



Final project report

Multimedia Applications

Edited on 06/10/2020

Florian WOTIN

Elodie PARISOT

Antoine COULON

Table of contents




Table of contents	1
Introduction	2
1. Presentation and organization of the project	2
1.1 Presentation of the project	2
1.2 Use of git	3
1.3 Group work	3
1.4 OpenCV	3
1.5 Structure and architecture of our code	4
1.6 QT/QT Creator	4
2. Implemented features	6
2.1 Resizing	6
2.2 Cropping	6
2.3 Color / Grey image	6
2.4 Blur / Gaussian blur	7
2.5 Rotation	7
2.6 Dilatation / Erosion	8
2.7 Contrast / Brightness	9
2.8 Panorama / Stitching	9
2.9 Canny edge detection	10
Conclusion	11
Encountered difficulties	11
What the project has taught us	11
GitHub link	11

Introduction

We are a team of 3 students composed of Elodie PARISOT, Antoine COULON and Florian WOTIN. Each one of us is currently studying at ISEP, preparing the Software Engineering degree. We all decided to experience multimedia applications in order to discover a specific field led by low level programming and high level computations.

This report will retrace our work: the different softwares and libraries we used, the distribution of tasks, the functionalities we implemented, ...

To begin, here is a small summary of our team with our situation at ISEP:

	Elodie Parisot	Software Engineer apprentice at Naval Group
	Florian Wotin	Software Engineer apprentice at Thales LAS
	Antoine Coulon	Software Engineer apprentice at Thales

1. Presentation and organization of the project

1.1 Presentation of the project

The main goal of this project is to **recreate a software implementing various functionalities related to image manipulation**. For that, we have to develop a small image editor, allowing us to go through all functionalities from a user interface. We have to use for this the image manipulation. To develop these features, we have to use **OpenCV**, a wide open-source library which comes in with a lot of built-in functions in a lot of fields such as image / video manipulation.

To implement these features on top of OpenCV, we decided to develop a user interface using **Qt**, giving us a complete environment to develop interfaces. This user interface allows each user to manipulate images through windows actions, dodging a command line interface.

1.2 Use of git

In order to be able to work efficiently as a team on the realization of the image manipulator, we have created a **Git repository** that contains the code of our project.

This common repository allows you to move forward on distributed and parallel tasks as the project progresses.

We also created a branch for the GUI part.

Each of us, after advancing the code, we would do a push command and merge if necessary.

1.3 Group work

The project has clearly been cut in two distinct parts :

- implementing all the functionalities on top of OpenCV
- developing a user interface using this implementation

As a team of three developers, we **split the work by functionality**. Each one of us picked a set of functionalities and then developed them. Due to the fact that we used Git for the code versioning, the split was really simple ; we dedicated a branch for some features, and a branch for the qt application.

Once our core functionalities were developed, we kept the same organization for the UI components : each person that developed a functionality had to implement the UI components and interactions to make it work.

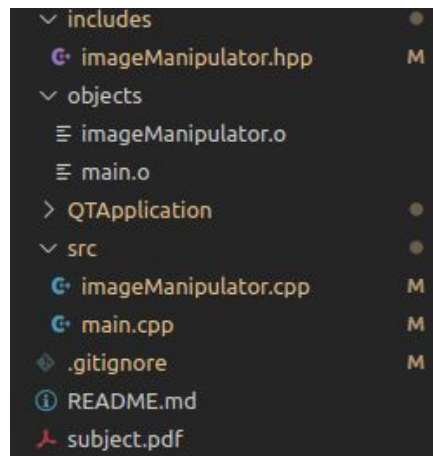
1.4 OpenCV

Then, for the development of the different features, we decided to use the **vscode IDE** (or Visual Studio Code). This IDE is very handy because it is compatible with opencv, the library we used for image processing.

OpenCV (Open Source Computer Vision) is a library of programming functions mainly aimed at real-time computer vision. In simple language it is library used for **Image Processing**. It is mainly used to do all the operation related to Images.

1.5 Structure and architecture of our code

This is how our code is structured on visual studio code:



The QtApplication file appears since the screenshot was taken after designing the Qt project (detailed more in the next part). But the principle remains the same. At the end of the project, the main.cpp has finally not been used (since it is the one from qt that is used). The picture shows that the includes have been separated from the sources. The **includes represent the headers** of the C++ files; they are very important since it is here that we announce the variables and methods that we will define in the C++ files. So inside the "imageManipulator.hpp", we have defined the different variables we would need such as the image and the different methods such as the function to switch from a color image to a grayscale image, the function to create a panorama...

The **sources contain the different functionalities that we have implemented**. We decided to gather them all in a single class named "imageManipulator.cpp", for the sake of simplicity for the future and for use in Qt.

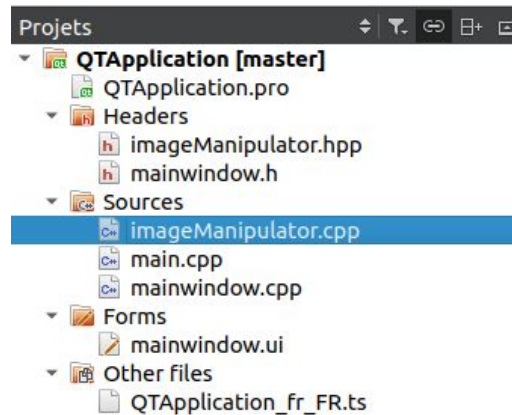
So, in the source file "imageManipulator.cpp", we defined these functions which act directly on a previously defined image.

1.6 QT/QT Creator

Qt is the **faster, smarter way to create innovative devices**, modern UIs & applications for multiple screens. Qt is a framework for C++ that started out as GUI library but has become much, much more. For example we can use 3D graphics with OpenGL and this is very useful. One key feature is the signal and slot mechanism, a great way to get different pieces of our program to send messages to each other. So we can use Qt creator to design our GUI, and we **can manage signals & slots** to define QObject's actions and interactions. We can, for example, open an image, and then depending on the button we click on, the image can get lighter, darker,...

After deploying all the features we wanted for our ImageManipulator class on visual studio code (this part will be explained in "Implemented features"), we have created a new project on qt creator.

The project consists of the following components:



In Qt creator, there is an **edition mode** which will be useful for the design of the graphical interface, and to start linking signals and slots. Qt projects contains a ui file which is like an xml containing the detail of our designed GUI, with default values, and so on. Here we named it mainwindow.ui. We can then create widgets in the ui, and give them names so we can use them in the code part. We can also **define slots, which represent the different actions** we will execute when the user will click on a button, or others.

Then we can complete the header and source parts.

The imageManipulator.cpp represents all the functionalities we have created in visual studio code (detailed later). We can use them in our mainwindow.cpp.

Here is an example for one of the features:

Let's consider the function that will **convert our image to grayscale** in the imageManipulator.cpp file.

Let's consider that we **have selected and displayed an image**.

The user clicks on the "to Grey" button then the slot we defined **will call the function** provided in imageManipulator.cpp.

We have also created a **function that converts the image matrix (using in C++) to QImage** so that it can be displayed on the qt interface.

2. Implemented features

2.1 Resizing

In order to resize an image, we have used the function **resize from opencv**.

It takes some parameters :

- the **source image**
- the **destination image**
- the **destination image size** : if it is equals to 0, it is computed as `dsize = Size(round(fx*src.cols), round(fy*src.rows))`
- the **scale factor along the horizontal axis** : if it is equals to 0, it is computed as `(double)dsize.width/src.cols`
- the **scale factor along the vertical axis** : if it is equals to 0, it is computed as `(double)dsize.height/src.rows`
- **interpolation** (an integer) which represent the method we want to use : it is either INTER NEAREST or INTER LINEAR. We will use the INTER LINEAR equals to 1.

Either dsize or both fx or fy must be different to "0".

Then we can perform the resizing of our image.

2.2 Cropping

In order to crop an image, named image, by a certain width and a certain height,, we have used : **image(Range(0, image.size().height - height), Range(0,image.size().width - width)).**

Then we can perform the cropping of our image.

2.3 Color / Grey image

In order to transform our image into a shade of gray or to put it back in color, we use the function **cvtColor from opencv**.

It takes 3 arguments : the source image, the destination image and the color code transformations. For the grey image we will use **COLOR_BGR2GRAY** and to color an image we will use **COLOR_GRAY2BGR**.

2.4 Blur / Gaussian blur

We can also **blur the image**. There are several types of blurring. For example, we can **smooth** an image. It is usually called “blur”. It is a simple and frequently used image processing operation. There is also the **gaussian blur**. It is the most useful filter but not the fastest. This filter convolves each point in the input array with a Gaussian Kernel. Then, it sums them all to produce the final image.

In order to smooth an image, we can use the function **blurImage** from opencv. It takes 4 arguments :

- the source image
- the destination image
- the size of the kernel we will use

In order to use gaussian blur, we can use the function **GaussianBlur** from opencv. It takes some arguments :

- the source image
- the destination image
- the size of the kernel we will use. The width and the height must be odd and positive numbers. Otherwise, the kernel's size will be calculated thanks to sigma x and sigma y.
- sigma x: the standard deviation in x. If it is equals to 0, it means that it is calculated using kernel size.
- sigma y: the standard deviation in y. If it is equals to 0, it means that it is calculated using kernel size.
- the borderType

Then we can blur our image using both type of blurring.

2.5 Rotation

In order to rotate an image we use the function **getRotationMatrix2D** from opencv which return a Mat. This function has 3 arguments:

- the center of the rotation of the the source image.
- the angle of rotation in degrees (the value will be positive for counter-clockwise direction and negative for clockwise rotation)
- The scaling factor of the image. If we want the original size, the factor will be 1.

Then we use the function **warpAffine**(InputArray src, OutputArray dst, InputArray M, Size dsize, int flags = INTER_LINEAR, int borderMode=BORDER_CONSTANT, const Scalar& borderValue=Scalar()).

This function will take some arguments such as :

- the source image
- the destination image (same type as the source image)
- M (a 2*3 affine transformation matrix)
- dsize : will be the size of the destination image
- flags, which represents the method of interpolation we use
- the borderMode, which represents the pixel extrapolation method. It exists these kind of values : BORDER_REPLICATE, BORDER_CONSTANT, BORDER_REFLECT, BORDER_WRAP, BORDER_REFLECT_101, BORDER_TRANSPARENT and BORDER_ISOLATED
- the borderValue. For example if we have choose to use "BORDER_CONSTANT", this argument will define the value used for the border.

Then we can perform the rotation of our image.

2.6 Dilatation / Erosion

Erosion and dilation are basic morphological operations. They are **used mainly for removing noise, isolating individual element,...**

Dilation consists of **convoluting an image with some kernel**. This kernel can have any shape or any size, but usually we use a square or a circle. The center of this kernel is usually the anchor point. Then we scan the kernel over the image. We compute the **maximal pixel value** overlapped by B and replace the image pixel in the anchor point position with that **maximal value**. This operation **maximize bright regions** (as the name dilation shows it).

Erosion is the opposite of dilation. It consists to compute a local minimum over the area of the kernel. As opposite to dilation, we scan the kernel over the image. We compute the **minimal pixel value** overlapped by B and replace the image pixel in the anchor point position with that **minimal value**. **Dark areas get bigger**.

For both dilation and erosion functions, in C++, we **retrieve the size of the dilation or erosion**, as well as the type we want to apply (circle, ellipse, rectangle). For example : Rectangular box: MORPH_RECT, Cross: MORPH_CROSS, Ellipse: MORPH_ELLIPSE. Then we use the **getStructuringElement** function which is a function of opencv to create the kernel to use.

Then, we use either the function dilate or erode from opencv.

The function dilate and erode take 3 arguments : the source image, the output image, and the kernel we have created. By default it is a simple 3*3 matrix. Otherwise, we can **specify its shape** by using the getStructuringElement (that is what we have used here).

The function `getStructuringElement` is used like this for erosion: `Mat element = getStructuringElement(erosion_type, Size(2*erosion_size + 1, 2*erosion_size+1), Point(erosion_size, erosion_size));`

We must specify the **size of our kernel** and the anchor point. If not specified, it is automatically the **center of the kernel**.

Then we can perform the erosion or the dilation of our image.

2.7 Contrast / Brightness

In order to modify the contrast or the brightness of our image named `image`, we will use the function `convertTo(image, rtype, alpha, beta)` from `opencv`.

This function takes 4 arguments :

- output image
- `rtype`, it is the type of the output image. If `rtype` is negative, the type of the output image will be the same as the input image. By default, in our application, we put "-1".
- `alpha`, each pixels in the input image will be multiplied by this number before assigning to the output image.
- `beta`, this value will be added to each pixels in the input image and assigned to the output image.

For example if we want to increase the brightness by 100, we will take an `alpha = 1`, and a `beta = 100`.

If we want to change the contrast this time, and increase the contrast by 4, we will choose an `alpha = 4`, and a `beta = 0`.

We can combine both functionalities : contrast and brightness.

2.8 Panorama / Stitching

In addition to functionalities directly editing the inputted image, we also developed a functionality allowing to **create a panorama from a splitted input source**.

The core here is to analyze chunks of an images and to try to detect commons areas that should be merged. This starts from a stitching point by point for each image comparison, that can result on a merge of images if the algorithm detects common areas.

The power of the functionality is that we can select many input sources, we are not limited to only two sources.

The first step of the process is to use the **OpenCV Stitcher class** and to set the "panorama" mode. Once instantiated, we can use the "stitch" function alongside with a vector of images representing our input sources.

Finally, we can retrieve our final panorama image represented as a matrix if the stitching could be processed.

What could make the stitching process fail?

The stitching will basically **search for each common area** through all given input sources. However, if one of the images doesn't have enough common area with the other images, meaning that there is a lack of information between all images, the panorama won't be processed.

If we could compare this process to a game, we would say that stitching is playing puzzles.

If you are missing few pieces of the puzzles, you won't be completing it ; it is the same thing for the panorama.

2.9 Canny edge detection

The Canny edge detector is an **edge detection operator** that uses a multi-stage algorithm to detect a wide range of edges in images. It was developed by John F. Canny in 1986.

Thanks to OpenCV, we can apply the Canny edge detection on various images.

How can we process it?

1. We need to establish a ratio of lower:upper threshold of 3:1 (with the variable ratio).
2. We need to set the kernel size (for the Sobel operations to be performed internally by the Canny function).
3. We set a maximum value for the lower Threshold.

All this parameters are allowing us to accept or not a pixel from the source image (as an edge). Making these change directly on the screen can show us the result difference when we change the threshold of Canny. It can produce several results.

To show the result of the detection, the best way is to copy the matrix of the inputted image and to turn it to gray. Besides that, the Canny detector produces a black background with bright lines representing edges. When we merge both, we can now have our original image with the highlighted edges.

Like we said before, changing the threshold can produce different results. To verify it in live, we used the Trackbar class given by OpenCV letting the user changing the threshold from min to max. Each time the trackbar changes the threshold value, the canny edge detection is launched and the applied masked is updated, letting the user check the difference.

Conclusion

Encountered difficulties

Qt Creator, Qt in general and opencv are quite simple to use, and they are intuitives. But one thing was a little bit harder for us : conversions from opencv to qt. Indeed, we are doing our calculations on images with opencv, and then we have to convert it into a QImage for Qt. It is simple for this purpose, but we encountered difficulties to resize images, because when we resize image in opencv, we can't just convert it after.

The main difficulties of this project for us, were our personal issues. Indeed, we all had personal issues during the project and it made us fall far behind from what we expected...

What the project has taught us

Being very unfamiliar with C++, this allowed us to **discover a new language. C++ is widely used**. Courses and tp were very useful since we had studied how to structure C++ code and different uses of opencv. Moreover, **image processing is very important** in everyday life since it is very present. For example, on different social networks, we can **apply filters on our faces**, there are **motion detection**,... This allowed us to make a link with the different image processing applications existing on smartphones. So **this project was very interesting**.

Moreover, C++ must be **compiled and executed**. That is different from other languages such as Java, for example, which uses virtual machines / interpreters. In addition, Java supports **automatic garbage collector**, whereas with C++ we have to be careful to keep memory space, because memory management is not automatic. We also had to pay attention in C++ to **the use of pointers**.

Finally, it was good to learn the basis of Qt, because this library is really popular and powerful, so it is always important to be familiar with.

GitHub link

For this project, we made our GitHub repository public. If you are interested in, you can check it here : https://github.com/arcreane/multimedia-equipe_007

The simplest way to use our application is to open the project in QT Creator, and then press the run button. It will compile and run our program.