Saturate values that go out of range 0 to 255, otherwise it will perform a modulo and you will get for example white pixels appearing in a dark region.