UNIVERSITÉ DE FRIBOURG SUISSE
UNIVERSITÄT FREIBURG SCHWEIZ

Ergonomic Gestures Recognition

Tom Forrer

January 2011

Thesis supervisors:

Prof. Dr. Rolf Ingold & Dr. Denis Lalanne
DIVA Group (Document, Image and Voice Analysis)

Department of Informatics - Master Project Report

Département d'Informatique - Departement für Informatik • Université de Fribourg - Universität Freiburg Boulevard de Pérolles 90 • 1700 Fribourg • Switzerland

phone +41 (26) 300 84 65 fax +41 (26) 300 97 31 Diuf-secr-pe@unifr.ch http://diuf.unifr.ch

Abstract

This master thesis proposes the use of ergonomic hand gestures for application control. Hand gestures recognition systems often have the disadvantage of being fatiguing over a prolonged period of use. For a practical use for application control the gestures have to share the same ergonomic features of a keyboard or a mouse: limited action space, wrist or arm support, precision and comfort.

This project presents a vision-based architecture where these ergonomic gestures can be recognized. Following a general overview of gesture recognition phases and techniques, this architecture implements the phases of tracking, model mapping, training and classification with the emphasis on real-time execution. The tracking module is implemented using the high-performance OpenCV library, whereas the training and classification of hand postures uses the dlib C++ machine learning functions.

To limit the scope of the project, color-marked gloves will be used to aid the tracking of the finger positions. Also three one-hand gestures are selected for their suitability for gestural application control: the pointing gesture, the zooming gesture (pinching motion with index and thumb) and the horizontal swiping gesture. These gestures are decomposed into their key postures. For the recognition of hand postures, the tracked finger blobs are mapped onto an abstract hand model which can be classified using a one-versus-one multiclass ν-Support Vector Machine (ν-SVM). The decision function of the ν-SVM is trained through the collection of hand model samples from annotated recordings containing the three gestures.

In a validation a small demonstration program is implemented using the gesture recognition architecture. This demonstration program shows that the gesture recognition architecture works with reaction delays under 0.4 seconds. It also shows that spatially small-grained aspects of a gesture are still recognized and that these gestures are easy to learn by other users.

Acknowledgments

First of all, thanks to Denis Lalanne, my thesis advisor, for all the support, reviews, tips and advices, the stimulating discussions and for the confidence in my work.

Thanks to Daniel Gremli, Mathias Vettiger, Rafael Libanori and Niklaus Kränzlin for reviewing the report, discussing ideas on computer vision and interface designs.

Thanks to my family, my sisters Jasmin and Lara, my father Martin and my mother Ivana, for their encouragements, support and love during my study time.

Contents

1 Introduction

1.1 Context

In the last two decades, the mouse and the keyboard were the dominant input devices for computer interfaces. Nowadays, new forms of interfaces come up, such as touch interfaces and even touchless interfaces without any additional devices for the user. For simple or short tasks, the pointing finger has been widely accepted, but for prolonged sessions at an computer interface, users still prefer mouses and keyboards. This is due to the *ergonomic features* of these input devices, mainly the arm support and limited action space, but also comfort and precision.

1.2 Ergonomic gestures

A gesture is, according to the definition of Kurtenbach and Hulteen [1], "a motion of the body containing information". There exist several classifications of gestures, but gestural interfaces mainly focus on symbolic and deictic gestures according to the classification of Rimé and Schiaratura [2].

Early attempts of vertical gestural interfaces had a ergonomic problem, later known as the *gorilla arm*. A user is not at ease lifting the arms for a prolonged period, nor is he comfortable at performing specific gestures in a big action space. This project aims at providing a gesture recognition for hand and fingers, in a situation where the forearm is supported at a table or elbow rests in a chair. These conditions limit the action space and focus on more subtle and articulated gestures.

This project will focus on these gestures with ergonomic features for the purpose of controlling an application interface.

1.3 Goals

Within the context of ergonomic gestures this project aims at:

- recognize spatial and temporal aspects of confined and small-grained gestures
- build an architecture capable of **real-time** gesture recognition
- implement demonstration program using this architecture

2 Gesture recognition

Gesture recognition is a complex topic in computer vision, trying to detect and interpret human movement, postures and gestures. Normally vision-based gesture recognition is composed of four steps, namely model initialization, tracking, pose estimation and gesture recognition and classification [3] (see figure 2.1). These steps are further discussed in the following sections.

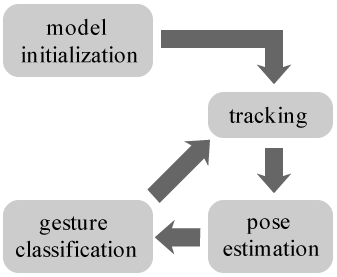


Figure 2.1: Four main steps in gesture recognition

2.1 Tracking

The visual tracking techniques in gesture recognition aim at extracting certain features or visual cues for posture estimation and gesture classification. While most of the feature extraction algorithms work principally on images from a single camera [4], many setups nowadays use stereoscopic vision from two or more cameras [5, 6]. The depth information gained from stereoscopic vision helps to surpass the big challenge of tracking body parts that are self-occluding in a monocular setting. But stereoscopic vision still has the disadvantage of requiring higher processing speed of the hardware [4].

While most tracking techniques described in this section can be applied in other computer vision problems or gesture recognition applications, which focus on other body parts such as arms, faces or torso, they are also relevant for the gesture recognition topic of tracking hands.

In gesture recognition, tracking mainly consists of the two processes of *figure-ground segmentation* and finding *temporal correspondences* [3]. Moeslund et al further classify the figure-ground segmentation into five categories: background subtraction, motion based, appearance based, shape based and depth based segmentation.

2.1.1 Figure-ground segmentation

Background segmentation

The simplest way to separate background from foreground is to take the difference of an empty reference background image and an arbitrary frame. However, this approach is not used, because pixels aren't independent and time-varying background objects should also be considered as background.

In PFinder, Wren et al. [7] represented (see figure 2.2) each pixel by a Gaussian described by a full covariance matrix, where each pixel is updated recursively with the corresponding statistical properties. This allowed for changes in the background scene like waving curtains or moving objects. The pixels belonging to the figure can then be determined by the Mahalanobis distance. But often the part that needs to be tracked is just a hand or the head in order to further refine a search area. In this case a color cue can be obtained by the same methods.

Figure 2.2: Background segmentation in PFinder [7]

Motion based segmentation

Motion based segmentation does not find temporal correspondences but inspects the difference of two consecutive frames, like Sidenbladh did for the figure-ground segmentation of sequences containing a walking figure [8]. Azoz et al. detect time varying edges as a motion cue to separate edges belonging to the figure from edges in the background.

Appearance based segmentation

Using the fact that the appearance of the searched body part can vary from person to person, but is definitively different from the appearance of the background, classifiers can be trained with sets of images containing the appearance of positives and sets with negatives for the background. These classifiers can then be used to detect that type of figure in the scene. Usually, this technique gives no figure-ground segmentation but indicates the location in the scene with a bounding box.

Shape based segmentation

Shape based segmentation often refers to silhouette based segmentation which relies on a good background subtraction. Agarwal and Triggs use silhouettes, because they are easily obtained and because shadowing in the figure, clothing and texture is not encoded in the information. But silhouettes have several disadvantages like occlusion problems or shadow attachments that can distort the shape. Takahashi et al. [9] and Chu and Cohen [10] also extract shape from multiple cameras to construct a visual hull of the figure. But shape based segmentation is not always silhouette based: Mori et al. use shapes obtained by edge detection [11] and in [12] by a normalized cuts segmentation in order to identify *salient* limbs 2.3, which are later assembled into a posture. Azoz et al. use shape filters to detect clusters of colors [13].

Figure 2.3: Normalized cuts segmentation [12]

Depth based segmentation

Using two ore more cameras, a depth map or a complete three-dimensional reconstruction can be estimated. A depth map 2.4 can be computed with a disparity map, which is obtained by a correspondence algorithm. Due to the estimation, occlusions or lens geometry problems, depth maps do not always reflect the 3D scene accurately. In [5], Chien et al. refines the accuracy of the correspondence algorithm with epipolar geometry, using the pointing direction from several images. In [6, 9, 10] silhouettes of the figure from multiple cameras are used to obtain a three-dimensional visual hull with a voxel reconstruction algorithm.

2.1.2 Temporal correspondences

Finding the temporal correspondences resolves ambiguities while tracking objects or figures in an image, given their states in past frames. Temporal correspondences can help, when multiple tracked objects occlude each other, or simply to correctly distinguish two nearby objects.

Figure 2.4: Scene with estimated depth map [14]

Commonly, the task of tracking objects across frames is done with an estimator, which first predicts the location of the object for the next frame, based on previous tracking results and secondly reconciles the measured location of the object with the predictions. A broader class of estimators called condensation algorithms, on which the particle filters are based, can also take occlusions into account by representing multiple possibilities as hypotheses or "particles" to predict future locations of the tracked objects.

Schmidt and Fritsch use a kernel based particle filter [4], which has the advantage over standard particle filters of avoiding a huge number of particles required for tracking the upper body and arms with 14 degrees of freedom.

2.2 Pose estimation

Once the searched limbs or figure features are tracked, pose estimation or model matching adds a layer of abstraction by assembling features into a superstructure which can later be classified. The main goal of pose estimation is to recover an underlying skeletal structure. Moeslund et al. propose a separation of pose estimation methods according to their use of a model [3]:

2.2.1 Model-free

Model-free methods do not use a-priori knowledge of the body configuration. The limb or parts are either assembled dynamically or their constellation can be matched to a pose by comparing them to a database of examples.

Agarwal and Triggs presented a method which recovered a 3D pose from monocular images without using manually labeled examples or templates nor using a body model. The pose is estimated by inferring joint angles from silhouettes with a Relevance Vector Machine (RVM) regression [15]

One of the first model-free pose estimations was PFinder [7]. Wren et al. dynamically added blobs to the pose, either by contour matching or a color splitting process.

Another approach is to dynamically decompose a visual hull into 3D Haarlets (see figure 2.5) using Linear Discriminant Analysis (LDA) [6] or into shape descriptor atoms with a matching pursuit algorithm and Singular Value Decomposition (SVD) resulting in shape descriptors with the largest eigenvalues [10]. Both methods have a key advantage: the atomic shape description and the 3D Haarlets have an additive property (see figure 2.6), drastically reducing the number of possible poses to which the measured visual hull can be matched.

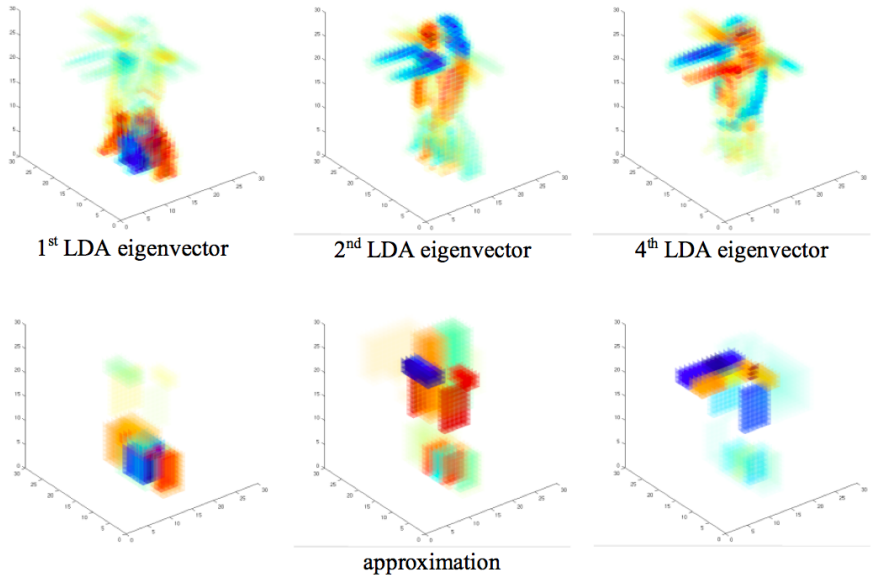


Figure 2.5: 3D Haarlets and their approximation[6]

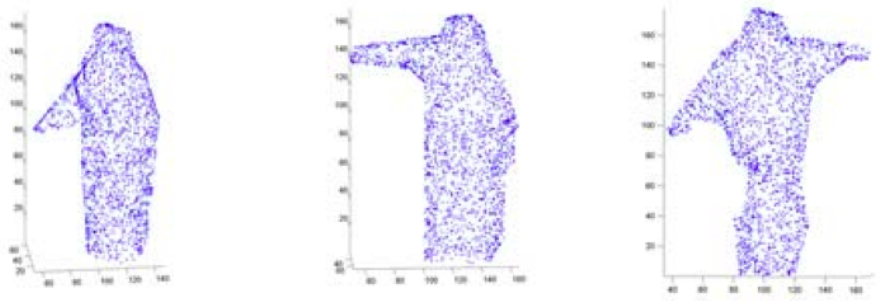


Figure 2.6: Additive property of visual hulls [10]

2.2.2 Indirect model use

This category does not directly map the tracked information onto a model, but rather guides the interpretation. Information like joints, limb length ratios, etc. is labelled into the examples.

Mori et al. use multiple constraints like torso length ratio, adjacency to torso or joints and a-priori body configuration and clothing properties [12]. In another approach Mori et al. recover the pose from labelled silhouette examples by deforming the measured shape and therein the skeletal structure to match the labelled example [11] (see figure 2.7).

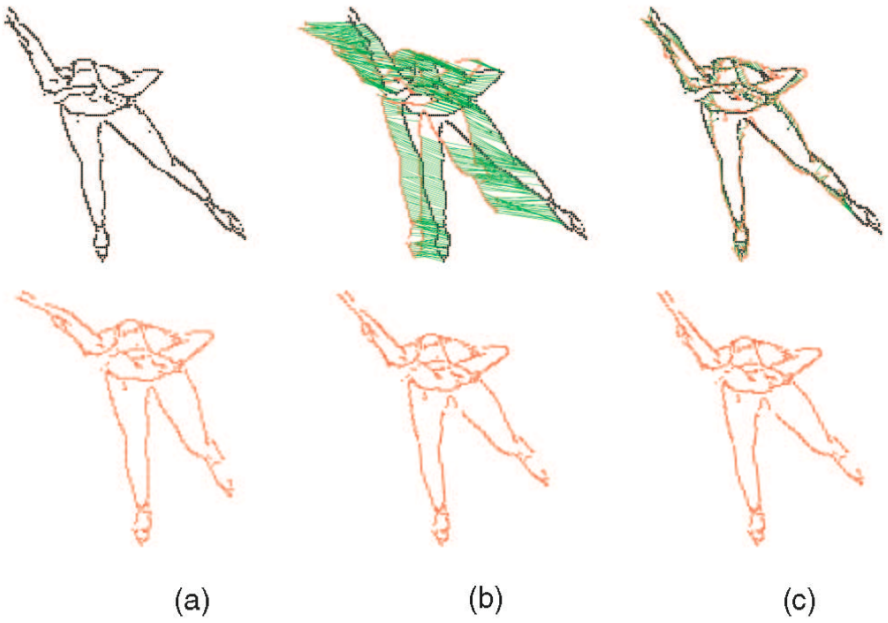


Figure 2.7: Deformable shape context matching [11]

2.2.3 Direct model use

With the direct model approach, the geometric representation of the figure is known beforehand, and the measured data is matched against the model. In newer approaches the model contains human motion constraints to further eliminate ambiguities.

Bray et al. use a smart particle filter (SPF) to fit the tracked results on to a hand model with 15 degrees of freedom and modelling the hand surface in 3D [16] while respecting the physical model constraints.

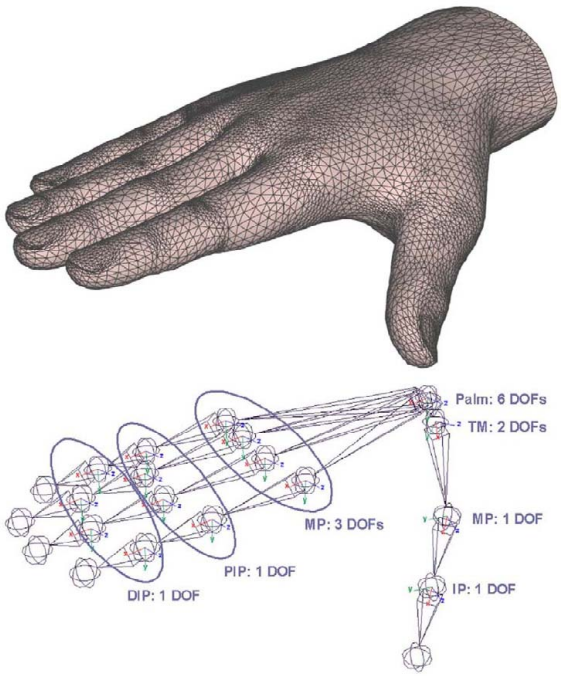


Figure 2.8: Hand model as polygonal surface and its skeleton with 15 degrees of freedom [16]

2.3 Recognition

There are several approaches to classify gestures, depending on what information to classify. Because gestures can be viewed as sequence of postures in time, the most widely used classification method is a Hidden Markov Model (HMM). HMMs are suited for the classification of gestures, they try to model the spatio-temporal variability and to maximize the likelihood of generating all examples of a gesture class [17].

Yang et al. trained a HMM for spotting specific body gestures like raising a hand, jumping etc., but also presented the concept of the garbage gesture model (see figure 2.9) and the key gesture spotting model (see figure 2.10). The garbage gesture model is an ergodic model and tied in the the key gesture spotter model at the beginning of a gesture and at the end [17].

Both Willson et al. and Chu and Cohen used the concept of a multiphasic gesture: in a sequence of postures, there is at least a transition to the key posture and a transition leaving it. Gestures can also have several key postures identifying a gesture. While Willson et al. did not use hidden states but a simpler markovian chain, Chu and Cohen used a dual-state HMM for a primary and secondary decomposition of gestures, which also gives smaller set of atoms and a HMM state space of linear complexity [10, 18].

Wang et al. introduce a powerful approach for gesture classification: hidden con-

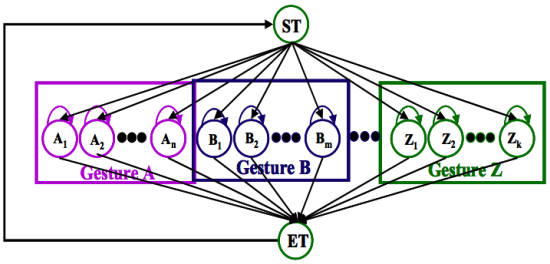


Figure 2.9: Garbage gesture model [17]

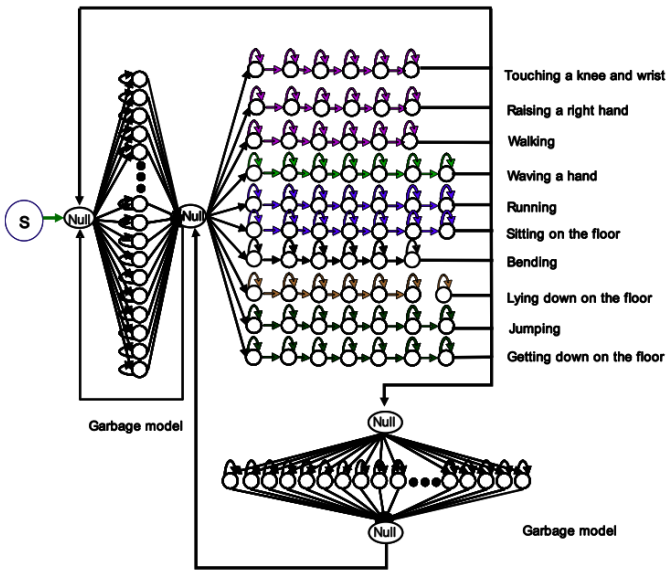


Figure 2.10: Key gesture spotter model [17]

ditional random fields (HCRF) (see figure 2.12). HCRFs incorporate hidden states variables in a discriminative multi-class random field. It is an extension of the spatial Conditional Random Fields and takes the main advantage of HMMs, the capability of modelling temporal sequences. HCRFs can be used like HMMs (see figure 2.11), but the hidden states can be shared among different gesture classes [19].

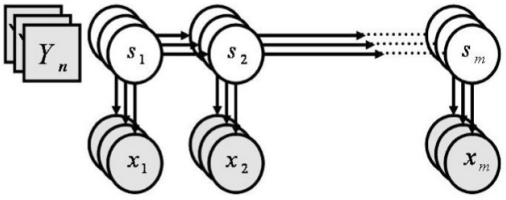


Figure 2.11: Hidden Markov Model [19]

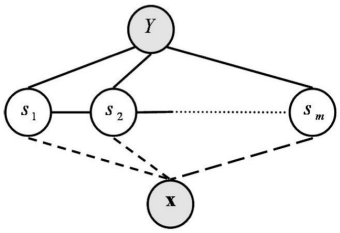


Figure 2.12: Hidden Conditional Random Field [19]

2.4 Our approach

Erol et al. [20] mention the difficulties in hand gesture recognition:

- high-dimensional problem
- self-occlusion
- uncontrolled environment
- rapid hand movements
- processing speed

There is an approach from Wang and Popovic from MIT, where the processing requirements are circumvented by generating a nearest neighbours lookup database for scaled-down 40×40 pixel images of a color marked glove (see figure 2.13) [21].

Figure 2.13: Pose estimation with a color-marked glove and its 40×40 pixels representation (tiny) [21]

However many tracking algorithms widely used in gesture recognition are confronted with real-time problems. Preliminary tests with advanced algorithms have been done in OpenCV [22] – a computer vision library that includes over 500 algorithms from research – using optical flow algorithm or background subtraction with mixtures of gaussians. While the results are real-time capable for smaller images, the frame-rate often drops to under 10 frames per second for bigger images.

Our approach aims at using simple techniques focusing on speed with some constraints limiting the scope of the project and still having an architecture where an real-time ergonomic gestural interface is possible.

3 Setup and frameworks

Following the overview of existing gesture recognition techniques in the previous chapter (chapter 2), this chapter focuses on the hardware and software requirements necessary for implementing a gesture recognition architecture. These requirements will lead to a choice of hardware components and of an ensemble of software libraries to be used in this project. Since there are many software libraries capable of performing the individual tasks of tracking or classification, but not all of them are suited for a real-time-capable integrated architecture, the choice of camera, environment constraints and libraries is greatly influenced by the main requirement of the real-time, or quasi-real-time component of ergonomic gestures.

3.1 Hardware

The main task of the hardware is to see a user's hand and fingers positions and movements while this user is sitting in front of the interface. While there are approaches based on accelerometer and gyroscopic sensors [23], this project will focus on a vision-based setup and will therefore use a camera. The interface can be anything from an LCD display to a beamer, as long it is in front of the user and not obstructing the field of view of the camera. In order for a camera to *see* the fingers, several important requirements have to be met for the camera:

- Resolution:
  The camera frame must contain enough information about the fingers in order to localize and distinguish them from each other. As ad-hoc tests have shown, resolutions of 320×240 pixels may work, depending of the user's action space and the distance between the camera and the user, but often overstep the threshold of being distinguishable or simply the threshold of detection of the fingers.

- Frame rate:
  Even if the recognition of a gesture can operate at a much lower sampling rate while still having enough consecutive samples to identify a gesture reliably (in the range of 5 to 20 frames per second), the articulated and sudden or rapid movements of the hand result in blurred fingers in a single frame taken at up to 30 frames per second [24]. Therefore, the camera should acquire the images at a speed where rapid movements can be captured without motion blur (even if each second or third frame will be skipped in the video processing loop).

- Color and image fidelity:
  Movements of hands and fingers should not result in distorted images often seen in conventional web-cams. In order to prevent difficulties in the tracking algorithms, the camera should have low color shifts and a low noise ratio;

Given a camera that meets these requirements and providing a solid basis for the video processing steps, this project further constrains the goal regarding the tracked object: in order to meet the requirement of being able to localize the finger positions in a given image, the user is wearing a glove with marked finger extremeties.

3.1.1 Camera

While most of the conventional USB Web-cameras do not meet the camera requirements established above, the following three cameras, often used in computer vision systems, do.

- Point Grey FireFly MV (752×480 at 60 FPS) [25]
- Unibrain Fire-i (501 Model: 640×480 at 86 FPS) [26]
- Sony PS3 Eye (640×480 at 75 FPS) [27]

While each of these cameras could be used in this project, the choice fell on the Sony PS3 Eye camera (see figure 3.1). Its main advantage is its cost, subsidized by Sony for the use with Playstation games, which is under CHF 70.- in most electronics stores.

Figure 3.1: Sony PS3 Eye camera

While Sony does not provide a driver, a member of the Natural User Interface group [28] provided a Windows driver that supports frame rates up 75 FPS in VGA (640×480) resolution and up to 120 FPS in QVGA (320×240) resolution [29]. The driver is also capable of grabbing frames from multiple cameras in OpenCV under Windows, which could be a requirement for stereoscopy, whereas under other Operating Systems, only one camera was recognized in OpenCV.

3.1.2 Glove

The use of a glove facilitates the tracking of the fingers: for the scope of this project, tracking color-based markers on the fingertips presented a acceptable shortcut to obtain the finger positions.
The glove was initially designed to put it on and off quickly, without having the disadvantage of placing all the color markers each time on the hand. The design also went through several versions optimizing the placement and color of the markers for better tracking results (see figure 3.2). The main improvement was obtained by placing the markers on the fingertips in a way that did not cover the whole circumference of the finger, which provided the ability to distinguish the fingers for the tracking phase.

(a) 1st version **(b)** 2nd version **(c)** 3rd version

Figure 3.2: Color-marked textile glove

3.1.3 Environmental constraints

Initially, the following setup had been proposed:

- the user is sitting in front of the interface,
- while sitting and resting the elbows on the table,
- and the camera being placed at a distance where the gestural action space is contained in the field of view of the camera (see figure 3.3).

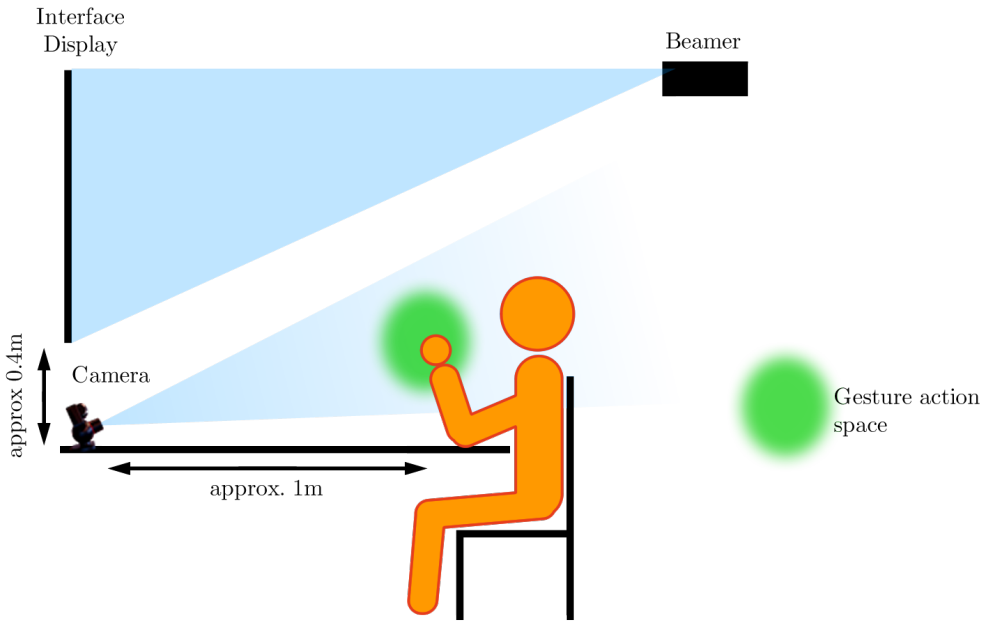


Figure 3.3: Setup of the interface

While this project aims at recognising hand postures and gestures without too much restricting the setup, some constraints have to be further imposed in order to reliably obtain good tracking and classification results.

Even if the tracker algorithm operates in a color space that is less dependent of the luminance of the color like HSL or Lab color space, there are some limits due to the gain of the camera: in a very dim lit environment the noise is too much amplified by the gain and the colors are more indistinguishable. Equally, with very bright spots or in direct sunlight, the pixels of the color markers lose their color information, turning to a color value close to white.

The glove's design was optimized for frontal control gestures. The placement of the wrist marker imposes a further environmental constraint: the camera has to be placed lower than the display, allowing it to film in a slightly rising direction and ensuring to capture the wrist markers whenever executing a gesture. But of course color markers on the glove also means that neither the user nor the background can contain these colors.

3.2 Software

The main requirement of the software is to allow real-time tracking and classification. To this regard, the tracking and classification algorithms should be written in C/C++. Of course, other languages might be viable for real-time image processing algorithms, capable of managing threads and offering helping mechanisms like a garbage collector.

But initial library tests have shown, especially for image processing algorithms, that with a slightly heightened awareness for memory management, thread management and algorithm execution times, better performances could be achieved more easily with C/C++.

3.2.1 Overview

A library or set of libraries is needed, where the following tasks are covered:

- fast image processing algorithms
- ability to train a classifier
- easily design a performant GUI for rapid application development
- take advantage of a multicore hardware architecture

Due to the chosen hardware (see section 3.1.1) for which the device driver operates best in the Windows operating system, the choice of the development environment falls to the Visual Studio 2008 IDE.

3.2.2 Libraries and frameworks

OpenCV

OpenCV [22] is a library containing a large set of algorithms covering computer vision and machine learning tasks. It is a cross-platform library initially developed by Intel and now maintained under active development by Willow Garage [30]. Its goal was to advance computer vision research and provide a basic infrastructure for computer vision. This library was chosen for its focus on real-time image processing. The current version is 2.2, however this project used the version 2.0, because at the time of the first integration of the OpenCV algorithms, the tracking was optimized for multithreading using OpenMP instructions. OpenCV abandoned its built-in OpenMP optimizations in favour of TBB (Thread Building Blocks).

dlibc++

The dlib library [31] was chosen for its well documented usage of Support Vector Machines (SVM). It provides a C++ API for a multitude of binary classifiers and regression tools: Relevance Vector Machines (RVM), kernel recursive least squares (KRLS), kernel ridge regression (KRR) and several implementations of Support Vector Machines.

QT

QT [32] was favored over Windows forms (.NET or MFC) because of the extension possibilities like the integration of OpenGL within QT widgets. The use of OpenGL to display enhanced video frames greatly improves the performance in combination with an OpenGL capable graphics hardware. Furthermore it is an multi-platform and open-source GUI framework.

libconfig

Initially the tracker program featured Boost object serialization for saving the states and parameters of the image processing steps. But due to consistency problems a simpler solution was needed. libconfig is a library for reading and writing structured configuration files, with type-awareness, groups, lists, arrays, strings and numbers.

4 Chosen gestures

This chapter introduces the gestures that this project aims to recognize. The establishement of a ground truth is discussed in section 4.1 and the selected gestures for this project presented in section 4.2.

4.1 Wizard of Oz experiment

In collaboration with Caroline Biewer and Thomas Holdener writing a master thesis in psychology [33] a setup was designed, where probands of a Wizard of Oz experiment would sit in front of a beamer screen and control an application (Google Earth and Powerpoint) through gestures (see figure 4.1). In this setup the probands were recorded in stereo according to the environmental constraints established in section 3.1.3 and wearing two gloves of the first version. This study had the goal of analyzing movements with regard to the comprehensiveness, comfort and learnability of their use in a usability evaluation. For this, the illusion of a working gesture recognition system was created, where in fact the "wizard of oz" was directly controlling the application with the keyboard and the mouse. The ergonomic gestures project helped provide this illusion by positioning two PS3 cameras in front of the probands and let them wear the color-marked gloves.

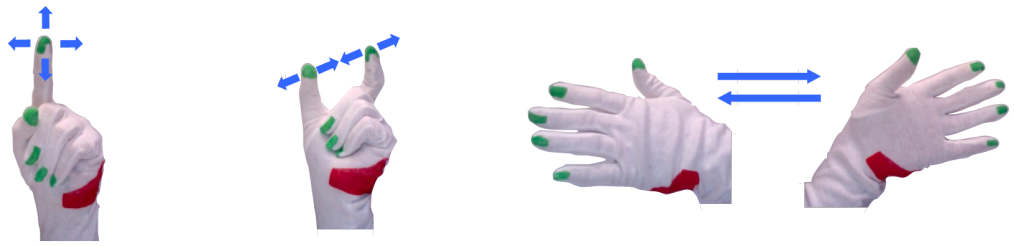
The stereoscopic recordings were intended to be used as a ground truth as well as to determine wich types of gestures were preferred for controlling a visual application. Unfortunately, due to unfavorable lighting conditions and several redesigns of the glove, the recordings were incompatible with later versions of the Ergonomic Gestures Recognition architecture.

Figure 4.1: Participants performing gestures

4.2 Gestures

After several glove redesigns and the use of monocular videos instead of stereoscopic videos, this project focused to recognize the following gestures, performed by the right hand:

- Pointing gesture: Index finger pointing upwards, other fingers contracted but with visible markers
- Zooming gesture: Index an thumb finger pointing upwards, zoom line between index and thumb more horizontal than vertical
- Horizontal swiping gesture: swiping motion mostly carried out by wrist, not the whole arm.



(a) Pointing **(b)** Zooming **(c)** Swiping

Figure 4.2: Gestures with motions labeled as arrows

These gestures were recorded in different videos containing all three gestures with no particular order and containing a single type of gesture repeated several times. In each video the gestures were performed exactly according to the specifications, but while trying maximize the wrist position variability.

5 Design and implementation

This chapter describes the architecture and implementation of the EGR project. The general three phases for gesture recognition, namely tracking, pose estimation and gesture classification, represents also the basis for this project (see chapter 2). Using the camera, the glove and the software libraries presented in chapter 3, an architecture that aims at recognizing the gestures chosen in chapter 4 is presented in section 5.1. In the sections 5.2, 5.3 and 5.4 implementation details of the tracking phase, the pose estimation phase and the classification phase are presented.

5.1 Architecture

The separation of the three gesture recognition phases into separate but linked modules allows a better understanding of the mechanisms involved and offers the possibility to replace a module by an alternate future version, which accomplishes the same task. There are two main modules and three applications:

- Tracking module
- Model module
- Recorder application
- Training application
- Demo application

The tracking module is implemented as a C library, which runs a continuous loop in a thread, and performs the image transformations and blob detection algorithms for each new frame from a video source. The hand model module is implemented as a C++ class object, which maps the information obtained from the tracker in a defined structure and optionally can classify a posture, assuming a decision function obtained from the training application is present. This hand object can then be directly instantiated in a potential gesture recognition application and initialized with the necessary information.

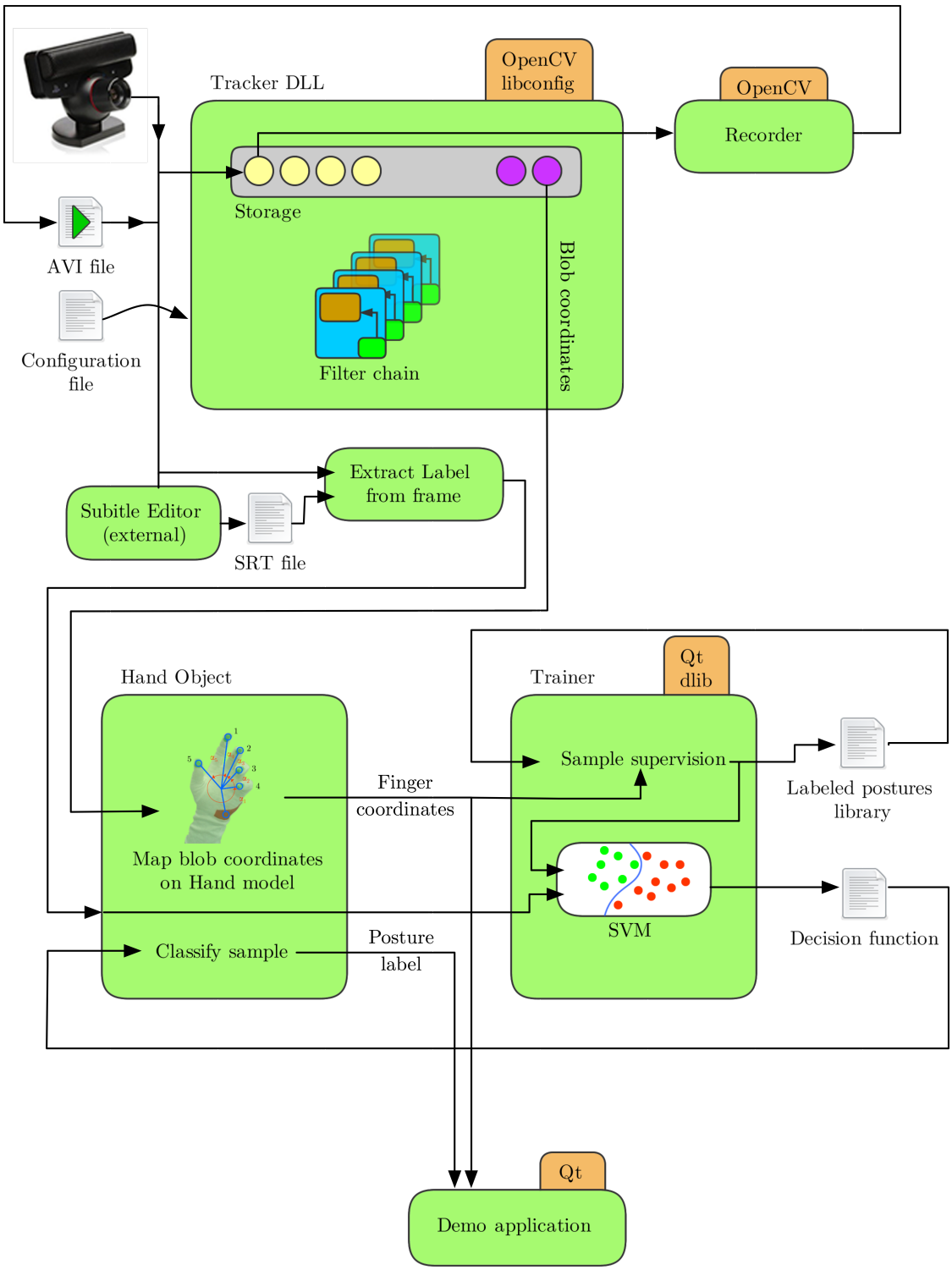


Figure 5.1: General architecture of the Ergonomic Gestures Recognition system

5.2 Tracker configuration

The tracker module is implemented as a library that can be dynamically linked from an another program. This tracker is optimized for speed: it has to run several image processing filters for each frame and achieve a reasonable frame-rate of 10 to 30 FPS. However the frame-rate depends entirely on the optimizations of each image processing algorithm.

The tracker consists of three parts: a storage object, a filter chain and a camera thread. The camera thread acquires the images from the PS3 Eye camera at 60 frames per second in a separate thread.

The storage object is a simple key-value storage, or more precisely a "key-anything" storage. Its only purpose is to eliminate the need to supervise each allocation and release of memory for the images at each iteration of the filter chain. It has also locking mechanisms implemented to ensure data consistency when an external thread accesses the memory in the storage object. The storage object is accessible from everywhere in the tracker: for example in the initialization phase, the camera descriptors are stored in it, but each image processing algorithm can also access it for its own purposes.

5.2.1 Filter chain

The filter chain allows an arbitrary configuration on how the image processing algorithms are linked. Each element in the chain, called step, can have one image processing algorithm, called filter, associated to it. Each step has also an overlay object, that can, whenever it is displayed, adjust the parameters and specify the input images of the filter. The filter object itself declares how many input images it needs, and upon executing its function returns a result image. These resulting images can now be used as input images, or sources, to each filter (see figure 5.2).

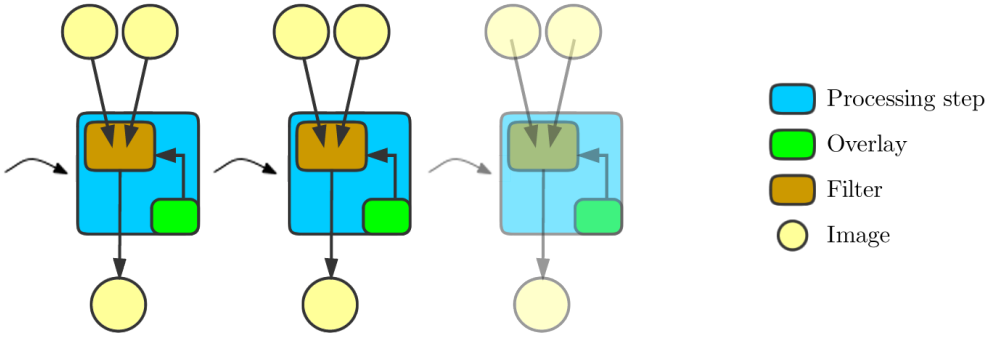


Figure 5.2: Image processing chain with reconfigurable processing steps: the filter and source image(s) can be specified using the overlay object belonging to the step.

The execution of the filter chain is sequential, but the specification of the input sources allow an arbitrary and reconfigurable order of image processing operations. To improve the execution speed, there is no GUI control elements used from QT or the built-in OpenCV GUI controls. Instead the user interface is directly in overlay mode with the result image of the selected step (see figure 5.3).

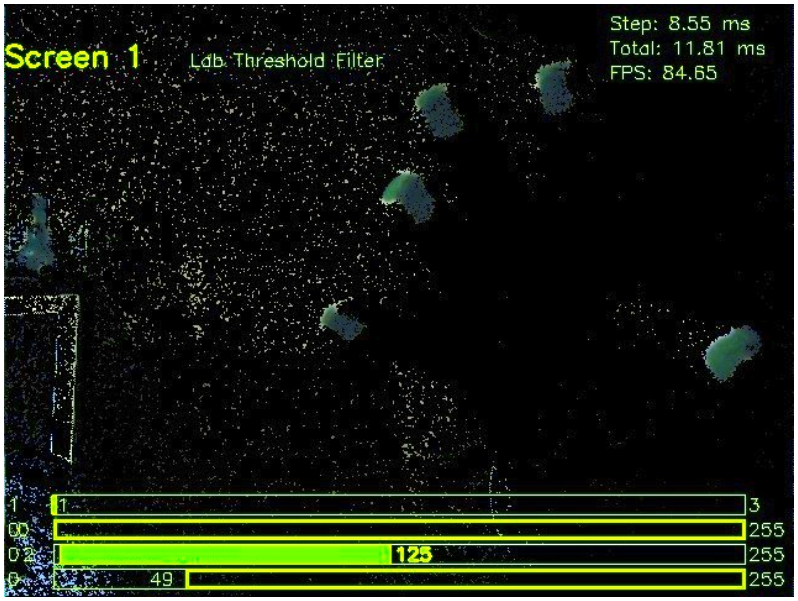


Figure 5.3: Lab threshold filter with overlay interface

As mentioned in 5.2, each filter has access to the global storage object, and can store temporary images in it, but also other objects like vectors containing the positions of the detected blobs. Knowing the identifying key to this vector of points, an external program linked to the tracker DLL can retrieve this information.

5.2.2 Image processing filters

The tracker provides a small but easily extensible list of filters:

- Lab threshold: convert from the RGB color space to the Lab color space, and threshold each plane with a value range.
- RGB threshold: threshold each RGB plane with a value range.
- Binary threshold: converts an image into a gray image if needed, and performs a binary threshold.
- Erosion: convolution directly implemented in OpenCV with a 3×3 kernel.
- Dilation: convolution directly implemented in OpenCV with a 3×3 kernel.

- Gaussian smooth: : convolution directly implemented in OpenCV with a gaussian kernel.
- Add: overlays non-zero parts of a first source image onto a second source image.
- Blob detection (see below)
- Fast corner detection (see below)

The blob detection and the fast corner detection filters use external algorithms, but the others are implemented with OpenCV functions. The blob detection filter uses the cvBlobsLib available at the OpenCV Wiki [34]. The blob detection itself is based on the linear-time component labeling algorithm using contour tracing technique by Chang et al. [35]. Tests have shown, that this algorithm performs well, as long the number of blobs remains small. However when the color thresholding filter and the erosion filter leave enough noise in the resulting image, the blob detection algorithm takes up to a second and thus impeaching the fast real-time execution. In order to circumvent this possible delay, other detection algorithms were investigated. The most promising was the FAST corner detection algorithm from Rosten and Drummond [36, 37]. This algorithm is capable of detecting corners at the same speed with noisy and clear images. However this approach was not continued, because the interpretation of the found corners would take more time than the blob detection filter.

5.2.3 Chosen filter chain

In this project three consecutive operations for each marker color of the glove (red and green) were empirically selected (see figure 5.4):

1. Lab Threshold filter
2. Erosion filter
3. Blob detection

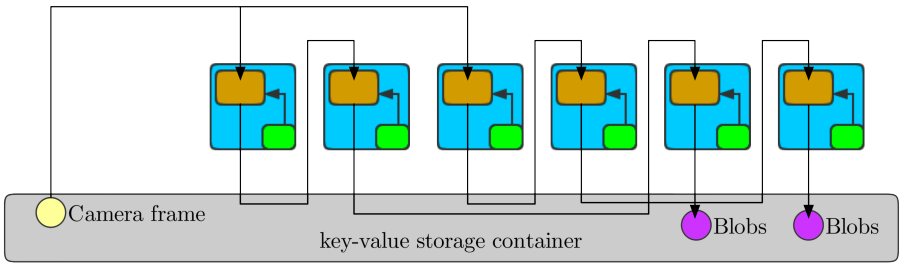


Figure 5.4: Specific chain used in this project: sequential execution order independent of input/output cross linking. Each output image is automatically stored in the global storage but filters can specify additional storage objects (like a vector of blob coordinates)

The six image processing algorithms could run on an average of 25 frames per second on a Macbook with an Intel Core Duo at 2.5 Ghz, and with a PS3 Eye camera providing 640×480 pixels frames at 60 frames per second. Initially the tracker loop was slightly faster, but had no locking mechanisms on the storage object. Counterintuitively, without locking mechanisms used the CPU up to 90% with an average of 29 frames per second, but with locking mechanisms the CPU utilization averaged at 45% and 25 frames per second. However the locking mechanisms were indispensable: without them random crashes occurred frequently.

The following table presents the filter chain CPU utilization in milliseconds:

Table 5.1: Filter processing times

Filter	**Processing time**
Lab threshold filter (green)	9.7 ms
Erosion filter (green)	1.5 ms
Blob detection (green)	6.6 ms
Lab threshold filter (red)	9.8 ms
Erosion filter (red)	1.2 ms
Blob detection (red)	5.5 ms
Add filter	1.5 ms
Add filter	0.4 ms
Tracker overhead	2.1 ms
Total	**38.3 ms**

The resulting filtered images of this specific filter chain are presented in chapter 6 (see figure 6.1).

5.3 Model mapping

The model mapping is a step in gesture recognition that is not always necessary, depending on the recognition process. But it is helping to accumulate information gathered from the tracker in a structured way, eventually with some transformations applied, in order to exploit physical constraints of the hand. In this project, several simple physical constraints are implemented, but the model of the hand could implement more complex constraints, like generalised constraints for the degrees of freedom for the fingers or the distinction of the left and right hand.

5.3.1 Hand model

In chapter 4 the interface control gestures of pointing, pinching (zoom) and swiping were chosen. These can be decomposed in the following distinct postures:

- Pointing posture
- Pinching posture
- Posture with all the fingers on the left side (part of the swiping gesture)
- Posture with all the fingers on the right side (part of the swiping gesture)

(a) Pointing **(b)** Pinching **(c)** Fingers left **(d)** Fingers right

Figure 5.5: Postures of hand with glove

The hand model should adhere to the following simple generalisations:

- A hand has five fingers.
- A hand has one wrist.
- Though crossing of fingers is physically possible, it is not very natural and easy to do: there is a circular order of the fingers and the model asserts that it remains (mostly) constant.
- The index finger and the thumb are the most articulated fingers, while the little finger, the ring finger, and sometimes the middle finger follow the same motion patterns (synchronously): the positions of the little finger and the ring finger can be estimated together.

A hand model constructed according to these generalisations should at least contain the following properties:

- Five finger coordinates
- Wrist coordinate

- Order to the fingers: index finger follows the middle finger, thumb follows the index finger
- Velocity vectors of fingers and wrist

5.3.2 Mapping

To map the unordered vector of finger coordinates and the wrist coordinates obtained from the tracker to the model described in 5.3.1, several mechanisms involving prediction of the coordinates for the next frame, along with an algorithm that finds the best permutation of recognized fingers to the prediction of the previous finger coordinates have been investigated. But the most robust mapping followed the generalization property, that the fingers have a specific order: once the index finger has been identified, the next fingers are mapped according to a circular path along the mass center of all the points (see figure 5.6):

1. Calulate mass center of all points. The wrist has double wheight.
2. Calculate angles $\alpha_{1...n}$ of the vectors from the mass center to each finger coordinate.
3. Sort the fingers according to this angle $\alpha_{1...n}$ in descending order.
4. Cycle the finger vectors in order to position the index finger as the first finger.
5. Missing fingers are calculated as an average of all the fingers without index finger and thumb (first and last finger of the fingers vector).

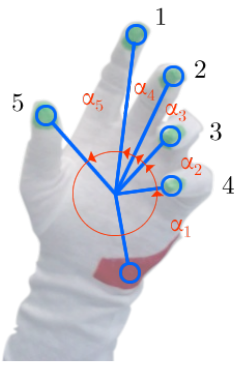


Figure 5.6: Glove with overlayed hand model. $\alpha_{1...n}$ indicates the angle of the vectors from the mass center to each finger coordinate.

5.4 Training and classification

Initially, attempts were made to directly classify the gesture using a Hidden Markov Model (HMM) as classifier and the set of coordinates of the fingers, or alternatively some other features best describing the gesture in question, as input of this classifier. For this, a test program was implemented, where different features of gesture could be temporally inspected: beside the coordinates, a feature that look promising was the distance of each finger to the wrist or the mass center. This length would also, in a general way, characterize the contraction of the finger, reducing the degree of freedom of each finger in a physical, two-dimensional model to one.

However, it became clear that even if this approach would eventually work, it still was not viable for real-time control gestures. Imagine a scenario where a trained classifier was fed with samples of finger coordinates for each new frame. Two common problems of gesture recognition would arise: when does a gesture begin, and when does it end? But assuming these key moments could somehow be identified, the HMM would still only classify the gesture correctly at its end. At a closer inspection of the gestures defined in section 4.2, it appeared, that the interface needed to know that the user was pointing somewhere, as soon the he entered a key posture (namely the pointing posture). Equally, as soon the user would stop pointing, the interface does not need to update a cursor position. This led to another approach, which follows more closely the definition of gestures being a sequence of postures.

The recognition approach implemented in this project consists of two parts:

1. Posture training and classification from finger coordinates
2. Gesture classification from posture sequences.

This separation allows real-time interaction, as posture classification is frame-based and could have an effect on the interface nearly instantaneously, whereas gestures are time-dependent and require several frames to take effect.

5.4.1 Sample collection

The process of sample collection requires one or more recorded videos containing sequences of postures selected in section 5.3.1 and conforming to the constraints of section 3.1.3. Additionally, a way of pre-labeling the postures semi-automatically was put in place, which eliminated the need of manually labeling each posture. This was not strictly necessary, but helped substantially for labeling one minute of video: 60 seconds $\times$ 25 frames per second results in 1500 frames, or approximately 1200 usable postures. The pre-labeling process included a file for the labels, which consisted in a simple text-based subtitle file. This subtitle file had the advantage that it could be edited in any subtitle editor (see figure 5.7). The combination of a video and its subtitle file containing the labels of postures makes it also possible to view and inspect the

posture-label mapping in any video player capable of displaying subtitles. With this pre-labeling technique the process of sample collection consisted of:

1. Roughly pre-label postures in a video with a subtitle editor, resulting in a SRT subtitle file,
2. Set the video as the source of the tracker,
3. Map the resulting blobs to a hand model and label it according to the corresponding frame in the subtitle file (see figure 5.8),
4. Save the supervised collection of labeled postures to a posture database file.

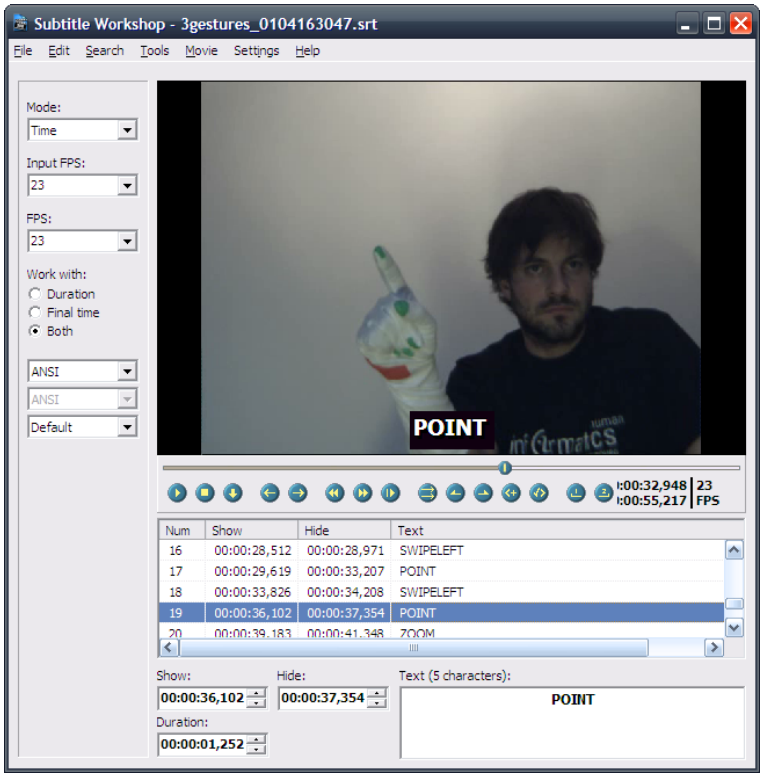


Figure 5.7: Posture labeling with an external subtitle editor. Each posture is labeled with the beginning and the end (in ms) of its presence in the video sequence. This demarcation is the basic structure of a subtitle file.

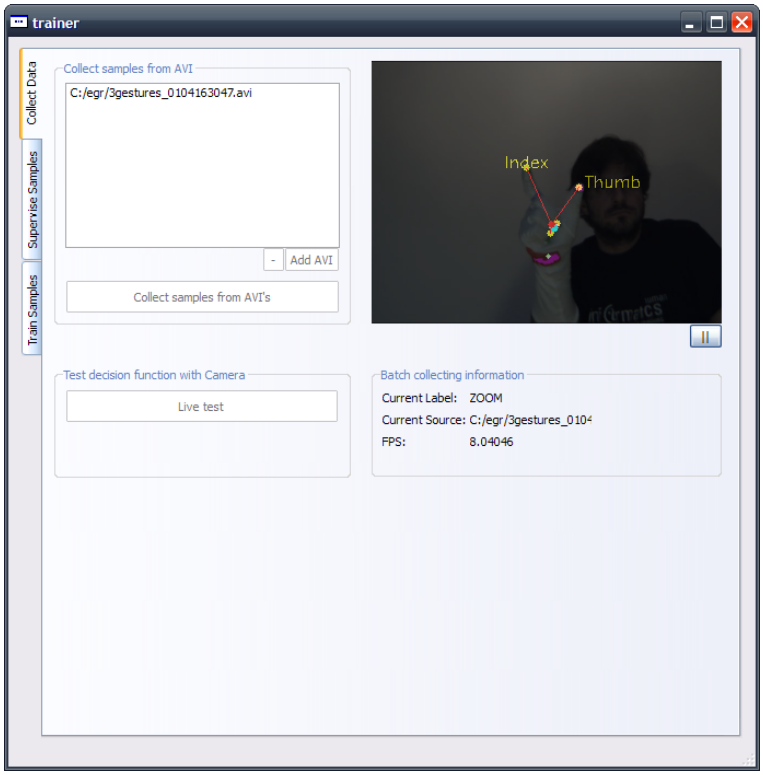


Figure 5.8: Posture sample collection

5.4.2 Sample supervision

In a first attempt a support vector machine was trained with the posture samples database obtained from section 5.8. But even the parameter estimation for the support vector machine with a cross validation algorithm did not finish to run in days. At a closer inspection of the collected samples, it became apparent that some samples have been mislabeled, due to the rough boundary of beginning and end of a posture sequence obtained from the subtitle file. Other samples included unusable tracking results from background noise or occluded finger positions. In order to provide a correctly labeled sample database to a classifier, a graphical user interface for quick relabeling and posture inspection was implemented (see figure 5.9). It allowed to view only samples from one label class, relabel them, or remove the sample altogether from the database, due to inconsistent finger coordinates. In a filtered mode, where only one label class was visible, a cumulative view of all the finger positions in this label class was displayed, helping to inspect the overall finger coordinate distribution of this label class (see figure 5.10). Each view of a sample displays the finger coordinates relative to the wrist position, giving a easily identifiable representation of a posture.

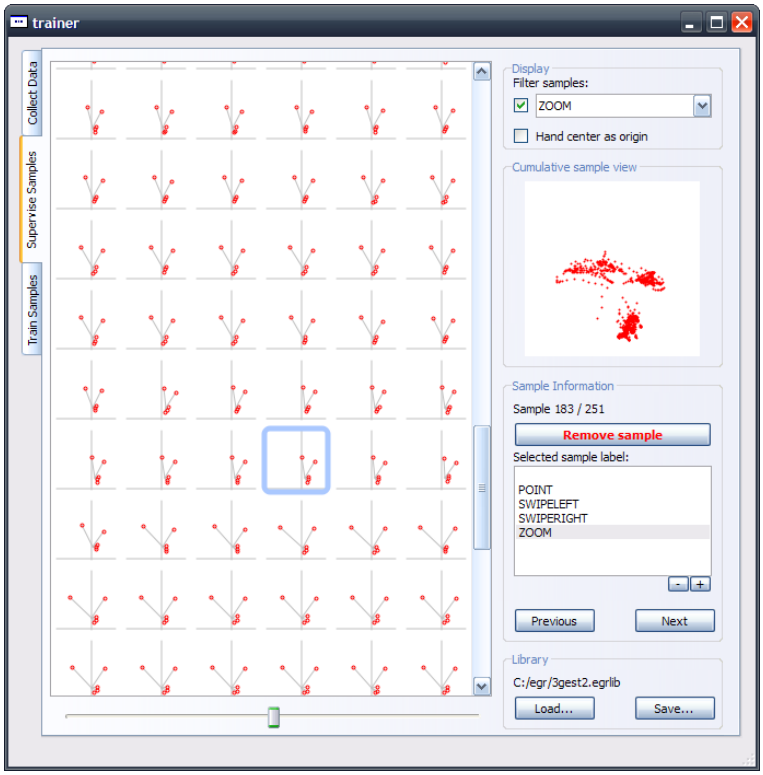
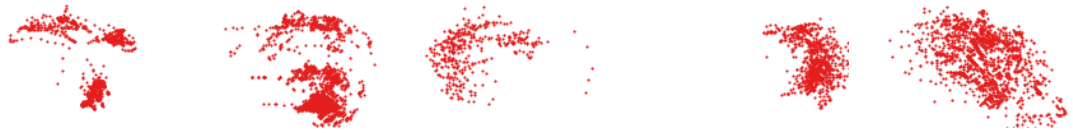


Figure 5.9: Sample supervision



(a) Pinching **(b)** Pointing **(c)** Fingers on the left **(d)** Fingers on the right **(e)** Garbage

Figure 5.10: Cumulative view of finger positions for each label

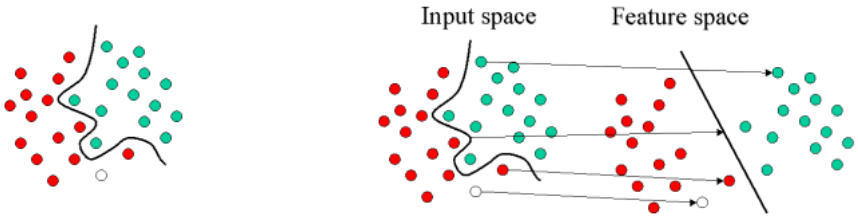
5.4.3 Posture training

The training of a classifier is done using the dlib C++ library, which packages many classification, regression and clustering algorithms in its machine learning module:

- Classification (binary): Relevance Vector Machine (RVM), C-Support Vector Machine (C-SVM), C-SVM using empirical kernel maps, linear C-SVM, ν-SVM, Pegasos-SVM

- Regression: Kernel Recursive Least Squares (KRLS), Kernel Ridge Regression (KRR), Multilayer Perceptron (MLP), ε Support Vector Regression (ε-SVR)
- Kernels: linear kernel, polynomial kernel, radial basis function kernel, sigmoid kernel

For this project a multi-class ν-SVM will be used to train a decision function, in order to distinguish between the different classes of postures. A non-linear support vector machine with a kernel function is ideally suited for this task: a support vector machine separates a set of objects using a linear boundary, or decision plane. However not every classification problem is linearly separable: but the kernel function can rearrange, or transform the objects in a higher-dimensional space, known as feature-space, so that a hyperplane in this higher-dimensional space can linearly separate these objects [38] (see figure 5.11).



(a) Non-linear decision boundary

(b) Linear decision boundary in feature space

Figure 5.11: Example of a non-linearly separable classification [38]

The feature vector $\mathbf{v}$ used to train the ν-SVM is a 10-dimensional vector containing the two-dimensional coordinates of the five fingers:

$$\mathbf{v} = \begin{pmatrix} x_1 \\ y_1 \\ \vdots \\ x_5 \\ y_5 \end{pmatrix}$$

The decision function is trained using a one versus one training structure: since the classifiers provided by dlib are binary classifiers, this structure creates for n possible classes $\frac{n \cdot (n-1)}{2}$ binary classifiers to vote on the identity of a test sample. For the 4 possible postures defined in section 5.3.1 there will be 5 classes, including a garbage posture representing everything not belonging to the other 4 postures.

This trained classifier produces a decision function that has an accuracy of 94% on the training data.

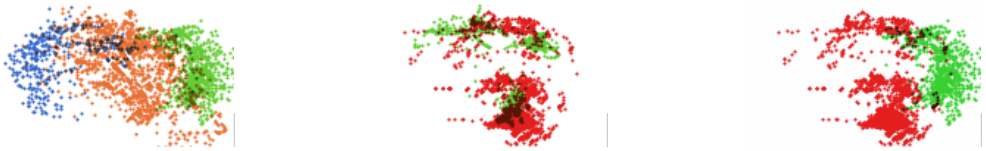
5.4.4 Posture classification

In a cross-validation test the decision function obtained in section 5.4.3 produces a confusion matrix, represented in table 5.2. This confusion matrix qualitatively gives an indication if the classifier will work on similar test samples.

Table 5.2: Confusion matrix of the trained decision function

Identified as:	Garbage	Pointing	Left	Right	Zoom
Garbage	283	11	8	12	3
Pointing	14	383	0	0	0
Left	6	0	71	1	0
Right	8	0	0	166	0
Zoom	4	1	0	0	247

In a live test with the camera, the accuracy is subjectively very good for the postures of pinching, with all the fingers on the left and with all the fingers on the right. However there are more misclassifications of the pointing posture, identified as a posture with all the fingers on the right. Going back to section 5.4.2, a simple comparison of the cumulative finger positions for each label class does not reveal large overlaying areas for the finger coordinates distribution of these postures (see dark regions in figure 5.12).



(a) Left (blue), garbage (orange), right (green)

(b) Pointing (red), pinching (green)

(c) Pointing (red), right (green)

Figure 5.12: Overlayed cumulative view of finger positions

But these misclassifications can be filtered out with simple majority filter with a window size that is still acceptable for real-time posture classification. With a window size of 10 frames, which equals 0.4 seconds at 25 FPS, a majority can often be reached in 0.2 seconds, therefore providing a more robust posture classification, that still has a very low reaction time.

5.4.5 Gesture classification

The recognition of a gesture can be achieved with an Hidden Markov Model (HMM) for each gesture. The state diagram for the pointing gesture (see figure 5.13) has exactly the same structure as the zooming gesture, where there is only one state for the gesture and the only transition allowing to remain in that state is the key posture to this gesture.

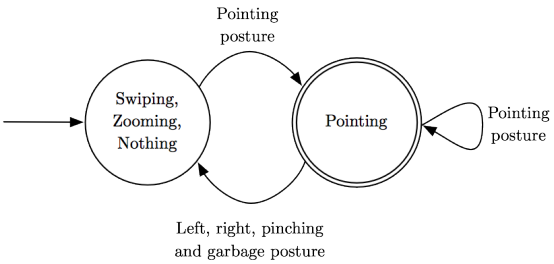


Figure 5.13: State diagram of pointing gesture

The swiping gesture has a temporal dependency: the only way to reach the final swiping gesture state is to have right postures follow left postures, or vice versa for a swiping gesture in the opposite direction (see figure 5.14).

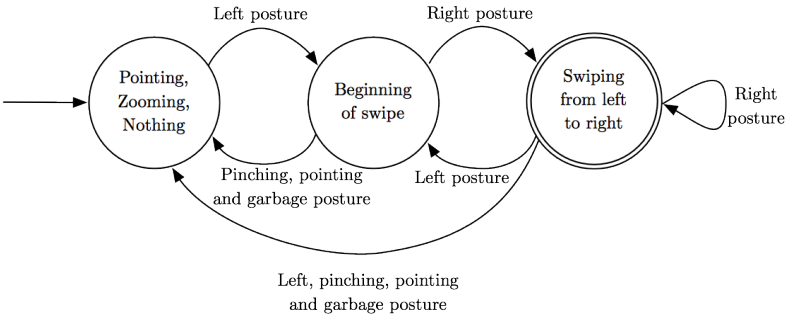


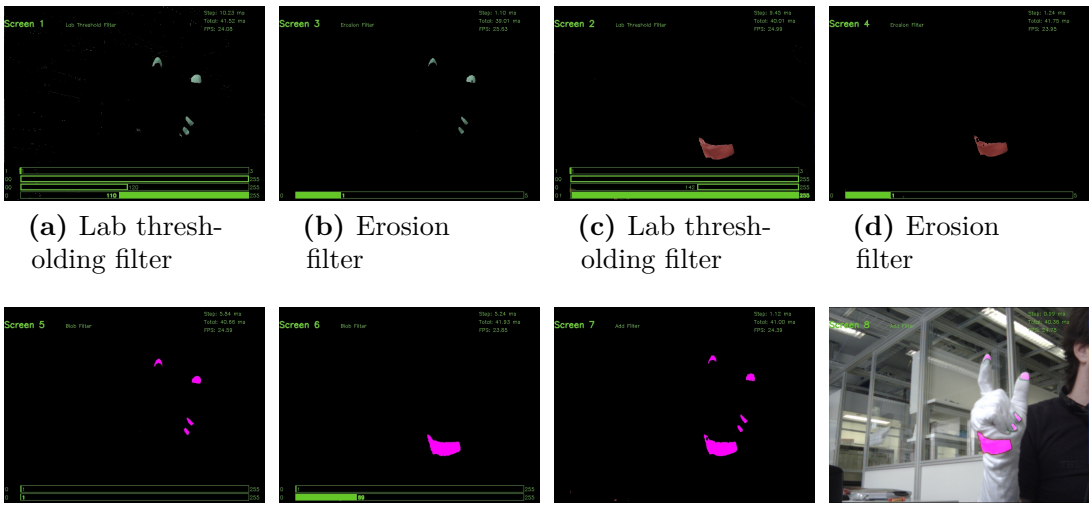
Figure 5.14: State diagram of swiping (from left to right) gesture

These HMM's could be implemented using the HMM Toolkit [39]. But for the scope of this project and because of the low complexity of the gesture states the recognition was implemented programmatically without HMM's but respecting the transition logic of the gesture state diagrams.

However, gestures with more complex spatial or temporal dependencies would definitively require of the use of an appropriate structure like a HMM. As a simple example, a gesture with more complex spatial dependencies would be a certain hand posture, i.e. the pointing posture, following a specific trajectory, i.e. a circle, that has a symbolic meaning and is not used for the direct control of a cursor following this trajectory. The feature vector for this specific HMM would not only require the posture label, but also the coordinates of the normalized index finger position.

6 Evaluation

Following chapter 5 which discusses the design and implementation of the gesture recognition architecture, this chapter presents the results obtained with this system. The results of the chosen filter chain (see section 5.2.3) can be seen in figure 6.1. The results of training a classifier for the postures is discussed in section 6.1.



(a) Lab thresholding filter **(b)** Erosion filter **(c)** Lab thresholding filter **(d)** Erosion filter

(e) Blob detection filter **(f)** Blob detection filter **(g)** Add filter **(h)** Add filter

Figure 6.1: Resulting steps from tracker with filter chain from section 5.2.3

6.1 Evaluation results

The different modules and programs have been tested qualitatively with a 55 seconds video recorded at 23 frames per second and containing 23 gestures (5 pointing gestures, 6 zooming gestures and 12 swiping gestures). These 55 seconds of video resulted in 1251 postures and after the sample supervision (see section 5.4.2) 1219 of them were used for the training (see table 6.1).

Once the trainer has a decision function loaded, either by directly training on a sample library or by loading it from a file, it can directly be tested within the trainer using live images from the camera. Analogue to the display of the label from the subtitle during the sample collection (see section 5.4.1) the label obtained from the decision function is

Table 6.1: Postures and gestures of training video

gesture name	number of gestures	number of postures
Nothing	24	317
Pointing	5	397
Zooming	6	253
Swiping	12	78 (left)
		174 (right)
Total	**47**	**1219**

displayed in the same position, indicating the posture that is currently recognized. The filtered posture (see section 5.4.4) worked remarkably well as long the setup constraints (see section 3.1.3) and the gesture definitions (see section 4.2) were respected:

- the distance between glove and camera during the live test should be similar to this distance in the recorded gestures.
- the lightning conditions are not extreme: the auto-exposure and the auto-gain of the camera have their limits.
- no background elements should contain green or red colors.
- tracker can see the characteristic finger blobs of a posture as well as the wrist blob.

However, the posture classification is not working well in the following situations:

- Sometimes when pointing very low, the red marker can not be detected.
- Pointing to the far right side, without moving the wrist can be classified as a right posture.
- During the zooming gesture, when thumb and index fingers almost touch each other the classification can result in a posture that is either classified as the pointing posture or the garbage posture.

6.2 Use case

For the demonstration of an interface controlled by gestures, a small program was implemented (see figure 6.2). This demo program mapped the three gestures from section 4.2 to the following behaviors:

- Pointing gesture: moves the cursor
- Zooming gesture: updates a bar indicating the zoom factor
- Swiping gesture: lets the interface slide in or out on the right side.

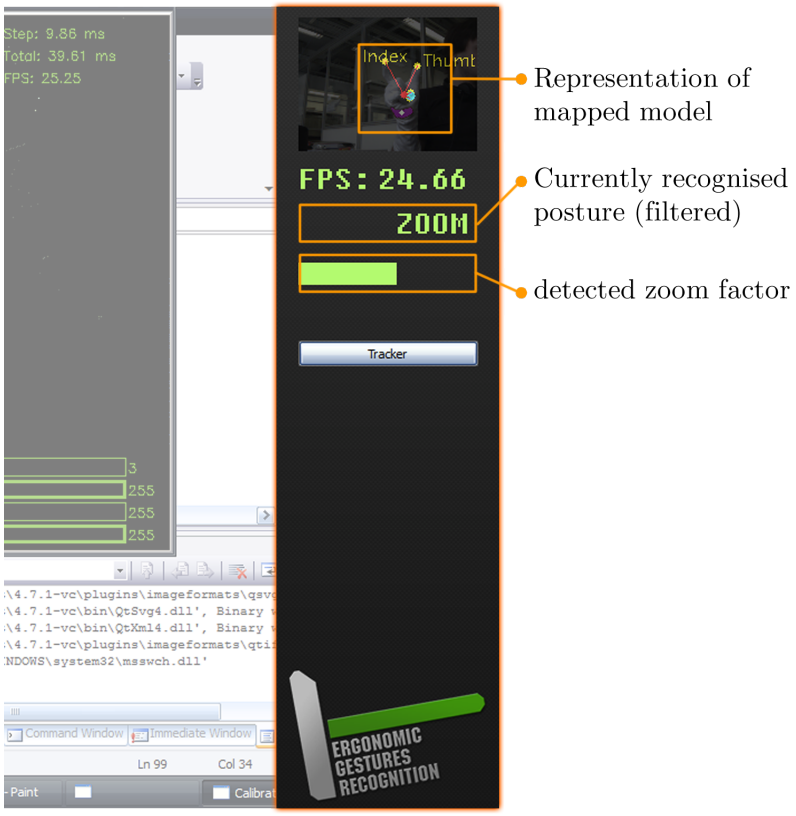


Figure 6.2: Demo application

This program has no real function other than to demonstrate the recognition of the control gestures. This program was also tested with several other users, who mentioned the following feedback:

- the reaction time is very fast,
- the gestures are easy to learn,
- a real gestural interface would require more gestures, of the same complexity level (i.e. a vertical swipe or grabbing gesture)

7 Conclusion and outlook

This chapter will recapitulate the master thesis, point out the achievements and mention areas of improvement and problems.
In chapter 1 a definition of *ergonomic gestures* and the goals of this project have been presented. In an overview, chapter 2 illustrated the different steps involved in gesture recognition, each with some examples from research. It also outlined the general structure and the focus that this project will have on an implementational level. Chapter 3 presented the necessary hardware and software, as well as the constraints limiting the scope of this project. With the proposition of the gestures that this project is focusing on (see chapter 4), all the preliminary informations were present to propose an architecture for real-time gesture recognition. The design and implementation details of this architecture were further discussed in chapter 5, whereas chapter 6 evaluated this architecture.

This master thesis project achieved its main goal, namely to design and implement a real-time capable gesture recognition system. The most important aspects for ergonomic application control gestures that were also obtained in the evaluation study [33] (see section 4.1) were:

- Fast reaction time
- Gestures that are easy to learn and can be generalized into an execution of motions and key postures that can be shared among multiple users

Additionally to these achievements, the system is also able to recognize small differences and track precisely the small-grained (i.e. zoom factor of the pinching gesture), articulated nature of ergonomic gestures.

For more complex gestures several future developments can be considered as extensions to this master thesis:

- train more postures and investigate the SVM parametrization to obtain distinguishable posture classification results,
- use HMM's to classify gestures with complex temporal or spatial dependencies,
- replace the tracking module with some other real-time capable hand tracking system that does not need the use of a color-marked glove

Bibliography

[1] Gord Kurtenbach and Eric Hulteen. *The Art of Human-Computer Interface Design*, chapter Gestures in Human-Computer Communications, pages 309–317. Add, 1990. (cited on page 1)

[2] B. Rime and L. Schiaratura. *Fundamentals of Nonverbal Behavior*, chapter Gesture and speech, pages 239–281. Press Syndicate of the University of Cambridge, 1991. (cited on page 1)

[3] T.B. Moeslund, A. Hilton, and V. Krueger. A survey of advances in vision-based human motion capture and analysis. *Computer vision and image understanding*, 104(2-3):90–126, 2006. (cited on pages 2 and 5)

[4] J. Schmidt, J. Fritsch, and B. Kwolek. Kernel particle filter for real-time 3D body tracking in monocular color images. In *Proc. 7th International Conference on Automatic Face and Gesture Recognition FGR 2006*, pages 567–572, 2–6 April 2006. (cited on pages 2 and 5)

[5] Ching-Yu Chien, Chung-Lin Huang, and Chih-Ming Fu. A vision-based real-time pointing arm gesture tracking and recognition system. In *Proc. IEEE International Conference on Multimedia and Expo*, pages 983–986, 2–5 July 2007. (cited on pages 2 and 4)

[6] M. Van den Bergh, E. Koller-Meier, and L. Van Gool. Fast Body Posture Estimation using Volumetric Features. In *Proc. IEEE Workshop on Motion and video Computing WMVC 2008*, pages 1–8, 8–9 January 2008. (cited on pages 2, 4, 6, and 42)

[7] C. R. Wren, A. Azarbayejani, T. Darrell, and A. P. Pentland. Pfinder: real-time tracking of the human body. 19(7):780–785, July 1997. (cited on pages 3, 6, and 42)

[8] H Sidenbladh. Detecting human motion with support vector machines. In *Proceedings of the 17th international conference on pattern recognition*, volume 2, pages 188–191, 2004. ISBN 0-7695-2128-2. (cited on page 3)

[9] Kazuhiko Takahashi, Yusuke Nagasawa, and Masafumi Hashimoto. Remarks on Markerless Human Motion Capture from Voxel Reconstruction with Simple Human Model. In *2008 IEEE/RSJ International Conference on Robots and Intelligent Systems*, pages 755–760, 2008. ISBN 978-1-4244-2057-5. (cited on page 4)

[10] Chi-Wei Chu and I. Cohen. Posture and Gesture Recognition using 3D Body Shapes Decomposition. In *Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 69–69, 25–25 June 2005. (cited on pages 4, 6, 8, and 42)

[11] Malik J. Mori, G. Recovering 3D human body configurations using shape contexts. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(7):1052 –1062, July 2006. ISSN 0162-8828. (cited on pages 4, 7, and 42)

[12] G Mori, XF Ren, AA Efros, and J Malik. Recovering human body configurations: Combining segmentation and recognition. In *Proceedings of the 2004 IEEE Computer Society conference on computer vision and pattern recognition, Vol 2*, pages 326–333, 2004. ISBN 0-7695-2158-4. (cited on pages 4, 7, and 42)

[13] Y. Azoz, L. Devi, and R. Sharma. Reliable tracking of human arm dynamics by multiple cue integration and constraint fusion. In *Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 905–910, 23–25 June 1998. (cited on page 4)

[14] K. Nickel and R. Stiefelhagen. Pointing gesture recognition based on 3D-tracking of face, hands and head orientation. In *Proceedings of the 5th international conference on Multimodal interfaces*, page 146. ACM, 2003. (cited on pages 5 and 42)

[15] A Agarwal and B Triggs. Recovering 3d human pose from monocular images. *IEEE transactions on pattern analysis and machine intelligence*, 28(1):44–58, January 2006. ISSN 0162-8828. (cited on page 5)

[16] M. Bray, E. Koller-Meier, and L. Van Gool. Smart particle filtering for 3d hand tracking. In *Automatic Face and Gesture Recognition, 2004. Proceedings. Sixth IEEE International Conference on*, pages 675 – 680, May 2004. (cited on pages 7, 8, and 42)

[17] Hee-Deok Yang, A-Yeon Park, and Seong-Whan Lee. Robust spotting of key gestures from whole body motion sequence. In *Proceedings of the Seventh International Conference on Automatic Face and Gesture Recognition - Proceedings of the Seventh International Conference*, pages 231–236, 2006. ISBN 0-7695-2503-2. (cited on pages 8, 9, and 42)

[18] A. D. Wilson, A. F. Bobick, and J. Cassell. Recovering the temporal structure of natural gesture. In *Proc. Second International Conference on Automatic Face and Gesture Recognition*, pages 66–71, 14–16 October 1996. (cited on page 8)

[19] Sy Bor Wang, A. Quattoni, L. P. Morency, D. Demirdjian, and T. Darrell. Hidden conditional random fields for gesture recognition. In *Proc. IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 2, pages 1521–1527, 2006. (cited on pages 9, 10, and 42)

[20] Ali Erol, George Bebis, Mircea Nicolescu, Richard D. Boyle, and Xander Twombly. Vision-based hand pose estimation: A review. *Computer Vision And Image Understanding*, 108(1-2):52–73, October-November 2007. ISSN 1077-3142. (cited on page 10)

[21] Robert Y. Wang and Jovan Popović. Real-time hand-tracking with a color glove. *ACM Trans. Graph.*, 28:63:1–63:8, July 2009. ISSN 0730-0301. (cited on pages 10 and 42)

[22] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000. (cited on pages 10 and 15)

[23] Ji-Hwan Kim, Nguyen Duc Thang, and Tae-Seong Kim. 3-D hand motion tracking and gesture recognition using a data glove. In *Industrial Electronics, 2009. ISIE 2009. IEEE International Symposium on*, pages 1013 –1018, July 2009. (cited on page 11)

[24] Ming-Hsuan Yang, N. Ahuja, and M. Tabb. Extraction of 2D motion trajectories and its application to hand gesture recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(8):1061 – 1074, August 2002. ISSN 0162-8828. (cited on page 11)

[25] Point Grey Firefly MV CMOS Camera. URL http://www.ptgrey.com/products/fireflymv/fireflymv_usb_firewire_cmos_camera.asp. (cited on page 12)

[26] Unibrain Fire-i 501 VGA high frame rate camera. URL http://www.unibrain.com/products/visionimg/fire_i_501.htm. (cited on page 12)

[27] Playstation 3 Eye Camera. URL http://us.playstation.com/ps3/accessories/playstation-eye-camera-ps3.html. (cited on page 12)

[28] Natural User Interface Group. URL http://nuigroup.com. (cited on page 12)

[29] Code Laboratories PS3 Eye driver. URL http://codelaboratories.com/products/eye/driver/. (cited on page 12)

[30] OpenCV Wiki. URL http://opencv.willowgarage.com/wiki/. (cited on page 15)

[31] Davis E. King. Dlib-ml: A Machine Learning Toolkit. *Journal of Machine Learning Research*, 10:1755–1758, 2009. (cited on page 15)

[32] QT: Graphical User Interface framwork. URL http://qt.nokia.com/. (cited on page 16)

[33] Thomas Holdener Caroline Biewer. Usability evaluation of a system for gesture recognition. Master's thesis, University of Fribourg, Department of Psychology, 2010. (cited on pages 17 and 37)

[34] Blob extraction library cvBlobsLib. URL http://opencv.willowgarage.com/wiki/cvBlobsLib. (cited on page 23)

[35] F Chang, CJ Chen, and CJ Lu. A linear-time component-labeling algorithm using contour tracing technique. *Computer Vision And Image Understanding*, 93(2): 206–220, February 2004. ISSN 1077-3142. (cited on page 23)

[36] Edward Rosten and Tom Drummond. Fusing points and lines for high performance tracking. In *IEEE International Conference on Computer Vision*, volume 2, pages 1508–1511, October 2005. (cited on page 23)

[37] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *European Conference on Computer Vision*, volume 1, pages 430–443, May 2006. (cited on page 23)

[38] Thomas Hill and Pawel Lewicki. *Statistics Methods and Applications. A Comprehensive Reference for Science, Industry and Data Mining*. Statsoft: Tulsa, 2006. (cited on pages 31 and 43)

[39] Hidden Markov Model Toolkit. URL http://htk.eng.cam.ac.uk/. (cited on page 33)

List of Figures

List of Tables

Appendix

The appendix contains the following sections:

- Installation
- Train a new posture
- Create a new tracker filter
- Write an application with EGRHand

Installation

All the required files are on the CD-ROM in the Prerequisites directory (except Visual Studio 2008).

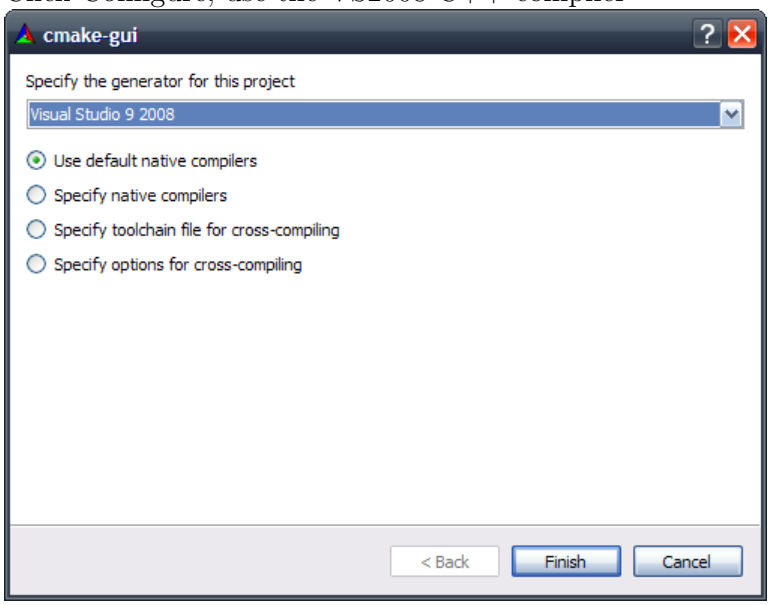
Visual Studio

Install Visual Studio 2008 Professional with standard installation options.

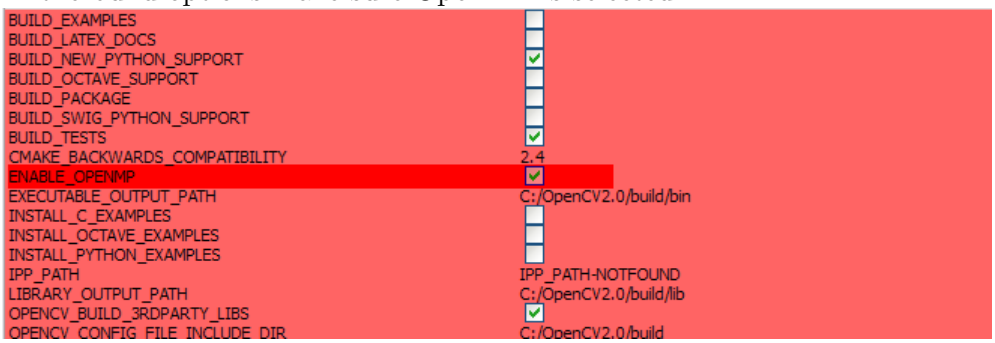
OpenCV 2.0

1. Download and install the newest 32-bit binary package for CMake from `http://www.cmake.org`
2. Download OpenCV 2.0 from `http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.0/`
3. During the installation, select to add the OpenCV to the system path
4. Make sure the source is also installed (*src*)
5. Open CMake
6. Set the source code to `C:\OpenCV2.0` and the build directory to `C:\OpenCV2.0\build`

7. Click Configure, use the VS2008 C++ compiler



8. In the build options make sure OpenMP is selected

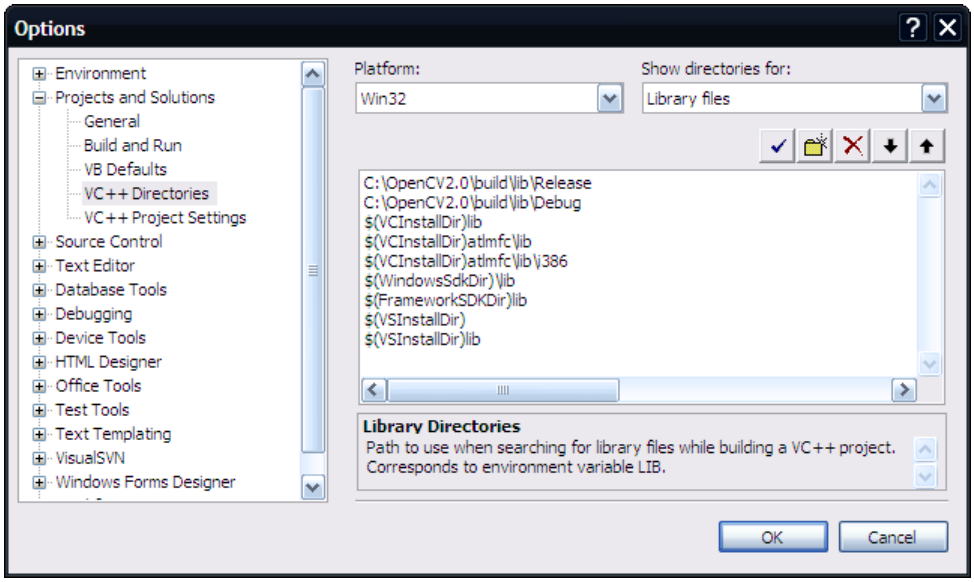


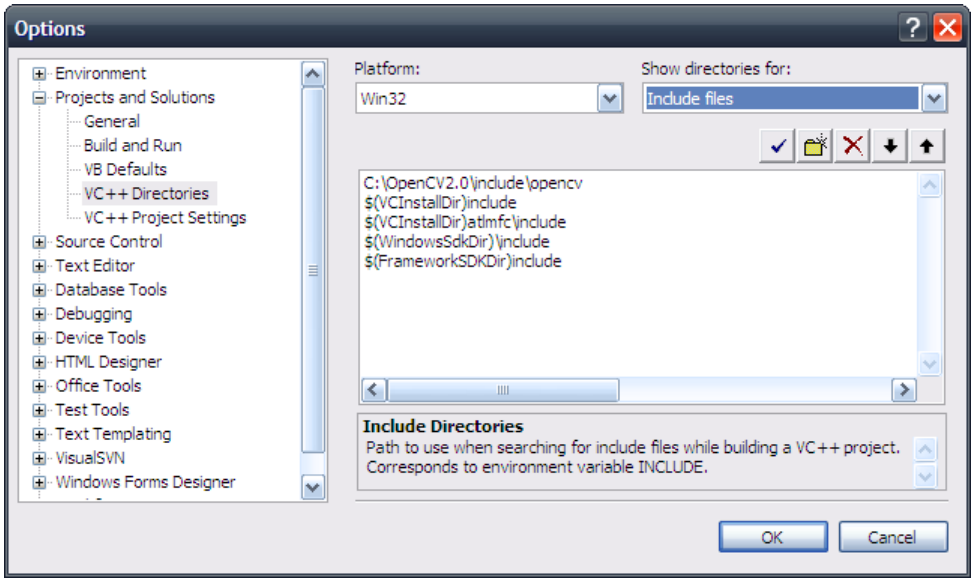
9. Click Configure again, and then Build

10. Under `C:\OpenCV2.0\build` open `OpenCV.sln`

11. Build the solution in Visual Studio under the Debug target and the Release target

12. In Visual Studio, under Tools → Options → Projects and Solutions → VC++ Directories, add the OpenCV library directories and the OpenCV include directory:





13. Add the following line to the system PATH environment variable:
`C:\OpenCV2.0\build\bin\Debug;C:\OpenCV2.0\build\bin\Release;`
(In Computer → Properties → Advanced → Environment Variables)

libconfig

1. Download libconfig 1.4.6 from
http://www.hyperrealm.com/libconfig/

2. Extract it to `C:\libconfig-1.4.6`
3. Open `C:\libconfig-1.4.6\libconfig.sln` and build the solution under the Debug and Release target.
4. Add the following line to the system PATH environment variable: `C:\libconfig-1.4.6\Debug;C:\libconfig-1.4.6\Release;` (In Computer → Properties → Advanced → Environment Variables)

cvBlobsLib

1. Download cvBlobsLib from http://opencv.willowgarage.com/wiki/cvBlobsLib
2. Extract it to `C:\cvblobslib`
3. Open `C:\cvblobslib\cvblobslib.dsp` and convert to a VS 2008 Solution
4. Build the solution under the Debug and Release target

dlib

1. Download dlib C++ from http://dlib.net
2. Extract the entire archive to `C:\dlib-17.34`

PS3 Eye

1. Download the CL Eye Platform Driver and the CL Eye Platform SDK from http://codelaboratories.com/downloads
2. Install the driver and the SDK
3. Add the CL Eye SDK library directory to the system PATH environment variable: `C:\Program Files\Code Laboratories\CL-Eye Platform SDK\Bin;` (In Computer → Properties → Advanced → Environment Variables)

Qt 4.7.1

These installation instructions were taken from https://dcsoft.wordpress.com/2010/01/30/

1. Download Qt from http://get.qt.nokia.com/qt/source/qt-everywhere-opensource-src-4.7.1.zip

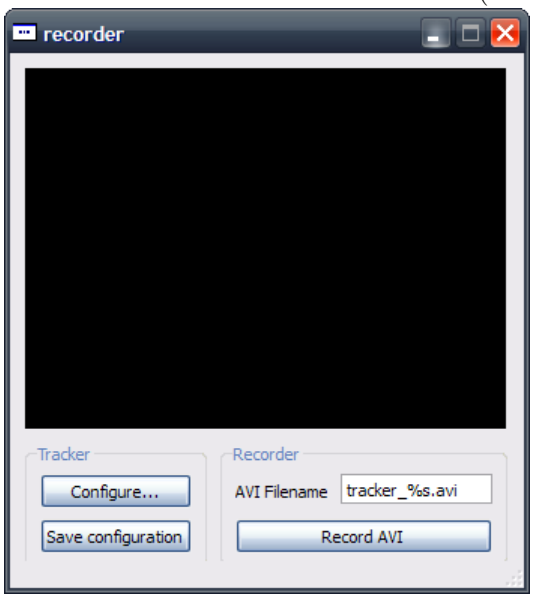
2. Extract the archive to `c:\qt\4.6.1-vc`
3. Add the environment variable QTDIR: `QTDIR=c:\qt\4.6.1-vc` and the following line to the system PATH environment variable: `%QTDIR%\bin` (In Computer → Properties → Advanced → Environment Variables)
4. Open a Visual Studio Command Prompt and navigate to the `c:\qt\4.6.1-vc` directory
5. Type `configure -platform win32-msvc2008` and follow the instructions
6. Type `nmake` to build Qt.
7. Make sure the following line is added to the system PATH environment variable: `c:\qt\4.6.1-vc\bin` (In Computer → Properties → Advanced → Environment Variables)
8. Download the Visual Studio Qt Addin from http://qt.nokia.com/downloads/visual-studio-add-in
9. Install the VS Qt Addin
10. In Visual Studio under Qt → Qt Options, add the Qt version 4.7.1-vc: `c:\qt\4.6.1-vc`

EGR

1. Create the `c:\egr` directory
2. Copy the the files `tracker.cfg` and `3gest.egrdf` to this directory

Train a new posture

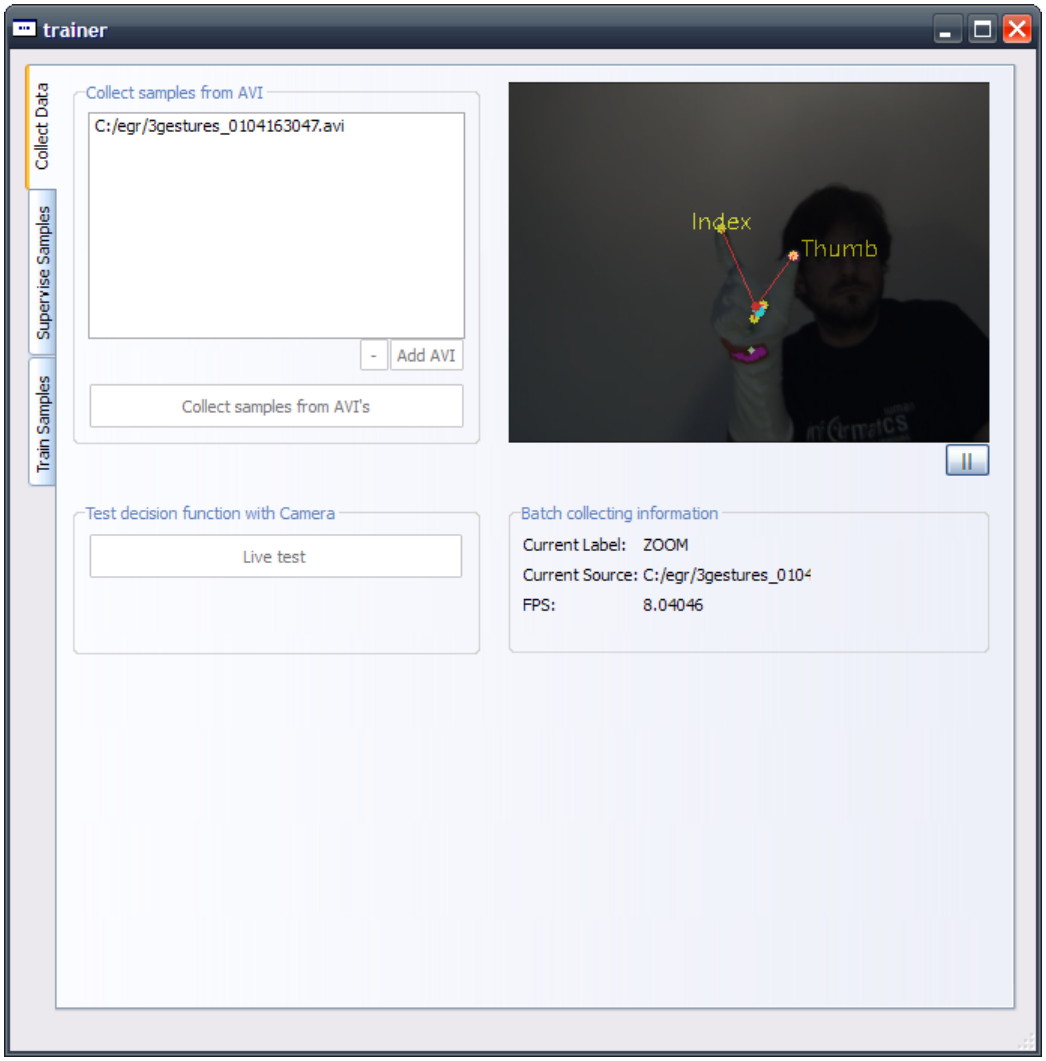
1. Record a video with `recorder.exe` (in `EGR\Release`)



2. Edit the posture labels with a subtitle editor (i.e. Subtitle Workshop from `http://www.urusoft.net`): As soon the posture is visible as it is intended to be captured, set the beginning of a new subtitle label. Set the end of the subtitle label when the posture is not anymore in the intended arrangement.

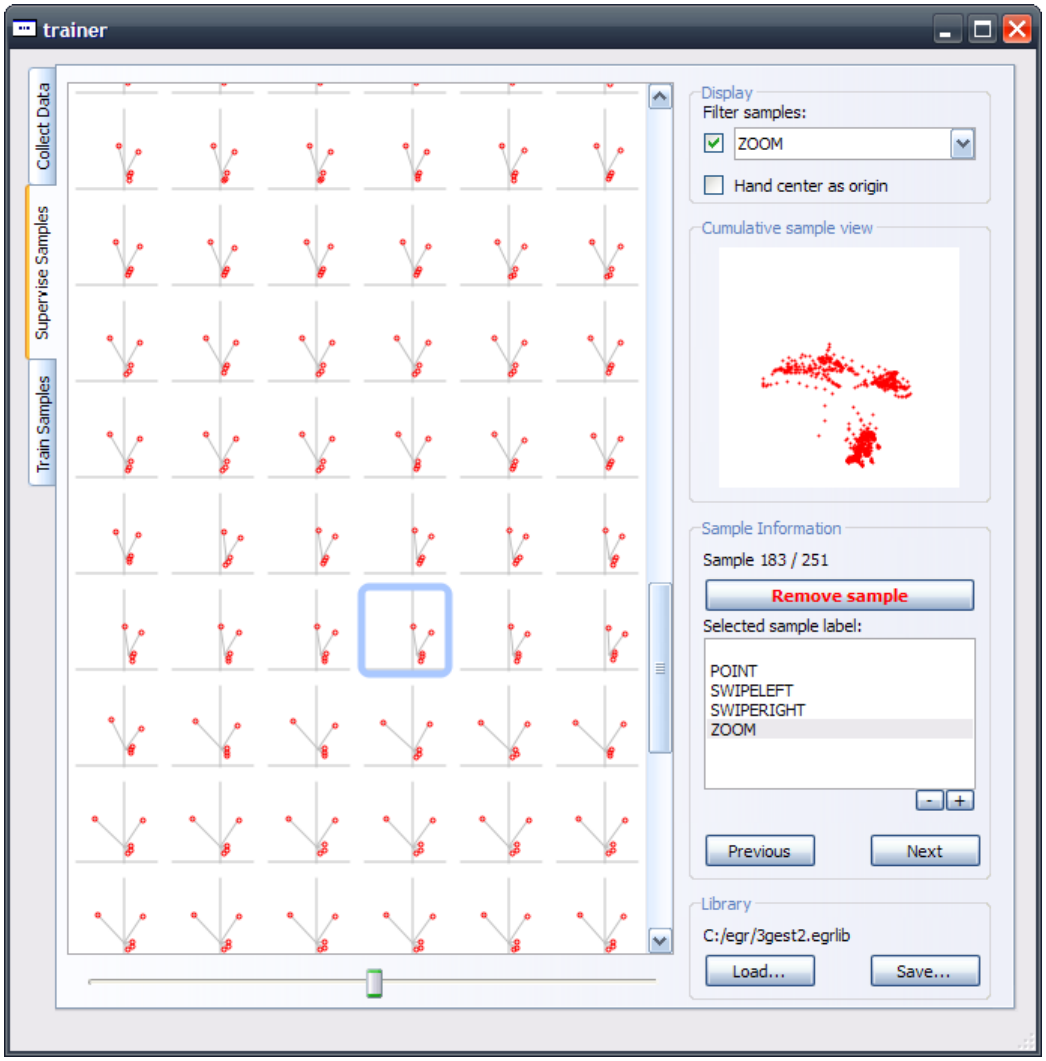
3. Save the posture subtitle file in SubRip (*.srt) format, with the exact same name as the video file.

4. Launch `trainer.exe` (in `EGR\Release`)

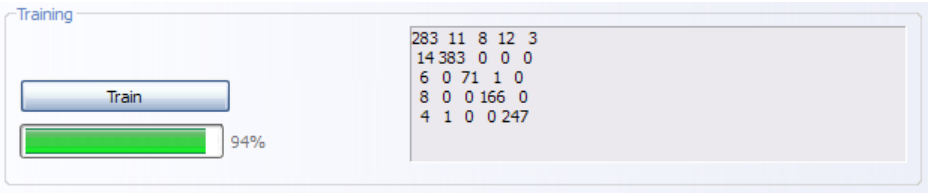


5. Add the Video file and click *Collect Samples from AVI's*

6. When the sample collection is finished, go to the *Supervise samples* tab and save the samples library.



7. Make sure one label class contains all and only correct postures:
   - erase postures with coordinates far off the usual area
   - navigate with a click on the posture representation or with *Previous* and *Next*
   - change the label of a posture sample by clicking on the correct label in the list *Selected sample label*
8. Save the supervised sample library again and go to the *Train Samples* tab



9. Click Train. When the training is finished the progress bar indicates the accuracy in a cross-validation and the text area contains the confusion matrix of the trained decision function

10. Save the decision function

Create new tracker filter

1. Create a new C++ class for `NewFilter` (`NewFilter.cpp` and `NewFilter.h`) in the tracker project.

2. `NewFilter.h`:

```
#pragma once
#include "filter.h"
#include "global.h"

class NewFilter :
  public Filter
{
public:
  NewFilter(void);
  ~NewFilter(void);
  NewFilter* clone();
  IplImage* filter(std::vector<IplImage* > images);
}
```

3. `NewFilter.cpp`:

```
#include "stdafx.h"
#include "NewFilter.h"

NewFilter::NewFilter(void)
{
  name = "New Filter"; //this name is displayed in the
     overlay
  sourceCount=1; //number of input images to the filter
    params.push_back(new Parameter(0.0f, 1.0f, 5.0f)); //
       specify range (minimum, lower bound, upper bound,
       maximum) with 4 float parameters
}

NewFilter::~NewFilter(void)
{
}

NewFilter* NewFilter::clone(){
  NewFilter *nf = new NewFilter();
    nf->params.clear();
    for(int i=0;i<(int)params.size();i++)
```

```
        nf->params.push_back(new Parameter(this->params[i])
           );
    return nf;
}

IplImage* NewFilter::filter(std::vector<IplImage*> images){
  IplImage *img = 0;
  if(images.size()>0)
    img = images[0];

  Parameter *p = params[0];
  float value = p->getVal();
  if(p != 0 ){
    //do something
  }
  return img;

}
```

4. Add the Filter to the `initFilters()` method in `main.cpp`. Make sure the name of the filter and the filter object have the same index in the `strFilterList` and the `filterList` vectors!

```
void initFilters(){
  strFilterList.push_back("New Filter");
  strFilterList.push_back("Empty Filter");
  strFilterList.push_back("Lab Threshold Filter");
  strFilterList.push_back("Erosion Filter");
  strFilterList.push_back("Dilation Filter");
  strFilterList.push_back("Blob Filter");
  strFilterList.push_back("Add Filter");
  strFilterList.push_back("Threshold Filter");
  strFilterList.push_back("RGB Threshold Filter");
  strFilterList.push_back("Split Filter");
  strFilterList.push_back("Fast Corner Filter");
  strFilterList.push_back("Smooth Filter");

  filterList.push_back(new NewFilter());
  filterList.push_back(new EmptyFilter());
  filterList.push_back(new LabThresholdFilter());
  filterList.push_back(new ErosionFilter());
  filterList.push_back(new DilationFilter());
  filterList.push_back(new BlobFilter());
  filterList.push_back(new AddFilter());
  filterList.push_back(new ThresholdFilter());
  filterList.push_back(new RGBThresholdFilter());
```

```
    filterList.push_back(new SplitFilter());
    filterList.push_back(new FastCornerFilter());
    filterList.push_back(new SmoothFilter());


}
```

5. Include the new filter header file in `tracker.h`

```
...
#include "NewFilter.h"
#include "EmptyFilter.h"
...
```

Write application with EGRHand

1. In Visual Studio, create a new Project within the EGR solution
2. Under Project Properties → Configuration Properties → C/C++ → General, add the following Include directories:
   - `$(SolutionDir)\model`
   - `$(SolutionDir)\trainer`
   - `c:\dlib-17.34`
3. Under Project Properties → Configuration Properties → Linker → General, add the following Library directory: `$(OutDir)`
4. Under Project Properties → Configuration Properties → Linker → Input, add the following library dependecies:

```
model.lib
cv200.lib
highgui200.lib
cvaux200.lib
cxcore200.lib
cxts200.lib
ml200.lib
opencv_ffmpeg200.lib
```

5. Include the following header files:

```
#include "EGRHand.h"
#include "egr_dlib.h"
#include "dlib/svm.h"
#include <iostream>
```

6. The EGRHand object is initialized as follows:

```
EGRHand *hand = createEGRHand();
//blobs_5: the finger blobs are detected in step 5
//blobs_6: the wrist blob is detected in step 6
hand->init("c:\\egr\\tracker.cfg", "blobs_5", "blobs_6");
//load the decision function
ifstream fin("c:\\egr\\3gest.egrdf", ios::binary);
deserialize(df, fin);
hand->setClassifier(df);
hand->start();
```

7. Retreive information from the EGRHand object as follows:

```
IplImage *tracker_frame = (IplImage *) hand->
   getTrackerImage("Screen 8");
string posture = hand->getClassifiedPosture();
```

8. Take a look at the methods `handleGesture(string)` and `filterPostures(string, int)` in the mouse project of the EGR solution.