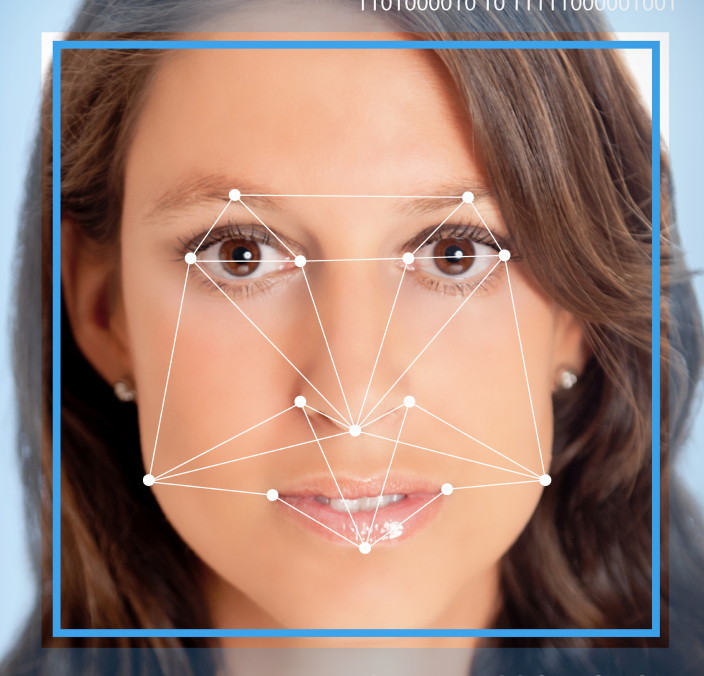
***Face recognition for medical data display***

*Techniques de programmation en C++*

14 /12/15



**Our Team:**

|  |  |  |  |
| --- | --- | --- | --- |
| BDDjpg/img3.jpg | BDDjpg/img6.jpg | BDDjpg/img5.jpg | ../Desktop/thumb_10409203_10205410289197249_7333369375538429535_n_1024.jpg |

**Ines Frebault, Guillaume Laurent, Jonathan Couturier, Karim Mimouni**

Table of contents

[Introduction 3](#_Toc438027172)

[Functional analysis 3](#_Toc438027173)

[OpenCV 3](#_Toc438027174)

[Architecture 4](#_Toc438027175)

[Layers 4](#_Toc438027176)

[Functions 5](#_Toc438027177)

[Data base 5](#_Toc438027178)

[Description 5](#_Toc438027179)

[UML diagram 6](#_Toc438027180)

[Face recognition 7](#_Toc438027181)

[Principle 7](#_Toc438027182)

[Implementation 7](#_Toc438027183)

[Face Extraction 8](#_Toc438027184)

[Results 8](#_Toc438027185)

[Graphical User Machine 10](#_Toc438027186)

[Launch screen 10](#_Toc438027187)

[Recognition screen 10](#_Toc438027188)

[Medical record screen 11](#_Toc438027189)

[Non-regression tests 11](#_Toc438027190)

[Conclusion 12](#_Toc438027191)

# Introduction

As part of our engineering background to ISEP, the module "Techniques de programmation sous C++" allowed us to understand the computer language C ++ and the use of OpenCV which is a free library. To do this, our group has decided to build a project relating to health, which enabled us to acquire and use the knowledge acquired during this module.

# Functional analysis

In this first part, we will analyze the project we choose by studying its specifications, identify all the concepts and introduce the technical solutions we’ve found.

In a second part we will explain these technical solutions and how we implemented them in our project.

Finally, we will conclude with the issues encountered, their solutions and the improvement points.

|  |  |  |
| --- | --- | --- |
| **Functionality :** | **Tools :** | **Developer :** |
| * Database | C++ | Guillaume |
| * Image processing and face recognition | C++ & openCV | Inès |
| * GUI | C++ & visual studio | Jonathan |
| * Interactions with the database | C++ | Karim |

## OpenCV

For our project, we had to use OpenCV (OpenSource Computer Vision) library, wich is a free library of programming functions, designed for computional efficiency with a focus on real-time applications. Generally used for image or video processing and for learning algorithms, it has C++, C, Python and Java interfaces and supports Windows, Linux, Mac OS, iOS and Android. The support is provided by the robotics company Willow Garage, since 2008.

In our case, we used some image processing operations in order to detect and compare faces with those store in our BDD ; we also used video operations to access the webcam of the device, take the picture and save it within our program’s folder.

## Architecture

### Layers

The source code of our application will be organized according to the architecture we developed. Inspired by the MVC architecture, the GUI part will contain the management of the user interface; the Business layer will contain the image processing algorithms, and the Com part will handle the communication with the database. The different layers that represent our architecture are represented in the next illustration:

We can resume the different interactions between layers with the following graph:

The scenario of our application is represented on the next image:

Architecture

### Functions

According to our architecture, we have implemented the following classes containing the functions we need to do our program:

|  |  |  |  |
| --- | --- | --- | --- |
| **Layer** | **Class** | **Input** | **Output** |
| Business | FaceExtraction : detects the face and extracts the associated matrix | Image | Matrix |
| Business | FaceRecognition : compares the face with the database’s images | Matrix | Int |
| Com | SendMatrixToBusiness : sends the picture database to the business layer |  | Vector<int> Vector<string> |

## Data base

### Description

Our database contains four tables, the three main tables are:

* the main table: it has the primary key **id** which is incremented for each patient, it includes the personal information of each patient
* disease table: it has the primary key **id** and a foreign key id patient; this division allows a patient to have between 0 and n diseases.
* picture table: it has primary key id of each line and for the foreign key id also patient; it allows each patient to have between 1 and n associated photos, which will be useful for the learning phase of our algorithm.

Main table:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **id** | Name | Firstname | Gender | Birthdate | SSN | Address | Telephone | Email |
| Int | Text | Text | Char | MMDDYY | String | Text | String | Text |

Disease table:

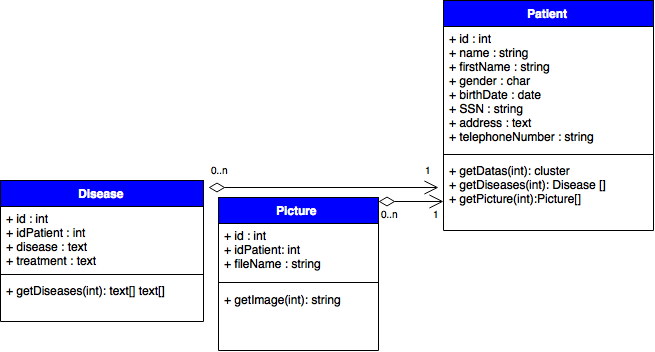
|  |  |  |  |
| --- | --- | --- | --- |
| **id** | id patient | Disease | Treatment |
| int | int | Text | Text |

Picture table :

|  |  |  |
| --- | --- | --- |
| **id** | id patient | Filename |
| int | int | text |

The last table is insertdata : it contains a primary key **id** andthe id of the patient recognized from the business layer’s algorithm. It is a solution we have found to communicate between the Business layer and the GUI layer, both using a local (different) database.

### UML diagram



# Face recognition

In this part, we will describe the algorithms we used in the Business layer. To recognise the patient on the input image, we chose the Eigen algorithm available in the “Face” contrib librairy of OpenCV: two other algorithms are also available but the Eigen algorithm was proven to be the best with our database.

## Principle

The Eigen algorithm use the Principal Component Analysis (PCA) algorithm. Given an *n\*p* matrix, the PCA calculates *p* principal components to summerize most of the dataset variability. The first principal component is a subspace of the dataset that is representative of the direction in which the data vary the most; the second principal component is a linear combination of the variables uncorrelated with the first principal component and that have the largest variance, and the same principle applies for the other principal components. The principal components are names “eigenvectors” that are ordered by the eigenvalues; both are calculated as a linear combination with the Covariance Matrix.

The algorithm is then performed in two steps:

* the training samples and the input image are represented as a linear combination of the eigenvectors
* the closest representation of the projection of the input image compared to the projections of the training samples is chosen as the result

## Implementation

Since the algorithm works with same-sized gray-scale images, we do the resizing in the PreTreatment file; we also have a histogram equalization option in the **treatment()** function to improve the results because the algorithm takes in account the illumination.

We will look a little bit more on the details of the face librairy:

The **FaceRecognizer** class is defined with **Algorithm** as its base class.

We have to start by creating the recognizer with the method **createEigenFaceRecognizer()**, where we can choose a number of eigenvectors to use (all in our case) and a confidence prediction value (no threshold in our case).

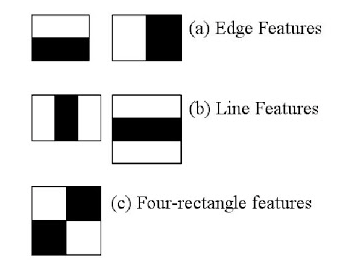
We used the **train()** function from the class FaceRecognizer where we specify the images and the corresponding labels to train the algorithm, as well as **predict()** where we specifiy the image to perform the algorithm on, the reference of its predicted label value and the reference of the confidence we want on the prediction. As a result, we get the corresponding label of the image recognised as the closest of our input image, or -1 if no image was identified as our patient.

There are a lot of functions we could have used in the class FaceRecognizer, like the function **save()** to save the state of our recognizer after the training, but we didn’t have the time to do so.

## Face Extraction

To improve the results, we decided to proceed on a face extraction before we launch our EigenFaceRecognizer to be more precise and avoid perturbations that could be on the picture.

The method we are using is the haar cascade classifier. The aim is to apply features (like the one represented under) at each point of the image and to compute the sum of the pixels in the white areas and in the the black areas to define some rules to recognise the object.



In order to reduce computation time, the image is divided in regions and the features are grouped; if the region is defined as the object we want with the first group of features, we apply stricter groups features to avoid fake detections; if not, we skip the whole region. In our case, our classifier has 38 groups of features: it is the CascadeClassifier that is contained in a .xlm file.

We have the **load()** method to use it, as well as **detectMultiScale()** method, that takes as input a reference to the image, a reference of the vector of objects detected from the image, the scale factor (how much the image is reduced at each iteration), the minimum number of neighbors, the value of the flags (not used), the min size of the object and the max size of the object. This algorithm can detect as many faces in a picture as we want, but in our case we know there is only one face per picture so we return the last face found in the picture in our FaceExtraction class.

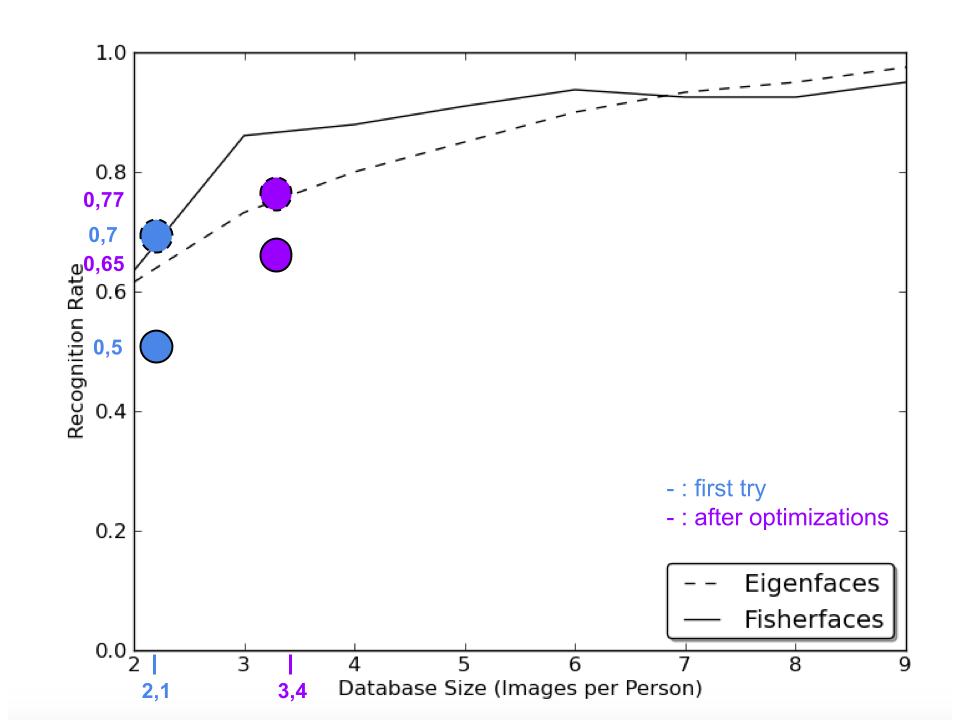
## Results

After an optimization phase, we have good results on the face recognition algorithm. The Face Extraction works perfectly well, with 100% of faces isolated from the pictures in the base. The scores for the recognition algorithm are showed on the next chart :

|  |  |  |  |
| --- | --- | --- | --- |
|  | Eigenfaces | Fisherfaces | LBHF |
| First try : | 0,75 | 0,5 | 0,5 |
| After optimisations : | 0,764705882 | Not tested | 0,647058824 |

This chart shows the recognition rate we obtain with our picture database, composed of some pictures of classmates. Optimisations include face extraction, treatment() function and threshold.

We’ve found some theorical results for the recognition rate according to the number of picture per person, so we added our experimental results on the graph:



We can see that our results are a little bit under what we should obtain, especially with the Fisherfaces algorithm. The final results are as followed for a base where there is at least two picture of each person, for a total of 5 different persons:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Recognition | No Recognition | Errors |
| Final rates : | 0,764705882 | 0,058823529 | 0,176470588 |

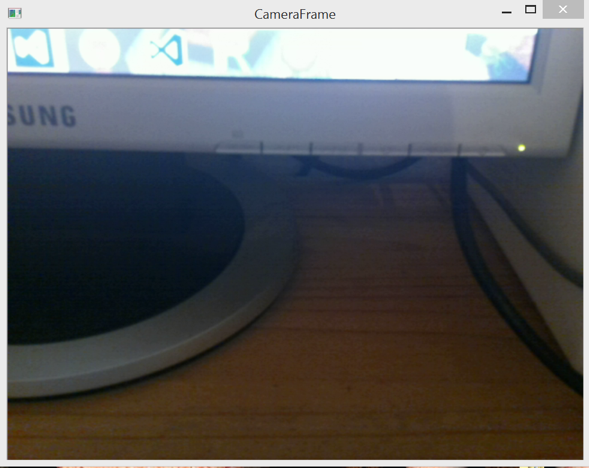
“No Recognition” is when the picture of someone is too far from the other pictures of himself;  “Errors” is when someone is mistaken with someone else. We improved the error rate by putting a threshold confidence empirically, we could optimize the system by changing it.

# Graphical User Machine

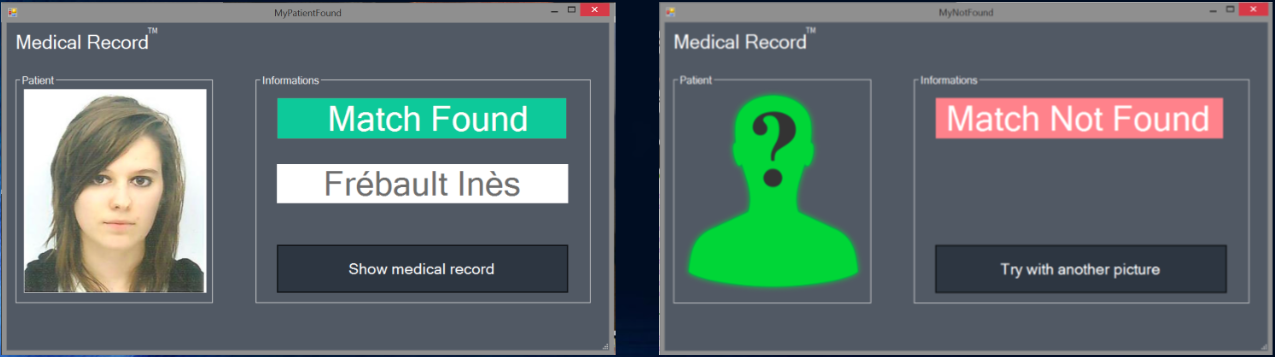
This section contains the GUI we realized for the project using Windows form API included in Visual Studio.

### Launch screen

When we launch the program, we get this first screen, allowing us to take a picture and launch the recognition on it:

When you choose to take a picture, the webcam preview pop up so that you can see what the result would be:

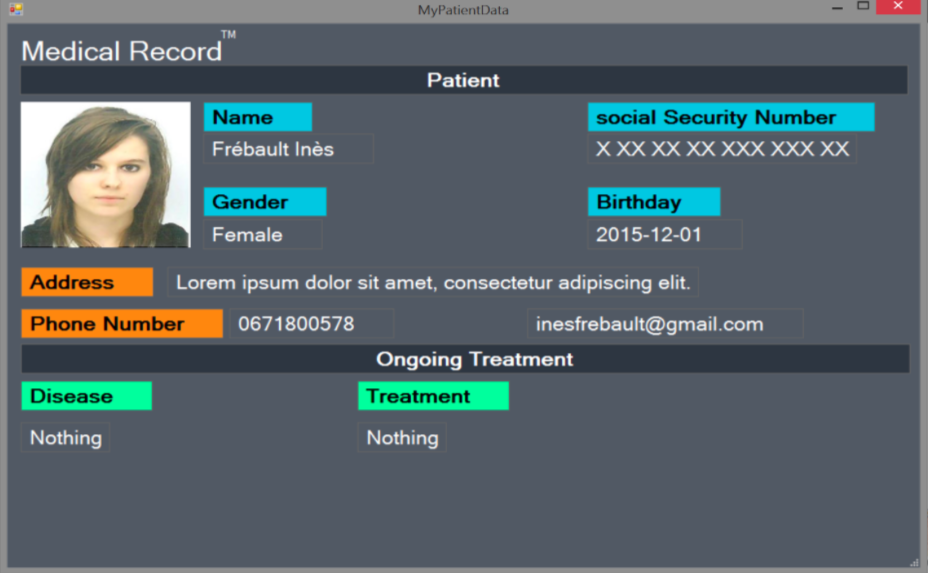
### Recognition screen

After that, we get two different screens according to the result of the face recognition algorithm:

The first screen is the screen indicating the patient identified from the picture we took and offers to display its related information; the second screen indicates that no patient has been recognized from the image and offers to take another picture.

### Medical record screen

After a successful identification, you can choose to see the patient’s information and access the last screen where you have all information contained in the database about this patient:

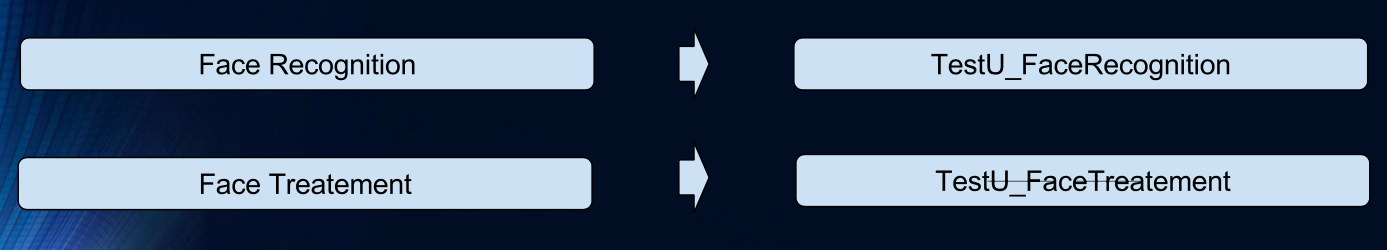


Key information are highlighted with a blue background and the medical history is located at the bottom of the page.

# Non-regression tests

As requested for this program, we made non-regression tests to check that the changes we’ve made between two commits does not change the results we already had in the different functions of the program. We decided to do our non-regression tests with unit tests : we test the results of functions independently and if one of the unit test fails, we have regressed. The advantage with unit tests is that we know which one has failed so we can investigate quickly to find out why.

Our goal was to cover 100% of our program, however we only had the time to cover 50% of our program, that is to say the Business layer functions :



The two unit tests were made on the most important functions of our program : Face Recognition and Face Treatment. We inject static parameters as an input and we check if the output of our function is what we expected.

To activate this tests, you have to de-comment “#define TESTU” in the Main.cpp file to get the output check results of Face Recognition and Face Treatment in the console.

# Conclusion

Unlike the beginning of our project, which was winding because of OpenCV installation on Mac, the realization went well. The major part of the operations used in image processing works properly. Indeed, our application is able to recognize the patient's face and to compare it with those stored in our database to gain access to its personal informations. The implementations of these functions is the culmination of this project but we can imagine some improvements such as, encrypting the database or implement an access to a remote database.