SINO-FRENCH LABORATORY FOR COMPUTER SCIENCE, AUTOMATION AND APPLIED MATHEMATICS – CHINESE ACADEMY OF SCIENCES

Eye Pose Tracking & Gaze Estimation

Guillaume BOULAY
Institut National des Sciences Appliquées de Lyon, France
Département Télécommunications – Services & Usages

Abstract: Computer vision applied to a fast and reliable features detector and tracker is a possible solution to improve Human-Computer interaction. This report describes two algorithms developed during my five months internship in LIAMA for eye pose tracking and gaze estimation. These algorithms are based on OpenCV library and use respectively openCV and SIFT for tracking part. Gaze estimation uses another approach which consists in using a matrix model and the iris to eye-corner vector to extract user gaze.

Summary

Table of figures

I. Introduction

This document presents the results of my end-study project realized between the 1st April and the 31 August 2008 as a research student. This project is a part of my Master of Engineering final year curriculum done at the National Institute of Applied Sciences (INSA Lyon, France) in the Telecommunications, Services & Uses department. In the framework of a newly created transversal option, I had the opportunity to lead this project in China within a Sino-French laboratory: Sino-French Laboratory for Computer Science, Automation and Applied Mathematics (LIAMA, Beijing).

I worked in a mixed team composed of Chinese and French people, lead by Pr. Chunhong PAN on the Mixing Video and Avatar (MVA) project. This team is divided in sub-teams which are working on different parts of the MVA project.

I would like to thank each person with whom I worked, and more particularly, Pr. Chunhong PAN and Dr. Haibo WANG for their help and knowledge. Of course, I also thank Vincent ENJALBERT, Lu WANGPING and Victor GRABAILLON which have worked like me on MVA project. It was a real pleasure to work in such a dynamic team. I would also like to thank Fabrice VALOIS, Stéphane UBEDA, Jialiang LU, Lucie LIU, and others teachers for the opening and success of OT-Chine which was a wonderful experience.

II. Project context

II.1. Mixing Video and Avatar Project (MVA)

The MVA project was at first proposed by Pr. CHAILLOU and Pr. PAN in November 2007 in Beijing. The following quote is a synopsis of the project:

"The research of distant collaborations, i.e. videoconferencing, collaborative virtual environment, remote medical treatment and remote educations, is a hot research topic in computer science field. One of the most challenging problems in this field is how to construct a functional humanoid agent that is either real-time controlled by a remote human counterpart or capable of receiving and expressing the perceptions of its remote human client.

In the project, we propose a novel humanoid agent driven by the performance of a remote person in the form of Mixing Video and Avatar. To best support mediated collaboration, the developed MVA agent is capable of imitating the gaze and gesture behaviors of its human-counterpart, and more amazingly, has the ability of 'cloning' the appearance of its human client. As a tool to represent human, the MVA is specifically suitable, but not limited to desktop virtual environment, by which physically remote partners (2 or 3 persons) can collaborate as in a face-to-face situation. In realizing the MVA, state- of-the-art computer vision and computer graphics technologies are mostly employed, which makes it independent of any complicated hardware except for an ordinary monocular webcam."

Research is carried out in the following aspects:

- *Performance-driven avatar Head:* 3-D head pose and 2-D eye pose are first tracked from video sequences used as input data. Next, the frontal-view faces textures and gaze data are archived with some intelligent computations. Finally, those data are transmitted and rendered by a virtual avatar. Certain results have already been gotten regarding head pose tracking.

- *Behavior-driven avatar arm-hand gestures:* real human behaviors are derived from webcam-captured videos; then, through a filtering process, the perceptually meaningful gestures will be reserved and copied by 3-D virtual arms.

- *A network platform supporting avatar communication:* the platform should support one-to-one, one-to-more and more-to-one data transmissions. Transmitted data are made up of audio, video and motion data. An initial platform has been developed by Fredric ROY for those purposes.

- *Performance Evaluations of avatar:* the evaluative application is a multi-player 3-D puzzle game, in which each player is embodied with the representation of his personalized avatar.

Following (Figure 1), the flowchart of Mixing Video and Avatar: in (1), both video tracking and data processing are done in the local computer; then in (2), extracted data are broadcasted to each partner; lastly in (3), each host terminal renders a Mixing Video and Avatar agent according to the received data.

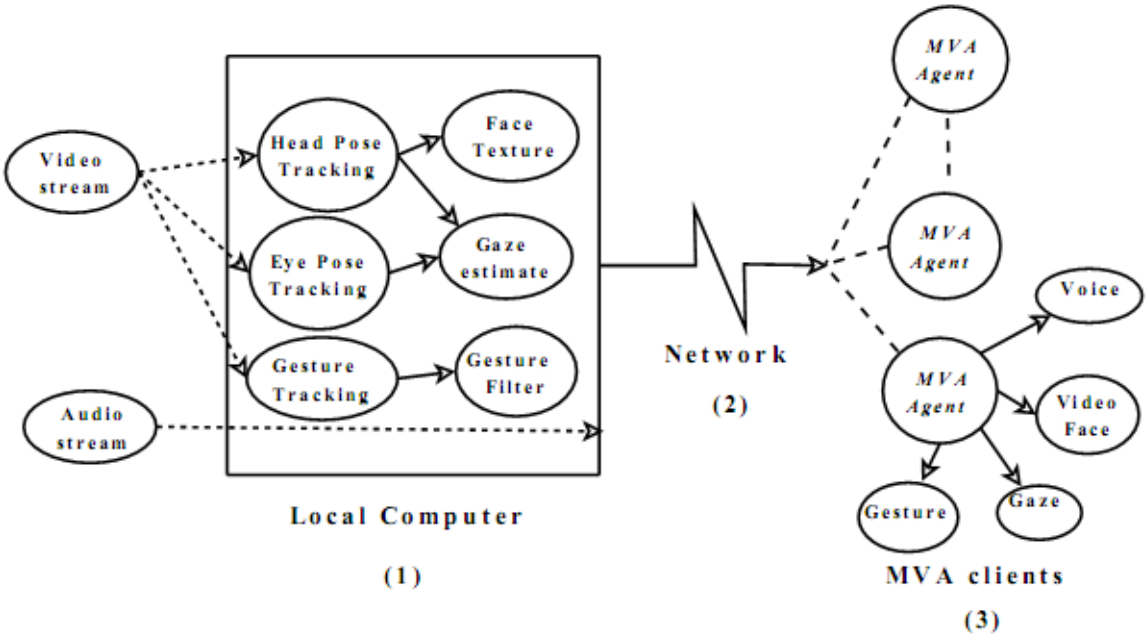


Figure 1. MVA Project flowchart

II.2. Eye Pose Tracking

Three internship subjects were proposed before my arrival, mainly oriented on Head and Eye tracking and processing. Considering my knowledge and current progressions in each part, I choose the "Eye pose tracking" subject. The following is a description of this last one:

"Gaze is an important channel for human-human communication. But gaze estimation from videos is still an open question. Our method is to estimate gaze in another perspective, which is to consider the coordination of head pose and eyes. Our previous developed head pose tracking is efficient to recover 3D head pose. But we haven't touched eye gaze tracking. Thus, this subject is to develop a simple eye tracking solution and then cooperate with the tracked head pose to estimate gaze."

From this sum up, we can extract main research directions:

- *Implementation of a simple gaze tracking system:* The first and the most important part which will be based on previous research results in eye tracking.

- *Integration with Head pose tracking (WANG Haibo):* Combining eye pose tracking and head pose tracking could significantly improve gaze estimation (Figure 2). Indeed, gazing on an object or a person is usually accomplished by rotating both eyes and head in the same direction.

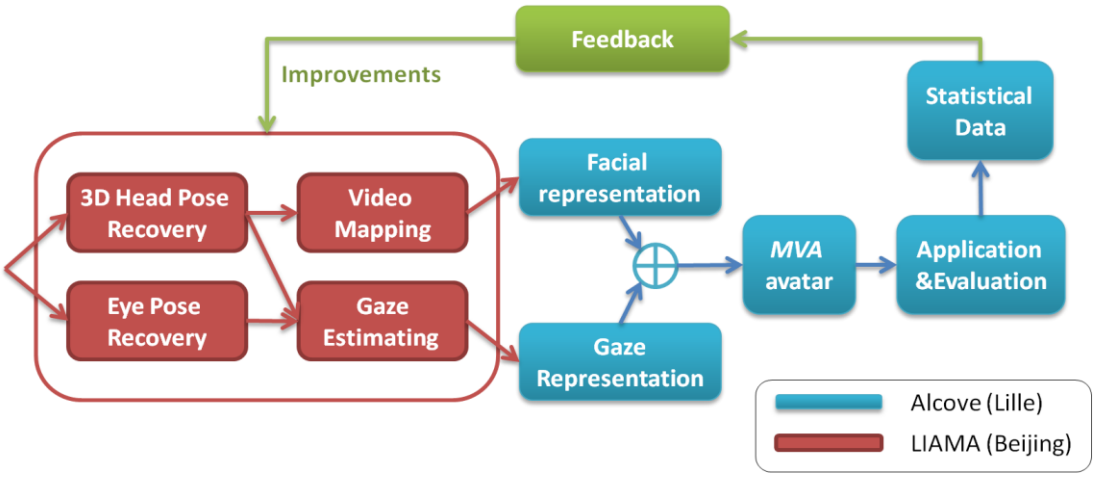


Figure 2. Head and Eye Pose Tracking framework

Figure 3 presents the initial status of Head and Eye Pose Tracking parts before my arrival.

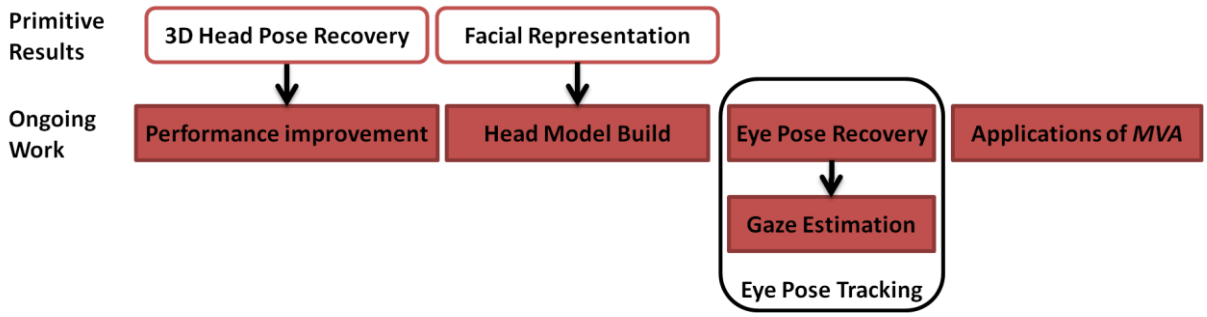


Figure 3. Status of Head and Eye Pose Tracking part

II.3. Specifications

Considering the subject and the current progression of MVA Team in eye pose tracking, no precise specifications were defined. In this way, I was totally free regarding the choice of the best solutions to combine in order to reach my objectives. Only one mandatory point was considered, the imaging system must be composed of only one camera producing single two-dimensional images. In this way, a 3,2 Megapixels web-camera was chosen to have images with a sufficient resolution.

Furthermore, C++ language and Windows based development platform were retained to ease integration of my part with ongoing works.

III. Technologies involved

First of all it is required to differentiate eye tracking and gaze estimation despite the fact that this two notions are intimately linked. Gaze estimation is the physical evaluation of the gaze point considering head position, eyes position or both and an object to gaze at. Eye tracking restraints itself to build an internal model representing eyes movements by following the movement of particular eyes features. Therefore, development of an efficient eyes tracking system is generally the first step to achieve efficient and robust gaze estimation.

III.1. State of the art in gaze estimation and eye tracking

Numerous techniques have been proposed for gaze estimation or eye tracking. But, earlier eye trackers are particularly intrusive because they required physical contacts with the user, such as placing a reflective white dot directly onto the eye or by attaching electrodes around it. In addition, most of those technologies also require the user's head to be motionless during tracking.

III.1.1. Eye-pose tracking

With the rapid technological progression in both of video cameras and microcomputers, gaze tracking technologies based on the digital video analysis of eye movements has been widely explored. Since it does not require anything attached to the user. The goal is to allow user unrestricted freedom of movement (Head movement). Major challenges in developing real-time gaze tracking systems include: tracking speed, accuracy, and robustness. They fall into two categories:

- *Analytical approaches*
- *Neural network approaches*

In an analytical approach, the gaze tracking accuracy relies on the resolution of eye images. However if users movement freedom has to be preserved, low resolution eye images have to be used, or high resolution full-frame equipment have to be preferred. Low resolution eye images are a major obstacle in achieving a high gaze tracking accuracy in an analytical approach. That is why most part of high precision non-intrusive gaze tracking results have been achieved only through neural network methods, which can use the intensity information of the low-resolution eye images. However, neural network based methods present a lack of robustness against environmental changes and are heavy to compute.

Among actual analytical methods, several are based on Scale-Invariant Feature Transform algorithm (SIFT) or Haar like Features detection. Those two methods on which relies my implementation will be detailed in this document.

III.1.2. Gaze estimation

Various techniques have been proposed to perform the gaze estimation based on eye images captured by video cameras. However, as it was said, most available gaze trackers have two characteristics that prevent them from being widely used: first, they must often be calibrated repeatedly for each individual; second, they still have low tolerance to head movements. This last characteristic explains clearly the current orientation of MVA project: to achieve efficient gaze estimation while allowing user unrestricted freedom of movement by combining independent head and eye pose tracking systems. Most of the non-intrusive gaze estimation techniques can be classified into two groups:

- *2D mapping-based gaze estimation method*
- *Direct 3D gaze estimation method*

III.1.2.1. 2D Mapping-Based Gaze Estimation

The gaze is estimated from a gaze mapping function by inputting a set of 2D eye movement features extracted from eye images, without knowing the 3D direction of the gaze. This gaze mapping function allows the establishment of a relationship between the gaze and extracted 2D eye movement features, and is obtained using a calibration stage for each person. Unfortunately, the gaze mapping function is very sensitive to head motion because extracted 2D eye movement features have significant variation with head position.

Pupil Center Cornea Reflection (PCCR) technique is the most commonly used 2D mapping-based approach for gaze tracking. The angle of the visual axis is calculated by tracking the relative position of the pupil center and a speck of light reflected from the cornea, technically known as the “glint” as shown in Figure 4. The calculation is possible only by knowing several constants computed during the calibration stage.

Figure 4. PCCR glint example

Several systems have been built based on the PCCR technique. Most of those systems show that if users keep their heads fixed, or restrict head motion, very high accuracy can be achieved in tracking results. On the contrary, when the head moves away from the original position where the user performed the gaze calibration, the accuracy of these gaze tracking systems drops dramatically.

III.1.2.2. Direct 3D Gaze Estimation

The 3D direction of the gaze is estimated so that the gaze point can be obtained by simply intersecting it with the scene. Therefore, how to estimate precisely the 3D gaze direction remains an important issue for most of these method-based techniques. Several attempts have been proposed to estimate the gaze from eye images, for example:

- Jie Zhu and Jie Yang presented an analytical method that uses edges and local patterns to obtain detection of eye features with subpixel precision. They developed two algorithms that can robustly detect inner eyes corners and iris edges, both with subpixel accuracy. Then a simple estimation scheme is used to calculate the gaze direction from the tracked features.
- Jian-Gang Wang, Eric Sung and Ronda VenkateswarluIn used the observation that the iris contour while being a circle in 3D is perspectively an ellipse in the image. The gaze, defined as the normal to the plane of the iris circle, can be estimated from the ellipse/circle correspondence. However, it will result in two possible solutions of the normal. Therefore, they proposed a “one-circle” algorithm to disambiguate solutions.

III.2. Intel OpenCV library

OpenCV library is a cross-platform library developed by Intel which mainly aimed at real time computer vision. This library implements a consequent number of images processing functions and can be modified to implement custom processing methods. OpenCV functions cover a wide panel of domains:

- Human-Computer Interaction (HCI)
- Object identification, segmentation and recognition
- Face and gesture recognition
- Motion tracking
- Etc.

OpenCV is the library used from the beginning of MVA project, and one of the most complete computer vision libraries nowadays. Thereby, it’s fully adapted to the realization of an eye pose tracking and gaze estimation system which will use videos and images processing, detection algorithm, etc. Besides, OpenCV in its actual version uses CPU-based computation, but a GPU-based OpenCV is currently under development and could consequently improve computation time without modifying futures implementations.

Implementations describe in this document are based on OpenCV library

IV. Eye pose tracking development

IV.1. Problematic

Existing documentation about eye tracking is very rich and offers a large spectrum of solutions with sometimes very disputable results. A first and nonetheless difficult issue was to choose the most adapted method considering the time whose I disposed to implement it and the fact that only a few among actual methods present relevant results.

Moreover, even if performances aren't an important requirement for my system, the frame rate has to reach at less a sufficient value, thus the tracking algorithm mustn't be too much time-consuming. Unfortunately, without an image already focused on eyes region, eye tracking needs several steps before allowing tracking strictly speaking: face detection, single or two-eye detection, features detection and extraction, etc. All these intermediary steps can be extremely time-consuming depending of each chosen algorithm. Thus, to sustain a decent frame rate, every intermediary steps have to be considered in addition to the tracking itself.

As it was said, literature is very prolific on the subject, but most part of the documentation I have found is based on multi-camera and therefore multi-imaging systems in order to improve the quality and the precision of features detection and tracking. Tracking from a single image is a bit more difficult in the way that no three-dimensional data can be extrapolated from multi-images. Solutions exist, but they are of course more limited even if some algorithms are able to achieve very efficient tracking at high computation cost.

The last issue comes from the integration of independent eye pose tracking system with head pose tracking to allow efficient gaze estimation while enabling the user to reasonably move his head. This computer vision domain hasn't been well explored until now, but few papers including one proposed to publication by WANG Haibo exist on the topic. In this way, possible correlations and constraints in such a combination still have to be defined.

IV.2. Solutions

Four of what seemed efficient and robust solutions considering their results and possible implementations were first chosen:

- *One-circle algorithm* (Jian-Gang Wang, Eric Sung, Ronda VenkateswarluIn): This algorithm is based on the fact that it's possible to infer analytically the supporting plane on which a circle lies by observing its perspective projection. Then, a normal to this plane can be extracted and is assumed to be the gaze of the user in a 3D scene.

- *Subpixel method* (Jie Zhu, Jie Yang): In this method, two subpixel tracking systems have been implemented to track respectively eyes corners and iris positions at subpixel precision. Knowing these positions and some constants defined during a calibration step, it is possible to estimate gaze direction with the following equation.

$$\alpha = \alpha_1 + \frac{x - x_1}{x_2 - x_1}(\alpha_2 - \alpha_1) \qquad \beta = \beta_1 + \frac{y - y_1}{y_2 - y_1}(\beta_2 - \beta_1)$$

  Where $\{(\alpha_1, \beta_1), (x_1, y_1)\}$ and $\{(\alpha_2, \beta_2), (x_2, y_2)\}$ are respectively gaze angle and corner-iris vectors for calibration points P1 and P2.

- *Scale-Invariant Feature Tracking – SIFT* (David Lowe): SIFT uses features which are locals and based on the appearance of the object at particular interest points. Those features are invariant to image scale, rotation, changes in illumination, noise, occlusion and minor changes in viewpoint. Moreover, they are highly distinctive and relatively easy to extract. SIFT will be detailed along next parts because it offers a robust tracking system due to its invariance properties.

- *Active Appearance Model - AAM* (Tim Cootes): AAM aims to locate variable non-rigid objects with distinct features in digital images through the use of a generative deformable model. Characteristics of the object are learned through statistical analysis of shape and grey-level variation in a training set.

AAM won't be detailed in coming parts due to an important lack of documentation, particularly examples. Thus, I have not used it, despite the fact that it seems to be very efficient for features tracking with its statistical model. The one-circle algorithm in spite of its evident simplicity eludes the problem of iris detection and so doesn't describe possible implementations of a detection method.

Finally, I have first chosen a fifth algorithm built on subpixel tracking idea to use a vector between eye inner-corner and iris center to estimate gaze. Those particular points can be detected through openCV functions using for example Haar classifiers. But, facing a cruel lack of precision, I developed a new tracking approach based on SIFT which is far more precise but unstable. The remainder of this document describes those two approaches.

V. OpenCV & Haar based gaze tracking

V.1. Overview

V.1.1. Tracking system

Most part of the tracking algorithm developed in this first approach is based on Haar-like object detector. This object detector has been proposed by Paul Viola and improved by Rainer Lienhart. It allows with a classifier trained with sample views of a particular object to detect this object in a whole image. This concept will be detailed in next part.

By applying it in a multi-step detection approach, it's possible to detect precisely the eye region by first recognizing the face and then by setting a region of interest on it. It needs of course the correct Haar cascade descriptors for front face and two-eye region.

The following algorithm uses this multi-step approach (Figure 5) to extract two-eye region from a video sequence containing a face. Then, by setting a region of interest on two-eye region, it's possible to extract what will be called the "gaze vector" (vector between a corner and the iris-center of each eye) by combining one more detection step on each eye with Haar-like object detector and openCV methods. Gaze vector is essential here because its accuracy is crucial to achieve efficient gaze estimation. In this way, the tracking system is focused on its detection.

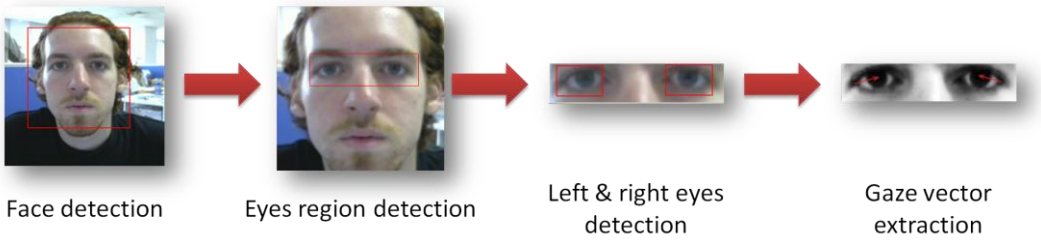


Figure 5. Tracking system steps

The choice of features like iris centers and eye corners is justified by requirements to have a fixed point (corner) from which estimate the movement of a dynamic point (iris-center). Moreover, iris center and generally eye corners are very stable points easy to detect.

V.1.2. Gaze estimation

For gaze estimation, I have chosen a very simple representation. The screen is treated as an n-order square matrix and is equally divided in 2n areas. The higher will be n value, the thinner will be the screen division.

In this matrix, each area is associated with a gaze vector extracted either directly by the tracking system during the calibration stage or extrapolated after this stage. At least four

gaze vectors are estimated from calibration and placed in matrix edges. Those vectors combined with an extrapolation mechanism allow filling the matrix with intermediary gaze vectors for each area. Thus, a matrix of order 2 is only composed of vectors extracted from the calibration step while a superior order matrix is composed of four calibration vectors in its edges and extrapolated vectors elsewhere (Figure 6).

Next, for each gaze vector computed by the tracking system, the gaze estimation part compares this vector with each one in the matrix. The best fitting vector is considered to be the area on the screen where the user is looking. Clearly, the thinner the division will be and so the higher will be n value, the more precise will be the gaze estimation assuming an accurate tracking.

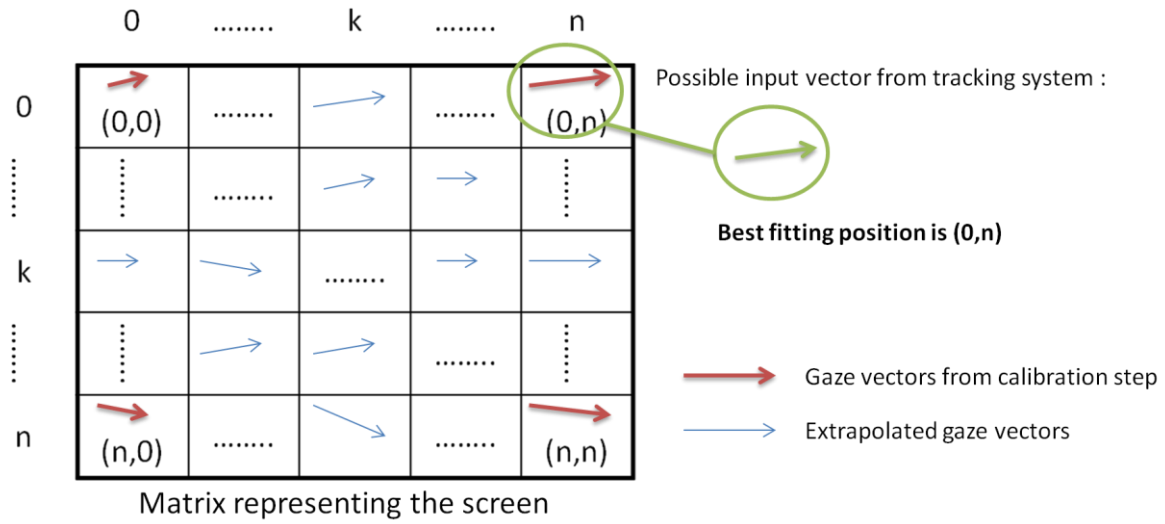


Figure 6. Gaze estimation model

V.2. Haar-like features detection

The value of an image pixel is relatively poor regarding provided information like luminance or color. In order to detect an object, we need detectors which can give more global information about an image.

V.2.1. Weak Haar-classifiers

Weak classifier refers to entity which taken alone cannot find an object in an image but can only give global information. Weak Haar-classifiers belongs to this category because they use changes in contrast values between rectangular groups of pixels. Indeed, a weak Haar-classifier is an assembly of white and black rectangles under different forms like in Figure 7. The value of the classifier is then given by the difference between the sum of pixels in white areas and the sum of pixels in black areas:

$$f_i = \sum_{white\ area} r_i - \sum_{black\ area} r_i$$

where f_i is the descriptor value of the classifier and r_i is a pixel value.

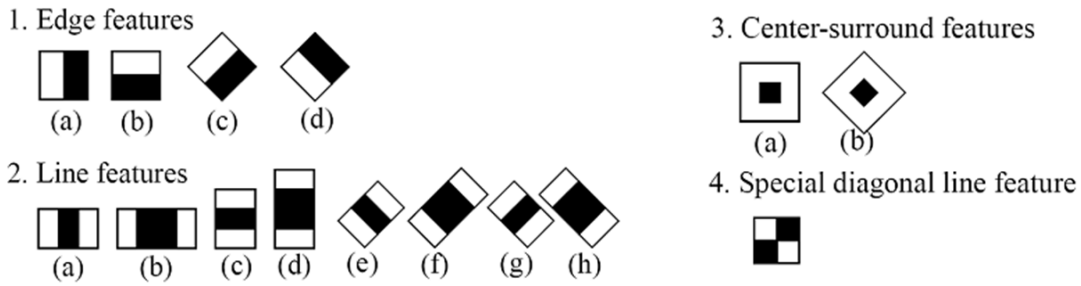


Figure 7. Haar classifiers

A weak Haar-classifier is generally estimated in a 24x24 pixels window. A weight is associated with each region; white regions have a positive weight as black regions have a negative weight. Main characteristics of a weak Haar-classifier are the following:

- Number of rectangles (2, 3 or 4-rectangles)
- Top-left corner position of each rectangle (x,y)
- Width (w) and height (h) of each rectangle with $0 < x, x + w < W; 0 < y, y + h < H$, where W and H are sizes of the window in which the classifier is computed.
- The weight (Positive or negative) of each rectangle

For a given 24x24 window about 180 000 classifiers can be defined which is of course impractical regarding the required computation time. In this way, and to improve significantly performances of classifiers computation, the integral image concept has been developed.

V.2.2. Integral Image

This concept introduced by Viola (2001) is very simple and time-efficient (computation using a constant time - $o(1)$). The sum of image pixels is estimated one time before using Haar-classifiers with the following equality:

$$ii(x,y) = \sum_{x' \leq x, y' \leq y} i(x', y') \quad where \quad \begin{cases} ii(x,y)\ is\ a\ pixel\ from\ the\ integral\ image \\ i(x', y')\ is\ a\ pixel\ from\ the\ input\ image \end{cases}$$

In this way, the pixel at position (x,y) contains the sum of pixels values in a rectangle whose origin is *(0,0)* with a *x* width and a *y* height like the left side of Figure 8. From this integral image, sum of pixels in any rectangle can be quickly calculated whatever is its position by only considering its four corners coordinates. For example, on the right side of Figure 8, D region can be efficiently computed by using known values of the integral image at positions 1, 2, 3 and 4, namely $D = 3 - (2 + 4) + 1$.

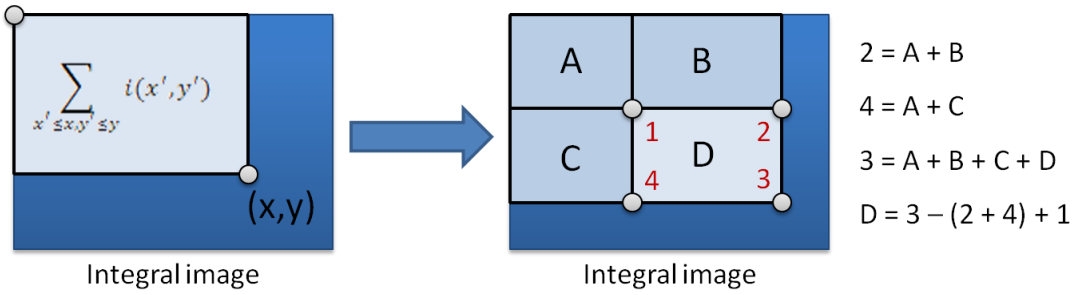


Figure 8. Integral image concept

V.2.3. Strong Haar-classifiers

Weak classifiers used alone aren't enough to determine if an area is a part of an object to detect. For this purpose, it's a real requirement to dispose of more precise classifiers based on group of weak classifiers. Mathematically weak Haar-classifiers $h(x)$ are composed of a descriptor f, an optimal threshold ϑ and a parity p:

$$h_t(x) = \begin{cases} 1 \; if \; p_t . f_t(x) < p_t . \theta_t \\ 0 \; otherwise \end{cases}$$

Using a boosting algorithm like Discrete AdaBoost, it's possible to create a strong classifier $C(x)$ in T iterations selecting weak classifiers $h_t(x)$ weighted by a_t regarding their contributions where:

$$C(x) = \begin{cases} 1 & \sum_{t=1}^{T} \alpha_t . h_t(x) \geq \frac{1}{2} \sum_{t=1}^{T} \propto_t \\ 0 \; otherwise \end{cases} \qquad where \; (\alpha_t = log \frac{1}{\beta_t})$$

In this way, a strong or boosted classifier is only a complex build of weak classifiers based on one of four well known boosting algorithms: Discrete AdaBoost (DA), Real AdaBoost (RAB), LogitBoost (LB) and Gentle AdaBoost (GAB) implemented in OpenCV with many other variants.

V.2.4. Haar cascades

To benefit of a particularly efficient processing, strong classifiers are grouped in cascades which are nothing else than decision trees. Their construction is simple: considering f the false positive rate for a detection step, d the minimal detection rates for a detection step and F_{target} the global rate of accepted false positive; cascade steps are built by training strong classifiers with a boosting algorithm and by adding weak classifiers until detection rates reach targeted values for the current step. Then, steps are added until global rate for the cascade is reached.

$$F = \prod_{i=1}^{K} f_i \quad and \quad D = \prod_{i=1}^{K} d_i \quad where \quad \begin{cases} F: cascade \; gloabal \; false \; positive \; rate \\ D: cascade \; minimal \; detection \; rate \end{cases}$$

V.2.5. Application

After a classifier has been trained, it can be applied to a region of interest in an input image. The classifier returns a "1" if the region is likely to show the object and "0" otherwise, as detailed in $C(x)$ equation. In order to search for the object in the whole image the search window can be moved across the image to check every location using the classifier. The use of a cascade just means that the resultant classifier consists of several simpler classifiers or stages that are applied subsequently to a region of interest until at some stage the candidate is rejected or all the stages are passed.

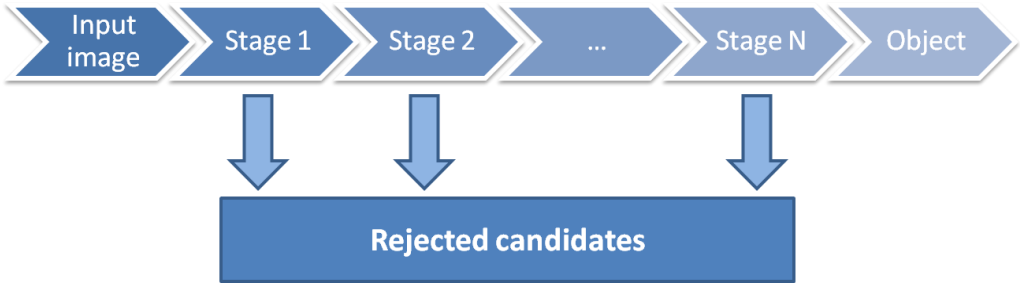


Figure 9. Cascades application

Those cascades are fully used in my algorithm, to detect face, eyes region, and then each eye separately across eyes region. For that purpose, a group of four cascades of classifiers trained specifically for each features has been chosen:

- *Face:* haarcascade_frontalface_alt2 (FA2) – Size: 20×20 - 46 stages
- *Two-eyes:* haarcascade_eyes (EP3) – Size: 35×16 – 19 stages
- *Right eye:* haarcascade_right_eye (RE) – Size: 18×12 – 18 stages
- *Left eye:* haarcascade_left_eye (LE) – Size: 18x12 – 16 stages

Referring to the paper “Face and facial features detection evaluation” published by M. Castrillon-Santana, O. Deniz-Suarez, L. Anton-Canalıs and J. Lorenzo-Navarro those cascades present the best results for their respective features. In addition, certain of those data have been confirmed experimentally, particularly for face detection and two-eye detections. It’s interesting to denote that right and left eye cascades offer far more accurate results when applied directly over eyes region rather than applying them on face region.

V.3. Implementation

V.3.1. Class structure

The implementation of the previously define gaze tracking system is structured around four classes, like detailed in Appendix A.

V.3.1.1. CamCvCapture class

This class is responsible of camera management based on openCV Highgui API. Only two methods are implemented *runCapture()* and *setResolution()*, respectively to acquire an image from the camera, and to set the resolution of the capture peripheral. OpenCV offers two APIs for camera management, CvCam and HighGui.

CvCam and HighGui

The choice of Highgui API rather than CvCam API is justified regarding the conception of CvCam. Image processing with CvCam requires a callback method which is particularly difficult to implement into an object-oriented program due to C++ constraints. Indeed, to work properly the callback method needs to be static and in this way C++ define that all attributes and methods called in this function must be static too, to prevent access to member of a non-instantiate class. As a result, *CamCvCapture* and *Detector* would have been very heavy to implement properly despite the fact that CvCam API offers more possibilities regarding the capture than Highgui. Moreover, callback methods from CvCam are really load-consuming (CPU-load difference of 40 % between HighGui and CvCam for a same task).

As a consequence HighGui has been chosen as default API and because its development isn't finished yet, I had to modify its implementation to add resolution control over the camera. In this way *cvcap_cfw.cpp*, *cvcap.cpp* and *cvcap.h* which contain OpenCV Highgui implementation have been modified and recompiled after adding management of a new parameter: *CV_CAP_PROP_FRAME_WIDTH_*HEIGHT represented as a double value "widthheight" (Example : 640480). Associated processing functions have also been modified to consider this new parameter. Those modifications can be an issue regarding portability of the implementation.

V.3.1.2. Detector class

This class contains an entire set of Haar-cascades based methods for face, eyes and eyes features detection. Its use is really simple; the image to process is by default extracted from the first available acquisition device via *setIplImageFromCam()* call and is put in *m_referenceImage* attribute. This call uses a *CvCamCapture* object for image acquisition. If no acquisition device is available, then the user can manually set an image by calling *setIplImage()* and by using an image converted to a local OpenCV representation *IplImage*.

From the current image stocked in *m_referenceImage* attribute, gaze vectors can be computed as a class internal *std::vector<CvPoint>* structure by calling *extractEye()* method. This structure is a vector composed of openCV *CvPoint* structures which contains two *int* values *x* and *y* (Figure 10). Then by calling *getVector()* method the global gaze vector of current image can be obtained.

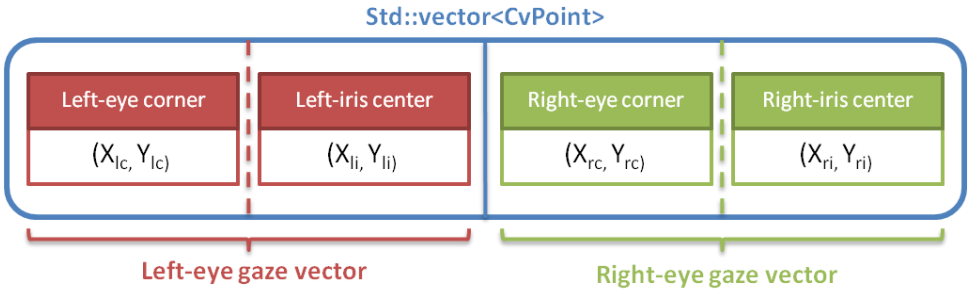


Figure 10. *extractEye()* method result – gaze vector

V.3.1.3. Matrix class

Matrix class is a representation of the screen as described in V.1.2. In this way, the screen is represented by *m_matrixT* attributes as an assembly of Vector structure containing *CvPoint,* to finally obtain a three dimensional matrix defined as:

std::vector <std::vector<std::vector<CvPoint>>> m_matrixT

A vector structure has been preferred rather than a dynamic allocate *CvPoint* matrix because of the simple utilization of vectors far more easy to manage than dynamic matrix, even in the case of multiple dimensions vectors. Each element of *m_matrixT* contains a gaze vector (Figure 10) extracted by a *Detector* object or computed from other vectors.

Before using the matrix, a calibration stage is required to initialize the matrix with at least four gaze vectors – one for each corner – by using *Detector* object and *setVectorAt()* method. Then, for each gaze vector passed as input parameter to *selectArea()* method, the best fitting area for the vector is returned and is assumed to be the area looked by the user.

Vector comparison

The selection mechanism to estimate gaze is based on vector comparison. For each gaze vector in the matrix, associated affine function coefficients from the gaze vector and the input vector are estimated and subtracted. Relative angle between the two vectors is also computed. At this stage, the gaze vector with the minimal slope difference (i.e. near zero) is selected as the best fitting vector. If a better vector is found only by comparing slopes differences, then it will become the new selected one. In the particular case where only one coefficient is better among two gaze vectors, the vector with the minimal relative angle is selected.

In Figure 11, we consider that (1), (2) and (3) are vectors from the matrix, and the red one is an input gaze vector. The best fitting vector is first (1). Then, (2) is chosen because of its better angle and despite its slope which isn't exactly inferior to (1). Finally (3) is retained because of its better slope.

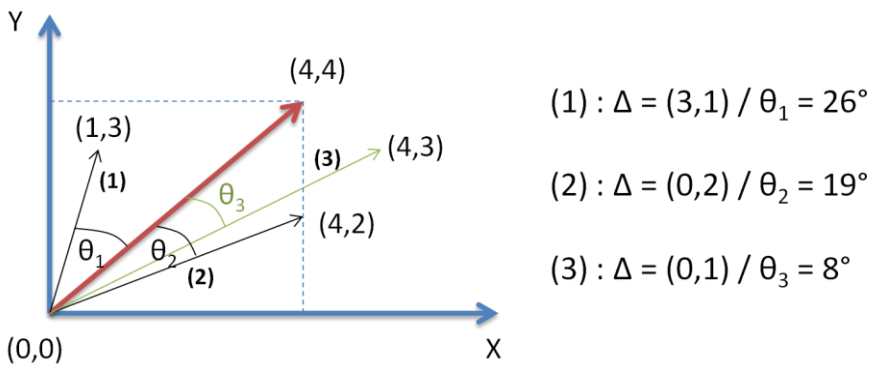


Figure 11. Matrix class vector comparison

Most methods in *Matrix* are replicated with a "TE" suffix which means "Two-eyes". Indeed, "TE" marked methods are exactly the same that not marked methods except that they use the full gaze vector. In this way, "TE" methods use two eyes of the user to make estimation while not marked method use only one eye (left or right).Of course, structures returned by "TE" methods are more complex because comparison results must be returned for both eyes in one structure.

V.3.1.4. Locator

Locator class is an interface between *Matrix* and *Detector* classes. Calibration steps, images and coordinates acquisition from *Detector* and gaze estimation from *Matrix* are managed by this class. Figure 12 and Figure 13 are sequence diagrams which detailed coordinated operations between *Matrix*, *Detector* and *Locator* objects for each operation mode (calibration or capture). Indeed, only two methods are to be called in Locator:

- *eyesCalibration():* To begin calibration stage and fill Matrix with gaze vectors extracted from calibration process.
- *standardCaptureMode():* To begin the capture and gaze estimation.

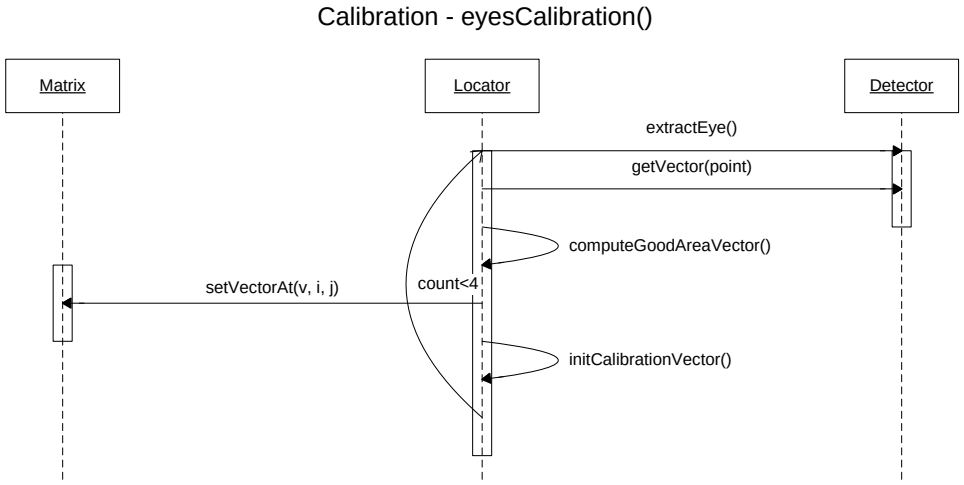


Figure 12. Locator sequence diagram (Calibration mode)

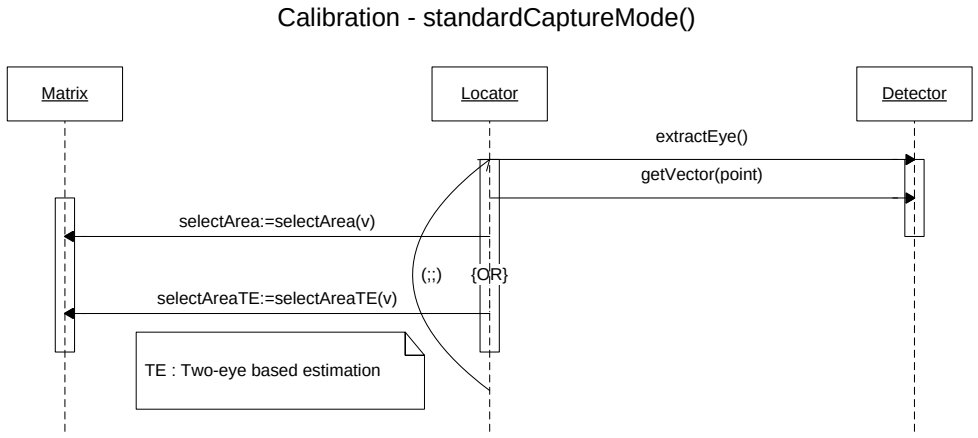


Figure 13. Locator sequence diagram (Capture mode)

Calibration stage

Calibration consists in a step-by-step process during which the user is asked to look each corner of the screen. For each corner, about fifty gaze vectors are captured to limit potential errors using a mean. Each averaged gaze vector is then added to the corresponding corner position in *Matrix* object by *computeGoodAreaVector()* method.

V.3.2. Eye features detection

Iris center and eyes corners detection are based on two different approaches, using respectively Haar cascades and *CvGoodFeatureToTrack()* method from openCV:

- *Iris center:* This feature is detected with specific Haar cascades for left and right eyes. The result is a rectangle region around each eye whose center is assumed to be the iris center. This approximation is really relevant considering characteristics of Haar detection but requires the user to be very close from his camera to detect each eye. Figure 14 (left side) shows a result of this detection: Red rectangles have been detected using Haar cascades dedicated to each eye and the red dot is the assumed position of the iris center.

- *Eye corner:* They are far more complex to detect because there aren't efficient algorithm for this purpose. An OpenCV function dedicated to strong corners detection in a whole image has been chosen: *CvGoodFeaturesToTrack()*. This method computes the covariation matrix of derivatives of every pixel considering an NxN neighborhood. Then, it finds minimum eigenvalues of this matrix following by a non-maxima suppression based on a threshold to define. Finally, the function ensures that all corners found are distant enough one from another. From this point, results have to be discriminated using the knowledge of the most probable position of an eye corner in the image. A possible constraint is based on the known position of iris centers, and the minimal distance between iris-corner couples considering image size.

Figure 14 (right side) shows a result for an eye region focused image with a high tolerance threshold required to detect eye corners (red dots).

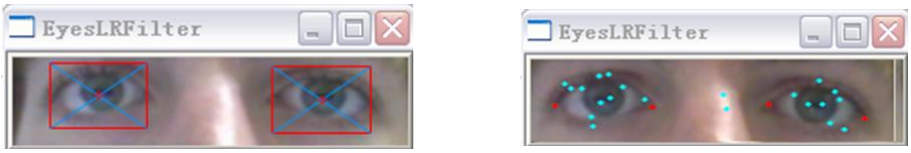


Figure 14. Iris centers and eyes corners detection

V.3.3. Results

Results obtained using previous algorithm aren't very relevant and don't allow to achieve neither a good gaze tracking, nor a good gaze estimation. Indeed, there is a cruel lack of stability in corners detection part. It's in fact particularly difficult to discriminate eye corners among the non-negligible quantity of possible corners returned with a too low threshold. Moreover, when this discrimination is possible, *cvGoodFeatureToTrack()* method is not able to detect a same point between two images, creating each time a different error on a gaze vector for which accuracy is a key point.

In the same way, Haar detection is particularly difficult to use with little object like each eye despite of its performance regarding face and two-eye region detection. As for corners, the stability is a key point to build the gaze vector, and Haar detection cannot offer sufficient one. Detection isn't wrong, but on two successive images which differ slightly, an eye is detected with an error proportional to the variation of image size and so are iris center coordinates. For those reasons, no relevant results are shown here.

V.3.4. Futures

Regarding previous results, it's obvious that an algorithm capable of extracting invariant features is required to create an efficient gaze tracking system. These features have to be detected with a high probability between two successive images which can differ regarding luminosity, scale, rotation, etc. That's why, rather than improving the previous system which is fundamentally deficient, I have chosen to try another algorithm based on invariant features tracking which can be combined with parts of the previous one. Following is a description of this new system.

VI. SIFT Tracking

VI.1. Overview

Considering imprecise results of the previous tracking method, a more adapted tracking system was required which can be adapted with the gaze estimation part. Because of its stability regarding features detection SIFT seemed a particularly efficient algorithm to achieve a good gaze vector detection.

In this part, a SIFT based tracking system is described. It uses exactly the same estimation structure than previous algorithm; hence this part won't be detailed. On the contrary, the tracking part differs radically from the previous one. Tracking is now realized using only one method without considering if a feature is an iris, a corner or something else. Features are then discriminated to extract gaze vector through dedicated recognition methods.

VI.2. Scale Invariant Features Tracking

VI.2.1. Overview

Scale-Invariant Features Tracking (SIFT) is an approach based on transforming image data into scale-invariant coordinates relative to local features. Those features have many properties which make them very suitable for matching in different images. They are invariant to image scaling, rotation and partially to change of illumination. Moreover, due to their well localization in both spatial and frequency domains, the probability of disruption by occlusion, clutter or noise is reduced. Following are the major computation stages used to generate a set of image features:

- *1 - Scale-space extrema detection:* This first stage searches over all scales and image locations for potential interests points using a difference-of-Gaussian function.
- *2 - Keypoint localization:* The stability of each interest point found in first stage is measured to determine if they are possible keypoints.
- *3 - Orientation assignment:* A consistent orientation is assigned to each keypoint based on local image properties (gradient). The keypoint can be represented relative to this orientation and therefore achieve invariance to image rotation.
- *4 - Keypoint descriptor:* The local image gradients are measured at the selected scale in the region around each keypoint. Weighted gradients are then used to build a descriptor. Purpose of this last stage is to achieve invariance to remaining variations such as change in illumination or 3D viewpoint.

SIFT features are extracted from one or more reference images and stored in an appropriate structure. A new image is matched by individually comparing each feature from the new image to this previous structure and by finding candidate matching features, based on Euclidean distance of their features vectors.

VI.2.2. Scale-space extrema detection

As it was said, the goal of this first stage is to identify locations and scales that can be repeatedly assigned under differing views of the same object. Detecting locations that are invariant to scale change of the image can be accomplished by searching for stable features across all possible scales, using a continuous function of scale known as scale space (Witkin, 1983). Koenderink (1984) and Lindeberg (1994) have demonstrated that the only possible function is the Gaussian function.

In this way and to efficiently detect stable potential keypoints, SIFT used a difference-of-Gaussian function. Assuming $D(x, y, \sigma)$ the result image from the difference of two Gaussian functions $G(x, y, k\sigma)$ and $G(x, y, \sigma)$ convolved with an input image $I(x, y)$:

$$D(x, y, \sigma) = \big(G(x, y, k\sigma) - G(x, y, \sigma)\big) * I(x, y)$$

This operation is simply a subtraction between two blurred images using different orders as in Figure 15.

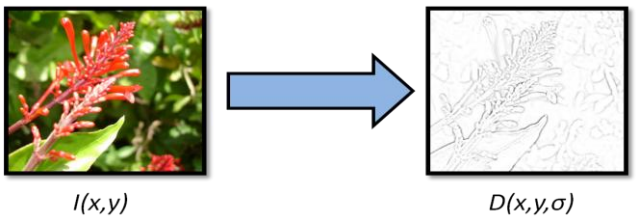


Figure 15. Difference-of-Gaussian

A normalization factor σ^2 is applied to the Laplacian of the Gaussian to have true scale invariance (Lindenberg 1994) giving the following equality:

$$G(x, y, k\sigma) - G(x, y, \sigma) \approx (k - 1)\sigma^2\nabla^2 G$$

Construction of $D(x, y, \sigma)$ uses successive blurred image at different levels to compute difference between each adjacent images. A group of blurred image and resulting subtractions in a same scale level formed an octave. Once a complete octave has been processed, the Gaussian image is resampled to create another octave at a new scale level like in Figure 16.

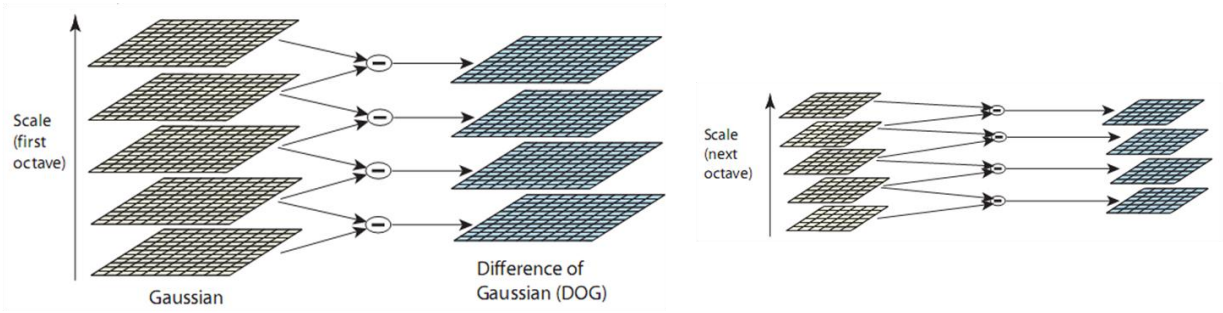


Figure 16. D(x,y,σ) construction

Then begin the local extrema detection by comparing a sample with each of its eight neighbors in the current image and each of its nine neighbors in the scale above and below. A point is assumed to be a local extrema only if it is larger or smaller than all of its neighbors in the scale above and below.

VI.2.3. Keypoint localization

Among every points found in previous stage, it is required to reject low-contrast points which are sensitive to noise and to consider edge responses. Indeed, along edges the difference-of-Gaussian function presents a strong response even if the current potential keypoint is poorly determined or unstable to a small amount of noise. A possible approach uses the Taylor expansion of the space-scale function $D(x, y, \sigma)$ shifted so that the origin is at the sample point:

$$D(\boldsymbol{x}) = D + \frac{\partial D^T}{\partial \boldsymbol{x}}\boldsymbol{x} + \frac{1}{2}x^T\frac{\partial^2 D}{\partial \boldsymbol{x}^2}\boldsymbol{x}$$

Where D and its derivatives are evaluated at the sample point and $\boldsymbol{x} = (x, y, \sigma)^T$ is the offset from this point. The location of the extremum $\hat{\boldsymbol{x}}$ is determined by taking the derivative of this function with respect to $\boldsymbol{x}$ and setting it to zero, giving:

$$\hat{\boldsymbol{x}} = -\frac{\partial^2 D^{-1}}{\partial \boldsymbol{x}^2}\frac{\partial D}{\partial \boldsymbol{x}}$$

If the offset $\hat{\boldsymbol{x}}$ is larger than 0.5 in any dimension, then it means that the extremum lies closer to a different sample point. In this case, the sample point is changed and the interpolation performed on this new point.

VI.2.4. Orientation assignment and local descriptor

The scale of the keypoint is used to select the Gaussian smoothed image L, with the closest scale. For each image sample L(x, y) at this scale, the gradient magnitude m(x, y) and orientation θ(x, y) are pre-computed. An histogram is then built using gradient orientation of samples around the keypoint weighted by their amplitude and a circular Gaussian window. This histogram is composed of 36 possible directions covering 360° with peaks corresponding to dominant directions of local gradients. A keypoint is build with the

orientation of the highest peak in the histogram and new keypoints are created for each peak reaching at less 80% of the value of the highest one.

A keypoint descriptor is computed using the gradient magnitude and orientation weighted by a Gaussian circular window at each sample point in a region around the keypoint location, as shown on the left of Figure 17. Samples are then accumulated into orientation histograms summarizing the contents over 2x2 descriptor array computed from 4x4 set of samples, as shown on the right. The length of each arrow corresponds to the sum of the gradient magnitudes near that direction within the region.

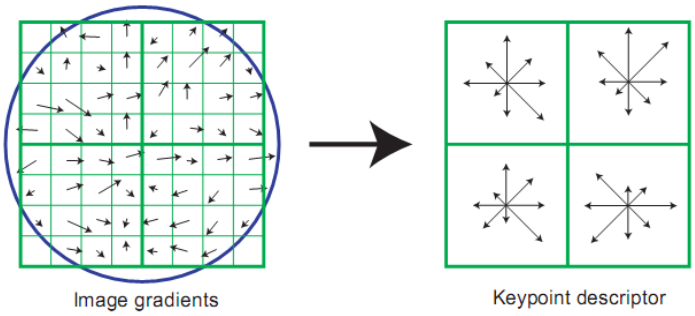


Figure 17. SIFT keypoint orientation assignment

Therefore a descriptor is formed from a vector containing the values of all the orientation histogram entries, corresponding to the lengths of the arrows on the right side of Figure 17. Descriptors are the elements which are compared and matched between two images during tracking.

VI.2.5. Application

By default, SIFT implementation detects only “interesting” features which presents good invariance properties. Most of the time, iris-centers and eyes corners are part of those particular features, among others. It is important to know than on a classical non-focused image from a camera using a 640x480 pixels resolution, SIFT algorithm is able to detect more than 1000 features which is extremely time consuming, without considering the tracking.

By focusing an image on the face, it is possible to reduce this features number to less than 600 features. The result is more probing for an image limited to the eyes region where SIFT generally found less than 30 features. In this way, it’s obvious that SIFT detection must be directly applied over an eye focused image to efficiently detect interesting eyes features. This focus can notably be efficiently realized using Haar cascades trained for face and eyes detection. In this way, exploitation of previous *Detector* objet is relevant because it can dramatically improve computation-time by adding only a small detection stage to set a region of interest on eyes region.

VI.3. Implementation

As it was said, the implementation of the SIFT Tracking system uses most part of previous developments, i.e. *Matrix*, *Detector* and *CamCvCapture* classes. Of course, old *Locator* class is replaced by *SiftCommand* class which is an interface between C implementation of Sift, other C++ objects and *SiftGazeEstimator* which pilots calibration and detection stage.

VI.3.1. Class structure

The implementation of the previously define gaze tracking system is structured around five classes, like detailed in Appendix B. An overview of the entire process is available in Appendix C.

VI.3.1.1. SiftCommand

SIFT implementation uses a complex assembly of C modules visible in the following list. *SiftCommand* class has been created in order to simplify SIFT calls for tracking and features management.

- Kdtree: Functions and structures for maintaining a k-d tree database of features.
- Imgfeatures: Functions and structures for image features.
- Minpq: Functions and structures for implementing a minimizing priority queue.
- Sift: Detection of SIFT image features.
- Utils: Utility functions.
- Xform: Functions for computing transforms from image feature correspondences.

SiftCommand object must be initialized by calling *initSiftTracking()* method which set a reference image - preferably eyes focused - and computes interest features on this image. It is the base for detection and tracking. From this features set, and due to the fact that SIFT has no means to select eye features by itself, a selection stage is mandatory to limit the features set to four or less elements. This can be accomplished using two methods:

- *manualSelectFeatures():* The user is responsible for selecting interest points among features found by SIFT algorithm directly on an image. The features structure is then rebuilt to consider only his choices. Manual selection uses a callback method for mouse events detection with the same limitations that in V.3.1.1. For this reason, main features structure is always replicated in a static attribute.

- *automatedFeaturesSelection():* A detection algorithm based on openCV *cvHoughCircle()* method and using a part of subpixel algorithm for corners detection is applied to select iris centers and eyes corners. Manual selection is still possible for potential corrections.

Next, common features between two features structures are matched using *siftTracking()* method. For this purpose, an image is needed as an input parameter. Features from this image are extracted using SIFT algorithm and are then compared to the current features structure named *m_referenceFeatures*. For each matching features, an *int* table containing matched points index is updated. At the end of this process, if all features in *m_referenceFeatures* have been matched, then a new structure is created to replace *m_referenceFeatures* with information from the *int* table. This mechanism allows a better robustness rather than always keeping same features. Indeed, despite of its invariance properties, experiences have shown that SIFT tracking becomes ineffective over a certain level of image modifications.

VI.3.1.1. SiftGazeEstimator

SiftGazeEstimator object has the same function than previous *Locator* object, but has been adapted to SIFT tracking. In this way, the calibration is based on four points (each edge of the screen) that the user must look. For each point, the corresponding gaze vector is sent to *Matrix* object. Regarding needs of the user, calibration and detection stages can be realized considering one eye (left or right) or the two eyes.

The gaze estimation stage is the same than in previous algorithm and uses exactly the same *selectArea()* or *selectAreaTE()* methods.

VI.3.2. Automatic features detection

Because manual selection isn't very relevant, an automatic features detection mechanism relying on openCV detection methods and subpixel algorithm has been implemented:

Iris center detection

Based on *cvHoughCircle()* method which finds circles in grayscale image using some modifications of Hough transform. This method takes two threshold parameters:

- Threshold 1: Number of points which constitute a good circle.
- Threshold 2: Contrast threshold which define a good circle.

The detection uses a set of parameters optimized for eyes detection. In this way, *cvHoughCircle()* method is applied with different parameters (defined experimentally) until exactly two circles are found, which means for considered images that two iris have been found. To avoid the case where two circles are found on one eye, a constraint is applied on the minimal distance between the two circles which must exceed an eye width.

HoughCircle detection is particularly efficient if eyes region has a sufficient resolution. In our case with 640x480 pixels images, iris can't be detected only when user is too far from his camera. In order to avoid such situation, a zoom mechanism is included during image processing to artificially improve iris detection.

Eye corners detection

Following subpixel algorithm directives for corner detection, each eye image is convoluted twice with an adapted filter (Figure 18) using openCV *cvFilter2D()* method.

$$right = \begin{bmatrix} -1 & -1 & -1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \quad left = \begin{bmatrix} 1 & 1 & 1 & -1 & -1 & -1 \\ 1 & 1 & -1 & -1 & -1 & -1 \\ 1 & -1 & -1 & -1 & -1 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure 18. Filters for eye-corner detection

Before the second convolution, coordinates of the point with the maximum intensity are computed and a region of interest is set around this point. Then the second convolution is applied over this region to determine the corner position using the new maximum intensity point.

When two corners have been found, the algorithm browses the current eye features structure computed by SIFT algorithm to find the nearest points from estimated corners. Those points are assumed to be features to track. Contrary to iris center, corners detection isn't really accurate by detecting randomly real corners or nose edges depending of the room luminosity. But considering SIFT features position, real corners are generally the nearest point from detected points.

VI.3.3. SIFT Constraints

Human eyes generally present a perfect symmetry which can induce false detection from SIFT. It's particularly true for corners which can move from one eye to another. For example, a point which has been associated with the left corner can become in another image the right corner superposed to the real one. Here is a specific issue of my algorithm: without constraints on features displacement tracking can become inefficient.

To correct this problem, displacement constraints between two successive images have been implemented. If a feature is detected and matched between two images, this feature is accepted only if its x-axis displacement is inferior to the size of an eye; else this feature is discarded during the matching process. To preserve the scale invariance, an eye size is estimated to represent ¼ of the total image width and so this value is computed each time considering the current size of the image to match.

VI.3.4. Results

Compared to previous results, SIFT Tracking is a really efficient tracking system, in the way that two features are always matched with accuracy allowing a very precise evaluation of the gaze vector for each eye. Consequently, the gaze estimation part gives very good first experimental results. Unfortunately, due to a lack of time I couldn't test this part more accurately. Besides, combined with the automatic detection which is very efficient assuming

that eyes features have been correctly detected by SIFT, it's possible to fully automate the calibration process.

On the contrary, SIFT stability is a real issue here. Indeed, between two successive but not so different images, a feature may not be found. It's particularly problematic for calibration when SIFT is not able to detect every eyes features. Moreover, in spite of their invariance properties, SIFT features aren't resistant to large movements. Considering this last fact, it can be difficult to detect features after a too large user movement without detecting intermediary features positions.

SIFT algorithm is also time-consuming despite different optimizations by using Haar cascade and by limiting features structure size. Figure 19 is a table detailing computation time required by each step of the algorithm. For each part minimum and maximum durations are given. Those durations are fully dependant of image resolution. Following results are given for an input image of the user in 640x480 pixels using a 25 fps frame rate.

	Single-eye tracking	Two-eyes tracking
Face detection (Haar)	70 – 80 ms	70 – 80 ms
Eyes detection (Haar)	16 – 46 ms	16 – 46 ms
Features tracking (SIFT)	60 – 70 ms	94 – 110 ms
Gaze estimation	Negligible	Negligible
Total time	**146 – 186 ms**	**180 – 236 ms**
Frame rate	**6,8 – 5,4 fps**	**5,6 – 4,2 fps**

Figure 19. SIFT tracking performances

This table confirm the computation cost of SIFT which achieve very low frame rates. In the best case, 27% of the acquisition frame rate is conserved which makes the program rather slow, and so which imposed slow movement from the user in order to catch potential intermediary positions of eye features.

VI.3.5. Futures

Due to a dearth of time, I haven't been able to implement advanced functions to improve SIFT stability. One stated idea was to update independently each tracked feature rather than waiting for all tracked features to be found in a single image. In this way, each point can be tracked separately by using the most up-to-date associated feature. Such solution can improve consequently the probability to find features and to have an up-to-date reference structure. Moreover, this solution doesn't significantly increase computation time which is rather interesting regarding current frame rate.

Another improvement regarding the frame rate can be to use GPU implementation of SIFT. Indeed, assuming certain hardware hypothesis just switching between CPU SIFT and GPU SIFT can improve performances by about 25%. But, this change isn't so simple, because the call stack for features tracking has been deeply modified in GPU SIFT which implied deep modifications in *SiftCommand* implementation.

VII. Project feedback

VII.1.Objectives

Two objectives have been initially extracted from the Eye Pose Tracking subject which are for recall:

- Implementation of a simple gaze tracking system.
- Integration with Head pose tracking System (WANG Haibo).

Among these objectives, the first has been fulfilled because SIFT Tracking can be really efficient assuming certain conditions. The integration with head pose tracking has begun several days before the end of my internship and hasn't been finished at this time. The main problem at this level is that the algorithm to adapt matrix representation of the screen dynamically to user head movements isn't implemented yet. In this way, work had still to be done for vectors extrapolation from calibrated vectors when user moves his head.

Considering the time whose I disposed, I have chosen to lead my subject in order to obtain preliminary results rather than achieving high efficient eye pose tracking system. Assuming of course, that my program can be improved later to reach a sufficient efficiency level. For this last reason, I have not tried to improve the first detailed algorithm because of evident methods limitation and because SIFT tracking that I have discovered later is far more precise, robust and flexible. But the one and half month spent on the first algorithm weren't lost because it enabled me to develop a large panel of tools useful for SIFT tracking development. Besides SIFT is currently used in the Head Pose tracking part of the MVA project.

VII.2.Project beginning

The beginning of this project was particularly difficult because of a dearth of knowledge in computer vision. As a consequence, I needed a long adaptation phase to acquire sufficient knowledge in order to have a clear vision on the subject and its implications.

Added to this, the subject quoted in part II.2 is particularly imprecise regarding the final realization. No technologic choices have been made, surely it's an advantage for a "from scratch" development, but a counterpart is clearly the large diversity of existing algorithm with doubtful results among which a choice must be done. Not that every algorithm are bad extracted from their experimental frameworks, but it exists hundred of papers on the subject. Each paper based on a particular mathematical approach which has to be apprehended. This last fact was particularly true for such a complex algorithm like SIFT.

VII.3.Realization

Some technologic choices were evident, it's the case for openCV API which is nowadays the most developed API in computer vision, or SIFT which clearly appears as the best choice for what I was looking to do. But those libraries have quickly shown several limitations.

First, considering callback methods utilization, it's now clear that they aren't made to be easily integrated in an object-oriented program. Having done this development choice for simplicity and portability purposes, I have spent a lot of time in circumventing this weakness. For example, I have to implement static parts in my classes while minimizing impact on the rest of the program because of C++ constraints for static implementations.

Another weakness is the current development state of openCV. Despite a 1.0 marked version, a lot of methods or constants aren't defined yet but are well documented. It was the case for camera resolution adjustment through Highgui API which I assumed could be easily done regarding the documentation. As a consequence, and because I needed higher resolution, I have to recompile a part of openCV to allow this change.

Because of a lack of method, I also have important issues regarding the management of memory leeks caused by particular structures used by openCV, namely images which are often replicated to avoid alteration during processing. Those memory leeks have sometimes reached more than 4 MB/s which is of course unsustainable.

Except those facts, development stage strictly speaking wasn't difficult because it only consists in implementing interfaces between existing components and adding constraints to limit possible results and errors. In this way, this internship was clearly more engineering than real research.

Bibliography

Research papers

[1] Jie Zhu , Jie Yang, Subpixel Eye Gaze Tracking, *Proceedings of the Fifth IEEE International Conference on Automatic Face and Gesture Recognition*, 131-137, 2002.

[2] Jian-Gang Wang, Eric Sung, Study on eye gaze estimation, *IEEE Transactions on Systems, Man, and Cybernetics*, Part B 32(3), 332-350, 2002.

[3] Paul Viola, Michael Jones, Rapid Object Detection using a Boosted Cascade of Simple Features, *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Vol. 1, 511-518, 2001.

[4] David G. Lowe, Distinctive image features from scale-invariant keypoints, *International Journal of Computer Vision*, 60, 2, 91-110, 2004.

[5] Zhiwei Zhu, Qiang Ji, IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Volume 1, 918 – 923, 2005.

[6] Paul Kuo, John Hannah, An improved eye feature extraction algorithm based on deformable templates, *International Conference on image processing*, Vol. 2, 2005.

[7] M. Castrillon-Santana, O. Deniz-Suarez, L. Anton-Canalis and J. Lorenzo-Navarro, Face and Facial Feature Detection Evaluation, *International Conference on Computer Vision Theory and Applications*, 2008.

[8] Intel Corporation – Software and Solutions Group. OpenCV Object Detection: Theory and Practice [online]. Available on: http://fsa.ia.ac.cn/files/OpenCV_FaceDetection_June10.pdf (04.2008).

Documents

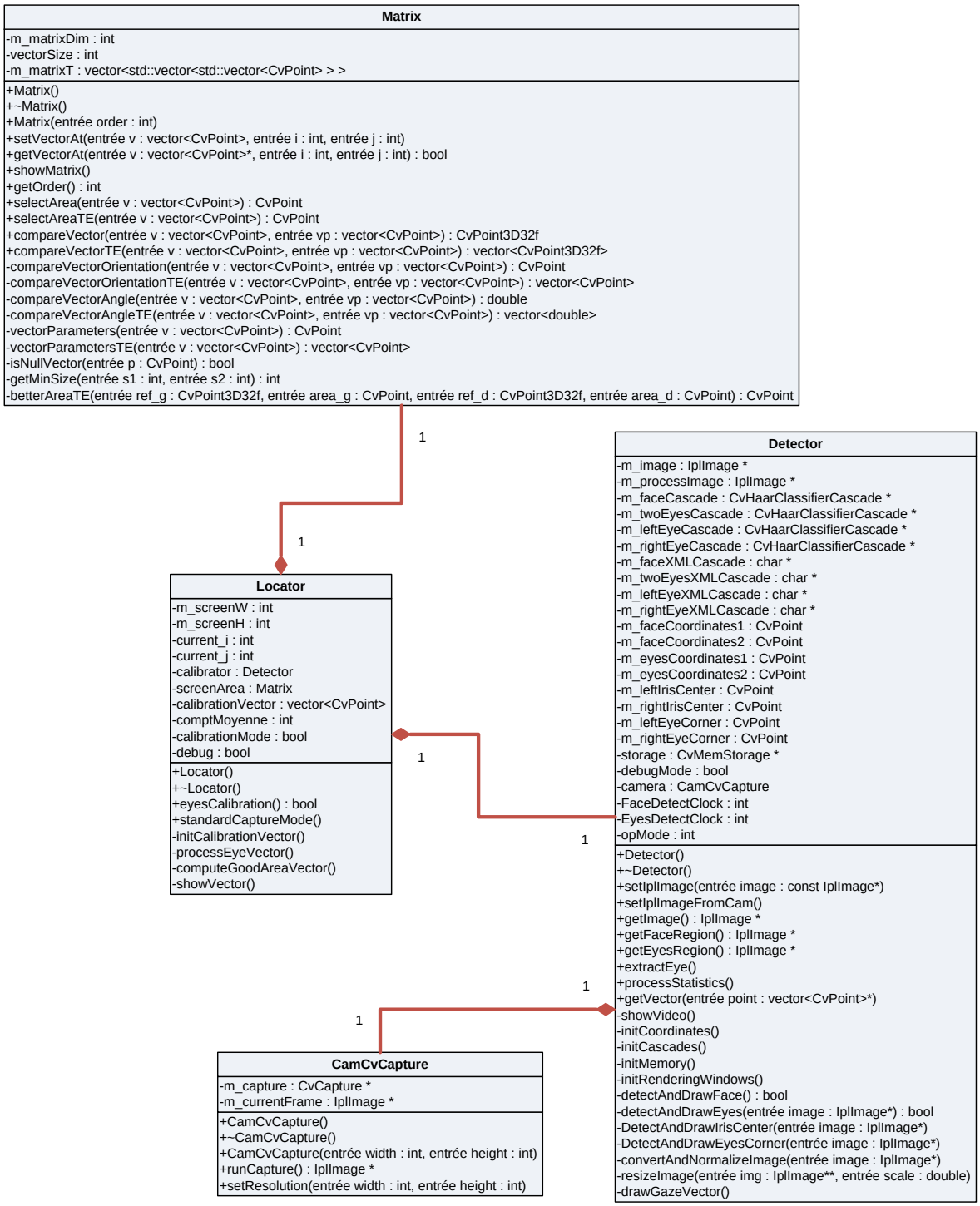
http://opencvlibrary.sourceforge.net/

Reference site for openCV library

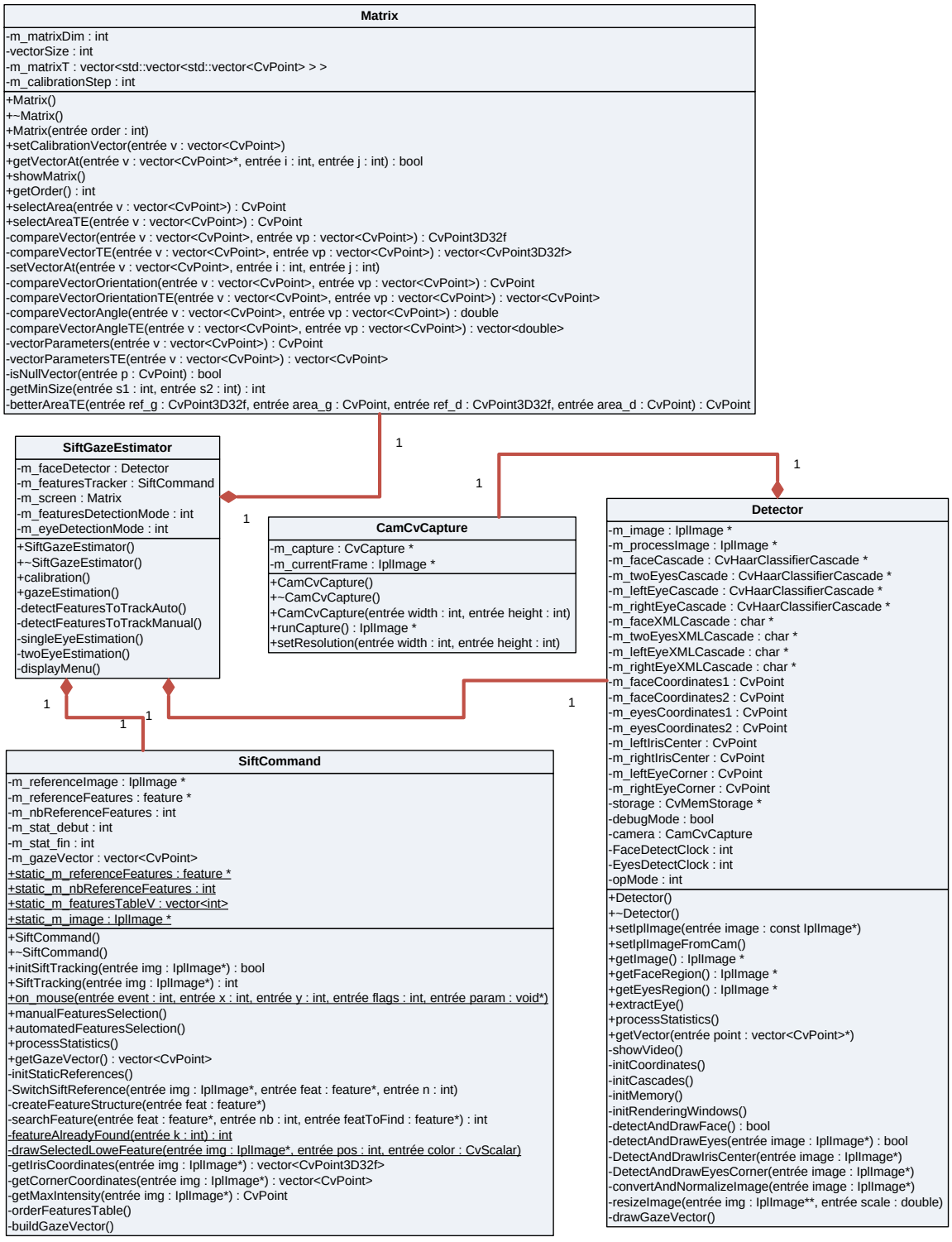
http://tech.groups.yahoo.com/group/OpenCV/

A very active OpenCV Forum at Yahoo Groups

Appendix A. Haar-based implementation



Appendix B. Sift Tracking implementation



Appendix C.Sift tracking global diagram