UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Kinect Integrated Biometric System for Face Recognition in Light Varying Environments

Trabajo Fin de Grado

Grado en Ingeniería Informática

Author: Enrique Vicente Garnelo Prediger
Tutor: Manuel Agustí-Melchor

2014/2015

Abstract

Face Recognition is a common Computer Vision problematic nowadays that dates back to the early 1960's. Along those years many issues arose while trying to tackle this task such as light conditions, pose estimation, head rotation, tilt, aging, amongst others.
This project focuses on the development of a computer vision software robust to light variations named Kinect Integrated Face Recognition System (KI-FRS) which takes advantage of the Microsoft Kinect sensor's capabilities and computes biometric features from human faces to later recognize them.
The KI-FRS simplifies common operations for standard Face Recognition Systems by using a user-friendly, intuitive and minimalist Graphical User Interface. Furthermore, its operations are performed with great performance and scalability.
The developed system and its complementary documentation to this project's content is the major output of this dissertation, easing the path to future lines of work as well as other branches of study.

Key words: facial recognition, varying illumination, biometrics, kinect.

Abstract

El Reconocimiento Facial es una problemática muy común hoy en día que se remonta a los tempranos 1960. Durante aquellos tiempos muchos inconvenientes surgieron en el intento de resolver dicha problemática, como condiciones de iluminación, estimación de pose, rotación de la cabeza, inclinación, edad, entre otros.
Este proyecto se encuentra enfocado en el desarrollo de un software de visión por computador robusto a las variaciones de iluminación bajo el nombre de *Kinect Integrated Face Recognition System (KI-FRS)* el cual saca provecho de las capacidades del sensor *Microsoft Kinect* para luego computar características biométricas de rostros humanos en vistas de reconocerlos.
El KI-FRS simplifica operaciones comunes en Sistemas estándar de Reconocimiento Facial mediante el uso de una interfaz gráfica de usuario amigable, intuitiva y minimalista. Más aún, lleva a cabo sus operaciones con gran rendimiento y escalabilidad.
El sistema desarrollado y su documentación complementaria al contenido de este proyecto es el mayor resultado de este estudio, facilitando el camino a futuras líneas de trabajo como así también ramas de estudio.

Palabras clave: reconocimiento facial, iluminación variable, biométrico, kinect.

Acknowledgments

This work is the result of a challenge I proposed myself in the year 2012 when I was first introduced to the Computer Vision world when asked to implement the PCA algorithm for Face Recognition.

It was a great journey since then and while developing the project I've learned lots of new concepts and techniques bringing to life a working application that hopefully could contribute to many others that, like me, are fascinated with this amazing field of study.

I would also like to use this space to thank each and every person in my life that made this project possible starting with Viviana that supports me on a daily basis, Manolo who greatly encouraged me on developing and documenting this project from scratch and my sister Ana Luz who helped me on formatting as well as the redaction and consistency of this document.

Special thanks to the students who kindly offered on helping me to build the Face Database. They are Floris, Andrea, Johan, Estela, Katerina, Javier, Rodrigo, Marc, Andrés, Beatriz, Enrique, Alejandro, Carlos, Anna and José Manuel.

Finally I am conscious that nothing in my career so far would have been possible if it wasn't for my family that led me pursue my dreams and study what I truly enjoy and makes me who I am today.

"All our dreams can come true, if we have the courage to pursue them"
Walt Disney

Contents

List of Tables

List of Figures

Chapter 1

Introduction

"The serial number of a human specimen is the face,
that accidental and unrepeatable combination of features.
It reflects neither character nor soul, nor what we call the self.
The face is only the serial number of a specimen"
Milan Kundera[1]

The human face is probably the best feature that distinguishes one person from another but if we refer to the cited quote by Milan Kundera, is just an identification or serial number of a human being up to the point that in most personal identification documents we carry on our daily basis, our face is present. Some people find hard to forget a face rather than a name and this happens thanks to specialized regions of our brain such as the fusiform area, intimately related to our visual system, our main source of information of our surrounding environment. Unfortunately, when trying to recreate this task artificially, complexity arises to an unreachable level. Though, and in the same way we identify or recognize faces, recognition could be focused on specific face features such as the eyes, lips, nose, eyebrows, etc. In other words, the Biometric features of a human face.

This project is dedicated to the challenging thus interesting task, the Face Recognition task, clearly a Biometric assignment were perception, information storage and pattern matching come to place. Specializing on the perception device, in this case a depth sensor and color camera, we will extract information from the environment to later process the extracted data with a computer software.

The project is structured as follows: in this Chapter we introduce the state of the art of this work by briefly referring to some background on the topic in Section 1.1 as a base to later in Sections 1.1 and 1.1.1 define our problem approach and objectives. Chapter 2 is dedicated to a more detailed yet concise literature review of the discussed topics. Chapter 3 focuses on the problem solution, a system implementation, as a proposal for achieving the established goals. This is the main Chapter of the work where implemented components are introduced and development decisions are justified and explained objectively without falling into code explanation. Chapter 4 is dedicated to experiment over the mentioned components. The dissertation concludes in Chapter 5 were conclusions are drawn from the previous sections, goal achievements are discussed and future lines of work are proposed.

The recommended reading for this project is the numbered order, though if you are familiar with the topics covered in Chapter 2, you can quickly start your reading in Chapter 3.

1.1 Background, problem approach & boundaries

If we refer to the Face Recognition task from its origins in relation to computers we must recall the pioneers Woody Bledsoe, Helen Chan Wolf, and Charles Bisson and their study from 1964 and 1965 where they tried to tackle this problem with a large database of mugshot images[6]. Their study lead to what are nowadays the topmost difficulties in the discipline:

1. **Head rotation and tilt:** pose estimation is a major line of study in order to align faces and normalize databases.

2. **Lighting intensity and angle:** light types also affect the resulting image if texture images are used. Appearances could be drastically changed due to different light conditions.

3. **Facial expressions:** for the main reason that, depending the expression, face features could be extremely deformed.

4. **Other factors:** such as aging, beard, glasses, etc.

Since then and for the following years many other studies were performed to grasp the enumerated difficulties with a great variety of results, experimentations and methods. In 2006 as a result of the Face Recognition Grand Challenge[4] it became a fact that new algorithms are 10 times more accurate than Face Recognition algorithms of 2002 and 100 times more accurate than those of 1995.[3] Viewed from Moore's law perspective, the matching accuracy will be increased by one-half every two years.[5] The improvements come from faster processing units, efficient algorithms, better image quality and information such as 3D data.

The described scenario leads to the imminent decision of establishing boundaries and focusing on certain essential topics. Recalling the enumerated difficulties for the Face Recognition task in Section 1.1 this project focuses on improving matching rates under lighting variations. The rest of the problematics will be out of scope. Nevertheless they will be considered as a limiting factor.We will tackle this problematic based on the idea that by using a depth sensor we will be able to recognize facial features no matter the light conditions as depth data remains invariant in those cases. For this task a system will be developed under the name of Kinect Integrated Face Recognition System (KI-FRS)[7] as we will be using the Microsoft Kinect depth sensor capabilities to obtain such data.

1.1.1 Objectives

It is expected that the KI-FRS – the developed system – accomplishes the following objectives:

- **Robustness to light variations:** being the main pillar for this work, aims to a high performance on Facial Recognition under light variations.
- **Portability:** it must be cross-platform for its immediate execution in various environments, a highly recommendable quality attribute for any software development.
- **Versatility:** it will be composed of exchangeable modules providing different functional characteristics to the System while also contributing to code maintenance.

By the end of this work, this characteristics and system performance will be evaluated to decide whether this items were suitably fulfilled.

Figure 1.1 shows the basic block diagram used as a reference for developing the KI-FRS. Later in this dissertation we will cover a detailed description of the actions taken in order to fulfill the defined objectives with the help of well-known tools in the field such as OpenCV and Python.

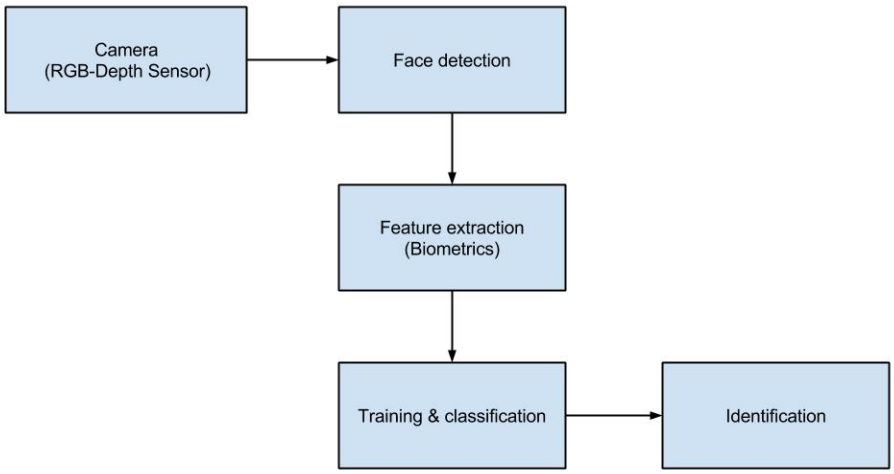

Figure 1.1: Basic KI-FRS block diagram

Before starting the development it became essential to research about similar case studies where this approach was used – covered in Chapter 2.

Chapter 2

Literature review

The set of established objectives bring to the necessity of reviewing ongoing and historical investigation related to the study topic in order to determine which path should be followed, what has been already analyzed and to avoid committing the same mistakes or going through misleading lines of work.

In Section 2.1 of this Chapter we will introduce the concept of Biometrics, what are their pillars and how do they relate to this work. Secondly the Kinect device will be presented in Section 2.2 to afterwards explain its capabilities and limitations for depth imaging. Regarding those images in Section 2.3 there is a brief review on the topic. Finally in Section 2.4 we will inspect related work using the Kinect device for Face Recognition under light varying environments, what have already been done in this field and what methodology and line of work will be reused and adapted for our development.

After reading this Chapter, the reader will be able to understand the concept of Biometrics and its implication in our society nowadays, the advantages of using the Kinect device and its depth data and a general idea of Face Recognition approaches for challenging environments, specially when light conditions are not ideal.

2.1 Biometrics

One of the main topics of this work is **Biometrics** so it is essential for our purposes to clearly understand what this concept means, and in technical terms, how it works. As it is explained in [12] the term Biometrics refers to an automated technique for measuring physical characteristics or personal trait of an individual and comparing them with a database in order to **recognize** it. Needless to say, those features must be truly unique to differentiate one individual from another. Those defying characteristics are called **discriminant features** or **main features**.

Nowadays we have biometric identification systems accessible for everyone like the fingerprint sensor attached to Apple devices [13]. But there are also others not so handy such as retinal or iris scan. Those three identification mechanisms that we have mentioned are based on a group of biometric features that up to date are considered to be unique for every individual. But there are many others such as:

- Facial features and thermal emissions,
- Hand geometry,
- Wrist/hand veins

and many others. It is more than clear how useful biometrics are for preserving critical information related to an individual but at the same time they could be dangerous if managed with bad intentions. A more clarifier debate on this issue could be found at [14]. The cited article concludes with an interesting conclusion about what it was projected for Biometrics in those days. By reviewing the article in perspective we are able to comprehend the concept in legal and privacy terms and relate it to our current problems nowadays. Identifying an individual's biometric features may also lead to some more personal information such as Facebook profile, Twitter account, behavioral actions, etc. [16]

Letting apart the concept itself and focusing on the application perspective, The National Science and Technology Council [15] defines the essential components for a **Biometric System** as follows:

- **Sensor:** collects data and converts it into digital format to be interpreted by a computer system.
- **Signal processing algorithms:** perform quality control activities and develop the Biometric template, in other words, extracts the discriminant features of the sensed data.
- **Data storage:** which keeps the information that would be later compared to other discriminant features.
- **Matching algorithm:** compares the new features with the stored ones.
- **Decision process:** uses the results from the matching component to make a decision. This process could be automated, human-assisted or hybrid.

Biometrics comprises a large field of study that leads to certain controversy regarding legality and policies but on the other hand brings us the possibility of absolute privacy when used to protect by encrypting critical data.

In the case of this work we focus our study on Biometric Facial Features for individuals based on the idea that every human being has distinctive physiognomy and in the particular case of a reduced set of faces they happen to be unique. Following the explained framework [15] we later in Section 3 proceed to the analysis and design of its components.

2.2 Microsoft Kinect device

Depth images became truly popular after the launch of the Microsoft Kinect® in 2010. In the following years other devices such as the Asus Xtion PRO or the DUO3D –amongst others[42][43]– where released to the public, allowing developers to experiment with depth map images and cloud point data. Not so long after in 2011 Microsoft announced the launch of their official SDK to attract developer communities towards the device. Since then many applications and programmers dedicated several studies using the Kinect functionalities. The device is made out of the following components:[41]

- **IR Projector / Emmiter:** emits a grid of IR light to the facing environment. Any object or surface in front of this grid reflects the light back to the IR Sensor.
- **IR Sensor:** receives the pattern for the projected objects from the IR Projector and decodes data to depth information. This information could be later used by a computer vision application for further processing.
- **RGB Camera:** with a 1280x960 resolution makes possible to capture color images.

In addition to these components we have the tilt motor to modify the sensor's point of view and a microphone array to work with audio data. Because of its functionalities and low

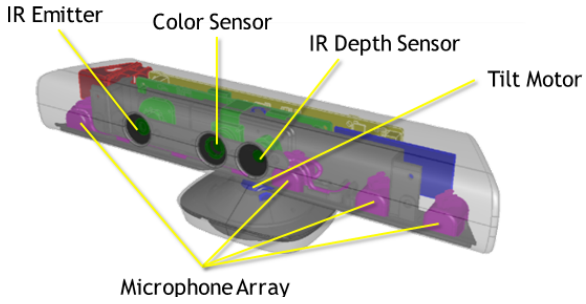

Figure 2.1: Microsoft Kinect components[41]

cost the Kinect has been the most used device of its kind. Now lets take a look to one of its main functionalities, the depth image sensing.

2.3 Depth images

The Kinect device will allow us to get depth data that describes the sensed environment for later processing and feature extraction. The term "depth image" is quite vague to describe the resulting data pulled off the depth camera. There are a few depth-related channels that need to be clarified for further understanding, listed as follows:

- **Depth map:** is a gray scale image where each pixel value is the estimated distance from the camera to the sensed surface. Getting this kind of image is not trivial as the Kinect device works with raw data that needs to be preprocessed. This problem will be discussed and exposed later in Section 2.3.1.
- **Point cloud map:** a color image in which each color corresponds to a spatial dimension (x,y, or z). In the sample shown in Figure 2.2 the R represents the z value, the depth; simultaneously the G is the y value while B is the x.
- **Disparity map:** being also a gray scale image, the value stored in each pixel is the **stereo disparity**. It is a useful tool for stereo imaging and 3D space reconstruction.
- **Valid depth mask:** gives the trustiness for a certain pixel in the image. This happens usually due to uncovered regions by the infrared sensor. In Section 2.3.2 we will cover the problems this issue brings.

Figure 2.2: Depth image types: Point cloud (Left), Disparity map (Center) and Depth Mask (Right) [17]

2.3.1 Depth normalization

Depth data values from the raw depth data are not always the distance nor intensity values. In the case of the Kinect, it is a 11-bit disparity value, in other words, is a number that ranges from 0 up to 2047. Using these values is useless and may come with other disadvantages as it is shown in [20], a coped problem in Section 2.3.2.
Following the empirical methods explained at [18] it is possible to transform the raw values into meters:

$$distance = 0.1236 * \tan(\frac{rawDisparity}{2842.5} + 1.1863) \tag{2.1}$$

This type of estimation eases the process of distance related operations such as background extraction and measure calculation.

Once the distance is calculated, we have to clearly understand that it represents the distance between the horizontal level were the device is located and the sensed object. In other words is not the distance of the projection rays such as the ones shown in Figure 2.3.

2.3.2 Ghost effect

One of the biggest issues commonly exposed in depth mask images is what is called "the ghost effect". As we can see in Figure 2.2 for the depth mask there is a black shape around the person's figure. The same could be seen in Figure 2.4 but in white, as it is not a mask image but a depth map. This effect is caused by unreached areas of the infrared sensor just like a projected shadow from an object over a certain surface.

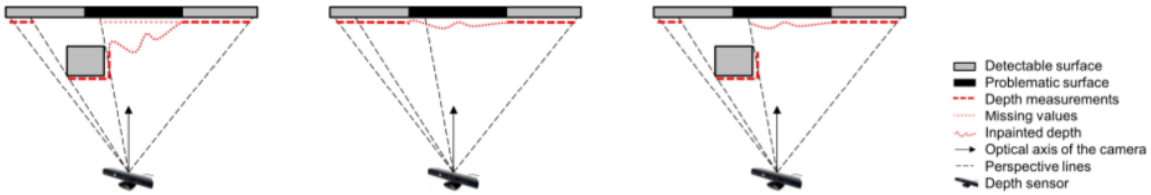

Figure 2.3: Depth ghosting effect explained [21]

To clarify this issue lets take a look to Figure 2.3 there are multiple situations were sensed data could be potentially, completely inaccurate or even absent. In multiple articles [22] [29] methods for enhancing and improvement of data contained in the image are proposed in order to build a database with the less useless, redundant or inaccurate data as possible.

Figure 2.4: Depth ghosting

There is also an important matter to be taken into account that could usually be confused with the ghost effect. In this case, it is related to the constructive specification of the device. For the Microsoft Kinect the technical specifications by Primesense are as follows[41]:

- **Field of view (Horizontal, Vertical, Diagonal):** 58 H, 45 V, 70 D.

- **Spatial x/y resolution (2m distance from sensor):** 3mm.
- **Depth z resolution (2m distance from sensor):** 1cm.
- **Operation range:** 0.8m-3.5mm

Objects located either near or far from the device's operational range will not be sensed hence will not be represented in the resulting image. The shown specifications also lead to another problem related to the field of view of the depth sensor in comparison to the RGB camera, a major topic of discussion in Section 2.3.3.

2.3.3 Camera calibration

The RGB camera has an slightly larger angle of view than the depth camera so the raw output from both images do not match at all [18].

Figure 2.5: Lack of correlation between dept and RGB images

Figure 2.5 clearly shows by overlapping an RGB image with its same depth map –and modifying its alpha values– that there is a difference not only in the position but also in scale between both of them. This is a common issue while trying to match pixels from both images and to analyze coherence between the texture and depth data. There are standard techniques provided by OpenCV that will be later shown in Section 3.5 to deal with this problem.

2.4 Face Recognition

To end this Chapter, lets focus on the project's major topic, Face Recognition[25]. Rather than explaining Face Recognition methods or techniques we will analyze the background articles that address this topic related to the recognition process using depth images.

At [26] it is shown that Face Recognition for depth images performs worse than with texture pictures – also known as RGB images. However, in [27] it is proven that texture images are highly sensible to illumination and pose changes and for that reason they are not significantly helpful for the recognition process. They need to be preprocessed in order to get higher accuracy results at expense of computing time as in [28] where local features are extracted from the original sample to be later used for recognition leading to high accuracy results. On the other hand in [27] to avoid preprocessing or local features extraction, a mixed algorithm is proposed employing texture and depth images proving an overall recognition rate of 88.96 percent.
As a result from this reviewing, textured pictures will not be used for Recognition but for light estimation to decide whether to apply preprocessing on the used data or not, Section 2.4.1 and Section 3.6 are dedicated to explain this task. Based on the described sources I decided to don't extract local features due to computational time and counter this issue by using depth maps and enhancements on the original input images.

Two characteristic, well known and popular Face Recognition algorithms with high performance without using local features that will be used are[32]:

- **Eigenfaces:** first introduced in 1991 [30] derives from Karhunen-Love's transformation [31] is a fast method for face recognition also known as Principal Component Analysis or PCA that reduces data dimensionality by projecting it to a smaller space while preserving linear separability. Though this boundary could be sensitively changed due to light variations in the used images.

- **Fisherfaces:** this method works similarly to the Eigenfaces with the difference that when building the projection matrix it tries to maximize the within-class separation in the new resulting space. This method is commonly known as LDA.

2.4.1 Light variations

Light conditions are usually an unpleasant problem to deal with when applying Face Recognition algorithms [38] and that is the main reason RGB images are not commonly used for this purposes. As covered in the cited article, many approaches have been used over the last few years to tackle this inconvenient [39] having good results in terms of accuracy in exchange of computational complexity.
In this project we develop a simple and fast computing method to enhance input images and improve recognition rate [Section 3.6].

2.4.2 Face databases

For standardization sake there are several Face Recognition databases available dedicated to different tasks within the topic. Obtained results of using specific algorithms and/or enhancement methods could be easily compared and benchmarked as a proof of success or failure.

A general list and download links is shown in [33] but just a few of them completely fit the purposes of our work such as [29] while some others use high resolution 3D sensors being out of scope for our use. In Section 3.8 we explain how we built our own database to test the KI-FRS functionalities.

Chapter 3

Kinect Integrated Face Recognition System (KI-FRS)

This Chapter is focused on the system implementation that puts into practice the topics covered in Section 2. This is an ongoing Open Source project distributed under the Apache License that could be found at [7].
Developed from scratch, serves as a basic framework to test Face Recognition algorithms, image preprocessing and interface with depth sensors. It could also be used to build your own Kinect Database with normalized data to be later used for Face Recognition as well as for Face Detection.

In the first part of this Chapter we discuss about the decisions made towards designing and developing the KI-FRS and how to set up your working environment to run it. The current system implementation is later explained in detail for each implemented module. Very few source code will be shown as the idea is to explain, in a general manner, what is implemented without getting lost into the statements written.
Finally, two working processes, the database creation with preprocessing and the On-line Face Recognition are presented.

3.1 Functionalities

In the previous Sections 1.1 and 1.1.1 we have mentioned the concepts and objectives that were put into practice in the application. In Section 2.1 by defining what a Biometric System is, we have a clue about the needed functionalities which sum up to:

- **Data sampling:** this includes directory and file management. Structure and name conventions must be established in order to build indexes and references for the data that will be needed by the system. There must be a set of sampled images for training as well as for testing to check Face Recognition model's performance.
- **Image preprocessing:** the pulled data must be preprocessed in order to be used by the Recognition Algorithm. It may occur that some samples are not suitable for training. In that case, the Face Recognition Model and the application should be capable of determining whether to use it or not.
- **Face recognition:** the main purpose of the application is to do Face Recognition from a given input image. Before this, it is also necessary to train different models based on the possibility that the data provides inherent meaning, choose between depth or black and white images – or even both of them – to improve recognition performance as much as possible. The application must be able to perform with considerably high accuracy on recognizing faces under light varying environments.
- **Auxiliary functionalities:** the application must have a Graphic User Interface and detection assistance to generate suitable sampling images, discard or process them for their later usage in the Face Recognition Model.

3.2 Design & development decisions

With the defined functionalities in Section 3.1 we are capable of modeling the application Conceptual Model shown in Figure 3.1.

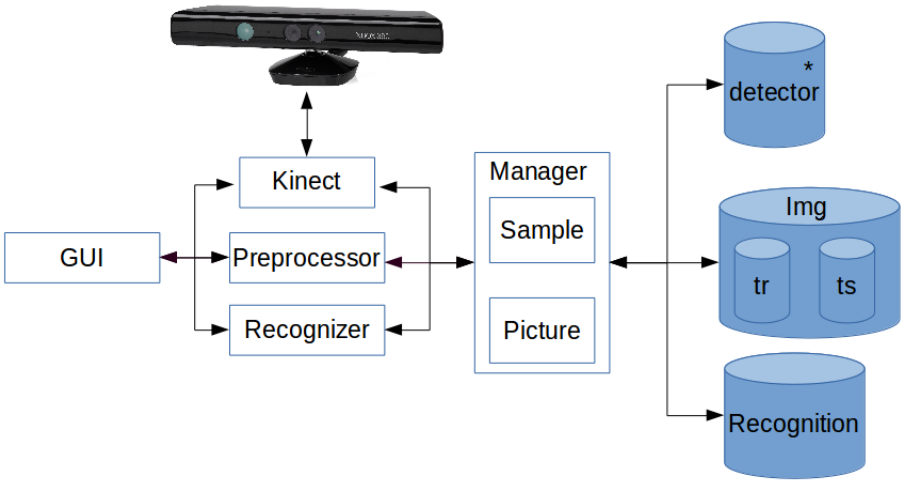

Figure 3.1: KI-FRS conceptual model

The main blocks represent constructive modules of the system being each one of them independent from the rest. The user interacts directly to the GUI which manages the other packages. The Conceptual Model of the system shown in Figure 3.1 represents the following constructive blocks:

1. **Kinect device:** being the chosen depth sensor, is the only specific piece of hardware needed to run the application – besides a computer.

2. **Packages:**

   - **Kinect:** [**Section 3.5**]is the main interface to the device, managing the depth and image data pulled out from the hardware.
   - **Preprocessor:** [**Section 3.6**]module dedicated to image processing, this covers depth image enhancement and light estimation amongst others.
   - **Recognizer:** [**Section 3.7**]this module uses multiple trained Face Recognition models to perform the task over a given sample – or group of them – providing an output.
   - **Manager:** [**Section 3.4**] is composed of two different submodules:
     - **Sample:** provides the functionality of adding new data to the database. This process is usually called as "sampling".
     - **Picture:** provides the functionality of modifying the stored image data.

3. **Directories:** where all the application data is stored. The system uses the following main directories:

- **Detector:** trained models for Face Detection. There is a mention over this block in particular mainly because the entire project is based on face recognition and not detection.
- **Img:** for test and training samples.
- **Recognition:** trained models for face recognition.

After defining the conceptual model, some other decisions had been made towards the project development.

To start developing applications using the Microsoft Kinect device the possibilities for communication are narrowed into two main options:

- Kinect for Windows SDK [40]
- OpenKinect [56]

Libraries offered by PrimeSense before its purchase by Apple are currently deprecated and without or very vague documentation.
As mentioned in Section 1.1.1, one of our main goals is to develop a cross-platform application with Open Source code to promote future enhancements and functionalities. This leads us to the OpenKinect library, distributed under an Apache20 or optional GPL2 license. This library is so far the best choice for multi-platform and language independent development. **libfreenect** is the main communication library with the device, developed in C with wrappers for multiple major programming languages such as Java, Python and .NET languages.
A useful tool compatible with the library is the **Fakenet**, a Microsoft Kinect simulator to test the code while not having the device physically, so it is not necessary to own a Microsoft Kinect device to run the current development.

Regarding the programming language to be used for developing the decision was based on the idea of applying the **Rapid Application Development (RAD)** methodology[57] to get immediate results and work over them incrementally. The decision lies on a language capable of managing abstractions – such as objects or inheritance – providing code maintainability by being readable and clear enough to continue the development at any and every level. Time productivity is highly valuable in this case as well as compatibility with the **libfreenect** library. For that reason Python has been chosen over any other possibilities because it completely fits in with the described requirements. Needless to go into further detail about its ability to run on multiple platforms [8].

Recalling the earlier explanation in Section 2.1, the imminent requirement for signal processing algorithms shows up. And in this case I made the decision of using OpenCV [50] as it is released under a BSD license, it is free for any use and comes with interfaces for multiple programming languages being also platform independent – up to the point of being used for iOS or Android – and supports most of the required functionality for this project. To work in conjunction with OpenCV we will use numpy[11] to handle matrix operations

and algebraic calculations and matplotlib[10] for displaying analytic results and plotting.

Finally the idea of using a graphical user interface is based on basic user interaction with the application functionalities without difficulties and in an intuitive manner. In this case, there are numerous options based on the library to be used. I decided to use **wx** because its basic usage and WYSIWYG [1] tools. In Figure 3.2 we show an example of wxGlade[9], a GUI builder for C++, Lisp, Perl, and Python that uses the mentioned library.

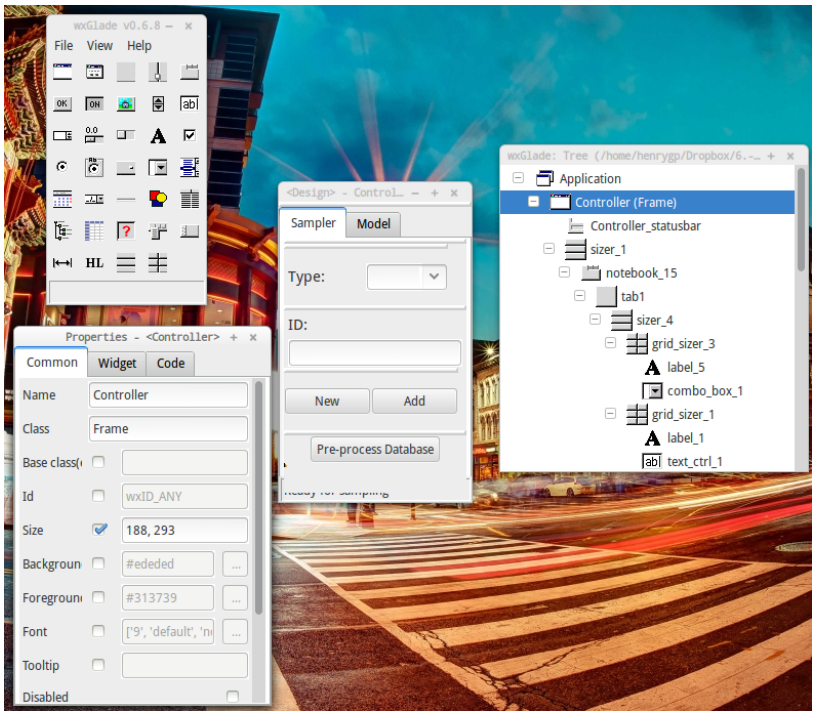

Figure 3.2: wxGlade GUI builder

[1] Acronym for "What you see is what you get". Commonly on referring to intuitive and easy to use development tools for graphics.

3.3 Setting up the application

In order to make the application work in your environment is necessary to have installed the following main utilities:

1. **OpenCV:** for all implemented algorithms and data processing.
2. **OpenKinect:** this includes libfreenect-dev as well as python-freenect, the python wrapper for the mentioned library.
3. **Python development environment and libraries:** numpy, matplotlib and cython are essential.

Other libraries and packages are also needed for the usage of the mentioned components. For further detail just take a sneak peek to the **setup.sh** file located at the root of the project's repository[7]. This file provides with the commands needed to install the enumerated utilities and libraries:

```
#!/bin/bash
echo "Hello!, I will just execute a couple of commands to make sure the system
    will work"

echo "Installing dependencies"
sudo apt-get install git-core cmake freeglut3-dev pkg-config build-essential
    libxmu-dev libxi-dev libusb-1.0-0-dev

echo "Installing OpenCV"
sudo apt-get install python-opencv

echo "Installing libfreenect"
sudo apt-get install libfreenect-dev freenect python-freenect
sudo apt-get install cython
sudo apt-get install python-dev
sudo apt-get install python-numpy
sudo apt-get install python-matplotlib
```

To run the KI-FRS in other environments such as OS X or Windows, please refer to the following citation for detailed step-by-step installation process [47] [48] [49]

3.4 Data management - Manager module

The manager module handles all the data being used by the application. This goes from detection and recognition models to raw images and indexes. In this case two different classes were implemented to split functionality:

1. **Sample Manager:** focuses on maintaining the **/data/img/** directory, stores new samples for training & testing and maintains a small index for further reference during recognition.

2. **Picture Manager:** this class wraps functionalities focused on managing the raw data stored by the Sample Manager. It communicates directly with the Preprocessor and Recognizer.

Even though these classes are independent, during an on-line recognition, their functionalities overlap in order to manage a real time Face Recognition result. More on this will be explained in its corresponding use case in Section 3.9.

An auxiliary class called Index is also implemented for the main purpose of tracking the sampled individuals and provide a "more human" response to the recognition prediction rather than a number that represents the person's position in the database.
All the referred code is defined and documented in **file_manager.py** package in the project's **src** directory [7].

3.4.1 Data sample directory

The data directory where the datasets are stored is the most important part of the project composing the working database. A basic scaffold to structure this data is proposed in order to program an efficient and well organized development. Figure 3.3 shows the directory tree for the image dataset, composed of the following subdirectories:

- **tr:** contains the training data organized in folders in numerical order. That value is mapped to a name using the index where each folder corresponds to a sampling individual.

- **ts:** contains the testing data organized in the same manner as the training directory with numbers exactly matching the ones used for the same individuals.

The stored data samples for every individual are also ordered numerically and used to group the sample files which are:

- **Images:** all image sizes are 192 x 256 pixels.
  - **Black & White (_bw):** gray scale image.
  - **RGB (_rgb):** texture color image.
  - **Depth map (_depth):** gray scale image representing the depth map.

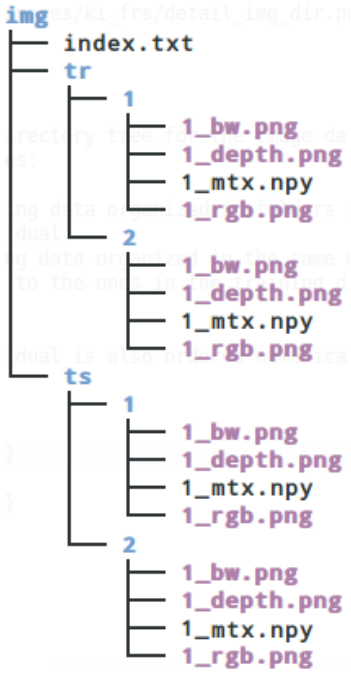

Figure 3.3: Detailed img directory

- **Matrix file(_mtx):** with unprocessed data, stores the raw 11-bit disparity values from the Kinect device. Its file extension is a numpy matrix that could be easily loaded and integrated in a Python program.

Images shown in Figure 3.4 expose the substantial differences between the mentioned files. Note that the black and white image could have not been stored as it may be obtained from the RGB, though for performance reasons we sacrifice disk space to gain processing time. This will become clearer after explaining how the pre-processing module works in Section 3.6.

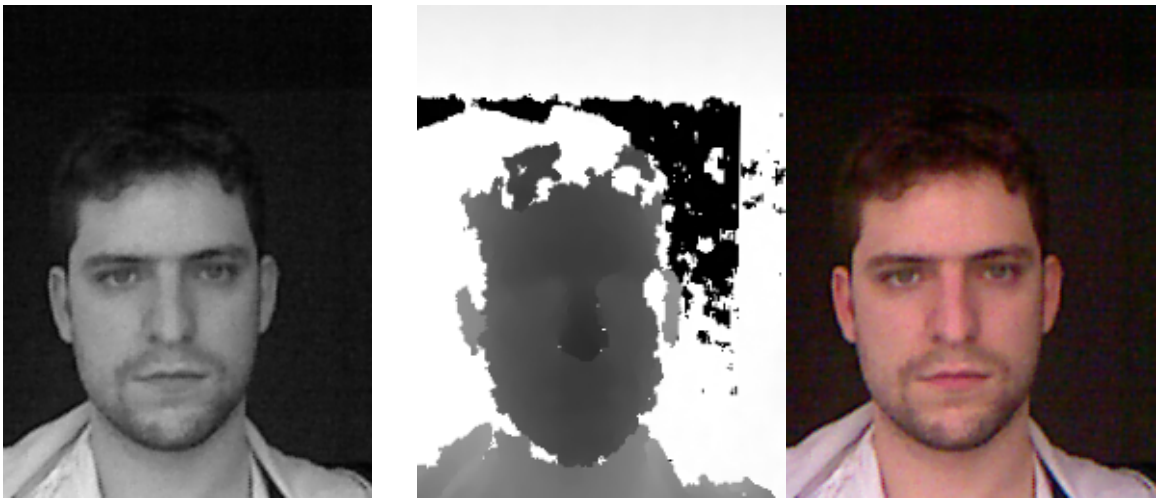

Figure 3.4: Examples of BW (Left), Depth (Center) and RGB (Right) images in **img** directory

3.5 Device Interface - Kinect module

In our project directory the Kinect interface is implemented in the file named **kinect.py** [7] under the **src** directory. The main idea of this package is to implement basic data input from the Kinect device and also from the user by managing keyboard and GUI events.This module works with the Manager package for sampling and on-line recognition.

The development decision of using the Open Kinect library **libfreenect** for this matter limits the development to the implemented functionality. Under the installed directory for the library in **wrappers/python** we can find a series of examples written in Python to communicate with the device plus the wrapper itself, written in Cython named **freenect.pyx** where library functions are specified and documented.

An important fact to be remarked is the lack of well organized documentation for the Python wrapper, up to date there is no official nor unofficial wiki for consulting. Nevertheless, by reading the comments written in the wrapper implementation and the native library documentation in C plus the given examples you can deduct how functionality is being handled.
Data is pulled from the device in around 30 FPS. This includes texture images and depth information. The **libfreenect** handles this process with a running loop and event handlers and also a body block that executes general purpose code. The structure is defined as follows:

```
"""RGB event handler"""
def rgb_handler(dev, data, timestamp):
   """Block to work with RGB data"""

"""Depth event handler"""
def depth_handler(dev, data, timestamp):
   """Block to work with depth data"""

"""Body block, executed at every cycle"""
def body(dev,ctx):
   """General purpose code"""

"""Run loop, define the handlers"""
freenect.runloop(video=rgb_handler,depth=depth_handler,body=body)
```

There is an important issue to point out regarding data's format. The library used for data processing as stated in Section 3.2 is OpenCV [50] that represents RGB images with 3 bytes per pixel inside a matrix structure in inverted order, BGR. On the other hand the matrix denoted by *data* in the video handler shown above is also a matrix structure with 3 bytes per pixel but in RGB order, thats why mixing channels is a recurrent problem. As for the depth image, the hardware gives us a matrix with 2 bytes per pixel representation. This data could be later transformed for different depth images as explained in Section 2.3. The described transformations are implemented in the file **frame_convert.pyx** under the python sample directory.

Something already covered in Section 2.3.3 is the problematic that arises while matching pixels from the color and depth images. This inconvenient requires an standard calibration process that OpenCV already provides in a library function that calculates a transformation matrix for later operations over the depth image. The function to calculate that matrix is called *getOptimalNewCameraMatrix* and requires the following main parameters:

1. **cameraMatrix (K):** is the input camera matrix, in this case the one from the Kinect device.
2. **distCoeffs (d):** this are the distortion coefficients.
3. **imageSize (w,h):** so the resulting matrix matches with the image to be transformed.
4. **Complementary parameters:** for our usage are irrelevant.

The output matrix from this function, named as *newcameraMatrix* has to be later used to transform the depth image as shown below:

```
depth_align = cv.undistort(depth_img, K, d, None, newcameraMatrix)
```

Figure 2.5 clearly shows the initial difference in correspondence between pixels from each image. Figure 3.5 shows the accuracy of the correlation process.

Figure 3.5: Correleation between RGB and depth after calibration

So far everything is clear but, how do we get the K and d values to get the transformation matrix? Fortunately at ROS laboratories[19] they have already calculated this values for the Kinect device and not only for transforming the depth image but the RGB to make the transformation the other way. In our case we downloaded the *calibration_depth.yaml* and based on its contents we are able to calculate the transformation matrix the following way:

```
K = np.array([[385.58130632872212, 0, 371.50000000000000], [0,
    385.58130632872212, 236.50000000000000], [0, 0, 1]])
d = np.array([-0.27057628187805571, 0.10522881965331317, 0, 0, 0])
newcamera, roi = cv.getOptimalNewCameraMatrix(K, d, (640,480), 0)
```

3.6 Image normalization - Preprocessor module

This module handles all the heavy processing over the samples introduced in Section 3.4 by modifying the original data as a previous step for recognition model to be trained.

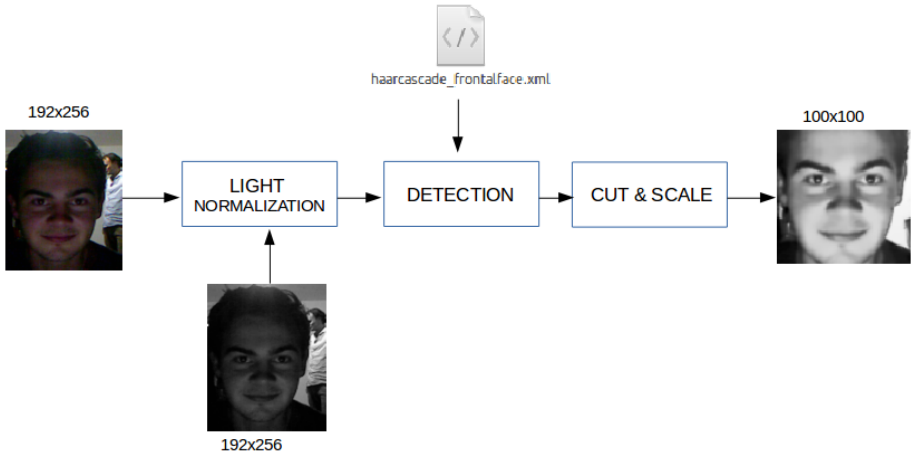

Figure 3.6: Black and white image preprocessing

In Figure 3.6 we show the preprocessing flowchart corresponding to black and white image normalization while Figure 3.7 shows the process for depth normalization that involves similar steps and discards the light normalization process.

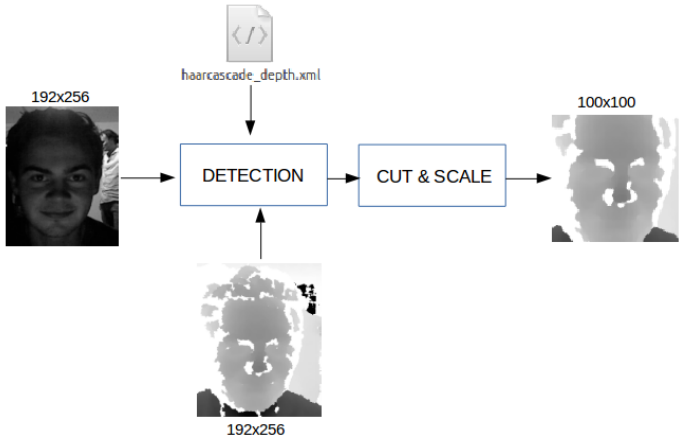

Figure 3.7: Depth image preprocessing

In both cases the preprocessing input are 192x256 pixels sized images with the output being 100x100 pixels ready to be used by the Recognizer either for training, testing or on-line recognition. The following sections are dedicated to explain in detail each block.

3.6.1 Light normalization

One of the objectives for this system [Section 1.1.1] is the capability of handling images from light varying environments. We have already mentioned in Section 2.4.1 RGB images tend to be misleading for Face Recognition algorithms as they are sensible to light variations. Though and mainly because of that reason we need an RGB or texture input to estimate light conditions.

The light normalization process involves two actions:

- **Light estimation:** this first step is the most important as it discards the image from being processed. To do this, the RGB image is transformed to a different color space, the HSV. In that color space components from the V dimension, the "value", represent for each pixel how bright it is. High intensity values will represent high lighting conditions and the opposite for low values. Based on that premise we empirically and flexibly define upper and lower thresholds for the mean pixel value over the V dimension.

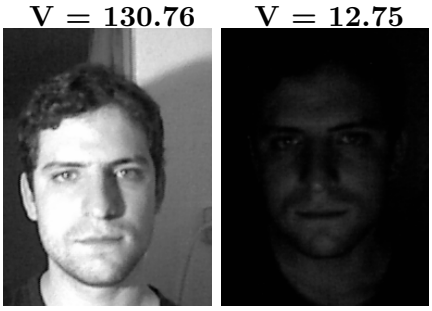

Figure 3.8: Example of V measurement on images

Figure 3.8 shows the V component measurements for a highly illuminated and a darker image of the same person.

- **Image processing:** after analyzing the sample luminosity, if the measurement tells us that is too dark or too bright the image is processed as Figure 3.9 shows. Below we explain the involved steps:

1. **Equalize histogram[51]:** this operation stretches the intensity range from the input image and as a result we get a clear improvement in contrast. In OpenCV the process is resumed in one function:

```
out_img = cv.equalizeHist(in_img)
```

2. **Gamma correction[52]:** by applying a simple power operation over pixel values, we improve image's luminosity.

```
in_img = in_img/255.0
out_img = cv.pow(img,0.8)
out_img = np.uint8(out_img*255)that
```

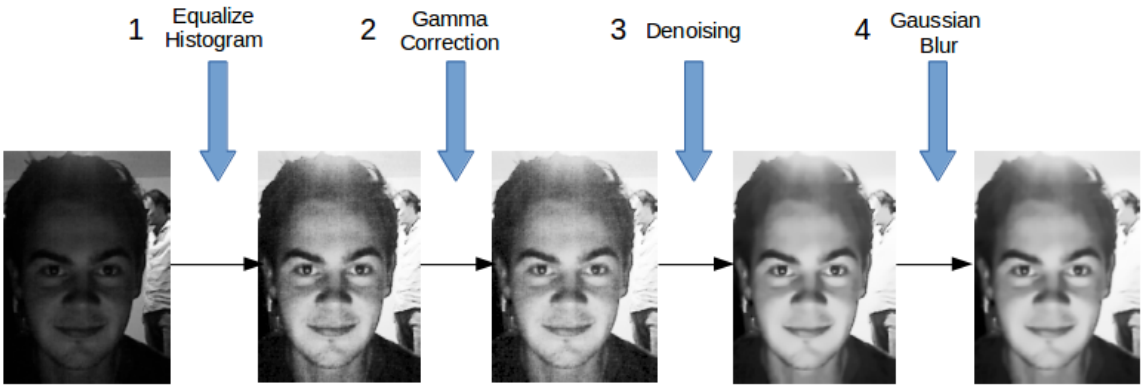

Figure 3.9: Light normalization process

Pixel values from the input image are limited to a range between 0 and 1 and they are later powered by a value within the same range. The lower the value the brighter the output image will be; in this case the used value is 0.8 and as we can see in Figure 3.9 pixels around the forehead brighten up while those from the mouth area that are almost bright enough remain nearly the same.

3. **Denoising[54]:** the resulting image from the previous step tends to be noisy so it needs to be smoothed to get a more natural result. The chosen method is a non-local means denoising that results in a smooth and sometimes caricaturist image.

```
out_img =
    cv.fastNlMeansDenoising(in_img,None,h,templateWindowSize,searchWindowSize)
```

Function arguments are:

   - **h:** is the filter strength so higher values remove noise better but being too high could delete image details being dangerous for face recognition. We use the recommended value of 10 for this parameter.
   - **templateWindowSize:** its value must be odd, the recommended value is 7.
   - **searchWindowSize:** also an odd value, the recommended is 21. This window's size determines how fast the algorithm will work as it considers neighbor pixels inside the defined area.

4. **Gaussian Blur[53]:** the last step is a blurring process. The resulting image from the last step goes through a light Gaussian filter to smooth pixel values and blur some uniform areas. Edges and face features will be preserved and enhanced, being clearly distinguishable.

```
out_img = cv.GaussianBlur(in_img,(3,3),0)
```

The Gaussian filter's window size is 3x3 with an standard deviation of 0.

The output image of this whole process is a normalized image in lighting and potentially suitable for the Face Detector.

3.6.2 Detection

Even though this project is not based on Face Detection, this functionality is useful for cutting the sample images automatically and provide "cleaner" data to the Face Recognition Model. As shown in Figures 3.6 and 3.7 the detection process corresponds to both BW and depth images so a special detector must be used for each case.

BW detection

In the OpenCV library under the **samples** directory for the Python programming language [55] inside the file named **facedetect.py** there is an example of face detection using a trained model. This model has been trained with 5000 frontal face samples with slight deviations as well as varying light conditions. Detection performance proves to be quite precise and robust and even useful for multiple faces as shown in Figure 3.10. In order to

Figure 3.10: OpenCV trained Detector examples

use the detection algorithm in OpenCV first the model needs to be loaded into the program.

```
faceCascade = cv.CascadeClassifier("detection_model.xml")
```

The variable denoted as *faceCascade* stores the model and the parameters from the detection model defined in the .xml file.
The next step is to use the model to detect faces in images such as the ones shown in Figure 3.10.

```
    faces = faceCascade.detectMultiScale(in_img, scaleFactor=1.1,minNeighbors=5,
        minSize=(30, 30),)
```

Given an input image *in_img* – that must be a gray scale image – you need to specify the scale factor to compensate faces that are closer or distant to the camera. The algorithm

works moving a window of the specified minimum size and exploring possible matches. The parameter *minNeighbors* defines the number of objects detected near the current one before declaring a face as found. The result is stored in the variable *faces* that is simply a vector where each component stores the (x, y) value of the window corner and its width and height so drawing the detection windows is as simple as iterating over that vector.

```
for (x, y, w, h) in faces:
   cv.rectangle(img, (x, y), (x+w, y+h), (0, 255, 0),2)
```

With this information, the sample is ready for cutting and scaling. In the case of the developed application, the false positives detection rate will be reduced as only one individual will be in front of the camera.

Depth detection

The presented model in Section 3.6.2 shows the default detector from the OpenCV samples but, how was that model obtained? The .xml file is the result of a tedious and long process of training the detection algorithm with over 5000 sample images for testing its performance.
A face recognition system could be tricked by making it believe that what it "sees" is a person but instead, is looking at a picture. As the detector uses a black and white image

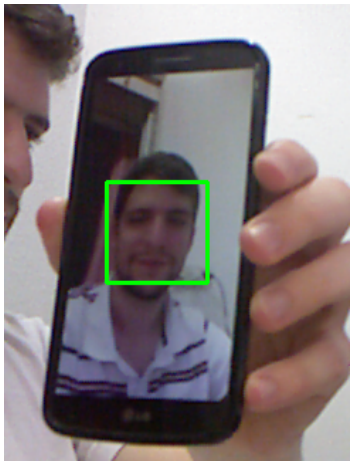

Figure 3.11: Tricking face detector

it lacks from depth information making it impossible to determine if the face is actually a real person or an image. For that reason and as we count with that depth information, I decided to train a Depth Face Detector.

The explained process is based on the tutorial shown by Naotoshi Seo at his blog [44], the utilities and software used were downloaded from [45] to accelerate building the training and testing directories and definition files.
The first step to train the detection model is to find images of the desired object to be detected – it could be anything – in our case, depth map faces labeled as "positives". Some

of the used images are shown in Figure 3.12.

Figure 3.12: Positive training images

The number of positive samples used is 153 depth images of faces. Quite smaller compared to the one used for the black and white detector but good enough as we will see at the end of this section.
Opposite to the positive samples, we have the negatives which are depth images of anything that is not a face so the detector could differentiate between them. The number of negative samples is the same as positives and also easier to generate.
Two auxiliary files must be created named *negatives.dat* and *positives.dat* that are basically lists of the sample names stored in each directory. This files could be easily created using a basic Linux command:

```
find negatives/ -name '*.png' > negatives.dat
find positives/ -name '*.png' > positives.dat
```

Using the **opencv_createsamples** function [46] and the script developed by Naotoshi Seo [45] **createtrainsamples.pl** we generate the training samples.

```
perl createtrainsamples.pl positives.dat negatives.dat samples 153
    "opencv_createsamples -bgcolor 0 -bgthresh 0 -maxxangle 1.1 -maxyangle 1.1
    maxzangle 0.5 -maxidev 40 -w 20 -h 20"
```

Under the samples directory this script will create the number of indicated samples – in this case 153 – by creating a file with *.vec* extension. With this tool you could create training samples from existing ones by applying minor changes such as rotation and scaling. Just like we did for the negatives and positives, we need to create a *.dat* file referring to the stored files in the samples directory.

```
find samples/ -name '*.vec' > samples.dat
```

Yet another *.vec* file needs to be created, in this case by merging the ones stored in the sample directory. For this task we use a python script developed by Blake Wulfe [45].

```
python mergevec.py -v samples -o samples.vec
```

So far we created and defined the training samples as well as *.dat* index files to them. Now is time to create the testing set and in this case we will use Naotoshi Seo's [45] script named **createtestsamples.pl** that takes the positive samples and after a transformation, places them in a negative sample that is being used as background.

Figure 3.13: Training images labeled as positive samples

As we can see in the generated images shown in Figure 3.13 the positive sample is placed rotated and scaled in a 3D space randomly over the negative image. This images will be later used to test the trained model's accuracy.

```
perl createtestsamples.pl positives.dat negatives.dat tests 153
    "opencv_createsamples -bgcolor 0 -bgthresh 0 -maxxangle 1.1 -maxyangle 1.1
    -maxzangle 0.5 maxidev 40"
```

In this case, the output result is saved in the directory named *tests* and for each training sample generates a directory with the same name placing the test images such as the ones in Figure 3.13.
To finally conclude the samples creation, a *.dat* file must be created to index the test directory files.

```
find tests/ -name 'info.dat' -exec cat \{\} \; > tests.dat
```

The command merges the information stored at every subdirectory of the samples folder. Now we are able to start the haarcascade training by providing all the created files. For further explanation and tuning of the learning algorithm refer to OpenCV documentation at [46] as in this case I only show the command I had used for training my own model.

```
opencv_traincascade -data haarcascade -vec samples.vec -bg negatives.dat -nsplits
    2 -npos 2500 -nneg 2500 -mem 500 -w 20 -h 20 -minhitrate 0.999 -maxfalsealarm
    0.5
```

At the very beginning of the execution we get the complete information of the parameters being used to train the model. Figure 3.6.2 shows the parameters for the given command. The algorithm iterates over the specified stages, adjusting the model incrementally at every step as shown in Figure 3.6.2 storing the temporary result in an xml file. When the training finishes, the resulting model is stored in a definite xml within the model directory. Just a snippet of the resulting file is shown below.

```
<opencv_storage>
   <cascade><stageType>BOOST</stageType>
   <featureType>HAAR</featureType>
   <height>20</height><width>20</width>
   <stageParams><boostType>GAB</boostType>
   <minHitRate>9.9500000476837158e-01</minHitRate>
   <maxFalseAlarm>5.0000000000000000e-01</maxFalseAlarm>
   ...
```

```
PARAMETERS:
cascadeDirName: haarcascade3
vecFileName: samples.vec
bgFileName: negatives.dat
numPos: 2000
numNeg: 1000
numStages: 20
precalcValBufSize[Mb] : 1024
precalcIdxBufSize[Mb] : 1024
acceptanceRatioBreakValue : -1
stageType: BOOST
featureType: HAAR
sampleWidth: 20
sampleHeight: 20
boostType: GAB
minHitRate: 0.995
maxFalseAlarmRate: 0.5
weightTrimRate: 0.95
maxDepth: 1
maxWeakCount: 100
mode: BASIC
```

```
===== TRAINING 0-stage =====
<BEGIN
POS count : consumed   2000 : 2000
NEG count : acceptanceRatio    1000 : 1
Precalculation time: 18
+----+---------+---------+
|  N |    HR   |    FA   |
+----+---------+---------+
|   1|        1|        1|
+----+---------+---------+
|   2|        1|        1|
+----+---------+---------+
|   3|        1|        1|
+----+---------+---------+
|   4|        1|        1|
+----+---------+---------+
|   5|    0.997|    0.492|
+----+---------+---------+
END>
Training until now has taken 0 days 0 hours 0 minutes 28 seconds.
```

Figure 3.14: Haarcascade model training

This file has the exact same structure as the one introduced in 3.6.2 but in this case is specialized on detecting faces in depth maps.

The precision of this detector is highly inaccurate compared to the one provided by OpenCV but is good enough to determine if the sensor is in front of a picture or a real person.

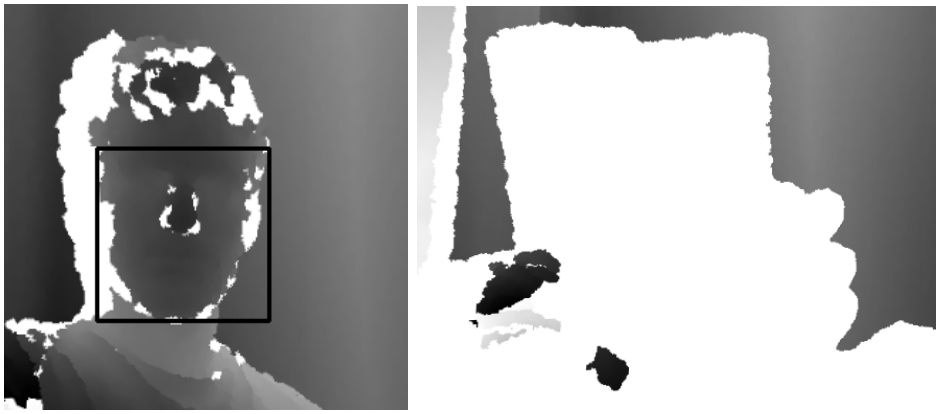

Figure 3.15: Depth Detector testing

In Figure 3.14 in the left image the detector is tested in a real time situation with good results for static images. While the image on the right side shows the same situation as in Figure 3.11 and in this case – logically – there is no detection.

By the end of this process the depth image is smoothed to minimize the noise and remark

Face Features as shown in Figure 3.16.

```
cv.medianBlur(depth_img,5)
```

3.6.3 Cut & scale

In Section 3.6.2 we mention the resulting structure of the detection process. As we said, it is a vector where each component contains the x and y position of the detection rectangle and the height and width so its quite simple to calculate the pixels where the face is located or even draw a rectangle as we did in many of the examples shown. The samples will be discarded if they don't pass both detection tests and for the case of trimming, the area to be cut off will be determined by the the detection result of the B&W image exclusively.

The resulting samples must have an standardized size to train the face recognizer properly. That is the reason why every sample is scaled to a 100x100 size.

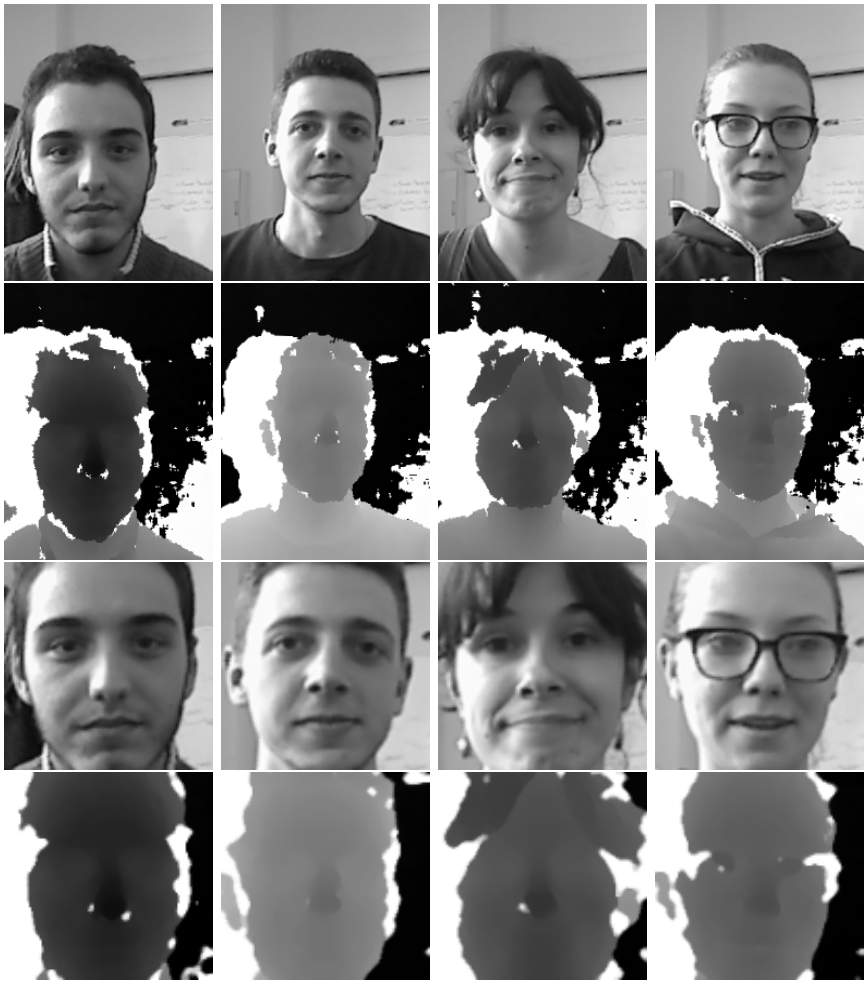

Figure 3.16: Preprocessing samples

In Figure 3.16 we can see for the first two rows the BW and depth raw data and in the next two rows the resulting data of the preprocessing explained along this section. The depth image is cut using the black and white detector as a reference mainly because the trained face detector for depth images tends to come up with false positives.

3.7 Face recognition - Recognizer module

The Recognizer Module sums the desired functionalities related to Face Recognition by using the algorithms cited in Section 2.4 with the addition of functionality for graphics creation for further analysis and understanding of the provided results.

Fortunately OpenCV provides straightforward methods for Face Recognition model creation, training and testing [23] so basically this module combines the functionality provided by the Manager module [Section 3.4] to query data from the database and the Preprocessing Module [Section 3.6] for enhancements and data processing before training and testing the Recognition Model.

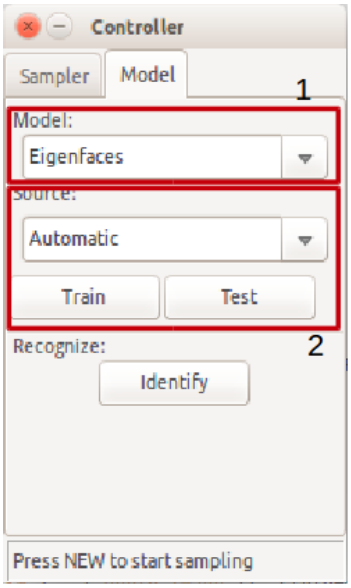

Figure 3.17: Model tab

Recalling Figure 3.22 in the Controller window under the tab named *Model* we have the functionality shown in Figure 3.17:

1. **Model selector:** you can choose between the two different algorithms; Eigenfaces or Fisherfaces.

2. **Sources selector:** with the possibility of choosing between BW – Black and White – and Depth images as source for training the model with the specified algorithm. The automatic mode in this selector works for the On-line Recognition explained in Section 3.9. The two buttons below the selector allow the user to train and test models.

Using the built-in functions from OpenCV the only problem that arises is data preparation. The KI-FRS solves it using a method provided by the Manager that prepares the training or testing matrix with the corresponding labels vector. A snippet of this function is shown below:

```
def get_samples(self,mode="tr",type="bw"):
   ...
   samples_matrix = np.vstack((samples_matrix,img_vector))
   samples_labels = np.vstack((samples_labels,int(label)))
   ...
   return samples_matrix,samples_labels,names
```

The function iterates over the image collection, converting each sample into a vector and stacking them to build a matrix, working in the same manner for the labels. As a result, returns the matrix of samples where each row is a vectorized image, the row vector of labels and an array with the name references of the image files.
The Recognizer Module uses the returned values to train the model:

```
"""Training model method"""
def tr(self,source=None):
   ...
   tr,labels,names=self.manager.get_samples("tr",source)
   self.model.train(tr,labels)
   ...
```

As for the testing function by providing a single sample, predicts its corresponding result to be later compared to its real label. By counting the times the prediction fails over the total number of test samples, we get the overall recognition error.

3.7.1 Recognition model

We have already mention the training and testing functionalities performed over the model but so far we did not get into detail with the model itself. OpenCV provides the functionality of creating Face Recognition Models in two different styles:

1. **Let the program decide for you:** in this case you have to simply create the model without any parametrization:

   ```
   model = cv.createEigenFaceRecognizer()
   ```

   In this case we are creating a model without threshold value nor number of components to preserve. Though OpenCV will decide a certain value of components with the risk of not being the optimum.

2. **Specifying threshold value and/or number of components:** the threshold value defines a border between recognized samples and unrecognized ones. The closest stored sample to the provided image for prediction will have to be in the range defined by the threshold value to be qualified as recognized and in that case, a label will be returned as the prediction result. In the case of exceeding the threshold, a -1 will be returned indicating that there is no match classifying the individual as unknown.

```
model = cv.createEigenFaceRecognizer(threshold = 1500.00,
    num_components=40)
```

The other parameter to specify is the number of components to preserve. This is always an important matter to resolve that could be tackled by a simple analysis of the resulting error when testing the model with different number of components.

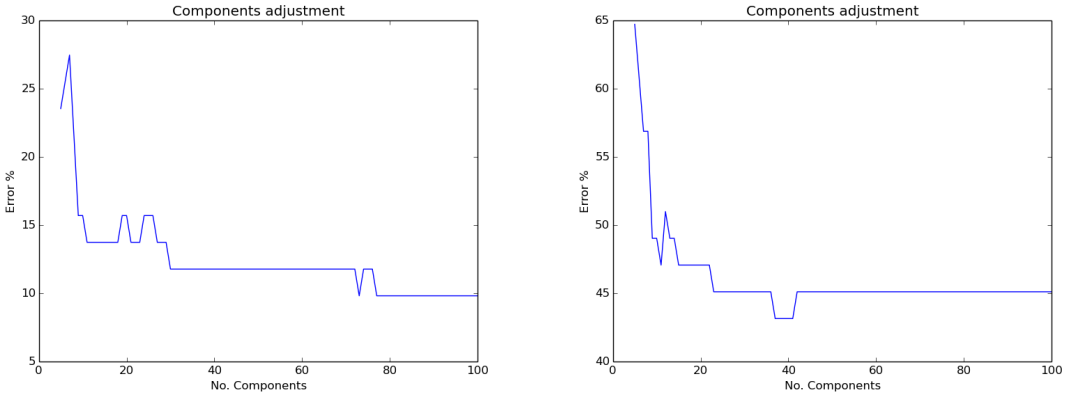

Figure 3.18: Eigenfaces component adjustment for BW (left) and depth (right) images

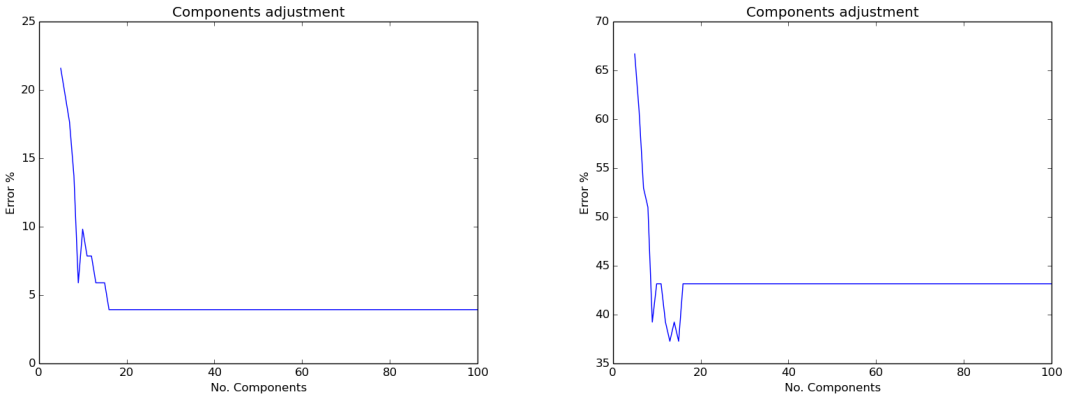

Figure 3.19: Fisherfaces component adjustment for BW (left) and depth (right) images

Figures 3.18 and 3.19 show the mentioned analysis. Based on the resulting minimum values we can determine which is the number of components that reduces the output error.

Table 3.1 shows the specific number of components value that gives us the minimum error rate. In Table 3.2 we have the error percentage for those number of components.

To build the shown results, we execute the following function script named *optimum_components* that creates the graphics automatically and also provides us the number of components with the minimum error percentage:

		Face Recognition Model	
		Eigenfaces	**Fisherfaces**
Source images	**BW**	73	16
	Depth	37	13

Table 3.1: Number of components analysis

		Face Recognition Model	
		Eigenfaces	**Fisherfaces**
Source images	**BW**	9.80 %	3.92 %
	Depth	43.13 %	37.25 %

Table 3.2: Error percentage analysis

```
def optimum_components(type,source,mini,maxi):
   x = []; y=[]
   for i in range(mini,maxi+1):
      rec=Recognizer(type,source,num_components=i)
      rec.tr(source)
      ret = rec.ts(source)
      x.append(i); y.append(ret)
   plt.plot(x,y)
   plt.title("Components adjustment")
   plt.xlabel("No. Components")
   plt.ylabel("Error %")
   plt.show()
   return x[y.index(min(y))]
```

This function was also included inside the Recognizer Module – with some minor changes as it creates an object of the class where it is defined – to create a model with the best precision possible in order to get the best results for the on-line recognition.

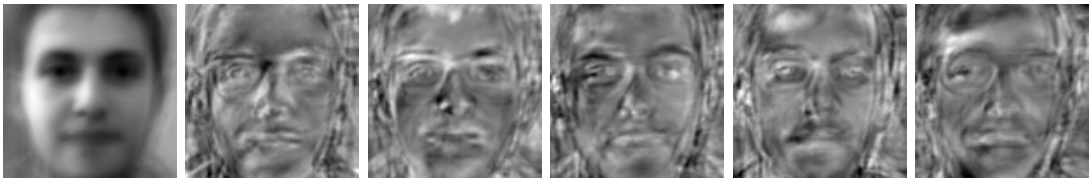

Figure 3.20: Mean and 5 topmost eigenfaces for BW (left to right)

The trained model stores the principal components of the sample images under the label of training. In Figure 3.20 we show the mean face and the topmost eigenfaces for the black and white images while in Figure 3.21 we have the same information for the Depth images. The model is stored in the *recognition* directory inside the *data* folder of the project.

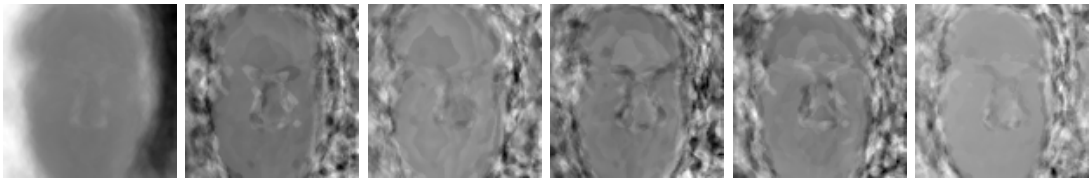

Figure 3.21: Mean and 5 topmost eigenfaces for Depth (left to right)

The code for this Section could be found in the project's repository [7] in the *src* directory under the file named **recognizer.py**.

3.8 Sampling - Database creation

Without face samples we are not able to develop a Face Recognition application. As we have mentioned in Section 2.4.2 there are multiple Face Recognition databases available but very few for our purposes using the Kinect device. The requirement of building our own database arose and it has been fulfilled with the KI-FRS application as explained in this specific Section.
We must have the Kinect device connected and all the libraries and software necessary to run the application [Section 3.3].

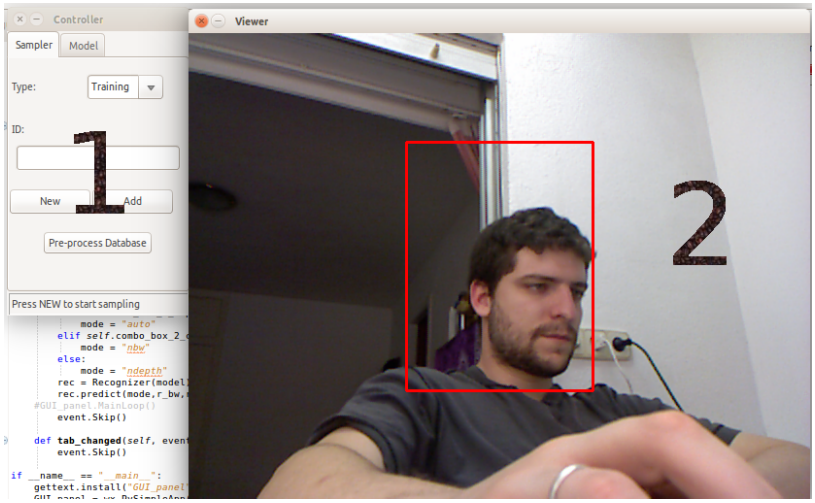

Figure 3.22: KI-FRS sampler

In Figure 3.22 we have the KI-FRS running. This application has two windows.

1. **Controller:** a floating panel with options for sampling and model training for recognition –covered in Section 3.7. This floating panel has the following functionalities:

   (a) **Type combo box:** to select whether the sample is going to be considered for training or testing.

   (b) **Sample creation:** the text area must be filled with the name of the person being sampled. By clicking on *New* a directory will be created within the type folder to store the samples. A detailed explanation of this directory structure could be found in Section 3.4. The *Add* button automatically creates a group of files following a numbering order – Figure 3.24.

   At the same time when the first sample is stored, the *Index.txt* file is updated with the name of the person for future reference in the Recognition process [Section 3.7].

   (c) **Pre-process database:** by clicking on this button, the system automatically executes the entire preprocessing cycle [Section 3.6] creating the files *ndepth* and *nbw* for every image file processed.

   (d) **State bar:** the state bar displays short messages as response to the user inputs.

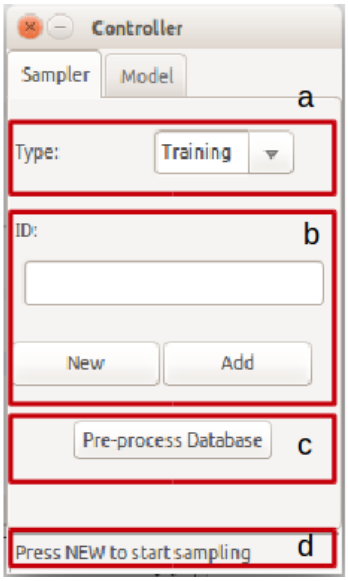

Figure 3.23: KI-FRS sampler

Figure 3.24: Files created from a single sample

2. **Viewer:** is the main window of the application. It displays what the camera is currently capturing and also has a fixed rectangle drawn in red. In Figure 3.25 we see how the color of the fixed rectangle changes to green when a face is detected. A face detector is being used for assisting the user while sampling. The main idea of this assistant is to make the person look at the camera as frontal as possible to facilitate the preprocessing refinement [Section 3.6]. It is not necessary for the application to detect the face in order to take the sample as the detector may fail in poor light conditions but by being detected it guarantees that it will be used to train the Recognizer.

Figure 3.25: Face Detection assistance

3.9 On-line Recognition

The KI-FRS is capable of performing on-line recognition in a straightforward manner. Recalling to Figure 3.17 we have in the lower part of the controller window a button option named *Identify*. By clicking on it automatically triggers the following actions:

1. KI-FRS captures the image from the Kinect device. As shown in Figure 3.25 the part of the whole displayed image to be cutted will be within the rectangle of 192x256 pixels.

2. Both the RGB and depth images from the previous step are sent to the preprocessor following the process described in Section 3.6. The final result are the same kind of images with enhancements and resized to 100x100 pixels.

3. Finally this data is sent to the Recognizer that, depending on what type of recognition you have chosen, determines what person is in front of the camera. To do this, it displays a new window with the person's name and image from the training samples. Figure 3.26 shows the result of the explained process.

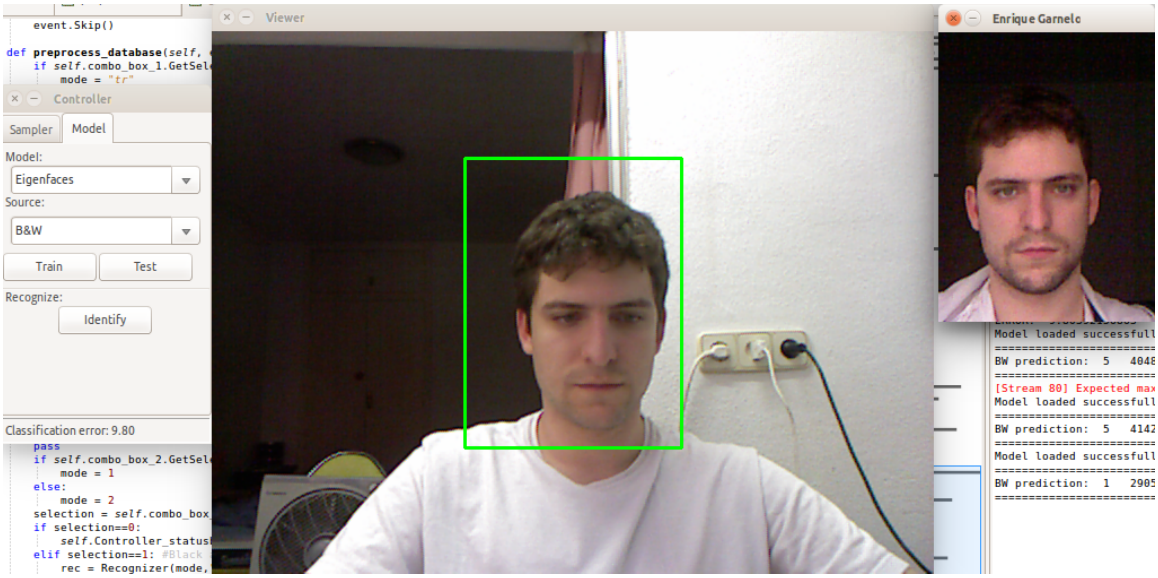

Figure 3.26: On-line Face Recognition result

In Figure 3.27 we show a more challenging case where the Recognizer performs correctly.

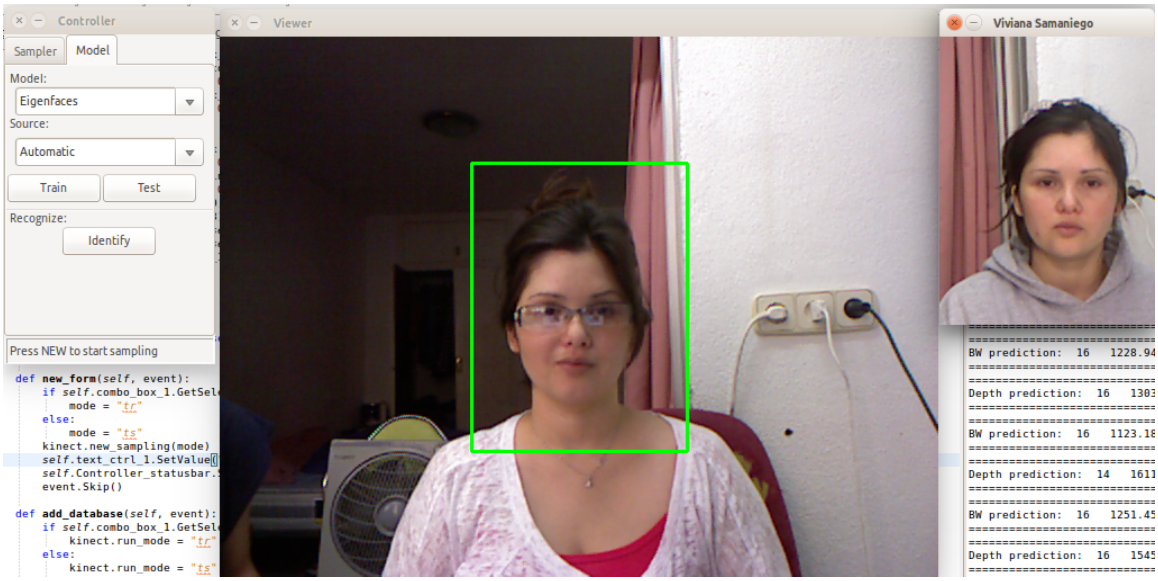

Figure 3.27: Challenging on-line Face Recognition result

Chapter 4

Experimentation

Up to now we have covered the implementation difficulties and challenges while developing a Face Recognition System using depth images to get better results in light varying conditions.

The Chapter is structured as follows: first in Section 4.1 we will test the KI-FRS for our own built database – as explained in Section 3.8 – to later in Section 4.2 focus tests on bigger and well-known datasets for Face Recognition.

By the end of this chapter the reader will have a clear view of how accurate is the KI-FRS and how this experiments could be recreated and scaled for further tests. The results will later define whether the dissertation objectives defined in Section 1.1.1 were fulfilled or not.

4.1 KI-FRS testing

Along this project we introduced the functionalities and capabilities of the KI-FRS up to this Section. The following experiment will put them into test.

4.1.1 Preprocessor - Detection

The preprocessor is one of the most important components in our system. Being the filter for any image being processed in the Face Recognition process, works with fast algorithms for image enhancements.

To test the preprocessor we provide different input images to the algorithm, measuring the intensity with the method explained in Section 3.6 to later detect faces in the enhanced image.

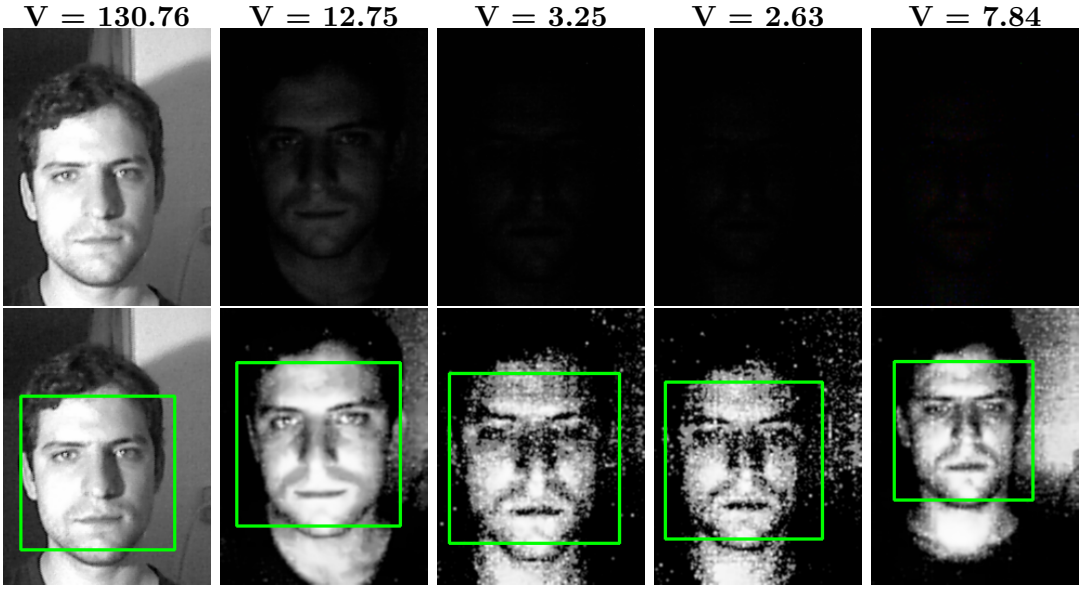

Figure 4.1: Preprocessor results in varying light conditions

As we can see, even with extremely dark input images, after enhancing – and despite the resulting noise – the detector is able to perform correctly. As for the input images with V values below 10 we can appreciate face features in the resulting images.

The case of the depth sensor is quite different as the detector is under-trained in comparison to the black and white detector. Though it is able to determine whether is a face in front of the camera or a flat image of a face as shown in Section 3.6.2.

4.1.2 Recognizer

After adjusting input images for our model training or testing we are capable to assure that we have the highest number possible of adequate images for this purpose.

The resulting database consists of 102 images for training and 51 for testing. The samples were taken from a group of voluntary students who offered to appear in this work.

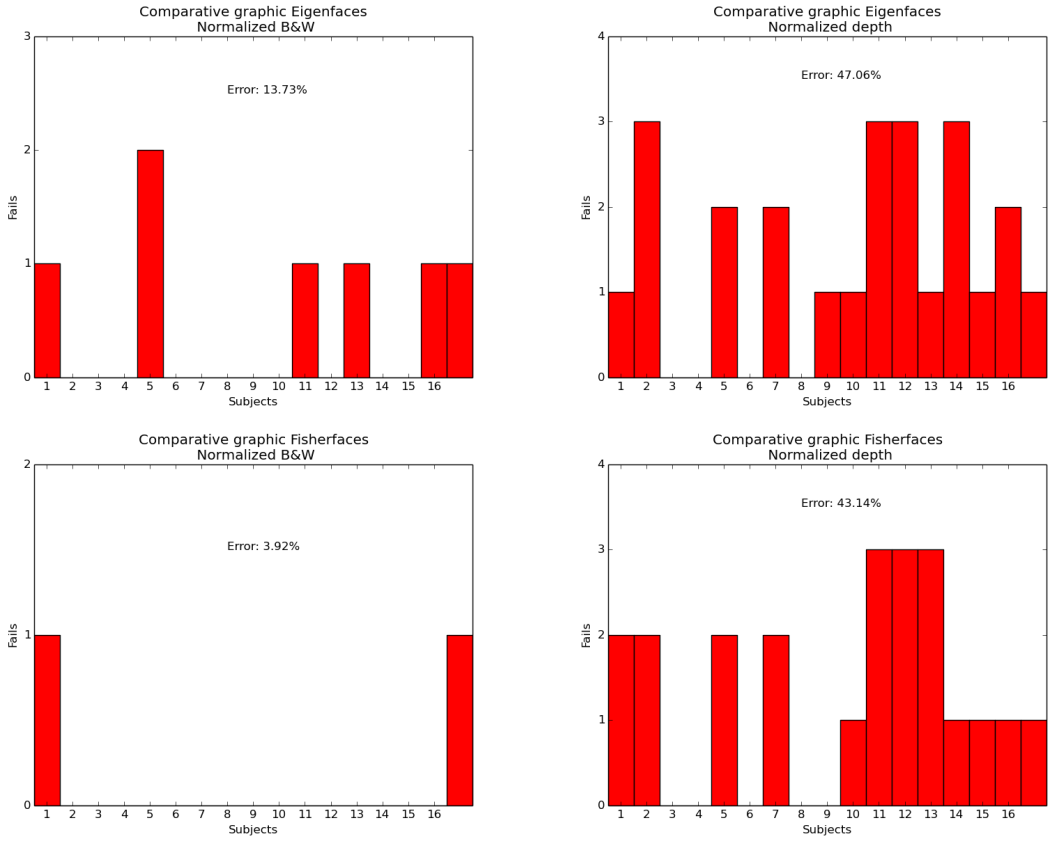

Figure 4.2: Number of errors per individual

In the balance per subject, the Fisherfaces method performs better for black and white images as well as for depth samples. This could be easily explained with Figure 4.3[32].

The resulting graphics were created from the Recognizer class with the following instructions:

```
rec = Recognizer(2,"ndepth")
rec.tr()
ret = rec.ts(True)
```

As our number of training images is relatively low per person, the Fisherfaces performs more accurately than the Eigenfaces method. In the other hand we can see that precision rates are inaccurate in general for the depth images. The main reason being that even though they are recognized as faces with their features and characteristics, there is not a great difference amongst samples.

The “auto” mode performs correctly in the case there is a matching for both black and

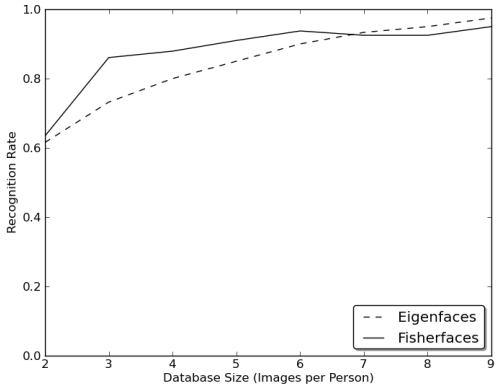

Figure 4.3: Eigenfaces vs Fisherfaces recognition rate analysis

white as well as depth. If this doesn't happen, the trusted result would be the black and white image output as it performs better.

4.2 Testing on external databases

In Section 2.4.2 we referred to the external databases available for benchmarking face recognition algorithms and about the fewer options – up to date – that we have for our problem approach.

During the development of this project I wasn't able to work with a bigger database that fulfills the system needs just like the one presented in Section 4.1. So the purpose of this Section is to benchmark the recognition algorithm on black and white images with considerable light variations in order to test the Preprocessor Module of the KI-FRS.
The selected databases for this testing are the following:

1. **The ORL Database of Faces:**[34] also known as the AT&T Face Recognition Database is composed of 10 images of 92x112 pixels with 256 grey levels per pixel for 40 individuals taken at different times, varying the light, facial expressions and details such as glasses with a black background.

2. **MIT CBCL Face Recognition Database:**[35] consists of high resolution pictures with varied illumination, pose and background. The set has 200 images per subject.

3. **Extended Yale Face Database B:**[36] is the extended version of the original Yale Database with 16128 images of 28 people under 9 poses and 64 illumination conditions.

For this particular purpose and to prevail modularity and reduce code complexity, two separate classes were created, *ExternalDB_Recognizer* that inherits the code implementation from the original Recognizer and a class named *External_DB_Manager* within the Manager package. Detailed implementation of this classes could be consulted in files **recognizer.py** and **file_manager.py** under the *src* directory at the KI-FRS repository[7].

4.2.1 Data preparation

Logically the referred databases in Section 4.2 have different conventions for naming and classifying their data. For that reason a data preparation process was implemented. Independently the database we are working with there is a function in the *External_DB_Manager* that creates the following index or reference files:

- **tr.dat:** listing the images that will be considered for training the recognition model. The file has the following structure:

```
s11
/home/henrygp/git/KI-FRS/data/extern_dbs/att/s11/10.pgm
/home/henrygp/git/KI-FRS/data/extern_dbs/att/s11/5.pgm
...
s26
/home/henrygp/git/KI-FRS/data/extern_dbs/att/s26/6.pgm
/home/henrygp/git/KI-FRS/data/extern_dbs/att/s26/5.pgm
...
s17
/home/henrygp/git/KI-FRS/data/extern_dbs/att/s17/10.pgm
```

```
/home/henrygp/git/KI-FRS/data/extern_dbs/att/s17/6.pgm
...
```

The shown example is a snippet of the training file created for the **AT&T Database** where at first the class label is indicated and below the full path for each training file under that label is listed. This sequence is repeated until the end of file.
For creating this file the function groups the set of images under each label and divides it based on a certain percentage that is provided by a function parameter. So for example if we set the percentage to 80% in the **AT&T Database** for the ten images under the label *s11* will be randomly divided into 8 images for training and 2 for testing.

- **ts.dat:** this file has the same structure than the **tr.dat** file with the only difference that this files are considered later for testing the trained face recognition model.

For the case of the **MIT CBCL Face Recognition Database** as the data is already divided into training and testing, the creation procedure is straightforward. For the two other databases an specific function is previously executed to create a file named **sample.dat** that groups all the images under each label. This file is later processed to create the training and testing files described above.

4.2.2 Testing

The first part of experimenting with this databases is to test the face recognition performance for the randomly created datasets. Except for the **MIT CBCL Face Recognition Database** where data is already separated, we iterate 10 times and from the obtained results we calculate the mean rate of the recognition testing. For fragmenting the dataset we used 70% from each individual's images for training and the rest for testing.

For the filtered dataset we use the KI-FRS preprocessor to estimate light and apply the corresponding enhancements as we explained in Section 3.6. In this case the **Extended Yale Face Database B** becomes a problem for the face detector as faces are already fitted to the image's size. In that particular case the enhancement is applied without face detection.

As for light variations in Figure 4.4[1] we show some input images from **Yale Database** that have extremely poor lightning and its corresponding result after preprocessing. Face features are clearly differentiated being suitable samples for the face recognition model.
Table 4.1 reflects the recognition rates obtained for the explained experimentation while Table 4.2 sums up the number of sample images used for the described tasks.
Note that the results of filtered images reflected in Table 4.2 may vary as the training and testing sets are randomly built.

[1] Images taken from Yale Extended B database

Figure 4.4: Preprocessor results in extremely poor light conditions

	Dataset			
	Unfiltered		**Filtered**	
Database	**Eigenfaces**	**Fisherfaces**	**Eigenfaces**	**Fisherfaces**
AT&T	96.25%	94.00%	89.74%	92.45%
MIT	11.55%	8.80%	17.91%	36.66%
Yale	77.86%	96.13%	89.94%	100 %

Table 4.1: Recognition rates for external databases

	Dataset			
	Unfiltered		**Filtered**	
Database	**tr**	**ts**	**tr**	**ts**
AT&T	280	120	226	95
MIT	3240	2000	3232	2000
Yale	1420	630	1325	587

Table 4.2: Sample filtering analysis

Chapter 5

Conclusions

The experimentation explained in Chapter 4 clearly shows in what extent were the goals established in Section 1.1.1 fulfilled. In this concluding Chapter we will discuss about the project's contribution to the Face Recognition field, particularly when using depth images to cope with light varying environments. The software component is also evaluated since it probed to work consistently and with a simple and friendly Graphical User Interface.

Finally future lines of work are proposed to improve experimental results and enhance the current Biometric Face Recognition System implementation.

5.1 Experimental results

In Chapter 4 we used two different approaches to test the KI-FRS results, the first one with our own built database using the GUI interface that the System facilitates and the second approach on three different well-known Face Databases that are also commonly used for measuring a Face Recognition System's accuracy.

The first part for the KI-FRS probed outstanding results for images with poor light conditions that were later used in the detector. Though, the image quality in the end of the process tend to be fairly noisy – despite all the pre-processing – being also an important factor to be considered. But despite this fact, the resulting number of images for a certain individual that were fed to the Recognizer increased considerably, something that could be clearly noticed in the later experimentation over the standardized databases.

As for the preprocessing of the depth image as we mentioned, there is not much room for improvement as the resulting image from the Kinect sensor is not particularly precise and tends to be noisy and with areas that are not sensed. Though, many methods are proposed either to fill areas that were out of range for the depth sensor [29] or smoothing the image with a mean filter – as we did in our case.

For the case of tests performed on the KI-FRS we can conclude that there is room left for further experimentation as well as for improvement not in relation to the amount of people included in the database but on the number of samples per individual, something that probes, between certain limits, to improve Recognition performance as stated in Figure 4.3 where the Eigenfaces and Fisherfaces methods are contrasted for the described situation.
Regarding the Recognition task in depth images, results were not concise and need further experimentation in order to come up with a decisive result. With the amount of samples per person used, the results were fairly random.

Using the external databases for the experimentation was a reasonable chance to test how the KI-FRS Recognition module behaves in the case of large sets of images. Unfortunately this couldn't be put into test for depth images, leaving this task for further experimentation. Recalling the expectations of either increasing the number of samples involved in the Recognition process or avoiding useless samples to be used, Table 4.2 reflects the accuracy and the importance of the KI-FRS' Preprocessing module and how this later affects the Recognition rate.
We can clearly identify a low Recognition pattern for the MIT database despite the fact that after filtering, we are using almost the entire dataset. This low rate could be caused mainly by two aspects:

- Nearly 50% of the samples are being used for testing purposes. Meaning that it may not be enough samples for training the model properly.

- There are over 300 images for each subject, meaning that the model may be overtrained hence, it lacks of generalization capability.

Yale database on the other hand presents a major accuracy in Recognition for the filtered set of images but don't let this high result trick you since this accuracy may correspond to an over-trained model that lacks of generalization capability.

Based on this conclusions we are able to define the goals achieved in this project and the future lines of work that could enrich and increase the KI-FRS qualities.

5.2 Achieved goals and contribution

Recalling the established goals in Section 1.1.1 and the performed tests in Chapter 4 we have achieved the following milestones:

- **Robustness to light variations:** in Section 4.1 was proven that the utilized algorithm for light enhancement was able to improve the image quality. This leaded to Face Detection effectiveness for cutting and scaling the sample for later training of the recognition model.
- **Portability:** as we used cross-platform libraries and Python for the development, we assure that the KI-FRS could be executed in any platform. Though some troubles may arise during installation process as libraries and dependencies setting up is not straightforward in some cases.
- **Versatility:** Section 4.2 is a great example of the KI-FRS versatility. By simply inheriting from an implemented class, we were able to integrate the recognition functionality for external databases.

The most important – and interesting – contributions of this work are listed below:

1. Development of a portable system that works with the Kinect sensor using open-source libraries and code accessible for everyone.
2. Experimental results on face detection with depth images in particular, scalable to any kind of object.
3. Well documented and readable code for future lines of work. Uses OpenCV, a very well-known library in the Computer Vision field.

5.2.1 Future lines of work

Even though the resulting conclusions achieve the expected goals, there is much to work regarding depth images. In Section 4.1 we showed that recognition rates tend to be very low specially because facial features are not significantly different in that particular type of images. Moreover in some cases the facial features are not captured at all because the person is too close or far from the sensor. Taking this facts into account we propose the following future lines of work:

- **3D Face Recognition:** the KI-FRS could potentially be scaled to a 3D Face Recognition System. To achieve this, the extracted values stored in the raw matrix could be translated into distances and using the black and white image, a 3D model could be built.
- **Other recognition methods:** the Kinect sensor also counts with an array of microphones that could be used for voice recognition and in conjunction with the KI-FRS integrate a complete system that takes full advantage of the sensor's functionalities.

- **Build a more complete database:** KI-FRS has all the tools and functionalities for building a normalized database for further study. One of the weakest points of this work was the lack of databases to test its performance for depth images taken from the Kinect device. A more complete database with more individuals and samples per person could lead to an accurate analysis of the system's functionalities.
- **Use of local features:** being another perspective for this work and sacrificing performance, could be a good line of work for analyzing the improvements that local features could make to face recognition in depth images.
- **Testing with a larger database of depth images:** being less challenging than the above items, this could be the chance to probe whether the Recognition rate improves when using multiple depth images per person or is only getting worse.

Bibliography

[1] KUNDERA M.: *Immortality*, NY 1991, ISBN 0571166784

[2] WILLIAMS M.: *"Better Face Recognition Software"*, May 30 2007, `http://www.technologyreview.com/news/407976/better-face-recognition-software/`

[3] KIMMEL R., SAPIRO G.: *"The Mathematics of Face Recognition"*, April 30 2013, `http://siam.org/news/news.php?id=309`

[4] *"Face Recognition Grand Challenge Official Website"*, December 16 2014, `http://www.nist.gov/itl/iad/ig/frgc.cfm`

[5] BRONSTEIN A., BRONSTEIN M., KIMMEL R.: *Three dimensional Face Recognition*, December 10 2014, `http://www.cs.technion.ac.il/~ron/PAPERS/BroBroKimIJCV05.pdf`

[6] THORAT S.B., NAYAK S.K., DANDALE J.: *Facial Recognition Technology: An analysis with scope in India*, January 2010, International Journal of Computer Science and Information Security, Vol. 8, No. 1 `http://arxiv.org/pdf/1005.4263.pdf`

[7] GARNELO PREDIGER E.: Project repository with described files and used functionality. `https://github.com/HenryGP/KI-FRS`

[8] Python's official webpage. `https://www.python.org`

[9] wxGlade official project's website & wiki. `http://wxglade.sourceforge.net`

[10] Matplotlib, plotting library for Python. `http://matplotlib.org`

[11] Numerical Python –numpy– library. `http://www.numpy.org`

[12] MILLER B.: *"Everything you need to know about automated biometric identification"*,Security Technol. Design, Apr. 1997.

[13] Apple support official website. Touch ID security on Iphone and Ipad `https://support.apple.com/en-us/HT204587`

[14] WOODWARD, J.D.: *"Biometrics: privacy's foe or privacy's friend?"*,Page(s): 1480-1492, Proceedings of the IEEE Volume: 85 , Issue: 9 DOI: 10.1109/5.628723, 1997.

[15] BICHLIEN HOAN, ASHLEY CAUDILL: IEEE Emerging Technology portal, 2006 - 2012. `http://www.ieee.org/about/technologies/emerging/biometrics.pdf`

[16] What facial recognition technology means for privacy and civil liberties. Committee on the judiciary United States Senate, One hundred twelfth Congress, Second session, July 18, 2012. Serial No. J11287 http://www.gpo.gov/fdsys/pkg/CHRG-112shrg86599/pdf/CHRG-112shrg86599.pdf

[17] HOWSE, JOSEPH,*OpenCV Computer Vision with Python*,Packt Publishing 2013

[18] OpenKinect official website, section dedicated to the device's images. http://openkinect.org/wiki/Imaging_Information

[19] ROS official website wiki, section dedicated to depth camera calibration. http://wiki.ros.org/kinect_node

[20] KAN SANFORD: *"Smoothing Kinect Depth Frames in Real-Time"*, January 24 2012, http://www.codeproject.com/Articles/317974/KinectDepthSmoothing

[21] M. Stommel, M. Beetz and W. L. Xu, Inpainting of Missing Values in the Kinect Sensors Depth Maps Based on Background Estimates, Sensors Journal 2013, IEEE (Volume:14 , Issue: 4)

[22] HG R.I., JASEK P., ROFIDAL C., NASROLLAHI K., MOESLUND T.B., TRANCHET G.: *"An RGB-D Database Using Microsofts Kinect for Windows for Face Detection"*,Eighth International Conference on Signal Image Technology and Internet Based Systems,Laboratory of Visual Analysis of People, Aalborg University, Aalborg, Denmark, 2012

[23] Face Recognition with OpenCV. http://docs.opencv.org/modules/contrib/doc/facerec/facerec_tutorial.html

[24] WAGNER P.: *"Local binary patterns"*, November 8 2011, http://www.bytefish.de/blog/local_binary_patterns/

[25] VIOLA P., JONES M.: *"Rapid Object Detection using a Boosted Cascade of Simple Features"*, Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition,Volume: 1, pp.511-518, 2001.

[26] ABATE A., NAPPI M., RICCIO D., SABATINO G.: *"2D and 3D face recognition: A survey"*, Pattern Recognition Letters, vol. 28, Issue 14, pp.1885-1906, 2007.

[27] ASSADI A., BEHRAD A.: *"A new method for Human Face Recognition using Texture and Depth Information"*, 10th Symposium on Neural Network Applications in Electrical Engineering, NEUREL-2010, Faculty of Electrical Engineering, University of Belgrade, Serbia, September 23-25, 2010.

[28] ROSE R., SURULIANDI:*"Improving Performance of Texture Based Face Recognition Systems by Segmenting Face Recognition"*, St. Xavier's Catholic College of Engineering, Nagercoil, India. ACEEE International Journal on Network Security, Vol. 02, No. 03, July 2011.

[29] MIN R., KOSE N., DUGELAY J.: *"KinectFaceDB: A Kinect Database for Face Recognition"*, IEEE Transactions on systems, man, and cybernetics systems, Vol. 44, No. 11, November 2014.

[30] TURK M., PENTLAND A.: *"Eigenfaces for Recognition, Journal of Cognitive Neurosicence"*, Vol. 3, No. 1, 1991, pp. 71-86, `http://www.face-rec.org/algorithms/PCA/jcn.pdf`

[31] STARK H., WOODS J.: *"Probability, Random Processes, and Estimation Theory for Engineers"*,Prentice-Hall, 1986, ISBN 0-13-711706-X

[32] BELHUMEUR P., HESPANHA J., KRIEGMAN J.: *"IEEE Transactions on Pattern Analysis and Machine Intelligence"*, vol. 19, Number 7, July 1997. `http://www.cs.columbia.edu/~belhumeur/journal/fisherface-pami97.pdf`

[33] Face Recognition Homepage - list of available databases. `http://www.face-rec.org/databases/`

[34] The ORL Database of Faces - AT&T Face Recognition database. `http://www.cl.cam.ac.uk/research/dtg/attarchive/facedatabase.html`

[35] MIT CBCL Face Recognition Database `http://cbcl.mit.edu/software-datasets/heisele/facerecognition-database.html`

[36] Extended Yale Face Database B `http://vision.ucsd.edu/~iskwak/ExtYaleDatabase/ExtYaleB.html`

[37] GIPSA Laboratories Official Website. `http://www.gipsa-lab.grenoble-inp.fr`

[38] PARK S.:*"Retinex Method Based on CMSB-plane for Variable Lighting Face Recognition"*, School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Republic of Korea, 978-1-4244-1724-7 2008 IEEE.

[39] GROSS R., BRAJOVIC V.: *"An image preprocessing algorithm for illumination invariant face recognition"*, 4th International Conference on Audio and Video Based Biometric Person Authentication, pp. 10-18, 2003.

[40] Microsoft Kinect for Windows® official SDK webpage. `https://msdn.microsoft.com/en-us/library/hh855347.aspx`

[41] Kinect for Windows Sensor Components and Specifications `https://msdn.microsoft.com/en-us/library/jj131033.aspx`

[42] MOLLER B., BALSLEV I., KRUGER N.: *"An automatic evaluation procedure for 3-d scanners in robotics applications"*, Sensors Journal, IEEE, Vol. 13, No. 2, pp. 870878, 2013.

[43] HAN J., SHAO L., XU D., SHOTTON J.: *"Enhanced computer vision with microsoft kinect sensor: A review "*, , IEEE Transactions on Cybernetics, vol. PP, no. 99, pp. 11, 2013.

[44] SEO N.: *"Tutorial: OpenCV haartraining (Rapid Object Detection With A Cascade of Boosted Classifiers Based on Haar-like Features)"*, October 16 2008, http://note.sonots.com/SciSoftware/haartraining.html

[45] Auxiliary files to train a detector. http://note.sonots.com/SciSoftware/haartraining/mergevec.cpp.html#o95e5d87 https://github.com/wulfebw/mergevec

[46] OpenCV's official documentation for training your own detector. http://docs.opencv.org/doc/user_guide/ug_traincascade.html

[47] OpenKinect installation process for different platforms. http://openkinect.org/wiki/Getting_Started

[48] Installing OpenCV on Windows. http://docs.opencv.org/doc/tutorials/introduction/windows_install/windows_install.html OpenCV on MAC OS step-by-step http://tilomitra.com/opencv-on-mac-osx/

[49] Python's download page with multiple installers depending on the platform. https://www.python.org/downloads/

[50] OpenCV project official website. http://opencv.org

[51] Histogram equalization OpenCV explanation with code. http://docs.opencv.org/doc/tutorials/imgproc/histograms/histogram_equalization/histogram_equalization.html

[52] Simple and fast gamma correction using OpenCV. http://subokita.com/2013/06/18/simple-and-fast-gamma-correction-on-opencv/ Gamma FAQ. http://www.poynton.com/notes/colour_and_gamma/GammaFAQ.html

[53] Filtering images in OpenCV. https://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_imgproc/py_filtering/py_filtering.html#filtering

[54] Non local means denoising. http://www.ipol.im/pub/art/2011/bcm_nlm/ http://opencv-python-tutroals.readthedocs.org/en/latest/py_tutorials/py_photo/py_non_local_means/py_non_local_means.html

[55] Official repository samples for the python programming language. https://github.com/Itseez/opencv/tree/master/samples/python2

[56] http://openkinect.org/wiki/Main_Page OpenKinect project official website & wiki.

[57] MARTIN, J.: *Rapid Application Development*, Macmillan. 1991 ISBN 0-02-376775-8.