Universitat Politècnica de Catalunya
Facultat d'Informàtica de Barcelona

Final year project

Human-Computer interaction using hand gesture recognition

Author:
Pin-Sho Feng

Supervisor:
Joan Climent Vilaró

March 15, 2010

I would like to thank my supervisor for his sincere dedication and my friends who have shown interest in what I was doing, especially my friend Alan for testing the final application.

Contents

Chapter 1

Introduction

Gesture tracking and recognition is relatively new within the field of object recognition, even though tracking objects has been studied for a long while already. In fact, it has only been the latest years when it has really taken a leap forward and is being increasingly used in commercial systems. Some known examples today which make use of related technology are Microsoft's Project Natal or Sony's EyeToy in the gaming industry.

Aside of gaming, the technology can have as many applications as the human can imagine with activities involving interaction with a computer with our bare hands. One field in which this technology is becoming increasingly useful is medicine, where it finds numerous applications. Reilly and Hanson (1995) created a three-dimensional contactless motion sensing device which patients with physical disabilities, for example, could interact with the PC with the movements of their head or limbs, converted into the movements of a mouse device, which as we will see is the functionality we are looking forward to implement with easier means. Wachs et al. (2008) implemented a gesture recognition system for browsing radiology images without touching any device helping them not to lose their focus of attention (figure 1.1). Another application would be interpreting sign languages, which Lockton and Fitzgibbon (2002) (American sign language) and Holden and Owens (2003) (Australian sign language) have researched.

Gesture recognition is usually combined with tracking, which finds applications in many other areas such as videoconferencing. Askar et al. (2004) implemented segmentation of moving hands as part of a project which involved 3D videoconferencing where it would help generating the virtual environments. Ramani et al. (2001) used multiple target tracking to keep track of the location of the people in a videoconferencing room, but we can see that gesture recognition would further enhance the experience letting participants control typical functions such as skipping slides.

It should be noted that it is necessary to differentiate between static gesture recognition and dynamic gesture recognition. Our project fits in the former, which is about recognizing hand postures, while the latter is about recognizing motion patterns, which is out of our scope.

With this project we intend to produce a proof of concept with which a human can interact with a computer by means of a webcam and his hand. Specifically, the idea is to be able to use our hand as a mouse device, which means a person should be able to move the mouse pointer and

Figure 1.1: Two doctors browse brain images using Wachs et al. (2008)'s gesture driven user interface. Source: Ben-Gurion University of the Negev.

perform mouse functionality, such as a click. We were concerned, however, that being able to do this acurately is difficult and thus we tried to set realistic goals and further improve the system once a solid base had been constructed.

The objectives to be accomplished included but were not limited to:

- Hand tracking
  - Skin detection
  - Motion detection for controlling the mouse pointer
- Gesture recognition
  - Rotation-invariant recognition
  - Scale-invariant recognition
  - Left-button click simulation

The expected final result was a demonstration application with which a user can control the mouse pointer with his hand. This application would integrate with the operating system and while it is running the mouse pointer could be controlled with our hand movements and gestures, substituting the common mouse device.

The real applications of this are many and some were already mentioned previously. The result would simply be another form to interact with a computer.

1.1 Project overview

In order to construct a human-computer interface using hand gesture recognition, input hand data is needed. There are several ways to gather this input data, such as using data gloves, 3D shape scanners or cameras. For our project a camera was used since was cheap, easily available and we did not need 3D information, as our recognition system would be based on hand shape.

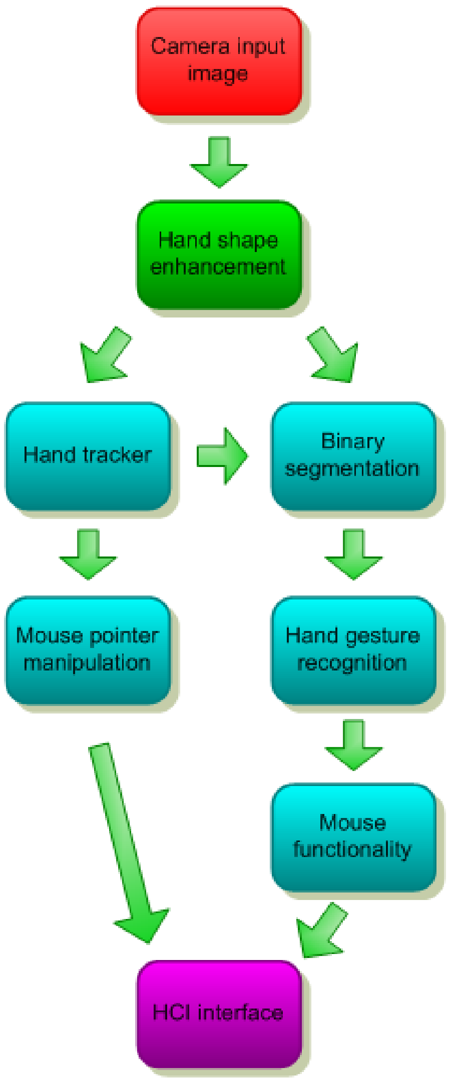


Figure 1.2: Overview of our application. Two parts can be clearly differentiated: hand tracking and gesture recognition. Both are combined to construct a hand gesture-driven human-computer interface.

The input image captured by the camera is processed in order to extract hand shape information in the form of a probability density image. This density image is then fed to the hand tracker and to the gesture recognition system.

The tracker locates the hand on the image and uses this information to control the mouse pointer motion by calling the corresponding functions of the operating system. The tracking system is therefore used to move the mouse pointer.

Mouse functionality (e.g. clicking) is achieved through gesture recognition. The input image is firstly processed using the previously obtained density image in order to obtain a binary representation of the hand. The recognition system then extracts geometrical features from the silhouette of the hand and the corresponding functionality is called depending on the features found.

Combining the tracking system and the gesture recognition system the gesture driven human-computer interface was successfully achieved. The whole process is summarized in figure 1.2.

1.2 Report overview

This report has been divided into eight chapters following an 'introduction, method, results and conclusion' structure. The present chapter and the next are the introduction which deal with the motivation for the project and its planning.

The next three chapters (hand shape enhancement, hand tracking and gesture recognition) form the theory block underneath the final application, in which all three were necessary. The hand shape enhancement chapter describes the method considered to obtain a probability density image of the input hand shape, which is then used by the tracker and the gesture recognition system. The hand tracking chapter explains the theory behind the tracking algorithm, which will in the end enable us to control the mouse pointer using hand motion, as well as how the presence of a hand in front of the camera is detected and the mouse controller logic. And finally, the gesture recognition chapter elaborates on the theory behind the algorithm that was used for the implementation.

The seventh chapter is a small guide to the implemented code, which should be of interest to another developer but the general reader can probably skip it.

Our approach to theory is to explain it only in as much detail as we think is necessary, i.e. what is necessary to understand in order to implement the system. Most noticeably, a difference can be seen between the tracking and gesture recognition chapters since the former has more algorithmic details than the latter, but then it was necessary. In any case, references will be given for further information.

As we wanted to make a clear separation between theory and practice, the results chapter includes not only the final application and how it performs, but also discussion on the theory in practice together with the problems encountered.

The final chapter is the conclusion, which summarizes the project and discusses the degree of success we had in achieving the goals set, as well as possible further work or continuation of the project.

As extras, some appendices which were related to our work but somehow did not fit in the general outline of the report are included at the end. Appendix A deals with the topic of multiple targets tracking, which was considered at an initial stage but dropped in the end, appendix B contains the images that were used for hand shape enhancement and how they were obtained, and finally appendix C contains data relevant for the tracker.

Chapter 2

Planning

This chapter will discuss the planning required for the project, including organization, the methodology used, the hardware and software required and the time schedule, as well as the total cost of the project.

2.1 Organization

The project involved basically two people: the student and the supervisor. Project development was controlled by the student, while the supervisor provided guidance and supervised the work in order to ensure it was progressing adequately.

The communication method used was mainly the e-mail, which was used to exchange documents and report about the project status, as well as to schedule meetings, which occurred usually once a week.

2.2 Methodology

The objective of the project was to produce a piece of software that could achieve the goals we had proposed, thus the methodology chosen had to be specific to software development.

Due to the initial uncertain nature of our project, meaning that we could not be sure about how far it could go, we chose an **iterative and incremental development** approach, which responds to the weaknesses of the classic waterfall model. This kind of approach consists of a cyclic development process and is essential in processes such as the Rational Unified Process, Extreme Programming and Agile software development methodologies.

The classical waterfall model consists of a sequential software development process with a series of stages that succeed one another. These stages are Conception, Initiation, Analysis, Design, Construction, Testing and Maintenance. While they follow a logical order, the main problem of this model is that it is inefficient in time. All the stages require a certain amount of time and if any error is discovered at any stage or any requirement has to be added it would imply returning to the corresponding stage and start over again. An extreme case would be, for example, that in Testing it is discovered that the program lacks some required functionality. In that case the project would have to go again to the initiation stage and progress through all the stages.

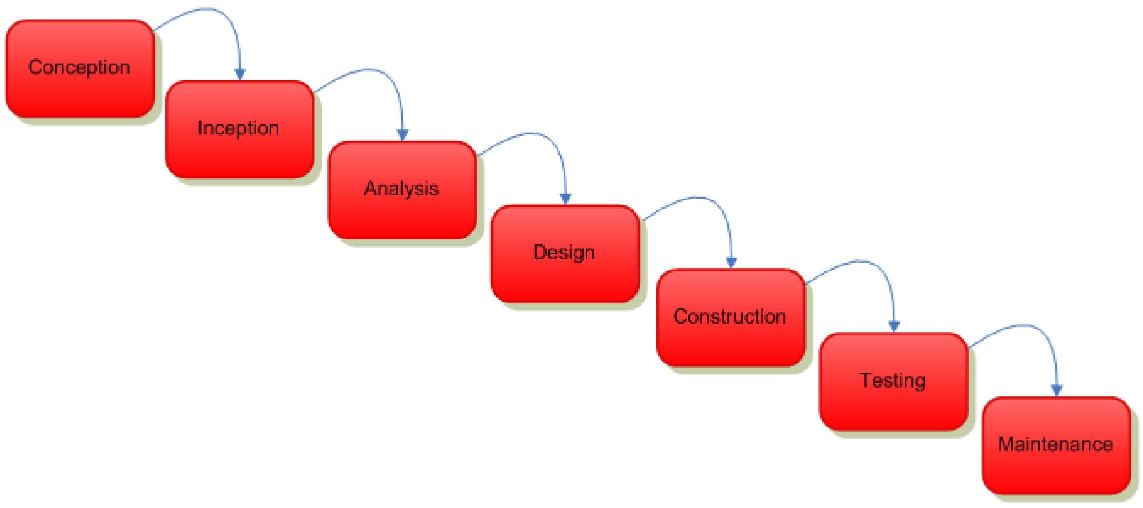


Figure 2.1: The waterfall software development model.

On the other hand, in the iterative and incremental development process the initial stages and the last stages are not separated so far, which makes it much easier and faster to correct errors and add features as the development progresses. The iterative process consists of the same stages as the waterfall process, with the difference that they are carried much faster, as each iteration focuses on individual parts of the project and the results obtained are used for the next iteration, which allows fast prototyping in order to test the analysed and designed components. For the hand tracker, for example, we first developed a prototype in order to test its functionality and ensure that it was working as expected, before finally using this tracker to follow a hand in real-time.

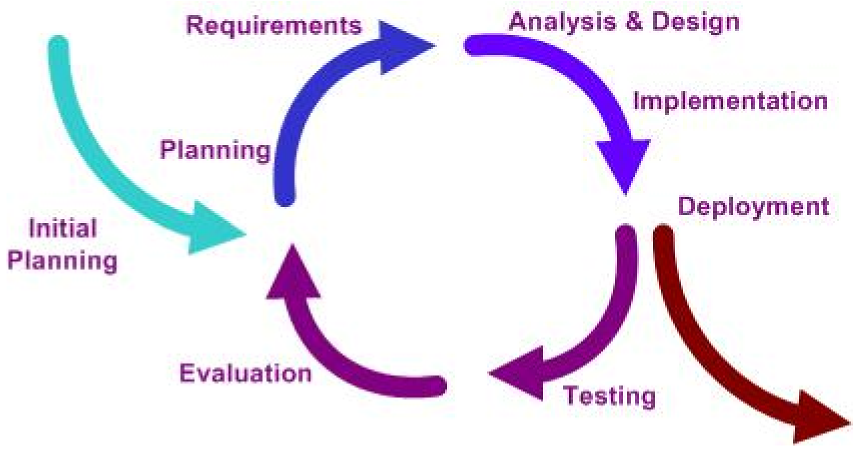


Figure 2.2: The iterative and incremental software development model.

In our project mainly three iterations were used in order to develop the application, as can be seen in section 2.4.1.

2.3 Environment setup

This section will describe the hardware and software used for the project and the reason why they were chosen.

2.3.1 Hardware setup

Having set the goals for the project, we knew that we needed to track the motion of a hand and interpret hand gestures.

Since the main objective was to be able to control the mouse pointer with the movement of a hand in space it was not necessary to use any 3D motion capture device, even though a hand is naturally a three-dimentional object. Also, we knew that we were interested in scale-invariant recognition, thus we did not need image depth information for this either.

If we did not need depth information, a single camera was enough for the task so a **simple USB webcam** was chosen for the task. Specifically, a '**Creative Live! Optia**'. It was not chosen for any particular feature, just because it was easily accessible to us and the Linux distribution we used had drivers for it that worked out-of-the-box.

The computer used for the project was a Lenovo T61 laptop equipped with an **Intel Core 2 Duo 2400MHz with 2GB of memory**, working in 32bits mode. Despite the availability of two CPU cores, the project was not developed with paralellism in mind.

We wanted it to work in a controlled environment, but we also wanted this environment to be as real and common as possible. The place of work was an average **office room with fluorescent lamps**, which was not precisely the most adequate lighting for a uniform illumination, but our intention was to make the application robust under different lighting conditions (as long as the place would remain reasonably lit, of course). The webcam could be placed on any horizontal surface facing a user's hand.

2.3.2 Software setup

The project was developed using mainly open source tools so as to keep the cost to a minimum. The underlying operating system was Kubuntu 9.10 and the language chosen was **C++** together with **Qt 4** and **OpenCV 1.1** libraries.

The programming language was chosen for performance since OpenCV was already written in C and Qt in C++. The compiler used was **GCC-4.4** with optimization options always activated. Qt, a user interface library, was chosen for its integration in Kubuntu, well-documented classes and portability across platforms, even though this last feature was not exploited. OpenCV is a known open source Computer Vision library developed by Intel and it provided implementations for most common algorithms used in the field. In the middle of the project the 2.0 version of the library was released, offering many improvements over the previous one. However, in our tests it performed slower for our application so we stayed with 1.1.

Other software used:

- **Matlab 2008** and **Octave 3.2.2** for simple tests and data plots.
- **The Gimp 2.6.7** and **Inkscape 0.47** for image and vector graphics creation and editing.
- **OpenOffice Calc 3.1.1** for data calculations.
- **OpenOffice Presentation 3.1.1** for preparing the project presentation.
- **Kile 2.0** for writing the final report (Latex environment).
- **Microsoft Office Visio 2003** for flowcharts.
- **Microsoft Office Project 2007** for time scheduling.

2.4 Time and costs

This section will discuss the time and costs of the project. Firstly, an estimation of the project schedule will be shown based on the initial planning. Secondly, the estimated costs will be divided into hardware, software and human resources costs. Finally, a comparison of the estimated cost and the real cost will be shown.

2.4.1 Estimated time schedule

The initial planning of the project considered a semester for the realization of the project. This semester consisted of five months, starting from the 1st of September 2009 to the 31st of January. The project was then scheduled considering this time span, even though in the end a little more time was needed.

The first month would be dedicated to the inception of the project as well as acquiring the knowledge necessary to develop the project. In the next three months one development iteration each month in order to complete all the parts of the application. Finally, in January the application is expected to be finished and final testing is done. Afterwards, the remaining time is used to prepare the final report as well as the presentation. This **time schedule** can be seen in figures 2.10 and 2.11 (last two pages of the chapter).

Considering an average dedication of 6 hours a day, the total amount of hours required is broken down in figure 2.3.

2.4.2 Estimated costs

Hardware

The only hardware used were a laptop and a webcam. This hardware was not only used for the project, thus summing their cost to the total costs would be inaccurate.

Inception	54 hours
Research and learning	78 hours
Iteration 1	132 hours
Iteration 2	132 hours
Iteration 3	132 hours
Final testing	60 hours
Documentation and presentation	102 hours
Total	**690 hours**

Figure 2.3: Estimated time required for the project

- Creative Live! Cam Optia - 74 €.
- Lenovo T61 7664R5G - 1630 €.

Assuming that the average life of a webcam and a laptop is 5 years, and that the camera is used for 1 hour a day and the laptop for 6 hours a day during 22 days of a month, the following cost per hour can be derived:

$$Cost_{camera} = \frac{74€}{5years * 12months/year * 22days/month * 1hours/day} = 0.056€/hour$$

$$Cost_{laptop} = \frac{1630€}{5years * 12months/year * 22days/month * 6hours/day} = 0.206€/hour$$

Considering figure 2.3, the camera was used for 396 hours (the three iterations) while the laptop was used for the entire project. The entire hardware cost can then be calculated and is shown in figure 2.4.

Creative Live! Cam Optia	396 hours	22.18 €
Lenovo T61 7664R5G	690 hours	142.14 €
Total		**164.32 €**

Figure 2.4: Estimated hardware cost required for the project

Software

The software used was mostly open source and the propietary software used, namely Matlab, Microsoft Office Visio and Microsoft Office Project were either available at the university or could be freely downloaded. In the case of Visio and Project they could be freely downloaded thanks to an academic alliance between the faculty and Microsoft, which makes their software free for the faculty's students.

This means that the total cost of the software used was exactly **0 €**.

Human resources

In order to calculate the cost of the human resources we considered the cost of the different roles involved in the project. Figure 2.5 shows the roles and the cost associated. All the roles are performed by the student except for the role of project supervisor.

Project Manager (M)	48 €/hour
Project Supervisor (S)	42 €/hour
Software Analyst (A)	40 €/hour
Software Architect (AR)	40 €/hour
Programmer (P)	30 €/hour
User (U)	15 €/hour

Figure 2.5: Estimated costs for each participating role

Then, considering the cost of the different roles and their percentage of participation the final human resources cost can be estimated, as is shown in figure 2.6

Inception	54 hours	20%S, 40%M, 40%A	2354.4 €
Research and learning	78 hours	40%A, 30%AR, 30%P	2886 €
Iteration 1	132 hours	5%S, 5%M, 18%A, 18%AR, 50%P, 4%U	4554 €
Iteration 2	132 hours	5%S, 5%M, 18%A, 18%AR, 50%P, 4%U	4554 €
Iteration 3	132 hours	5%S, 5%M, 18%A, 18%AR, 50%P, 4%U	4554 €
Final testing	60 hours	5%M, 15%A, 15%AR, 15%P, 50%U	1584 €
Documentation and presentation	102 hours	15%S, 85%A	4110.6 €
Total human resources cost			**24597 €**

Figure 2.6: Estimated human resources cost required for the project

Total cost

Considering the previous sections, the total estimated cost was:

Hardware	164.32 €
Software	0 €
Human resources	24597 €
Total	**24761.32 €**

Figure 2.7: Total estimated cost

The total estimated cost of the project is approximately **24761 €**.

2.4.3 Real time schedule and costs

The planning was more or less followed without major problems except for the last tasks, consisting of the preparation of this report and the presentation. The preparation of the report took two weeks more than expected and, in addition, an illness of the student impeded progress for further two weeks, accounting for a total delay of a month (but two weeks of extra work).

The software costs remained the same as they covered all the requirements. With respect to the hardware no equipment was damaged, thus the increment in the cost only accounted for the extra time it had to be used.

Two weeks is equivalent to 60 hours of work. This 60 hours of work were exclusively for documentation and presentation, thus summing a total of 162 hours for this phase and a new cost of 6528.6 €. In the case of the hardware, the laptop cost increases to 154.4 €. The difference between the estimated cost and the final real cost, which is shown in figure 2.9.

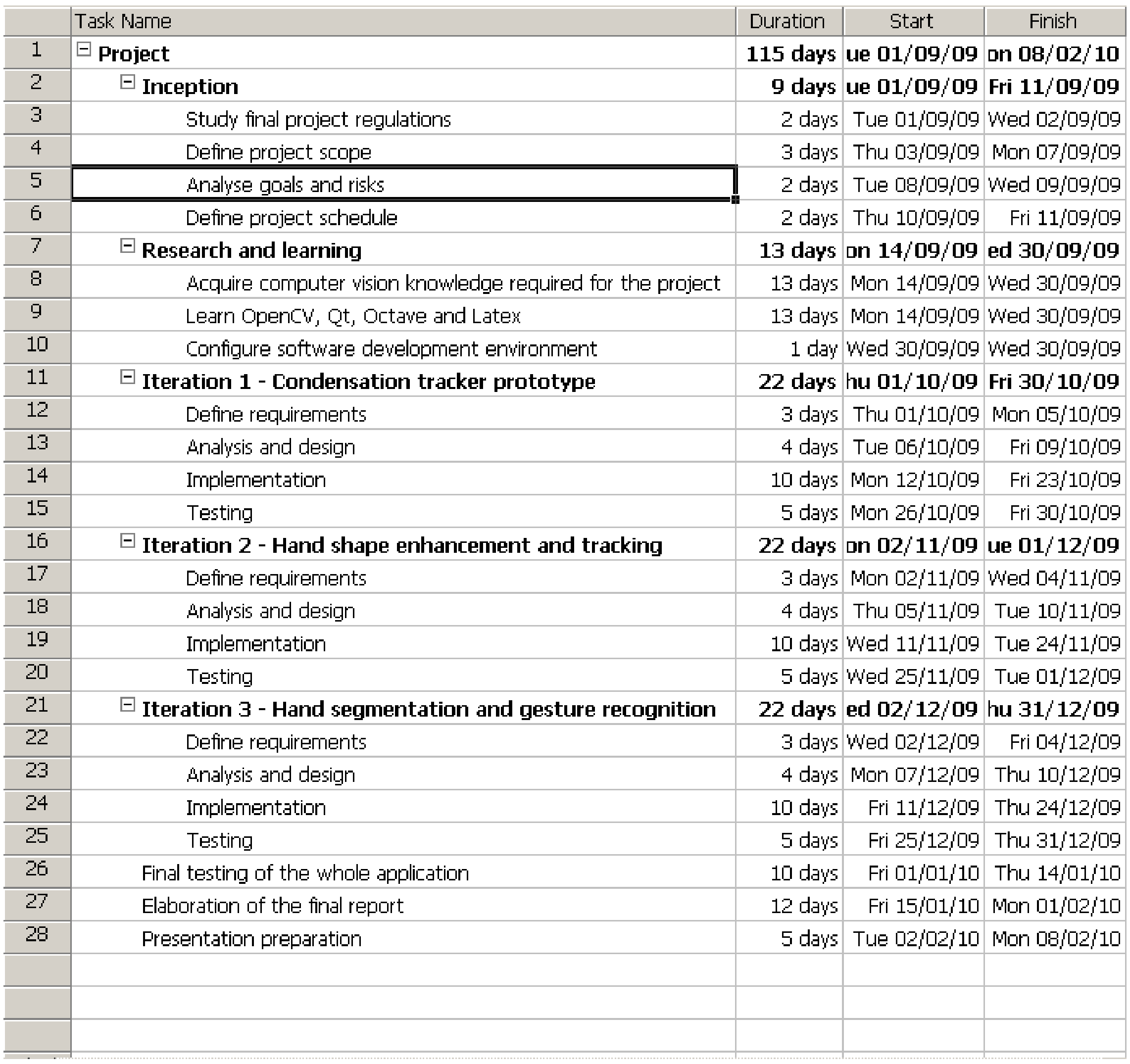
	Estimated time	**Real time**
Inception	54 hours	54 hours
Research and learning	78 hours	78 hours
Iteration 1	132 hours	132 hours
Iteration 2	132 hours	132 hours
Iteration 3	132 hours	132 hours
Final testing	60 hours	60 hours
Documentation and presentation	102 hours	162 hours
Total	**690 hours**	**750 hours**

Figure 2.8: Estimated time required for the project

	Estimated cost	**Real cost**
Hardware	164.32 €	176.78 €
Software	0 €	0 €
Human resources	24597 €	27015 €
Total	**24761.32 €**	**27191.78 €**

Figure 2.9: Comparison between the estimated cost and the real cost

In conclusion, the total amount of work required **750 hours** and an economic cost of approximately **27192 €**.



	Task Name	Duration	Start	Finish
1	⊟ **Project**	**115 days**	**ue 01/09/09**	**on 08/02/10**
2	⊟ **Inception**	**9 days**	**ue 01/09/09**	**Fri 11/09/09**
3	Study final project regulations	2 days	Tue 01/09/09	Wed 02/09/09
4	Define project scope	3 days	Thu 03/09/09	Mon 07/09/09
5	Analyse goals and risks	2 days	Tue 08/09/09	Wed 09/09/09
6	Define project schedule	2 days	Thu 10/09/09	Fri 11/09/09
7	⊟ **Research and learning**	**13 days**	**on 14/09/09**	**ed 30/09/09**
8	Acquire computer vision knowledge required for the project	13 days	Mon 14/09/09	Wed 30/09/09
9	Learn OpenCV, Qt, Octave and Latex	13 days	Mon 14/09/09	Wed 30/09/09
10	Configure software development environment	1 day	Wed 30/09/09	Wed 30/09/09
11	⊟ **Iteration 1 - Condensation tracker prototype**	**22 days**	**hu 01/10/09**	**Fri 30/10/09**
12	Define requirements	3 days	Thu 01/10/09	Mon 05/10/09
13	Analysis and design	4 days	Tue 06/10/09	Fri 09/10/09
14	Implementation	10 days	Mon 12/10/09	Fri 23/10/09
15	Testing	5 days	Mon 26/10/09	Fri 30/10/09
16	⊟ **Iteration 2 - Hand shape enhancement and tracking**	**22 days**	**on 02/11/09**	**ue 01/12/09**
17	Define requirements	3 days	Mon 02/11/09	Wed 04/11/09
18	Analysis and design	4 days	Thu 05/11/09	Tue 10/11/09
19	Implementation	10 days	Wed 11/11/09	Tue 24/11/09
20	Testing	5 days	Wed 25/11/09	Tue 01/12/09
21	⊟ **Iteration 3 - Hand segmentation and gesture recognition**	**22 days**	**ed 02/12/09**	**hu 31/12/09**
22	Define requirements	3 days	Wed 02/12/09	Fri 04/12/09
23	Analysis and design	4 days	Mon 07/12/09	Thu 10/12/09
24	Implementation	10 days	Fri 11/12/09	Thu 24/12/09
25	Testing	5 days	Fri 25/12/09	Thu 31/12/09
26	Final testing of the whole application	10 days	Fri 01/01/10	Thu 14/01/10
27	Elaboration of the final report	12 days	Fri 15/01/10	Mon 01/02/10
28	Presentation preparation	5 days	Tue 02/02/10	Mon 08/02/10

Figure 2.10: Estimated project schedule calendar.

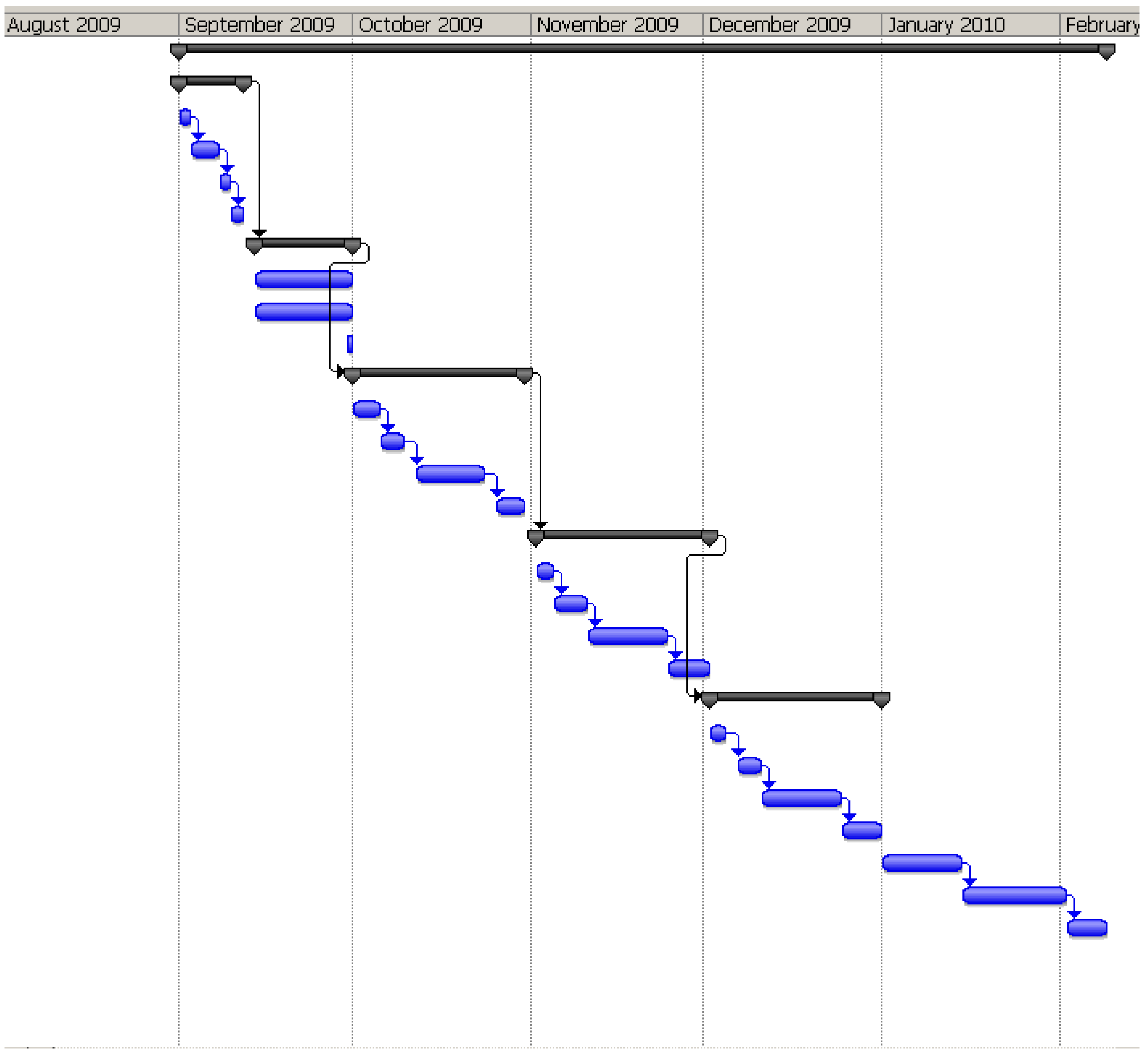


Figure 2.11: Estimated Gantt diagram of the schedule.

Chapter 3

Hand shape enhancement

Before we started tracking and recognizing hand gestures we preprocessed the image so as to obtain the hand probability density image (figure 3.1). This probability refers to the probability that a pixel is of skin colour. We refer to this process of obtaining the hand probability density image as hand shape enhancement.

The mentioned image skin probability density image is used in two different ways. On one hand it is fed to the *tracker*, which will use it in the measurement stage, as will be seen in the next chapter. On the other hand, it is also used for *binary segmentation* before gesture recognition (chapter 5).

We chose to use *skin colour* instead of other properties for its specificity (we are interested in a moving hand, not other moving objects), which has been proven useful and robust for face detection and tracking. It has the advantage to be fast and invariant to changes in shape and orientation. Furthermore, the human skin colour has been found to have special characteristics which make it easier to be distinguished.

However, it must be pointed out that the enhancement process alone is prone to error (results are noisy), so while it is helpful in assisting another process it cannot usually be completely relied upon, which is the reason we used a probability based tracking algorithm (see chapter 4).

In the process of enhancement there are mainly two issues on which to make a decision. Firstly, the colour space that will be used and secondly, the distribution of the skin colour in this colour space. Colour space is important as robustness to changes in light conditions as well as good colour resolution are desirable features and thus, colour spaces which separate chrominance (the 'real' colour information) from luminance are favoured against those which do not. The distribution of the skin colour refers to the values in the colour space within which a sample is considered 'skin' and varies depending on the camera (due to different CCDs and lenses).

Colour spaces

The most common colour spaces in literature are RGB, HSV (or HSI or HSL) and YCrCb (Vassili et al., 2003)(Kakumanu et al., 2007).

RGB is the most popular colour space in computer graphics due to its use in CRT screens, which

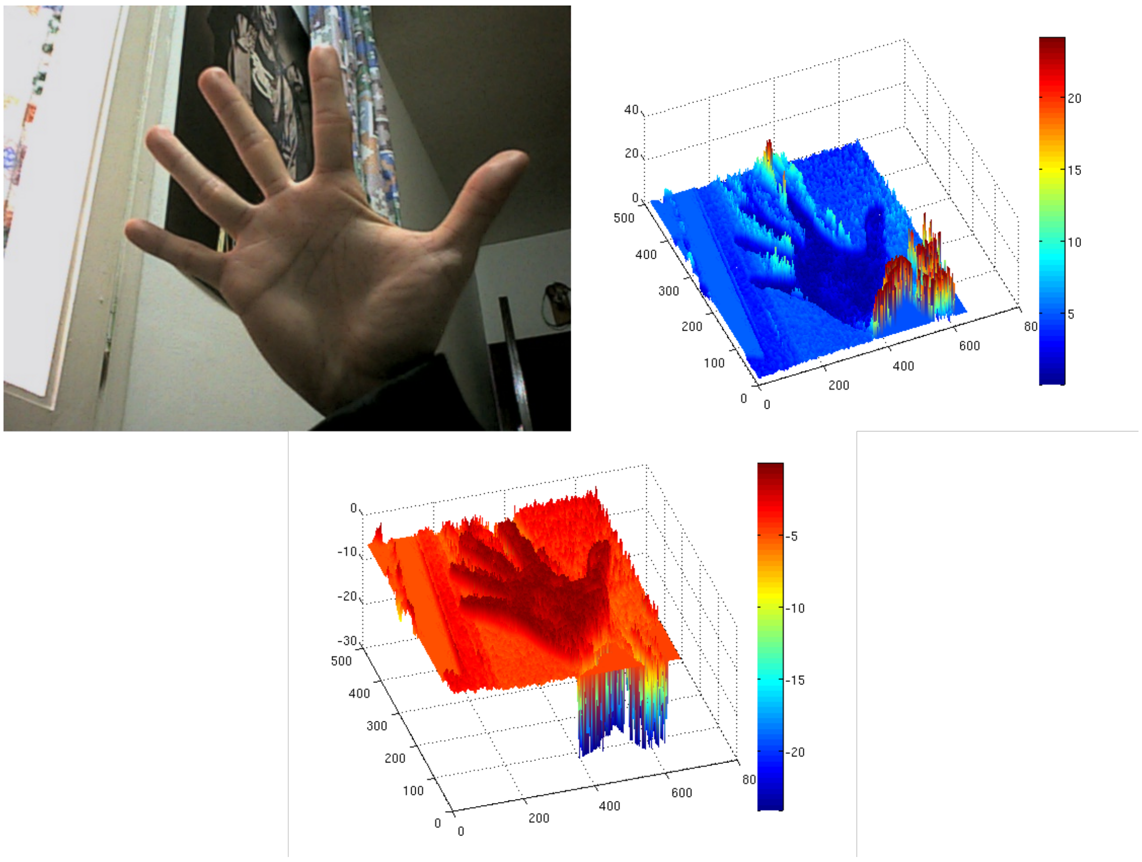


Figure 3.1: An image of a hand (*top-left*) and its skin colour density (*top-right*) in terms of Mahalanobis distances to the mean skin colour in HSV colour space (less means higher skin probability). For illustration purposes, the symmetric image with respect to the XY plane is shown at the *bottom*.

combined the primary colours (red, green and blue) of three rays to produce different colours. Most digital cameras also capture images in RGB. It is the most basic colour space as the rest are based on it and has been favoured by several authors such as Jones et al. (1999) and Peer et al. (2003) for its simplicity. However, RGB has the drawback of mixing chrominance and luminance. This problem is reduced in normalized RGB space, which consists of dividing each colour component by the sum of all three components and is favoured by other researchers (Brown et al., 2001)(Storring et al., 2003).

HSV (Hue, Saturation, Value) and similar variants (HSI - Intensity, HSL - Lightness) are colour spaces which aim to separate chrominance from luminance. They describe colours based on perceptual features to a user's eye such as tint, saturation and tone. *Hue* defines the dominant colour in an area and refers to pure colours without tint or shade (red, green, blue or yellow). *Saturation* defines the colourfulness of a colour, which is the difference of a colour against its own brightness. *Value* defines the amount of brightness perceived from the colour. Intuitively, what this means is that any object has the same hue and saturation except for a different value under different lighting conditions. This clear separation between luminance and chrominance makes this space highly popular (McKenna et al., 1998)(Zhu et al., 2004)(Deriche and Naseem, 2007). Further insight will be given in section 3.1.2 since we eventually used it as well.

YCrCb is used image/video processing and separates chrominance from luminance by a simple transform involving a weighted sum of RGB colours for luminance (Y) and basic operations between colours for chrominance (Cr and Cb). Its simplicity and efficiency has made it a very popular choice for skin detection (Phung et al., 2002)(Hsu et al., 2002).

Apart from these colour spaces, there are others which attempt to find a set of relevant colour features. The skin has been found to contain significant levels of red, which some transformations from RGB intend to take advantage of.

Skin colour distribution

In our choice of the enhancement algorithm, we had to take efficiency especially into account since the application was expected to work real-time or close and we already knew the tracking algorithm would be resource consuming. Simplicity was also an important factor, so methods which required heavy training with large sets or complex skin models were not considered.

Skin colour distributions can be modelled non-parametrically or parametrically. Methods of the former type estimate the skin colour distribution directly from the training data without deriving a model, while the latter derive a model which they use afterwards to classify skin. As Vassili et al. (2003) points out, non-parametric methods need considerably more storage space and have high dependence on the representativeness of the training data used, even though they are fast and independent to skin colour distribution shape. Parametric methods, on the other hand, can generalize the model, need much less training data, do not depend so much on the representativeness of this data and require little storage while being fast as well. For these reasons, a parametric method was chosen.

In the section to follow, we will introduce the skin detection method we used and the choice of colour space.

3.1 Single Gaussian skin colour distribution

The human skin colours tend to cluster in a small region of the colour space under controlled lighting conditions (Kakumanu et al., 2007) (see figure 3.2). One parametric way to model this is by using an elliptical Gaussian joint probability density function:

$$p\left(c_{i,j}\right) = \frac{1}{\sqrt{2\pi}\sqrt{|\Sigma|}} \exp\left[-\frac{1}{2}D_{i,j}\right] \tag{3.1.1}$$

$c_{i,j}$ is the colour vector of a pixel where $i, j \in [1...n]$ and n is the number of pixels in the image. $D_{i,j}$ is the square root of the Mahalanobis distance, defined as

$$D^2_{i,j} = (c_{i,j} - \mu)^T \Sigma^{-1} (c_{i,j} - \mu) \tag{3.1.2}$$

where μ and Σ are the distribution parameters, namely mean (μ) and covariance matrix (Σ) respectively estimated over *skin* colour samples c_k, where $k \in [1...m]$ and m is the number of skin colour samples:

$$\mu = \frac{1}{m}\sum_{k=1}^{m} c_k$$

$$\Sigma = \frac{1}{m-1}\sum_{k=1}^{m}(c_k - \mu)(c_k - \mu)^T$$

The Mahalanobis distance $D_{i,j}$ was used as the probability measure of the skin-likelihood of a pixel (i, j). Such distances were calculated for all the pixels in the input image and the result obtained was the **skin probability density image** we were looking for, which is later used for tracking and binary segmentation.

An equation similar to 3.1.1 is used in the observation model of the tracking algorithm as a weight function (see section 4.2.6).

Along with the skin colour distribution model, a colour space had to be chosen. Two were considered and their effectiveness was evaluated by comparing their results in binary segmentation (see chapter 7). A small set of training images taken under different lighting conditions were used and are attached in appendix B.

3.1.1 PCA transformation of RGB space

The first colour space we chose was a transformation of RGB space based on a principle component analysis (PCA) of colours, inspired by Holden and Owens (2003), who used it for their hand tracker:

$$(R, G, B) \rightarrow (a, b, c) = (\frac{R+G+B}{3}, R-B, \frac{2G-R-B}{2})$$

From the set of training images, colour means and covariance matrices for different lighting conditions were obtained.

While this method had very good performance in the conditions under which the training images were taken, fast prototyping showed that it was very sensitive to illumination changes and thus we did not look further into it. Another colour space had to be considered.

3.1.2 HSV space

As mentioned in the introduction, the HSV colour space provides a theoretic separation between chrominance and luminance, which is what we are interested if we are after robustness under different illumination conditions. In practice, however, we have seen that this is not entirely true but does provide more robustness than the previous colour space. HSV is also a transformation from RGB so that:

$$\begin{aligned} H &= \arccos\frac{\frac{1}{2}((R-G)+(R-B))}{\sqrt{((R-G)^2+(R-B)(G-B)}} \\ S &= 1-3\frac{min(R,G,B)}{R+G+B} \\ V &= \frac{1}{3}(R+G+B) \end{aligned}$$

Since we are only interested in chrominance, only H and S were used. Histograms (figure 3.2) of the skin samples in the training set under different conditions show their shape can be effectively approximated by Gaussian functions, as mentioned before.

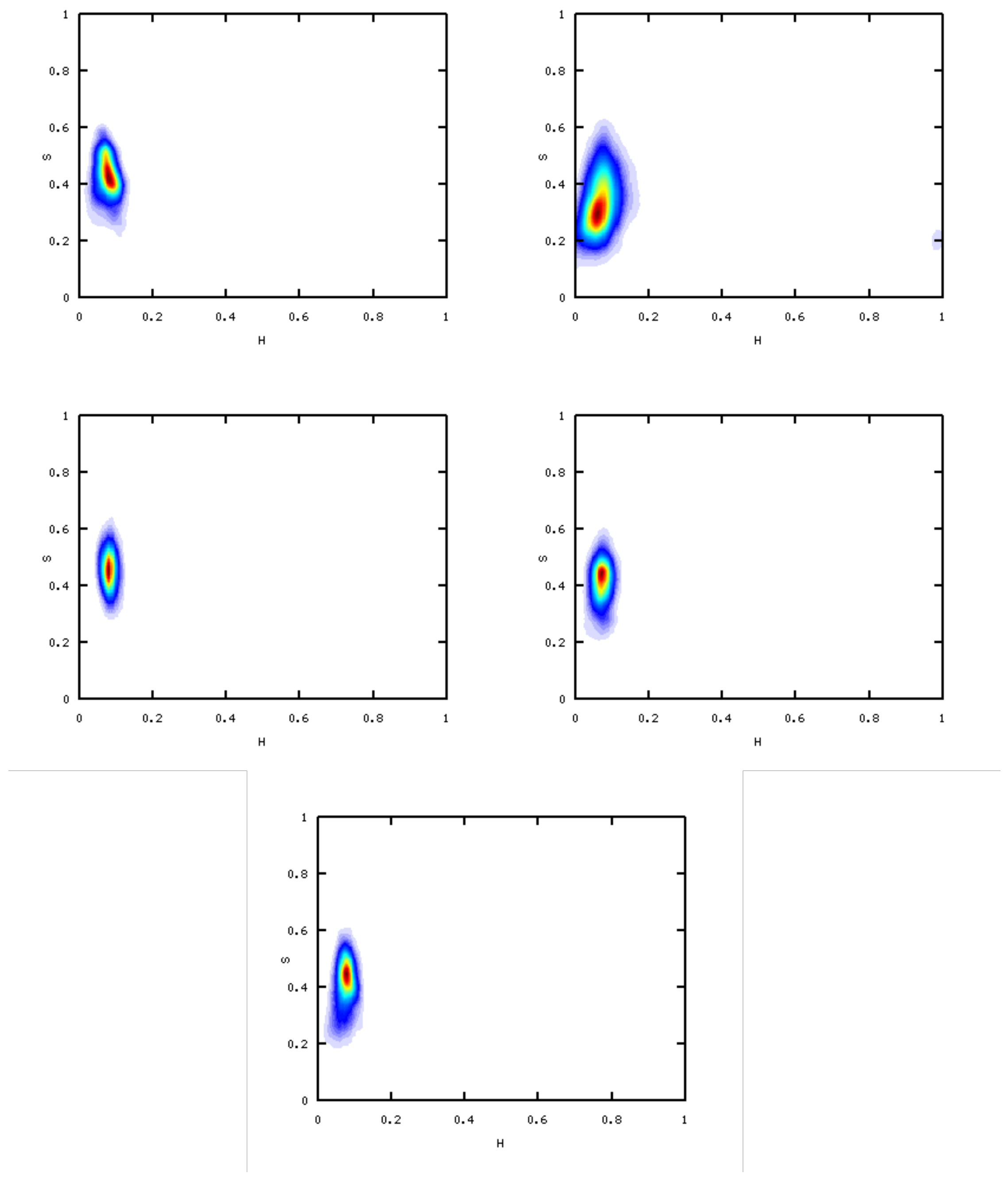


Figure 3.2: Skin colour histograms in HS of the training image set obtained using a number of bins of approximately the square root of the total number of points. We can observe distributions are approximately Gaussian. *Top-left*: daylight under shadow. *Top-right*: home at night, fluorescent light. *Middle-left*: office during the day, fluorescent light. *Middle-right*: office during the night, fluorescent light. *Bottom*: all combined.

Same as in the previous section, colour means and covariance matrices were calculated from the training image set. A combined mean and a covariance matrix were calculated over all the images, without distinguishing between different illuminations. It should be noted that hue has the particularity of being cyclic, so the colour at 0 degrees is basically equivalent to the one at 359 degrees, and the colours which are close are similar. This could be problematic in implementation as the Mahalanobis distances computed from the same colours at different equivalent hue values would be different. To solve this issue, we used an equivalent transformation in Cartesian coordinates suggested by Brown et al. (2001):

$$X = S\cos H, \quad Y = S\sin H$$

Using this new space the found two-dimensional all illumination conditions combined mean and covariance matrix of the training set were:

$$\mu = [0.35920 \quad 0.19378]$$

$$\Sigma = \begin{bmatrix} 0.0055332 & 0.0023075 \\ 0.0023075 & 0.0034047 \end{bmatrix}$$

The individual matrices of each lighting condition can be found in appendix B.

Even though using this colour space had a big impact on performance due to the transformation from RGB (camera images) to HSV, the results with this colour space were better and more robust to illumination changes, thus it was chosen over the other tested colour space.

Chapter 4

Hand tracking

This chapter deals with the part of the project consisting of tracking a hand, which involves being able to follow its trajectory, detect when it is actually visible and eventually control the mouse pointer.

It could be argued that only with segmentation we could do the hand tracking. As Manresa et al. (2000) points out, however, the low quality images that USB webcams produce and varying illumination conditions can often cause errors in the segmentation process. An original image will always provide more information than a binary image and we would like to make maximal use of this information. When we want to track a moving object with a camera we estimate its location at each frame. This estimation is inherently inaccurate due to possible occlusions, shadows, changes in shape, clutter or appearance of other similar objects in the scene.

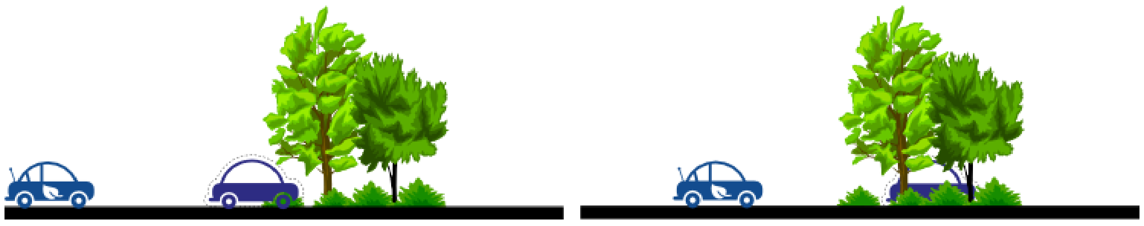


Figure 4.1: Location of two cars at a time step t (left) and at $t + 1$ (right). Suppose we are following the car on the right. Segmentation only would detect that a car has disappeared while a motion estimator would be able to shortly follow the car even though it is occluded.

Suppose that we are tracking the movements of a car. Using the segmentation process we know exactly where the car is but then suddenly the car goes behind some trees. In the next frame, the car has passed the trees but another car appears further back on the same road. Which one is the car we were following? In our minds it is clear because we know how the car we were following was moving but the segmentation process alone cannot differentiate between both. The idea behind motion estimators is to use this information we already know in order to predict the new location of the tracked object in the next frame minimizing the effect of noise. Therefore, they need a motion model for the prediction. A model for our car, for example, could be that we assume it travels at constant velocity and thus we can easily predict its location at a subsequent time step knowing its location at the previous time step.

However, as the name implies already, it is a *prediction* and may be erroneous. We would like to be able to correct this prediction using the data we gather at each frame so that in predictions at subsequent frames this error will not be propagated. Hence, we can see that the tracking process involves two phases: prediction and correction. These tracking systems are called estimators.

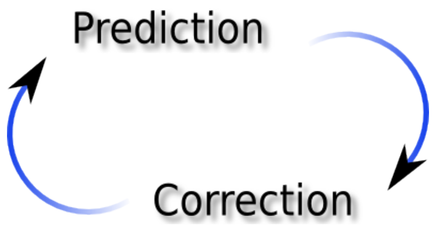


Figure 4.2: The estimation of an object's location is usually achieved through a two-phased process consisting of a prediction and its correction.

The two most used estimators in Computer Vision are the Kalman filter (Kalman, 1960) and the CONDENSATION filter (Isard and Blake, 1998a). The *filter* term refers to them filtering the input data so as to infer the motion of an object. They are both probability based and estimate the **state** of a process (position, orientation, velocity...). The Kalman filter aims at predicting while minimizing the mean of the squared error so that the new model constructed from what we already knew is the most probably correct, while CONDENSATION is a particle filter which combines stochastic methods together with a learned object dynamics model in order to estimate the state.

One important term which will often be used is **density**, which refers to the probability density function which describes the likelihood of the location of the tracked object in the frame.

In the next section we will look into the differences between the estimators relevant to us and why one was chosen over the other.

4.1 Kalman filtering vs Condensation

In this section we will present an informal comparison of the characteristics of both algorithms and why we chose CONDENSATION as our tracking algorithm. We will only discuss practical considerations since a deeper understanding would require a proper introduction to both and we will only describe CONDENSATION (see next section).

For our project, both estimators would be perfectly valid since both would accomplish the objective of tracking a hand. Therefore, the choice of one or another would depend on other factors.

The main difference between both estimators is that the Kalman filter assumes Gaussian densities, which are unimodal and thus cannot represent simultaneous hypotheses of the location of multiple objects, whereas CONDENSATION can be used for tracking multiple targets 'out-of-the-box' (actually there are further complications, but theoretically it allows this possibility). Even though the project did not aim at tracking multiple objects, further extension of the project could consider it so CONDENSATION was more favourable in this aspect. Furthermore, cluttered backgrounds could provide false alternative hypotheses as well.

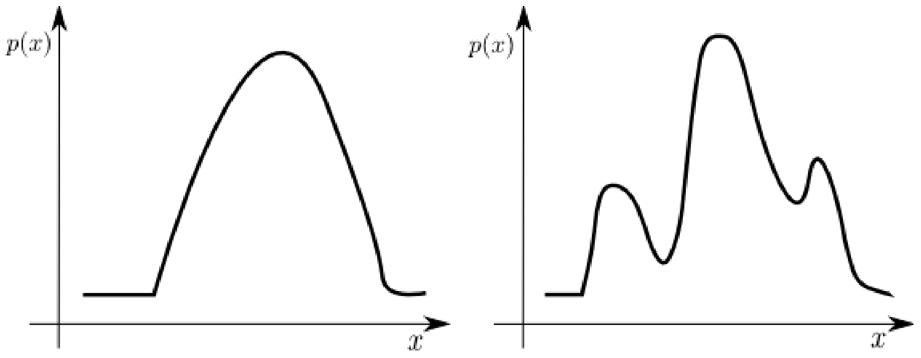


Figure 4.3: Kalman assumes unimodal Gaussian densities (left) while CONDENSATION can deal with multimodal densities (right).

Another difference is ease of implementation. CONDENSATION is considerably simpler and easier to understand than the Kalman.

Accuracy wise, we will refer to Petrie (unknown), who performed several experiments comparing both algorithms which involved the tracking of one or three bouncing balls and compared the accuracy of both. Her results showed that CONDENSATION was more accurate around bounces and tops of the arcs, while in between they had similar accuracy (figure 4.4). Considering that hand movements can be quite random, the higher accuracy at bounces and tops of the arcs suggested CONDENSATION would have higher accuracy for tracking sudden direction changes of a hand. The reader is referred to her website for visual comparisons in video.

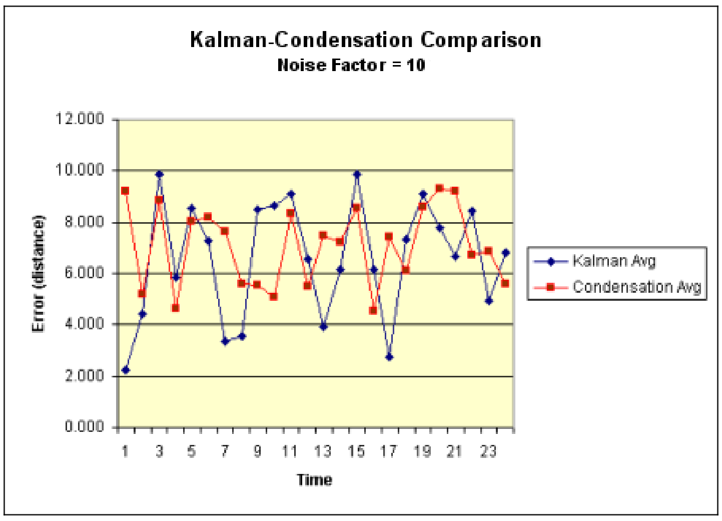


Figure 4.4: Comparison of the Kalman filter and the CONDENSATION algorithm for tracking three bouncing balls. As can be seen in the chart, the two have very similar performance. Image by Petrie (unknown).

As for performance, no profiling has been done to compare them but the stochastic nature of CONDENSATION would suggest it is much slower than Kalman. Nevertheless, CONDENSATION has been widely used in literature for real-time applications.

All in all, CONDENSATION was used for the reasons mentioned.

4.2 Motion tracking using Condensation

Condensation (Isard and Blake, 1998a) stands for "Conditional Density Propagation" and is based on particle filter techniques in order to track objects. The algorithm was originally conceived as a solution to tracking curves in visual clutter. The Kalman filter was not adequate since the cluttered scene could often present multimodal densities and, as stated in the previous section, it was meant for Gaussian densities. Condensation is in this sense more general since it addresses a situation which is more common.

Before going into the algorithm some background theory is necessary.

4.2.1 Propagation of state density

As we are dealing with computers, the propagation through time t will have to be discrete. The state of the object (position and other parameters if needed) to be tracked at time t is denoted as $\mathbf{x}_t$ and the vector of all states throughout time is $X_t = \{x_1, ..., x_t\}$. Observations (object features in the image) are denoted as z_t and its vector as $Z_t = \{z_1, ..., z_t\}$.

The object dynamics are assumed to form a temporal Markov chain in the form of

$$p\left(x_t \mid X_{t-1}\right) = p\left(x_t \mid x_{t-1}\right) \tag{4.2.1}$$

so a new state is only *conditioned* by the immediate previous state.

The observations z_t are assumed to be independent both mutually and from the dynamical process. Hence, the observation process will be reduced to a time-independent function $p\left(z \mid x\right)$, which compares the image to the state.

Given that the observations are independent and our assumption that the dynamics form a Markov chain, we can effectively deduce the state density p_t at time t if we have all image observations with

$$p_t\left(x_t\right) \equiv p\left(x_t \mid Z_t\right).$$

The rule for propagation of state density over time is equivalent to the Bayes rule and is used for inferring the posterior state density:

$$p\left(x_t \mid Z_t\right) = k_t p\left(z_t \mid x_t\right) p\left(x_t \mid Z_{t-1}\right) \tag{4.2.2}$$

where the prior density $p\left(x_t \mid Z_{t-1}\right)$ is a prediction taken from the posterior density of the previous time step.

4.2.2 Factored sampling

Factored sampling is a statistical method to deduce the posterior density $p\left(x \mid z\right)$, which represents all we know about the state $\mathbf{x}$ from the observations. Applying Bayes' rule we can obtain

$$p\left(x \mid z\right) = kp\left(z \mid x\right) p\left(x\right)$$

where k is a normalizing constant. $p(z \mid x)$ compares the observations to the state $\mathbf{x}$ and when it cannot be expressed in closed-form, that is when the factored sampling comes into scene. In the factored sampling algorithm a set of sample points $\left\{s^{(1)}, ..., s^{(N)}\right\}$ is randomly generated from the prior density p(x). Each of the samples are then assigned a weight proportionate to the value of the observed density $p(z \mid x) = s^{(i)}$ where $i \in \{1, ..., N\}$:

$$\pi_t = \frac{p_z\left(s^{(i)}\right)}{\sum_{j=1}^{N} p_z\left(s^{(j)}\right)}. \tag{4.2.3}$$

$p_z(x) = p(z \mid x)$ is the conditional observation density. Notice that the weight is normalized so that $\sum_{i=1}^{N} \pi_i = 1$.

The Condensation algorithm is based on factored sampling, as we will see next.

4.2.3 Notation summary

This section will wrap up all the notation used in the previous sections so that the algorithm in the next section can be understood more clearly.

x - state vector (position of the object, for example).

z - observation, image feature that we can measure (pixel intensity, for example).

$p(x)$ - probability density that represents the state x.

$p(z \mid x)$ - conditional observation density. Compares the image observation with the expected state (weights samples).

$p(x \mid z)$ - state density after measurements z from the image have taken place.

4.2.4 The algorithm

The Condensation algorithm is based on factored sampling applied iteratively to a series of successive images. Each iteration at time t takes as an input a sample set $\{s_t^{(n)}, n = 1, ..., N\}$ (also called particles) and its associated weights $\pi_t^{(n)}$. The input sample set is the posterior density from the previous iteration $p(x_t \mid Z_{t-1})$ and the output, or posterior density, is $p(x_t \mid Z_t)$.

The whole process is shown in figure 4.5. Firstly, we will sample N times from $s_{t-1}^{(n)}$ when each element is chosen with probability $\pi_{t-1}^{(n)}$ (cumulative probability $c_{t-1}^{(n)}$ will be used to do this efficiently as we will see later). Elements with higher weight will tend to be chosen several times (and thus there may be identical samples in the output set) while lower weighted ones might not be chosen at all. In each iteration the samples undergo three processes in this order: drift, diffuse and measure.

Drift and diffusion form the **prediction** phase of the new state of the tracked object. The samples are drifted deterministically according to a dynamical model (their location, for example, is predicted with a dynamical model). Afterwards, they undergo random diffusion so the ones with the same state will no longer be the same (mimics the inherent noise in the tracking process).

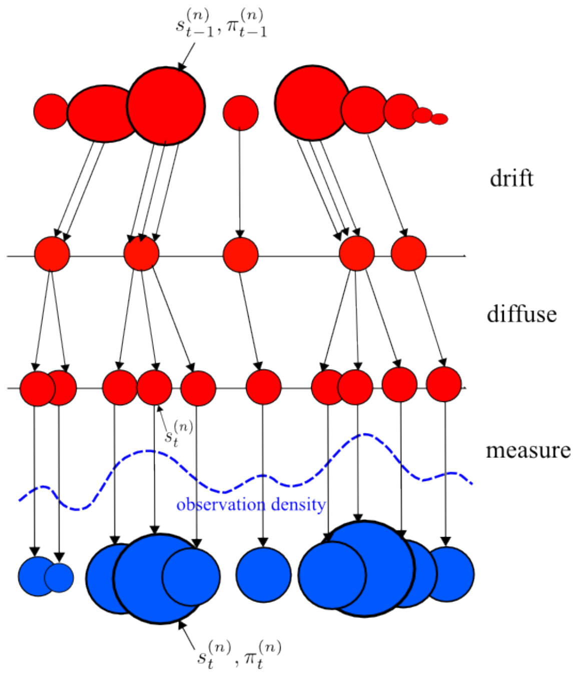


Figure 4.5: An iteration of the CONDENSATION algorithm. Samples with higher weight are more likely to be selected several times and ones with lower weight may not be selected at all. The selected samples undergo drift and diffusion. After measuring with the observation, the samples are assigned a new weight.

Having finished prediction, the next phase is the **measurement** or correction phase we mentioned in the introduction to this chapter. Samples are assigned a weight $\pi_t^{(n)}$ according to observation of the image.

At the end of any iteration it is possible to know the current state (where the object is located, for example).

Outline of the algorithm

The sample set of the previous time step $t-1$ is $\{s_{t-1}^{(n)}, \pi_{t-1}^{(n)}, c_{t-1}^{(n)}\}$. An iteration of the algorithm constructs a new sample set $\{s_t^{(n)}, \pi_t^{(n)}, c_t^{(n)}\}$, $n = 1, ..., N$ at time t.

The n^{th} of N new samples is constructed as follows:

1. **Select** a sample $s_t'^{(n)}$ as follows:
   a) Draw random number $r \in [0, 1]$ from the uniform distribution
   b) Find the smallest j for which $c_{t-1}^{(j)} \geq r$
   c) Set $s_t'^{(n)} = s_{t-1}^{(j)}$
2. **Predict** by sampling from
$$p(x_t \mid x_{t-1} = s_t'^{(n)})$$
to choose each $s_t^{(n)}$ of the new sample set (recall equation 4.2.1). According to *drift* and

diffusion, the new sample may be generated as

$$s_t^{(n)} = As_t'^{(n)} + Bw_t^{(n)} \tag{4.2.4}$$

where A corresponds to the dynamical matrix which determines the amount of drift the sample is expected to experience while B is the standard deviation of the amount of diffusion applied, being $w_t^{(n)}$ a random value from a zero-centred Gaussian distribution, representing process noise.

3. **Measure** (the observed) and calculate weights for the new samples with the measurements z_t. Recalling equation 4.2.3

$$\pi_t^{(n)} = p(z_t \mid x_t = s_t^{(n)})$$

normalized so that the sum of weights is 1. For efficiency, a cumulative probability is associated where

$$\begin{aligned} c_t^{(0)} &= 0 \\ c_t^{(n)} &= c_t^{(n-1)} + \pi_t^{(n)} \end{aligned}$$

Notice steps 2 and 3 follow equation 4.2.2.

At each time step we can estimate the object's position by computing the weighted average of the positions of the samples.

It should be noted that the algorithm does not need special initialization conditions. At the beginning, when there is no object, the samples are spread randomly and resampled constantly while each particle has the same weight (the object to track has not appeared yet), thus the density will settle to a steady state.

4.2.5 Dynamical model and noise

The dynamical model is the deterministic part of the prediction phase and determines the amount of drift the samples experience (how much they have moved with respect to their previous location in the previous time step). A good dynamical model will be able to minimize the prediction error and thus contribute to more robust tracking.

For the sake of simplicity, we started considering a dynamical model of constant velocity based on the position of the sample at time t and $t-1$:

$$\begin{aligned} x(t+1) &= x(t) + [x(t) - x(t-1)] + B_x w_t^{(n)} \\ &= 2x(t) - x(t-1) + B_x w_t^{(n)} \end{aligned}$$

where the state x is the position of the tracked object across the x-axis and likewise for y. From empirical tests the standard deviations of process noise for both dimensions were found and set as $B_x = 29$ for the x dimension and $B_y = 21$ for y. $w_t^{(n)}$ is a random value from a zero-centred Gaussian distribution (please refer to appendix C for how the data was gathered and examined).

Equation 4.2.4 is therefore expanded as:

$$\begin{bmatrix} x(t+1) \\ y(t+1) \\ x(t) \\ y(t) \end{bmatrix} = \begin{bmatrix} 2 & 0 & -1 & 0 \\ 0 & 2 & 0 & -1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} + \begin{bmatrix} 29w_{xt}^{(n)} \\ 21w_{yt}^{(n)} \\ 0 \\ 0 \end{bmatrix}$$

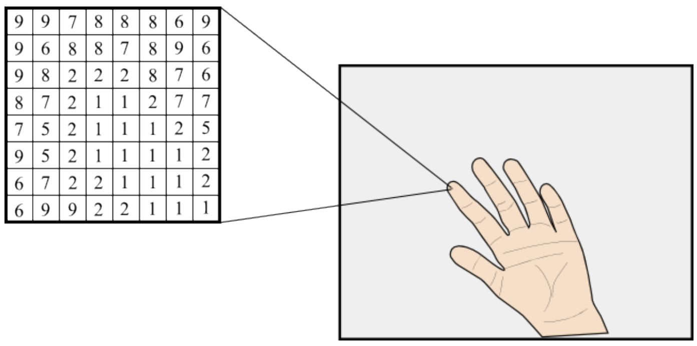
In the tests the results were satisfactory for our purpose, so a second-order model based on acceleration was discarded as it would add to space complexity and slow the application.

4.2.6 Observation model

The observation model is used to compare the image at time t with the state at time t after the prediction phase has taken place. The location of the samples are evaluated probabilistically, i.e. the more probable the object is located at the position of a sample, the higher the weight it is given.

Inspired by Holden and Owens (2003), we employed a window-based technique to measure the weight of the particles, where the window is simply a square block of pixels which represent a skin area. From the hand shape enhancement process we had the skin probability density function of the image (recall section 3.1). The window was then centred at each pixel of the image and the sum of the probabilities (Mahalanobis distances) of each pixel in the neighbourhood was used to weight the pixel at the centre.

A window allows for more precise tracking instead of considering the probability of each pixel individually as the impact of noise is reduced.



9	9	7	8	8	8	6	9
9	6	8	8	7	8	9	6
9	8	2	2	2	8	7	6
8	7	2	1	1	2	7	7
7	5	2	1	1	1	2	5
9	5	2	1	1	1	1	2
6	7	2	2	1	1	1	2
6	9	9	2	2	1	1	1

Figure 4.6: An 8x8 window centred on a pixel. The Mahalanobis distances of all pixels under the window are summed for the weight function. The values on the window are just examples.

The given weight to the pixel at the centre of the window was established as a function of the sum of probabilities (Mahalanobis distances) of all pixels in the window, and thus bounds to this sum had to be set. We used a window of size 8x8, which amounts for a total of 64 pixels. In order to determine the weight limits we calculated a maximum bound for the window based on the threshold $\theta = 2.4$ used to discriminate between skin and non-skin pixels in binary segmentation (see section 5.2). The maximum bound for the window was then $2.4 * 64 = 154$, and a value higher than that would have to be given much less weight as the pixel would be considered non-skin.

The highest weight would be given at a sum of 0, which would mean that all the pixels in the neighbourhood have a colour exactly the same as in our skin model.

The 154 bound, however, often resulted in inaccurate tracking. The reason was that our window technique required a harder discrimination since there could be pixels in the neighbourhood window which were very close to our skin model (summing close to 0) and others very far, but in the end they would yield the same value as in neighbourhoods where the pixels were mostly considered skin but not necessarily close to the model. By trial and error we found that a higher accuracy could be achieved using $\theta = 2$, which accounts for a total weight bound of 128. The weight function was then designed using this bound and is shown in figure 4.7.

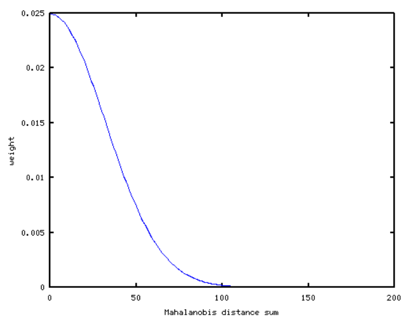


Figure 4.7: The weight function used in our observation model. The lower the sum of Mahalanobis distances of the pixels under the window, the higher the weight. Notice the lower asymptotic limit of the weight is 0 but it will never be exactly 0 since then the sample cannot be selected in a subsequent iteration of the CONDENSATION algorithm (and if that happened to all the samples the tracker would stop working).

Our weight function was a modification of Holden and Owens (2003)'s Gaussian weight function:

$$\pi_t^{(n)} = \frac{1}{\sqrt{2\pi}\phi} \exp\left[\frac{-v^2}{8\phi^2}\right]$$

where v is the sum of Mahalanobis distances to the skin model of each pixel within an MxM window (figure 4.6), $\phi = \dfrac{M^2 * \theta}{10}$, $M = 8$ and $\theta = 2$. Figure 4.7 shows the function plot where it can be seen that the parameters were determined to maximize weight of lower distances and minimize weight of those higher.

4.3 Hand detection

With the previous section the CONDENSATION algorithm has been fully explained. As we pointed out earlier, initially when there is no hand the tracker will be in a steady state with all particles being randomly spread across the image and resampled iteration after iteration. If we calculated the weighted mean position with them the estimation of the location would be around the centre of the image. We know, however, that a hand is not in front of the camera so this estimation would be invalid.

Using the median of densities

We know the probability densities are measured as the Mahalanobis distance from the colour of a pixel to the mean colour of a skin pixel. One method consisted of calculating the median of the probability densities at each pixel within an 8x8 window centred at the estimated location to determine if a hand was in place. The reason behind this approach was that if the median was evaluated as 'skin' then most pixels within the window would be 'skin' and hence we could say a hand has been detected.

This method worked as expected but also had a natural inconvenience. In case the hand was temporarily occluded the tracker would not be able to give an estimated location as it would believe that no hand is in the image area. Being this one of the justifications to use a motion estimator, this method was therefore discarded.

Motion detection

Another method we thought is often used in the field. The idea would be to start the tracker with basic motion detection. For example, a hand moving left-right-left could be interpreted as a sign to start following the hand. However, one could argue that some automatism and convenience is lost.

At the same time we had also thought of another method, which we describe next.

Using the standard deviation

When the particles are tracking an object they concentrate around it and if the object disappears they are again randomly spread around the image. This gave us the idea of using the standard deviation of the particles to detect the presence of a hand (figure 4.8). An assumption was made that if it was below a certain threshold then a hand was detected. Again, by trial and error this threshold was set at 100 for x direction (only one direction was actually sufficient).

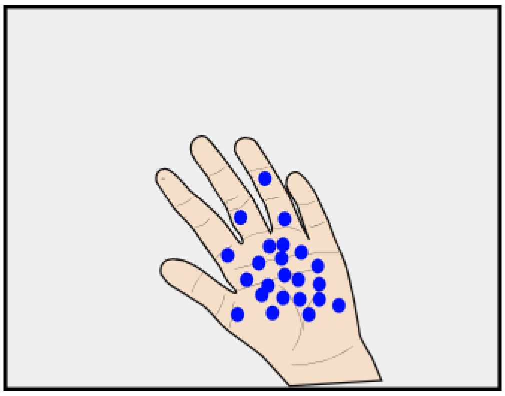


Figure 4.8: When the samples detect a hand they concentrate on it, which means their dispersion is much lower. This dispersion is used to detect the presence of a hand.

This method proved to be very robust even though it still had a weakness. If the noise in the image was not properly removed the tracker could be following wrong objects and thus mistakenly detecting a hand. Another problem was that when the hand disappeared of the screen it would have a bouncing effect, which is explained in the results section.

Nevertheless, we omitted these problems and eventually chose this method as we were not expecting to work with a heavily cluttered background and the bouncing effect was not a big issue taking into account the mouse functionality we implemented (see section 7.2.3).

4.4 Mouse pointer movement

After successfully tracking a hand, the first objective is to be able to move the mouse pointer before going into recognising basic hand gestures corresponding to mouse events or other functionality.

Once a hand is detected, making the mouse pointer move as the hand moves is quite straightforward. A mouse controller does not consider absolute positions but relative positions in the form of increments or decrements in one direction or another. Knowing this all we had to do was to keep variables that stored the location of the hand at each time step and calculate the differences (in x and y directions) with respect to the previous time step. These differences were then used to move the mouse pointer.

4.4.1 Algorithm

We define a time step t the same as with the tracker, since this algorithm is executed at the end of each iteration of the CONDENSATION filter. The hand's position at t is compared to its position at $t-1$. The difference is then added (or substracted) to the current mouse position.

Let (m_x^t, m_y^t) be the mouse position at time step t, (h_x^t, h_y^t) the hand's position and s a boolean variable indicating if the hand is in sight.

Input

From the result of the CONDENSATION algorithm, we have the hand's position estimation (h_x^t, h_y^t) at time t.

Initialization

Set $s = true$.

Iteration

- Detect hand using samples' standard deviation.
- If a hand was detected, then calculate the new mouse position as

  – If $s = true$ then

$$\begin{aligned}\Delta_x &= h_x^t - h_x^{t-1}\\ m_x^t &= m_x^{t-1} + \Delta_x\end{aligned}$$

  and likewise for y direction. Move mouse pointer to new position.

  – Else set $s = true$.

- Else set $s = false$.

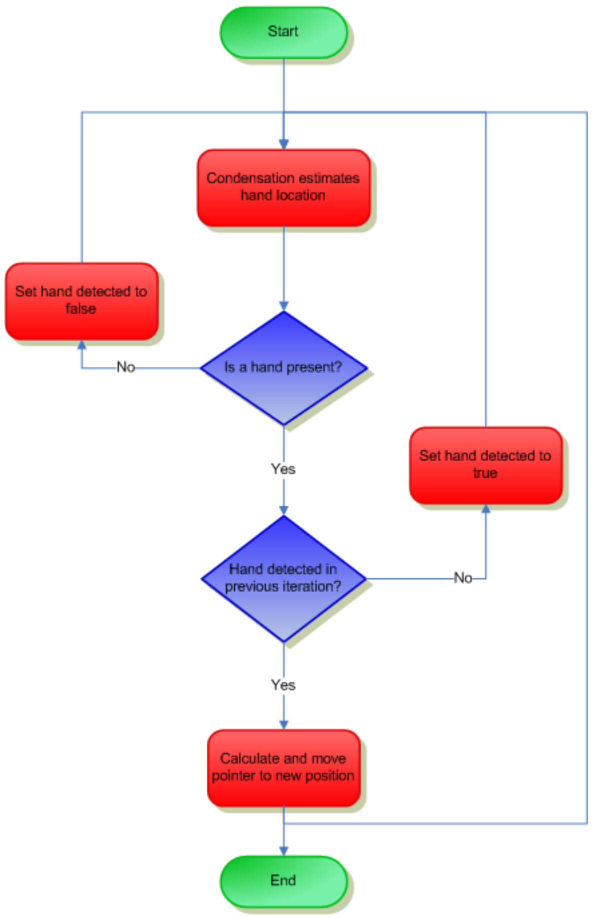

Start
Condensation estimates hand location
Set hand detected to false
No
Is a hand present?
Yes
Set hand detected to true
Hand detected in previous iteration?
No
Yes
Calculate and move pointer to new position
End

Figure 4.9: The mouse pointer algorithm combined with an iteration of the CONDENSATION algorithm. The tracker estimates the hand's position and the mouse pointer's new position is calculated accordingly.

Notice that if a hand is not detected at $t-1$ and is detected at t, the current pointer's position is used to calculate its new position at $t+1$, but remains logically static at t. Figure 4.9 depicts the algorithm.

It should be added that since the images are not captured at the same speed as a conventional optical mouse device captures images, the resulting motion of the pointer on screen is not as smooth as with a mouse. In any case, using artificial acceleration and pixel-by-pixel pointer displacement this issue could be solved.

4.4.2 Pointer stabilization

A problem we encountered was pointer vibration due to noise in the tracking process. Since it was not completely precise, as such would require maximum pixel precision, this would translate into small movements of the pointer in any direction.

To minimize such problem we added a condition to the algorithm to only update the pointer position when it had moved more than a certain amount of distance. This distance was found to provide good results at 4 pixels in any direction.

Chapter 5

Gesture recognition

Having achieved the first part of the project, which enabled us to control the mouse pointer, the next step was to recognize gestures so that we are able to perform typical mouse actions such as right-click.

Gesture recognition has several applications ranging from sign language alphabet recognition to human-computer interaction (HCI) systems. Face-to-face, humans communicate with each other mainly through their visual and auditive senses. In the case of deaf people, they have to rely entirely on their vision, so apart from a heightened interaction experience we are also improving on accessibility to the machines.

Traditionally humans have interacted with computers through a keyboard and a mouse device which, while very useful and functional, one could argue they do not feel as natural as using our bare hands. As of writing this report, touch systems have experienced a boom, especially in the mobile phones sector. The videogames console Wii has exploited this more natural experience with its famous motion-sensing controller. The logical next step is to be able to interact with the machine without any device attached, which is being researched by Microsoft in its Project Natal (figure 5.1) as of writing this report. Specifically speaking of hand gesture recognition, aside of HCI systems, another application could be a deaf sign language automatic translator and combined with audio recognition systems we could translate audio into images or viceversa, which would make TV programs more accessible, for example. We can see that there are as many applications as we can think of that involve interacting with gestures.

The problem we are dealing with is thus making the computer capable of understanding us. Traditionally this has been done by processing data collected from data gloves which can provide accurate hand information. However, it has the big downside of being expensive as well as attaching a device to our hand, which is what we are trying to avoid. Being a hand naturally 3D, 3D scanners have also been used but again we encounter the same problem as before. A cheaper and more convenient alternative is the use of multiple cameras to retrieve depth information from 2D images. However, not all applications require 3D knowledge, which adds to overall complexity, and 2D shape analysis could be enough, which is our case.

In literature gesture recognition is a fairly recent and not many research papers have covered the

Figure 5.1: Microsoft's Project Natal, a new completely hands-free human-computer interface for their gaming console Xbox 360.

subject, probably because it has traditionally been resource expensive, but with the computers of today this problem is much smaller. In general, the methods published can be classified into learning-based and model-based methods. Learning methods rely on training a classifier using the features detected from great amounts of samples. Model-based methods, on the other hand, compare sets of features of hand shapes with models in order to classify them.

Lockton and Fitzgibbon (2002) proved that real-time performance could be achieved with a learning method. They were able to recognize 46 symbols of the American sign language fast and reliably. Ong and Bowden (2004)'s hand shape detector also made use of a machine learning algorithm to train classifiers from a database of hand shapes classified by k-means clustering of distance metrics. Even though it obtained good results, it could not perform in real-time. Dinh et al. (2006) used AdaBoost to achieve computational efficiency and obtained fast and robust detection.

Even though learning methods proved to be successful in gesture recognition, they have the downside of requiring great amounts of training data to be accurate, thus other researchers have investigated model-based methods, which are much less costly in this aspect. Starner et al. (1998) used a feature vector of sixteen geometric elements for recognizing the American sign language, such as hand location, previous location, area or angle of axis of least inertia. O'Hagan and Zelinsky (2000) used a set of geometric properties such as areas of high curvature, which could indicate fingertips location or hand-wrist separation. Templates of these areas were then made and used for gesture recognition. Bretzner et al. (2002) presented a hand tracking and posture recognition system which used hierarchies of multi-scale colour image features to detect a hand and its fingers, together with information about their relative orientation, position and scale. Based on these features they could define models for different hand states and achieved real-time recognition. Our first reference (Holden and Owens, 2003) extracted the topological formation of the fingers region from its polar representation and used a technique from speech recognition (cepstral coefficients) to interpret four different shapes, which they wanted to extend to the Australian sign language.

To increase gesture classification accuracy many methods rely on Markov models (Starner et al., 1998), which means that frames over time are interpreted as being *dependent* from each other. The logic behind it is, for example, that we know the word about to be recognized is an article and the first interpreted sign is 't', thus 'he' becomes the logical continuation (effectively compounding the article 'the'). Usually different temporal gestures are allowed for each frame and their combination represents a certain gesture. However, this implies that the number of recognized gestures is significantly restricted, compared to a system which has a higher single-frame recognition accuracy.

With this vast range of methods, the choice would entirely depend upon the goals we wanted to achieve, which is what we will discuss in the next section.

5.1 Goals

As we had already stated at the beginning of this report, our main idea was to create a human-computer interface using solely a webcam. Specifically, we wanted to be able to to execute common mouse actions with one hand.

The first functionality we thought about was 'left click', as we already had the pointer movement functionality. We soon realized that a *click* is composed of two actions, which are 'button up' and 'button down', clearly noticeable in the case of a user dragging a window to displace it. This suggests that we needed two gestures to implement them.

While testing, we also realized that changing gestures affects the position of the pointer, even though one might think his hand is stationary. The reason is that the tracker's estimated weighted average position is generally close to the centroid of the hand, which obviously changes if the number of fingers or their positions are different. This could be a problem in several scenarios, such as a user trying to click on an icon. In case he performs the click gesture and the pointer moves away the target will be missed. This suggests that for the system to be accurate we needed to recognize gestures in a stabilized position. Apart from this, it improves usability from a user's point of view in the sense that a user will usually stop before changing his gesture. It was then decided that gesture recognition would only be triggered when the mouse pointer was completely stopped, which means that we needed two more gestures for activating and deactivating pointer motion.

As the gesture for different functionality could be shared, three gestures instead of four were then set as the main goal. For a more complete experience, right click, double click and mouse scrolling should also be implemented but we decided to start modestly with only basic functionality since more gestures could be added afterwards.

Summarizing, the gestures to be recognized implement the functionality of:

- Mouse pointer motion trigger on/off
- Left-button click up/down

Figure 5.2 shows the gesture set chosen for the task. Notice the same gesture was chosen for stopping pointer movement or left button click up since they seemed more natural actions.

There are also non-functional requirements needed for enhancing interactivity which can be listed as:

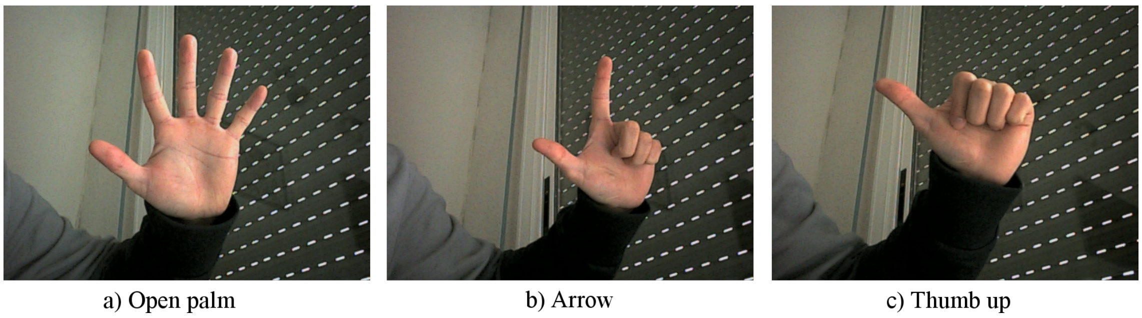


Figure 5.2: a) Open palm: used for the functionality of mouse pointer motion trigger off and left button click up. b) Arrow: used to move the mouse pointer. c) Thumb up: used as left button click down.

- Scale-invariance. While the user should remain within a reasonable distance with respect to the camera, changes in hand size should not affect gesture recognition.
- Rotation-invariance. It would be quite difficult to move a hand naturally without rotating it, thus recognition should not be affected by changes in this aspect either.

Once our initial goals are defined we can proceed to select the most appropriate method for the task.

5.2 Binary segmentation

Before we are able to recognize hand gestures we need to isolate the hand from the background. For this purpose, skin colour segmentation was chosen, which consisted of obtaining a binary representation of the skin and non-skin pixels (figure 5.3).

However, there are other techniques based on learning a background model to effectively separate it afterwards. Background substraction consists of substracting the current image with the learned model (usually another frame representing the background) of the background and compute the differences. The differences are supposedly new objects in the scene. The simplest background substraction method is frame differencing, which as its name indicates, consist of substracting one frame from another. In simple scenes this method works fairly well but it has several problems. It does not distinguish between a hand or any other object and it becomes difficult to know which frames to substract, plus it would only detect regions of motion and a static hand could pose a problem. Another more advanced method is the *Codebook method* (Kim et al., 2005), which can be used in scenes where there are complicated moving objects such as trees waving. The general idea of this method is that it keeps 'boxes' with all the common pixel values seen in an image and these are constantly updated through time. Each box has a low threshold and a high threshold to determine if a new colour of a pixel is still considered part of the background. These thresholds expand when new values fall between the learning thresholds and when a new colour of a pixel falls out of these limits, then it is considered part of a new object in the scene. While this method works fairly well, it is more costly computationally and most importantly, it does not respond well to changes in light, plus again we have the problem that we are only interested in a moving hand, not other moving objects.

The main problem we encountered with skin colour segmentation was the changing lighting conditions but with respect to the other problems, it seemed a better choice than the other background substraction techniques mentioned. In fact, a combination of different methods would probably yield better results but could involve other problems as well, such as a higher impact on efficiency, and would have to be carefully analyzed and tested.

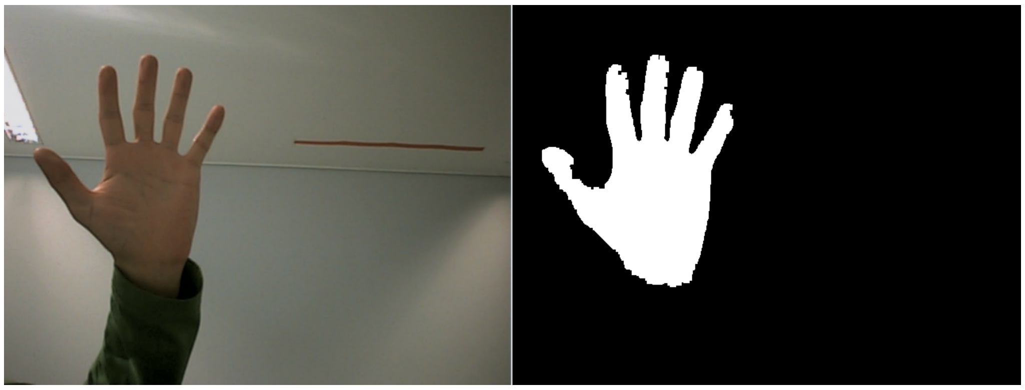


Figure 5.3: An image of a hand and its *binary* segmentation. The hand is effectively separated from the background.

In chapter 3 we obtained the skin probability density image of the input image. From this image segmenting the image into skin and non-skin regions was a matter of finding the correct threshold. A pixel j was classified as *skin* if its probability was $D_j < \theta$, where θ was adjusted as a trade-off between true positives and false positives (see figure 7.7).

In the chosen colour space (HSV), pixels were considered skin when their Mahalanobis distance to the mean was less than $\theta = 2.4$, which is not perfect but choosing a universally good threshold is simply impossible because of varying illumination conditions in different environments or non-uniform illumination on the hand (caused by shadows, for example).

Since there are usually objects in the background that have similar colour to skin the segmentation process usually yields noisy images. Further processing was then required to minimize noise and for that matter morphological operations and connected components detection were used. Finally, we also tried to improve segmentation under varying lighting conditions using adaptive segmentation. The images obtained from each step can be found in the results chapter.

5.2.1 Foreground cleanup

We know from the previous section that we have a reasonably good segmentation of the image but false positives produce noise in the segmented image, which is what we are looking forward to remove. We will first apply morphological operations to remove small areas of noise and bigger areas will be cleaned by detecting connected components.

Morphological operations

By observation we can see that noise is small in area when compared against the hand. The noise was removed by firstly applying the morphological operation *open* followed by *close*. These two operations are based on the basic morphological operators *erode* and *dilate*. *Opening* consists of an erosion followed by a dilation with the same structuring element, which will be defined afterwards. *Closing* consists of a dilation followed by an erosion.

Morphological operations can be applied to binary, gray-scale intensity or colour images but we will focus on binary images, as our input is the binary segmentation of the hand. In general terms, *opening* shrinks the areas of small noise while *closing* rebuilds the remaining areas which were affected by the opening.

Morphology operators test how an image fits or misses a predefined shape called the *structuring element*, which can be understood as a convolution kernel in linear filtering terms. Let A be a binary image in an Euclidean space, and B the set of coordinates of a structuring element. *Erosion* is defined as

$$A \ominus B = \{b \mid B_b \subseteq A\}$$

where b are points in the Euclidean space and B_b is the structuring element translated by b. Consider a structuring element with a centre. Let foreground pixels have a value of 1 and background pixels 0. Assuming that the origin of B is at its centre, for each pixel the origin of the structuring element is superimposed. If every pixel under the structuring element match the values of the structuring element, then the value of the pixel at b is set to 1. Otherwise, it is zeroed. Figure 5.4 exemplifies the operation.

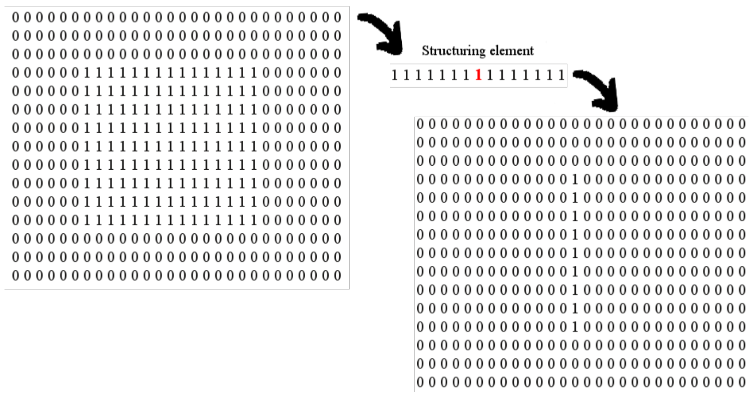


Figure 5.4: Example of a binary image eroded by a 1x15 structuring element. The structuring element only fits the foreground pixels of the image at its centre, in which case the pixel underneath the centre column is not zeroed. When applied over all the image, only the middle column pixels will be left.

Dilation is the opposite operation and can be similarly defined:

$$A \ominus B = \{b \mid B_b \cap A \neq 0\}$$

At each position b, if the pixel at b under which the structuring element is centred is 1, then all the pixels under the structuring element acquire the value of 1.

In the project a common 3x3 square structuring element was used. A bigger structuring element produced better results in terms of noise removal but slowed the application so the original was kept.

Connected components

After applying morphological operations bigger noise areas may still be left. By detecting **connected components** in the segmented image we can then selectively delete them if their area is inferior to a certain arbitrary value - i.e. not the hand. To detect these connected components contour detection was used.

The algorithm used for contour detection was included in OpenCV and derived from Suzuki and Abe (1985). Since understanding the algorithm is not important for our process we will only briefly describe it.

We will first define what is understood as a contour in OpenCV (Bradski and Kaehler, 2008). A contour is a list of points that represent a curve in an image. Consider the segmented test image in figure (figure 5.5) where the background is gray and segmented regions are in white. OpenCV distinguishes two types of contours: *contours* (labeled 'c') and *holes* (labeled 'h'). When contours are dashed red lines, they represent exterior boundaries of the white regions, *contours*, whereas if they are inside a contour that has been labeled as an exterior boundary already - gray regions within the white regions -, they are considered holes and their boundaries are marked with dotted blue lines.

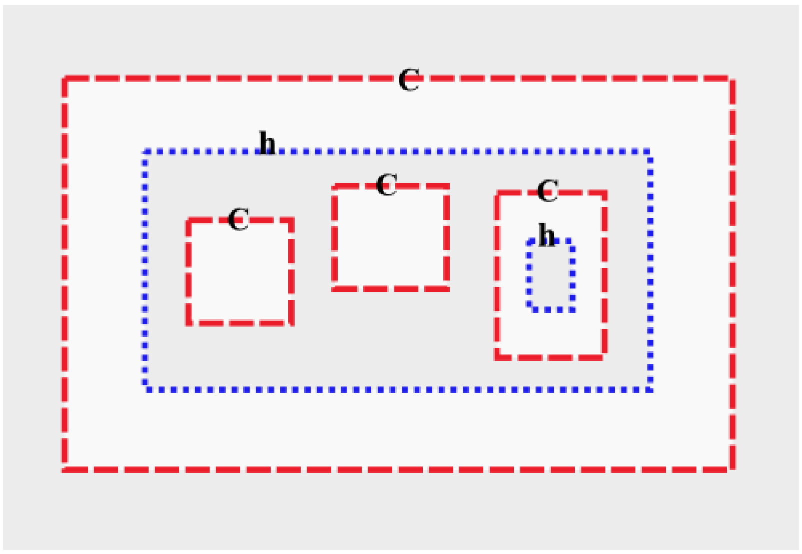


Figure 5.5: A segmented test image where the segmented regions are white and the background is gray. Contours are marked with dashed red lines and holes with dotted blue lines.

For our project we are only interested in the **very exterior contours** and their areas. Suzuki and Abe (1985)'s algorithm scans a binary input image for contours. In few words, the algorithm starts by scanning row by row from left to right. An exterior contour starts as a 0-pixel (background pixel) followed by a 1-pixel (foreground pixel). Once an exterior contour starting pixel is found, it is labeled with an identifier and the contour is followed completely by scanning the neighbourhood of the pixel for 1-pixels and so on successively until it ends back at the starting pixel. The scanning is them resumed at the aforementioned starting pixel until the lower right corner of the image is reached.

Once all the connected components were found, an arbitrary area value was set such that if the area size was smaller than **5000 pixels**, it would be discarded. This way the bigger areas of noise which were not removed through morphological operations could be eliminated.

When only the final contour is left it is filled since there might be holes caused by erosion which were not fully closed.

5.2.2 Adaptive dynamic segmentation

Even though the previous method yielded satisfactory results, varying illumination still posed a challenge. To avoid changing skin models manually we were interested in a model that could adapt itself in changing situations.

A method suggested by Khan et al. (2008) consisting of detecting the skin colour in human faces to retrieve the skin model for a certain frame was first considered, but was soon discarded as we found it would not solve our problem under non-uniform lighting conditions. Another big concern was performance, since face detection was time consuming in itself. Other methods suggested in literature were mostly probabilistic, such as Expectation Maximization (Shamsi et al., 2008) or Gaussian distribution adaptation (Yang et al., 1997).

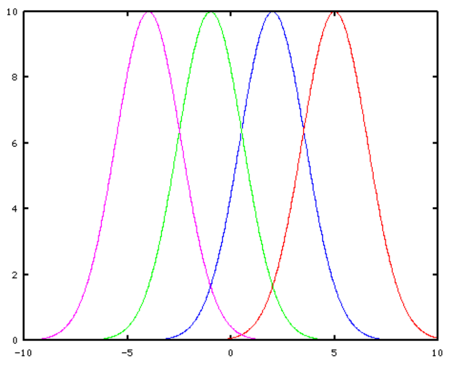


Figure 5.6: Example Gaussian distributions centred at different means. Gaussian mean shifting was used as an adaptive segmentation method.

We chose a form of Gaussian distribution adaptation much simpler than Yang et al. (1997)'s that would only add little processing overhead to our system. Observing figure 3.2 we saw that the distribution shapes were similar but centred at slightly different locations. This suggested us that we could dynamically move these distributions to improve segmentation by **mean shifting**. The idea was then to adapt the skin colour distribution mentioned in the hand shape enhancement chapter. Colour information of the pixels under the samples of the CONDENSATION tracker was used in order to determine the current weighted average skin colour.

The method is detailed below:

1. Calculate the weighted average skin colour m of the pixels under the samples of the CONDENSATION tracker.
2. Compute the Mahalanobis distance $D = \sqrt{(m-\mu)^T\Sigma^{-1}(m-\mu)}$, where μ and Σ are the combined mean and covariance matrix found in section 3.1.2.
3. If $D < 1$ then we use m as the new mean for segmenting the next frame.

And segmentation was done as before, considering skin those pixels which were at a Mahalanobis distance $D < 2.4$ to m.

Notice that in every different frame the new mean m is always compared to μ and Σ, which are absolute references and do not change. Otherwise the mean could shift completely out of the skin distribution bounds. $D < 1$ was calculated by computing the maximum Mahalanobis distance of the means of all lighting conditions considered with respect to the 'all conditions combined' mean.

The results obtained (see 7.3.1) suggest this adaptive method provides more robust segmentation to varying illumination, especially in cases when the amount of light that each part of the hand receives is different and generates shadows.

5.3 Method choice

Once the hand is successfully separated from the background we can proceed to analyse it in order to recognize gestures. As seen previously, many methods exist for gesture recognition. However, taking into account the functionality we need, which is not much, most of the methods mentioned before are too complex, plus each of them may have some kind of inconvenience for our case.

For example, our first reference, Holden and Owens (2003), did not satisfy our requirement of rotation-invariance as the polar image they obtained required that the hand starts facing away from the camera with fingers pointing downward. In our case, the camera does not face the hand from above but towards the user and we would like to restrict the user as minimum as possible, since enhancing interactivity is the main purpose.

Learning methods, while proven very effective, were discarded as they required much training and implementation time and were conceived for classifying large sets of gestures. For our small gesture set, a model-based approach using geometric features seemed more convenient.

O'Hagan and Zelinsky (2000)'s approach could detect hand pose and fingertip positions, but it was based on 3D information obtained from two cameras. Bretzner et al. (2002)'s method based on localizing the different parts of a hand by detecting blobs and ridges was similar to what we were looking for (figure 5.7).

Detecting fingertip positions was set as our objective as the possible combinations of the five fingers, even without changing their vertical axis orientation, provided more than enough gestures. Bretzner et al. (2002)'s method, however, seemed slightly complicated and we thought the same

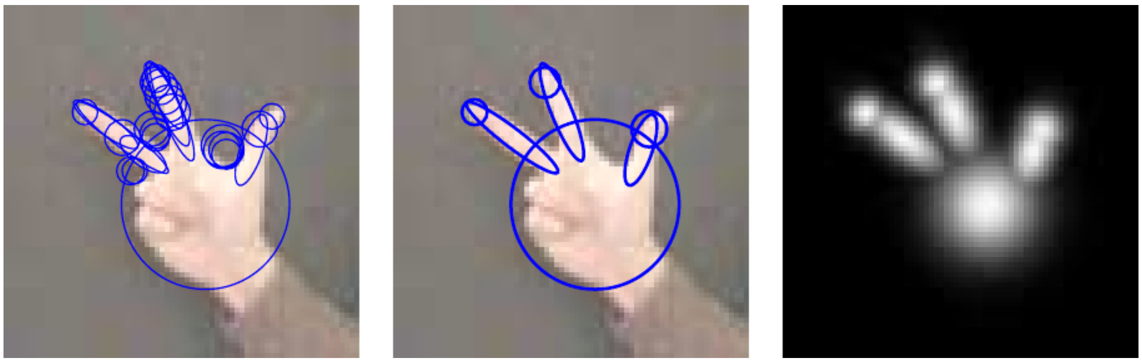


Figure 5.7: Image by Bretzner et al. (2002). They used multi-scale analysis to compute blob and ridge features to characterize the hand's geometry.

could be done by considering the geometric shape of the hand. Malik (2003) detected fingertips by detecting *peaks* and *valleys* on a segmented hand, but their approach was not robust enough as it often missed valleys. Our method aimed at detecting the same features, but we achieved more robustness.

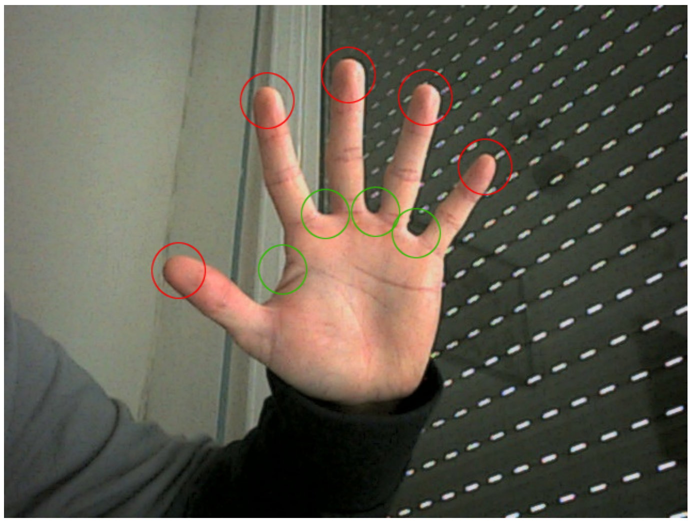


Figure 5.8: The red circles indicate the peaks (fingertips) and the green circles the valleys. These are the hand geometrical features we are looking for.

5.4 Our method

The method we conceived consists of detecting the valleys (convexity defects) between the peaks (fingertips) by using simple geometrical concepts. Polygon areas, and hence perimeters, are robust measurements independent of rotation and proportional in scale. These properties were used to construct our recognition system.

5.4.1 Contour detection

The starting point of the algorithm was to detect the contour of the hand (figure 5.9), which was already detected during the segmentation phase (section 5.2.1).

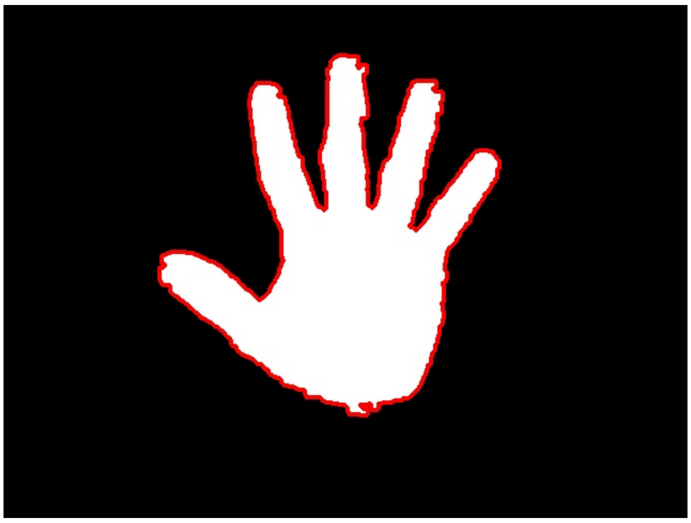


Figure 5.9: The contour of the segmented hand is marked with a red line.

This contour describes the shape of the hand but with too much detail. We would like to be able to represent the shape with fewer points so as to simplify the problem, which is why we performed a polygonal approximation in the next section.

5.4.2 Polygonal approximation

In order to simplify the representation of the shape of the hand we intended to reduce the number of points in the contour to represent this shape. This simplification was necessary so that the next steps of the algorithm are able to find the peaks and valleys we are looking for. But how exactly are we simplifying the shape? To understand the process we will explain the Douglas-Peucker algorithm (Douglas and Peucker, 1973), which is the one OpenCV offers for the task.

Let $\{p_0, p_1, ..., p_n\}$ be an ordered set of $n + 1$ points in the plane which form a polygonal chain of line segments $\overline{p_0p_1}, ..., \overline{p_{n-1}p_n}$. Cartographers Hershberger and Snoeyink (1992) identify the line simplification problem as finding, given a polygonal chain C with n segments, a new chain C' with fewer segments that approximates C, where approximation has a broad meaning (area difference between C and C', C' with the most critical points...).

The OpenCV's Douglas-Peucker algorithm implementation takes as input a contour C, which is a set of ordered points, and a distance parameter ϵ. It first searches for two maximally separated points. Once found, it searches for the point which is farthest to the imaginary line that connects the two extremal points and this distance is $> \epsilon$. If this point meets this condition, the contour is further simplified recursively by splitting the contour at this point and repeating the process with the slices until the points left do not longer meet the $> \epsilon$ condition. If the point does not meet the condition, the approximation is accepted. The points of the resulting approximation are therefore a subset of the original points set (figure 5.10).

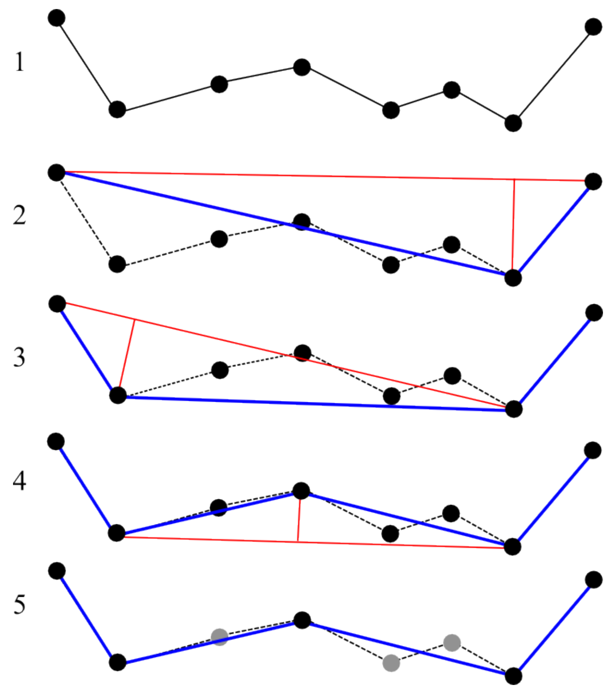


Figure 5.10: Polygonal approximation of a set of points using the Douglas-Peucker algorithm. The points in gray were discarded for being within $< \epsilon$. This algorithm is used to approximate the contour of a hand.

After repeated trial and error tests, the distance parameter was set as $\epsilon = 20$ (pixels). Figure 5.11 shows the polygonal approximation of the previous detected contour using this distance parameter.

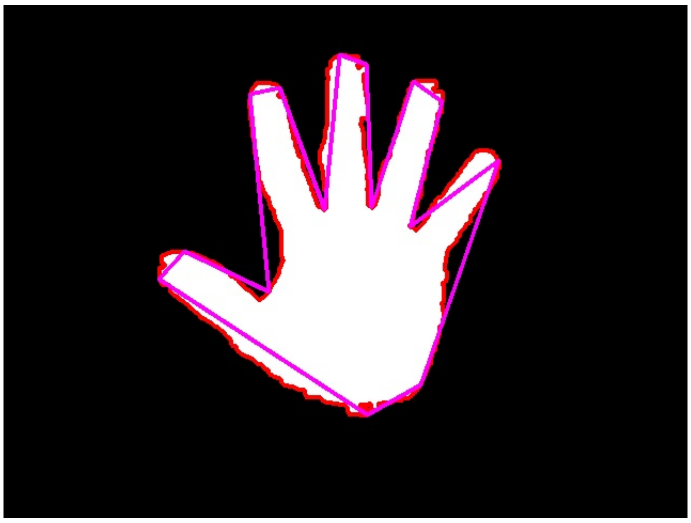


Figure 5.11: Polygonal approximation (purple lines) of the previous detected contour.

Algorithm

Input

C = ordered set of $n + 1$ points
$\epsilon = 20$

Initialization

$< p_i, p_j > = findExtremalPoints(C)$, where a and b are indices of points in C.

Procedure

```
DOUGLAS-PEUCKER(C, i, j, $\epsilon$)

  1. Find farthest point $p_f$ from imaginary line $\overline{p_i p_j}$. Let $d$ be its distance.
  2. If $d > \epsilon$ then
       (a) listA = DOUGLAS-PEUCKER(C, i, f, $\epsilon$)
       (b) listB = DOUGLAS-PEUCKER(C, f, j, $\epsilon$)
       (c) return {listA $\cup$ listB}
     else
       (a) return {i, j}
```

5.4.3 Convex hull

Once the polygonal approximation has been made we proceed by calculating its convex hull.

A set C is said to be convex if for all p and q within the set the line segment $\overline{pq}$ that joins them is in C (figure 5.12).

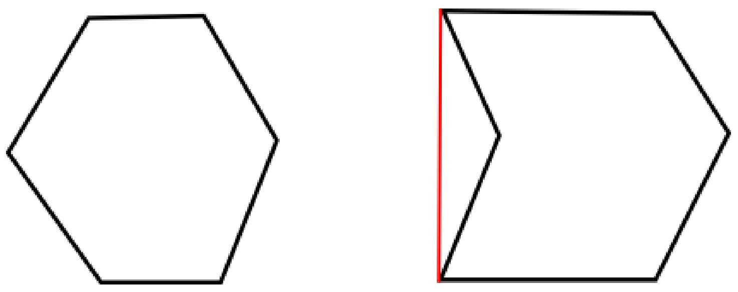


Figure 5.12: The *left* polygon is convex while the *right* polygon is not. The red line proves the stated convexity condition is not met.

The convex hull of a set X is then the smallest convex set C that contains X, or more specifically, the boundary of C (figure 5.13). There are many algorithms for calculating the convex hull and OpenCV implements Sklansky (1982)'s algorithm, which we will not explain as it is unnecessary for understanding our project.

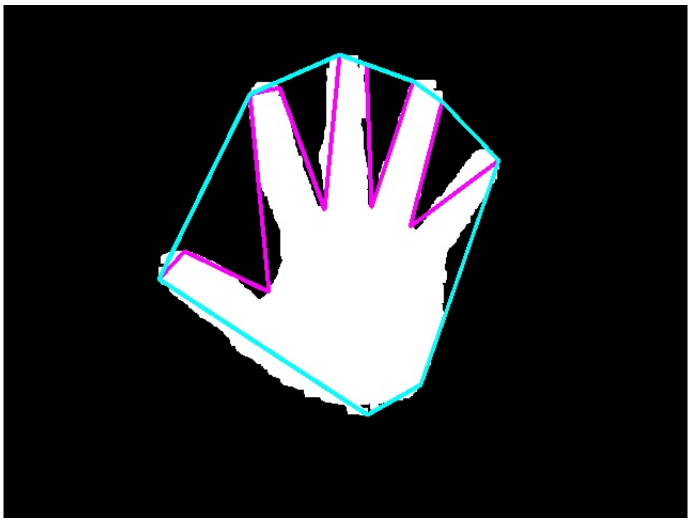


Figure 5.13: Convex hull of the polygonal approximation (marked in teal).

5.4.4 Convexity defects

The concept of convexity defect is illustrated in figure 5.14, which can be understood as concativity areas with respect to the convex hull. As we can see in the picture, there are many convexity defects in a hand, not only the areas between fingers. We can now see the advantage of having calculated the polygonal approximation as the small convexity areas disappear.

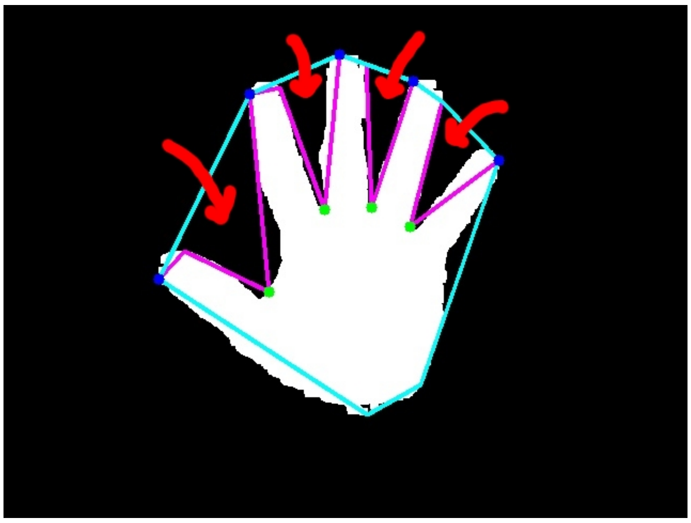


Figure 5.14: Convexity defects are the valleys discussed at the beginning of the chapter. The deepest points in the convexity defects are marked with green dots and the peaks with blue dots.

Concretely, we were interested in the deepest points in the convexities, which were the valleys we were looking for. To obtain them the convex hull's and the polygonal approximation's points are read simultaneously. The points in the polygonal approximation which are between two points common to the convex hull form a convexity defect. Furthermore, these two common points are the starting and ending points of a convexity defect, which can be used to detect fingertips.

5.4.5 Gesture recognition

After getting to this point, we are ready to use these detected features for gesture recognition. The problem with the deepest points in convexity defects, however, is that often there are points which are not of of our interest, as we can see in figure 5.15.

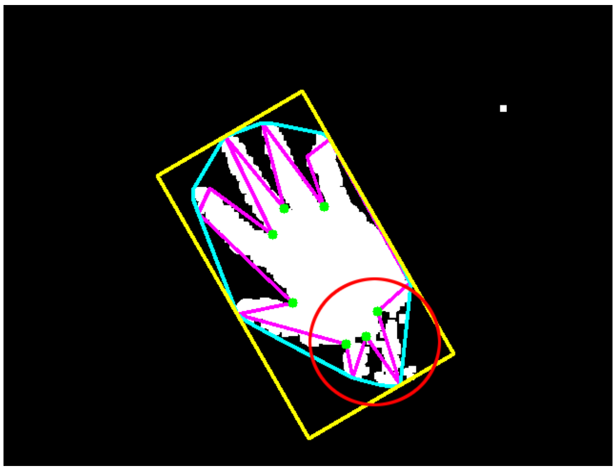


Figure 5.15: The green points representing the deepest points of the convexity defects within the red circle are points which are not valid for recognition. They are result of poor segmentation.

At first, the minimum bounding box was considered to see if it could be used to discard the unwanted points. The idea behind it was to fix a point of the box, join it with the estimated hand location (approximately at the centre of the hand) and compute the angle with the points. The problem was that it was not so easy to detect the orientation of the hand, and thus fix a point at the bounding box.

Another way was necessary and the distance of the points to the convex hull was considered. By observing the results we had so far, we noticed that the points at the valleys between fingers were at a further distance with respect to the unwanted points. The maximum distance d to the convex hull was calculated and then the points which were at $< 0.6d$ were considered discardable (they will be referred as *invalid convexity defects*), which eventually worked fairly well. Important valleys were discarded sometimes, but they appeared to be more related to poor segmentation.

However, this approach still had a problem, which is shown in the figure 5.16. Apart from discarding valid valleys, poor segmentation can yield false positives as well.

We did not find an easy way to tackle it, so in the end it was decided that this gesture would not be recognised. Instead, we decided that the only gesture with one valid convexity defect which would be accepted would be the thumb-up gesture, rotation-invariant.

As can be observed (figure 5.17), this gesture's minimal bounding box has a particular width to height ratio, so gestures which don't meet a certain ratio threshold can be discarded. In particular, 1.6 was selected, being the result of dividing width/height or the inverse, whichever is > 1.

Another criteria we used to validate gestures was to establish a minimum and maximum number of sides that the gesture's polygonal approximation can have. With the polygonal approximation

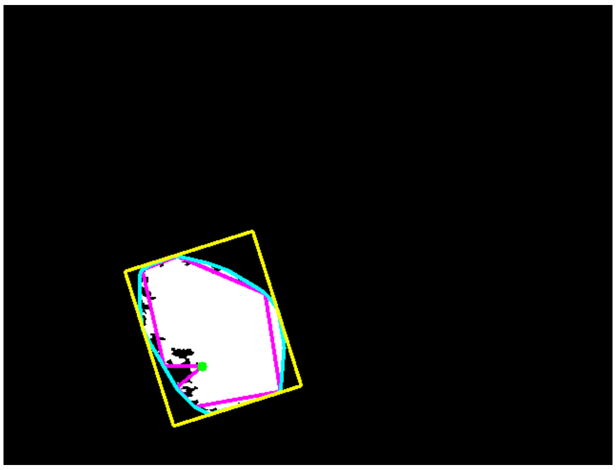


Figure 5.16: (image needs correction)The closed fist's segmentation will often result in a false positive. No simple way was found to discard it so this gesture was not considered for the gesture set.

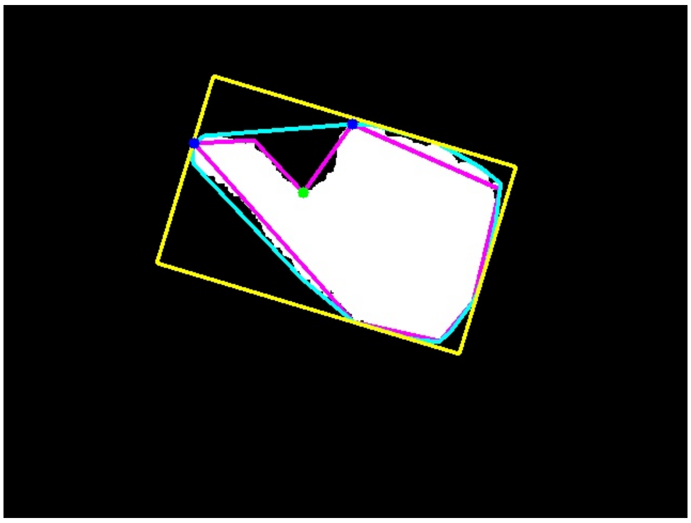


Figure 5.17: The thumb gesture's minimal bounding box has a particular width to height ratio, which we use to enhance detection.

we had chosen, the start and finish points of the convexity defect, the fingertips, could be either spikes or flat (figure 5.18).

By observation, we defined a minimum number of sides of $min\_sides = 6$ for the polygonal approximation, which we found happened with a closed fist. Then, for every convexity defect we can count 4 sides. Invalid defects change what would have been one side to two or more. Removing already counted sides, we can get a number of maximum sides for a polygonal approximation to be considered valid:

$$max\_sides = min\_sides + (4 * valid\_defects) - (valid\_defects - 1) - invalid\_defects$$

Even though not perfect, with these two simple methods we are able to discard many invalid

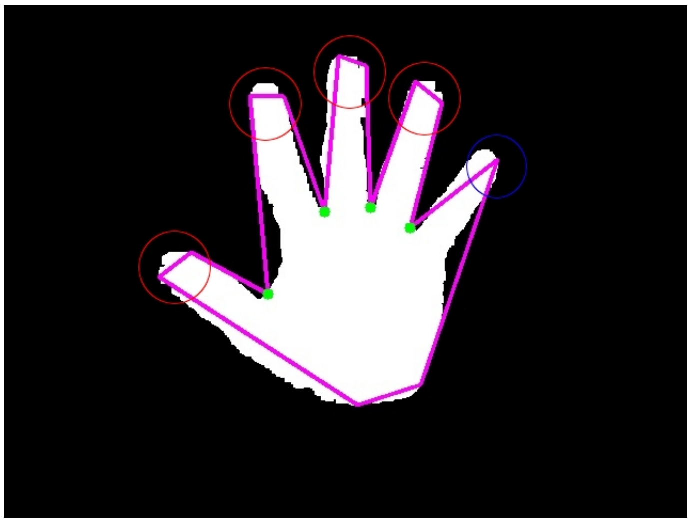


Figure 5.18: The red circles indicate the fingertips approximated as flat and the blue circle as a spike.

gestures due to poor segmentation and thus have a more robust recognition.

It is also important to note that partial hands cannot be interpreted, thus the system cannot be triggered when the hand is not completely visible. At first the minimal bounding box had been used to check that the whole hand was inside the viewport but this proved unreliable when the hand orientation was horizontal/vertical, as the box would be horizontal/vertical as well when the hand was close to the edges. Instead, the minimal bounding circle was used. Only if it was completely inside would the gesture be considered for recognition.

After this processing the system proceeds to recognize the gesture. Given that our defined gesture set is small, simply counting the number of convexity defects was enough. For a broader gesture set fingertip detection and angles between them can be processed, which is described in the next section.

Gesture features summary:

1. Open palm. 4 valid convexity defects.
2. Arrow gesture. 2 valid convexity defects or 1 valid convexity defect + 1 invalid convexity defect.
3. Thumb up. 1 valid convexity defect and the hand has stopped moving (± 4 pixels). Minimal bounding box width to height ratio of 1.6.

Algorithm outline

Input Segmented hand binary image.

Procedure

1. Contour detection of the hand

2. Contour polygonal approximation using Douglas-Peucker algorithm
3. Calculate convex hull of the polygonal approximation
4. Compute convexity defects and its deepest points
5. Check gesture validity and recognize

5.4.6 Further extending gesture recognition

The existing system already has the functionality required for the project, but one could argue it can only recognize quite a limited range of gestures. Even though we did not use it eventually, we found a simple yet effective extension to the original algorithm by using the geometric features we had detected already.

Having obtained the convexity defects, apart from the deepest points we also had the starting and ending points of the convexity, which means we could detect the fingertips (peaks). However, sometimes the fingertips could have been approximated as flat (figure 5.18), thus an ending point of a prior convexity defect and a starting point of the next convexity defect could be detected and would represent the same fingertip. This is trivially solved by considering only the first starting point and only the ending point for the remaining peaks to process.

Once we have all the peaks and valleys, the angle between the different peaks from the centroid could be used to detect other gestures. Given that one of our requirements was rotation invariance, we had to add the restriction that the thumb should be visible and separated from the rest of the fingers so that it could be used as a reference for hand orientation.

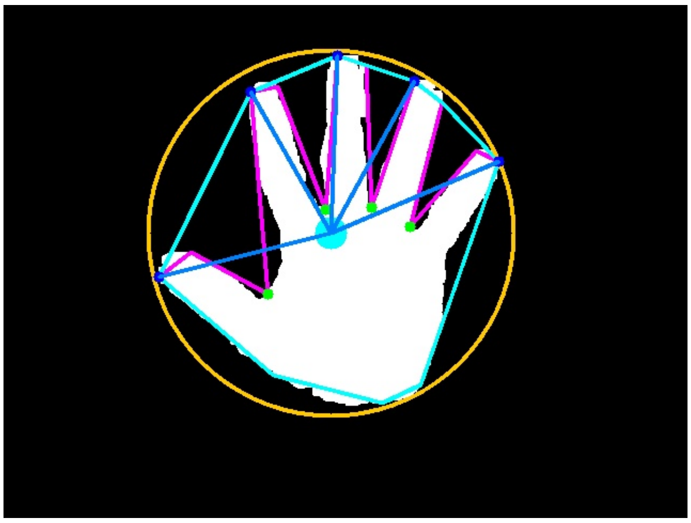


Figure 5.19: The angles between the (approximate) centroid of the hand to the fingertips can be used for recognizing more gestures.

To detect the thumb, angle discrimination is also used. If we consider the fingertips as if they were on a circle, the tips besides the thumb will be at an angle $> \pi$ on one side and another angle θ different from the rest on the other side.

The angle between fingertips considering line segments from the centroid of the hand was calculated through basic trigonometry. To avoid calculating the centroid, the centre of the hand's minimal bounding circle was used, which is a good approximation (figure 5.19).

Therefore, more gestures could be added in case of need using this angle approach. Even though our approach obviously cannot interpret a full sign language such as the American Sign Language due to our restrictions and the nature of an angular approach, many combinations are possible and would probably be enough for most human-computer interfaces.

Chapter 6

Implementation

In this chapter the essential implementation details of project will be shown. The structure of the program will be first presented and the rest of the sections in the chapter will deal with each part separately. In all sections pseudocode/plain text will be used so as to make the chapter more readable, except for specific parts where it is interesting to see how specific external library functions are used.

The programming language chosen was C++ and the libraries used were Qt 4.1 for the GUI, OpenCV 1.1 for common computer vision functions and XTest library for manipulating the mouse pointer. It should be noted that OpenCV itself provides GUI functionality but Qt was first chosen in order to build the Condensation prototype, which will not be shown in this chapter as it has little relevance. These choices were made upon the assumption that a compiled program would be more efficient than its equivalent in Matlab form, even though no comparative tests were done, and due to Matlab lacking an interface editor in its Linux version.

The program consists of two main windows, namely the camera window and the segmentation window. The former show the images being captured by the camera in real time while the second one shows the binary segmentation and the corresponding detected geometrical features (figure 6.1).

The program was structured as follows:

- **main.cpp**. Application main entrance which only creates the camera window.
- **Camera_window.cpp**. The code for the whole logic of the application is contained in this file.
- **Segmentation_window.cpp**. File which contains the code for creating the segmentation window and is used by Camera_window.cpp. Does not contain other algorithmically relevant code.
- **Condensation.cpp**. Contains the code of the Condensation tracker.
- **MouseController.cpp**. Contains the code for controlling the system's mouse pointer.

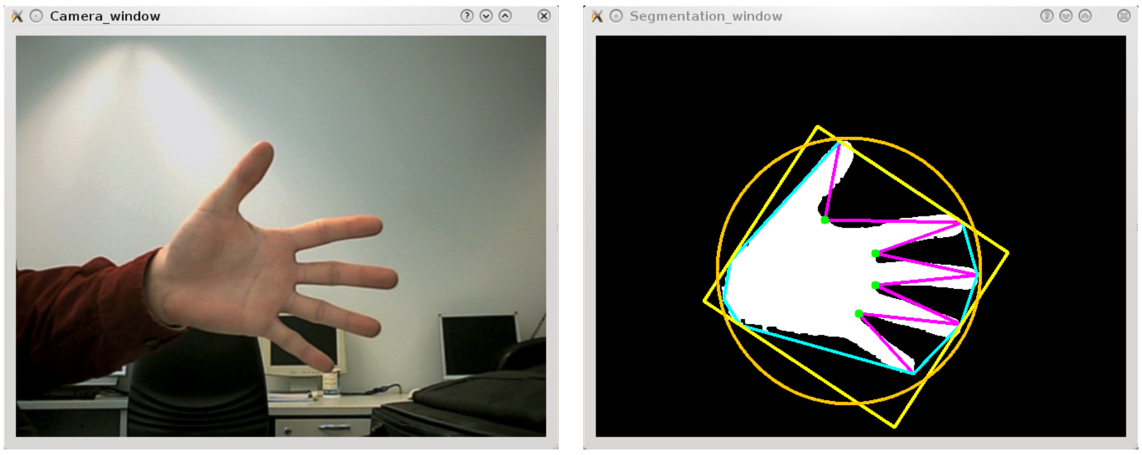


Figure 6.1: Our application interface: the camera window (left) and the segmentation window (right). The camera window is used to visualize the image captured by the camera as well as the samples of the CONDENSATION tracker in case of need. The segmentation window shows the binary segmentation of the hand together with its geometrical features.

The names of the functions used in this chapter may not match the actual functions found in the source code. Some of the function names refer to actual inline portions of code but were summarized with the name of a function in order to increase the readability of the algorithms. If necessary, the pseudocode can be easily matched with the actual source code. Only the most relevant code will be shown and most auxiliary functions will be left.

Note: the OpenCV functions showed in this chapter will only be explained so that the code shown is easily understandable. For more detail the documentation for the OpenCV functions can be found at http://opencv.willowgarage.com/documentation/.

6.1 Main program

The logic of the program will be shown in this section. As mentioned before it is contained in Camera_window.cpp. The file contains the class 'Camera_window', which is created by main.cpp. There are no public functions in the class as it was thought as a main program instead of a reusable class.

Upon creation it initializes internal parameters and then proceeds to capture images. The relevant variables which could be changed to change the behaviour of the program are shown below:

- **IMAGE_WIDTH**, **IMAGE_HEIGHT**. The captured image size.
- **NUM_SAMPLES**. Number of samples for the CONDENSATION tracker.
- **mean**. Reference skin colour mean for hand shape enhancement and binary segmentation.
- **meanSeg**. Variable skin colour mean used in adaptive segmentation.
- **invCov**. Inverse of the covariance matrix used for hand shape enhancement and binary segmentation.

Once the relevant parameters are set, the program creates the segmentation window and then calls setCameraWindow(), which initializes the camera window and the segmentation window.

setCameraWindow()

```
initCameraWindow(); /* initialize camera window background to black */
initSegmentationWindow(); /* initialize segmentation window background to black */

if !cameraIsConnected() then throw error;

tracker = createTracker(); /* create the condensation tracker structure */

configureCamera(); /* configure camera capture parameters */

capture(); /* called every 30 msecs */
```

As can be seen, at the end of initialization the function **capture()** is set to be called every 30 msecs. This function contains the main logic of the application. It grabs the image from the camera and then the corresponding processes are performed, namely hand image segmentation, binary segmentation, tracking and gesture recognition.

capture()

```
initVariables(); /* initialize image structures and other variables */

frame = cvQueryFrame(camera); /* capture image from the camera */
cvFlip(frame, img); /* flip image for mirror effect (OpenCV function)*/
convert img to HSV;

forall i,j in img do
   mahal[i][j] = handShapeEnhancement(img[i][j]);
   imgSeg[i][j] = binarySegment(img[i][j]);
endforall

processBinSeg(); /* see next section for details */

tracker->estimate(x, y, img, mahal); /* estimate hand position */

forall i in tracker->samples() do
   drawSample(i);

   /* calculate weighted mean colour of the samples */
   meanColour += sampleColour(i);
endforall

if handIsVisible() then
```

```
      updateSegmentationModel(meanColour); /* update skin colour model */

      /* if hand was within sight in the previous iteration */
      if handWithinSight then
         if handIsMoving() and totalHands == 1 then
            mouse->update(x,y);
         endif

         if !handIsMoving() and hand_totally_inside then
            if totalHands == 1 then
               gestureRecognition();
         endif
      endif
   else
      /* default state, set at initialization of the class */
      handWithinSight = false;
   endif

   tracker->update(); /* update Condensation tracker state */

   /* show the original image and binary segmentation
      in the corresponding windows */
   camera_window->setImage(img);
   segmentation_window->setImage(imgSeg);
```

The main logic has been listed. In the next sections more detailed information will be given on the specific parts that the code above used.

6.2 Hand shape enhancement and binary segmentation

The objective of hand shape enhancement is to obtain a probability density function which represents the skin pixel density in the image. This is done in the first for-loop in the function capture() of the previous section.

Recalling section 3.1, the probability measure we used was the Mahalanobis distance of the pixels to a previously calculated mean colour, which we save for each pixel in a matrix called mahal. Binary segmentation is done for gesture recognition, but it was more efficient to do it together with the calculation of the probability density function. Since we used adaptive segmentation, we had another variable *meanSeg* which is initialized as a copy of *mean* and is updated with the information from the tracker. Considering the mean colour *meanSeg* and the inverse of the covariance matrix *invCov*, the corresponding code was as follows:

```
   mahal[i][j] = cvMahalanobis(pixelColour, meanSeg, invCov);
```

where **cvMahalanobis** is an OpenCV's function that returns the Mahalanobis distance from *pixelColour* to *meanSeg* using the inverse of the covariance matrix *invCov*. All three parameters are matrices.

Having calculated this Mahalanobis distance computing the binary segmentation is straightforward as it is a matter of comparing the obtained distance to a threshold ($\theta < 2.4$).

```
if mahal[i][j] < 2.4 then
   imgSeg[i][j] = 1;
else
   imgSeg[i][j] = 0;
endif
```

In order to update *meanSeg* with the information of the samples from the tracker a weighted average skin colour was calculated from the colour of the pixels under the samples and their assigned weight. The first thing that has to be checked is that the sample being examined is within the viewport, as the prediction process of the tracker may have drifted it out of sight. Afterwards the mean can be calculated. Depicted as *meanColour += sampleColour(i)* in capture(), it is expanded as follows:

```
...
forall i in tracker->samples() do
   if withinViewport(i) then
      meanColour += i.weight * i.colour;
      totalWeight += i.weight;
   endif
endforall
meanColour = meanSeg/totalWeight;
...
```

Afterwards, if a hand was actually visible, *meanSeg* is updated with *meanColour* if *meanColour* is within 1 Mahalanobis distance from *mean*. This functionality was summarized as updateSegmentationModel(meanColour) in capture():

updateSegmentationModel(meanColour)

```
mah = cvMahalanobis(meanColour, mean, invCov);
if (mah < 1) then
   meanSeg = meanColour;
endif
```

In the next iteration of capture() meanSeg is used for calculating the skin probability density and the binary segmentation. It should be noted that both mean and meanSeg are actually two-dimensional as the mean colour used consists of components Hue and Saturation.

The reader might have noticed that the mean skin colour is calculated irrespectively of a hand being detected or not. In case that a hand has not been detected then that calculation would be useless. However, since the particles were painted to show the tracker in action it was decided that the mean colour would also be calculated together.

6.3 Tracking

Tracking is done using the CONDENSATION functions offered by OpenCV. OpenCV provides the main structures needed and the algorithms. These structures contain information of the samples (state, weight...) as well as the dynamic matrix, which we have to define. Apart of the dynamic matrix, the weight function must also be defined. We will first introduce these functions.

The structure that contains all the information required by the tracker is **CvConDensation**:

```
typedef struct CvConDensation
{
    int MP;     //Dimension of measurement vector
    int DP;     // Dimension of state vector
    float* DynamMatr;       // Matrix of the linear Dynamics system
    float* State;           // Vector of State
    int SamplesNum;         // Number of the Samples
    float** flSamples;      // array of the Sample Vectors
    float** flNewSamples;   // temporary array of the Sample Vectors
    float* flConfidence;    // Confidence for each Sample
    float* flCumulative;    // Cumulative confidence
    float* Temp;            // Temporary vector
    float* RandomSample;    // RandomVector to update sample set
    CvRandState* RandS;     // Array of structures to generate random vectors
} CvConDensation;
```

MP is the number of dimensions of the tracking area. *DP* is the number of dimensions of the vector that represents the state of the tracked object. *DynamMatr* is the dynamical matrix. *SamplesNum* is the number of samples used by the tracker. *flConfidence* is the array of weights assigned to each sample. *RandS* is an array of random values that represents the amount of noise to be added to the array of samples. The rest of the fields are mainly used internally.

In order to allocate the previous structure, **cvCreateConDensation** is used:

```
CvConDensation* cvCreateConDensation(int dynam_params,
   int measure_params, int sample_count)
```

dynam_params is assigned to DP. *measure_params* is assigned to MP. *sample_count* is assigned to SamplesNum. The function returns a pointer to the allocated CvConDensation structure.

Once the tracker structure is allocated, the samples are initialized with the function **cvConDensInitSampleSet**. This function simply sums the noise to the samples' state (recall the stochastic diffusion of the samples at the prediction stage).

```
void cvConDensInitSampleSet(CvConDensation* condens,
   CvMat* lower_bound, CvMat* upper_bound)
```

condens is a pointer to the tracker's structure. *lower_bound* is a vector of the lower boundary (minimum values) for each dimension and *upper_bound* likewise for the upper boundary.

To proceed to a new iteration of CONDENSATION the function **cvConDensUpdateByTime** is used. This function performs the selection and prediction stages.

```
void cvConDensUpdateByTime(CvConDensation* condens);
```

condens is a pointer to the tracker's structure.

Now that we know the functions that OpenCV provides for implementing the CONDENSATION tracker we can proceed to see how they were actually used.

In our project a Condensation class was created in Condensation.cpp with functions that call OpenCV's corresponding functions, as well as defining the dynamic matrix and weight function.

The relevant public functions of our class are:

- **initialize()**. Initializes the structures of the tracker.
- **estimate(x, y, img, mahal)**. Estimates the position of the hand, which is saved to x and y.
- **update()**. Updates the state of the tracker (called before entering the next iteration).
- **isObject(x, y)**. Determines if the object at (x,y) is a hand using the standard deviation of the particles. Returns true if the standard deviation is below 100 pixels.

Relevant variables that can be used to change the behaviour of the tracker:

- **NOISE_STD_DEV_X = 29**. Standard deviation of the amount of noise added to the samples at the prediction phase for X dimension.
- **NOISE_STD_DEV_Y = 21**. Same as above for Y dimension.
- **SKIN_BOUND = 2**. Skin bound used for the window based weight function.

A condensation class is created as follows:

```
    Condensation(num_samples, window_width, window_height, template_size);
```

where *num_samples* is the number of samples, *window_width* and *window_height* the size of the area to be tracked and *template_size* is half the length of a side of the template for the observation model (section 4.2.6). This constructor calls the OpenCV's CONDENSATION structure constructor cvCreateCondensation:

```
    filter = cvCreateConDensation(dynam_params, measure_params, num_samples);
```

After creation of the class, the function **initialize()** is called to initialize the corresponding structures of the tracker. The dynamic matrix is defined in this function, as well as the amount of noise that will be used for the stochastic diffusion of the samples. Therefore, it is necessary to change the source code of this function in order to change the dynamic matrix and noise parameters. initialize() calls OpenCV's cvConDensInitSampleSet, which initializes the random sample set's state (x and y location at time steps t and $t-1$ in our case, refer to section section 4.2.5) between an upper bound and a lower bound so that the samples are not out of the tracking area.

initialize()

```
CvMat *lowerBound;
CvMat *upperBound;

lowerBound = cvCreateMat(4, 1, CV_32F);
upperBound = cvCreateMat(4, 1, CV_32F);

float F[] = {
 2, 0, -1, 0,
 0, 2, 0, -1,
 1, 0, 0, 0,
 0, 1, 0, 0,
};
memcpy(filter->DynamMatr, F, sizeof(F)); // set dynamical matrix

lowerBound->data.fl[0] = 0.0f;
upperBound->data.fl[0] = (float) WINDOW_WIDTH;
lowerBound->data.fl[1] = 0.0f;
upperBound->data.fl[1] = (float) WINDOW_HEIGHT;

lowerBound->data.fl[2] = 0.0f;
upperBound->data.fl[2] = 0.0f;
lowerBound->data.fl[3] = 0.0f;
upperBound->data.fl[3] = 0.0f;

cvConDensInitSampleSet(filter, lowerBound, upperBound);

cvRandInit(&(filter->RandS[0]), 0 /*mean*/, NOISE_STD_DEV_X /*std dev*/,
   0 /*seed*/, CV_RAND_NORMAL /*distribution*/);
cvRandInit(&(filter->RandS[1]), 0 /*mean*/, NOISE_STD_DEV_Y /*std dev*/,
   1 /*seed*/, CV_RAND_NORMAL/*distribution*/);
...
```

cvConDensInitSampleSet takes as input the pointer to the filter structure and the aforementioned lower and upper bounds. Notice the third and fourth dimensions are all 0 as they refer to the position of the particles at time $t-1$. **cvRandInit** initializes the random noise vector applied to the samples. The parameters accepted are self-explanatory in the code above.

After initialization, an iteration of the tracker consists of calling **estimate** followed by **update**. The former function performs the measurement stage of CONDENSATION while the latter performs the selection and prediction stages.

estimate(x, y, img, mahal)

```
/* Output parameters
 * x, y - estimated position of the object
 *
 * Input parameters
 * img - image to be measured
 * mahal - image skin probability density
 */

 x = y = 0;
 totalWeight = 0;

 forall i in samples do
    if !withinViewport(i) do
       i.weight = 0;
    else
       i.weight = measureWeight(i, img, mahal);
       x = x + i.weight * i.x;
       y = y + i.weight * i.y;
       totalWeight = totalWeight + i.weight;
    endif
 endforall

 x = x/totalWeight;
 y = y/totalWeight;
```

measureWeight simply implements the weight function described in 4.2.6.

Afterwards the call **update()** simply calls OpenCV's **cvConDensUpdateByTime(filter)**.

The remaining function **isObject(x, y)** is used to evaluate the standard deviation of the particles in order to detect the presence of a hand. The value of this standard deviation and its change requires editing of the source code of the function.

6.3.1 Controlling the mouse pointer

The functionality of the mouse pointer is encapsulated in **MouseController.cpp**. This file contains the functions required to move the system's mouse pointer, concretely the X Window System (does not work for other Windows, MacOS or GTK based systems). The library which contained this functionality was XTest, which is part of Xlibs.

A MouseController class is constructed using the empty constructor **MouseController()**, which

The relevant public functions are:

- **setCurrentReferencePosition(x, y)**. Sets the current reference position in order to calculate amounts of change in different directions in the future. Used by the tracker to set the current hand's position.
- **update(x, y)**. Moves the mouse pointer from its current position to the corresponding location calculated from the input position (x,y) and its difference with respect to the reference position.
- **leftClickDown()**. Performs the left click button down functionality.
- **leftClickUp()**. Performs the left click button up functionality.
- **getDeltaX()**. Returns the amount of change in X direction between the last two mouse position updates.
- **getDeltaY()**. Same as above for Y.

It should be noted that the reference position mentioned is not the mouse pointer's position, but any position which could be served as a reference to calculate relative movements. In the context of our project it is the hand's current position. **setCurrentReferencePosition(x, y)**, apart of setting the current reference position also obtains the current position of the mouse pointer, which the class needs internally to move the mouse pointer, as it's location is obviously different from the input reference position. In order to obtain the pointer's location the following calls are necessary:

```
Display* display = XOpenDisplay(NULL);
/* Screen where the pointer is displayed */
Screen *screen = ScreenOfDisplay(display, i);

/* obtain pointer position */
ret = XQueryPointer(display, RootWindowOfScreen(screen),
                    0, 0,
                    &currentCursorX, &currentCursorY, 0, 0, 0);
/* ret = true when the pointer's position has been obtained. False otherwise */
```

The important external function here is **XQueryPointer**, which obtains the pointer coordinates.

```
Bool XQueryPointer(Display *display, Window w, Window
 *root_return, Window *child_return, int *root_x_return, int *root_y_return,
 int *win_x_return, int *win_y_return, unsigned int *mask_return);
```

display is a pointer to the structure that controls a display in the X server and is obtained using the **XOpenDisplay** function. The current pointer position is saved to *currentCursorX* and *currentCursorY*. The rest of the parameters are irrelevant in our case and should be used as indicated above. The function returns true if the pointer's position was successfully obtained and false otherwise.

update(x, y) moves the pointer to the new corresponding location.

update(x, y)

```
deltaX = xpos - currentReferenceX;
deltaY = ypos - currentReferenceY;

if abs(deltaX) > 4 || abs(deltaY) > 4 then
   currentCursorX += deltaX;
   currentCursorY += deltaY;

   /* Check the cursor does not go out of the screen */
   if (currentCursorX < 0)
      currentCursorX = 0;
   endif
   if (currentCursorY < 0)
      currentCursorY = 0;
   endif

   ret = XTestFakeMotionEvent(display, screen, currentCursorX,
          currentCursorY, CurrentTime);
endif
```

The important external function here is **XTestFakeMotionEvent**, which moves the mouse pointer.

```
int XTestFakeMotionEvent(display, screen_number, x, y, delay);
```

The parameters are the *display* and output *screen_number*, which are the same as previously. *x* and *y* are the locations to which to move the pointer and *delay* the amount of delay of the event, which is assumed to be 0 when delay is set to *CurrentTime*. The function returns true in case the motion event was successful and false otherwise.

The mouse click functionality is achieved through **XTestFakeButtonEvent**, which is used by **leftClickDown()** and **leftClickUp()**.

```
ret = XTestFakeButtonEvent(display, button, is_press, delay)
```

The specification of XTestFakeButtonEvent is:

```
int XTestFakeButtonEvent(display, button, is_press, delay);
```

display and *delay* adopt the same values as previously. *button* is self-explanatory (equals 1 for the left button) and *is_press* is true for button press (click down) and false for button release (click up). The function returns true in case the button event was successful and false otherwise.

6.4 Gesture recognition

In order to perform gesture recognition the input image has to be binary segmented first, which we already did before. The segmented image, however, is noisy and has to be processed in order to remove the noise. In capture() we summarized the operations as processBinSeg(), which in reality consists of applying morphological operators, finding and removing connected components.

The morphological operators of erode and dilate were applied using OpenCV's **cvErode** and **cvDilate**. Since both have the same structure, only cvErode is shown below:

```
cvErode(source_img, dest_img, struct_element, iterations)
```

source_img is the source image, *dest_img* the destination image, *struct_element* the structuring element (default is 3-by-3) and *iterations* is the number of iterations applied (default is 1).

Connected components are found using OpenCV's **CvContourScanner**, which is created using **cvStartFindContours**.

```
CvContourScanner cvStartFindContours(CvArr* image, CvMemStorage* storage,
   int header_size=sizeof(CvContour), int mode=CV_RETR_LIST,
   int method=CV_CHAIN_APPROX_SIMPLE, CvPoint offset=cvPoint(0, 0))
```

image is the binary source image. *storage* is a container of the retrieved contours. *header_size* is the size of the sequence header. mode is the retrieval mode. *method* is the approximation method used and offset the region of interest offset. The function returns CvContourScanner, which is a structure used to traverse the contour.

We used this function in the following manner:

```
CvContourScanner scanner = cvStartFindContours(image, contourStorage,
            sizeof(CvContour), CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE);
```

where CV_RETR_EXTERNAL means we are only interested in the outer contours and CV_CHAIN_APPROX_SIMPLE simply determines how the points of the contour are stored (for further information please refer to OpenCV's documentation).

The contours are then scanned using **cvFindNextContour**, which on successive calls returns the next non-visited contour.

```
while ( (c = cvFindNextContour(scanner)) != NULL ) {
  area = cvContourArea(c);

  if (area < 5000) {
    cvSubstituteContour(scanner, NULL);
  }
}
```

The code above checks that all contours found have an area greater than >= 5000. Otherwise they are eliminated. This way larger portions of noise which were not removed by the morphological operators are now eliminated.

Once the segmented image is clean of noise, we proceed by applying our gesture recognition algorithm. The relevant OpenCV functions used were **cvApproxPoly**, **cvConvexHull2** and **cvConvexityDefects**.

```
CvSeq* cvApproxPoly(const void* src_seq, int header_size,
  CvMemStorage* storage, int method, double parameter, int parameter2=0)
```

src_seq is a sequence of points. *header_size* is the header size of the approximated curves. *storage* is where the approximated contours are saved. *method* is the approximation method used. *parameter* is a method-specific parameter; in the case of CV_POLY_APPROX_DP it is a desired approximation accuracy.

```
CvSeq* cvConvexHull2(const CvArr* input, void* hull_storage=NULL,
   int orientation=CV_CLOCKWISE, int return_points=0)
```

input is a sequence of points. hull_storage is where the convex hull is saved. orientation is the orientation of the convex hull (points ordered clockwise or counter-clockwise). return_points is non-zero when the points will be saved in the return.

```
CvSeq* cvConvexityDefects(const CvArr* contour,
   const CvArr* convexhull, CvMemStorage* storage=NULL)
```

contour is the sequence of points. *convexhull* is the convex hull of the contour. *storage* is where the found convexity defects will be stored.

These functions were used in the following way:

```
// Polygonal approximation of the contour (hand)
cPoly = cvApproxPoly(contour, sizeof(CvContour), 0, CV_POLY_APPROX_DP, 20, 0);

// Convex hull of the polygonal approximation
cHull = cvConvexHull2(cPoly, 0, CV_CLOCKWISE, 1);

// variable checked in capture()
// true when the enclosing circle of the cHull is completely within the viewport
hand_totally_inside = drawEnclosingCircle(cHull, img);

// compute convexity defects
cDefects = cvConvexityDefects(cPoly, cHull);
numDefects = cDefects->total;
```

```
    // filter defects for valid and invalid defects
    filterDefects(cDefects);
```

Once the defects are detected the corresponding checks based on the deepest points in the defects are used for gesture recognition, as explained in 5.4.5.

Chapter 7

Results

In this section the results and evaluation of each phase of the project are presented in approximate chronological order.

Firstly, hand shape enhancement will be evaluated. Subsequently, we will proceed with the results of the hand tracker showing the different phases of the implementation process and the results obtained. Afterwards, the results of binary segmentation and our gesture recognition algorithm will be shown. To conclude, the results of the whole human-computer interface in action will presented.

It should be noted that when performance tests were performed both the images captured in real-time and the segmented image (both 640x480) with the corresponding calculations on it were shown on screen, which is computationally expensive but gives more feedback to the user. Therefore, this should be factored when interpreting the performance numbers shown in the next sections.

7.1 Hand shape enhancement

Recalling from chapter 3 the starting point was to obtain a probability density image that would describe the skin-likelihood of each pixel in the image, which was thereafter used for the measurement phase in the CONDENSATION tracker and for binary segmentation. A measure of the adequacy of each colour space was their performance in binary segmentation.

The test set of images consisted of a series of photographs of the author's right hand under different lighting environments (appendix B). These environments were basically the places where we developed our system. Under fluorescent light three images were taken at home and four at the workplace at the university at night, so that daylight would not affect them. During the day, four pictures were taken at the workplace under both fluorescent light and daylight (coming from the window). And finally, a set of three photographs were taken in the outside under daylight, in the shadow, as photos under direct sunlight could not be used as they were severely overexposed. The hands from the images were segmented manually from the background.

All the photographs were taken with our webcam. It could be argued that the images obtained are not representative as only a hand of one subject was used, plus the images were all taken with the same camera and thus have a specific chromacity dependent on it. However, it is our belief that our approach could be easily extended to be more generalistic by simply using a broader test set, even though the number of false positives would most probably increase.

The first step in the enhancement phase was to choose an appropriate colour space. We selected two colour spaces and trained a skin detector based on single Gaussian skin distribution modelling. In figure 7.1 an original hand image and its segmentation (marked as green) in the (a,b,c) space (section 3.1.1) and HSV space are shown. In both colour spaces the hand could be segmented reasonably but the segmentation quality in the (a,b,c) space depended heavily on the skin model used and only performed better when these skin models have been trained specifically for those illumination conditions. Selecting a specific model each time, however, is not desirable as the process should be as automatic as possible, involving less user interaction. Even though it is technically possible to automatically test the skin models 'on the fly' and choose the most appropriate one each time, it is not convenient for our system as efficiency is a major concern and such method would certainly have a big impact on the performance.

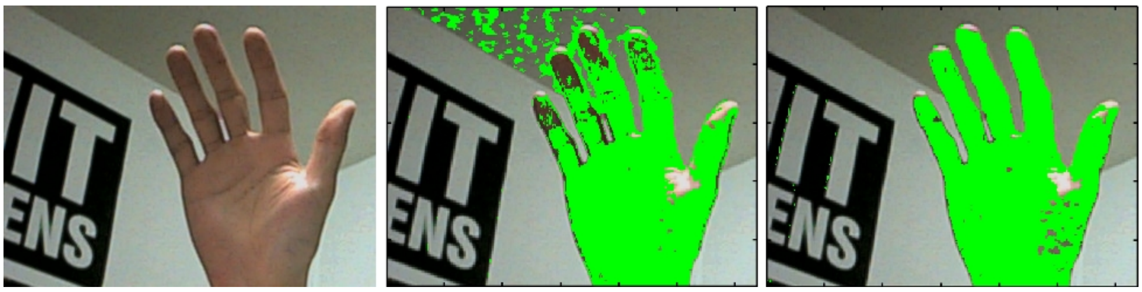


Figure 7.1: Segmentation of a hand under fluorescent light. The matrices used for both were those obtained for daylight conditions. The picture in the *middle* was segmented in the (a,b,c) space and the Mahalanobis distance limit to the calculated mean skin colour was $\theta = 2.8$. The *right* picture was segmented in the HSV space, and the Mahalanobis distance limit was $\theta = 2.4$. It can be seen that when comparing matrices associated to the same conditions HSV's performance is better.

Since the HSV colour space offered better binary segmentation the resulting skin probability density image would also be better, hence we chose HSV over (a,b,c). Being the HSV space more suitable for our case does not mean it does not have its drawbacks too. Even though no performance tests were carried out, conversion to the HSV space was noticeably slower than to the (a,b,c) space, so there is area for improvement in this sense, starting by considering other colour spaces for example.

7.2 Hand tracking

The hand tracker implementation stage was divided into two phases so that the test environment would be more controlled. The first phase involved creating a prototype in order to test our understanding of the Condensation algorithm and the parameters it needed. This test would consist of a very controlled environment with no noise. The second phase was about testing the tracker in a real situation and thus the parameters considered previously were evaluated and tuned for the movement of a hand.

7.2.1 CONDENSATION prototype

The CONDENSATION prototype consisted of a small window with a black background on which a white square could move in any direction at constant speed (figure 7.2). The white square simulated the segmented element which we had to follow. The tracking algorithm was then applied in order to see if it could follow the white square successfully. Recall the tracked object's location was estimated as a weighted average of the samples' positions.

The first problem encountered was with the initialization of the algorithm. We used a dynamical model based on the positions at time step t and $t+1$, which were calculated by applying a dynamic matrix to a matrix of positions at t and $t-1$ (section 4.2.5), thus the position of the tracked object at the previous time step was needed. The initialization call of the algorithm in OpenCV created a set of random samples at random locations, treating all dimensions of a sample equally. During tests we saw that it was then necessary to make sure that the position of the samples at t and $t-1$ were the same initially, as otherwise they would be drifted away during the prediction phase, or more intuitively, samples would move out of the viewport whether the actual object to be tracked was visible or not.

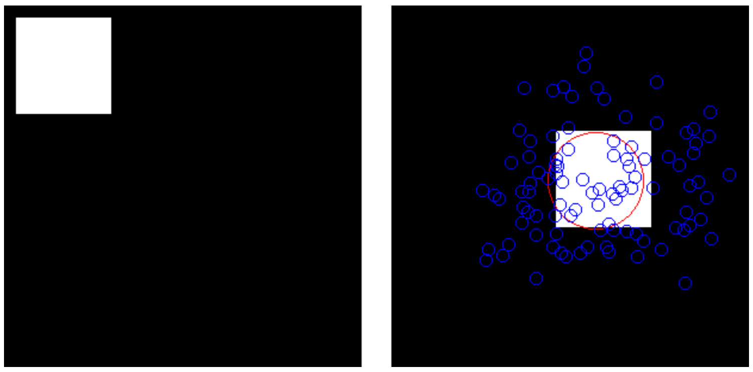


Figure 7.2: Condensation prototype used to test the algorithm. The tracker had to follow a moving white square, which simulated a segmented element.

The next task was to check that the weight function satisfied our requirements. Initially, Holden and Owens (2003)'s weight function was used, which consisted of a Gaussian function based on the number of skin pixels within a 40x40 square window. We considered that such window of 1600 pixels was too big and tried a much smaller window of 64 pixels (8x8), which yielded the same results in the end and was much more efficient. However, this function was conceived for an already segmented hand and therefore we had to change it to suit our approach, which was to apply it directly on the unprocessed image.

Once we were sure all the parameters were correct and the tracker was working, we tried to add a second white square in order to test the supposedly natural multitracking capability of the CONDENSATION algorithm. If both squares were visible from the start the samples would revolve around both squares. On the contrary, if they were initially hidden, then the samples would settle around the square which would become visible first, completely ignoring the other when it would finally appear. If the elements to be tracked were not of the same size, the density tended to shift towards the bigger one. The CONDENSATION algorithm as it was originally concieved was then not very suitable for multiple target tracking as the multimodal density could only be maintained for a small period. However, with certain changes the algorithm could definitely do so. For example, it could be set that after detection of an object a certain percentage of samples would still keep independent of the prior density and move randomly across the image. For further information the reader is encouraged to read appendix A.

7.2.2 CONDENSATION applied

Once we were certain that the prototype implementation was working as expected we proceeded to apply it to a real situation, but first we had to change the weight function since we did not have the benefit of an already segmented image or a background model(recall section 4.2.6).

In order to determine the parameters for the dynamical model a test set consisting of images of a hand held moving lantern was used so as to mimic the random movement of a hand. This way we could obtain a generalization of the stochastic parameters of the horizontal and vertical movement of a hand in front of the camera through time. Further information is shown in appendix C.

When the dynamical model had been defined the tracker was tested against a moving hand at the workplace with it randomly moving in front of the camera.

Initialization consisted simply of letting the tracker settle to a steady state (samples uniformly distributed across the image) through iterations without any hand. When a hand appears on scene the samples rapidly concentrate around it approximating a Gaussian distribution, which is also maintained during temporary occlusion. Figure 7.3 shows that thanks to the dynamical model the distribution can be maintained when the hand disappears momentaneously as the tracker estimates its location given its previous position in time. Even though the camera was able to capture at 30 frames/sec, we only achieved 5 frames/sec with a distribution of $N = 500$ samples at a resolution of 640x480 with both CONDENSATION and segmentation turned on. At $N = 100$ the frame rate increased to 6 frames/sec only, so the former amount of samples was left for higher accuracy.

Figure 7.4 shows the samples gather across the whole hand but dispersion seems rather high, which suggests that the weight function could be further improved. The sequence of images show that when moving the hand at a reasonably high speed the estimated position by the tracker is slightly off the centre of the hand (if too fast, track would just be lost). This is due to our first order dynamical model, which does not take the hand's acceleration into account and the samples cannot keep up with the pace of the hand. However, since the purpose of the tracker was not to track the location as exact as possible but for moving the mouse pointer, adding complexity to the model with a second order approach did not seem necessary. In any case, if the tracker happened to have lost track of a rapidly moving object the recovery speed was instantaneous when the object slowed down. Accuracy could be increased with a higher amount of samples but the impact on performance would be considerable.

A possibility that might happen in presence of skin coloured elements in high clutter (noise, for the most part) is that the clutter may have the higher posterior density. In that case the particles will settle around the hypothesis with highest probability and the appearance of a hand in other areas of the image may be ignored. In figure 7.5 we can see that the hand is completely invisible to the tracker, only reacting when it approaches the previous hypothesis. While this does not affect the tracker much in the sense that tracking (and thus pointer movement) is still possible,

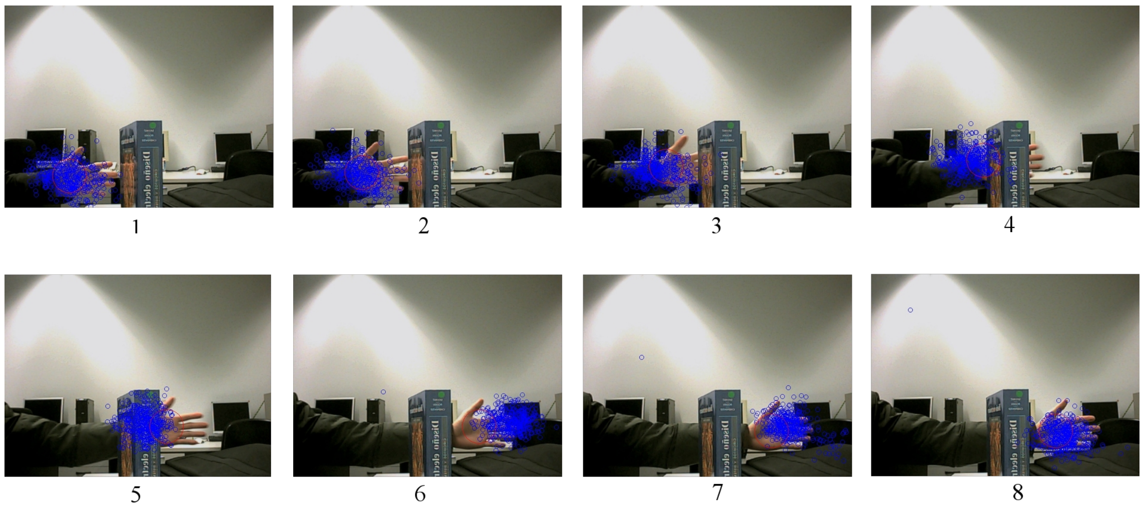


Figure 7.3: Sequence of images showing the benefits of a motion estimator when partial occlusions occur. As the hand travels from left to right, it is occluded by the book but the samples continue moving along based on the state at the previous time step. Notice the transition from frame 3 to frame 5, where the samples are over the book due to our dynamical model of constant velocity, even though the hand is not visible there (the estimated position is not the visual mean position of the samples because the samples on the hand at the left have more weight).

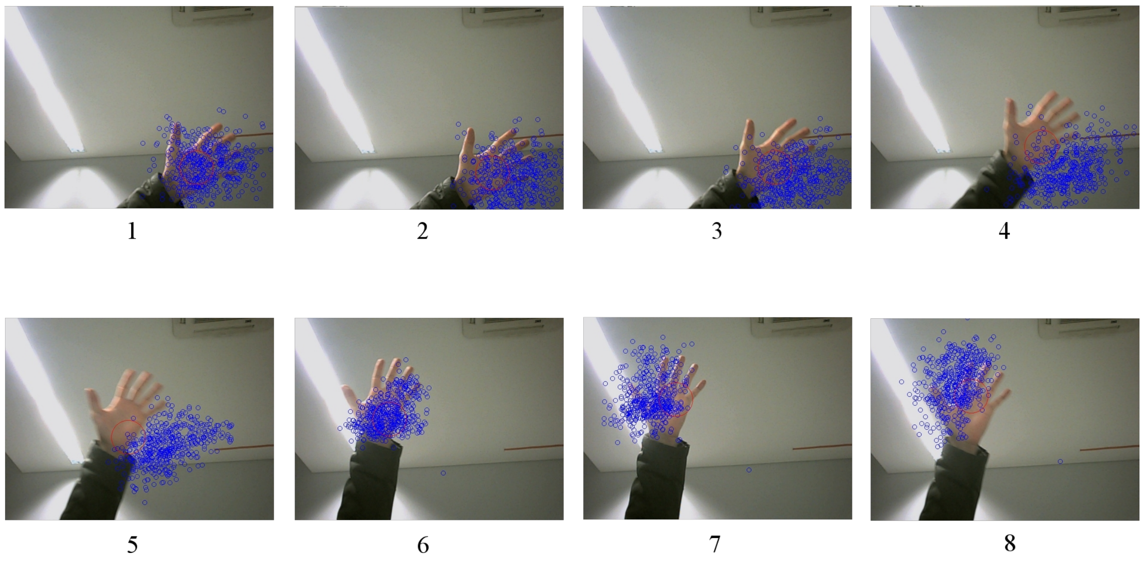


Figure 7.4: Sequence of images of a moving hand. It can be seen that in frames 4, 5, 7 and 8 the tracker loses a bit of track of the hand as it travels faster (red circle indicates the estimated location), but when the hand is stopped the tracker recovers instantly.

the segmented result will not be useful for recognition if it cannot be completely cleaned, as it is prone to error. On the other hand, this can also be seen from another perspective in which a hypothesis is not so much affected by noise. If the tracker is following a hand and the hand passes over a small noise area or close to it, then the noise will only be assigned temporary probability, which will disappear eventually for lack of high weight samples around. In any case, the problem could be solved by using knowledge from binary segmentation or with multitracking, in which case multiple hypothesis could be tracked and the small area of noise could be ignored.

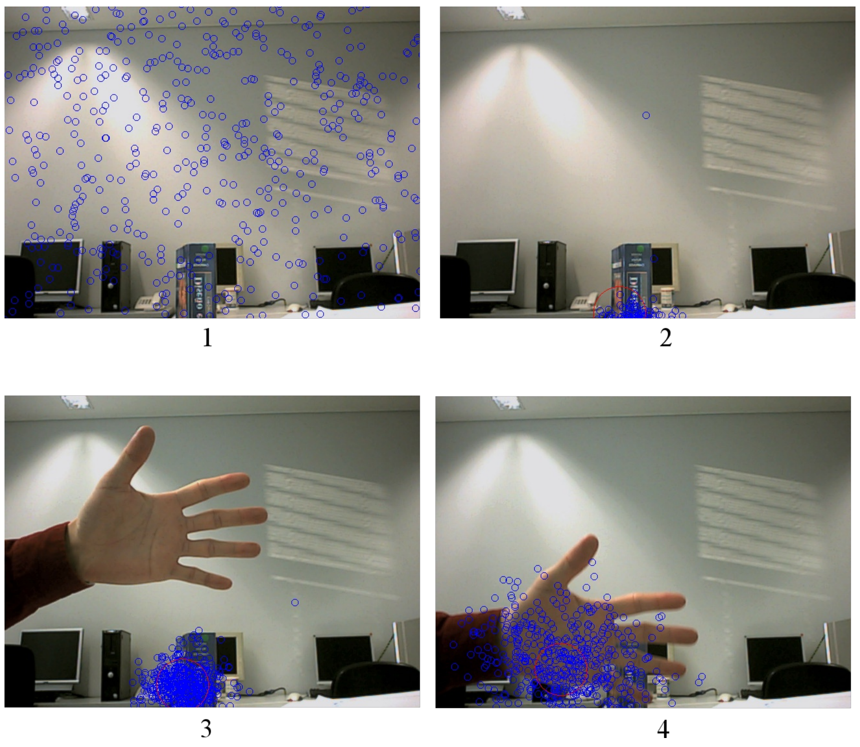


Figure 7.5: Sequence of images where the samples have been initially distracted by noise. When a hand appears in the third frame it is completely ignored until it approaches the samples.

7.2.3 Hand presence detection

We have already seen that our final chosen method of hand detection based on the standard deviation of the samples positions was enough for our purposes. In this section we will compare this method to the first method conceived, which consisted of calculating the median of the probability densities at each pixel within an 8x8 window centred at the estimated location.

Both methods effectively detect the hand when it is within the viewport, either when the hand is moving faster or slower, as long as the particles are able to track the hand. Which method is more efficient depends on the number of samples (for the standard deviation) and the window size (for the median). In our case the window size is smaller, so the median seemed as the first choice. However, using the median the hand would be lost during temporary occlusions. Even though the hand is not expected to be occluded in general, it is a good feature from motion estimators and we decided to keep it, thus the standard deviation was finally used.

It is noticeable in the images in this chapter that the hand's appearance on screen is effectively detected with our samples dispersion approach (section 4.3). However, it should be noted that this method is not the best as in situations where the hand disappears from the viewport the particles return to a balanced steady state (uniformly distributed across the image) and during this return their dispersion is slowly increased, thus the tracker believes a hand is still visible when it clearly is not.

To picture the consequences of this, we have to see the tracker's estimation applied for moving the mouse pointer. Straightforward implementation of our pointer movement algorithm using the estimated position of the hand allows us to move the pointer. Consider now a situation where we start with our hand at the centre of the viewport and move to the left until it completely disappears. The pointer will subsequently move towards the right relative to the position where it was initially, but when the hand disappears the pointer will **bounce** slightly to the right again as momentaneously the tracker thought the hand was still visible (figure 7.6).

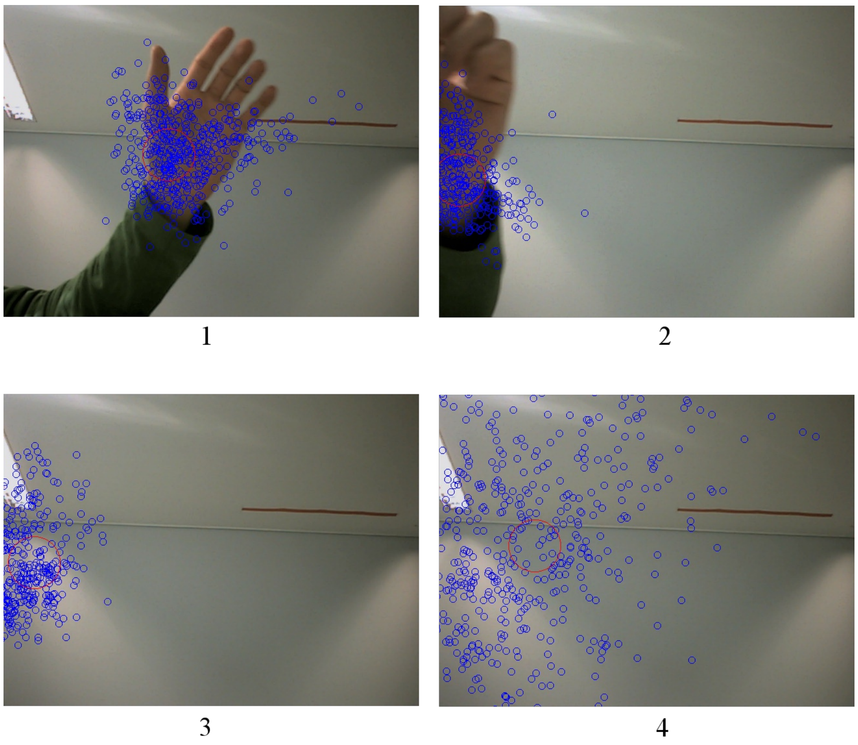


Figure 7.6: Sequence of images where the tracked hand disappears towards the left side. With our hand presence detection method the estimated location bounces back since the dispersion of the particles is still low.

A solution to this issue was not found without resorting to segmentation, but in the end it was left as was since the gestures for triggering the tracker on and off meant this was no longer a problem.

7.3 Gesture recognition

7.3.1 Binary segmentation

Having chosen the adequate colour space during evaluation of the hand shape enhancement method, the problem remaining for binary segmentation was to choose an adequate threshold to discriminate skin and non-skin pixels. This threshold was the Mahalanobis distance of the colour of a pixel to the mean skin colour we had found.

In figure 7.7 the chart shows the ratio of true positives to false positives depending on the Mahalanobis distance (squared) used for the same picture and table 7.8 presents the data in more detail. A threshold can be determined as a trade off between true positives and false negatives.

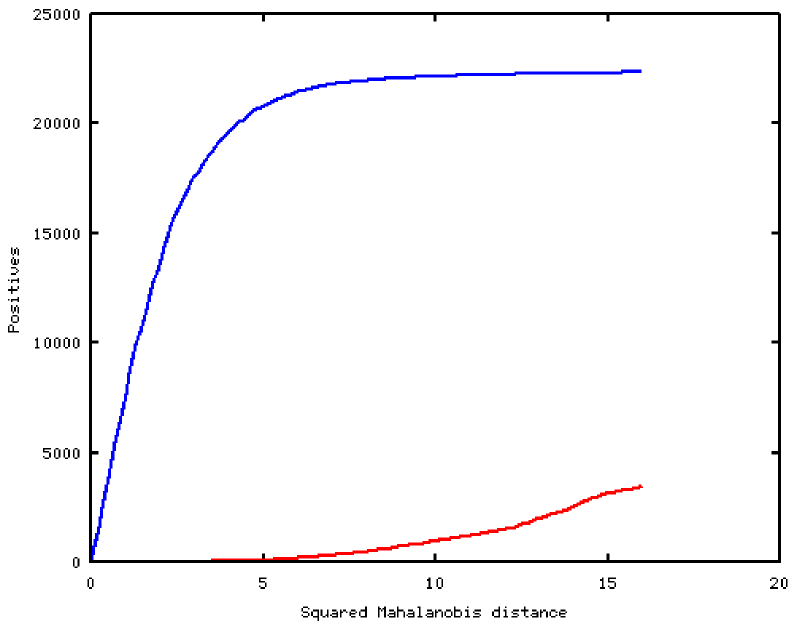


Figure 7.7: True positives (blue) and false positives (red) as functions of the skin detection threshold measured in terms of squared Mahalanobis distances to the mean colour in HSV space. Using this a value that maximizes the number of true positives for a determine amount of false positives can be determined. In our case a value around $\theta^2 = 6$ was chosen.

Squared Mahalanobis distance	**True positives**	**% Total skin pixels**	**False positives**
1	7719	(34.19%)	10
2	13856	(61.37%)	28
3	17633	(78.09%)	40
4	19657	(87.06%)	59
5	20849	(92.34%)	107
6	21477	(95.12%)	206
7	21803	(96.56%)	324
8	21979	(97.34%)	494
9	22086	(97.82%)	734
10	22154	(98.12%)	978
11	22208	(98.36%)	1241
12	22248	(98.53%)	1515

Figure 7.8: True positives and false positives based on the Mahalanobis distance. Between 5 and 6 the increase in detection rate begins to slow down while the number of false positives begins to grow faster.

However small the number of false positives or undetected skin pixels, they can affect the recognition of the gesture and thus noise clean up is necessary, which is done first applying the *opening* morphological operation followed by *closing*. Once small noise areas are removed, bigger noise areas might still be left, which are removed by detecting connected components and discarding those which have an area smaller than our considered threshold. The final result should be a cleanly segmented hand ready for recognition (see figure 7.9), as it would not be useful otherwise.

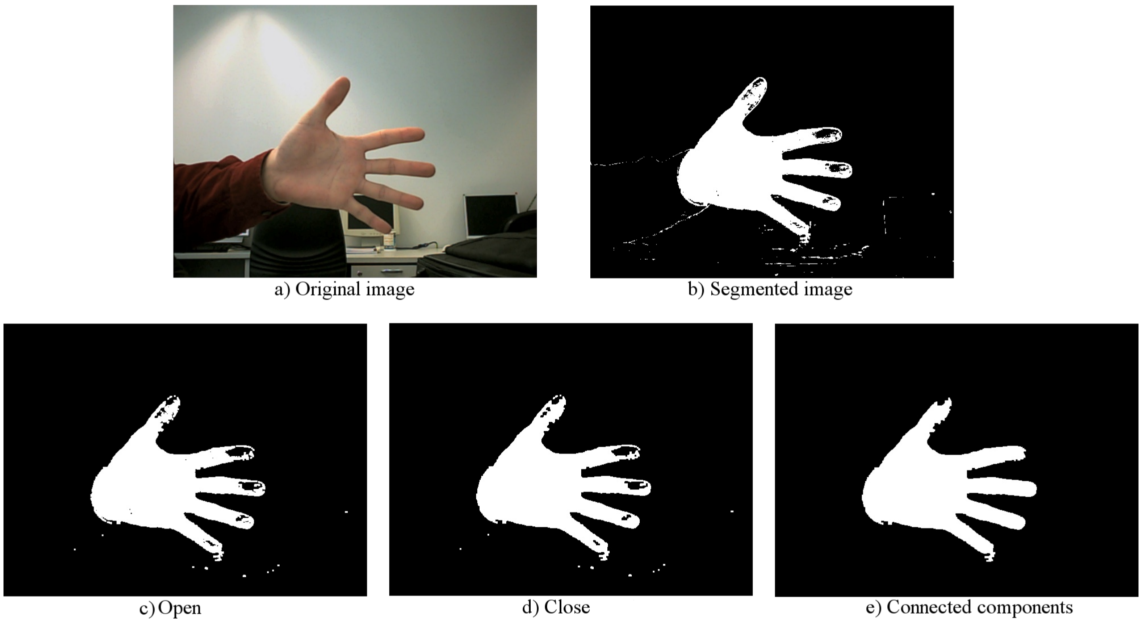


Figure 7.9: a) Original input image. b) Segmented image, with a lot of noise. c) Open shrinks small areas of noise. d) Close fills small holes and reconstructs area deleted by the opening. e) Bigger noise areas are detected as connected components and removed. The largest remaining connected component is then filled.

So far the results achieved were already good enough for recognition, but we still wanted to make it more robust to varying illumination and thus we implemented an adaptive skin colour segmentation algorithm as detailed in section 5.2.2. We used a testing image set consisting of a sequence of 84 frames of a moving hand where segmentation could be improved in some of them.

In our tests (figure 7.10) the adaptive algorithm provided important improvement in some cases and only resulted in marginally worse segmentation in few others. It can be observed that the images obtained with the adaptive algorithm are in general less noisy. In frames where the non-adaptive model yields an important amount of holes in the hand it can be seen that these same holes are much smaller when using the adaptive model.

7.3.2 Recognition

For gesture recognition to be triggered the segmented hand must be stationary (or almost) in order to minimize erroneous detections. Furthermore, the system will only try to recognize the gesture if the segmented hand is the only segmented area in the image. This means that if there is too much clutter and more than one skin patch is detected the recognition system will be inactive. If the hand is not completely inside the viewport, then it will also not be considered for recognition.

In the case of figure 7.11 we can see that other skin-coloured regions were detected gesture recognition cannot be triggered. This problem could potentially be solved by background substraction methods or multitracking, or maybe all combined. In the case of background substraction methods we have the problems mentioned in 5.2. Using multitracking we could detect different clusters in

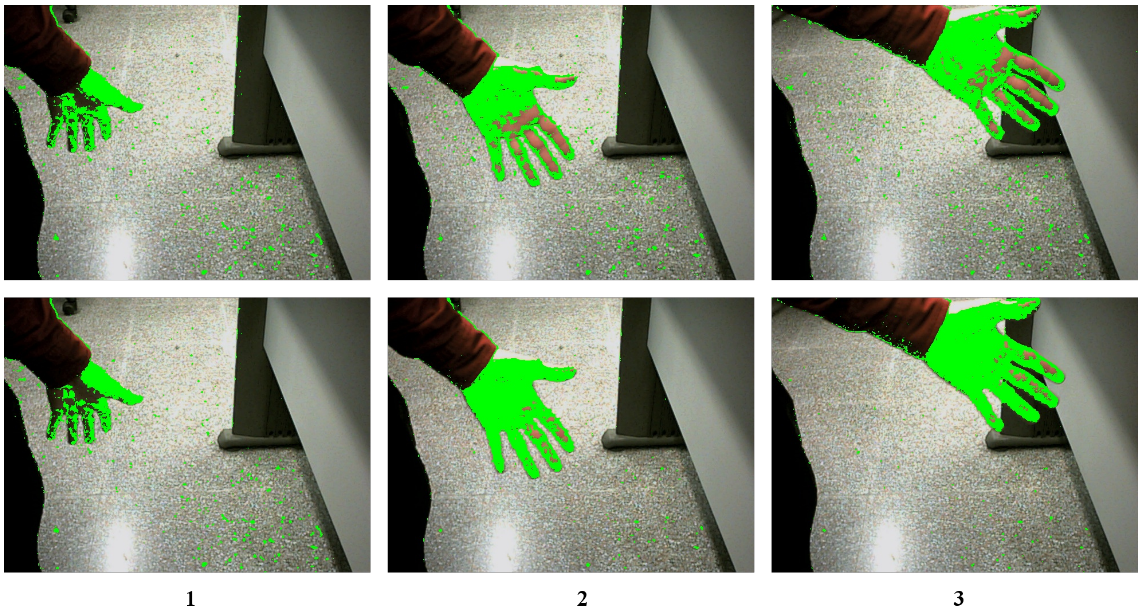


Figure 7.10: Some frames from a test sequence where the hand received uneven light. The skin pixels are marked in green while the background pixels are left as is. Normal segmentation is used in the upper rows and adaptive segmentation in the lower rows. It can be seen that the results of the adaptive segmentation are in the worst case only slightly worse, as is the case of frame 1 where there are more true positives in the case of normal segmentation. Especially noticeable improvement can be observed in frames 2 and 3, where the percentage of detected hand pixels is visibly higher. Normal segmentation left several holes in the fingers while adaptive segmentation left much smaller holes. Another observable improvement is that the amount of noise (false positives) is much smaller.

the image and our gesture recognition method could be used to discard valid and invalid gestures. In any case, it seems that the problem can be solved by combining these methods.

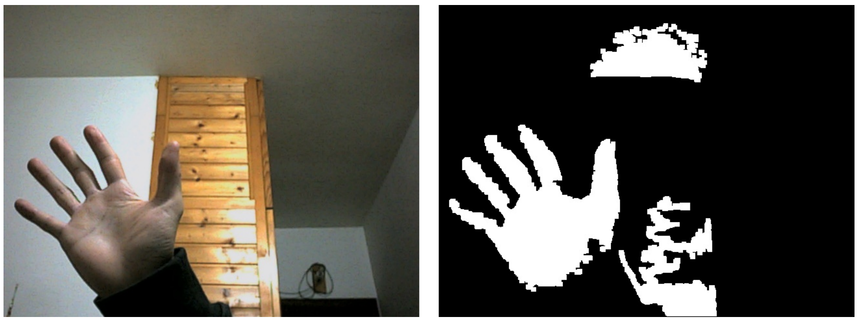


Figure 7.11: An image and its segmentation. When the hand cannot be reliably segmented gesture recognition is not possible. Since the tracker uses a weight function based on skin colour it gets confused as well, so it cannot be used to distinguish the hand cluster from the rest.

In figure 7.12 the corresponding segmentations of our gesture set are shown. The functions of these gestures were mouse pointer motion trigger on and off, and mouse left button click up and down (useful for drawing or dragging elements, for example). Motion stopping and left button up gestures were chosen to be the same as it appears to be more intuitive to the user. From the pictures it is obvious that the method is both rotation-invariant and scale-invariant, which is useful in our project as the hand is expected to move around and staying exactly the same all through is difficult. However, in case of need orientation can be calculated from the position of the fingertips and thus a new set of gestures could be conceived, taking advantage of this.

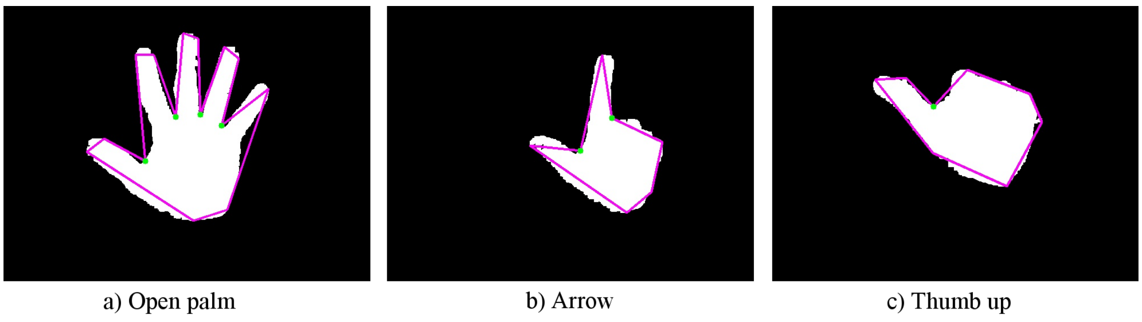


Figure 7.12: Segmented images of our gesture set. Only the deepest points in the convexity defects have been indicated (green points).

The system also proved adaptable to different hand morphologies, as it was tested with two other subjects with the same results. This is due to the fact that we only made use of convexity defects, but further testing would be necessary to confirm that the system is also robust if the extension of considering fingertips and the angles between them were to be implemented.

Performance of gesture recognition was completely related to segmentation quality and thus no specific test set was made in order to test the accuracy as they could all be biased towards a certain result. In general, slight defects in segmentation did not affect the recognition due to our mechanism of approximation, but it was difficult to determine to which extent the system was still able to recognize a gesture properly. In absence of clutter and with uniform illumination the system could achieve 100% accuracy due to the simplicity and robustness of the geometric characteristics considered. As far as time performance, the frame rate dropped from 5 fps to 4 fps when segmentation, tracking and gesture recognition were combined. Without the tracker the frame rate rose to 7 fps, which appears to be on par with some methods consulted in literature such as Fang et al. (2007), who achieved 10fps. It is important to take into account that these frame rates were based on live capture from the camera, not a previous set of images, which would be faster. Due to different equipment, algorithms and programming approach it is certainly difficult to establish comparisons, but the simplicity of our method leads us to believe in its performance.

7.4 All combined: Human-Computer interface

Having tested all parts individually it was time to assemble them together and test the whole system. The functionalities associated to the recognized gestures were added so that we could move the mouse pointer and simulate left click.

The first important thing was to make sure that gestures would only be recognized when the movement of the hand is minimal (± 4 pixels), as errors would be much less likely to appear. For this we simply moved our hand in front of the camera and saw this is the case. We found that for higher robustness it would probably have been better to have implemented a triggering moving gesture as otherwise, if our hand stays stationary for a frame, gesture recognition could be triggered.

Afterwards we proceeded to test that mouse pointer trailing could be activated or deactivated on demand. By default mouse pointer motion was disabled. We started by placing our open palm in front of the camera so that it would fit completely inside the image area, since the open palm is recognized as pointer motion switch off. Then we tested that trailing could effectively be activated and deactivated on the fly by switching between open palm and arrow gesture, which worked as expected. Figure 7.13 shows the transition from an open palm to the arrow gesture and the posterior mouse pointer movement.

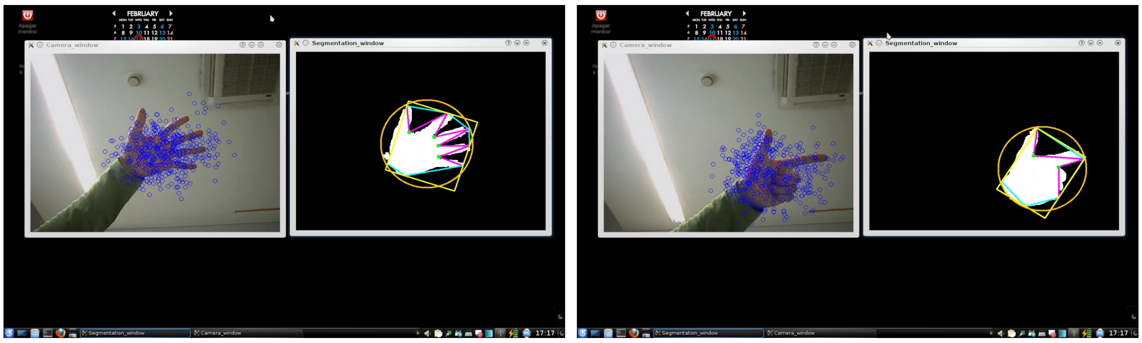


Figure 7.13: *Left*: open palm gesture, which stops pointer motion. *Right*: arrow gesture, which triggers pointer motion. Notice the new position of the hand in the *right* image and the new position of the pointer.

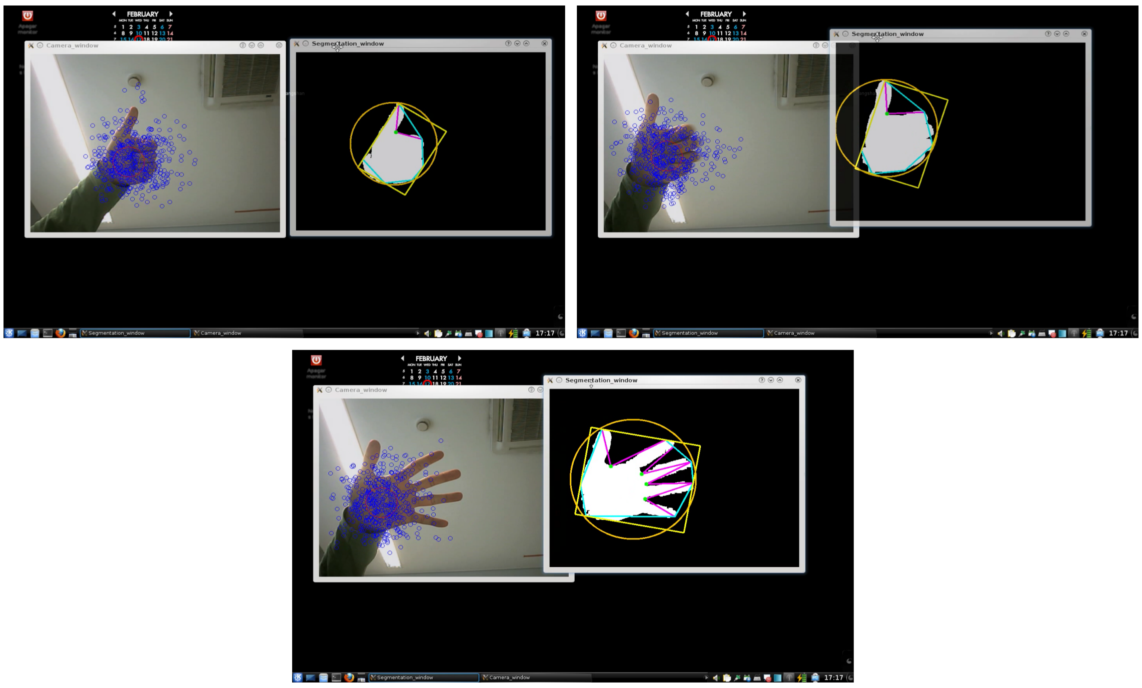


Figure 7.14: In the top-left image the mouse pointer is over the window bar and the gesture thumb-up has been performed, which means left button click down and thus the window is dragged. The top-right image shows that the window can be moved while it is dragged. Bottom image shows the window has been released after the open palm gesture triggered the left button click up functionality.

To test the left button click we decided to drag a window. Figure 7.14 shows the thumb-up gesture when the pointer is on the top of a window (recall that the necessary previous gesture was the open palm, as tracking has to be disabled in order to recognize a new gesture). The thumb-up gesture meant left button click down, which meant that maintaining this gesture we were then able to move the dragged window around the screen. It is interesting to note that given the small image resolution of the camera the hand has very limited motion, thus it was difficult to drag the window from a top corner to a bottom corner, for example. A trick to overcome this was to make the hand disappear towards the direction we intended to move and reappear at the opposite side, iterated as long as necessary. The problem with this is that we would face the bouncing issue related to hand detection with samples dispersion mentioned in a previous section, thus doing so while triggering mouse pointer motion on and off successively could solve the issue. After opening the palm again this time the gesture would be interpreted as left button up and the window is dropped.

With all the required functionality tested we could conclude that the goals set initially were achieved. More than the number of actual gestures recognized the concerns were related to robustness and performance. Robustness has been achieved for the functionality required but a performance of only 4 fps with a 2.4GHz processor can be certainly improved, even though these 4 fps were calculated with the captured images and the corresponding processed segmentation shown on screen. While the tests indicate that acceptable real-time interaction is possible, a clear path of progression in the future is to make the experience smoother.

Chapter 8

Conclusions

Human-Computer interaction using hand gesture recognition is a field that increasingly gets attention year after year. The methods involved have been traditionally computationally intensive for real-time usage but newer and more powerful computers have allowed much progress in the field. We have achieved our initial goals of implementing a gestual interface based on just a camera and even though we did not achieve competitive real-time performance we constructed a solid basis which can certainly be further improved.

The logical first step in such an interface was to obtain the probability density of the location of the hand. The use of a single Gaussian model for skin distribution modelling meant that we required little training data in order to obtain a general model, as well as being simple and storage-efficient. The choice of colour space played a very important role as we struggled for robustness under different light conditions at the initial stages of the project. The (a,b,c) space was competitive when the models used in each environment had been trained specifically for them, but it was too much of an inconvenience to switch between models if a more robust one could be found. HSV space (HS, concretely), on the contrary, offered much more robustness to these changing conditions, as it explicitly separated chrominance from luminance, even though at the expense of some efficiency.

The location of the hand would serve for moving the mouse pointer. Hand tracking was necessary but segmentation alone had many inconveniences. Out of two motion estimators, Kalman filter and CONDENSATION, the latter was chosen as we believed it would perform better for our conditions based on comparisons found in literature. The first order dynamical model used based on the tracked object's positions at current and previous time steps proved sufficient for moving the mouse pointer, even though it could not achieve high accuracy at higher motion speeds. The window-based weight function served for the purpose even though the samples dispersion and estimated position suggest it could be further improved. In tests the number of samples required for decent accuracy was as low as $N = 100$ but we finally increased that number to $N = 500$ as accuracy improved dramatically and did not have as much impact on performance as could be expected, probably because the performance of the CONDENSATION algorithm improves as N increases (Isard and Blake, 1998a). Hand presence detection (in front of the camera) was achieved through analysis of the dispersion of the samples (standard deviation), which was useful but had inconveniences when the hand disappeared towards the sides of the image area, thus another better method could probably be conceived.

Once the mouse pointer movement part was done, the second part consisted of recognizing gestures to add functionality to the interface. Before being able to recognize gestures the hand had to be isolated from the background. From the skin probability density image obtained it was straightforward to obtain the binary segmentation of the hand. As the obtained segmentation was noisy, it was cleaned using morphological operators and finding small connected components. The resulting segmentation was generally good but additional robustness was sucessfully added through an adaptive segmentation technique, achieving improvement and reducing the amount of noise.

When the hand was successfully separated from the background we could proceed to recognize the gesture. A small gesture set for the functions of mouse pointer movement on/off and left button click up/down was considered, but the method is extendable to a broader gesture set. Geometrical features were obtained from the segmented hand for the recognition algorithm and thus its robustness was highly dependent of the quality of the segmentation. Even though the method had some requirements regarding the position of the hand with the palm necessary facing the camera, fingertips and valleys between them could be successfully detected without heavy impact on efficiency, thus we believe the algorithm obtained was quite powerful.

Aside of the issues mentioned, the major concern was the efficiency obtained, which was only of 4 fps with images of 640x480 pixels. This number, however, was obtained with all the samples and the segmentation result being painted and shown on screen together with the images captured in real time, thus the true performance is probably much better. In any case, some areas seem conceptually inefficient. For example, the tracking process could probably help segmentation in order to specify a region of interest and avoid segmenting the whole image area. We did not find a way to do so, but if this could be improved, then the whole process would be sped up by several orders.

All parts combined, the initial goal of constructing a simple human-computer interface substituting a common mouse device was achieved. Were the project to be continued, the working direction would seem to be to address the issues mentioned.

8.1 Further work directions

Apart from the possible improvements mentioned previously, there are also other possible directions, some of which were considered but finally left out of our scope. Some of these are:

- Find a way to segment only the hand and discard the forearm. In the project, one important requirement was to wear sleeves to avoid this problem, since this was not an easy task and existing literature has also opted to omit it.
- Expand gesture set making use of the fingertips and the angles between them.
- Improve segmentation in cases where there is heavy skin-coloured clutter.
- Recognize gestures in movement. To minimize errors, recognition was only triggered when the hand was static, but adding recognition in movement is useful for gaming applications, for example.

- Multiple hands tracking for enhanced functionality. This seems to be a natural extension of our system as we can take advantage of the multi-modality of the CONDENSATION algorithm.
- Motion recognition. Only static gesture recognition was considered but recognizing moving gestures would add even more possibilities.

And so forth.

In conclusion, a whole new world of possibilities are possible and there is certainly room for future work.

References

S. Askar, Y. Kondratyuk, K. Elazouzi, P. Kauff, and O. Schreer. Vision-based skin-colour segmentation of moving hands for real-time applications. In *Visual Media Production, 2004. (CVMP). 1st European Conference on*, pages 79–85, March 2004.

D. G. R. Bradski and A. Kaehler. *Learning OpenCV, 1st edition.* O'Reilly Media, Inc., 2008. ISBN 9780596516130.

L. Bretzner, I. Laptev, and T. Lindeberg. Hand gesture recognition using multi-scale colour features, hierarchical models and particle filtering. In *FGR '02: Proceedings of the Fifth IEEE International Conference on Automatic Face and Gesture Recognition*, page 423, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1602-5.

D. Brown, I. Craw, and J. Lewthwaite. A som based approach to skin detection with application in real time systems. In *in Proc. of the British Machine Vision Conference*, 2001.

M. Deriche and I. Naseem. A new approach to face localization in the hsv space using the gaussian model. pages 373–383, 2007.

T. B. Dinh, V. B. Dang, D. A. Duong, T. T. Nguyen, and D.-D. Le. Hand gesture classification using boosted cascade of classifiers. In *RIVF*, pages 139–144, 2006.

D. Douglas and T. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Canadian Cartographer*, 10:112–122, 1973.

Y. Fang, J. Cheng, K. Wang, and H. Lu. Hand gesture recognition using fast multi-scale analysis. In *ICIG '07: Proceedings of the Fourth International Conference on Image and Graphics*, pages 694–698, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2929-1. doi: http://dx.doi.org/10.1109/ICIG.2007.105.

J. Hershberger and J. Snoeyink. Speeding up the douglas-peucker line-simplification algorithm. In *Proc. 5th Intl. Symp. on Spatial Data Handling*, pages 134–143, 1992.

E.-J. Holden and R. Owens. Representing the finger-only topology for hand shape recognition. volume 12, pages 187–202, Warsaw, Poland, Poland, 2003. Polish Academy of Sciences.

R. Hsu, M. Abdel-Mottaleb, and A. Jain. Face detection in color images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24:696–706, 2002. ISSN 0162-8828.

M. Isard and A. Blake. Icondensation: Unifying low-level and high-level tracking in a stochastic framework. In *ECCV '98: Proceedings of the 5th European Conference on Computer Vision-Volume I*, pages 893–908, London, UK, 1998a. Springer-Verlag. ISBN 3-540-64569-1.

M. Isard and A. Blake. Condensation—conditional density propagation for visual tracking. *Int. J. Comput. Vision*, 29(1):5–28, 1998b. ISSN 0920-5691. doi: http://dx.doi.org/10.1023/A:1008078328650.

M. Jones, , M. J. Jones, and J. M. Rehg. Statistical color models with application to skin detection. In *International Journal of Computer Vision*, pages 274–280, 1999.

P. Kakumanu, S. Makrogiannis, and N. Bourbakis. A survey of skin-color modeling and detection methods. *Pattern Recogn.*, 40(3):1106–1122, 2007. ISSN 0031-3203.

R. E. Kalman. A new approach to linear fi ltering and prediction problems. volume 82, pages 35–45, 1960.

R. Khan, J. Stöttinger, and M. Kampel. An adaptive multiple model approach for fast content-based skin detection in on-line videos. In *AREA '08: Proceeding of the 1st ACM workshop on Analysis and retrieval of events/actions and workflows in video streams*, pages 89–96, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-318-1.

K. Kim, T. Chalidabhongse, D. Harwood, and L. Davis. Real-time foreground-background segmentation using codebook model. 11(3):172–185, June 2005.

R. Lockton and A. W. Fitzgibbon. Real-time gesture recognition using deterministic boosting. In *Proceedings, British Machine Vision Conference*, 2002.

S. Malik. Real-time hand tracking and finger tracking for interaction. *Paper for Vision Course*, 2003.

C. Manresa, J. Varona, R. Mas, and F. J. Perales. Real-time hand tracking and gesture recognition for human-computer interaction. volume 0, pages 1–7, 2000.

S. McKenna, S. Gong, and Y. Raja. Modelling facial colour and identity with gaussian mixtures. 31(12):1883–1892, December 1998.

R. O'Hagan and A. Zelinsky. Visual gesture interfaces for virtual environments. In *AUIC '00: Proceedings of the First Australasian User Interface Conference*, page 73, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0515-5.

E.-J. Ong and R. Bowden. A boosted classifier tree for hand shape detection. In *Automatic Face and Gesture Recognition, 2004. Proceedings. Sixth IEEE International Conference on*, pages 889–894, May 2004.

P. Peer, J. Kovac, and F. Solina. Human skin colour clustering for face detection. 2003.

T. Petrie. Tracking bouncing balls using kalman filters and condensation, unknown. `http://www.marcad.com/cs584/Tracking.html`.

S. Phung, A. Bouzerdoum, and D. Chai. A novel skin color model in ycbcr color space and its application to human face detection. pages I: 289–292, 2002.

D. Z. Ramani, R. Duraiswami, H. N, and L. S. Davis. Multimodal tracking for smart videoconferencing. In *In Proc. 2nd Int. Conference on Multimedia and Expo*, 2001.

R. Reilly and A. Hanson. Gesture recognition for augmentative human computer interaction. In *Engineering in Medicine and Biology Society, 1995., IEEE 17th Annual Conference*, volume 2, pages 1275–1276 vol.2, Sep 1995.

M. Shamsi, R. A. Zoroofi, C. Lucas, M. S. Hasanabadi, and M. R. Alsharif. Automatic facial skin segmentation based on em algorithm under varying illumination. *IEICE - Trans. Inf. Syst.*, E91-D(5):1543–1551, 2008. ISSN 0916-8532.

J. Sklansky. Finding the convex hull of a simple polygon. 1:79–83, 1982.

M. Spengler and B. Schiele. Towards robust multi-cue integration for visual tracking. 14(1):50–58, 2003.

T. Starner, A. Pentland, and J. Weaver. Real-time american sign language recognition using desk and wearable computer based video. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(12):1371–1375, 1998. ISSN 0162-8828.

M. Storring, T. Kocka, H. Andersen, and E. Granum. Tracking regions of human skin through illumination changes. 24(11):1715–1723, July 2003.

S. Suzuki and K. Abe. Topological structural analysis of digitized binary images by border following. 30(1):32–46, April 1985.

V. V. Vassili, V. Sazonov, and A. Andreeva. A survey on pixel-based skin color detection techniques. In *in Proc. Graphicon-2003*, pages 85–92, 2003.

J. P. Wachs, H. I. Stern, Y. Edan, M. Gillam, J. Handler, C. Feied, and M. Smith. A gesture-based tool for sterile browsing of radiology images. volume 15, pages 321–323, 2008.

T. Wilhelm, H. J. Böhme, and H. M. Gross. A multi-modal system for tracking and analyzing faces on a mobile robot. *Robotics and Autonomous Systems*, 48(1):31 – 40, 2004. ISSN 0921-8890. doi: DOI: 10.1016/j.robot.2004.05.004. European Conference on Mobile Robots (ECMR '03).

J. Yang, W. Lu, and A. Waibel. Skin-color modeling and adaptation. In *ACCV '98: Proceedings of the Third Asian Conference on Computer Vision-Volume II*, pages 687–694, London, UK, 1997. Springer-Verlag. ISBN 3-540-63931-4.

Q. Zhu, K. Cheng, C. Wu, and Y. Wu. Adaptive learning of an accurate skin-color model. pages 37–42, 2004.

Appendix A

On multiple hands tracking

On a stage of the project we considered multiple hands tracking as a possible feature for our application but we found that our was unable to track multiple hands. At first it was thought it could have been a bug or some configuration errors in the parameters of the tracker. However, due to the simple nature of the few configuration parameters of the CONDENSATION algorithm, a further revision suggested the reason must lie somewhere else.

Theoretically the CONDENSATION algorithm, by definition, should be able to track multiple objects as all objects detected in the scene will have its own density. The problem relies on the fact that when an object is firstly detected the samples density defines the location of this object (samples revolve around the object) and thus any new object that appears in scene at another location will not be detected. This suggests that for multimodal density to be possible the objects to be detected should already be on the image. However, the tracked objects need to be similar size and even if so the multimodal density will usually only be kept for a short period after which it would become unimodal (samples around one of the objects).

Let's assume an initial state consisting of a black background with two identical still white blocks placed at different locations. We initialise the tracker with 1000 particles per time step, an identity matrix as the dynamics matrix so that no movement prediction is taken and a standard deviation of 20 pixels for the stochastic diffusion of the particles chosen as a value to represent mean variation in hand movement. The weight of the particles depend on the number of white pixels under a window centred at each pixel in the image. The initial state of the tracker is a steady state with the particles scattered randomly across the window while no measurements are taken (figure A.1).

The next stage is to place the aforementioned two white squares at different locations. The tracker should assign higher weights to particles on both squares, effectively giving a multi-modal state density. As can be seen figure A.2, the experiment so far confirms this behaviour.

The red circle in the figure A.2 is drawn around the centre of the object estimated by the tracker. Considering the two blocks the theoretical estimated centre should be in the middle of the imaginary line that connects the centres of both squares. In the image this is not happening exactly due to the random factor of the algorithm. If we consider our initial steady state and our weight function based on the number of white pixels, it is highly improbable that the accumulated weight of the

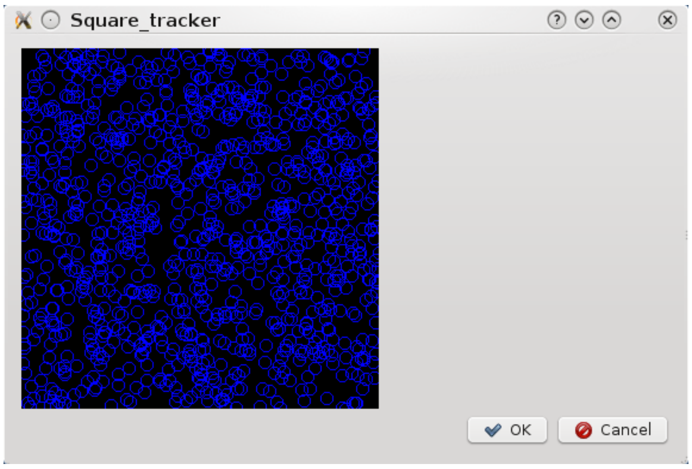


Figure A.1: Initial state with all samples scattered randomly.

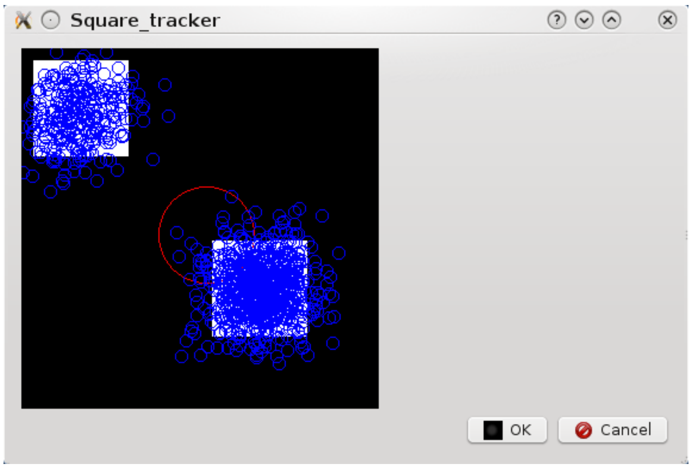


Figure A.2: Multimodal density. There are samples on both squares.

particles around both squares will be exactly the same, as well as the number of samples, which will not be exactly the same either. One square will have more samples than another, which explains why the red circle is biased towards a certain square. In fact, as time passed by the red circle fluctuated a bit towards one square or another, but in the end the samples would settle around one square, making the density unimodal.

In our case we believe this happens due to the stochastic nature of the algorithm and our window-based weight function. The diffusion part of the prediction stage of the Condensation causes the particles to be drifted randomly (figure A.3), sometimes just out of the square causing them to have lower weight, or far enough from the square so that their weight is close to null and hence discarded at the next iteration. The samples around the square with most weight will be more likely to be chosen and, at the same time, the fewer particles around the other square may be gradually drifted and discarded, resulting in the particles generated in the next iteration being around a certain square.

The ability of the Condensation algorithm to track multiple hypothesis at the same time was also tested in a realistic environment. We started by placing both hands together so as to concentrate

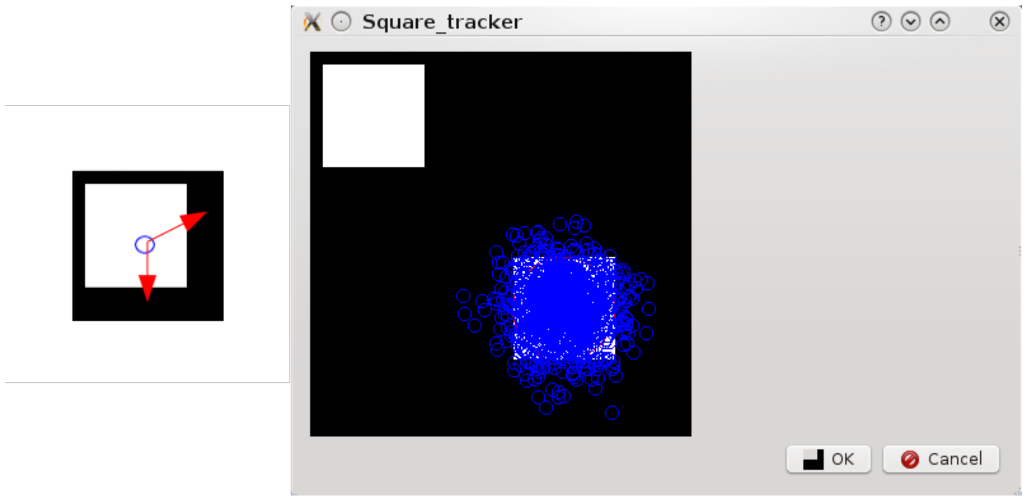


Figure A.3: Stochastic diffusion may cause the samples to be drifted outside of the square in some cases.

the density and afterwards the hands were separated in opposite directions, dividing the density. Figure A.4 shows that the algorithm can effectively track more than one hand at the same time for a small period, but inevitably the density on each hand will not be equal, thus the samples in the higher density area are more likely to be resampled more times.

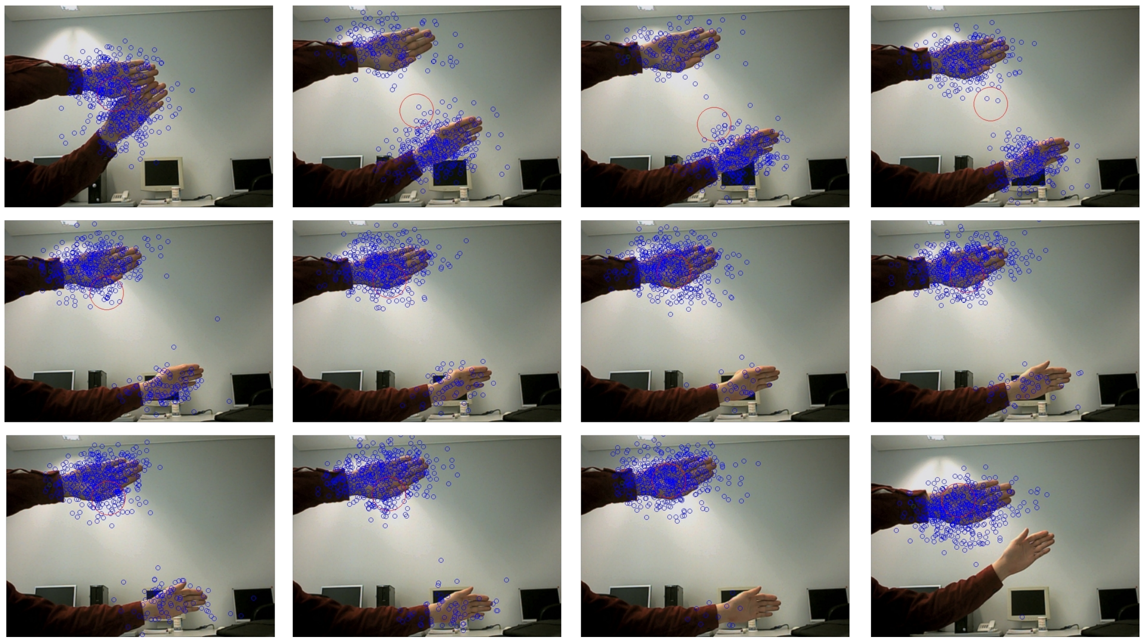


Figure A.4: Sequence of images of two hands separated by five time steps each. The test started by placing both hands together and slowly separating them apart. It can be seen that the particles will be divided in two clusters for a period but will eventually settle around one hand.

Multi-tracking with CONDENSATION

We have seen that even though initially the CONDENSATION algorithm manages to keep track of both squares, eventually it will gradually shift towards a certain square. The reason we stated was the drift that occurred to the particles, leading some of them to be discarded, mostly affecting the object with less particles around.

This statement suggests that if we are able to limit this drift it will be possible to maintain the particles as they will not be discarded for the next factored sampling. The previous set-up used 20 as the standard deviation. In order to test our hypothesis a new standard deviation of 2 was set.

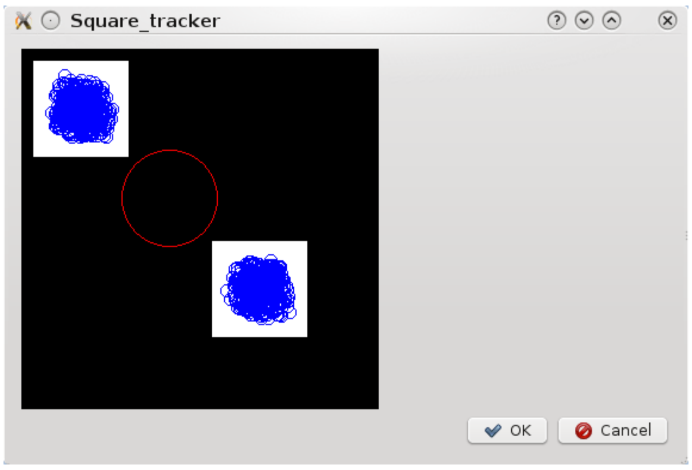


Figure A.5: Constant multimodal density was possible with a much smaller noise parameter

As figure A.5 shows, a smaller noise parameter effectively allows the tracker to follow both squares. All particles are well inside the squares, which implies they will all have the same weight given that all the pixels in their neighbourhood are white. Hence, no particle will be discarded for the next iteration. This situation, however, is unrealistic. In the case of our project, we would need balanced skin regions so that the previous approach would work, considering how we compute the weight of the particles. However, the small added noise is not adequate to real hand movement and the tracker would constantly lose track of the hand. In addition, we would still have the problem that the objects may not appear at the same time and if the samples distribution was set around a specific object the new objects would still be ignored.

In order to solve this problem, several approaches can be found in the current literature. Wilhelm et al. (2004) also used skin segmentation and dynamically added or removed new CONDENSATION trackers which covered different regions of the scene in order to detect different skin patches, together with a sonar which detected the presence of people, so as to increase the tracking precision. Spengler and Schiele (2003) used colour and intensity change cues to track multiple faces, as well as newly generating a 10% of the samples randomly and not from the prior density so that there will always be particles for detecting new objects. Other researchers have tracked the shape of the objects or used other mechanisms to assist the tracker. Isard and Blake (1998b), the original creators of the algorithm, updated it to combine low-level information (e.g. skin colour) and high-level information (e.g. hand contour), and generated sample positions from the detected important locations. Kumar et al.[2] on the other hand uses an adaptive multiple object tracking system by using colour intensity and segmentation cues. Their approach to the problem we have been discussing in this section is to maintain the weight of the pixels of a prior iteration for the next iteration so that they will not be discarded and evaluate the accuracy of the hypotheses.

In conclusion, we have seen that the CONDENSATION algorithm strictly applied with our chosen weight function based on the number of skin pixels is unable to track multiple hands. However, we have also seen that this problem has been successfully addressed in literature.

Appendix B

Test images for hand shape enhancement

Our training set consisted of a series of photographs different lighting conditions. In order to calculate our skin colour model the photographs were segmented manually from the background. In this appendix, the corresponding images are listed together with the mean H-S colour, the covariance matrix and the maximum and minimum values of these.

Notation used:

- μ. Mean colour [H S].
- Σ. Covariance matrix.
- max(X). Maximum value of X, where X is H or S
- min(X). Minimum value of X, where X is H or S

1. Daylight, in the shadow

$\mu = [0.081452 \quad 0.425547]$

$$\Sigma = \begin{bmatrix} 6.6954e-04 & -3.8606e-04 \\ -3.8606e-04 & 4.5605e-03 \end{bmatrix}$$

$max(H) = 0.16667$
$min(H) = 0$

$max(S) = 0.61047$
$min(S) = 0.069444$

2. At home, in the evening, fluorescent light

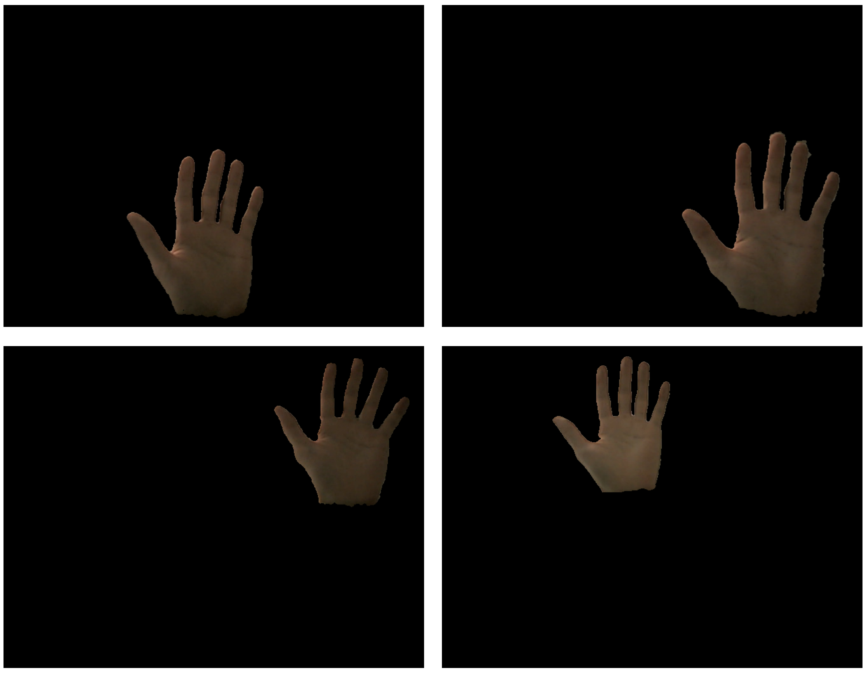
$\mu = [0.082877 \quad 0.329488]$

$$\Sigma = \begin{bmatrix} 9.3504e-03 & -7.3467e-04 \\ -7.3467e-04 & 8.8476e-03 \end{bmatrix}$$

$max(H) = 0.99775$
$min(H) = 0$

$max(S) = 0.87179$
$min(S) = 0.020408$

3. At the office, during the day, fluorescent light



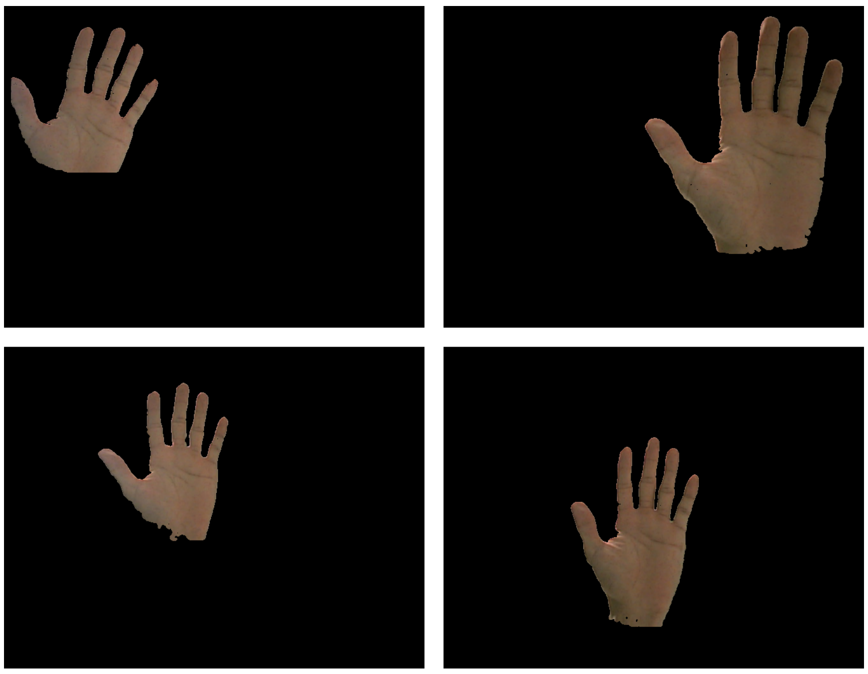
$\mu = [0.083700 \quad 0.453989]$

$$\Sigma = \begin{bmatrix} 1.8222e-04 & -5.3235e-05 \\ -5.3235e-05 & 3.2304e-03 \end{bmatrix}$$

$max(H) = 0.13333$
$min(H) = 0.036458$

$max(S) = 0.60526$
$min(S) = 0.17532$

4. At the office, in the evening, fluorescent light



$\mu = [0.075260 \quad 0.412146]$

$$\Sigma = \begin{bmatrix} 4.5435e-04 & 7.9546e-05 \\ 7.9546e-05 & 4.5221e-03 \end{bmatrix}$$

$max(H) = 0.99583$
$min(H) = 0$

$max(S) = 0.64567$
$min(S) = 0.18571$

4. All conditions combined

$\mu = [0.080303 \quad 0.410846]$

$$\Sigma = \begin{bmatrix} 2.0724e-03 & -2.1952e-04 \\ -2.1952e-04 & 6.7185e-03 \end{bmatrix}$$

$max(H) = 0.99775$
$min(H) = 0$

$max(S) = 0.87179$
$min(S) = 0.020408$

Appendix C

CONDENSATION - data for determining the parameters of the Dynamical Model

In order to determine the parameters of the dynamical model the movement of a real hand had to be analyzed. To mimic the situation the white light of a hand held lantern was segmented from a black background so as to minimize noise and simulate real hand movement.

The positions (2D) of the white light were then recorded for a short period (a total of 412 frames) and then used to calculate the velocity, acceleration and common statistics which are detailed in figure C.1. The standard deviations shown were used for the noise parameter in the dynamical model.

Figures C.2 and C.3 show the velocity and acceleration variation charts of the recorded sequence through time. They suggest sudden high accelerations are rare and short in time, thus it did not seem necessary to consider acceleration for our dynamical model, which would have slightly impacted efficiency.

	X	**Y**
Mean Velocity	35.29	19.36
Standard Deviation	28.83	20.66
Maximum Absolute Deviation	99.71	88.71
Minimum Absolute Deviation	0.29	0.29
Mean Acceleration	25.1	14.57
Standard Deviation	19.67	15.66

Figure C.1: Statistics for the recorded sequence.

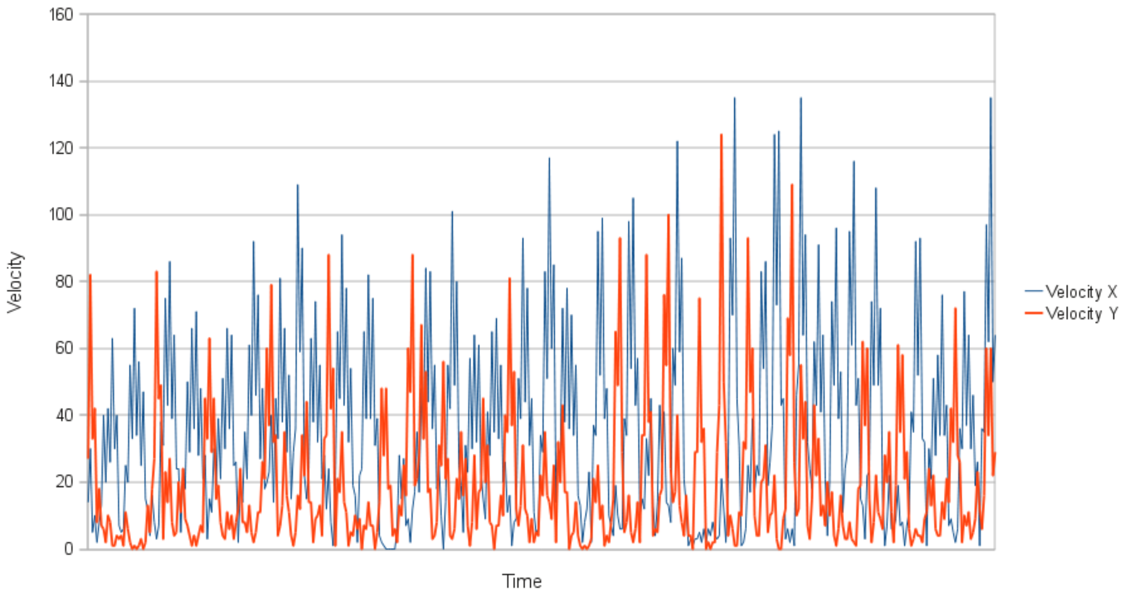


Figure C.2: Velocity chart. The spikes in the chart indicate that high variations in speed were very short in time.

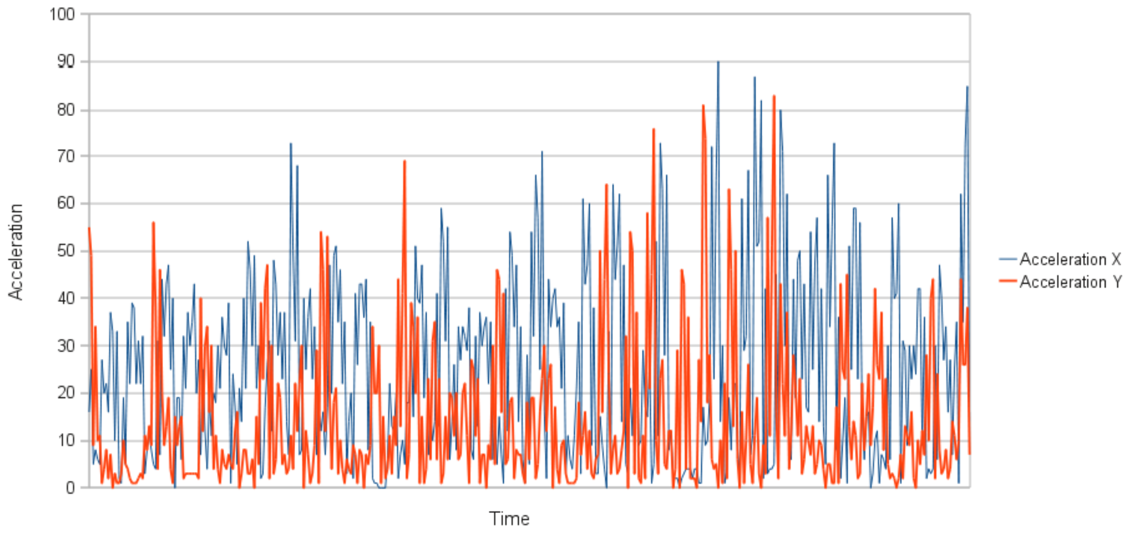


Figure C.3: Acceleration chart. High accelerations were short and not very often.