Human-Computer Interaction Using Robust Realtime Gesture Recognition

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

Author:
Matthias Endler

Reviewers:
Prof. Dr. Michael Guthe
Prof. Dr. Dominik Henrich

September 2013

Abstract

This work deals with the development of a Human-Computer interface using hand gesture recognition. As a natural form of human interaction, gestures allow us to express many ideas, actions, and intents in a straightforward way – without the need for long training periods. In this work we try to *enhance* the accessibility of graphical user interfaces with fast and simple hand gestures, instead of trying to replace well-established devices like the mouse and the keyboard.

In contrast to many similar systems we focus on robustness and realtime capability and use a single 2D camera for video input. We implement a prototype which runs as a background process on commodity hardware. Our system can control the mouse and trigger keyboard events.

In order to improve the robustness, we use multiple image filters, also called detectors, at the same time. We distinguish between position and feature detectors. The first group detects the hand position and consists of a Haar, a CAMshift and a Shape detector. The second group detects hand features like the hand posture and the finger positions. We use a skin filter for this task.

All detectors implement the same interface to facilitate this approach. They receive the same input data, which greatly simplifies the process of adding additional components in the future.

Using an innovative bootstrapping mechanism, we can start all detectors without manual intervention. This automatic initialisation is ordinarily completed after only a few camera frames.

Based on prior tests, we implement multiple heuristics for each detector. This allows us to give a confidence value for each detector result and indicate error conditions. We combine the results of all detectors into a final estimate of the hand state based on these confidence values. This way, the system is more robust against the shortcomings of single detectors.

Two use-cases demonstrate the viability of the system. First we control a window manager on behalf of the user. Furthermore we use our prototype to play computer games.

We evaluate both, the robustness and the real-time capability of the system, using test video sequences and statistical methods. Based on our evaluations, 90% of all gestures are correctly recognized. The system runs with a worst-case frame rate of more than 45 fps and is therefore realtime-capable; even on slow hardware. The solution works on complex backgrounds and with difficult lighting conditions.

Many more use-cases, like controlling audio or video applications, are conceivable. For this purpose, new gestures can be added with ease using the scripting language Python.

Our prototype is available as an open source application.

Zusammenfassung

Diese Arbeit beschäftigt sich mit der Entwicklung einer Mensch-Maschine Schnittstelle mithilfe von Handgesten-Erkennung. Als natürliche Form menschlicher Interaktion erlauben es uns Gesten, viele Ideen, Aktionen und Absichten einfach und unkompliziert auszudrücken, ohne dafür lange üben zu müssen. Diese Arbeit stellt den Versuch dar, den Umgang mit grafischen Benutzeroberflächen zu verbessern, ohne bereits etablierte Eingabegeräte wie Maus und Tastatur ersetzen zu wollen.

Im Gegensatz zu ähnlichen Ansätzen konzentrieren wir uns auf Stabilität und Echtzeitfähigkeit des Systems unter Verwendung einer einzelnen 2D Kamera für Videoeingaben. Wir implementieren einen Prototypen, welcher als Hintergrundprozess auf Standard-Hardware ausgeführt wird. Unser Programm kann die Maus steuern und Tastatureingaben auslösen.

Um die Stabilität zu steigern, benutzen wir mehrere Bildfilter – auch Detektoren genannt – gleichzeitig. Wir unterscheiden zwischen Positions- und Merkmaldetektoren. Die erste Gruppe erkennt die Handposition und besteht aus einem Haar, einem CAMshift und einem Shape Detektor. Die zweite Gruppe erkennt Merkmale wie die Handhaltung und die Position der einzelnen Finger. Hierfür verwenden wir einen Filter für Hautfarben, im Weiteren Skin Detektor genannt.

Um dieses Vorgehen zu ermöglichen, verwenden wir eine einheitliche Schnittstelle für alle Detektoren. Jeder Detektor erhält dieselben Eingabedaten, wodurch das spätere Hinzufügen zusätzlicher Komponenten stark vereinfacht wird.

Die Verwendung eines innovativen Startvorgangs macht es möglich, alle Detektoren ohne Zutun des Nutzers zu starten. Diese automatische Initialisierung ist in der Regel nach der Analyse weniger Kamerabilder abgeschlossen.

Auf Basis vorangehender Tests implementieren wir mehrere Heuristiken für jeden Detektor. Dies erlaubt uns, einen Konfidenzwert für jedes Ergebnis eine Detektors anzugeben und Fehlerzustände anzuzeigen. Basierend auf diesen Konfidenzwerten kombinieren wir die Ergebnisse aller Detektoren zu einer abschließenden Einschätzung über den Zustand der Hand. Dadurch ist das System weniger anfällig für die Defizite einzelner Detektoren.

Zwei Anwendungsfälle demonstrieren die Praxistauglichkeit unseres Systems. Im ersten Fall kontrollieren wir einen Window-Manager stellvertretend für den Benutzer. Desweiteren verwenden wir unseren Prototypen, um Computerspiele zu steuern.

Wir evaluieren sowohl die Stabilität als auch die Echtzeitfähigkeit des Systems mithilfe von Testaufnahmen und statistischen Methoden. Anhand unserer Untersuchungen werden 90% aller Gesten korrekt erkannt. Das System arbeitet mit einer Framerate von mindestens 45 fps, wodurch wir sogar bei schwacher Hardware die Echtzeitfähigkeit garantieren können. Die Lösung kommt sowohl mit komplexen Hintergründen als auch schwierigen Lichtverhältnissen zurecht.

Viele weitere Anwendungsfälle, wie etwa das Steuern von Audio- und Videoanwendungen, sind denkbar. Hierfür können neue Gesten ohne großen Aufwand mit der Skriptsprache Python hinzugefügt werden.

Unser Prototyp ist als Open-Source Lösung frei zugänglich.

Acknowledgements

For his constructive suggestions, for the trust he has shown in my person and work and for his encouragement I would like to express my gratitude to Prof. Michael Guthe, my reviewer.

I would like to thank Prof. Dominik Henrich who agreed to be my second reviewer and who supported me throughout my studies in his role as an academic advisor.

My deepest appreciation goes to Dr. Oleg Lobachev for being a diligent advisor while preparing and writing my thesis, a steady source of inspiration, and a friend. He helped keeping my progress on schedule and taught me new ways to think about our craft.

I would like to acknowledge the support provided by my girlfriend and my family during the preparation of this thesis.

Contents

Abstract **ii**

Zusammenfassung **iv**

Acknowledgements **vi**

Contents **vii**

1 Introduction **1**
1.1 Object detection as a field of research 1
1.2 Viability analysis 2
1.3 Project goals 2
1.4 Historical developments 5
1.5 Outline 8

2 Related work **9**
2.1 Surveys 9
2.2 Special devices 9
2.3 Depth information 10
2.4 2D images 10

3 System overview **13**
3.1 Use-cases 13
3.2 Difficulties 16
3.3 Taxonomy 17
3.4 High level architecture 19
3.5 Detector Framework 20
3.6 Heuristics 22

4 Available detectors **24**
4.1 Viola–Jones 25
4.2 Camshift 34
4.3 Shape 45
4.4 Skin detector 54
4.5 Summary 62

4.6 Bootstrapping . . . 65
4.7 Consolidation of detector results . . . 67
4.8 Enabling and disabling detectors . . . 69
4.9 Summary . . . 70

5 Gestures & Actions **71**
5.1 Architecture . . . 71
5.2 Gestures . . . 72
5.3 Actions . . . 74
5.4 Review . . . 75

6 Evaluation **76**
6.1 Robustness . . . 76
6.2 Real-time capability . . . 81
6.3 Use-cases . . . 82

7 Conclusions **87**
7.1 Achievements . . . 87
7.2 Future work . . . 88

List of Figures **90**

List of Tables **94**

A Supporting material **95**

B Keyboard shortcuts **96**

C Code **97**
C.1 Metrics . . . 97
C.2 Camshift ellipse . . . 99
C.3 Position estimation . . . 100
C.4 Hierarchy of an OpenCV Haar cascade . . . 103

Glossary **105**

Bibliography **107**

Chapter 1

Introduction

While cheap commodity hardware is readily available, its use for interactive human-computer interaction is still a novelty – especially purely image-based solutions. Is such an approach even feasible for robust real-time gesture recognition?

With the advent of portable computers and smartphones, the means to record video material became ubiquitous. Most modern devices have built-in cameras which are mostly used for video-chat and taking still images[1]. Throughout this work we will consider the question of whether this commodity hardware, i.e., a 2D webcam and a standard processor, can also be used for robust realtime gesture recognition.

1.1 Object detection as a field of research

One reason why the broad dissemination of perceptual human-computer interfaces is still a distant goal might be, that realtime image processing is a very active field of research and robust feature detection (e.g. for faces or hands) in images is considered a tough problem – let alone motion and gesture recognition.

In order to be useful, the applied techniques need to be reliable in most real-life scenarios. A tool for human-computer interaction which is only working under special circumstances is worthless at best and harmful at worst, since incorrect input detection leads to undefined behavior, which irritates the users.

Therefore, practical use of feature detection is currently limited to a very narrow problem space. Smile detection, for instance, is one of the newer features of consumer cameras. The idea is to wait for a good moment to take a photo – right when everybody is smiling – and press the button automatically. Reviews have shown, that this gimmick is not always working as proposed, especially in

[1] See Do, Blom, and Gatica-Perez [2011] for a recent survey on smartphone usage.

real-world situations due to varying lighting conditions or partly hidden faces (e.g., people wearing sunglasses or hats) [Whitehill et al., 2009].

1.2 Viability analysis

Robust, gesture based device control enables a plethora of possibilities to simplify our everyday life. After all, gestures are an important natural form of human expression. In every moment we convey our feelings with gestures and poses – most of the time subconsciously. For deaf people, it is the primary way of communication. Gestures are a natural part of human interaction, many of which don't require any learning and are universally understood. Using this intrinsic form of communication for human-machine interaction, one could intuitively control lights and music players with a swipe of the hand. One could move windows from one computer screen to another or play games without a controller. More practically thinking, this technology enables handicapped people to use a computer more naturally. Medical technical personnel could use a contact-free diagnostic computer while performing surgery without increasing the risk of infections.

1.3 Project goals

The initial set of requirements for our prototype is quite demanding. We aim for a realtime capable system that is very robust in terms of everyday usage. The system should work out of the box, without the need for long calibration steps. It should be platform independent and mostly unobtrusive. If any configuration is needed, it shall be optional. Our goal is to combine multiple gesture detection methods in a stable pipeline to achieve robustness and realtime capability for the detection of user's hand movement from webcam-captured live video. We measure the success of our endeavor based on primary and secondary goals.

1.3.1 Primary goals

The project needs to fulfill the following requirements to be of any practical relevance. These would be realtime-capability, robustness, and readiness for real-world use. We will make a quantitative analysis for each primary goal.

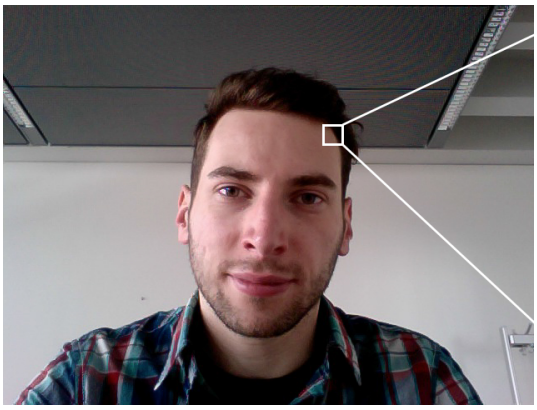
Realtime-capability Calculations on image data are expensive. Modern cameras provide hundreds of thousands or even millions of pixels per frame. Given

an average framerate of 24 fps, a calculation time of about 40ms sets a challenging boundary. Nevertheless, it is a major project goal to present a realtime solution, since ever so slight delays during the interaction with the device would dramatically affect user experience.

Therefore, the underlying algorithms have to be optimized in order to be applicable for realtime usage. Since the system will be permanently running in the background, it may not be resource intensive. Only a minimal amount of computing power and memory may be used by the system at any time.

We will use profiling techniques to identify bottlenecks and evaluate the performance under real-life conditions.

Robustness One could easily underestimate the complexity involved in object recognition. It took millions of years of evolution to create a neuronal network which is capable of detecting minor differences of facial expressions and can interprete gestures almost instantly [D. G. Lowe, 1999]. To expect, that it should only take a few years for machines to catch up, would be unreasonable. As of now, human perception does have little in common with its artificial counterpart (see Figure 1.1).



(a) *Perception of a human being*

```
09 0A 06 07 01 00 03 01 01 00 00 00 00 00 00
02 07 00 00 00 00 00 00 00 00 01 02 03 04 11
01 02 02 07 06 05 05 00 00 00 00 00 00 00 00
52 06 81 C1 E1 22 23 FF DA 00 0C 03 01 00 02
F1 5B AC 4D 65 B2 A5 B8 D2 C3 79 96 F4 71 7D
CA 95 A9 71 D2 BB 84 A8 AC 79 5F 6D 90 24 F9
6B 09 21 41 29 49 E2 28 69 86 27 46 D7 73 52
2F 3E D1 64 70 EC 87 36 5E BF 1C 70 43 00 51
5C 25 DF 24 CE 76 32 A3 C7 8D 91 51 BB 4A 19
1A C6 86 B9 72 27 86 92 92 A7 E1 C9 59 B6 80
3F A9 73 46 F4 17 40 A7 C8 74 7D C7 43 3A 63
6F DD B3 6E 8A AE 6C D6 EB 4A 57 2D 7A E1 95
8B 5D 1E 91 5F 85 A6 73 A1 48 53 60 66 6B D3
71 55 BA 33 FB CE 06 93 3B 67 EB D1 75 59 D0
39 D4 8E 1F 41 8E 18 5A 27 B0 82 31 07 84 DF
00 6D 76 1D C1 AE A5 EF 5A 5F B6 17 4B C1 D0
41 DD B1 E6 4B C6 95 62 6A F1 2D 49 1F FF D9
```

(b) *Actual numerical representation as seen by the computer*

Figure 1.1 – *Human perception and machine processing of image data. Inspired by OpenCV Development Team [2013b]*

That said, image recognition already works surprisingly well – given some carefully defined constraints. Face recognition systems currently reach a detection rate of over 90% if a person is directly facing the camera [Paul Viola and M. J. Jones, 2004] and key facial features like eyes are not hidden to the camera (e.g., by wearing sunglasses). But even marginal modifications – like tilting the head or turning the head more than 25 degrees to the side – might create major difficulties and drop the recognition rate [see Chellappa, Sinha, and Phillips, 2010, p. 49].

Occlusion is another difficult scenario: If objects are partially hidden or shadows obstruct the sight, correct classification gets harder and often times requires heuristics to attain an acceptable result.

Especially when it comes to human-computer interaction, the system must work reliably. Any unpredictable software behavior becomes an inconvenience for the user [Sears and Jacko, 2007, p. 190]. It takes more time to remedy incorrect actions caused by the framework than it would help to save in the first place.

As a consequence, the number of false positives (gestures which were not intended by the user) needs to be kept to a minimum. It is better to omit a correct gesture (a false negative) than to execute an incorrect one (a false positive).

Taking all of the above points into account, we will focus on an approach which considers a wide range of image cues to detect a hand. We will create a final, weighted estimate based on all of these cues. This step will make the system more robust against outliers, e.g., a hand partially hidden by an object.

Rigorous testing in real-world situations We evaluate both the robustness and the real-time capability of our implementation.

To investigate the robustness of our system, we record video data and assign a "difficulty grade" to each video feed. Afterwards we take these recordings as an input to our program and observe the detection results; further, we regard the per-frame confidence values of all filters which means the detectors receive a stream of typical input data and we measure the certainty of a correct hand recognition of each detector. We provide statistical analysis of our system in Chapter Evaluation.

Additionally we liken our results to other similar projects as far as the measurements are comparable.

1.3.2 Secondary goals

The second category contains goals, which add value to the system but are not strictly necessary for a minimum viable product. We will only make a qualitative evaluation of meeting these objectives.

Immediate usability The system shall be working out of the box, without the need for long calibration steps. We want to create a non-invasive input-technique which is suitable for everyday-use. It shall be platform independent and be mostly unobtrusive, which means it should work on consumer laptops and computers equipped with a standard webcam. If any configuration is needed, it shall be optional. This requires, that the system adapts to the user during runtime and adjusts its parameters to yield valuable detection results

even if the lighting conditions change over time or the hand is partially hidden during a certain amount of time.

Furthermore, keyboard shortcuts need to be provided to enable or disable certain functionalities of the system (like the execution of commands on behalf of the user) at runtime. This way, the system can run in the background and be mostly unobtrusive.

Extensibility The final application shall be flexible and expandable. All components, especially the algorithms inside the system, need to be easily exchangeable. This gets achieved by using a modular plugin system for the detection algorithms written in a high-level scripting language – namely *Python*. New algorithms and modules only need to comply to an interface in order to be integrated. The hand detector – a core part of the software – can be used as a framework for other applications.

Only critical parts of the software (also known as *hot paths*) need to be written in a compiled language (like C or C++), if more performance is required. In addition, we make use of the highly optimized object recognition library: *OpenCV*.

1.4 Historical developments

From sophisticated special-purposes devices for hand and gesture recognition to purely image-based systems which solely rely on algorithms, there are a number of vastly different approaches to consider when we try to build a custom framework to solve the hand detection problem.

All of the different approaches have their specific strengths and weaknesses which need to be considered. The following paragraphs give a short overview of the historical developments of human gesture recognition. We will mostly base our summary on the use for virtual reality applications (videogames, above all) and interfaces for the control of desktop environments. This decision has its roots in the observation that gesture recognition was substantially driven by these two related fields of research.

A comparison with other gesture interfaces, which use technical specifications that resemble our own, can be found in a separate chapter, *Related Work*.

1.4.1 Special hardware

Previous research has tackled the problem of gesture recognition by using special hardware. For instance, a combination of markers and sensors is often used to track the hand position.

One of the earliest devices using gesture recognition for Human-Computer interaction was the *DataGlove* invented by *VPL Research Inc.*, the predecessor of the *Nintendo©PowerGlove*, which was used as a video controller for the Nintendo Entertainment System (see Figure 1.2).

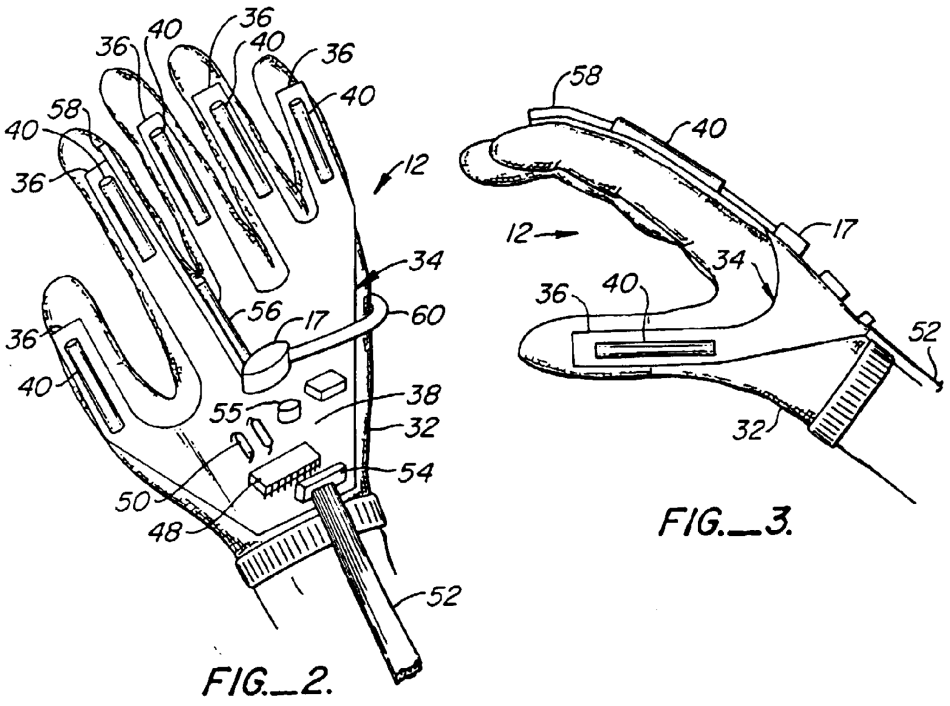


Figure 1.2 – *A figure from the patent specification of the VPL* DataGlove. *Source: Vpl [1996]*

It features a set of two ultrasonic speakers inside the glove as transmitters and three ultrasonic microphones, mounted on the television, as receivers [2]. The device has often been criticized for its imprecise controls and was no commercial success [Wikipedia, 2013e].

Erol et al. [2007] provide a timely literature review on these systems. Others have used controllers – one popular choice being Nintendo's Wii controller – for gesture recognition (e.g., Schlömer, Poppinga, Henze, and Boll [2008]).

Of course, such devices entail an additional hurdle for potential users. The artifical boundary – using a glove to control a computer – can easily result in scepticism and aversion. But until very recently it was a necessary tradeoff between reliability and price for a gesture processing environment. From that perspective it can be seen as pioneering work.

[2] For an interesting in-depth look at the *PowerGlove*, see Pausch [1991]

1.4.2 Gesture recognition based on image data

Research in pure vision-based gesture recognition dates back to the late 1980s and early 1990s (e.g., F. K. Quek [1994]), although realtime-usage was still out of sight due to insufficient processing power.

With the ever increasing computing power of household devices, an exclusively visual approach became viable. No sensors or markers are needed with this technology. Instead, a set of algorithms operates on raw image and video data and infers object positions and movements.

Microsoft's Kinect camera was one of the first consumer devices to feature a marker-free gesture recognition system. Additionally to a 2D video stream, depth information was also analyzed simultaneously [Zhang, 2012]. This way, it's easier to separate the foreground (typically including the person we want to track) from the background; and, more specifically, to recognize hand gestures directly in front of the body as well as forward and backward movements (compare with Figure 1.3).[3]

In order to utilize the massive potential of aforementioned cameras inside cellphones or laptops for human-computer interaction, the next logical step is to find a robust way of interacting with a device in realtime using visual input from built-in cameras without any additional information like depth. This approach is non-intrusive and comparably inexpensive [see Zabulis, Baltzakis, and Argyros, 2009].

[3] It is not uncommon, that the game industry is responsible for pushing the boundaries of human-computer interaction. Natural, playful interaction with an entertainment device contributes to a rich user experience and opens up new market possibilities. Also, marginal inconsistencies between the detected and the actual movement can often be neglected in a gaming environment.

Figure 1.3 – *Extracting a hand shape using depth information with* Microsoft Kinect. *Source: Biswas and Basu [2011]*

1.5 Outline

We introduce a system, which is able to track hand and finger motions under difficult lightning conditions and interact with a graphical user interface on the users behalf. We use a performant, robust cascade of state-of-the-art image filters, together with a database of heuristics and rulesets to create a realtime gesture based environment.

Chapter 2

Related work

Over the past few years a substantial amount of work has been done in the field of gesture recognition. Although all contributions share the the same goal – a native human-computer interface – the strategies to achieve it vary tremendously. We provide a review of recent work and, if possible, compare the results with our own system in a later chapter.

2.1 Surveys

Rautaray and Agrawal [Rautaray and Agrawal, 2012] present a recent survey of hand recognition and tracking methods. Most of the methods regarded here were first implemented for *face* detection and tracking, we refer to Hjelmås and Low [Hjelmås and Low, 2001] for a survey of traditional methods. A recent comparison of binary features was presented by Heinly et al. [Heinly, Dunn, and Frahm, 2012].

2.2 Special devices

In a further work with a quite different from our approach, Bedregal, Costa, and Dimuro [2006] [Bedregal, Costa, and Dimuro, 2006] used a fuzzy rule-based method for Brazilian sign language (Língua Brasileira de Sinais – Brazilian sign language). They use a dataglove for hand recognition. They classify their hand configurations based on a set of angles of finger joints. Although they claim promising results, the success rate was not disclosed.

Schlömer, Poppinga, Henze, and Boll [2008] use the Nintendo Wii controller for gesture recognition. They use a combination of K-Means, Hidden-Markov-Model and a Bayes classifier to recognize five distinct gestures as shown in Figure 2.1.

During their evaluation with six participants they achieve an average recognition rate of 90%.

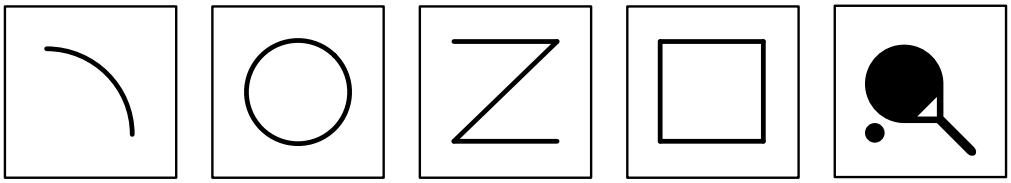


Figure 2.1 – *The five gestures implemented by Schlömer, Poppinga, Henze, and Boll. The last symbol stands for the gesture of serving a tennis ball*

2.3 Depth information

Ren, Yuan, and Zhang use the Microsoft Kinect depth camera for hand segmentation and gesture recognition [Ren, Yuan, and Zhang, 2011]. They use the depth information to segment the foreground (the hand) from the background. Then, for each hand gesture, they compare the histogram of a reference image with the image that is currently seen by the camera with a custom metric called *Finger-Earth Mover's Distance*. This scale-invariant procedure measures the difference between two histograms by calculating the *work* (for a specified definition of *work*) that would be required to transform one histogram into the other. With that approach they can identify the number of fingers shown even if they are very close together or the hand is rotated. During an experiment the authors defined 10 (unspecified) hand poses and let each of them perform by 10 subjects. They achieve a detection rate of 93.9% with a runtime of 4.0012s per frame. Thus, the system is not realtime capable. Furthermore it requires the user to wear a black belt around the wrist.

Wen, Hu, Yu, and Wang also use the Kinect infrared sensor for additional depth information to recognize gestures [Wen, Hu, Yu, and Wang, 2012]. They use the LAB color space for skin segmentation. The result are *candidate regions* – coherent contours which might be hand objects. They separate the hand from other skin-colored objects by selecting the biggest of these contours. After that they compute the convex hull around the hand object to determine the number of fingers. They achieved a success rate of 95% for a rock-paper-scissors game where rock was represented as "zero fingers", paper can be represented as "five fingers" and scissor with "two fingers".

2.4 2D images

Ong and Bowden detect and classify a set of hand shapes using K-Mediod (a variety of K-Means clustering) and a cascade of shape detectors [Ong and Bowden, 2004]. The cascade was boosted with FloatBoost, a variant of AdaBoost. They

achieve a cumulative success rate of 97,2%. It is important to note that they achieve this rate only on simple, non-skin colored backgrounds, which is a strong limitation for practical use.

Agarwal, Desai, and Saha use a combination of Haar, CAMshift and a curve fitting method to implement four *swipe* gestures as seen in Figure 2.2. They achieve a recognition accuracy between 89.3 and 95.5% (depending on the gesture) [Agarwal, Desai, and Saha, 2012].

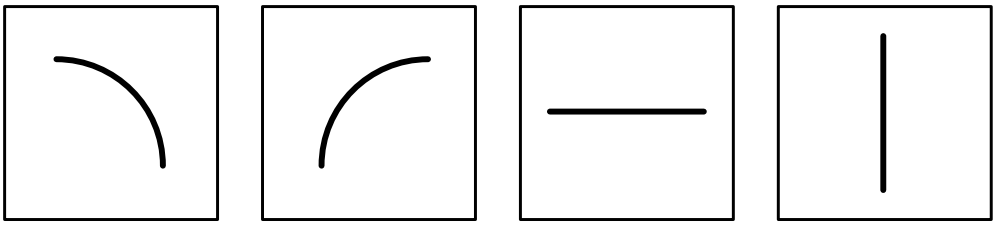


Figure 2.2 – *The slide gestures implemented by Agarwal, Desai, and Saha*

Stergiopoulou and Papamarkos firstly segment the hand by skin color filtering in YUV color space, secondly use a special kind of a neural network for palm morphology, thirdly use finger features to identify raised fingers, and finally recognize the gesture with a likelihood classification [Stergiopoulou and Papamarkos, 2009]. They achieve a success rate of around 90%. We agree with them (and the general state of the art) on the segmentation of the hand but use a different approach for steps two and three.

Stenger, Woodley, and Cipolla [2010] uses "multiple cues, appearance, color and motion" for a vision based remote control of computers and television sets [Stenger, Woodley, and Cipolla, 2010]. The project is based on previous work by a cooperation between Toshiba Research Europe and the University of Cambridge [Stenger, Woodley, Kim, et al., 2008]. They combine a set of well-known image filters and evaluate their suitability as hand detectors. A single gesture was implemented to test the robustness of the system. The user can control the mouse with his fist (see Figure 2.3). For their tests they use a selection of 12 videos, which show the gesture at various speeds. They define two individual metrics: precision and robustness. The former measures the distance between the detected and the actual hand position, the latter indicates how often a detector loses track of the hand and has to be reinitialized. With a combination of template matching and color segmentation (NCC-C), they reach a robustness of 99.7% and a precision of 89.2% at 30 frames per second.

Figure 2.3 – *The gesture interface presented by Toshiba at the Internationale Funkausstellung in Berlin 2008. Source: Stenger, Woodley, and Cipolla [2010]*

Chapter 3

System overview

Starting from typical use-cases, we create the requirements for a natural human-computer interface based on unaltered image data. We consider the difficulties which might arise during the development. At the end of this chapter we create a taxonomy for gesture-based machine commands and classify our own approach based on our findings.

3.1 Use-cases

3.1.1 Productivity tool

Usage scenario Even after more than forty years of research in *Graphical User-Interfaces* (see Lineback [2013] for a historical review of desktop environments), some tasks still feel cumbersome when working with a typical desktop environment. One of them is window movement and selection. Using a mouse pointer to click on the top area of a window can be tedious. Sometimes it happens that windows get hidden behind each other which makes even *finding* them hard in the first place. On the other hand, memorizing keyboard shortcuts to switch between two windows or two applications requires an effort to learn these shortcuts by heart to be efficient. A more natural way to interact with these windows would be to use simple gestures which symbolize commands like *show all windows* or *move this window*.

One target environment are office computers. In this case the user will be seated in front of a computing system, at a distance d of about 40cm to 1m from the webcam as in Figure 3.1. Another target platform are mobile computers such as laptops or tablets where the distance can vary from a few centimeters to several meters when controlling the device remotely.

Advantages With these gestures one could act quickly without memorizing keyboard shortcuts. It would require little overhead to use the application since

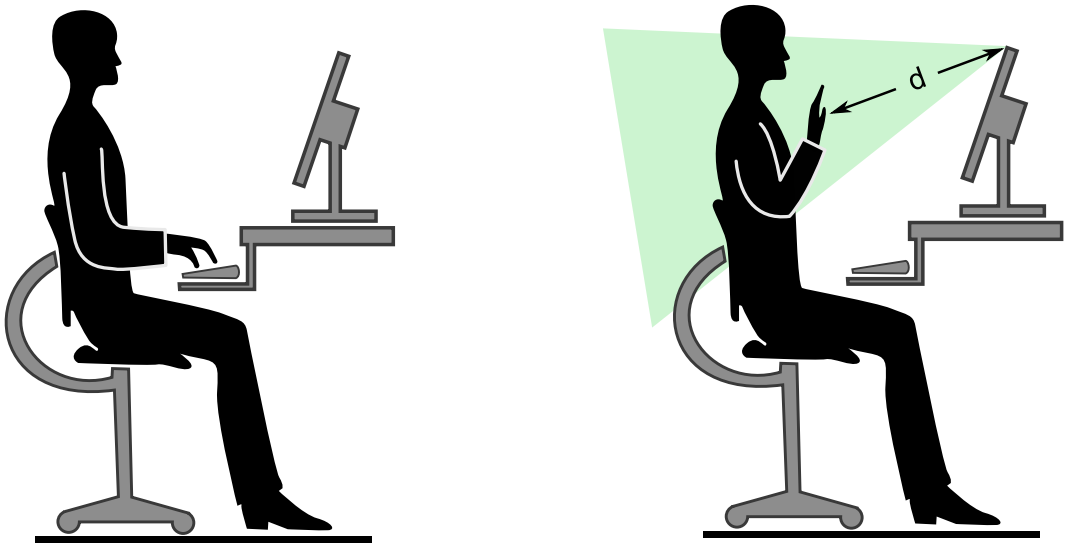


Figure 3.1 – *A typical use-case for the application. The green area symbolizes the camera's field of view. Adapted from [Loong, 2013]*

the system will be working mostly as a background process throughout the day, waiting for occasional commands from the user.

Difficulties Since the system is idle for most of the time but running constantly in the background, there is a high risk that the environment conditions change over time. One difficulty could be the varying lighting conditions during day and night use. The software can be used on mobile devices, so it needs to cope with changing backgrounds which can be arbitrarily complex and can show skin-toned objects like other body parts (e.g., faces). It can happen that the user of the computer changes throughout the day. The system must be able to adjust to all of these new conditions. The detection also needs to work with outdated hardware like cameras with low resolutions or slow CPUs. All of this needs to be considered throughout development.

Precautions We might need a calibration process which ensures that system is correctly detecting the gestures. If this is not possible (e.g., because the camera is not properly calibrated), we need to prevent unwanted actions.

Input and output We use unmodified live 2D input from a webcam. The system can work with any camera resolution greater or equal to $320\text{px} \times 240\text{px}$. Many internal and external cameras are supported. For an exhausting list, see [Willow Garage, 2011] We will execute key commands on behalf of the user (for instance, to show all open windows; a function we refer to as exposé), so the system needs a keyboard interface. In this usage example we will mostly focus on

static commands which trigger shortcuts. Nevertheless, we also want to control the mouse pointer, which requires a respective driver.

3.1.2 Gaming device

Usage scenario As we have already seen in the introductory chapter, computer games were often times a driving factor behind the development of new gaming devices. From joysticks and gamepads to hands-free control, games provide an ideal context for thriving innovation and experiments. We test our system as a control device for modern and classical computer games.

Advantages As with new technology in general, the novelty of new control interfaces enable many possibilities for innovative game design. The application seems to be fitting for a wide range of scenarios. One could think of simple, intuitive games for children, which require no prior training or sophisticated strategy games where the user controls his game units with complex hand gestures.

Difficulties Players are sensible to mishaps of a gaming device. Where precise inputs make the difference between victory and defeat, errors on the side of the device will not be forgiven by the user. Using our gesture interface as a gaming device mainly faces the same difficulties as the as the previous use-case. Additionally, we need to capture fast hand movements and execute keyboard and mouse actions instantly.

Precautions We need to optimize both, our detector and our interface to the operating system, for speed. We need straightforward to a way to out filter unwanted mouse movements. We test our assumptions on simpler games where fuzzy input is acceptable to the user.

Input and output We will use the same input as above, but we focus mainly on mouse output for this use-case. Simple keyboard shortcuts shall be easy to implement.

3.1.3 Required hand data

In order to make both of the above use-cases possible, we need access to the information in Table 3.1. The data will be stored as a *hand* object for easy access and used by the gesture recognition module for further analysis. We these hand properties we can fully recognize the required input for keyboard and mouse control.

Hand object	
Hand position	The coordinates of a rectangle surrounding the hand
Fingers	The number of fingers shown as well as the position of each finger (given as a line segment)
Hand shape	The outer hand contour and the hand area

Table 3.1 – *The hand data we want to collect*

3.2 Difficulties

All of these obstacles can lead to incorrect detections or interrupt the workflow and thus need to be taken into account while developing the software. We will address each issue next to our proposed solution in Table 3.2.

Category	Difficulty	Remedy
Software	Changing illumination	Make detectors agnostic to light changes
	Complex backgrounds	Test detector with various backgrounds and adjust settings accordingly
	Distracting body parts	Use heuristics to detect misbehavior
	Fast hand movements	Optimize detectors for performance
	Changing users at runtime	Constantly calibrate the system
	Fast command execution	Use native libraries for hardware control
Hardware	Processing power	Aim for low resource usage
	Low camera resolution	Optimize detection algorithms or use heuristics
	Motion blur	Implement at leas one detector which is robust against motion blur

Category	Difficulty	Remedy
User	Bad camera positioning	Prevent unintentional command execution and warn user

Table 3.2 – *Issues which need to be considered during development*

3.3 Taxonomy

Pavlovic, R. Sharma, and Huang suggest a taxonomy for hand gestures based on the study of many human-computer interfaces [Pavlovic, R. Sharma, and Huang, 1997]. We use this taxonomy to categorize our gesture system (compare with Figure 3.2).

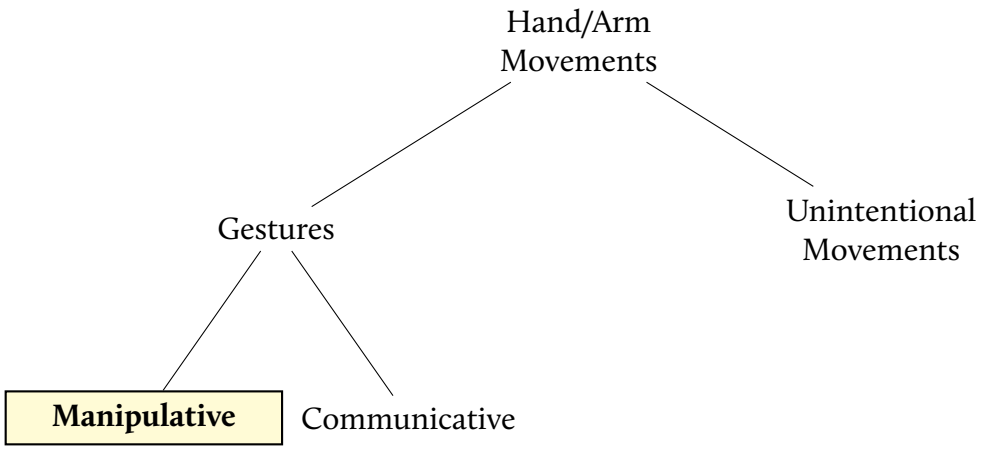


Figure 3.2 – *Hand movements – a taxonomy by [Pavlovic, R. Sharma, and Huang, 1997]*

As a first step, unintentional movements have to be distinguished from gestures. The goal is to achieve high confidence values in separating the two categories and ignore the former but act on the latter.

Second, a gesture can appear in one of two forms: it can either be of communicative nature, which means it conveys information about an abstract *concept*, or it can be manipulative. The former occurs very often in spoken language (for instance, a heart symbol formed with two hands can represent *love*). As an example for a manipulative gesture we can imagine to virtually grab and move an object.

Our application is reacting on manipulative gestures, thus is part of the latter group.

Since we are working in a virtual environment, we can not interact with objects directly, that is, by means of touching and moving. Instead, we need a *gesture interface* to do this on our behalf (Figure 3.3).

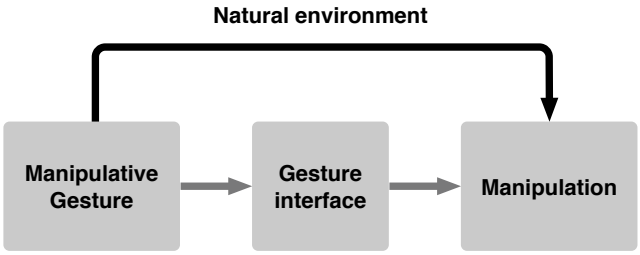


Figure 3.3 – *Manipulative gestures in natural and virtual environments*

Pavlovic, R. Sharma, and Huang [1997] distinguish 3D model-based systems from appearance-based systems as can be seen in Figure 3.5. The former category is often used in the field of computer animation. In that category we separate skeletal from volumetric models. Skeletal models infer the hand posture from a model of simple joints and segment lengths as depicted in Figure 3.4.

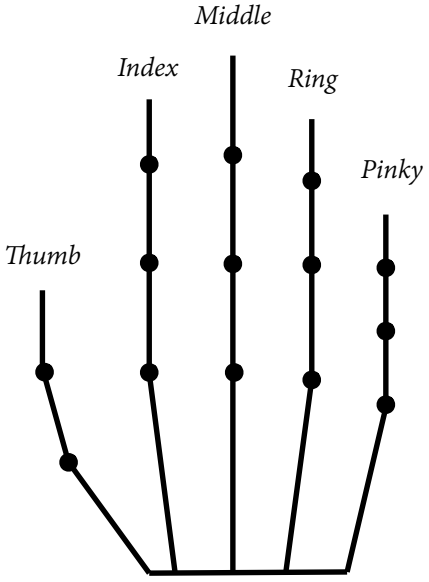


Figure 3.4 – *An example for a typical skeletal model. Source: Pavlovic, R. Sharma, and Huang [1997]*

Volumetric models are often times complex 3D surfaces like NURBS or splines. This technique can be very computing intensive. Since we focus on a low-memory, realtime approach, our method of establish a gesture model is in the second category, appearance-based systems. This means we infer gestures directly from the visual images observed as in Figure 3.5.

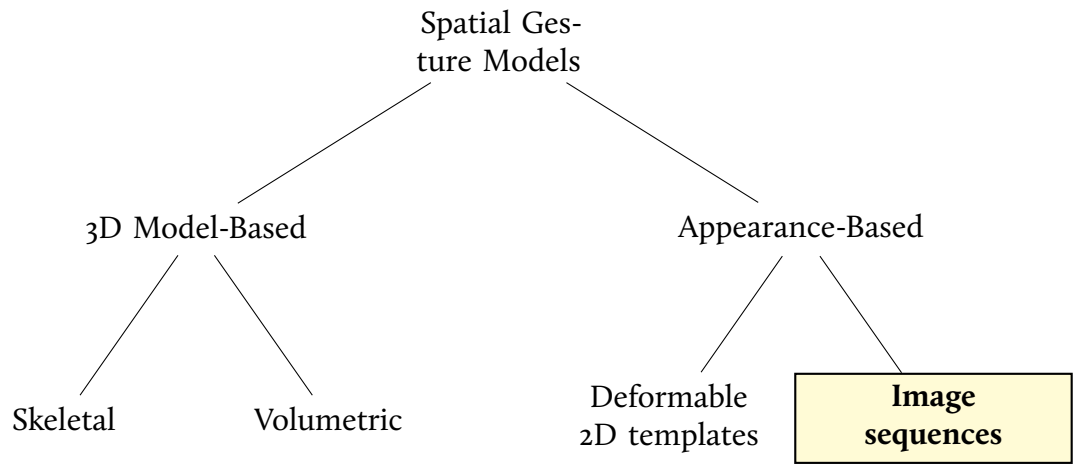


Figure 3.5 – *A taxonomy of hand gesture models for HCI. Adapted from: [Pavlovic, R. Sharma, and Huang, 1997]; [Wikipedia, 2013c]*

3.4 High level architecture

Based on the requirements of our use-cases, we divide the application into four main parts:

We need a high-level user interface we call *tracker*, which reads image data from a webcam, sends the data to the detector framework and reacts on the detection results by matching the gestures and executing system commands (see Figure 3.6). It is the controller of the application and connects all the separate parts. The other three parts are the detector framework as well as the gesture and action modules.

The detector framework, which can autonomously analyze a video stream and recognize hand shapes. It detects the position and properties of a hand. This part can be used independently from the rest and serve as a drop-in hand-detector where needed. The gesture module matches our findings to predefined signs and motions while the action module is a custom interface to the operating system.

We will introduce all necessary parts as we go along. First, we will consider the hand recognition part of the software in the following chapter, *detector framework*. After that, we give a detailed explanation about the gesture recognition module and our action subsystem.

A prototype implementation of the system is freely available at https://github.com/mre/tracker under the LGPL license [Free Software Foundation, Inc, 2013].

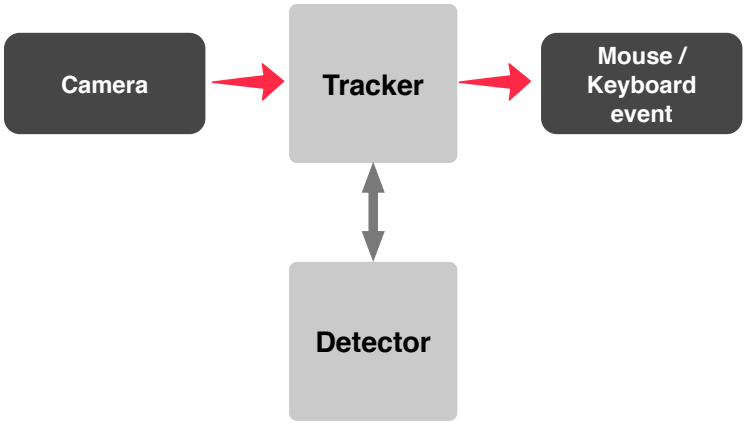


Figure 3.6 – *A high level view of the system. We will extend this schematic as we progress*

3.5 Detector Framework

Because of the many potential use-cases, a considerable amount of work covering object recognition in 2D images has been published. Good algorithms need to be lightweight (which means, they should have a small memory footprint and be reasonably fast) and general enough for practical application. Especially realtime performance is hard to achieve. Contrary to many recent research results, we do not train a neural network or apply some other machine learning technique, but combine and augment known gesture detection algorithms.

3.5.1 Framework features

The detector module consists of two parts: the `main detector class` and a number of image filters. Our `main detector class` calls these image filters which are used as hand detectors. We essentially treat each method as a *black box*, receiving input frames and returning a detection result (e.g., the hand position) and the confidence level for the detection. These results of all detectors are combined in the main detector module.

We divide the detectors into two groups based on their functionality in our framework – *position* and *shape*. The first group is responsible for obtaining the location of a hand in an image (referred to as *position cascade*), the second group is utilized to expose hand features in a given area of an image (called *feature cascade* or *shape cascade*).

We utilize the Haar classifier [P. Viola and M. Jones, 2001; Lienhart, Kuranov, and Pisarevsky, 2003], CAMshift [Bradski, 1998] and Shape [Belongie, Malik, and Puzicha, 2002] inside the position cascade and a Skin detector [Tripathi, V. Sharma, and S. Sharma, 2011; Pulkit and Atsuo, 2012] inside the shape cascade.

Figure 3.7 gives an overview of all detectors as well as the dataflow between them. We will explain the functionality of each detector in the next chapter.

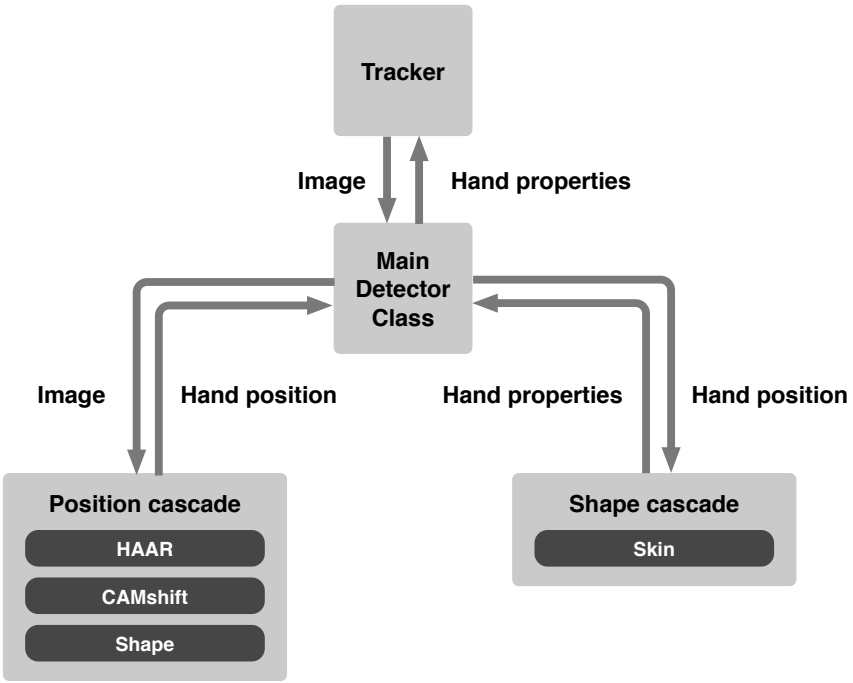
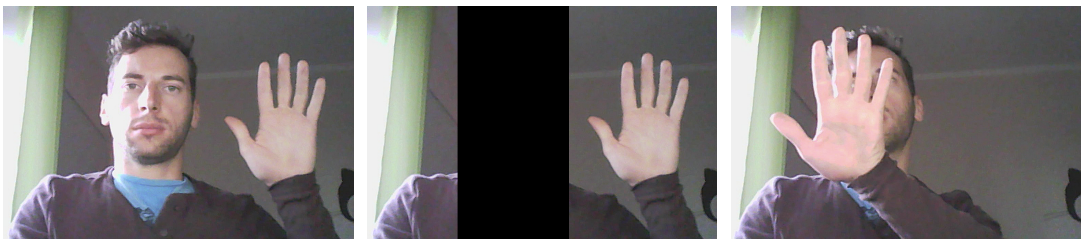


Figure 3.7 – *Dataflow between the system components*

3.5.2 Preprocessing

During initial tests, we experienced many false-positives, where a face was mistakenly classified as a hand. Therefore we remove both frontal and profile faces from the input image before we run the detectors (Figure 3.8). This step improved the hand detection rate significantly. For the actual implementation of the face removal, see 4.1.2.



(a) *Input image*

(b) *Face and upper body removed*

(c) *The hand detection still works in front of the face*

Figure 3.8 – *Face removal improves the detection process.*

3.5.3 Detector interface

No matter the type, all detectors share the same unified interface and have to implement two methods: `detect()` and `train()`, both of which receive an image as input data.

```
1 class Detector:
2   def train(img)
3   def detect(img)
4   def set_config(config)
```

Listing 3.1 – *Common interface for the detectors, an outline.*

Training The `train()` method can safely make some assumptions about the input image, which are

- At least one hand is fully visible in the image
- All five fingers are shown
- The hand is neither rotated nor tilted

This information can be used to calibrate the system. A user can specify such training data at runtime. It is important to repeat, that the framework can also run without it.

Detection The `detect()` method tries to recognize the most prominent hand in a given image. Note, that it is not certain that there is a hand in a given picture. Therefore, each position detector makes a proposal for an estimated hand position but also provides a probability for the correctness of the result. It depends solely on the detector, how the detection mechanism is implemented. Afterwards all results of all position detectors will be aggregated and weighted to produce an assumption of the final hand position. Finally, the feature detectors will extract more hand properties like the number of fingers based by analyzing the region around the hand position.

Configuration The `set_config()` method is optional. It can be used to send new configuration parameters to a given detector. We will give an overview of all configuration options for each detector in a the following chapter.

3.6 Heuristics

Just like humans are able to identify objects through reasoning, our system leverages domain-specific heuristic knowledge, which improves the recognition effort.

Every detector can have a separate knowledge base where it can store information about the world environment. Such data could be the last hand position or the delay between two valid detections. Also, the knowledge base includes simple heuristic tests to verify a detection. For instance, one test could check if the area of the last two detections is overlapping. We will explain each of these "sanity checks" which we also call *predicates* in detail for every detector.

Chapter 4

Available detectors

This chapter gives an overview of the algorithms we used for feature detection in images (also referred to as image filters).

We will discuss the purpose of every detector inside our system as well as strengths and weaknesses implementation details and our modifications. The description of each detector is divided into the following subsections. We will make slight modifications to the ordering of the subsections as far as it serves to clarify our explanations.

Subsection	Explanation
Introduction	Origin and history of the algorithm
Informal Description	A rough summary of the algorithm
Implementation	How we implemented and customized the detector
Heuristics and Confidence	Measurements which improve the detection results
Confidence	How the certainty for a correct hand detection is calculated
Strengths and Weaknesses	Advantages and limitations of the process
Configuration Options	Parameters which can be adjusted at runtime
Conclusion	A short, critical assessment of the algorithm

4.1 Viola–Jones

The Viola–Jones algorithm (also referred to as Haar filter) was introduced by Viola and Jones [P. Viola and M. Jones, 2001], further extensions (e.g., rotation) were proposed by Lienhart et al. [Lienhart, Kuranov, and Pisarevsky, 2003]. A more elaborate name for Haar would be Viola–Jones cascade-classifier. It is the first of three position detectors in our system.

4.1.1 Informal description

Recent advances in neuroscience suggest that the human retina performs a massive amount of simple feature tests to perform object recognition [Alexandre Alahi, Raphael Ortiz, and Pierre Vandergheynst, 2012]. This process can be mimicked with an array of simple binary tests over pixel regions.

We look for a number of simple hand properties in an image to detect our hand object. The properties in question could be the hand borders or the darker area between two fingers. For each hand-sized image region, which we also call a *sample*, the detector runs a cascade of simple classifiers to determine if it contains a hand. Figure 4.1 shows the principles behind Haar.

The detector turned out to be very fast and robust and is the first part of our hand position cascade.

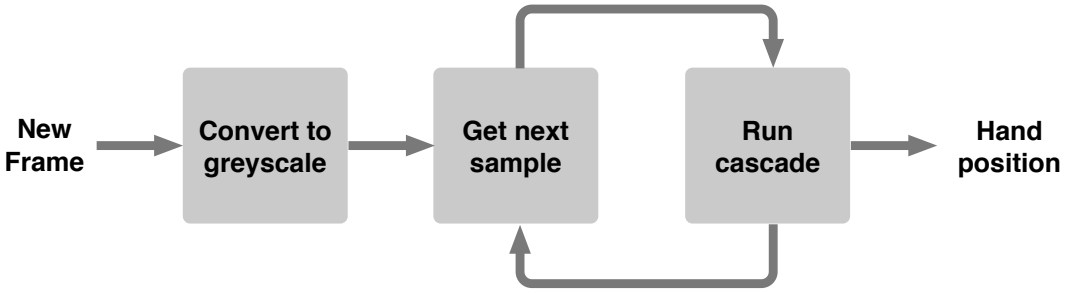


Figure 4.1 – *A slightly simplified schematic of Viola–Jones.*

4.1.2 Implementation

Object features Haar creates a magnitude of simple object features using the basic shapes from Figure 4.2.

This approach is similar to Haar bases, hence the abbreviated name[1]. A shape consists of several black and white segments (in rectangle form). Two examples

[1]For a more detailed explanation of the origin of the name, see [Chui, 1992].

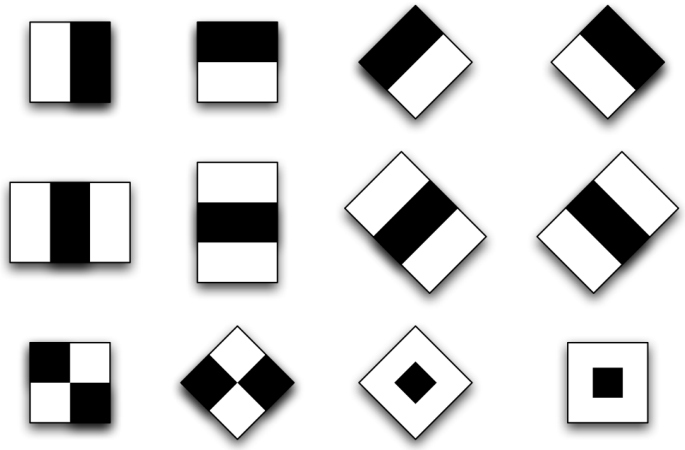
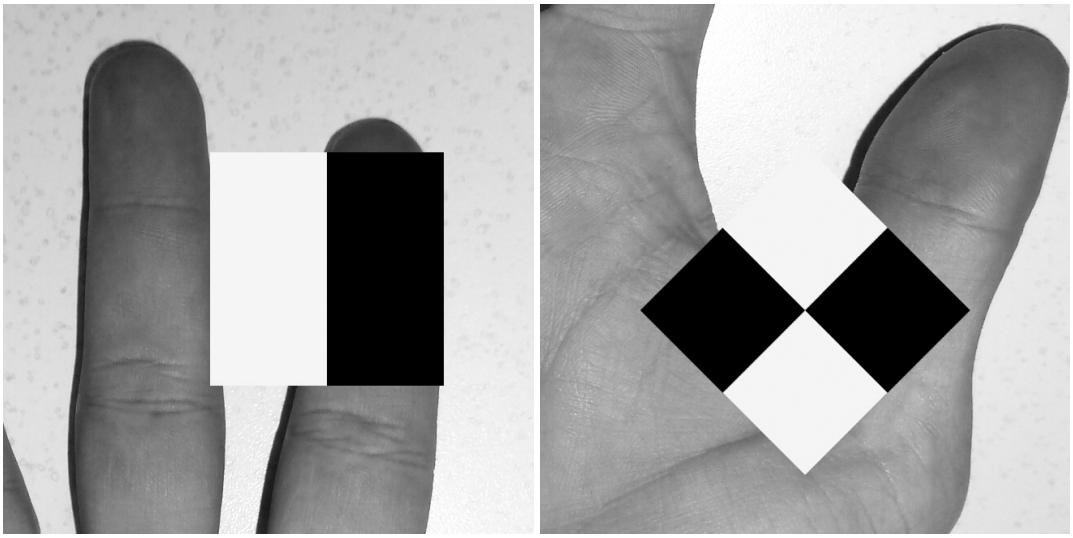


Figure 4.2 – *Basic shapes for features (wavelets)*

are given in Figure 4.3. Each shape is moved across the image at various scales. The black segments of the shape are compared with the white segments. It compares the sum of the black rectangles with the sum of the white rectangles. If the difference of these two sums is smaller than a threshold, the hand candidate is discarded.



(a) *In this example a finger is darker than the space between two fingers*

(b) *Correctly detecting the area between the thumb and the palm. The rotated shape was proposed by Lienhart, Kuranov, and Pisarevsky [2003]*

Figure 4.3 – *Two matching Haar features from a hand candidate*

Integral image For faster computations it creates an integral image of every input file. With the integral image representation, the features can be computed

in constant time (at any scale).

First, the input image gets converted to greyscale. So, for each pixel p_i of the image we have $p_i \in]0, 255[$. Then, for every location (x, y) in the original image I, the following condition holds for the integral image I':

$$I'(x, y) = \sum_{x<X, y<Y} I(x, y)$$

As a basic example for the computation of object features we investigate a shape consisting of two vertical rectangles as depicted in Figure 4.4. This shape measures the difference in intensity between the left region (the white rectangle) and the right region (the black rectangle).

We can calculate the sum of all pixel values (the intensities) of the white rectangle with

$$S_1 = F - C - E + D$$

and in an analogous manner for the black rectangle:

$$S_2 = A - B - F + C$$

Finally, to determine if the intensity difference is high enough for a positive feature recognition we evaluate

$$S_1 - S_2 \overset{!}{>} \textit{threshold}$$

Only seven additions and one comparison are necessary. Obviously the integral image representation is making feature computation very efficient.

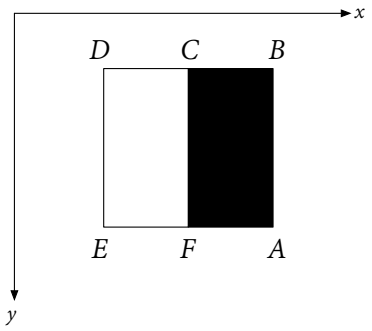


Figure 4.4 – *Using an integral image representation to calculate a haar feature*

AdaBoost Haar utilizes AdaBoost [Freund and Schapire, 1995; Freund and Schapire, 1996] to select a small set of these features, which are the most promising to rapidly classify an image region into one of two classes: either the image region contains the object we are looking for or it does not. In each round of AdaBoost, the weights on positive examples get increased such that the error criterion biases towards having low false negative rates.

As long as all weak classifiers can make correct predictions on more than 50 percent of the training examples, they can be combined to form a strong classifier for improved performance.

We see an example of this principle in Figure 4.5. The first weak classifier, $h_1(x)$, correctly predicts nine out of ten samples, thus has a success rate of 90 percent. We increase the weight of the wrongly classified sample (symbolized by the green dot in the upper right corner). After that, we accept a weak classifier $h_2(x)$ which correctly predicts this sample. It does not matter that the prediction rate of $h_2(x)$ is only 60 percent. The important fact is that it correctly categorizes samples which were previously misclassified. The combination of $h_1(x)$ and $h_2(x)$ becomes our strong classifier $H_(x)$.

Generally speaking, we create a strong classifier by calculating the weighted sum over T weak classifiers with

$$H(x) = \sum_{i=1}^{T} \alpha_i h_i(x)$$

where α_i is the weight for the weak classifier i determined during the boosting process.

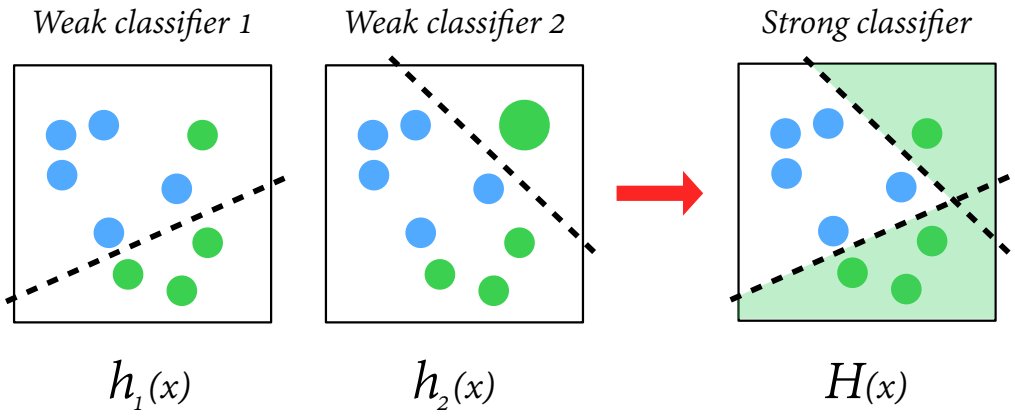


Figure 4.5 – *Combining two weak classifiers to one strong classifier.*

Classifier cascade Based on the most promising features selected with AdaBoost, Haar creates a cascade of strong classifiers. We can think of each strong classifier as a *stage* in the detection process. Each *stage* consists of several

weak classifiers that try to detect exactly one feature of the object we want to detect. If a candidate does not reach a required *threshold* in any given stage it is discarded. The classifier cascade is in fact a degenerated decision tree (see Figure 4.6). Each stage detects almost all positive features and removes a significant part of negative candidates, thus the chain yields highly accurate results.

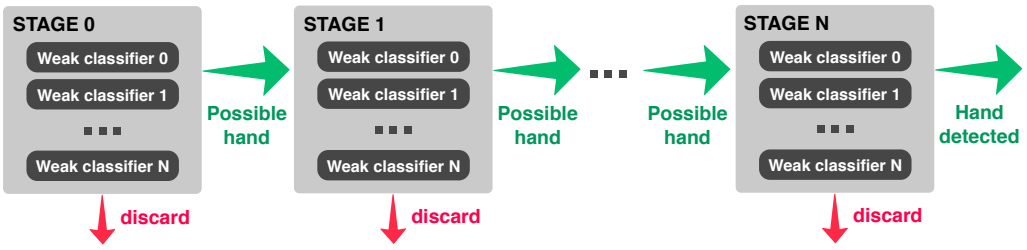


Figure 4.6 – *A Haar cascade boosted with AdaBoost. Weak classifiers are combined to strong classifiers, also called stages. At each stage, a possible hand can be discarded. The objects which remain at the end are classified as correct hand detections.*

Cascade files In order to extract the most relevant features for hand detection, a lot of training data is required. In our case these were hands of different persons, photographed under varying lighting conditions. Also, a large body of negative samples – images which don't contain a hand – is necessary.

These samples need to be manually classified into hand and non-hand images. To create a well-trained cascade, an enormous number of images need to be collected. For instance, a detector for frontal faces was trained with about 15000 test images [P. Viola and M. Jones, 2001]. The *traincascade* application that comes with *OpenCV* helps to synthesize a great number of the training data by randomly placing a face inside an arbitrary background. Nevertheless, searching, licensing, editing and tagging sample images is a tedious task.

For every feature, the program calculates the value of a stage threshold to split the non-objects from the objects in the image database. The OpenCV `haartraining` program chooses a stage threshold that admits 99.5% of the training positives but only 50% of the training negatives [Seo, 2008]. For a sophisticated sample size, actual training can take days or even weeks.

As a consequence, we did not train our own cascade but instead use two freely available resources which can be passed as `XML` files to the *OpenCV* Haar implementation. Our primary cascade comes from Li [2011] and was constructed from approximately 800 positive samples and 1100 negative samples. The author notes, that an increase of the sample size did not yield better results. This cascade is characterized by a comparatively small number of hand features which improves detection speed. Only if this primary cascade fails to detect a hand position, we use our secondary cascade from Markou [2012]. With a sample size of 20,000 positive and equally many negative images it is significantly larger than

the first cascade. As the author notes, it works on any orientation but has a high false positive rate. Both cascades detect *open* hands but we also included a cascade for fists from Nambissan [2013].

When we run the cascade on our image we receive a number of bounding rectangles around the are where Haar assumes an open hand.

Dynamic detection accuracy It is often the case that the Haar classifier detects the same object at a slightly different position (referred to as *neighbors* in the *OpenCV* documentation (see [OpenCV Development Team, 2013a])). The more neighbors were found, the more likely an object has been correctly detected in the considered image region. The OpenCV Haar classifier implementation offers a parameter to adjust the minimum number of neighbors needed in order to retain a candidate. Our implementation dynamically adjusts this parameter based on heuristics like hand proportions. If more than one hand is found in an image, the number of required neighbors will be increased, if no hand is found for a longer period of time, the number decreases accordingly. With this option we can support a wide variety of use-cases: Under optimum conditions this adjustment guarantees stable results and in difficult environment it sustains a minimal viable product.

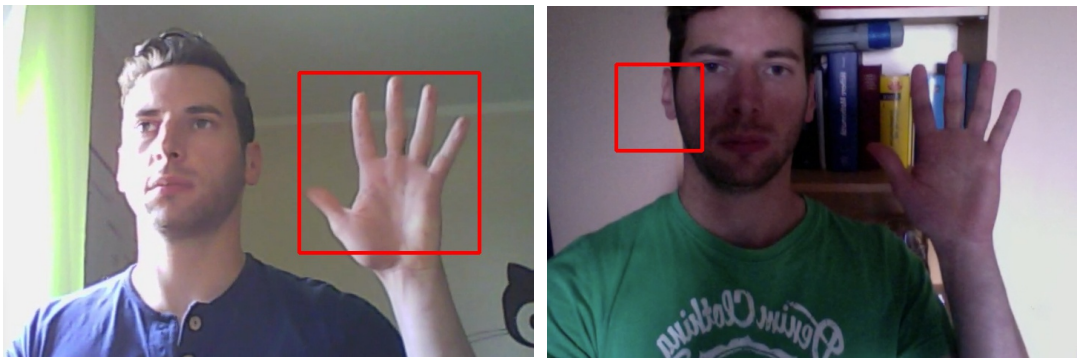
Since we assume that the user is sitting directly in front of the camera, we expect the largest of all bounding rectangles to contain the hand.

Face detection Besides using Haar for hand detection, we also use it for initial face removal in a preprocessing step inside the detector framework itself (we separated both use-cases – hand and face detection – logically within the system but use the same underlying framework for both). We use two cascades for face detection: the standard cascade for frontal faces (created by Rainer Lienhart from Augsburg University [Lienhart, 2013]) which comes with *OpenCV* and a special cascade for profile faces (contributed by David Bradley from Princeton University [Bradley, 2003]) which is also included in the framework. Since the user is sitting in front of the screen, his face is typically not moving very much during a short timespan. Thus, we only run the face detector every n frames to save processing time. The rest of the time we use the previous detection result.

4.1.3 Strengths and weaknesses

Viola Jones proved very robust during our tests. Especially for simple, plain-colored backgrounds it yield good results. On the other hand Viola Jones falls short on complex backgrounds (as seen in Figure 4.7) and in difficult lighting conditions. Also, it is quite vulnerable to hand rotations (see Figure 4.8). Since all implemented hand gestures require the hand to be upright, this is not an issue.

It is fast and can purely work with offline data thus can be used from the start without prior adjustments. A major advantage is that the filter is agnostic to skin color. It is only recognizing greyscale image features. Therefore, it works on backgrounds which have the same color range as human skin.



(a) *A successful hand detection on a simple background*

(b) *An unsuccessful hand detection on a complex background. Note that the hand is partially overcast by a shadow*

Figure 4.7 – *Comparing Haar detection results on simple and complex backgrounds*

(a) *Multiple, mostly correct detections of varying size when the hand is upright*

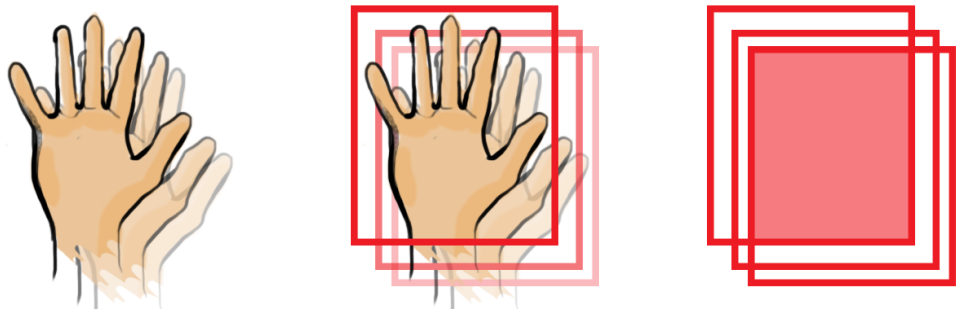
(b) *A single detection when the hand is slightly tilted*

(c) *An incorrect result for a heavily rotated hand*

Figure 4.8 – *The hand*

4.1.4 Heuristics and certainty

The Haar detector runs a number of "sanity checks" or *predicates* at every frame. These predicates are simple heuristics which enable us to make an estimation of the quality of our detection result. Since they are lightweight they can be executed in realtime. The quality is expressed by a confidence value that will be returned by the detector along with the estimated hand position. In order to keep track of the detection quality the Haar detector employs a history of the estimated hand positions of the last few frames.



(a) *Hand movements during three consecutive frames*

(b) *Rectangles around the hands*

(c) *The area where the three "hand rectangles" overlap*

Figure 4.9 – *The Haar filter often "jumps" to a completely unrelated region of an image for one frame during hand tracking. Therefore a valid detection requires that consecutive hand positions overlap.*

Overlapping frames During initial tests we observed that the Haar position detector tended to make sudden jumps from one image location to another in complex settings (e.g., with a bookshelf or a poster in the background) or with high image noise due to bad lighting conditions. In both cases the result were many false positives. With a targeted frame rate of at least 24 frames per second, each frame is analyzed in about 40ms. During that short timespan we can safely assume that the hand is roughly at same spot during successive frames. With that in mind we can give a heuristic for correct detection: If the estimated hand position did change only slightly during the last few frames, chances are the detection is correct. Sudden jumps on the other hand are penalized. By default, two consecutive "hand rectangles" need to overlap by 70 percent (see Figure 4.9). If they don't the history will be reset.

Gaps In the same vein we track the number of gaps in the detection process. We speak of a gap if a hand object is constantly detected in an image sequence but a few outliers where no hand was found. Currently we don't allow two consecutive frames to be outliers. If this case occurs we invalidate the current history.

Irregular face detection If we find a hand in an image and no faces are detected at the same time, the probability for a correct detection is relatively high (no user is facing the camera). The same is true if a face is constantly found over many frames. Problems arise, when a face is detected irregularly – less often than a hand. In that case we lower our confidence for a correct hand detection based on the graph in Figure 4.10.

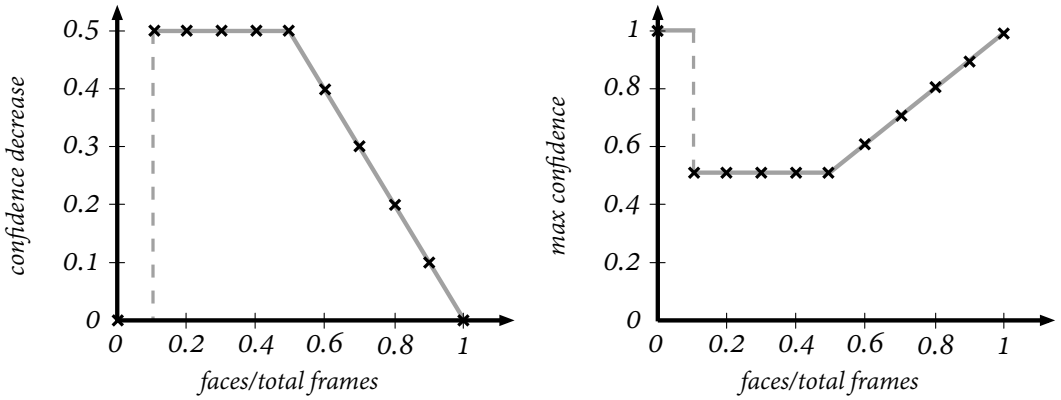


Figure 4.10 – *Influence of face detection on the confidence in the hand detection*

Confidence We combine all of the above factors to attain a performance indicator for the detection process – a probability for the correct hand position. We mainly base our rating on the history length which has proven to be a good indicator for the stability of the detector. The first two predicates, namely *Overlapping frames* and *Gap* influence the history length. A new frame only gets added to the history if these two predicates run successfully. If one of them fails, the history will be cleared.

Additionally the result of the third predicate – *Face detection* – will be taken into account such that

$$p_{haar} = min\left(\frac{h_{current}}{h_{max}}, 1\right) + p_{face},$$

where p_{haar} is the confidence in the detection result, $h_{current}$ is the current history length, h_{max} is the maximum history length and p_{face} is the result of the *Face detection* predicate.

4.1.5 Configuration options

We provide sane default values for the hand size which are both conservative enough to find hands in all reasonable sizes for our application and optimized for realtime usage. Also we provide a number of other cascades for the detector which can be loaded at runtime. They can be used to implement a wide variety of gestures.

4.1.6 Conclusion

Haar is flexible and fast. It requires no prior training other than a well-trained cascade. Since it is mostly agnostic to lighting conditions it is a good foundation

to bootstrap other detectors from its output as we will see after the presentation of the other detectors.

4.2 CAMshift

> *CAMshift was initially proposed by [Bradski, 1998]. It has since seen use in countless object tracking processes from people tracking [Guraya, Bayle, and Cheikh, 2009] to traffic surveillance [Xia, Wu, Zhai, and Cui, 2009]. We too use it as a position detector within our system.*

4.2.1 Informal description

CAMshift is a color-based tracking algorithm. It requires a frame which shows the hand and initial bounding box around the hand to get started. The algorithm calculates the hue values inside the bounding box. These values will be tracked from now on. One could say we "follow" the hand movement. In each subsequent frame we look at the area where we found the hand before and see how the position has changed. We update our bounding box so that the hue values inside the box resemble the old values. Figure 4.11 depicts the general procedure of the algorithm.

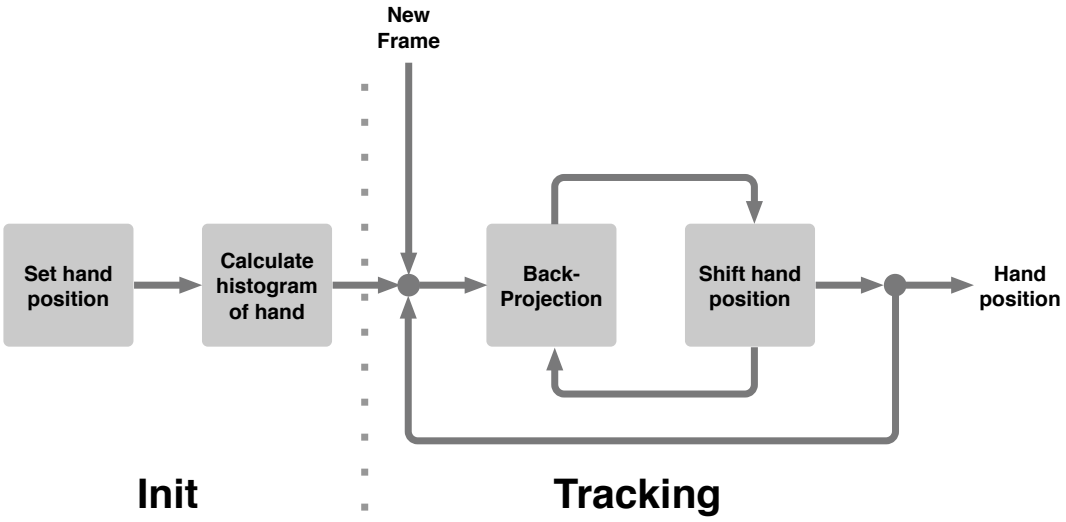


Figure 4.11 – *A slightly simplified schematic of CAMshift.*

4.2.2 Implementation

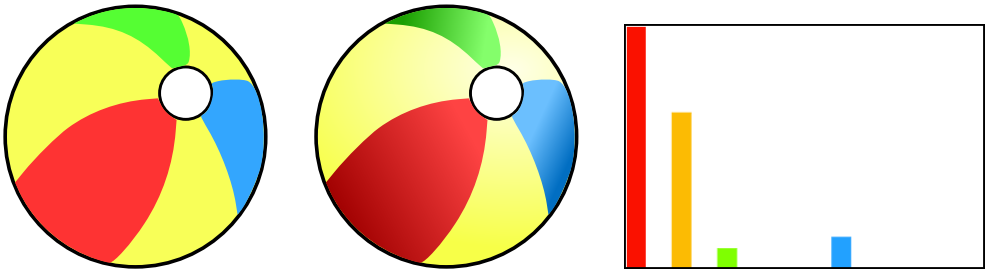
CAMshift (Continuously Adaptive Mean-Shift) computes a hue histogram of a rectangle area (a search window as shown in Figure 4.12) and, with every new

Figure 4.12 – *A typical use-case for the program. The hue values of the hand are only a tiny part of all values of the histogram*

frame, shifts the tracked area based on probabilities of the pixels to be part of the tracked area (hence one part of the name). More precisely, the new tracked area is the old one shifted to the center of gravity of most probable pixels. After the new area has been determined, the histogram is updated; this continuous adaptation explains the other part of the name.

Basically the algorithm compares two histograms: the one of the tracked object and the one of the area defined by the search window [Jung, 2013].

Creating a color histogram The algorithm is based on the idea that the color distribution of every object is very specific. A color histogram is a way to express this observation in a mathematical way. We take the hue of our HSV color space and divide it into n ranges, often called *bins*. Next we count the number of pixels which belong into each bin. We store this histogram of the object we want to track for future use. It will be compared with the histogram of the succeeding frames Note that we only consider the hue values of the image. This way the process is more robust against light changes (we ignore the changes in the lightness value *V*). It also works for both light- and dark-skinned people (since the fleh color only differs because of varying saturation and not hue, see [Bradski, 1998]). For instance, consider the beach ball in Figure 4.13a – which has four distinctive hue values (red, green, blue and yellow). Changing the luminance of the these color values (Figure 4.13b) does not influence the histogram (Figure 4.13c). In the original Paper [Bradski, 1998], an *RGB* color model was used. Bradski [1998] switched to a *HSV* model due to the benefits mentioned. We also use the *HSV* color space in our implementation.



(a) *A plain beach ball drawing*

(b) *A more vivid beach ball drawing with shadows.*

(c) *The histogram is the same for both drawings*

Figure 4.13 – *Because we use the* HSV *colorspace for our CAMshift implementation, the result is invariant to lighting variants. Both drawings share the same color histogram.*

Back–Projection With the color histograms we can follow the hand while it is moving through the room by applying a technique called *Back–Projection*.

We calculate the bin for each pixel of a new image. Then, for a random pixel in that image, we can give a probability that it belongs to the tracked object with

$$p_i = min\left(\frac{H_i(object)}{H_i(image)}, 1\right),$$

where $H_i(object)$ is the quantity of pixels in bin number i for the tracked object and $H_i(image)$ is the number of pixels in bin i for the whole image.

When we calculate this probability for each pixel in the image, we can create a matrix, commonly referred to as the *Back–Projection* of the image. We can visualize the *Back–Projection* by creating a greyscale image B from these probabilities. The brighter a pixel in B, the higher the probability that it belongs to the object we want to track. An example is depicted in Figure 4.14. Note that this is also true for other skin-colored objects like faces and perhaps objects in the background (compare with Figure 4.15). We need an additional step to get rid of these false positives.

Adaptive mean shift With each new frame, CAMshift centers its search window over the area with the brightest pixels in the *Back–Projection* image B [2]. It does so by using the previous location as a starting point and climbing the gradient of the probability distribution. One could say we are gravitating towards the brightest pixels of the image. It *shifts* the search window to the center of gravity. This *shift* can be executed several times during each frame. The algorithm

[2]We recall that the brightness of a pixel provides information about the probability that the pixel belongs to the tracked object. The brighter a pixel the more likely it belongs to our object.

Figure 4.14 – *The result of a Back–Projection process. The brighter a pixel, the higher the probability that it belongs to the tracked hand.*

converges at the mean location of the probability distribution. We can see the process for a single frame in Figure 4.16.

More precisely, we are using image moments to find the center, size and orientation of the hand object (see [Freeman, Tanaka, Ohta, and Kyuma, 1996]). The definition of the moment for a continuous function $f(x, y)$ is defined as

$$M_{pq} = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x^p y^q f(x, y)\, dx\, dy; p, q \in \mathbb{N}_0$$

We can adapt this function for discrete image data to derive the image moment i, j:

$$M_{pq} = \sum_x \sum_y x^p y^q I(x, y)$$

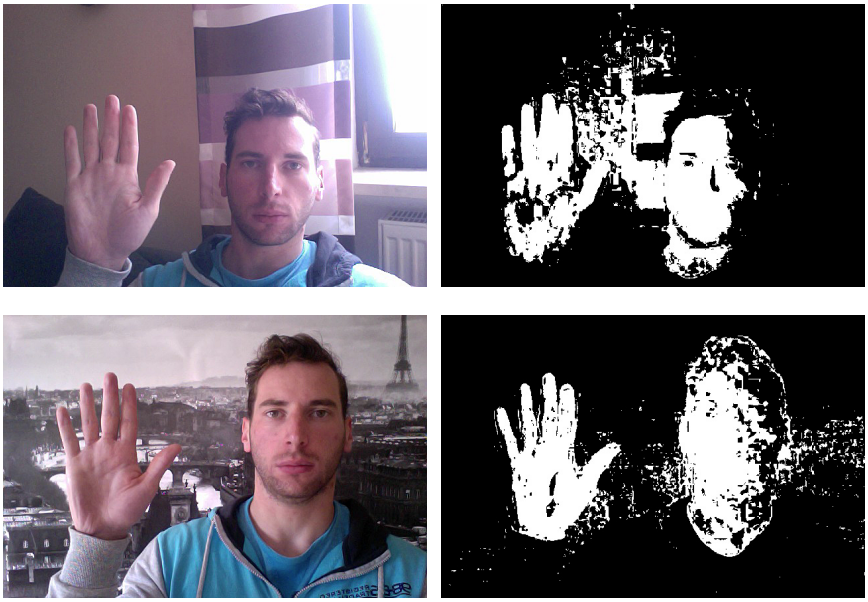


Figure 4.15 – *The result of a backprojection iteration. We applied a threshold to the result for better visibility. All white pixels belong to the tracked object with a probability greater than* 0.5*. The algorithm produces a lot of false positives with skin-colored backgrounds (the top two images) but yields much better results on neutral backgrounds (the bottom two images).*

we use M_{00}, M_{10} and M_{01} to calculate the center of the and M_{11}, M_{00}, M_{20} M_{02} to determine the size and rotation of the hand.

To get these values, we are executing the following steps:

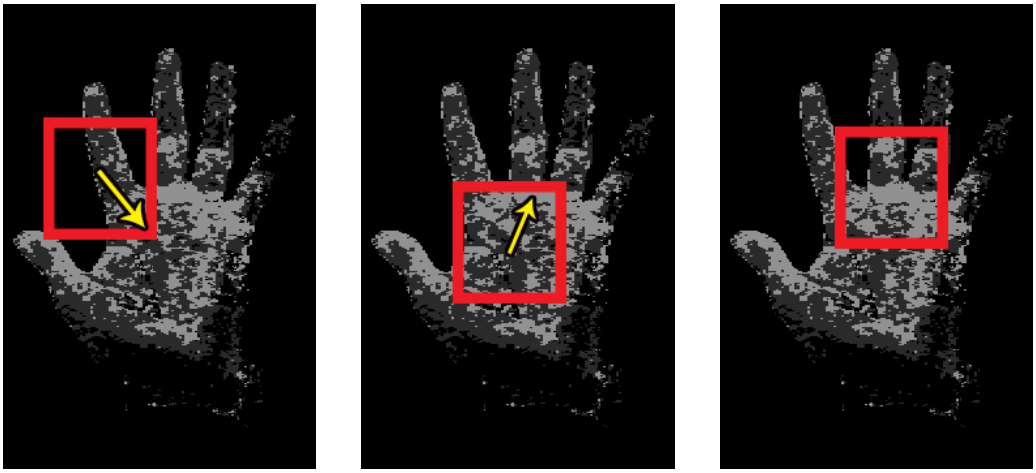
1. Choose a search window W of size s centered at the data point p_k.
2. Calculate the zeroth moment as well as the first moments of the search window:

$$M_{00} = \sum_x \sum_y I(x, y)$$

$$M_{10} = \sum_x \sum_y xI(x, y); \quad M_{01} = \sum_x \sum_y yI(x, y)$$

where $I(x, y)$ is the value of the probability distribution at position (x, y) of the image.

3. Shift the search window to its new location at $x_c = \frac{M_{10}}{M_{00}}$ and $y_c = \frac{M_{01}}{M_{00}}$



(a) *An arbitrary search window is chosen. We calculate the gradient of the probability distribution of our hand pixels*

(b) *We relocate the search window to the location with the highest density of object pixels (the brightest pixels).*

(c) *After another iteration of calculating the gradient and moving the search window the algorithm converges.*

Figure 4.16 – *The Meanshift algorithm is used to determine the optimal position of the search window. Inspired by Belaroussi [2006]*

This process is based on research by [Comaniciu and Meer, 2002], which was not originally designed for object tracking but for the "analysis of a complex multimodal feature space". It had been used for low-level vision tasks like image segmentation.

For each new frame, CAMshift is limiting the search space (the area analyzed for hand detection) to a rectangle around our current tracking window. This way, only a part of an image needs to be examined which tends to result in major performance gains.

Hand size and angle Using the second moments of the back projection image, we can calculate the hand angle.

$$M_{20} = \sum_x \sum_y x^2 I(x,y); \quad M_{02} = \sum_x \sum_y y^2 I(x,y)$$

$$\alpha = \arctan\left[\frac{2\left(\frac{M_{11}}{M_{00}} - x_c y_c\right)}{\left(\frac{M_{20}}{M_{00}} - x_c^2\right) - \left(\frac{M_{02}}{M_{00}} - y_c^2\right)}\right]$$

We can calculate the length l and the width w (the two main axes of the ellipse) of the distribution centroid around the hand object with

$$l = \sqrt{\frac{(a+c) + \sqrt{b^2 + (a-c)^2}}{2}}$$

$$w = \sqrt{\frac{(a+c) - \sqrt{b^2 + (a-c)^2}}{2}}$$

where

$$a = \frac{M_{20}}{M_{00}} - x_c^2,$$

$$b = 2\left(\frac{M_{20}}{M_{00}} - x_c y_c\right),$$

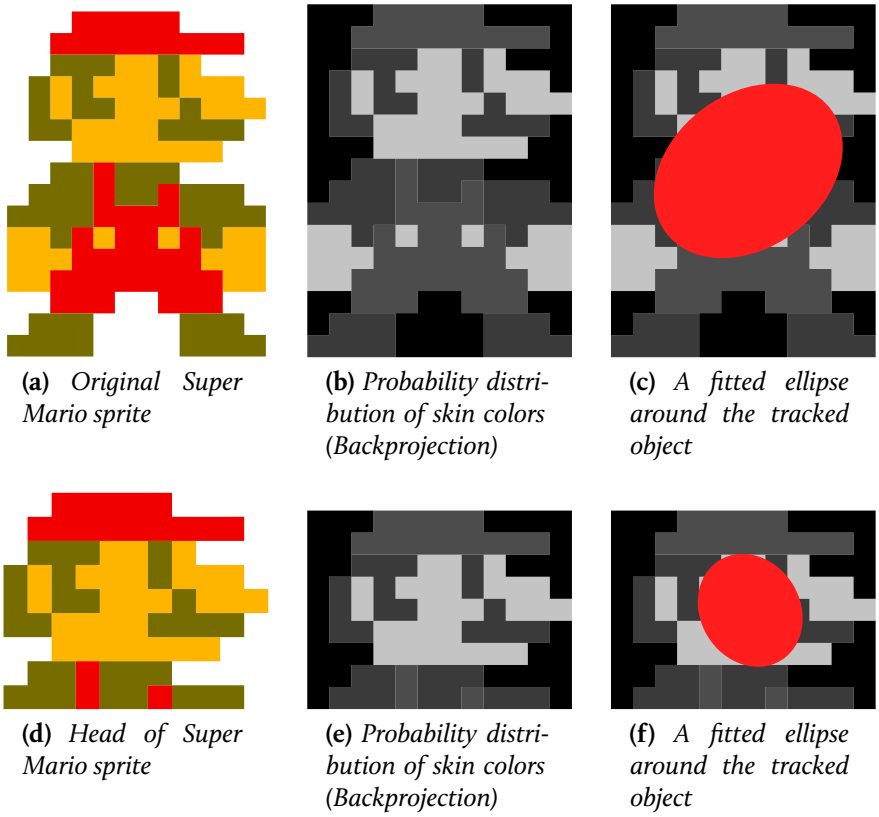
$$c = \frac{M_{02}}{M_{00}} - y_c^2.$$

Figure 4.17 shows an example ellipse which was calculated with the above formulas. See the appendix for an explanation of the source code which was used to create the image.

4.2.3 Strengths and weaknesses

The ingenious tracking approach of of the algorithm makes CAMshift ideal for our application: It is optimized for execution speed and low memory overhead. Compared to Haar, Camshift was most useful in complex backgrounds. With its backprojection technique, it can adapt to changing brightness and hand rotation.

An initial region of interest (a significant area of the image which shall be tracked) is required to get the tracking started. If it were the only detector we are using, we would need to select the region manually. This requirement would make the process quite cumbersome and essentially render the detector useless for human-computer interaction. We will later see that we can provide the data automatically using the results of other detectors. A convenient quality of the algorithm is the resistance to common error sources of other detectors. Since the tracking window is following the *gradient* of the probability distribution, it won't get influenced by outliers (e.g., image noise which produces bright or dark spots in the distribution). This property is also helpful for selecting the initial region around the tracked object. An approximate selection of the region is fully sufficient since the algorithm adapts to the actual hand size pretty quickly using mean-shift as described above. The result can be seen in Figure 4.18. Furthermore the adaptive mean shift allows tracking of objects which change their size over time (e.g., by moving away or towards the camera).



(a) *Original Super Mario sprite*

(b) *Probability distribution of skin colors (Backprojection)*

(c) *A fitted ellipse around the tracked object*

(d) *Head of Super Mario sprite*

(e) *Probability distribution of skin colors (Backprojection)*

(f) *A fitted ellipse around the tracked object*

Figure 4.17 – *Fitting an ellipse to the skin color distribution of a simple 2D object. Note how the hands of the character influence the centroid of the ellipse as well as its size and orientation. Source: Vectorized drawing of output from own program. Source of the original sprite: Nintendo Inc.*

But the adaptive adjustment of the tracking window can also become a disadvantage. Figure 4.19 shows two typical misaligned ellipses. Initially the hand object gets correctly tracked but the region of interest grows near other skin-colored image parts like the wrist or the face. In that case the tracking area gets expanded over an area much larger than the hand itself and the quality of the detection decreases. Our tests have shown that it can be hard to recover once this situation has emerged. As a consequence, the filter is prone to error, when other skin-colored objects (like the wrist or the face) are visible. We use heuristics to mitigate that fact.

4.2.4 Heuristics and certainty

Ellipse ratio In order to assure reasonable hand detection results, we continuously control the ellipse size around the object. We found that the ratio of the two main axes of the ellipse provides information on the hand state: A ratio of about two suggests an open hand (palm visible) and a ratio of about 1 suggests

Figure 4.18 – *Adaption of the hand ellipse size over time.*

(a) *The ellipse around the object gets extended to the face*

(b) *The ellipse around the object gets extended to the wrist*

Figure 4.19 – *Typical error symptoms of CAMshift*

a fist (see Figure 4.20 and 4.21). This observation could be used to implement a *grab* gesture. An initial ratio of about 2 followed by a (linear) decrease of the ratio to about 1 and a subsequent increase to its previous value can indicate that the user "grabbed" and object and "released" it again. A ratio which is significantly above or below these two values indicates problems with the tracking system. A countermeasure could be to restart the detector or to decrease its confidence significantly.

Hand rotation During normal usage the hand is neither tilted nor rotated. With that in mind, we assume normal behavior if the hand rotation is within reasonable boundaries (about ± 30 degrees). If it increases too much we signal erroneous behavior and can adjust the confidence accordingly.

Confidence With each new frame we calculate the average skin probability inside the search window (the skin color ratio around the hand). We compare these values over time to look for changes which could demonstrate that we lost track

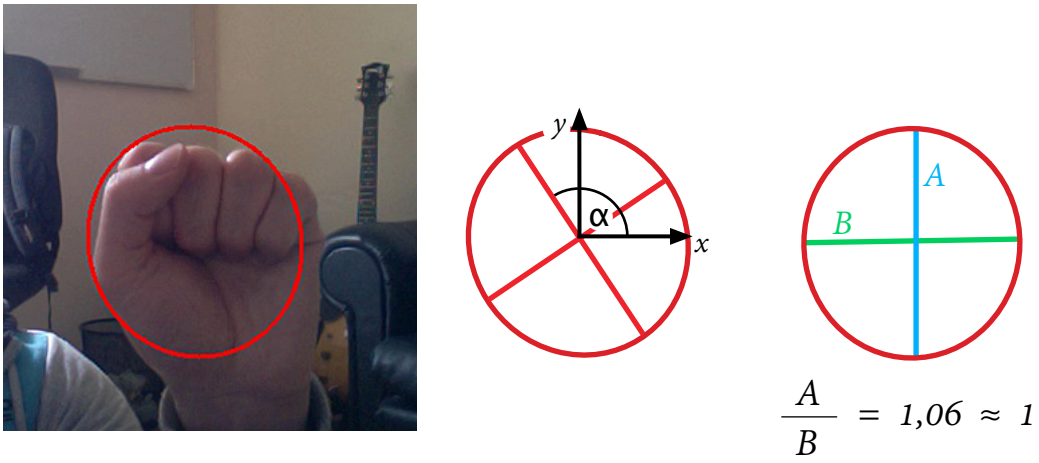


Figure 4.20 – *In our tests the ratio of the both axes of the ellipse around the fist is almost one and resembles a perfect circle.*

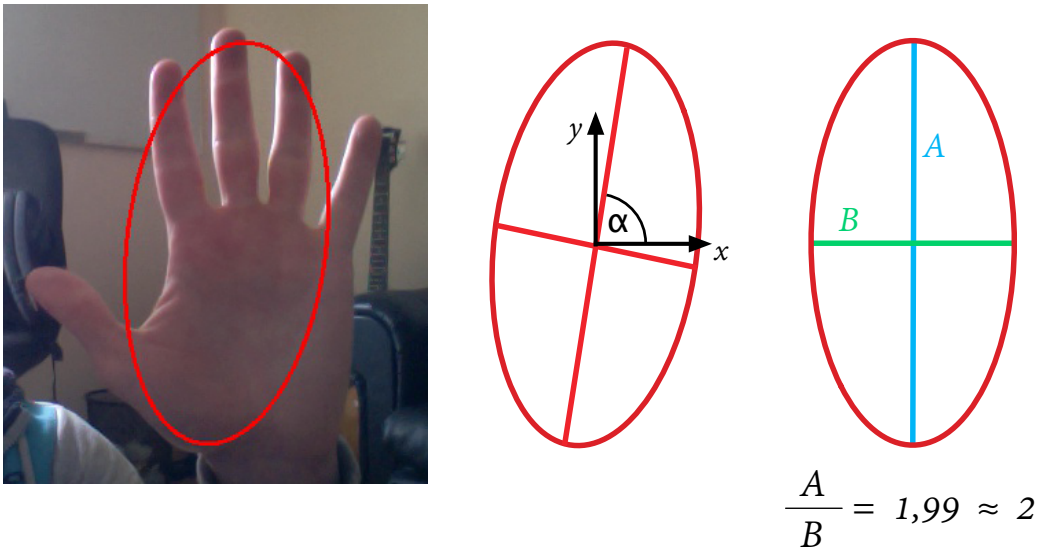


Figure 4.21 – *The ratio of the both axes of the ellipse around the palm is roughly two*

of the hand. Since the hand ellipse is bounded by the search window this heuristic is a good estimation for the above two heuristics. Therefore we use it to return our confidence value for the detector.

4.2.5 Configuration options

We can change from a basic skin-color model which covers most use-cases to a more sophisticated dynamic model which we generated ourselves at runtime. With the dynamic model we have multiple separate settings which were created from many sample images of different people with varying flesh colors. We use the same dynamic model for our skin detector which we will describe later.

4.2.6 Conclusion

The issues of CAMshift can be mitigated by Haar and vice versa. This way, the two filters complement each other quite well which we will do in the next chapter.

Some promising improvements to the algorithm have been proposed. Of particular interest is the work of @6045533, which assert to increase the stability of the process by avoiding background information (other objects which shall not be tracked) to become part of the model, which prevents *drifting* (we loose track of the object). We could test this method in a future version.

4.3 Shape

A prime work on template matching [Belongie, Malik, and Puzicha, 2002] introduced the 2D shape search for object detection*, although did not use it for the hand or face detection. See also the paper by Tahmasebi, Hezarkhani, and Sahimi [2012]. We use the algorithm as the last of three position detectors within our system.*

4.3.1 Informal description

We can get an idea of the algorithm by looking at Figure 4.22. Given an base image with size $W \times H$ (also referred to as *haystack*) and a smaller template image of size $w \times h$, (referred to as *needle*), we slide through the base image and compare patches of size $w \times h$ with the template image using a mathematical function. The result of this operation will be stored in a matrix. It is to be expected, that the needle appears at the patch position with the maximum or minimum (depending on the mathematical function we use) matrix value. We use this position for our hand proposal.

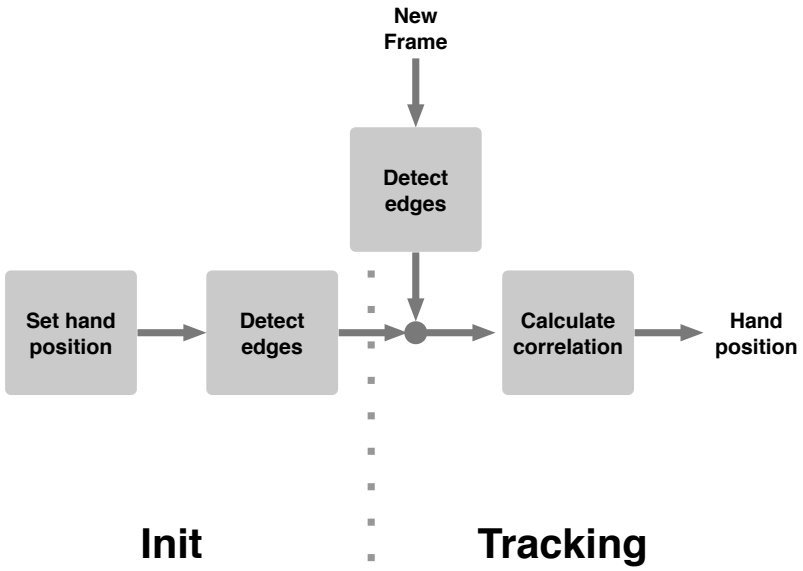


Figure 4.22 – *A slightly simplified schematic of the shape detector.*

4.3.2 Implementation

Our Shape filter detects the hand position in an image by moving a template image across the camera frame and calculating the likelihood of a match at each location Figure 4.23. We can find the optimum position for the template image

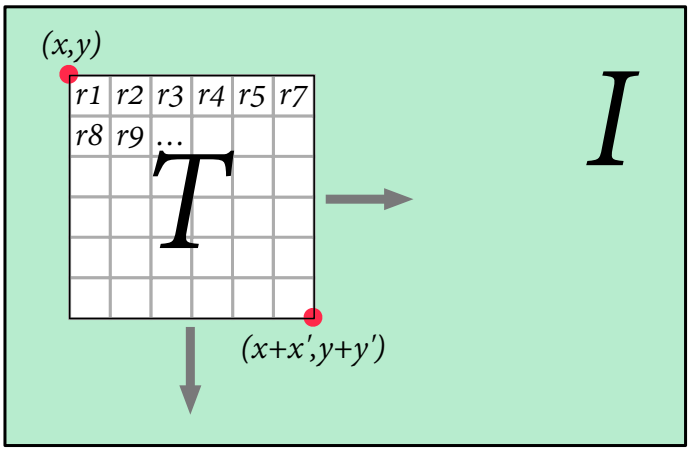
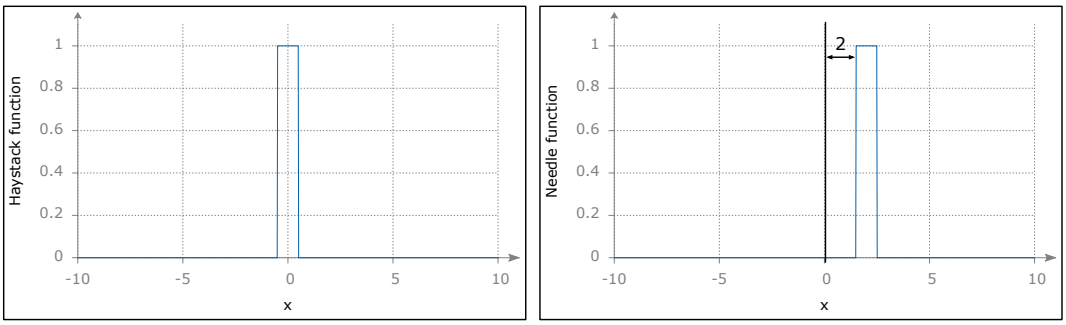


Figure 4.23 – *The template matching process. Moving the small template image* T *across a base image* I*. We calculate the residuals at every position. Comparing the sums of these residuals gives a good estimate for the best match.*



(a) *The haystack function* **(b)** *The needle function*

Figure 4.24 – *Our running example is a search problem using two rectangle functions. We try to find the needle in the haystack.*

within the haystack image at the place where the difference between the template image pixel values and haystack pixel values is minimal.

Several mathematical functions – also called *likelihood models* – are commonly used for this template-based comparison. We use two very common functions, the squared difference and the cross correlation.

As a running example we will consider two real valued functions, the needle function and the haystack function (as in Figure 4.24. One of them (the needle) was shifted along the x-axis by an unknown factor. In order to find the factor, we slide one function (the needle) along the x-axis and calculate the *difference* of both functions at every position. Depending on the method to calculate this *deviation*, we find the best postion for our needle at either the minimum of the maximum of this operation.

Squared difference A good measure for the deviation of two datasets is the method of least squares, proposed by Carl Friedrich Gauss in 1809 [Gauss, 1809] [3].

We define S as the sum of squared residuals:

$$S = \sum_{i=1}^{n} r_i^2$$

where the residual r_i is defined as the difference between the values of both datasets at position i.

Given two functions $f(x)$ and $g(x)$ we can define

$$r_i = g(x_i) - f(x_i)$$

to calculate our residuals.

Similarly, we can define the residual for scalar functions of n dimensions. In the case of template matching, we interprete both images as a function of two variables: $T(x, y)$ for the needle and $I(x, y)$ for the haystack.

Thus, we can define our residuals as

$$r_i = T(x_i, y_i) - I(x_i, y_i)$$

and derive the sum of squared residuals as

$$S = \sum_{i=1}^{n} r_i^2 = \sum_{i=1}^{n} (T(x_i, y_i) - I(x_i, y_i))^2.$$

If we move the needle image T across the haystack image I, we can give a criterion for the quality of the match at position (x, y). We do this by comparing all pixel values of T with the values of I at a given position.

$$S = \sum_{x',y'} (T(x', y') - I(x + x', y + y'))^2$$

In order to make the outcome of the template matching comparable with each other, we are normalizing our results to get the final function used by the detector:

[3] For a recapitulation of the historical events surrounding the method, see [Sorenson, 1970].

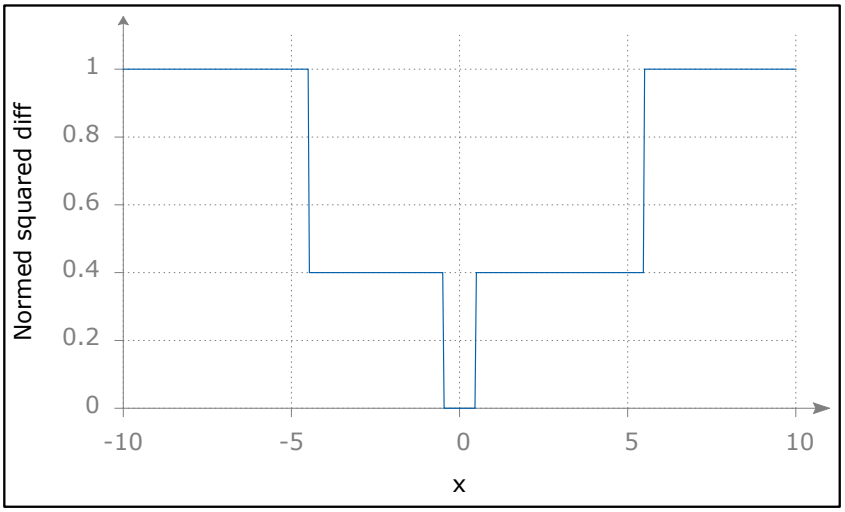


Figure 4.25 – *Applying the normed squared difference to the rectangle functions from above. We scaled the function so that its values are between 0 and 1.*

$$R_{nsqdiff}(x,y) = \frac{\sum_{x',y'} (T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

In the above formula, T refers to the template image, I refers to the search image and $R_{nsqdiff}$ is the result of the operation. The minimum of $R_{nsqdiff}$ is our estimated hand position (see Figure 4.25).

Cross correlation A second commonly used function is the the (normed) cross correlation.

For two continuous functions $f(t)$ and $g(t)$ it is defined as

$$(f \star g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f^*(\tau)\, g(t+\tau)\, d\tau$$

where f^* denotes the *complex conjugate* of f (see [Weisstein, 2013]).

This can be expanded to functions with n dimensions (see [Kheng, 2013]). For two continuous functions with two dimensions, $T_c(t,u)$ and $I_c(t,u)$ we can write

$$(T_c \star I_c)(t,u) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} T_c^*(\tau,v)\, I_c(t+\tau,u+v)\, d\tau\, dv$$

where T_c^* denotes the of T_c.

For image data, this can be written in discrete form as

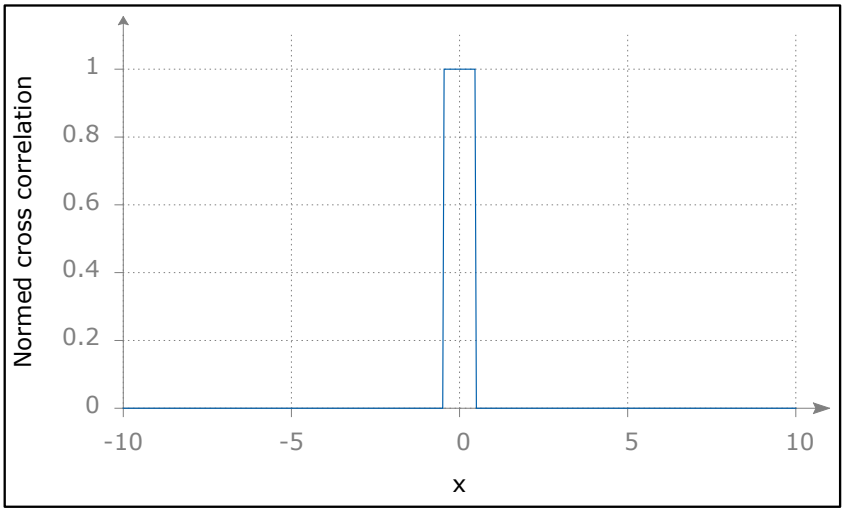


Figure 4.26 – *Applying the normed cross correlation to the rectangle functions from above. We scaled the function so that its values are between 0 and 1.*

$$\sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))$$

where x' and y' are used to iterate over the template image at position (x, y). As above, we normalize the result to receive our final likelihood model.

$$R_{nccorr}(x,y) = \frac{\sum_{x',y'} (T(x',y') \cdot I(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}$$

Contrary to the normed squared difference function, we look for the maximum value of this function for get the best match (see Figure 4.26).

Result When we apply one of the two likelihood functions – $R_{nsqdiff}$ or R_{nccorr} – to our input images, we get a matrix of *similarity values* for each point (x, y) in I. Depending on the function, we assume our hand object at the position with the highest or lowest value. In Figure 4.27 we compare the results of both methods. In our simple example case, both methods deliver a the correct result – a perfect match at $(0, 0)$. We get a more complex result (a matrix of coefficients) when we apply these methods to our hand recognition problem, as can be seen in Figure 4.28.

4.3.3 Heuristics and certainty

Channel limitation We can apply the above techniques to image recognition. Our detector matches the template $T(x, y)$ with the base image I. The detector

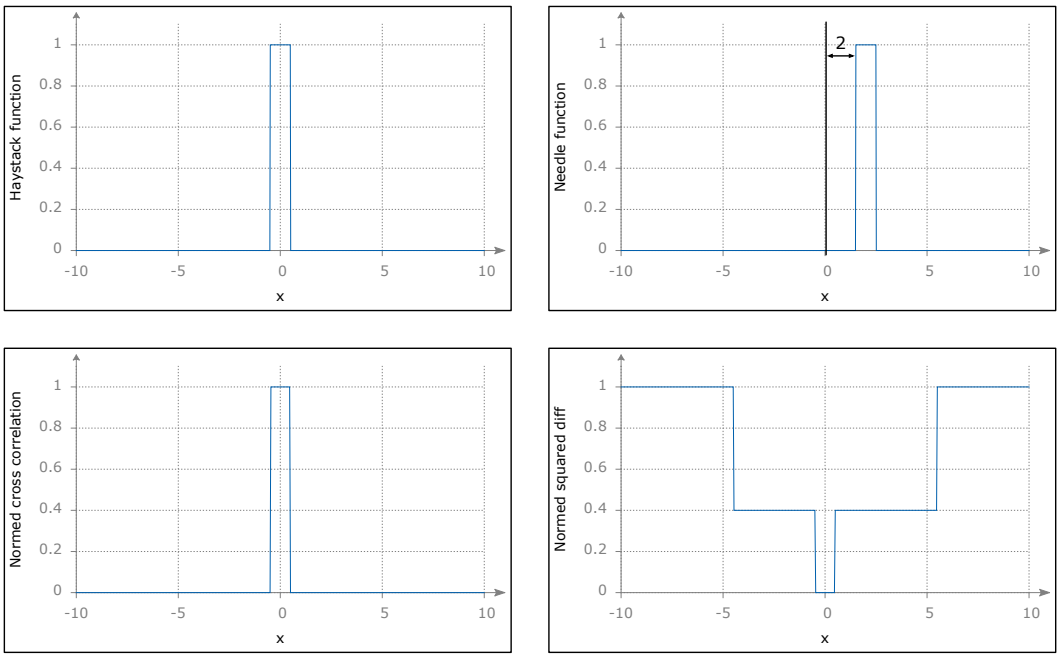


Figure 4.27 – *A qualitative comparison of the two likelihood models. We look for the best position of the needle function in the haystack function. Top left: The* haystack *rectangle. Top right: The* needle *rectangle (which has an offset of -2 from the origin). Bottom left: The normed cross correlation of both rectangles. Note that it has the exact shape of the haystack image. The best location is (0,0). Bottom right: The Normed squared difference function. The best match is also at the origin.*

receives both images in the RGB color space. By default we utilize only the red channel for this detector and use the normed cross-correlation metric. We dismiss the other channels which makes computations easier. This combination has delivered the best results in our tests. We can obtain the 'needle' automatically at runtime by using the detection results of other filters as described in the next chapter.

Edge detection The edges of the fingers are a unique characteristic for a hand object. Consequently we run a fast edge detection on the red channel to improve our detection results (see Figure 4.29). With this preprocessing step we could avoid some common error cases where the hand was not correctly detected in complex background and lighting conditions. One such case was the detection of the hand in front of the face. This scenario has proven difficult for all position detectors: CAMshift, Haar and shape. With edge detection the shape filter can handle this situation better than the other two.

Confidence The bigger the difference between the estimated hand position, defined as the minimum or maximum of our likelihood function R, and the median

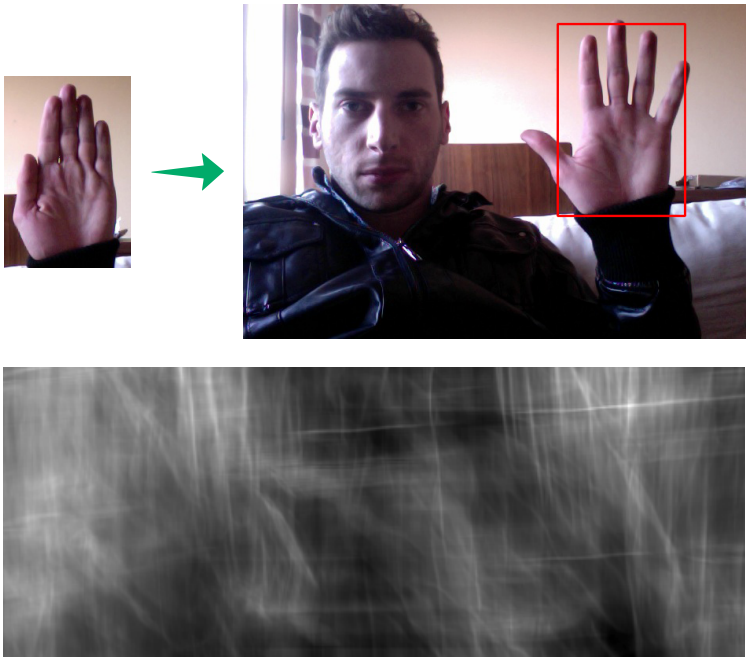


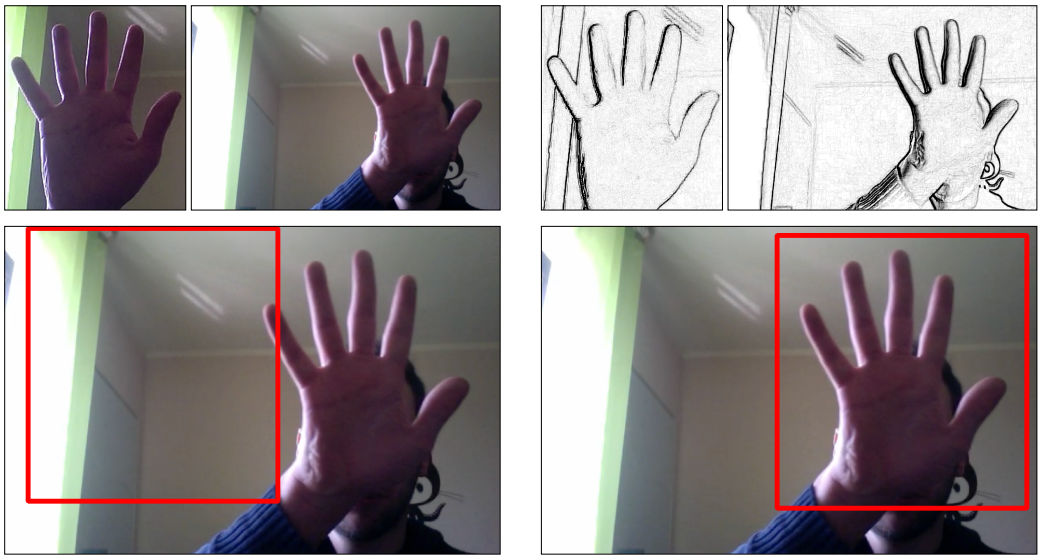
Figure 4.28 – *Top: An example for a successful match. Note that the fingers are not spread apart in the template image. Nevertheless the correct position was found (although the detected boundaries are those of the template image). Bottom: The result of the normed squared difference method. The bright white spot in the top right corner of the image marks the best match.*

value of R, the better our detection result. Conversely, the closer the values of R are together, the harder it gets to make an assertion of the correct hand position. Many hand positions could be equally likely. We use this simple heuristic to determine the confidence of our detection. Our implementation uses NumPy for Python:

```
1 median = np.median(R)
2 conf = (minmax - median)/(minmax)
```

4.3.4 Strengths and weaknesses

The filter is extremely vulnerable to varying hand sizes. It is of not much use when the hand is moving away far from the camera or very close to the screen. We do not scale the template image at every run since this would currently be be too expensive for a realtime environment. Instead, we can observe an enormous drop in the confidence of the result. Thus, a simple countermeasure would be to restart the detector once the confidence falls below a threshold for a certain amount of time.



(a) *An unsuccessful match using color images*

(b) *A successful match using prior edge detection. (We inverted and thresholded the image for better visibility)*

Figure 4.29 – *Using edge detection we can eradicate a common error cases where the hand would not get detected in front of the face.*

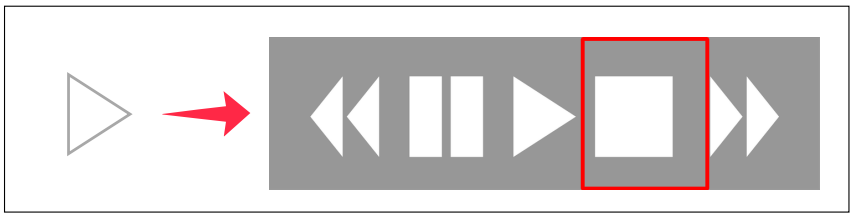
The template image is crucial for the quality of the search result. If there is too much background information in the template image, the number of false positives is increasing. Figure 4.30 shows a typical error case where the background information of the template image (the white area around the play button in that case) has more influence on the search result than the actual object (the play button), which leads to a false positive. As mentioned before we remedy this behavior with a fast edge detection preprocessing step (see Figure 4.29). With that we increase the number of correct detections significantly.

By default it won't detect tilted hands. This feature could be added by rotating the template image by a few degrees in each direction and comparing it with the haystack image. We did not pursue this matter since it would be computationally expensive and the success of the approach would be questionable.

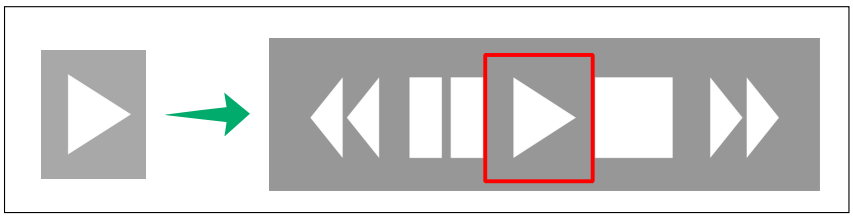
The shape detector is invariant to light changes and complex backgrounds. Especially on skin-colored backgrounds or when the hand is in front of the face, we observed good results. Like CAMshift it requires an initial hand position which can be retrieved automatically from the output of other detectors, such as Haar.

4.3.5 Configuration options

We allow to change the correlation function at runtime. Also, new template images can be specified at any time. We additionally provide a stock hand image, if no prior hand position was specified.



(a) *The area around the template is white. Therefore it gets placed incorrectly at the area with the most white pixels*



(b) *A correct match. The background color of the template image matches that of the haystack image*

Figure 4.30 – *A typical error case of the shape detector. All pixels of the template image are of equal importance to the matching process. This means that the background can be more decisive than the foreground.*

4.3.6 Conclusion

The Shape detector is mostly used to cover some of the edge cases where the other two detectors, Haar and CAMshift, have specific weaknesses. Namely it is an enrichment in situations with changing lighting conditions and complex, skin-colored backgrounds.

4.4 Skin detector

Finger detection based on skin color segmentation is an approach which dates back at least to the beginning of the millenium. Many surveys focus on skin detection methods (see [Vezhnevets, Sazonov, and Andreeva, 2003]). Because of its computational simplicity it is appropriate for realtime applications. Based on the estimated hand positions proposed by the three position detectors we presented earlier, we use skin detector for the finer hand details such as its contour and finger recognition.

4.4.1 Informal description

In order to design a versatile gesture interface we need to determine hand features such as the number of fingers shown or the hand posture. We already presented three ways to determine the hand position. Based on that information we apply color segmentation on the hand region. After removing noise and getting the biggest cohesive area which is assumed to be the hand contour, we find the gaps between the fingers by calculating the convex hull around the hand contour and looking at the defects between the hull and the hand contour to make conclusions about the fingers shown (see Figure 4.31).

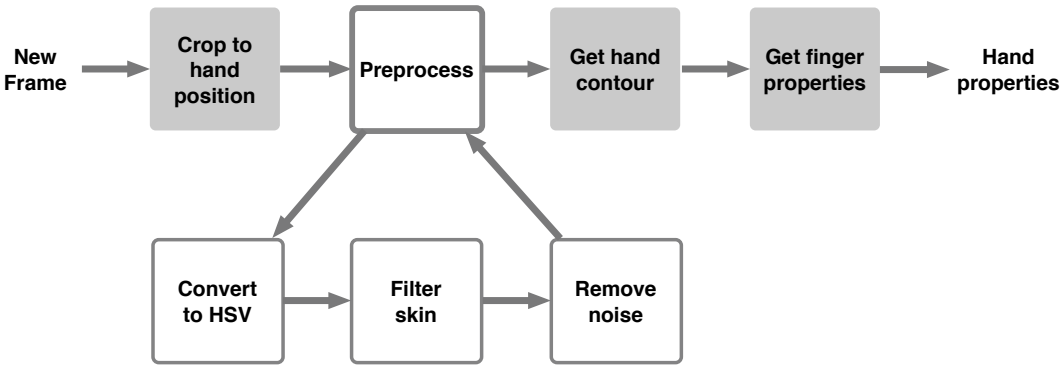


Figure 4.31 – *A slightly simplified schematic of the skin filter.*

4.4.2 Implementation

Selecting a color model An important first step of the skin detection process is the selection of a color model. The obvious choice might be the *RGB* color since this the color space the camera images were taken in. The success rates for *RGB* color segmentation are stated as 80-90% (see Vezhnevets, Sazonov, and Andreeva [2003]). When using this color space for skin segmentation a huge disadvantage is the fact that the color and the brightness attributes are not orthogonal. Changing one of them affects the other [Jordao, Perrone, Costeira, and Santos-Victor, 1999].

But there are many alternatives to choose from. Prominent candidates are *HSV* [Zarit, Super, and F. K. H. Quek, 1999], *YCbCr* (used by Tripathi, V. Sharma, and S. Sharma [2011]) and *YUV* [Marques and Vilaplana, 2000] to name a few.

A good choice should simplify the task of separating skin tones from other colors. Our research has shown that the two most popular variants are still *RGB* and *HSV*. We have settled on *HSV* because it produced slightly better results than *RGB* for *our* use case based on our own observations during tests and studies on skin segmentation (see [Shin, Chang, and Tsap, 2002] and [Zarit, Super, and F. K. H. Quek, 1999]). Also the intuitiveness and convenience of the colorspace components as well as the separation of luminance and chrominance have played an important roll in our decision.

As Bradski [1998] noted, a single skin color model can be successfully used for most human beings in HSV color space. The hue value won't vary much, only the saturation and the darkness needs to be adjusted individually (compare with Figure 4.32).

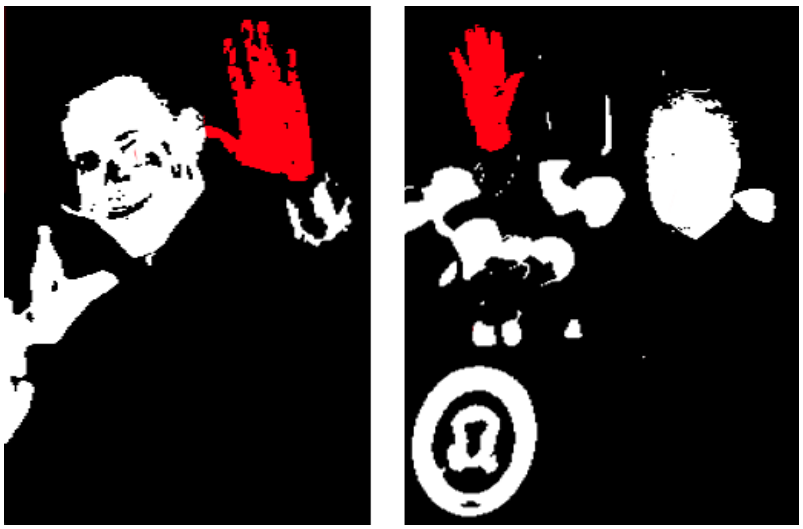
The *HSV* color space The *HSV* color space (also sometimes imprecisely referred to *HSB* or *HSL* in literature [Wikipedia, 2013d]) separates three dimensions of color: the *hue*, the *saturation* and the *value* (also called *intensity*).

> Hue-saturation based colorspaces [are] based on the artist's idea of tint, saturation and tone. Hue defines the dominant color (such as red, green, purple and yellow) of an area, saturation measures the colorfulness of an area in proportion to its brightness [Poynton, 2006]. The "intensity", "lightness" or "value" is related to the color luminance.
>
> *Vezhnevets, Sazonov, and Andreeva [2003]*

One can think of the color space as a cylinder (compare with Figure 4.33a). The hue channel is represented by the angular dimension. It has a degree value between $0\,^\circ$ and $360\,^\circ$. For instance, a purely red color tone has a hue of $0\,^\circ$, green is at $120\,^\circ$ while blue is located at $240\,^\circ$. All other colors can be represented as a combination of these three (e.g., yellow lies between green and red). The saturation is defined by the radial value of the color. The closer the color is to the middle of the cylinder the more white gets mixed with the pure color. Finally, the "value" or intensity of a color defines makes a color darker while preserving the hue and the saturation. We use only the portion marked in Figure 4.33b for skin color detection.

Color conversion and preprocessing We define a color as a vector $\vec{c}^{\top} = \left(C_1, C_2, C_3\right)^{\top}$. Every color c belongs to a color space $\mathfrak{C}$. A color conversion ν is a

(a) *Barack Obama (left) and George W. Bush*



(b) *Skin-colored pixels are white. The hand is marked with red.*

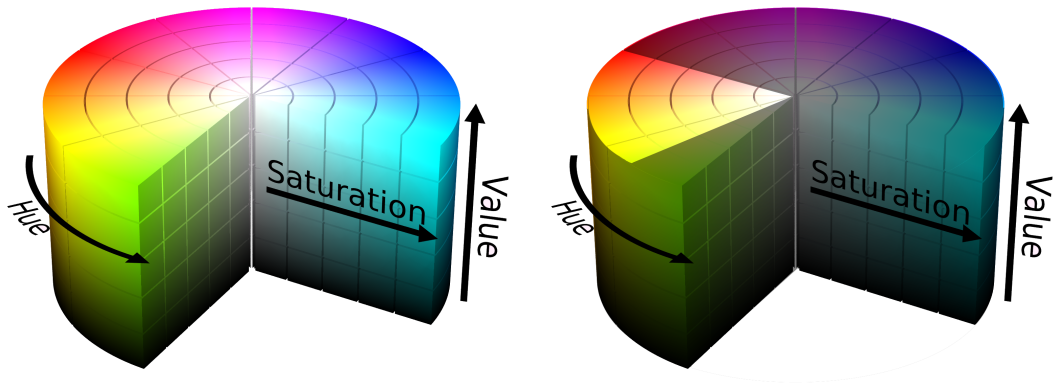
Figure 4.32 – *We use the same settings for the HSV colorspace to extract both hands in the above sample. The hand is marked with red. Sources: Barack Obama by Troy [2008], George W. Bush by Doss [2005]*

bijective function with the following property:

$$v\colon\ \vec{c} \leftrightarrow \vec{c^*}$$

where $\vec{c^*}$ is a color in $\mathfrak{C}^*$.

We convert our input image from *RGB* to *HSV* using the following computations:



(a) *The HSV cylinder*

(b) *The part of the hue channel used for the skin color extraction*

Figure 4.33 – *Skin color extraction visualized using the HSV color cylinder*

$$
\begin{aligned}
H &= \arccos \frac{\frac{1}{2}\left((R-G)+(R-B)\right)}{\sqrt{\left((R-G)^2\right)+(R-B)\,(G-B)}} \\
S &= 1-3\frac{min\,(R,G,B)}{R+G+B} \\
V &= \frac{1}{3}\,(R+G+B)
\end{aligned}
\tag{4.1}
$$

Test framework One fact that needs to be considered when working with an image feed from an unspecified video source, is the brightness level (white balance) varies with every camera model. Therefore, a framework was built to adjust and test different settings for saturation and brightness on a variety of real-life situations with difficult lightning conditions (see Screenshot in Figure 4.34). We used a set of three different cameras [4] to take photos from realistic tracking situations. We loaded the images into our program which arranged them in form of a grid. Using a toolbar depicted in Figure 4.35 we could adjust the *HSV* values and see the detection result in form of the detected hand contours in realtime.

[4]We used three cameras for the test: The builtin iSight camera of a MacBook Pro, Mid 2010 (Resolution: 640x480px), an external Logitech C170 (Resolution: 1024x768) and a Panasonic Lumix DMC-ZX1 (5 Megapixel, 1280x720px video).

Figure 4.34 – *Using our framework to test and configure the Skin filter. Filtering for the skin colored hue component in various light settings. The settings can be adjusted at runtime. The photos above shows a wide variety of usage examples with most different lighting conditions, shadow projections and saturation values. All of the hands in the examples have been detected successfully with modified settings. For this test we used: Hue: 3-100 degrees, Saturation: 23-68 percent, Value: 28-88 percent.*

The results are stored in self-contained configuration files, and are subsequently used by the Skin detector.

Extracting the hand contour To separate our hand from other skin colored regions we use a similar approach to Pulkit and Atsuo [2012]. First, we filter the range of the hue values to detect the skin color (see Figure 4.33b). This is a state of the art approach [Stergiopoulou and Papamarkos, 2009]. Then we erode/dilate the detected skin mask to remove noise (we perform an *opening*, to be more precise) and take the largest blob (which we assume to be our hand). Our implementation builds on `findContours()` and `approxPolyDP()` functions from *OpenCV*. We show different skin blobs of pictures taken under varying lighting conditions in Figure 4.34. We used the settings of Figure 4.35 for all of these photos.

Convex Hull As a last step we detect the fingers by counting the defects of the convex hull around the obtained hand contour. Mathematically, a convex hull is

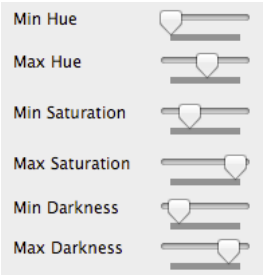


Figure 4.35 – *One of the general-purpose HSV settings we use for hand extraction. Since we only detect skin colors in a narrow region around the hand we can reduce the risk of noise based on skin-colored backgrounds. Hence, the settings can be quite generous.*

a set of X points is the smallest convex set that contains X. We can exploit the fact that the fingers of our hand will be stretched away from the palm when we perform a gesture and that the fingertips all lie on the convex hull at that moment. The spaces in between the gaps can be interpreted as fingers. Figure 4.36 depicts a segmented hand contour. The three points s, f and e stand for *start*, *far* and *end* respectively. f is the point with the greatest distance from the hull. The other two points, s and e are two points closest to f which are still on the hull.

We store the two lines which characterize each finger in a separate data structure. The red lines, as shown in Figure 4.37, are our final result.

4.4.3 Heuristics and confidence

We classify a defect of the convex hull as a finger based on three criteria:

Finger length We check if the hull defect is "big" enough to be a finger by comparing its length to the overall length of the image dimensions. Only if this ratio is within a specified range, we accept it.

Finger angle The angle between $\overline{sf}$ and $\overline{fe}$ (compare with Figure 4.37c) must be within a certain range for valid fingers. By default we accept angles between $10\,^\circ$ and $90\,^\circ$.

Finger orientation The hand is properly orientated if the fingertips are above the point closest to the palm (or just slightly below for the thumb).

We evaluate this property based on the following code:

```
start, end, far, depth = self.get_finger_points(hand_cnt, defect)
# Compare the y-coordinates of the points
return (far[1] - start[1] > self.finger_orientation_thresh) and \
       (far[1] - end[1]   > self.finger_orientation_thresh)
```

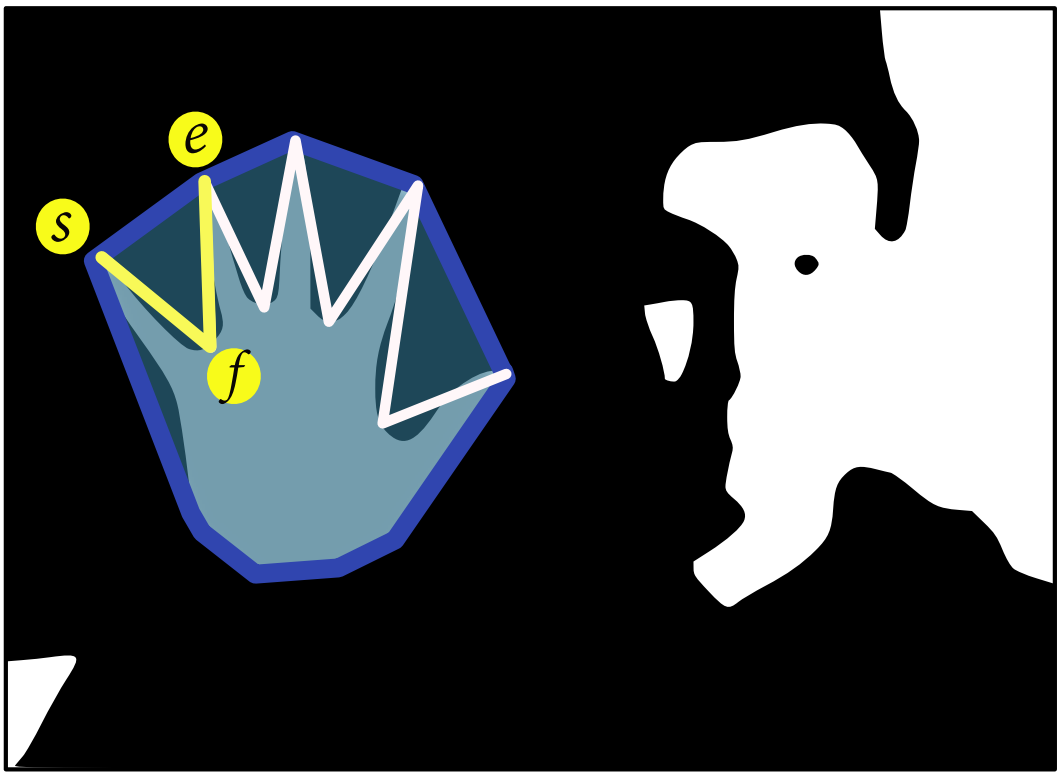


Figure 4.36 – *Detecting a finger from a defect in the convex hull. Vectorization of an actual skin segmentation (heavily dilated) with an estimated convex hull around the hand.*

where `self.finger_orientation_thresh` is set to 20 by default.

HSV color range On training images, the skin filter tries a number of default configurations in order to find the best settings for the current lighting situation.

4.4.4 Strengths and weaknesses

Like any other process, the skin filter comes with a specific set of flaws. We adjusted the parameters carefully for our use case but especially in dim environments with skin-colored backgrounds the hand segmentation has some issues. One reason is that low light conditions lead to a lot of image noise (as can be seen in Figure 4.38). As a countermeasure we could adjust the settings so that we allow more variance in the saturation and intensity channels with the risk of detecting more false positives. With respect to our goal of best possible robustness, we did not pursue this idea. Like Bradski [1998] we ignore color values with low brightness values as a result.

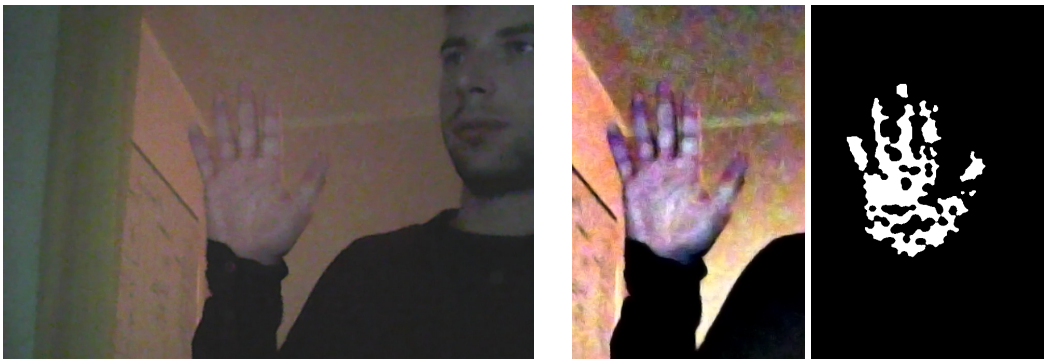
The skin detector has proven robust even with difficult lighting conditions (e.g., at nighttime). Also it is invariant to background changes and hand rotations. Other projects which use skin detectors reported problems where the wrist was

(a) *Sketch of hand in front of the camera*

(b) *Calculating the convex hull*

(c) *The convexity defects are interpreted as fingers*

Figure 4.37 – *Finger detection using skin-colored image regions.*



(a) *Noisy image with skin colored background*

(b) *Emphasized*

(c) *Our best skin segmentation*

Figure 4.38 – *A failed hand segmentation. Detecting the hand in this case is also non-trivial for humans*

detected as a part of the hand thus leading to incorrect finger detections. A solution was to require the user to wear a black belt around the wrist [Ren, Yuan, and Zhang, 2011]. This problem can be avoided with our approach since we only detect skin colors at the hand position we determined before as seen in Figure 4.39.

4.4.5 Configuration options

The thresholds for our heuristics as well as the the intensity of the initial smoothing, erode and dilate steps can be adjusted. This way, a user can configure the settings to his own needs in case the hand properties are not accurate.

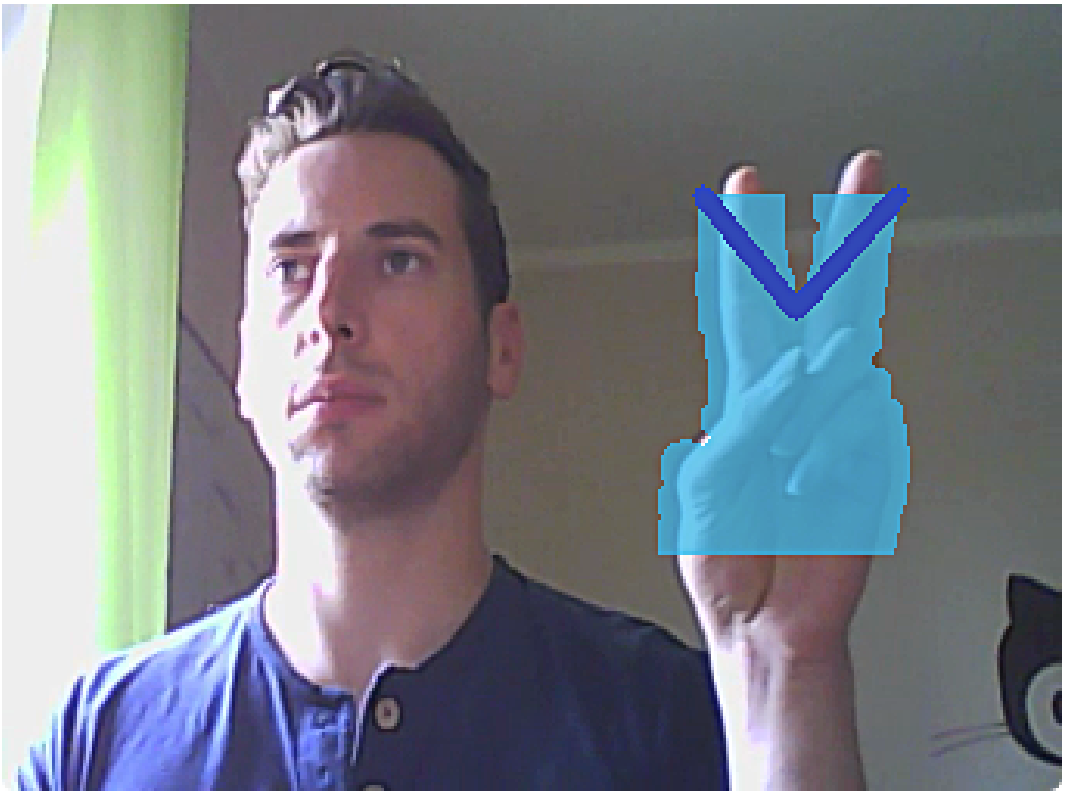


Figure 4.39 – *A common error case where the wrist is detected as a part of the hand can be avoided with our approach since we only detect skin colors at the hand position we determined before.*

4.4.6 Conclusion

The skin detector has proven very robust for hand feature detection. As it is crucial for the robustness of our system, we carefully adjusted the values for most real-life conditions. We also improved the detection speed throughout the development of the program. Combined with the aforementioned filters, we now have a system which can detect the hand position as well as the hand posture and fingers.

4.5 Summary

We have now introduced all detectors within our framework. Others could be added with ease as long as they adhere to the detector interface specified before (see Section 3.5.3).

Notable additions could be various keypoint and feature detection algorithms like SIFT [D. Lowe, 1999], SURF [Bay, Tuytelaars, and Gool, 2006], BRISK [Leutenegger, Chli, and Siegwart, 2011], and FREAK [A. Alahi, R. Ortiz, and P.

Vandergheynst, 2012]. During our initial tests we had issues with the realtime-capability of these detectors which is why we didn't integrate them in the first place. Another possibility would be to add a background subtraction mechanism to our feature cascade but some prior tests did not yield any improvements on the accuracy of the detection and another detector would decrease the overall performance of the system and lead to higher memory usage.

A summary of the observed strengths and weaknesses of each algorithm is given in Table 4.2.

Each algorithm is based upon certain assumptions about the hand properties. Table 4.3 gives an overview of the principles behind each detector.

Detector	Advantages	Disadvantages
Haar [P. Viola and M. Jones, 2001]	• Robust on simple backgrounds • Invariant to lighting changes • Offline training • Adapts to hand size • Fast	• Many false detections on complex backgrounds • Limited support for hand rotations
CAMshift [Bradski, 1998]	• Robust on complex backgrounds • Invariant to lighting changes • Adapts to hand size and rotation • Fast	• Initial hand position required • False positives with other skin-colored objects nearby
Shape [Belongie, Malik, and Puzicha, 2002]	• Robust on complex backgrounds (especially in front of the face) • Invariant to lighting changes	• Initial hand position required • Does not adapt to hand size and rotation • Relatively slow
Skin [Vezhnevets, Sazonov, and Andreeva, 2003]	• Adapts to hand size and rotation • Robust on complex backgrounds • Invariant to lighting changes • Fast	• Prone to image noise • False positives with other skin-colored objects below hand

Table 4.2 – *A summary of specific detector strengths and weaknesses*

Filter	Based on	Depends on
Haar	Object features	Cascade file
CAMshift	Histogram and Back Projection	Tracking window
Shape	Mathematical affinity	Template image
Skin	Regions of same color	*HSV* color range for skin

Table 4.3 – *Detector dependencies and foundations*

4.6 Bootstrapping

> *As we have seen, most object detectors require some initial information about the object being tracked before they can be used. All but Haar need an initial hand position to get started. Either we need to retrieve this information manually (i.e., by selecting the hand in the image) or we pursue an automatic take on the problem. Since we aim for an unsupervised system, we use a "bootstrapping" process to start the detectors based on their dependencies on other detectors. An example of the final bootstrapping process from the perspective of the user is shown in Figure 4.40.*

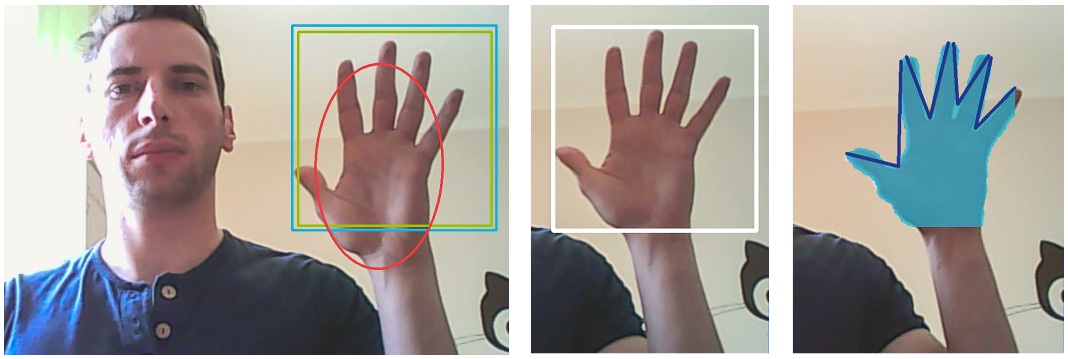


Figure 4.40 – *Snapshots of a successful bootstrapping process.*

Course of action Figure 4.41 shows the principle structure of the bootstrapping process and the dependencies between the detectors. At first we require the user to show his hand and run our Haar detector on the camera input. The hand must be kept still for the detection to work. In this case the consecutive bounding boxes detected by Haar will be overlapping.

This is a mandatory initial calibration step which requires no intervention from the user. After Haar has found the hand object and reached a confidence level which is above `threshold_1` (which means we have a reliable estimate of the hand position), we use its output (the estimated hand position) to *bootstrap* the other two position detectors – CAMshift and Shape. After that we wait until

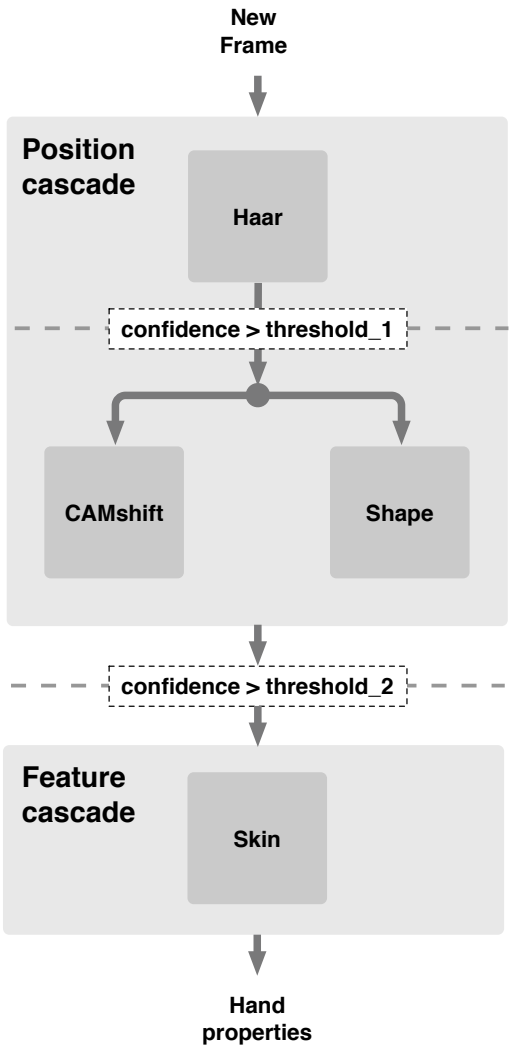


Figure 4.41 – *Schematic of the bootstrapping process*

the confidence for the combination of all three position detectors raises above `threshold_2` which allows us to start the feature detector (skin) to find the hand contour and the fingers.

This is an iterative process, which however completes quite fast and does not interfere with the actual human-computer interaction session. Under normal conditions (most importantly the camera footage may neither be *extremely* dark nor bright) the whole process will take a few dozen frames. Such a "filter cascade" allows us to execute the final training on the actual data, hence no discrepancy arises between the training and interaction data sets.

It would be possible to gather all necessary data before program use and skip the startup sequence. As an example, one could imagine to start the shape detector with a stock image of a human hand, but the obvious problem would be, that the detectors need to make some general assumptions about the user of the system

(e.g., about the hand color and shape) and the environment (lighting conditions, background complexity and so on) which are unlikely to be entirely correct. For that reason we rely on this algorithmic approach. With the bootstrapping technique we are able to track the actual hand of the current user. This ensures fast and consistent operation.

Visual feedback The user can watch the actual bootstrapping process as it happens (Figure 4.40). At first, there is a red rectangle around a hand estimate. This rectangle gradually turns into green as the confidence of Haar increases. Finally, when it is shining green, two new rectangles appear: one for CAMshift and the other for Shape. Shortly after that we can see the hand contours and fingers, marked with blue. The system is now ready to use.

4.7 Consolidation of detector results

Each position detector calculates an estimated hand position and a probability that the estimation is correct. The combination (weighted by the probability) of all estimated hand positions gets used to predict a more accurate hand position. This step increases the overall stability of the recognition process.

Calculating the estimate We combine the results of our detectors with the following formula:

Let $h_i = \langle r_i, p_i \rangle$ be the detection result of a detector $d_i \in \mathfrak{D}$, where $r_i = (P, Q)$ is the estimated rectangle around the hand defined by its upper left corner P and its lower right corner Q and p_i is the confidence of the detector. If two or more rectangles r_i are overlapping by a certain percentage (50% by default), we calculate our hand estimation with

$$e = \frac{\sum_i r_i p_i}{\sum_i p_i}; \langle r_i, p_i \rangle \in W$$

where $W \subset \mathfrak{D}$ is the set of all overlapping rectangles.

If $e > t$, where t is a threshold we have set to 0.7, our final hand estimate is e. If $e \leqslant t$ or the rectangles don't sufficiently overlap, our estimate is the hand position of the detector with the highest confidence.

The probability for our estimate is the arithmetic mean of all overlapping rectangle probabilities

$$\overline{p} = \frac{\sum_i p_i}{|W|}.$$

Examples We consider the situation in Figure 4.42a. The results of two detectors are overlapping (indicated by the hatched area) but not by the required percentage so we pick the leftmost rectangle for our estimate, which has the maximum confidence of all three.

In the second example in Figure 4.42b we see that all three detectors agree on the general hand position. The rectangles are overlapping by a high percentage bigger than the threshold, so we calculate the hand estimate as follows.

We have

$$\begin{aligned} r_1 = (P, Q) = ((97, 90), (248, 290)) ; p_1 = 0.7 \\ r_2 = ((151, 71), (300, 271)) ; p_2 = 0.2 \\ r_3 = ((152, 109), (303, 308)) ; p_2 = 0.1 \end{aligned}$$

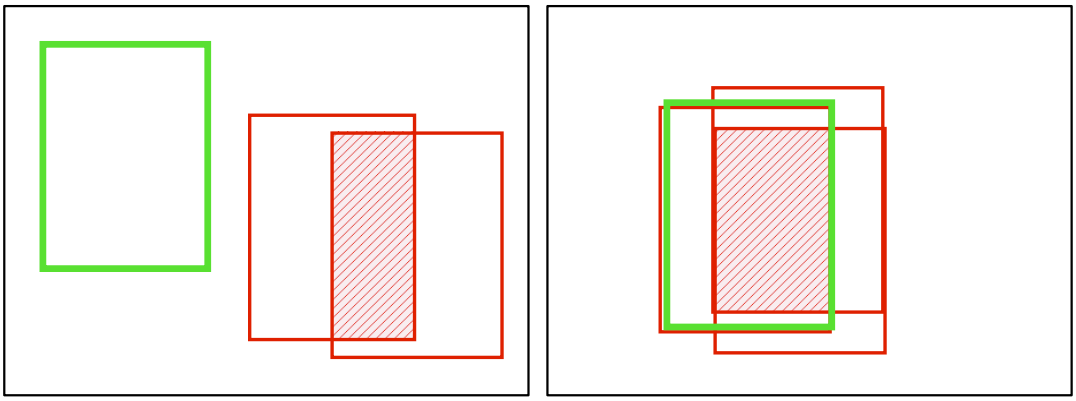
So we get

$$e = \frac{0.7\,((97, 90), (248, 290)) + 0.2\,((151, 71), (300, 271)) + 0.1\,((152, 109), (303, 308))}{0.7 + 0.2 + 0.1}$$

$$e = ((113.3, 88.1), (263.9, 288)) \approx ((113, 88), (263, 288))$$

Which we deem a good estimate for the actual hand position. We assign $\overline{p} = 0.\overline{3}$ for the confidence. This low value is a result of the low certainty of two out of three detectors.

Nota bene This formula is based on a simple heuristic: We trust the result of the most confident detector as far as there is no striking consensus (expressed by overlapping rectangles) between the majority of detectors. Other heuristics are equally valid. For instance we could always pick the *highest ranked* rectangle or calculate the minimum surrounding rectangle around all results. We tested several versions for our estimations (see Appendix C.3) and settled on the proposed method.



(a) *Since the area of the both rectangles on the right side are not overlapping by the required percentage we pick the leftmost rectangle, which has the maximum confidence of all three.*

(b) *The three rectangles are overlapping. We estimate the actual hand position to be in the middle of all three with a strong tendency to the leftmost result which has the highest confidence value.*

Figure 4.42 – *Two different possible results of all three position detectors. We use a heuristic to determine a refined hand estimate in both cases*

4.8 Enabling and disabling detectors

We register all available detectors in a `dictionary`. This allows us to enable or disable each detector with a simple `use` flag (even at runtime). We use reflection to start the detectors: The constructor specified by the name of a dictionary entry gets called, receiving the current camera frame as well as the current hand estimate (if available) and optional configuration options as parameters. We store the created detector as the value of `instance`. All dependencies declared in `deps` must be satisfied in order to start a detector.

```
1 self.detectors = {
2
3   "camshift": {"type": DetectorType.POS,
4     "use": False,
5     "constructor": "DetectorCamshift",
6     "deps" : ["haar"],
7     "instance": None},
8   # ...
9 }
```

4.9 Summary

We presented a cascade of detectors, or image filters, which can extract a hand position and its features (such as the number of fingers) from an input image. The system is adapting to the input and gets better over time.

The cascade consists of two different kinds of detectors, position and feature detectors.

Each position detector calculates an estimated hand position and a probability that the estimation is correct. The combination (weighted by the probability) of all estimated hand positions gets used to predict a more accurate hand position.

Afterwards the properties of a hand object will be determined by the feature detectors.

With that information we can create a versatile interface as seen in the next chapter.

Chapter 5

Gestures & Actions

Now we are able to analyze hand postures as well the properties of fingers. Using that information we can define and subsequently execute actions based on the user's intent.

In the preceding chapters we laid the groundwork for a solid and flexible gesture interface. We use our framework to recognize static and dynamic gestures. As we will see, each of them serves a different purpose.

5.1 Architecture

An overview of the gesture subsystem is given in Figure 5.1. The tracker module sends the hand data to a gesture module. Here we define the gestures in a high-level data structure (see Listing 5.1). If we detect a matching gesture, we will notify the tracker module about it. The execution of actions can be enabled or disabled with a shortcut. If it is enabled, the action module which will execute it. An action could be an arbitrary keyboard command (including special keys and any shortcuts) or mouse movement.

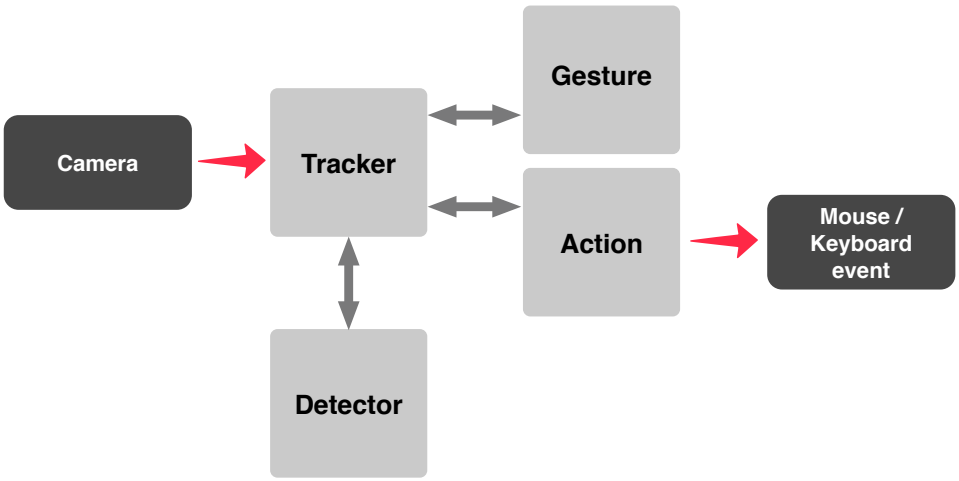


Figure 5.1 – *A high level view of the system. We will extend this schematic as we progress*

5.2 Gestures

Using Python, a user can define his own gestures. In the `Actions` module, he first specifies a name for the new gesture. As an example, we named our two available gestures `Actions.EXPOSE` and `Actions.POINTER`. To execute the Exposé gesture, the user has to show five fingers for a predefined duration which can be set in the configuration file. After each successful action, a delay of 5 seconds is necessary until a new Exposé gesture can be recognized. Similarly, the point gesture requires the user to show two fingers. Note that the delay for a consecutive execution is zero. This means that the gesture will be evaluated in every frame, once it was started. This makes movements of objects, like the mouse pointer or application windows, possible. It would also be possible to name the fingers in the hand object datastructure to allow even more versatile hand postures.

```
1 self.gestures = {
2   # Name: {fingers, action, delay between same gesture}
3   "Expose":{"fingers": [5], "action": Actions.EXPOSE,  "delay": 5.0},
4   "Point": {"fingers": [2], "action": Actions.POINTER, "delay": 0}
5   # define other gestures...
6 }
```

Listing 5.1 – *Defining gestures*

Purpose Both gestures are intended to be an *extension* of the user interface, rather than a *reorganization*. This means the gestures shall supplement the interaction with a computer in situation where it seems fitting. For instance the user could lift the hand shortly to show all windows instead of remembering the correct keyboard shortcut to do so. Or he could play an interactive computer game with gesture input, like a card game. The primary goal was, to make these two

gestures work as reliable as possible, but one could easily expand the system with other actions.

Static gestures If a predefined hand posture is recognized over a certain amount of time we trigger a *static gesture*. With our approach we can identify all upright, non-tilted hand gestures: the fist and all combinations of fingers shown. As a proof of concept we declared an easily executable and intuitive gesture which requires no prior training: show five fingers for one second to show the windows of all open applications. We call this gesture *Exposé*. A quick mnemonic could be: "Just as the fingers need to be spread away to toggle the gesture, all program windows shall spread across the display".

Dynamic gestures Gestures which involve hand movements are a flexible way to modify the virtual environment. They consist of three stages: *initialization*, *evaluation* and *completion* (compare with Figure 5.2. To initialize a dynamic gesture, the user needs to show a unique hand sign for a short amount of time. After that, we constantly *evaluate* the hand position. As long as the same gesture is shown in every frame, we move a virtual object across the screen. For our prototype we implemented a mouse-movement gesture. The dynamic gesture is completed when the user stops making the gesture.

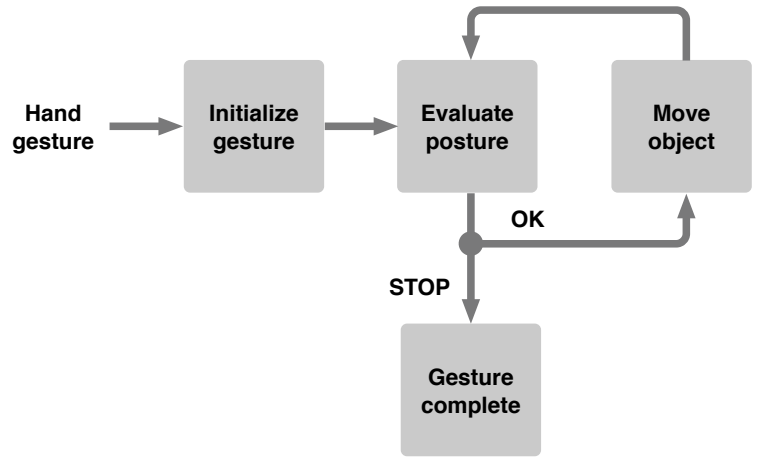


Figure 5.2 – *Executing dynamic gestures*

Gesture delay Most of the time the gesture system will be idle. In order to prevent a common source of false positives where a gesture is executed repeatedly within a short period of time (in a timespan of a second) we require a delay of a few seconds between two consecutive gestures of the same type.

5.3 Actions

Keyboard control Static gestures execute keyboard commands. With `PyUserInput` [Barton, 2013] we found a cross-platform library to trigger keyboard events but during our tests we found that the commands get executed with a noticeable delay which affected user experience. Therefore we built a wrapper around `osascript`, a commandline application for *AppleScript*. With it we can trigger any key event (as well as key-combinations for shortcuts). Note that this tool only supports Mac OS X. We can still fall back to `PyUserInput` on other operating systems. A user can define a keyboard event using the syntax of Listing 5.2.

```
# Switch to previous application
mac_keyboard(["command"],"tab")

# Mission Control / Expose
mac_keyboard([],"F1")

# Multiple modifier keys
mac_keyboard(["shift", "command"],"p")
```

Listing 5.2 – *Examples for keyboard shortcuts*

Mouse control We experimented with two types of mouse control. The first way we call "absolute movement" takes the *last* frame as a reference position for the movement. This control option is suitable for *swipe gestures* across the screen. Its behavior is comparable to touch interfaces: the user must move his hand constantly to execute a command.

The other way, "relative movement" always takes the *first* frame as a reference point. Using this controller feels more like an analog joystick (compare with Figure 5.3).

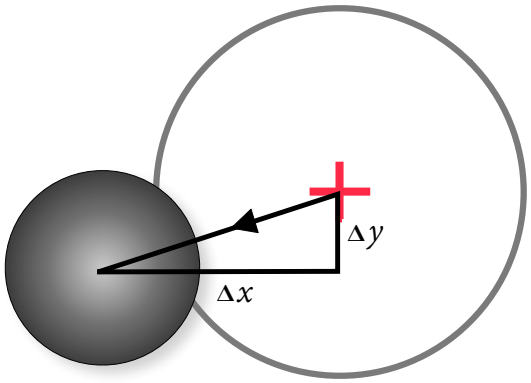


Figure 5.3 – *Displacement of a Joystick*

Regardless the movement type – absolute or relative – the principle to determine the movement direction stays the same. The cursor is always heading into the opposite direction of the hand movement. That is, if the hand was last seen further to the left, the mouse moves right and similar for the other directions. More precisely we calculate the x and y delta between the current hand position and the reference position. We determine the axis by comparing the absolute values of the two deltas. If $|x| > |y|$ we move horizontally. If the delta is positive, we move left and vice versa.

In *relative* mode, we move the mouse by a constant interval during each step, regardless of the actual delta value. This makes the mouse control more accurate. In *absolute* mode we use the delta value itself which makes fast mouse movements possible.

Figure 5.4 shows a typical mouse movement sequence. The red cross is the reference point.

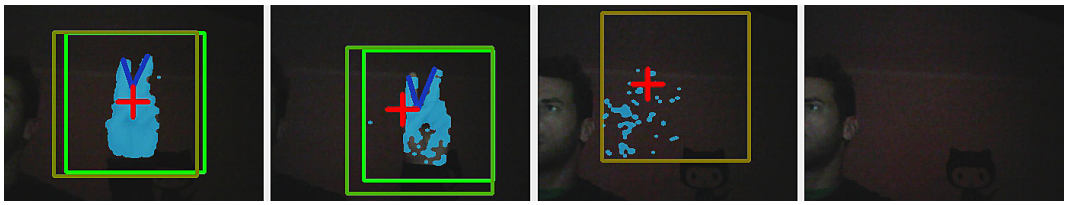


Figure 5.4 – *The mouse gesture. From left to right: Initializing a gesture and getting the reference point (the red cross). Moving the hand to the right moves the mouse, too. After we take the hand away the detector confidence decreases. No valid detections found.*

We use the aforementioned `PyUserInput` for mouse events. It supports three mouse buttons and holding buttons while moving and works on all platforms.

5.4 Review

Within our system we use the combination of hand position and fingers to describe a gesture. We distinguish static from dynamic gestures. Static gestures are coupled with simple actions like triggering a keyboard event. We can model the hand movements to describe dynamic gestures like mouse movement. We implemented a number of sample applications which each highlight a specific feature of our gesture system.

Chapter 6

Evaluation

We specify a test environment and evaluate our system on a set of pre-recorded videos of varying complexity.

How do we evaluate a vision-based system with scientific rigor? Is it even possible to test all of its aspects with verifiable standards? After all, abstract terms like "robustness" and "reliability" are subject to interpretation. Therefore we need to specify our testing environment. To test robustness, we recorded a frame sequence showing our typical gestures in various lighting conditions, with varying complexity of the background and at various speeds. Based on these criteria we classify each sequence into a difficulty class and assess the confidence of the detectors in each video frame. For the realtime tests, we measure the mean processing time per frame for each detector and analyze the results with statistical methods.

Boxplots We use *boxplots* as a tool of choice to visualize the distribution of the confidence and for presenting the frame rate measurements for each of the filters and for their combined execution time. The boxplot [Tukey, 1977] is one of the usual methods of statistical data evaluation; it draws a 'box' from first to third quantile, separated by a thick line for the median value. This gives us a visual hint on how 'balanced' the median is between the quantiles. The outer lines, 'whiskers', are extended to the most extreme data point that is no farther away from the box than 1.5 times its interquantile distance. Everything further than whiskers is deemed to be an outlier. Here we can quickly see whether the outer data points on both ends of the distribution are similar and if extreme outliers are present.

6.1 Robustness

For our tests we used ten video sequences. We show a screenshot of each video in Figure 6.1. We categorized the videos based on the criteria in Table 6.1.

Classification	Categories
Background	Simple, Skin-Colored, Complex, Mirroring
Lighting	Overexposed, Underexposed, Normal
Gesture	Exposé, Move Mouse
Gesture Speed	Slow, Fast

Table 6.1 – *Classification of our test videos*

The result of the assessment is presented in Table 6.2. The videos are roughly ordered by difficulty level. The system performed satisfactory except for the videos 6 and 9.

In Video 6 the system is challenged with a combination of problems: a very misleading background with almost the same hue value than the skin color, bad lighting conditions (which causes image noise) and fast hand movement. The situation could probably be saved with an appropriate color filtering and a custom erode/dilate configuration, but as of now, the results of this test-case were non-satisfactory.

In Video 9 the Haar classifier is not able to detect the hand. As a result the other detectors – including the skin detector – can not be bootstrapped. Because of this, the gesture is not recognized.

Frame-based evaluation We make a statistical evaluation of confidence values of all detectors over a large number of frames. The data was obtained after the warm-up, with other words, all filters were configured, bootstrapped, and running.

We evaluate the set of confidence values for all the filters for each frame of the video sequence #3 as boxplots in Figure 6.2. The confidence values are stated relatively, in the range from zero to one. We observe that the distribution of Haar is quite wide – at the beginning of the video Haar has not yet found a hand, thus reports zero confidence. It gradually works up to 1.0 at which point the other two detectors get bootstrapped. They start with a higher confidence level and stay within a narrow upper range of the scale throughout the video.

We observe similar behavior in the other test videos. During normal usage the detection confidence value stays is in the upper third of the scale.

#	Background	Lighting conditions	Gesture	Speed	Result
1	Simple	Normal	Exposé	Slow	+
2	Simple	Normal	Exposé	Fast	+
3	Simple	Overexposed	Move mouse	Slow	+
4	Simple	Overexposed	Exposé	Slow	+
5	Simple	Underexposed	Exposé	Slow	+
6	Skin-colored	Underexposed (Noise)	Exposé	Fast	−
7	Complex	Changing (Moving)	Exposé	Slow	+
8	Complex	Underexposed (Reflections)	Exposé	Fast	+
9	Complex	Underexposed (Reflections)	Move mouse	Slow	−
10	Complex	Normal (Reflections)	Move mouse	Slow	+

Table 6.2 – *Results of the robustness evaluation on sample video data.*

Figure 6.1 – *Screenshots of our test sequences, roughly ordered by ascending difficulty level*

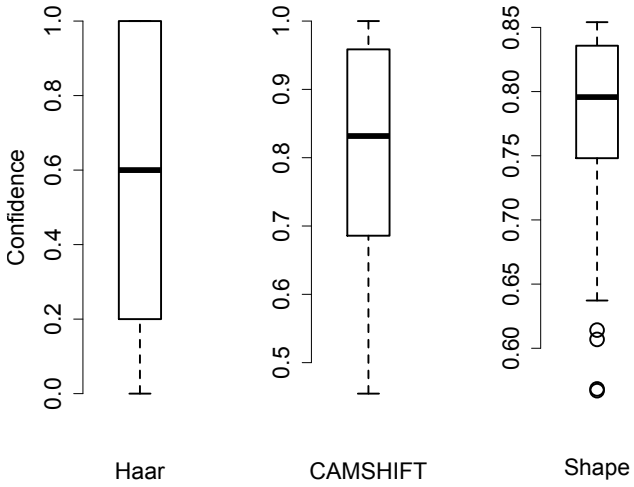


Figure 6.2 – *The distribution of the confidence values for the position cascade. (The skin detector yields no confidence values.)*

6.2 Real-time capability

The real-time capability was achieved using a MacBook Pro with a 2.4 GHz Intel Core 2 Duo and 4 GB of RAM, using Python 2.7.3 and OpenCV 2.4.6.0. In our tests, the system worked fluidly, with approx. 45 frames per second. We noticed no disturbance when the system was working on a live video feed. We used the machine's internal camera and also experimented with an external webcam. The video was sampled at 24 frames per second.

Profiling We statistically evaluate the data from our system.

We benchmarked a typical video which contains various gestures. The video has a length of 25.8 seconds. In this timespan we counted 25149 function calls. For each filter we note an average value, a median, variance, and the minimum of the frame rate. We summarize the results in Table 6.3.

Filter	mean	median	variance	min
Haar	107.74	109.98	331.46	60.51
CAMshift	394.4	403.9	1402.99	161.5
Shape	125.62	130.11	198.81	36.98
Skin	774.4	732.5	48210.8	157.7

Table 6.3 – *Benchmark results. We show the hypothetical frame per second rate for each filter, if it were executed solely.*

Frame-based measurement To support the profiling results from Table 6.3, we show the boxplots of the frame rates for all four filters in Figure 6.3. Because of bootstrapping the filters run for different number of frames. For this reason and due to varying execution times, each boxplot has its own scale. The left part of Figure 6.3 shows the boxplots for the filters separately. The video data originates from the internal camera of a MacBook Pro. We see the high performance of individual filters. The *combined* frame rate is presented in the rightmost part of Figure 6.3. Note, that this is a pessimistic approach, as a sequential execution of all filters is assumed. A version with threads would perform better. Also these values are pessimistic in a further aspect. As some filters process more frames than others, we needed to decide which times to utilize. We took the *worst* times when we had more frames than we could use. Hence, the worst-cumulated plot (Figure 6.3, right) shows the sum of the worst frame rates from Figure 6.3, left.

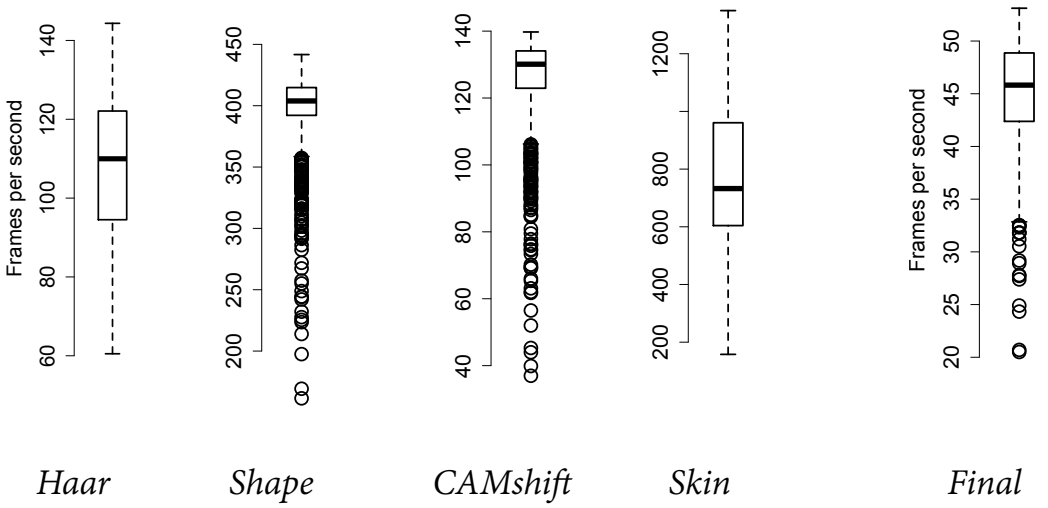


Figure 6.3 – *The boxplots for the frame rate, computed for each method (four plots on the left) and the cumulated worst frame rate (the rightmost plot). We combined the* worst *possible frame rates for all methods – it could be only better in practice. We observe that our implementation is definitely real-time capable.*

We see that the median of the *worst* combined frame rates never falls below 45 fps – on a quite dated machine! In a more favorable setting it is typically higher. Thus, we can confidently say that our approach is real-time capable.

6.3 Use-cases

We tested the suitability of our application for the use-cases we discussed in Chapter 3.1. We can only give some subjective impressions based on our own experiences.

6.3.1 Productivity tool

Assessment If the system shall be of use in a normal work environment as in 6.4, especially the dynamic gestures (moving the mouse or a window) need to be highly accurate. The user is very close to the device (roughly 50cm away from the camera), which means hand movements of a few centimeters should account for mouse movements of several tens or hundreds of pixels across the screen. With a high enough camera resolution (at least around 640×480px) we can achieve this accuracy. Despite our effort to optimize all parts of the system for efficiency, the processor usage was still too high for normal usage. It varied between 30 and 50 percent on the aforementioned MacBook Pro. One solution is to evaluate only a few frames per second. Since a gesture is typically shown for more than

two seconds, the user might not notice any interruptions. For our prototype we implemented shortcuts, which control the delay between each analyzed frame. With that technique, processor usage can be dropped to 10 percent.

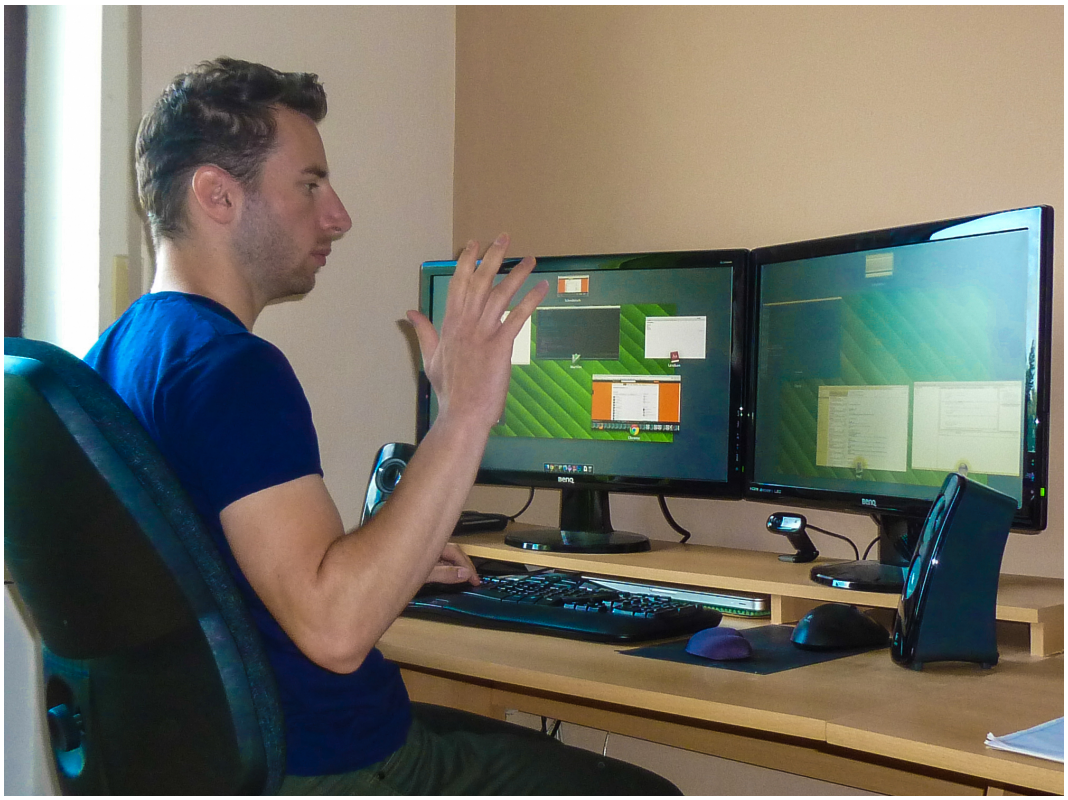
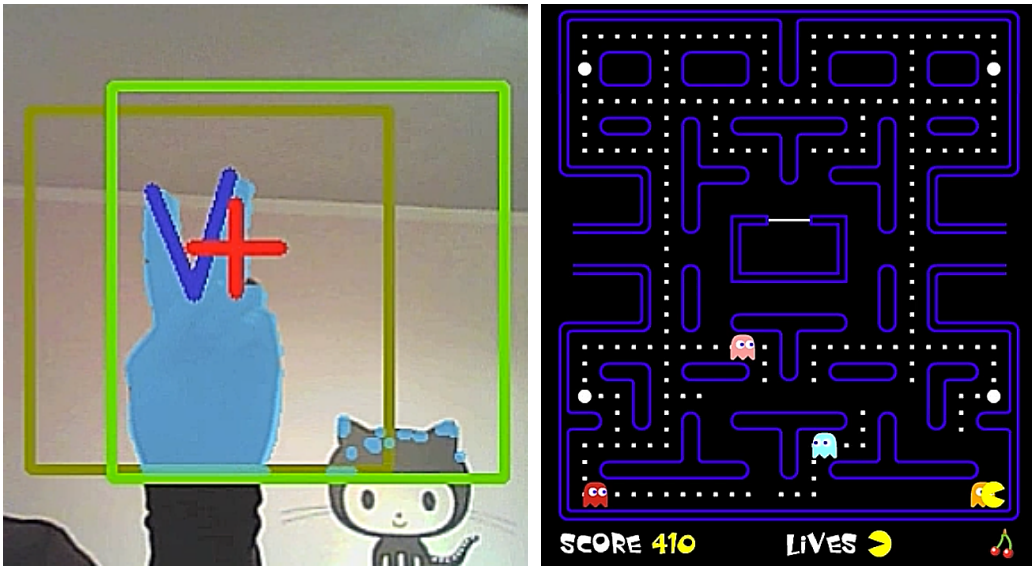


Figure 6.4 – *A home office environment with two computer displays with a webcam in-between*

6.3.2 Game device

Pacman demo For our first game of two games, Pacman , we used the same hardware as above: a consumer laptop and an external Logitech webcam. Pacman is a classic arcade game released in 1980 [Kent, 2001] with just four move commands (left, right, up and down). It is very appropriate for the gaming test. Because we do not own the original game ROM, we played a free browser version available at [Neave, 2013]. We used the arrow keys to control Pacman (see Figure 6.5. The game was modestly playable as we noticed quite some delay between the gesture detection and the key command. We suspect the delay occurs during the communication between Python and the ObjectiveC, framework which is responsible for the key events. Additionally the game was running in a web browser. This could also have caused the delay.



(a) *Gesture input for the Pacman game*

(b) *A free Pacman implementation running in a web browser*

Figure 6.5 – *The game on the right gets controlled by the input on the left. We can see that Pacman gets caught by Clyde right at that moment.*

NVIDIA Alien vs. Triangles Tessellation Demo As a result of the bad performance of the Pacman demo, we deployed a different testing system which runs Windows 7. Since the computer was much more powerful (featuring an Intel Core i7 processor and an NVIDIA GTX 680 graphics card), we also also settled for a graphically involved demo, Alien vs. Triangles (see Figure 6.6). We use the mouse control for free camera movement. During our tests the demo was running fluently even while recording a screencast in parallel.

Assessment The system works accurate enough for most realtime-games. With the right adjustments, one could imagine more complex games like strategy titles. For instance, one could think of a "grabbing" gesture to select units and "drop" them at a target.

Comparison with other projects An objective comparison with the work of others in the field of computer vision – especially gesture recognition – is difficult. The testing environment can vary tremendously for each project. With that in mind we tried to collect confidence values of other projects in Table 6.4.

The project with the most similarities to our own work is Stenger, Woodley, and Cipolla [2010]. As mentioned in 2.4, the system also uses multiple cues for hand detection. It also works with a live video feed and it was tested on complex backgrounds and with varying lighting conditions. As an example application, they also implemented mouse control. Nevertheless, the robustness of both systems

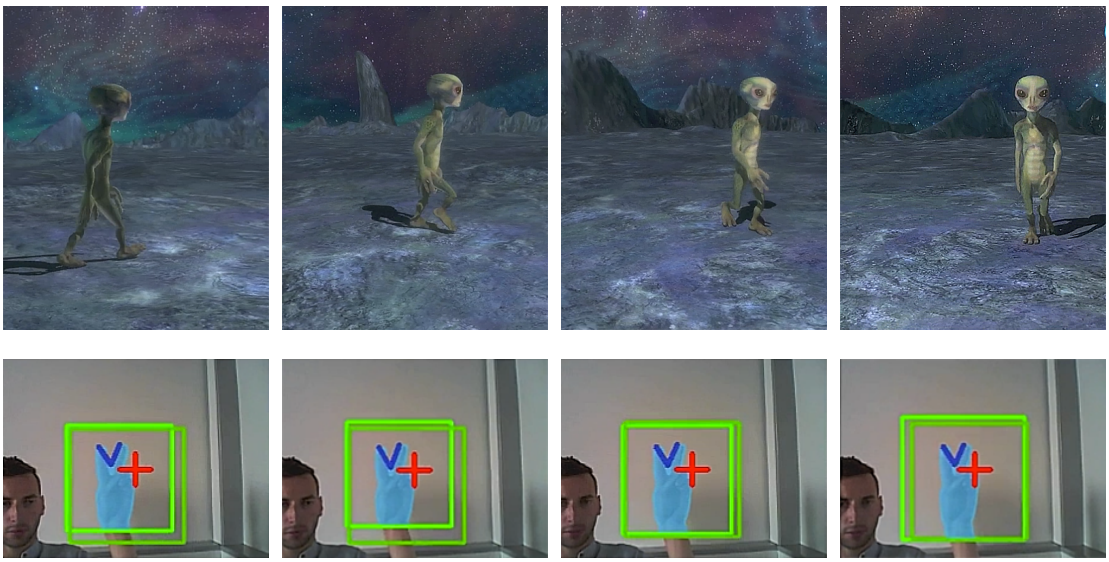


Figure 6.6 – *The NVIDIA Alien vs. Triangles demo. We rotate the planet with our hand gesture*

is hardly comparable. For one, Stenger, Woodley, and Cipolla [2010] does only detect one hand posture, the fist. The finger positions are not considered, while our system is able to recognize any arbitrary combination of fingers. Additionally we focused on creating an open source *framework* for gesture recognition, which runs on any device while they use a proprietary Toshiba television set as a reference system. Lastly, our scripting interface allows us to rapidly implement new gestures for keyboard and mouse commands instead of offering a fixed set of gestures as in their solution. Considering all of the above points, we decided not to include Stenger, Woodley, and Cipolla [2010] for our comparison in Table 6.4.

Project	Tracking system	Approach	Accuracy	Time/Frame
[Schlömer, Poppinga, Henze, and Boll, 2008]	Special Hardware	K-Means, Markov-Model, Bayes	90%	realtime
[Ren, Yuan, and Zhang, 2011]	Depth camera	FEMD	93.9%	4.0s
[Wen, Hu, Yu, and Wang, 2012]	Depth camera	Color Segmentation	95%	not disclosed
[Stergiopoulou and Papamarkos, 2009]	2D-camera	Neural Network	90.45%	1.5s
[Agarwal, Desai, and Saha, 2012]	2D-camera	Haar,CAMshift,Curve fitting	~92%	realtime[a]
[Ong and Bowden, 2004]	2D-camera	K-Mediod, Shape	97.2%	not disclosed[b]
Our system	**2D-camera**	**Haar, CAMshift, Shape, Skin**	**90%**	**realtime**

Table 6.4 – *Comparison of results of gesture recognition systems*

[a]Only the hand position is recognized, not the hand pose and the fingers.
[b]Accuracy on **simple** backgrounds.

Chapter 7

Conclusions

We summarize our findings and contributions. We also discuss ways to enhance the system in the future.

We have created a flexible framework which combines multiple image cues into one hand proposal. We can detect all desired hand features (position, contour, fingers) at difficult lighting conditions and with complex backgrounds with minimal training. On the basis of our data we created a gesture interface which we use to control a computer mouse and execute keyboard shortcuts on behalf of the user. We achieve an accuracy which is comparable with the state of the art.

7.1 Achievements

Our accomplishments include:

- A practical, test-driven choice of the "filter cascade" of gesture detection algorithms suitable for our task
- An iterative bootstrapping approach to supply the filters with required initial data, basing on the same data stream the detection will subsequently operate
- Modifications of the detector algorithms to optimize their performance for hand detection
- The definition of a unified interface to ensue correct operation of single filters; with this we are able to treat individual algorithms as interchangeable "building blocks" despite their very different nature
- A confidence-values-based geometric composition of the output of individual filters. Each filter uses a set of custom heuristics to determine a confidence value for its output. We use this information to combine the estimations of all detectors to a final result which is more robust against errors of a single detector.

- Evaluation of the resulting software, especially in terms of real-time capability and robustness of the detection process. Our system works with stock webcams on an inexpensive consumer computer hardware. The median of the worst measured filter performance was above 45 fps on a quite outdated machine; hence we can guarantee non-disturbing performance of our software. The realtime capability was achieved using a MacBook Pro with a 2.4 GHz Intel Core 2 Duo and 4 GB of RAM. In our tests, the system worked with 24 frames per second.
- Release of the detector framework as an open source application. It can be used as a gesture recognition *plugin* in other programs. The software achieves top results in various established code metrics (see Appendix C.1).

We have presented a combination of several *filters*, also known as detectors or trackers, for the robust, real-time hand gesture recognition. We have modified and adapted multiple methods from computer vision for this purpose: Haar, CAMshift, Shape, and Skin filters. They are typically used for face detection, but were tailored to hand recognition; changes to separate filters include using a different training set, permitting only a single color channel as input, etc.

The most important contribution of this work is not the adaption of each single filter, but their *combination* into a working system. This includes *chaining* the filters, organizing them into two groups – the position cascade and the shape cascade, – and *bootstrapping* the filters. Most of them require some initial images to track and/or compare with. We iterate one filter until it reaches sufficient confidence levels and then use the successfully detected hand position as input for the further, advanced filters. Such a technique proves to be very stable, as it adapts the whole chain to the particularities of the current user (hand shape, hand size, skin color, etc.) and to the current setting (e.g., background, light conditions, white balance).

We invested a special effort into the robustness of the system. The chain of the filters needs to agree on the hand position to ensure reliable tracking. Due to bootstrapping the adaptive approach always has recent and good-corresponding data; this increases the reliability. One of our goals was to minimize the cases of falsely detecting a hand where no hand was, we deem it as reached.

We studied the strengths and shortcomings of each filter in our setting and utilized heuristics to further improve the robustness. Our system has a short-time memory on the last few frames.

7.2 Future work

One important goal for the future is to port our software to systems with little processing power (such as embedded devices, smartphones or even ubiquitous

computers like Google Glasses). A way to achieve better performance is to reduce the number of analyzed frames (i.e., how often the detectors search for a hand). For that reason, we provide keyboard shortcuts to control the delay between each frame. Nevertheless, careful tuning would be necessary for any new supported platform.

All detectors need to be working under realtime constraints. Therefore we did not consider some of the newer developments of object recognition, namely feature detectors like *SIFT* [see D. G. Lowe, 1999] and *FREAK* [A. Alahi, R. Ortiz, and P. Vandergheynst, 2012]. These filters calculate a multitude of features – significant points within an image – and track them in a video stream.

It also sounds reasonable, that proper threading/multiprocessor support would greatly improve the overall performance. Independent filters, like CAMshift and Shape, could be executed in parallel, without affecting each other. So far we didn't see a necessity for this step because we already achieved realtime performance by calling the detectors sequentially.

A crucial issue for the usability of a product would be the minimization of false positives. We have already quite advanced at this point, but it would always be possible to improve even more. On that account, we want to implement more heuristics and algorithms, which help the system track hands that are partly or fully hidden during a limited amount of time (occlusion). One candidate is FREAK A. Alahi, R. Ortiz, and P. Vandergheynst, 2012, which is described as rotation and scale invariant.

For a production-ready program, a better feedback mechanism is crucial [Sears and Jacko, 2007, p. 191]. The system should signal when a gesture is activated or if the gesture recognition got interrupted. One could think of a tiny icon in the status bar of any operating system, which indicates the condition of the tracking system (a green icon could indicate a fully trained detector, a yellow icon could mean a working but uncalibrated system and a red icon might signal an error condition).

Of course, simply enabling the detection of more gestures is also important.

We are on the edge of a new ability to talk with computers in a radically different way. A long time goal in human-computer interface design is to communicate with these devices intuitively. With gesture recognition and similar technology we come closer to this goal than ever before.

List of Figures

1.1 Human perception and machine processing of image data. Inspired by OpenCV Development Team [2013b] . . . 3
1.2 A figure from the patent specification of the VPL *DataGlove*. Source: Vpl [1996] . . . 6
1.3 Extracting a hand shape using depth information with *Microsoft Kinect*. Source: Biswas and Basu [2011] . . . 8

2.1 The five gestures implemented by Schlömer, Poppinga, Henze, and Boll. The last symbol stands for the gesture of serving a tennis ball . . . 10
2.2 The slide gestures implemented by Agarwal, Desai, and Saha . . . 11
2.3 The gesture interface presented by Toshiba at the Internationale Funkausstellung in Berlin 2008. Source: Stenger, Woodley, and Cipolla [2010] . . . 12

3.1 A typical use-case for the application. The green area symbolizes the camera's field of view. Adapted from [Loong, 2013] . . . 14
3.2 Hand movements – a taxonomy by [Pavlovic, R. Sharma, and Huang, 1997] . . . 17
3.3 Manipulative gestures in natural and virtual environments . . . 18
3.4 An example for a typical skeletal model. Source: Pavlovic, R. Sharma, and Huang [1997] . . . 18
3.5 A taxonomy of hand gesture models for HCI. Adapted from: [Pavlovic, R. Sharma, and Huang, 1997]; [Wikipedia, 2013c] . . . 19
3.6 A high level view of the system. We will extend this schematic as we progress . . . 20
3.7 Dataflow between the system components . . . 21
3.8 Face removal improves the detection process. . . . 21

4.1 A slightly simplified schematic of Viola–Jones. . . . 25
4.2 Basic shapes for features (wavelets) . . . 26
4.3 Two matching Haar features from a hand candidate . . . 26
4.4 Using an integral image representation to calculate a haar feature . . . 27
4.5 Combining two weak classifiers to one strong classifier. . . . 28

4.6 A Haar cascade boosted with AdaBoost. Weak classifiers are combined to strong classifiers, also called stages. At each stage, a possible hand can be discarded. The objects which remain at the end are classified as correct hand detections. 29
4.7 Comparing Haar detection results on simple and complex backgrounds 31
4.8 The hand 31
4.9 The Haar filter often "jumps" to a completely unrelated region of an image for one frame during hand tracking. Therefore a valid detection requires that consecutive hand positions overlap. 32
4.10 Influence of face detection on the confidence in the hand detection . 33
4.11 A slightly simplified schematic of CAMshift. 34
4.12 A typical use-case for the program. The hue values of the hand are only a tiny part of all values of the histogram 35
4.13 Because we use the *HSV* colorspace for our CAMshift implementation, the result is invariant to lighting variants. Both drawings share the same color histogram. 36
4.14 The result of a *Back–Projection* process. The brighter a pixel, the higher the probability that it belongs to the tracked hand. 37
4.15 The result of a backprojection iteration. We applied a threshold to the result for better visibility. All white pixels belong to the tracked object with a probability greater than 0.5. The algorithm produces a lot of false positives with skin-colored backgrounds (the top two images) but yields much better results on neutral backgrounds (the bottom two images). 38
4.16 The Meanshift algorithm is used to determine the optimal position of the search window. Inspired by Belaroussi [2006] 39
4.17 Fitting an ellipse to the skin color distribution of a simple 2D object. Note how the hands of the character influence the centroid of the ellipse as well as its size and orientation. Source: Vectorized drawing of output from own program. Source of the original sprite: Nintendo Inc. 41
4.18 Adaption of the hand ellipse size over time. 42
4.19 Typical error symptoms of CAMshift 42
4.20 In our tests the ratio of the both axes of the ellipse around the fist is almost one and resembles a perfect circle. 43
4.21 The ratio of the both axes of the ellipse around the palm is roughly two 43
4.22 A slightly simplified schematic of the shape detector. 45
4.23 The template matching process. Moving the small template image T across a base image I. We calculate the residuals at every position. Comparing the sums of these residuals gives a good estimate for the best match. 46
4.24 Our running example is a search problem using two rectangle functions. We try to find the needle in the haystack. 46

4.25 Applying the normed squared difference to the rectangle functions from above. We scaled the function so that its values are between 0 and 1. 48
4.26 Applying the normed cross correlation to the rectangle functions from above. We scaled the function so that its values are between 0 and 1. 49
4.27 A qualitative comparison of the two likelihood models. We look for the best position of the needle function in the haystack function. Top left: The *haystack* rectangle. Top right: The *needle* rectangle (which has an offset of -2 from the origin). Bottom left: The normed cross correlation of both rectangles. Note that it has the exact shape of the haystack image. The best location is (0,0). Bottom right: The Normed squared difference function. The best match is also at the origin. 50
4.28 Top: An example for a successful match. Note that the fingers are not spread apart in the template image. Nevertheless the correct position was found (although the detected boundaries are those of the template image). Bottom: The result of the normed squared difference method. The bright white spot in the top right corner of the image marks the best match. 51
4.29 Using edge detection we can eradicate a common error cases where the hand would not get detected in front of the face. 52
4.30 A typical error case of the shape detector. All pixels of the template image are of equal importance to the matching process. This means that the background can be more decisive than the foreground. 53
4.31 A slightly simplified schematic of the skin filter. 54
4.32 We use the same settings for the HSV colorspace to extract both hands in the above sample. The hand is marked with red. Sources: Barack Obama by Troy [2008], George W. Bush by Doss [2005] 56
4.33 Skin color extraction visualized using the HSV color cylinder 57
4.34 Using our framework to test and configure the Skin filter. Filtering for the skin colored hue component in various light settings. The settings can be adjusted at runtime. The photos above shows a wide variety of usage examples with most different lighting conditions, shadow projections and saturation values. All of the hands in the examples have been detected successfully with modified settings. For this test we used: Hue: 3-100 degrees, Saturation: 23-68 percent, Value: 28-88 percent. 58
4.35 One of the general-purpose HSV settings we use for hand extraction. Since we only detect skin colors in a narrow region around the hand we can reduce the risk of noise based on skin-colored backgrounds. Hence, the settings can be quite generous. 59

4.36 Detecting a finger from a defect in the convex hull. Vectorization of an actual skin segmentation (heavily dilated) with an estimated convex hull around the hand. 60
4.37 Finger detection using skin-colored image regions. 61
4.38 A failed hand segmentation. Detecting the hand in this case is also non-trivial for humans 61
4.39 A common error case where the wrist is detected as a part of the hand can be avoided with our approach since we only detect skin colors at the hand position we determined before. 62
4.40 Snapshots of a successful bootstrapping process. 65
4.41 Schematic of the bootstrapping process 66
4.42 Two different possible results of all three position detectors. We use a heuristic to determine a refined hand estimate in both cases 69

5.1 A high level view of the system. We will extend this schematic as we progress 72
5.2 Executing dynamic gestures 73
5.3 Displacement of a Joystick 74
5.4 The mouse gesture. From left to right: Initializing a gesture and getting the reference point (the red cross). Moving the hand to the right moves the mouse, too. After we take the hand away the detector confidence decreases. No valid detections found. 75

6.1 Screenshots of our test sequences, roughly ordered by ascending difficulty level 79
6.2 The distribution of the confidence values for the position cascade. (The skin detector yields no confidence values.) 80
6.3 The boxplots for the frame rate, computed for each method (four plots on the left) and the cumulated worst frame rate (the rightmost plot). We combined the *worst* possible frame rates for all methods – it could be only better in practice. We observe that our implementation is definitely real-time capable. 82
6.4 A home office environment with two computer displays with a webcam in-between 83
6.5 The game on the right gets controlled by the input on the left. We can see that Pacman gets caught by Clyde right at that moment. . . . 84
6.6 The NVIDIA Alien vs. Triangles demo. We rotate the planet with our hand gesture 85

A.1 Scenes from the demo video 95

List of Tables

3.1 The hand data we want to collect 16
3.2 Issues which need to be considered during development 17

4.2 A summary of specific detector strengths and weaknesses 64
4.3 Detector dependencies and foundations 65

6.1 Classification of our test videos 77
6.2 Results of the robustness evaluation on sample video data. 78
6.3 Benchmark results. We show the hypothetical frame per second rate for each filter, if it were executed solely. 81
6.4 Comparison of results of gesture recognition systems 86

B.1 Key commands supported by our prototype 96

C.1 Important code metrics for our project 98

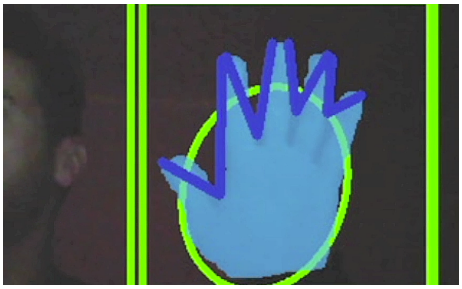
Appendix A

Supporting material

We recorded a demo video of our prototype, which can be viewed at http://www.matthias-endler.de/media/video/tracker.mov.

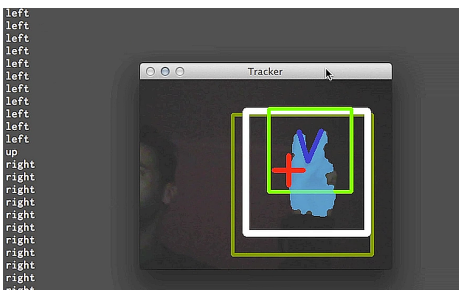
Figure A.1 shows some screenshots from the video. The video shows the following scenarios:

- Calibration/Bootstrapping (Figure A.1a)
- Exposé – A typical use-case for a desktop environment (Figure A.1b)
- Mouse movement – Computer game control (Figures A.1c and A.1d)



(a) *First scene: Bootstrapping process*

(b) *Second scene: Exposé*



(c) *Third scene: Mouse movement*

(d) *Fourth scene: Controlling a game*

Figure A.1 – *Scenes from the demo video*

Appendix B

Keyboard shortcuts

The tracker program can be controlled with a number of keyboard shortcuts, which are listed in Table B.1.

Key	Function
+	Reduce program speed (longer delay between analyzed frames)
-	Increase program speed (shorter delay between analyzed frames)
1	Toggle display of estimated hand position
2	Toggle display of all position detector positions
3	Toggle display of skin detector result
f	Switch between stored configurations
d	Make a screenshot
p	Pause program
r	Reset all detectors
s	Save current configuration in a file
t	Test mode (don't execute any actions)
ESC, q	Exit program

Table B.1 – *Key commands supported by our prototype*

Appendix C

Code

C.1 Metrics

Table C.1 shows a summary of our static code evaluation after we finished the project. We only selected some of our most important modules for the summary. Other modules are in a similar state.

For each module we measure the following code metrics:

Cyclomatic Complexity This metric (developed by McCabe [1976]) "directly measures the number of linearly independent paths through a program's source code" [Wikipedia, 2013b]. We calculate the average cyclomatic complexity of all methods inside a module. Grades from A (very good) to F (very bad) are available

Maintainability Index Additionally, we measured the maintainability index (MI). This is a metric, with the aim to determine how easy a body of code future can be maintained in the future. As above, the rating is similar to school grades. All modules (including helper functions) have a maintainability index of A.

Comment percentage We commented our code substantially. This makes it easier for future developers to work with our code.

Lines of code All modules shall have a *reasonable* size. This makes it easier to maintain the code in the future. No module is longer than a few hundred lines of code (LOC).

Module	Comment percentage	LOC	Average complexity
Main Detector	41%	280	A (2.29)
Haar	27%	109	A (2.29)
Camshift	35%	147	A (1.56)
Shape	42%	121	A (1.44)
Skin	36%	258	A (1.84)
Face	31%	121	A (2.00)
Gesture	40%	102	A (2.13)
Action	35%	255	A (2.29)
Tracker	44%	232	A (3.40)

Table C.1 – *Important code metrics for our project*

Results As can be seen, the system is loosely coupled. It achieves the best possible grade A for almost all methods of all modules. The average cyclomatic complexity of all blocks (classes, functions, methods) analyzed is A (2.102564102564). We used the open source tool `radon` [Lacchia, 2013] for our metrics.

C.2 CAMshift ellipse

In Section 4.2.2 we presented an example for the calculation of a CAMshift ellipse around a tracked object (Figure 4.17). We use a *Super Mario* sprite and calculate the ellipse based on a probability image. In order to focus on the calculation of the image moment from the pixel data, we use the red channel as our probability image. Listing C.1 shows the code we used to create Figure 4.17.

```
1  import Image
2  from math import atan, sqrt, degrees
3  import cv2,cv
4
5  def Mij(im, i, j):
6    """
7    Calculate moments
8    """
9    w, h = im.size
10   pix = im.load()
11   # To keep things simple, we use the red channel as
12   # an approximation for the skin color probability
13   return sum([x**i * y**j * pix[x,y][0] for x in range(w) for y in
         range(h)])
14
15 im_name = "mario.png"
16 im = Image.open(im_name) # Needed for image data access
17 im_cv = cv2.imread(im_name) # Needed for drawing
18
19 # Calculate moments
20 M00 = Mij(im, 0, 0)
21 M01 = Mij(im, 0, 1)
22 M10 = Mij(im, 1, 0)
23 M11 = Mij(im, 1, 1)
24 M02 = Mij(im, 0, 2)
25 M20 = Mij(im, 2, 0)
26
27 # Image center
28 cx = M10/M00
29 cy = M01/M00
30
31 # Image rotation
32 angle_rad = 0.5 * atan((2*M11/M00 - cx*cy) / ((M20/M00 - cx**2) - (
       M02/M00 - cy**2)))
33 angle_deg = degrees(angle_rad)
34
```

```
# Helper variables for length and width
a = M20/M00 - cx**2
b = 2*(M11/M00 - cx*cy)
c = M02/M00 - cy**2

# Length and width
l = int(sqrt(((a+c) + sqrt(b**2 + (a-c)**2))/2))
w = int(sqrt(((a+c) - sqrt(b**2 + (a-c)**2))/2))

# Print results
print cx, cy, l, w, angle_deg
# Draw ellipse
cv2.ellipse(im_cv, (cx, cy), (l, w), angle_deg, 0, 360, (0,0,255),
    -1)
cv2.imshow("Mario", im_cv)
cv2.waitKey()
```

Listing C.1 – *Calculate the tracking ellipse from a probability distribution*

C.3 Position estimation

In Section 4.7 we combined the results of all position detectors to create a more accurate estimate of the hand position. We created a test program to try different algorithms for this hand estimate. The procedure is straightforward: First we create a number of randomly positioned rectangles, which mark the area around the hand. Each rectangle gets assigned a probability value, which simulates the confidence of a detector that the search result is correct.

The algorithm which delivered the most accurate results is shown in Listing C.2. It calculates the weighted average of all overlapping rectangles based on the probabilities of these rectangles. The probabilities are random values between 0 and 1. If no two rectangles are intersecting by more than a given percentage, the rectangle with the maximum probability is selected to be the hand position estimate.

```
import cv2
import numpy as np
import random
import rectangle

height = 480
width  = 640

def max_rectangle(rectangles):
  """ Get the rectangle with the highest confidence """
```

```
  max_rects = sorted(rectangles, key=lambda x: x[1], reverse=True)
  return max_rects[0]

def get_bins(rectangles):
  """ Put overlapping rectangles into bins """
  bins = [ [r] for r in rectangles]
  for i, rect1 in enumerate(rectangles):
    for rect2 in rectangles:
      # Extract values from (rect_pos, probability) tuples
      r1, p1 = rect1
      r2, p2 = rect2
      if r1 != r2 and rectangle.intersect_percentage(r1, r2) > 0.5:
        bins[i].append(rect2)
  return bins

def rects_overlap(rectangles):
  """ Select the bin with the most overlapping rectangles """
  bins = get_bins(rectangles)
  print "Bins", bins
  max_bins = sorted(bins, key=lambda bin: len(bin), reverse=True)
  return max_bins[0]

def average_weighted(rects):
  """ Calculate the weighted average of all given rectangles """
  sum_p = sum([p for r,p in rects])
  # Prevent division by zero
  if sum_p < 0.1:
    return max_rectangle(rects)
  products_x1 = [r[0]*p for r,p in rects]
  products_y1 = [r[1]*p for r,p in rects]
  products_x2 = [r[2]*p for r,p in rects]
  products_y2 = [r[3]*p for r,p in rects]
  avg_x1 = int(sum(products_x1) / sum_p)
  avg_y1 = int(sum(products_y1) / sum_p)
  avg_x2 = int(sum(products_x2) / sum_p)
  avg_y2 = int(sum(products_y2) / sum_p)
  avg_p = sum_p / len(rects)
  return ([avg_x1, avg_y1, avg_x2, avg_y2], avg_p)

def predict(rectangles):
  """ Get an estimated position based on all input rectangles """
  # Check if overlapping
  overlapping = rects_overlap(rectangles)
  if len(overlapping) > 1:
```

```
    # Calculate a weighted average rectangle over all
    # overlapping rectangles
    return average_weighted(overlapping)

  # Not overlapping.
  # Fall back to rect with highest probability
  return max_rectangle(rectangles)

def rand_prob():
  """ Calculate a random probability between 0 and 1 """
  return round(random.uniform(0,1), 1)

def rand_rect():
  """ Calculate a random rectangle """
  rwidth = 150
  rheight = 200
  x = random.randrange(0, width - rwidth, 1)
  y = random.randrange(0, height - rheight, 1)
  return [x, y, x + rwidth, y + rheight]


# Main program

while True:
  # Create a blank image
  rect_image = np.zeros((height,width,3), np.uint8)

  # Create some rectangles
  rectangles = [(rand_rect(), rand_prob()) for count in range(3)]

  # Draw rectangles
  for r,p in rectangles:
    print r,p
    x1, y1, x2, y2 = r
    cv2.rectangle(rect_image,(x1,y1), (x2,y2), (0,255,0))

  # Predict estimate
  max_rect, p = predict(rectangles)
  print "Maximum rectangle", max_rect, p
  x1, y1, x2, y2 = max_rect
  cv2.rectangle(rect_image,(x1,y1), (x2,y2), (0,0,255))

  # Show estimate
  cv2.imshow("Rects", rect_image)
```

```
99  k = cv2.waitKey()
100 if k == ord('q'):
101   exit()
```

Listing C.2 – *Test environment to calculate the estimated hand position from a range of position detector results*

C.4 Hierarchy of an OpenCV Haar cascade

Listing C.3 shows an excerpt of an *OpenCV* cascade file. The file was obtained by running the training program `traincascade` that comes with the OpenCV package. Each cascade consists of a number of stages. An image region needs to pass all the stages in order to be classified as a positive detection result (a hand in our case). The stages are hierarchically organized using a set of trees. Each tree has a root node, which is a simple image feature. "A haar feature consists of 2-3 rectangles with appropriate weights [...] weights of opposite signs and with absolute values inversely proportional to the areas of the rectangles" [OpenCV Development Team, 2010]. For each feature, the pixel sum over each rectangle is calculated. These sums are compared with each other (normally by building a simple difference). Based on the result of the commparison, a response is returned for the feature at the particular image location. If the result is bigger than the given threshold, the left value will be returned, otherwise it returns the right value. A stage is passed, when the stage threshold is reached.

```
1  <opencv_storage>
2    <haarcascade_hand type_id="opencv-haar-classifier">
3      <size>48 48</size>
4      <stages>
5        <_>
6          <!-- stage 0 -->
7          <trees>
8            <_>
9              <!-- tree 0 -->
10             <_>
11               <!-- root node -->
12               <feature>
13                 <rects>
14                   <_>
15                     11 13 20 12 -1.</_>
16                   <_>
17                     11 13 10 6 2.</_>
18                   <_>
19                     21 19 10 6 2.</_>
20                 </rects>
```

```
                <tilted>0</tilted>
              </feature>
              <threshold>2.1414609364001080e-005</threshold>
              <left_val>0.5653824210166931</left_val>
              <right_val>0.4680362045764923</right_val>
            </_>
              <!-- other nodes... -->
          </_>
              <!-- other trees... -->
        <stage_threshold>-1.5993740558624268</stage_threshold>
        <parent>-1</parent>
        <next>-1</next></_>
      </_>
        <!-- other stages... -->
    </stages>
  </haarcascade_hand>
</opencv_storage>
```

Listing C.3 – *An OpenCV HAAR cascade file*

Glossary

We provide a list of definitions in order to avoid uncertainty of meaning from ambiguous technical terms within this document. The significance of these definitions is limited to the thesis and is limited to the context of our work. Therefore they raise no claim to completeness in any other context.

Bootstrapping The process of starting a more sophisticated detector with input data from already running detectors.

Cascade The concatenation of multiple detectors into one entity which delivers a unified output from all of its parts.

Confidence Used only in combination with the detection of hands. See hand proposal.

Convex hull Given a set of X points, the convex hull may be defined as the intersection of all convex sets containing X. (Adapted from [Wikipedia, 2013a].)

Cue Used as a synonym for *feature* within this document.

Detection The action or process of identifying the presence of something concealed [Oxford Dictionaries, 2013]. In the scope of this work we try to detect hand and facial features as well as gestures.

Detector An image processing technique used to emphasize a certain part or characteristic of an image. In our case a module within the system, which is capable of detecting hand features or the hand position.

False negative The null hypothesis is accepted although it is incorrect e.g., a detector finds a hand at an incorrect position. Also known as a type II error.

False positive The null hypothesis is rejected although it is correct e.g., a detector finds no hand in an image even though there is one. Also known as a type I error.

Feature A distinctive attribute of an object inside an image or a video. (Adapted from [Oxford Dictionaries, 2013].)

Filter Used as a synonym for *detector* within this document.

FPS Frames per second. A common speed measurement for multimedia applications.

Gesture A hand movement to express an idea or meaning. (Adapted from [Oxford Dictionaries, 2013].)

Hand proposal An estimated hand position as determined by a detector combined with a probability for the correctness of the detection.

Heuristic The term "heuristic" is of Greek origin, meaning "serving to find out or discover." From its introduction into English in the early 1800s up until about 1970, "heuristics" referred to useful, even indispensable cognitive processes for solving problems that cannot be handled by logic and probability theory alone (from [Peter M. Todd, 1999] referencing [Polya, 2008])

Neural network A computer system modeled on the human brain and nervous system [from Oxford Dictionaries, 2013].

Null hypothesis A proposition that undergoes verification to determine if it should be accepted or rejected in favor of an alternative proposition [Web Finance, 2013]. A basic assumption about a system. In our case, a hand shown in an image or a gesture made by the user.

Occlusion Parts of an object are hidden behind another. This makes object recognition significantly more difficult.

Predicate (Philosophy/Logic) A property, characteristic, or attribute that may be affirmed or denied of something. (from ["Collins English Dictionary - Complete and Unabridged 10th Edition" 2013])

Realtime The processing of information that returns a result so rapidly that the interaction appears to be instantaneous [IMCCA, 2013] .

Reliable Used as a synonym for *robust* within this document.

Robust Able to withstand or overcome adverse conditions like changing lighting conditions and adaptable to different users.

Search window A rectangular area in which an object can be found.

Bibliography

Agarwal, Bhavish Sushiel, Jyoti R Desai, and Snehanshu Saha (2012). "A Fast Haar Classifier based Gesture Recognition using camShift algorithm and Curve Fitting Method". In: (cit. on pp. 11, 86).

Alahi, A., R. Ortiz, and P. Vandergheynst (2012). "FREAK: Fast Retina Keypoint". In: CVPR '12. IEEE, pp. 510–517 (cit. on pp. 62, 89).

Alahi, Alexandre, Raphael Ortiz, and Pierre Vandergheynst (2012). "Freak: Fast retina keypoint". In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, pp. 510–517 (cit. on p. 25).

Barton, Paul (2013). *PyUserInput – A module for cross-platform control of the mouse and keyboard in python*. URL: `https://github.com/SavinaRoja/PyUserInput` (visited on 08/30/2013) (cit. on p. 74).

Bay, Herbert, Tinne Tuytelaars, and Luc Gool (2006). "SURF: Speeded Up Robust Features". In: *ECCV '06*. LNCS 3951. Springer, pp. 404–417. ISBN: 978-3-540-33832-1 (cit. on p. 62).

Bedregal, Benjamín C., Antônio C. R. Costa, and Graçaliz P. Dimuro (2006). "Fuzzy Rule-Based Hand Gesture Recognition". In: *Artificial Intelligence in Theory and Practice*. IFIP 217. Springer US, pp. 285–294. ISBN: 978-0-387-34654-0 (cit. on p. 9).

Belaroussi, Rachid (2006). "Localisation du visage dans des images et séquences vidéo couleur". PhD thesis. Université Pierre et Marie Curie (cit. on p. 39).

Belongie, S., J. Malik, and J. Puzicha (2002). "Shape matching and object recognition using shape contexts". In: *IEEE T. Pattern. Anal.* 24.4, pp. 509–522. ISSN: 0162-8828 (cit. on pp. 20, 45, 64).

Biswas, K. K. and S.K. Basu (2011). "Gesture recognition using Microsoft Kinect". In: *Automation, Robotics and Applications (ICARA), 2011 5th International Conference on*, pp. 100–103 (cit. on p. 8).

Bradley, David (2003). *HAAR Cascade for profile faces (Cached version. Original source no longer available)*. URL: http://web.archive.org/web/20040405113537/http://www.intel.com/research/awards/2003/bradley.htm (visited on 09/11/2013) (cit. on p. 30).

Bradski, Gary R. (1998). "Computer vision face tracking for use in a perceptual user interface". In: *Intel Technology Journal* Q2 (cit. on pp. 20, 34 sq., 55, 60, 64).

Chellappa, R., P. Sinha, and P.J. Phillips (2010). "Face Recognition by Computers and Humans". In: *Computer* 43.2, pp. 46–55. ISSN: 0018-9162 (cit. on p. 3).

Chui, C.K. (1992). *An Introduction to Wavelets*. Wavelet analysis and its applications. Academic Press. ISBN: 9780121745844. URL: http://books.google.de/books?id=JCXZJSfCVHUC (cit. on p. 25).

"Collins English Dictionary - Complete and Unabridged 10th Edition" (Aug. 19, 2013). In: URL: http://www.collinsdictionary.com/dictionary/english/ (cit. on p. 106).

Comaniciu, Dorin and Peter Meer (2002). "Mean shift: A robust approach toward feature space analysis". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 24.5, pp. 603–619 (cit. on p. 39).

Do, Trinh Minh Tri, Jan Blom, and Daniel Gatica-Perez (2011). "Smartphone usage in the wild: a large-scale analysis of applications and context". In: *Proceedings of the 13th international conference on multimodal interfaces*. ICMI '11. Alicante, Spain: ACM, pp. 353–360. ISBN: 978-1-4503-0641-6. URL: http://doi.acm.org/10.1145/2070481.2070550 (cit. on p. 1).

Doss, Marion (2005). *President George W. Bush waves to service members at Naval Air Station North Island*. URL: http://www.flickr.com/photos/7337467@N04/3051560800 (visited on 08/20/2013) (cit. on p. 56).

Erol, Ali et al. (2007). "Vision-based hand pose estimation: A review". In: *Computer Vision and Image Understanding* 108.1, pp. 52–73 (cit. on p. 6).

Free Software Foundation, Inc (2013). *GNU Lesser General Public License*. URL: http://www.gnu.org/licenses/lgpl.html (visited on 06/29/2007) (cit. on p. 19).

Freeman, William T, Ken-ichi Tanaka, Jun Ohta, and Kazuo Kyuma (1996). "Computer vision for computer games". In: *Automatic Face and Gesture Recognition, 1996., Proceedings of the Second International Conference on*. IEEE, pp. 100–105 (cit. on p. 37).

Freund, Yoav and Robert E. Schapire (1995). "A desicion-theoretic generalization of on-line learning and an application to boosting". In: *Computational Learn-*

ing Theory. LNCS 904. Springer, pp. 23–37. ISBN: 978-3-540-59119-1 (cit. on p. 28).

Freund, Yoav and Robert E. Schapire (1996). "Experiments with a New Boosting Algorithm". In: *Int. Conf. Machine Learning*, pp. 148–156 (cit. on p. 28).

Gauss, C.F. (1809). *Theoria motus corporum coelestium in sectionibus conicis solem ambientium*. URL: http://books.google.de/books?id=ORUOAAAAQAAJ (cit. on p. 47).

Guraya, Fahad Fazal Elahi, Pierre-Yves Bayle, and Faouzi Alaya Cheikh (2009). "People Tracking via a Modified CAMSHIFT Algorithm". In: (cit. on p. 34).

Heinly, Jared, Enrique Dunn, and Jan-Michael Frahm (2012). "Comparative Evaluation of Binary Features". In: *ECCV '12*. LNCS 7573. Springer, pp. 759–773. ISBN: 978-3-642-33708-6 (cit. on p. 9).

Hjelmås, Erik and Boon Kee Low (2001). "Face Detection: A Survey". In: *Comput. Vis. Image. Und.* 83.3, pp. 236–274. ISSN: 1077-3142 (cit. on p. 9).

IMCCA (2013). *Glossary*. URL: http://www.imcca.org/resources/glossary (visited on 08/12/2013) (cit. on p. 106).

Jordao, Luis, Matteo Perrone, Joao Paulo Costeira, and José Santos-Victor (1999). "Active Face and Feature Tracking". In: *Proceedings of the 10th International Conference on Image Analysis and Processing*. ICIAP '99. Washington, DC, USA: IEEE Computer Society, pp. 572–. ISBN: 0-7695-0040-4. URL: http://dl.acm.org/citation.cfm?id=839281.840805 (cit. on p. 54).

Jung, Benjamin (2013). *Camshift: going to the source*. URL: http://www.mukimuki.fr/flashblog/2009/06/18/camshift-going-to-the-source/ (visited on 08/20/2013) (cit. on p. 35).

Kent, Steven (2001). *The Ultimate History of Video Games: From Pong to Pokemon–The Story Behind the Craze That Touched Our Lives and Changed the World*. Three Rivers Press (cit. on p. 83).

Kheng, Leow Wee (2013). *Image Processing*. URL: http://www.comp.nus.edu.sg/~cs4243/lecture/imageproc.pdf (visited on 08/27/2013) (cit. on p. 48).

Lacchia, Michele (2013). *Radon – Code Metrics in Python*. URL: https://pypi.python.org/pypi/radon (visited on 08/20/2013) (cit. on p. 98).

Leutenegger, S., M. Chli, and R.Y. Siegwart (2011). "BRISK: Binary Robust invariant scalable keypoints". In: ICCV '11. IEEE, pp. 2548–2555 (cit. on p. 62).

Li, Andol X. (2011). *Detecting hand gestures using Haarcascades training*. URL: http://www.andol.info/hci/1830.htm (visited on 08/17/2013) (cit. on p. 29).

Lienhart, Rainer (2013). *Biography Prof. Dr. Rainer Lienhart*. URL: http://www.lienhart.de/Prof._Dr._Rainer_Lienhart/Welcome.html (visited on 09/11/2013) (cit. on p. 30).

Lienhart, Rainer, Alexander Kuranov, and Vadim Pisarevsky (2003). "Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection". In: LNCS 2781. Springer, pp. 297–304. ISBN: 978-3-540-40861-1 (cit. on pp. 20, 25 sq.).

Lineback, Nathan (2013). *Graphical User Interface Timeline*. URL: http://toastytech.com/guis/guitimeline.html (visited on 08/22/2013) (cit. on p. 13).

Loong, Joe (2013). *Office Worker Image*. URL: http://www.flickr.com/photos/joelogon/324259281/ (visited on 08/20/2013) (cit. on p. 14).

Lowe, David G (1999). "Object recognition from local scale-invariant features". In: *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*. Vol. 2. Ieee, pp. 1150–1157 (cit. on pp. 3, 89).

Lowe, D.G. (1999). "Object recognition from local scale-invariant features". In: vol. 2. ICCV '99. IEEE, pp. 1150–1157 (cit. on p. 62).

Markou, Nikolas (2012). *Haar XML File*. URL: http://nmarkou.blogspot.de/2012/02/haar-xml-file.html (visited on 08/17/2013) (cit. on p. 29).

Marques, F. and V. Vilaplana (2000). "A morphological approach for segmentation and tracking of human faces". In: *Pattern Recognition, 2000. Proceedings. 15th International Conference on*. Vol. 1, 1064–1067 vol.1 (cit. on p. 55).

McCabe, Thomas J. (1976). *A Complexity Measure*. URL: http://www.literateprogramming.com/mccabe.pdf (visited on 09/13/2013) (cit. on p. 97).

Nambissan, Aravind (2013). *Haar cascades*. URL: https://github.com/Aravindlivewire/Opencv/tree/master/haarcascade (visited on 08/23/2013) (cit. on p. 30).

Neave, Paul (2013). *FreePacman – A browser implementation of Pacman*. URL: http://www.freepacman.org (visited on 08/31/2013) (cit. on p. 83).

Ong, Eng-Jon and Richard Bowden (2004). "A boosted classifier tree for hand shape detection". In: *Automatic Face and Gesture Recognition, 2004. Proceedings. Sixth IEEE International Conference on*. IEEE, pp. 889–894 (cit. on pp. 10, 86).

OpenCV Development Team (2010). "OpenCV Reference Manual 2.1". In: (cit. on p. 103).

OpenCV Development Team (2013a). *Cascade Classification*. URL: http://docs.opencv.org/modules/objdetect/doc/cascade_classification.html (visited on 08/19/2013) (cit. on p. 30).

— (2013b). *OpenCV Tutorials – Release 2.4.5.0* (cit. on p. 3).

Oxford Dictionaries (2013). *Online dictionary*. URL: http://oxforddictionaries.com/ (visited on 08/11/2013) (cit. on pp. 105 sq.).

Pausch, Randy (1991). "Virtual reality on five dollars a day". In: *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology*. ACM, pp. 265–270 (cit. on p. 6).

Pavlovic, Vladimir I, Rajeev Sharma, and Thomas S. Huang (1997). "Visual interpretation of hand gestures for human-computer interaction: A review". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 19.7, pp. 677–695 (cit. on pp. 17–19).

Peter M. Todd, Gerd Gigerenzer (1999). *Simple Heuristics That Make Us Smart*. URL: http://www-abc.mpib-berlin.mpg.de/users/ptodd/SimpleHeuristics.BBS/ (visited on 08/11/2013) (cit. on p. 106).

Polya, George (2008). *How to solve it: A new aspect of mathematical method*. Princeton University Press (cit. on p. 106).

Poynton, Charles (2006). *Color FAQ - Frequently Asked Questions Color*. URL: http://www.poynton.com/notes/colour_and_gamma/ColorFAQ.html (visited on 08/29/2013) (cit. on p. 55).

Pulkit, Kathuria and Yoshitaka Atsuo (2012). *Hand Gesture Recognition by using Logical Heuristics*. Tech. rep. 25. Japan Advanced Institute of Science and Technology, School of Information Science (cit. on pp. 20, 58).

Quek, Francis KH (1994). "Toward a vision-based hand gesture interface". In: *Proceedings of the conference on Virtual reality software and technology*. World Scientific Publishing Co., Inc., pp. 17–31 (cit. on p. 7).

Rautaray, Siddharth S. and Anupam Agrawal (2012). "Vision based hand gesture recognition for human computer interaction: a survey". English. In: *Artif. Intell. Rev.* Pp. 1–54. ISSN: 0269-2821 (cit. on p. 9).

Ren, Zhou, Junsong Yuan, and Zhengyou Zhang (2011). "Robust hand gesture recognition based on finger-earth mover's distance with a commodity depth camera". In: *Proceedings of the 19th ACM international conference on Multimedia*. MM '11. Scottsdale, Arizona, USA: ACM, pp. 1093–1096. ISBN: 978-1-4503-0616-4. URL: http://doi.acm.org/10.1145/2072298.2071946 (cit. on pp. 10, 61, 86).

Schlömer, Thomas, Benjamin Poppinga, Niels Henze, and Susanne Boll (2008). "Gesture recognition with a Wii controller". In: *Proceedings of the 2nd international conference on Tangible and embedded interaction*. ACM, pp. 11–14 (cit. on pp. 6, 9 sq., 86).

Sears, Andrew and Julie A. Jacko (2007). *The Human-Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications (Human Factors and Ergonomics Series) Second Edition*. Hillsdale, NJ, USA: L. Erlbaum Associates Inc. ISBN: 0805858709 (cit. on pp. 4, 89).

Seo, Naotoshi (2008). *OpenCV Haartraining application*. URL: `https://code.google.com/p/tutorial-haartraining/source/browse/trunk/HaarTraining/src/haartraining.cpp` (visited on 08/17/2013) (cit. on p. 29).

Shin, M.C., K.I. Chang, and Leonid V. Tsap (2002). "Does colorspace transformation make any difference on skin detection?" In: *Applications of Computer Vision, 2002. (WACV 2002). Proceedings. Sixth IEEE Workshop on*, pp. 275–279 (cit. on p. 55).

Sorenson, H.W. (1970). "Least-squares estimation: from Gauss to Kalman". In: *Spectrum, IEEE* 7.7, pp. 63–68. ISSN: 0018-9235 (cit. on p. 47).

Stenger, Björn, Thomas Woodley, and Roberto Cipolla (2010). "A vision-based remote control". In: *Computer Vision*. Springer, pp. 233–262 (cit. on pp. 11 sq., 84 sq.).

Stenger, Björn, Thomas Woodley, Tae-Kyun Kim, et al. (2008). "AIDIA – Adaptive Interface for Display InterAction". In: (cit. on p. 11).

Stergiopoulou, E. and N. Papamarkos (2009). "Hand gesture recognition using a neural network shape fitting technique". In: *Eng. Appl. Artif. Intel.* 22.8, pp. 1141–1158. ISSN: 0952-1976 (cit. on pp. 11, 58, 86).

Tahmasebi, Pejman, Ardeshir Hezarkhani, and Muhammad Sahimi (2012). "Multiple-point geostatistical modeling based on the cross-correlation functions". English. In: *Computational Geosciences* 16.3, pp. 779–797. ISSN: 1420-0597 (cit. on p. 45).

Tripathi, Smita, Varsha Sharma, and Sanjeev Sharma (2011). "Face Detection using Combined Skin Color Detector and Template Matching Method". In: *Int. J. Comput. Appl.* 26.7, pp. 5–8 (cit. on pp. 20, 55).

Troy, Talbot (2008). *Obama Rally, Greensboro, North Carolina*. URL: `http://www.flickr.com/photos/8376385@N06/2896129274/` (visited on 08/20/2013) (cit. on p. 56).

Tukey, John W. (1977). "Exploratory data analysis". In: *Addison-Wesley* (cit. on p. 76).

Vezhnevets, Vladimir, Vassili Sazonov, and Alla Andreeva (2003). "A survey on pixel-based skin color detection techniques". In: *Proc. GraphiCon*. lCCGV' 03 (cit. on pp. 54 sq., 64).

Viola, P. and M. Jones (2001). "Rapid object detection using a boosted cascade of simple features". In: CVPR '01. lEEE, l:511–518 (cit. on pp. 20, 25, 29, 64).

Viola, Paul and Michael J Jones (2004). "Robust real-time face detection". In: *International journal of computer vision* 57.2, pp. 137–154 (cit. on p. 3).

Vpl, Research Inc. (Sept. 11, 1996). "Computer data entry and manipulation apparatus". Patent EP0211984 B2 (European Patent Office). URL: https://register.epo.org/application?number=EP85110377 (cit. on p. 6).

Web Finance (2013). *Business Dictionary*. URL: http://www.businessdictionary.com/ (visited on 08/12/2013) (cit. on p. 106).

Weisstein, Eric W. (2013). *Cross-Correlation From MathWorld–A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/Cross-Correlation.html (visited on 08/27/2013) (cit. on p. 48).

Wen, Yan, Chuanyan Hu, Guanghui Yu, and Changbo Wang (2012). "A robust method of detecting hand gestures using depth sensors". In: *Haptic Audio Visual Environments and Games (HAVE), 2012 IEEE International Workshop on*, pp. 72–77 (cit. on pp. 10, 86).

Whitehill, J. et al. (2009). "Toward Practical Smile Detection". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 31.11, pp. 2106–2111. ISSN: 0162-8828 (cit. on p. 2).

Wikipedia (2013a). *Convex hull — Wikipedia, The Free Encyclopedia*. [Online; accessed 10-September-2013]. URL: http://en.wikipedia.org/w/index.php?title=Convex_hull&oldid=568951987 (cit. on p. 105).

— (2013b). *Cyclomatic complexity — Wikipedia, The Free Encyclopedia*. [Online; accessed 13-September-2013]. URL: http://en.wikipedia.org/w/index.php?title=Cyclomatic_complexity&oldid=560593003 (cit. on p. 97).

— (2013c). *Gesture recognition — Wikipedia, The Free Encyclopedia*. [Online; accessed 18-August-2013]. URL: http://en.wikipedia.org/w/index.php?title=Gesture_recognition&oldid=568105365 (cit. on p. 19).

— (2013d). *HSL and HSV — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-August-2013]. URL: http://en.wikipedia.org/w/index.php?title=HSL_and_HSV&oldid=569106091 (cit. on p. 55).

Wikipedia (2013e). *Power Glove — Wikipedia, The Free Encyclopedia*. [Online; accessed 9-August-2013]. URL: %5Curl%7Bhttp://en.wikipedia.org/w/index.php?title=Power_Glove&oldid=567732779%7D (cit. on p. 6).

Willow Garage (2011). *OpenCV camera compatibility*. URL: http://opencv.willowgarage.com/wiki/Welcome/OS (visited on 08/16/2013) (cit. on p. 14).

Xia, Jie, Jian Wu, Haitao Zhai, and Zhiming Cui (2009). "Moving vehicle tracking based on double difference and camshift". In: (cit. on p. 34).

Zabulis, X, H Baltzakis, and A Argyros (2009). "Vision-based hand gesture recognition for human-computer interaction". In: *The Universal Access Handbook. LEA* (cit. on p. 7).

Zarit, B. D., B. J. Super, and F. K. H. Quek (1999). "Comparison of five color models in skin pixel classification". In: *Proc. Int. Worksh. Recognition, Analysis, and Tracking of Faces and Gestures in Real-Time Systems*, pp. 58–63 (cit. on p. 55).

Zhang, Zhengyou (2012). "Microsoft Kinect Sensor and Its Effect". In: *IEEE MultiMedia* 19.2, pp. 4–10. ISSN: 1070-986X (cit. on p. 7).

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Ölbrunn, den 10. September 2013 *Matthias Endler*