Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Bachelor's Program in Computer Science

Bachelor's Thesis

Multi-User Tracking in Instrumented Rooms

submitted by
Henning Lars Zimmer

on 10th November 2006

Supervisor
Prof. Dr. Dr. h.c. mult. Wolfgang Wahlster

Advisor
Dipl.-Inform. Michael Schmitz

Reviewers
Prof. Dr. Dr. h.c. mult. Wolfgang Wahlster
Prof. Dr. Joachim Weickert

Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, 10th November 2006

Acknowledgments

First of all I have to thank Prof. Wolfgang Wahlster for giving me the opportunity to write this thesis in his research project BAIR (**B**enutzer**A**daption in **I**nstrumentierten **R**äumen, User Adaptation in Instrumented Rooms).

In this context I am also much obliged to my advisor Dipl.-Inform. Michael Schmitz who helped me a lot with any problem I had and gave me really interesting ideas and inspirations.

I am also very thankful to all the nice people at our chair for providing me with a nice working atmosphere and being kind and helpful all the time.

Of course, I also have to thank Prof. Dr. Joachim Weickert for being so kind and accepting my request to be my second reviewer and for sparking my interest in Computer Vision.

Last but not least, I have to express my special thanks to my friends and family for their continuous support.

Abstract

In the ubiquitous computing (ubicomp) vision, one persues the idea of equipping everyday objects with computational abilities, i.e., one integrates computation into the environment of persons. This encompasses intelligent objects, displays and so on. An example of such an environment, an instrumented room, is available to us in our lab.

One open problem in such environments is multi-user distinction, as normally there is more than one person in such an environment, especially in a public room. This actually offers a problem, because associating interactions with devices or objects to a certain user is difficult, or even not possible in this case. On the other hand such an association is crucial for intelligent, personalized services, which may base on an interaction-history.
One existing approach is to require users to carry personalized devices like a PDA or a mobile phone, which are used to carry out interactions.

Our idea is to use a visual approach, decreasing the instrumentation of users by increasing the instrumentation of the environment, which is in our case the instrumented room. We use quite complex sensors, namely cameras, which offer a lot of informations, but are quite challenging to analyze. Therefore, we split our 'problem-space' in a macro- and a micro-space with a final fusion of results. The macro-space comprises the whole room, i.e., all persons and their position, whereas the micro-space deals with special areas of interest and gives detailled informations about the actions of persons in it.
The technique we are actually using is a Computer Vision tracking technique, allowing us to track movements of persons (users) in image sequences obtained by our cameras. Together with homography estimations and skin detection, we will be able to determine how many persons are in the room (tracking), where, approximately, the persons are located (homography) and who interacts with devices (skin-detection) by reaching towards or grasping them.
This approach helps us to do a visual multi-user distinction and an assignment of interactions with devices to users in the environment, which solves the problem mentioned above.

Zusammenfassung

In der Ubiquitous Computing Vision (ubicomp, zu Deutsch: allgegenwärtige Computertechnik), verfolgt man die Idee alltägliche Objekte mit komputionalen Fähigkeiten auszustatten, d.h., man integriert Informationsverarbeitung in die Umgebung von Personen. Dies umfasst intelligente Objekte, Displays usw.. Solch eine Umgebung, ein instrumentierter Raum, steht uns in unserem Labor zur Verfügung.

Ein offenes Problem in solchen Umgebungen ist die Mehrbenutzer-Unterscheidung, da sich normalerweise mehr als eine Person in einer solchen Umgebung befinden wird, vor allem wenn man es mit einem öffentlichen Raum zu tun hat. Dies ist in der Tat ein Problem, weil die Zuordnung von Interaktionen mit Geräten oder Objekten zu einem bestimmten Benutzer schwierig, oder sogar unmöglich ist. Andererseits sind solche Zuordnungen lebensnotwendig für intelligente, personalisierte Dienste, die auf einer Interaktions-Historie basieren.
Ein existierender Ansatz sieht es vor, dass Benutzer personalisierte Geräte, wie einen PDA oder ein Mobiltelefon bei sich tragen müssen, welche dann benutzt werden um Interaktionen auszuführen.

Unsere Idee ist es nun eine visuellen Ansatz zu wählen, welcher die Instrumentierung der Benutzer verringert, indem man die Instrumentierung des Raumes erweitert. Wir benutzen sehr komplexe Sensoren, nämlich Kameras, welche viele Informationen bieten, aber auch schwer auszuwerten sind. Deshalb teilen wir unseren Problemraum in einen Makro- und einen Mikroraum auf und vereinigen die Ergebnisse am Ende. Der Makroraum umfasst die ganze Umgebung, d.h. alle Personen und ihr Positionen, wohingegen der Mikroraum sich mit speziellen, interessanten Teilen unserer Umgebung befasst und detaillierte Informationen über Aktionen der Benutzer darin gibt.
Die Technik die wir hierzu benutzen ist eine Computer Vision Tracking (Verfolgungs-) Technik, die es uns erlaubt die Bewegungen von Personen (Benutzern) in Bildersequenzen unserer Kameras zu verfolgen. Zusammen mit Homographie Schätzungen und Hauterkennung, werden wir in der Lage sein zu bestimmen wie viele Personen im Raum sind (Tracking), wo sie sich ungefähr befinden (Homographie) und wer mit welchem Gerät interagiert, indem er zu ihm zeigt oder nach ihm greift (Hauterkennung).
Dieser Ansatz erlaubt es uns eine visuelle Mehrbenutzer Unterscheidung und eine Zuordnung von Interaktionen mit Geräten zu Benutzern der Umgebung zuzuordnen, welches das beschriebene Problem löst.

Contents

Chapter 1

Introduction

In this chapter we want to present the problem we tried to solve to the reader, state requirements we wanted to meet and shortly summarize the further organisation of this thesis.

Preliminary Note: In this thesis, a lot of technical terms from the Computer Vision and Image Processing world will be used. For persons familiar with this area, in most of the cases no further explanation of this terms is necessary and would disturb the readability of the thesis.
Nevertheless we provide short definitions of used technical terms in the appendix *Glossary - Basics in Image Processing* (see Glossary, Appendix A), where persons not familiar with them can look them up.
We will also denote in the text which terms are defined in the Glossary.

1.1 Issues Addressed

In the context of the `BAIR`-project (**B**enutzer**A**daption in **I**nstrumentierten **R**äumen, User Adaptation in Instrumented Rooms [WKB04]) of our research group, we want to add support for *multiple users* interacting with devices or objects in an ubiquitous computing (*ubicomp*) environment. In such environments, one refrains from having computers as distinct entities and integrates computation abilities into the environment. This encompasses so called *intelligent objects and devices*.
Our example for such an environment is an airport scenario, including a shopping setting, which is available to us in an *instrumented room*, our SUPIE lab (Saarland University Pervasive Intelligent Environment, described in [BK]).
What we now want to know is *how many persons* are in the room, *where* they are located and *which person interacts with which device or object*, e.g. with a display at a wall or with a product in a shelf in the shop.

At the moment, we can only distinguish certain users in our environment via PDAs,

mobile phones or similar personalized devices, which have to be carried along by persons and which have to be used to carry out interactions with devices in our environment.
A visual Computer Vision (CV) approach seemed to be quite fruitful to us, because it offers a good solution to detect movement of persons and interaction of persons with devices in an ubicomp environment without further instrumentation of our users and devices in the environment. We 'just' have to install several cameras, extract the needed informations from each one and finally fuse the informations to our desired result.
In this context, we also ensue a '*come as you are*' approach, i.e., we do not force our persons to wear special clothes, like colored gloves, for better detecting their hands. This, of course, offers some additional problems, but replacing the need of some auxiliary device by another (which may be much less expensive, actually) is not really satisfactory.
The remarks from above also answer the question why we accept the challenge of developing a CV system, i.e., a system where we try to teach a computer to 'see', or to be more exact, to extract semantic informations from images. This field of Computer Science offers a lot of (yet partly unsolved) problems, like sensitivity to image noise, dependency on many configuration parameters and so on.

The main goal addressed in this thesis is now to distinguish certain users and their interactions with devices or objects by *tracking* their location, movements and actions in images obtained by cameras. The approach we follow is to develop an extended person tracking system, we will call *MaMUT* (**Ma**cro- and **M**icro **U**ser **T**racker), and which we develop in the context of this thesis.

1.2 What is Tracking

As mentioned, we want to realize the multi-user distinction via tracking, but what exactly do we understand under this notion?
In Wikipedia we find that '(Video) Tracking is the process of locating a moving object [in our case: persons or their hands], or several ones in time using a camera. An algorithm analyzes the video frames and outputs the location, optionally in realtime.'
In the literature, there exist several solutions for tracking persons via cameras, like [Sie03], [MJD+00], [HHD98] or [KHM+00]. A detailed discussion of related work can be found in the according chapter 2.
The problem hereby is that there is no best solution, every solution has its advantages and is mainly adapted for the setting it was developed for. Examples for crucial differences are:

- Indoor versus outdoor settings
- How many persons have to be tracked
- Do we have to deal with occlusion of persons
- Does the lightning condition change distinctly

- How large is the considered area
- ...

In our case, we will also do a distinction between *macro-* and *micro-tracking* on a room scale. The first notion describes the tracking of persons throughout the whole room, including a *homography estimations* to estimate their position. The latter describes the tracking of hands of persons, using *skin detection*, in front of certain areas of interest, like shelfs, displays and so on. This also explains why we have to use multiple cameras. Of course, this yields the need of a fusing entity, which collects results from all cameras and fuses them, e.g., to assign detected object interactions (micro-tracking) to persons (macro-tracking).
To conclude this section, we want to mention that the notions of macro- and micro-tracking are inspired from [SBB+05], where the authors talk about macro- and micro-navigation in a shopping assistant system operating in ubicomp environments.

1.3 Requirements

The final result, the output, of our system is supposed to encompass the number of persons, their approximate positions and the position of their hands, if they are in an area of interest. These informations have to be made available to other applications operating in the ubicomp environment. To reach these aims, our system has to fulfill certain requirements, which we will now specify.

Robustness: As mentioned, the main Achilles heel of Computer Vision approaches is robustness.
Our system will have problems if there are too many persons in the room, but we require that it will work with a reasonable number of persons compared with the size of the room. Furthermore, lightning conditions and parameter setting will have a big influence on the quality of our results, but we will accept this, as this is the usual price one has to pay when using a Computer Vision system.

Realtime: The tracking task has to be fulfilled in realtime, because we immediately want to react to user interactions with the environment. This requirement poses a strict boundary to the complexity of used algorithms, the capability of the computer system and the infrastructure our tracker runs on and the size of the images to process.

Cameras: We will restrict ourselves to the use of standard, *off-the-shelf* cameras, and hence only obtain $2D$ information, i.e. we cannot a priori determine the *exact position* of a tracked person in the room. But as we use *static cameras*, we can

approximate the position of tracked persons via a *ground plane homography estimation*. A short discussion about moving, so called Pan-Tilt-Zoom (PTZ) cameras is given in section 5.4.3.
The *number* of cameras is not restricted. The *technical requirements* of our cameras encompass: LAN interface to transfer images to the computer system running MaMUT, a frame rate of at least 10 FPS, to meet the realtime needs and a reasonabe resolution. This means that on the one hand, the resolution has to be large enough to recognize all needed details (hands, arms, ...), but on the other hand it has to be small enough, so it does not spoil our realtime needs.

Integration into existing infrastructure: Usually, an ubicomp environment offers a commom communication infrastructure, where applications can post and receive informations. Of course, our system has to be able to at least post its tracking results to this infrastructure.

1.4 Organization

After this introductory chapter, we start to describe the background and implementation of our tracking system MaMUT.

In **chapter 2**, we will give references to *related work* and compare the presented approaches to MaMUT.

In **chapter 3**, we describe the *theoretical background* of our system, which encompasses person tracking, homography estimation, skin detection and fusion of tracking results.

In **chapter 4**, we describe how we concretely *implemented* the theoretical ideas stated beforehand, and finally,

in **chapter 5**, we shortly *conclude* what we have done, state *problems* and possible *solutions* to them and mention possible *extensions and enhancements* to our system.

Chapter 2

Related Work

As one can imagine, there are also other approaches which use visual informations to detect persons. Each has its advantages for the setting it operates in, but all in all none could solve our special problem described in the introduction, which forced us to develop our own system.
In the following we will present some approaches and finally do a comparison with the features to MaMUT.

2.1 Person Tracker

2.1.1 RPT

The Reading People Tracker (RPT) was developed at the University of Reading in England. It was developed in the context of the Ph.D. thesis [Sie03] of Nils T. Siebel, who already published some papers on robust realtime people tracking ([SM02],[SM01b],[SM01a]).
The RPT was designed as a person tracking module for the `ADVISOR`-project [1], where they built 'an integrated realtime surveillance system for use in underground stations'. In the `ADVISOR`-setting it ran as just a tracking module, processing input from an external video capturing module with a motion detector (see section 3.2.1), but it also has an integrated motion detector and the possibility to directly feed images into the system, so it also runs as a stand-alone program.
One nice approach of the RPT is the splitting of problems into smaller parts and solving them by (communicating) modules (*divide and conquer*).
The main features of the RPT are the following:

- It is a finished and tested project and operates in realtime under real-world conditions.

[1] Annotated Digital Video for Intelligent Surveillance and Optimised Retrevial, EU project code IST-1999-11287, `http://www-sop.inria.fr/orion/ADVISOR/`

Figure 2.1: The Reading People Tracker with tracked outline shapes [SM02].

- It was designed to be integrated into a bigger surveillance system.
- It uses a sophisticated person model to distinguish persons from other 'moving' objects, like opened doors and so on.
- The images are filtered, so that we can deal with noisy images up to a certain degree.

2.1.2 Tracking Groups of People

S.J. McKenna et al. develop in their paper [MJD+00] a quite sophisticated tracking system (TGOP), which can operate in an indoor as well as in an outdoor setting and can handle occlusion of persons via a depth ordering estimation.
The robustness is mainly achieved by using the RGB color information in probabilistic models and some interesting approaches to track persons that come together to form groups, eventually occluding each other, and split again.
Furthermore, simple interactions with objects can be detected, based on the (very) simple idea that carried objects that are dropped will give rise to new (small) regions, detectable by the motion detector. This part is also one of the highlights of this work, as it uses an *adaptive background model* (see 3.2.1), incorporating changes in the first-order image gradients (see Glossary, Appendix A) and chromaticity. This helps to overcome problems with shadows.
This tracker uses a very simple person model mainly based on color models (color histograms), which can make it difficult to delinate persons from other moving objects.

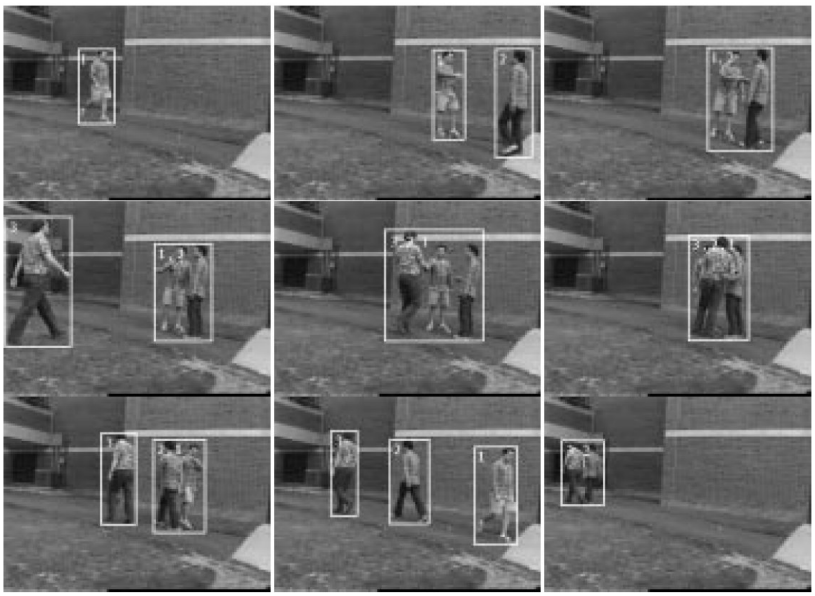

Figure 2.2: The TGOP tracker: Tracking people as they form groups and separate again [MJD+00].

Also the tracking part is quite simple and bases on matching overlapping bounding boxes of persons for tracking them.

2.1.3 W4 / W4S

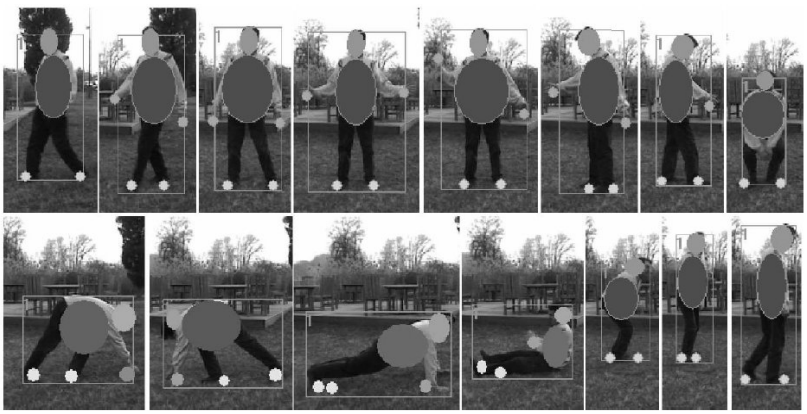

Figure 2.3: The W4 tracker with detected body parts [HHD00].

Haritaoglu et al. present in their paper [HHD00] a very interesting approach for tracking persons and their *body parts*, like head, hands, torso and feet without skin detection.

They use dynamic models of the human body to answer questions who is doing what, where and when in an outdoor environment, hence the name W4.
Another important fact is that W4 relies on monochromatic images, i.e., does not use color information, as for pure person tracking this is not needed.
W4 suffers from the problem to track persons through occlusions by static objects or other persons. To overcome this, the authors invented W4S [HHD98], which incorporates a realtime stereo system (SVM). Now, the authors could track persons in $2\frac{1}{2}$D, as SVM does not really gives you $3D$ information. But the data of SVM is enough to robustify the W4 tracking system to deal with sudden illumination changes, occlusions, merging and splitting of person groups, operating on a dual Pentium PC.

2.1.4 EasyLiving

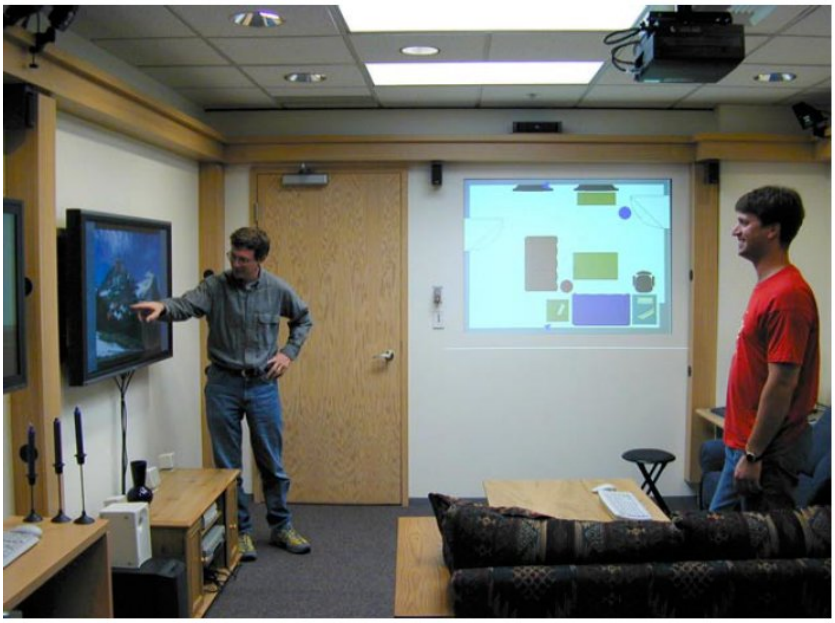

Figure 2.4: The intelligent environment of the EasyLiving-Tracker [KHM+00].

In [KHM+00] the authors develop a really sophisticated person tracking system for Microsoft's intelligent ubicomp environment, called 'EasyLiving'.
The main thing to mention here is that very expensive stereo cameras are used, which offer $3D$ informations without any further computing effort by the tracker. So, a homography estimation is not needed and the depth information will be more accurate then any estimate by a homography could be.
Furthermore, a statistical background model is also used, like in [MJD+00]. Here, sets of blobs (clusters) belonging to one person (see section 3.2.1) are extracted via a sophisticated algorithm, working on a graph where blobs are nodes and arcs are labelled with the distance between two blob centroids. This algorithm extracts the best clustering of

blobs, where each cluster corresponds to one person.
Another nice feature is that also here, color information, to be more exact color histograms, are used to maintain the identity of persons if they become occluded or leave the view of the camera.

2.2 Comparison to MaMUT

To conclude this chapter we want to summarize which ideas we could adapt and which we could not use while developing MaMUT. Finally, we give a tabularly comparison of features of the approaches presented in this chapter and the features of our system, MaMUT.

2.2.1 Similar Approaches

The *RPT* uses a nice divide and conquer strategy to distribute tasks to different modules with subsequent fusion, an idea which we considered to be useful for fusion of results from macro- and micro-tracking.

The *TGOP*-tracker gave rise to quite some ideas for our system, like the statistical background model, the region merging idea, the simple person models (color histograms) and the region tracking part (overlapping bounding boxes).

W4S uses a stereo vision system, which we judged to be too expensive for MaMUT. On the other hand the idea of tracking body parts influenced our micro-tracking approach.

Also in *Easyliving*, expensive stereo cameras are used, but the clustering of person blobs inspired our clustering of hand regions, as described in section 4.11.1.

Finally, we had to realize that no system does person tracking, detection of body parts and fusion of results as we wanted and needed it, therefore we decided to accept the challenge of developing a tracking system 'from scratch', which was also supported by the fact that except for the RPT, no source code is available for the mentioned trackers.

2.2.2 Comparison of Features

As promised, we will now list features of presented trackers and compare them to MaMUT.

	RPT	TGOP	W4S	EasyLiving	MaMUT
Runs with off-the-shelf PC-hardware	x	x			x
Uses off-the-shelf cameras		n/a			x
Incorporates more than 1 camera	x			x	x
Uses a sophisticated background model		x	x	x	x
Uses homography estimations					x
Tracks body parts			x		x
Detects interaction with objects		x		x	x
Does skin detection					x
Incorporates color person models		x		x	x
Fusion of results from multiple cameras	x			x	x

Table 2.1: Comparing features of related work to MaMUT.

Chapter 3

Theoretical Background

In this chapter we want to present the theoretical background needed to understand the implementation of our system.

Section 3.1 fixes the syntax we will use to formally talk about Computer Vision tasks.
Section 3.2 describes the tracking of persons, including background subtraction to initially detect movement in images, followed by blob extraction and region detection to detect regions (bounding box rectangles) encompassing moving persons.
Section 3.3 deals with the homography estimations we used to approximate the position of persons in the ubicomp environment, our small instrumented room, by mapping image coordinates to coordinates on the ground-plane, the floor, of our room, and finally **Section 3.4** will introduce our approach for detecting skin pixels in images from micro-tracking, based on histogram back projection.

3.1 Syntax

Before we can dive into detail, we need to fix a syntax which we can use to formally talk about Computer Vision problems.

Most of the time we will have to deal with RGB color images. In general, we consider a *RGB color image* as a tuple $\mathbf{f} = (f, width, height)$, with a function $f : \Omega \longrightarrow \mathbb{R}$ and dimensions $\mathbf{f}.width$ and $\mathbf{f}.height$.
Here, Ω is the image domain, usually a rectangle with width (x-dimension) $\mathbf{f}.width$ and height (y-dimension) $\mathbf{f}.height$, so $(x, y) \in \Omega \subseteq \mathbb{R}^2$, if and only if $x < \mathbf{f}.width$ and $y < \mathbf{f}.height$.
The co-domain is $\mathbb{R}^3$, representing the three *color channels* red, green and blue, so $(r, g, b) \in \mathbb{R}^3$.

To make such images representable in a computer, of course we have to discretize the infinite domain (*sampling*) and co-domain (*quantizing*) of $\mathbf{f}$. This yields a *discrete RGB color image* $\mathbf{f}$, which is represented as a function $f : \mathbb{N}^2 \longrightarrow \mathbb{N}^3$, or to be more exact if we have a 1-byte coding of our three color channels:

$$f : [0, \mathbf{f}.width - 1] \times [0, \mathbf{f}.height - 1] \longrightarrow [0, 255]^3.$$

Every single pair (x, y) is called a *pixel* of $\mathbf{f}$ and every function value $f(x, y) \in [0, 255]^3$ is called *color value* of pixel (x, y).
For convenience, we will write $\mathbf{f}(x, y)$ instead of $f(x, y)$, to make it easier to distinguish between images and other things.

Sometimes we have to deal with *greyscale images*. Here, pixels just have one grey-value $\in [0, .., 255]$, so $f(x, y) \in [0, .., 255]$.
Even simpler are black and white images, so called *binary images*, where each pixel is either black (0) or white (1), so $f(x, y) \in [0, 1]$.

With these notations, a *video stream* of a camera can be described as sequence of images $< \mathbf{f_0}, \mathbf{f_1}, \ldots >$, called *frames*, with an unique frame id.

3.2 Person Tracking

In this section, we want to give the reader an introduction to the theoretic background of realtime person tracking.
We have to note that most of these algorithms are also used to track hand- and arm-regions of persons, as described in section 4.4.
Another thing is that some algorithms are very complex in detail. Here, we will in the context of this thesis only give the coarse idea and refer to literature or to Appendix B, dealing with algorithm details.

The organization of this section is as follows. First, we describe an adaptive statistical background subtraction algorithm, which classifies movement in images. After that we describe how to extract blobs of moving pixels and merge those blobs to person regions, which we finally track.

3.2.1 Motion Detection

Our motion detection module will be applied to image sequences from *macro-* and *micro-tracking*. In the first case, we detect movement of persons, and in the latter, movement of hands and arms of persons located in areas of special interest.

The first step in every tracking module will be to detect where motion occurs in the currently processed frame $\mathbf{f}$ of our video sequence, i.e, we want to get a so called *motion image*, which is a binary image where every moving pixel has value 1 and every other pixel is set to 0.
Then, we can find *connected moving regions*, so called *blobs* and finally assign rectangular *bounding boxes*, so called *regions*, to all blobs. These regions should then correspond to persons, so called *person regions* (for macro-tracking) or to hands of persons, so called *skin regions* (for micro-tracking) we can track through consecutive frames.

Background Subtraction

To classify moving pixels in our frame $\mathbf{f}$, we use the widely accepted *background subtraction* technique ([Sie03], [MJD$^+$00], [KB01], [LHGT]).
The basic idea here is to compute a model m of the background for all pixels (x, y) in our images, i.e, $m = m(x, y)$.
Then we can pixel-wise 'subtract' every captured frame $\mathbf{f}$ from m and threshold the result to classify moving pixels, i.e,

$$moving(x, y) \Leftrightarrow ||\mathbf{f}(x, y) - m(x, y)|| > T(x, y),$$

with a threshold parameter $T(x, y)$, which may be constant for all (x, y), or different for each pixel.
Note that $||\cdot||$ can be any defined norm on the deviation of the current pixel from its

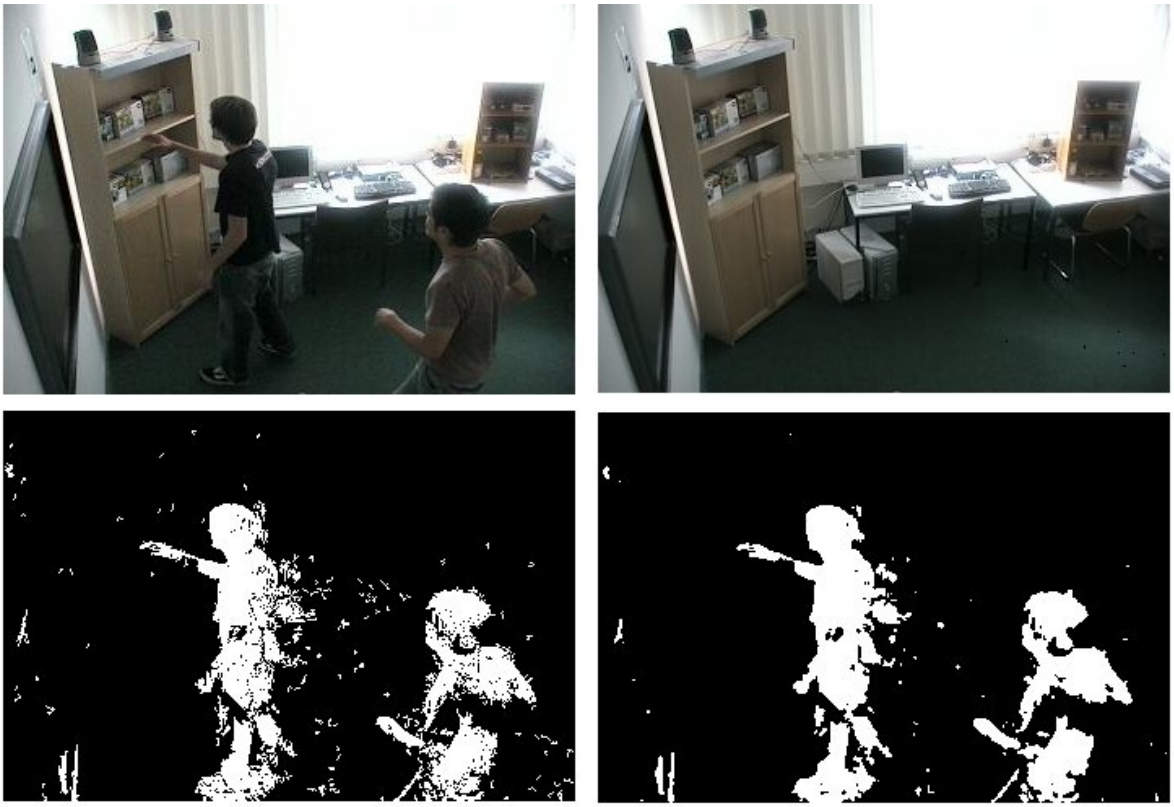

Figure 3.1: **Top Left:** Frame from camera. **Top Right:** Background model. **Bottom Left:** Result of background subtraction. **Bottom Right:** Result after filtering

model.
This allows us to delinate moving, foreground pixels from background pixels and the result of the process is usually a binary motion image (see figure 3.1) where moving, foreground pixels are white (have value 1) and static background pixels are black (have value 0).

Adaptive Statistical Background Subtraction

Easy approaches like used in [Sie03] suffer immensely from illumination changes and have problems when facing moving background objects or static foreground objects, which will create significant errors in the classification of movement. Furthermore they require a 'training phase' with no persons in the room.
We can only overcome these problems if we use a more sophisticated background model. In our case, we used the idea originating from [KB01].

Coarse Idea of a Statistical Background Model: The coarse idea of our approach is to use a sophisticated, *adaptive statistical* model of the background, i.e, we

model every pixel (x, y) with its *expected mean color value* $\mu(x, y)$ and standard deviation $\sigma(x, y)$ as a stochastic process, the so called *pixel process*, modelling the color values in the previously observed frames.
So, our model m is given by

$$m(x, y) = (\mu(x, y), \sigma(x, y))$$

Note, that we use color information and so we will get vector valued results $\mu(x, y) = (\mu_r(x, y), \mu_g(x, y), \mu_b(x, y))^T$ and the same for $\sigma(x, y)$.
This approach will help to better cope with above problems.

Another important fact is that here, the thresholding will of course use the statistical information, yielding a different threshold for every pixel:

$$T(x, y) = n \cdot \sigma(x, y) \ , \ 2 \leq n \leq 3,$$

which is used to classify moving pixels according to their deviation from the statistic mean (the model) in the current frame. This brings a huge advantage to simple approaches with a constant threshold. In regions with less changes, the threshold is small and so we get very sensitive results here. In contrast, if we have regions with lots of changes, we have a large threshold, which makes them less sensitive to noise.
So the classification of motion pixels in frame $\mathbf{f}$ in a binary motion image $\mathbf{d}$ is obtained by:

$$\mathbf{d}(x, y) = \begin{cases} 1 & \text{if } |\mathbf{f}(x, y) - \mu(x, y)| > T(x, y) \\ 0 & \text{else} \end{cases}$$

Note, that we also use the color version of $\mathbf{f}$, so actually $\mathbf{f} = (\mathbf{f_r}, \mathbf{f_g}, \mathbf{f_b})^T$.

This model is easily adapted online, by just updating μ and σ by the current measurements in frame $\mathbf{f}$.

An Adaptive Gaussian Mixture Model: The statistical model actually used is a so called *Adaptive Gaussian Mixture Model*, which is a multi-color background model, modelling each background pixel at time N by a *mixture of K Gaussian probability distributions* for all of its K color values. To adapt to changes in the scene, we have to update for all pixels those distribution out of the K which describes a pixel best, i.e., the color that is most significant to the pixel.
The updating in accordance to the actual pixel value is done by an (online) *Expectation Maximization (EM) algorithm*. This is an iterative method for optimizing a Gaussian mixture model, which is guaranteed to converge to a local maximum. It is described by quite sophisticated update equations we will present later in Appendix B.1.
There exist several classes of those algorithms and the one used here are so called *parametric estimation probability density functions*, originating from Nowlans Ph-D.

thesis [Now91]. To be more exact, we mainly use the *L recent window* technique, as described in the work of McKenna [MRG97].
It was first proposed in its full beauty by Grimson et al. in [SG99] and originates from the work of Friedman and Russel [FR97].

Creating the Gaussian Mixture Model: In detail, the background modelling looks like the following:
At each time N , $N \geq 0$, each background pixel $\vec{x}_N := (x, y)_N$ is modeled by a mixture (weighted sum) of K Gaussian probability distributions $\eta(\vec{x}_N; \mu_k, \sigma_k)$, $1 \leq k \leq K$ for K different colors. The parameter K with $3 \leq K \leq 5$ is chosen w.r.t the available time and computational power, a higher K requires more computational effort, but will yield a more detailed model.
The distributions (components) are obtained and updated by the history (the 'pixel process', as the authors call it) of the pixel values in the previous frames, i.e, $< \mathbf{f}_1(\vec{x}), \mathbf{f}_2(\vec{x}), \ldots, \mathbf{f}_{N-1}(\vec{x}) >$, which describes a stochastic process.
The weight parameters w_k used for the weighted sum (the mixture) correspond to the time proportion of color k , $1 \leq k \leq K$, staying in the background.

The actual computation of the model computes for every pixel its probability of having color value $\vec{x}_N$ at time N as

$$P(\vec{x}_N) = \sum_{j=1}^{K} w_j \cdot \eta(\vec{x}_N; \mu_j, \Sigma_j),$$

where w_K is the weight parameter of the k-th Gaussian component, and $\eta(\vec{x}_N; \mu_j, \Sigma_j)$ is the *Normal distribution* of the k-th component at pixel $\vec{x}$, with mean μ and covariance matrix $\Sigma = \sigma^2 \cdot \vec{I}$, where σ^2 is the variance.
In detail, we get:

$$\eta(\vec{x}_N; \mu_j, \Sigma_j) = \frac{1}{(2\pi)^{\frac{D}{2}} |\Sigma_k|^{\frac{1}{2}}} e^{-\frac{1}{2}(\vec{x}_N - \mu_k)^T \Sigma_k^{-1} (\vec{x}_N - \mu_k)}.$$

Updating the Gaussian Mixture Model: For every captured frame $\mathbf{f}$, we have to update our model, hence the notion *adaptive*.
The update method is selective, i.e, it compares every pixel value $\vec{x}_N$ at time N to the existing model components, the K Gaussian distributions w.r.t to a *fitness measure* w_k/σ_k.
The first matching component is updated by the current pixel value $\vec{x}_N$ according to the update equations of the EM-algorithm.
A match for component k is defined if $|\vec{x}_N - \mu_k| < 2.5 \cdot \sigma_k$.
If no match is found, the least probable distribution is replaced by a new Gaussian with $\mu_k^{N+1} = \vec{x}_N$, a large $\Sigma_k^{N+1} = \sigma^2 \cdot \vec{I}$ and a small w_k^{N+1}.

The idea of [KB01] is to use two different EM-algorithms, which will improve the performance of the method in [SG99].
In the beginning we use an expected sufficient statistics update method, which gives good estimates in the beginning and improves the quality of later estimates.
If we have observed more then L frames, i.e., our pixel process has advanced, we can skip to a L-recent window method, which only takes into account the last L frames, which is more appropriate, as more recent changes are more important then older ones.

If the reader is interested in the detailed update equations, he can look them up in Appendix B.1.

Classifying Motion: To finally classify motion, we exploit the fact that a moving object will create a high variance in the color space, due to reflecting surfaces during movement. This implies the fitness measure w_k/σ_k, inversely proportional to the variance and measuring how well an observed pixel fits its model.
Now the idea is that the background contains only $B < K$ highest probable colors, where the probability of a color being a background color is proportional to the time and staticness of it in the history. So we only need B Gaussian distributions.
To compute B, we order the K distributions w.r.t the fitness measure w_k/σ_k and only use the first B with $B < K$ for our background model, where

$$B = \text{argmin}_b(\sum_{j=1}^{b} w_j > T).$$

The threshold T is the minimal fraction allowed to be used in the model and is set empirically in accordance to the following tradeoff. Small T lead to an unimodal model (B is small), which is easy to compute, but not very robust, e.g., to small movement of leaves of a tree in the wind or similar phenomenas.
Large T lead to large B, which increases robustness, but is harder to compute.

To classify moving pixels in a frame f, we can now check if its value $\vec{x}$ deviates more than $m \cdot \sigma_b$ for *any* of the B distributions.
Usually we take a factor m of 2.5, yielding for the foreground motion image $\mathbf{d_N}$ at time N:

$$\mathbf{d}_N(x,y) = \begin{cases} 1 & \text{if } \exists\, b \leq B : |\vec{x}_N - \mu_b| > 2.5 \cdot \sigma_b \\ 0 & \text{else} \end{cases}$$

Parameters: As usual, this algorithm takes some parameters one has to specify initially. They are used to adapt the algorithm to the setting and get hopefully better segmentation results. As mentioned in [SG99], we mainly use two significant parameters, namely T and α.
The threshold $T \in [0,1]$ controls the proportion of the data that should be accounted for by the background, i.e., it is the 'minimum prior probability that the background

is in the scene' [KB01] and the learning constant α controls how fast the background model is adapted online to captured images, i.e., how fast objects are incorporated into the background.
Furthermore, we have to specify the number K of Gaussian distributions we want to use. As mentioned, larger K lead to more robust results, but will also increase the computational complexity.
We also need to set the threshold parameter m, which tells us at which deviation from σ we classify a pixel as foreground.
In addition, there are also two initial parameters, we have to set, namely w_{init}, the initial weight w_K of each of the K Gaussian distributions, and σ_{init}, the initial standard deviation for each of the K Gaussians.

Our values for those parameters can be found in section 4.7.1.

Blob Extraction

To obtain blobs from the binary motion image $\mathbf{d_N}$, we can use a simple *connected components labeling algorithm*.

But before we can assign rectangular bounding boxes (regions) to the extracted blobs, we need some filtering to cope with noise induced by our cameras.
The image $\mathbf{d_N}$ will suffer from a lot of tiny $1-3$ pixel large noise blobs, which were classified as moving due to difference to the model, although no movement occurred there. An efficient and easy way to remove such small noise blobs is to use a median filter (see Glossary, Appendix A).
The filtered image we will call $\mathbf{d_N^f}$, and use it for the actual blob extraction depicted above, followed by the region detection.

Region Detection

Ideally, we would now have one blob per person and so just assign a minimal bounding rectangle r to every extracted blob in the filtered motion image $\mathbf{d_N^f}$.
Formally, we specify r by its upper-left origin $(r.x, r.y)$ and its width and height $r.width, r.height$.
For later purposes, we will also compute a *tracepoint* c of every bounding box. This is just the middle of the lower horizontal bounding of the rectangle. Formally, $c(r).x = r.x + (r.width/2)$ and $c(r).y = r.y + r.height$.

Unfortunately, most of the time our persons will consist of two or even more blobs (see figure 3.1), due to low contrast areas which were not classified as moving beforehand.
A remedy comes from initially assigning regions to all blobs and then merge regions

which will most probable constitute a person.
The approach proposed in [MJD+00] is to merge all regions r_i, r_j with $i \neq j$ where four conditions hold for a new, larger region r_{new}, where r_{new} is just a minimal rectangular bounding box that encompasses r_i and r_j, i.e., $r_i \subseteq r_{new}$ and $r_j \subseteq r_{new}$ and $r_{new}.width \cdot r_{new}.height$ is minimal (see figure 3.2).
The four merge conditions are:

Horizontal overlapping: The projections of r_i and r_j to the x-axis (denoted by r_i^x and r_j^x), have to overlap,
i.e., $r_i^x \cap r_j^x \neq \emptyset$, with $r_i^x := \{\xi \mid r.x \leq \xi \leq r.x + r.width\}$.

x-distance of regions: The regions r_i and r_j must have a distance in x-direction which is smaller then $T_{dist} := 5$,
i.e., if $r_1.x < r_2.x$ then $r_1.x - (r_2.x + r_2.width) < T_{dist}$,
else $r_2.x - (r_1.x + r_1.width) < T_{dist}$
This means that we merge regions if they are less than T_{dist} pixels in x-direction away from each other.

Area of regions: The area of r_{new} is supposed to be larger then $T_{area-min}$ and less then $T_{area-max}$,
i.e., $T_{area-min} < r_{new}.width \cdot r_{new}.height < T_{area-max}$.

Ratio of regions: The aspect ratio of r_{new} should be less or equal to 1,
i.e., $r_{new}.width / r_{new}.height \leq 1$.
This is due to the fact that persons are normally higher then wide. This also implies that this threshold will *not* be applied for the detection of moving skin regions by the micro-tracking.

3.2.2 Region Tracking

As we have finished to detect moving regions in an image, we can now turn our attention on how to track those regions and hence the corresponding persons (for macro-tracking) or skin-regions (for micro-tracking) through consecutive frames of our video sequence. In the remainder of this section, we will mainly talk about persons, but if not said otherwise, all the mentioned concepts can also be used for skin regions.
We will first explain how to store tracking informations and then turn our attention to some common tracking algorithms.

Storing Tracking Results

To do so, we keep a *tracking database data structure*, inspired by [MJD+00], where we mainly store the position of the bounding boxes, a history of former bounding box positions and (for persons only) the *tracking state of that person*.
The state consits of several boolean variables, where the first one distinguishes between

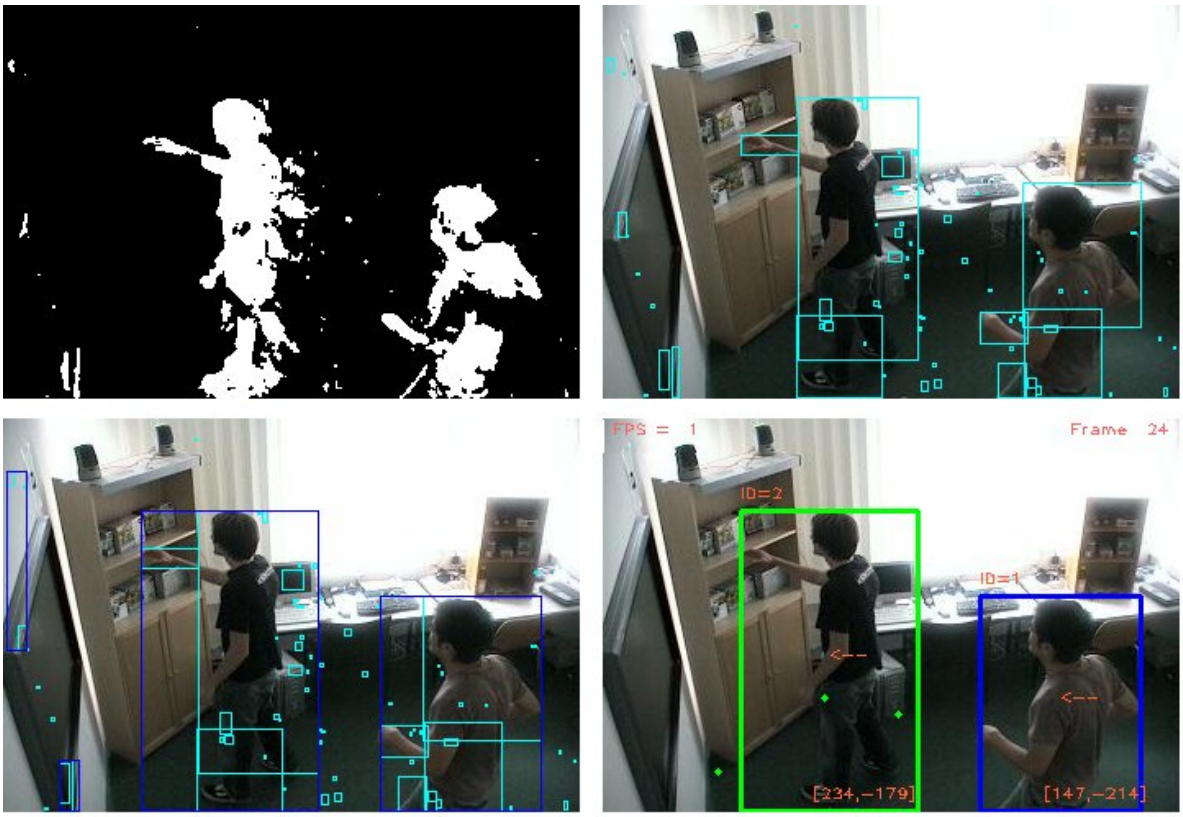

Figure 3.2: **Top Left:** Result of background subtraction. **Top Right:** Initial regions (one for each blob of background subtraction). **Bottom Left:** Merged regions (blue). **Bottom Right:** Tracking result.

moving currently tracked persons and persons, where we lost the track because they are standing still and hence are incorporated into the background model and no longer detected by our background subtraction algorithm.
The second state variable is *reliable*. A tracked person is considered as being reliable, if we track it for two or more consecutive frames. This helps to remove *pseudo-person* regions, existant for one frame, which result from sudden illumination changes. The existence of those pseudo-persons can be explained by the fact that objects like shelfs are detected as person regions if their illumination changes suddenly, caused by two factors. First, their appearance model (size and aspect ratio) matches our person model and second, they are marked as moving by the background subtraction algorithm, because the color values of their pixels changed enough due to the varying illumination.
A third state variable is *inside*, which distinguishes between persons inside the view of our camera and ones which have left the view of the camera. This distinction is useful, as we do not want to lose the last tracking information of persons who became static and are therefore incorporated into the background and no longer detected. The distinction if a person just became static or left the view is simply done by checking if the last known position of a persons bounding box is close (about 5 pixels) to the left or right border of the camera image. If so, we conclude that a lost person may probably has left the room and else we just consider it as non-moving.

Tracking Algorithms

Overlapping Bounding Boxes In [MJD+00], the authors propose to just track persons by overlapping bounding boxes, which is easy to implement and to compute.
This approach tracks a person by assuming that its movements will be so small with respect to the frame rate of our cameras, which will lead to the fact that the bounding boxes of persons will overlap in consecutive frames.
In every newly captured frame $\mathbf{f}$, we then just have to check for every detected moving region i, if its bounding box r_i overlaps with any bounding box r_j of a person in the tracking database. If so, we have to update the informations in the database by the current position. If no overlapping bounding box is found, we assume that we found a new person and have to add it to the database.
Overlapping in this case can be formalized by checking if two bounding boxes, i.e, rectangles, have both overlapping projections to the x- and to the y-axis. Formally:
$r_i^x \cap r_j^x \neq \emptyset$ and $r_i^y \cap r_j^y \neq \emptyset$,
with $r_i^x := \{\xi \mid r.x \leq \xi \leq r.x + r.width\}$ and $r_i^y := \{\eta \mid r.y \leq \eta \leq r.y + r.height\}$.

Color Models In [MJD+00] as well as in [KHM+00] it is proposed to use *color histograms* $H_i(x)$ of a person i to maintain its identity through occlusion (see Glossary, Appendix A for details on histograms). This approach is inspired by the idea that the histogram of a person will not significantly change during movement and change of pose, as it simply counts which color occurs how often in the person, and not where. Such histograms can be obtained from the *mask* of a person, i.e., the pixels in a person region

which were classified as moving by our background subtraction algorithm.
In [MJD+00], such a histogram is used to compute the probability of pixel x belonging to a certain person i via

$$P(x|i) = \frac{H_i(x)}{A_i},$$

where A_i denotes the number of pixels in the mask of person i.
In [KHM+00], the comparison of stored histograms and newly computed ones is done via a *histogram intersection* algorithm [SB91]. The histogram intersection algorithm outputs a value between 0 and 1, where higher values correspond to higher similarity of the histograms to be compared.

It is essentially to note that we have to adapt the histogram of each person during the tracking task. We chose to update the histograms cumulative, i.e., we add newly gathered histograms to the stored, old one and normalize afterwards. This normalization normalizes the sum over all histogram bins to 1, i.e., $\sum_i H(i) = 1$.

For *skin regions*, we cannot use any color model, as every skin region should have more or less the same color, a fact we actually exploit for detecting skin in images.

Forming Groups of Persons

In [MJD+00] or [HHD98], the authors use the notion of *groups* of persons. A group is just a set of tracked persons, which are close together. This notion is inspired by the fact that persons standing close together are hard to delinate by the background subtraction algorithm and often extracted as one, over-dimensional person region.

For skin regions, grouping is *not* interesting.

3.3 Homography

In this section, we present our approach for estimating the position of the tracked persons in our room. This means, we will usually use images and tracking results from our macro-tracking cameras. But we will also apply this approach to images from our micro-tracking cameras to obtain the position of skin regions in a coordinate system of shelfs for example, which we will call *shelf coordinate-system*.
In the remainder of this section we will mainly talk about the estimation of person positions. These approaches will also work for estimating shelf coordinates of skin regions.

3.3.1 Idea

The basic idea is that we use *static* cameras, i.e, cameras that do not change their position during the tracking task.
Furthermore, we exploit the fact that persons will stand on the *ground plane*, the floor of our room and so we can describe every person's position by its tracepoint coordinates on the ground plane. Remember, the tracepoint of the rectangular bounding box of a person was just the middle of the lower horizontal bounding of the rectangle.

The problem now is how to map the *image coordinates* in which we measure the tracepoints of our camera images to *world coordinates* of our room.

3.3.2 Approach

The approach we follow is to use the so called *Direct Linear Transformation (DLT) algorithm*, as described in Hartleys book [HZ04].

Input and Output

As input, the *DLT* algorithm uses n , $n \geq 4$, point correspondences $\underline{x_i} \leftrightarrow \underline{x_i'}$ for $i = 1, \ldots, n$, i.e., we measure for n points in (homogeneous) image coordinates $\underline{x_i} \in \mathbb{R}^3$ (see Glossary, Appendix A) the corresponding world coordinates of our room $\underline{x_i'}$ (see figure 3.3).
As notation we will use: $\underline{x_i} = (x_i, y_i, w_i)^T$, with an arbitrary scale factor w. The result of the computations of the *DLT* algorithm will be a matrix $H \in \mathbb{R}^{3\times3}$, s.t.,

$$H\,\underline{x_i} = \underline{x_i'}\,,\ \forall\, i \in \{1, \ldots n\}.$$

So, H is a so called *projective mapping* from image- to world-coordinates, which is exactly what we want to have.

For mathematical details on this algorithm, we refer the interested reader to Appendix B.2.

Figure 3.3: Our room with the 4 point correspondences $x_1, \ldots, x_4$ (red), which embrace the ground plane of our room (blue).

3.4 Skin Detection

3.4.1 Motivation

One important feature of MaMUT is that it will not only detect and track persons in our room, but also *skin colored regions* by our micro-tracking. Hence, this module of our system will use images from the micro-tracking cameras.

3.4.2 Basic Idea

We use a quite simple, but still quite robust technique to detect skin colored pixels, forming skin blobs, which are finally grouped together to skin regions (cp. person regions) we are interested in.
The applied technique is the skin detection part of the *CAMShift Algorithm* [Bra98], and goes under the slogan *Histogram Back Projection*.
It uses a model of skin color, obtained by sample skin images, which are just small images containing only skin, e.g., small patches cut out from captured images of our cameras (see figure 3.4).
The important idea is that we convert our images from the usual RGB color space to the HSV (*Hue-Saturation-Value*) color space (see Glossary, Appendix A). Here, we only use the H-channel, the hue channel, of the images, which carries the informations about the color, which is crucial to distinguish skin from non-skin pixels. Furthermore, we ignore in this way lightning changes which will be only visible in the V-channel (value) and ignore variants in skin color (dark skinned versus bright skinned persons), only visible

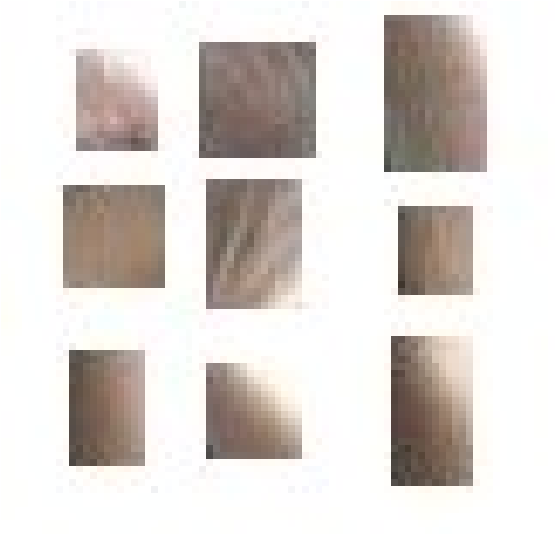

Figure 3.4: 9 small skin patches used to create our skin model.

in the S-channel, the saturation channel (see figure 3.5).

This model is used to compute for every pixel *in a moving region* of the actual processed frame (in a HSV version) the probability that this pixel is a skin pixel.

Simply thresholding this skin probability image yields a binary classification of skin pixels (c.p. classification of moving pixels for motion detection).

From RGB to HSV

As our images are stored in RGB format, we have to map them to HSV via a mapping

$$F : \mathbb{R}^3 \longrightarrow \mathbb{R}^3, (r, g, b) \mapsto (h, s, v),$$

where usually $r, g, b, s, v \in [0, 255]$ and $h \in [0, 360]$ (angle).

For the concrete mapping, the interested reader may refer to Appendix B.3.

Model

The initial step is to obtain a model of skin hue. Therefore we use a HSV version of n offline captured sample skin images.
For every of those images, we compute the hue-histogram (histogram of the H-channel), counting in its bins which hue-value occurs how many times in the image. The n histograms are simply averaged in the end to obtain a single histogram.
This histogram serves as a *skin hue probability distribution* $\mathcal{M}$, and therefore has to be

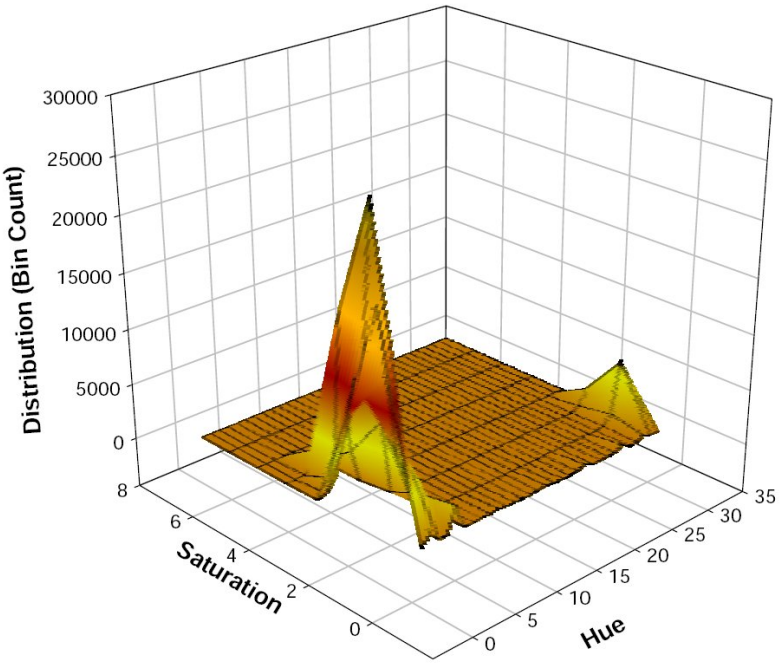

Figure 3.5: Example HS-histogram of human skin. Note that by far most of the hue values lie in the lower range from 0 to 10. **Author:** B. Kapralos [KJM03]

normalized in such a way that all of its bin contain values in the range $[0, \ldots, 255]$ and that the sum over all bins should equal 255.

Skin Probability Image

For every processed frame $\mathbf{f}$, we compute its HSV version $\mathbf{f}_{HSV}$ and only use the H-channel to compute, in moving regions only, the conditional probability of every pixel being skin, given its hue value h, according to our skin model $\mathcal{M}$ i.e.,

$$P(skin \mid h) = \mathcal{M}[h] \in [0, \ldots, 255].$$

This yields a greyscale *skin probability image* $\mathbf{S}$, where brighter pixels correspond to higher skin probabilities (see figure 3.6).

Thresholding, Filtering, Discarding and Merging

The final step is to threshold $\mathbf{S}$ at a reasonable threshold T_{Skin}, which yields a binary image $\mathbf{S}_{bin}$, classifying skin pixels.
This image is then median filtered to remove pixelwise noise.
But in spite of filtering, lots of tiny skin regions survive, due to erroneous classifications. These are simply discarded if their area is less than a skin region area threshold $T_{Skin-Area}$.

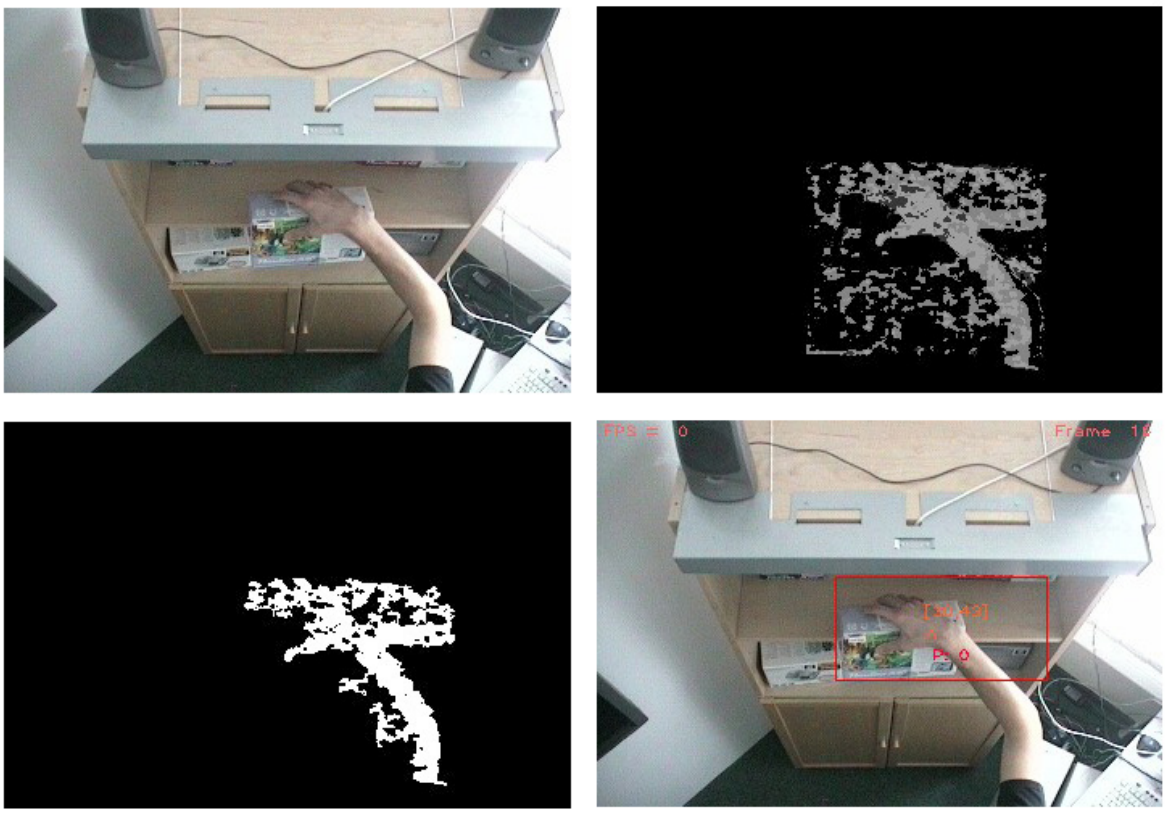

Figure 3.6: **Top Left:** Frame from one of the micro-tracking cameras. **Top Right:** Skin probability image. **Bottom Left:** Binary skin probability image, result of thresholding. **Bottom Right:** Resulting skin region (red), with position in shelf coordinates and ID of person it belongs to.

The last problem is, that we still may get multiple skin regions which correspond to one skin object (head or hands).
Fortunately, these are quite close together and so, we can merge skin regions, if the distance of their centers is less than a threshold $T_{Skin-Dist}$.

Chapter 4

System Implementation of MaMUT

After having dealt with the theoretical background behind our tracking approach, we can now turn our attention on how to implement our ideas.

4.1 Tools

In this section we want to describe which tools we used to implement MaMUT.

4.1.1 Intel's Open Computer Vision Library (OpenCV)

As MaMUT is a tracking module with skin detection and homography estimation, it is clear that we heavily had to use Image Processing and Computer Vision algorithms. Furthermore we also needed a GUI to visualize our results.
All these needs are fulfilled by Intel's Open Source Computer Vision Library, in short OpenCV [Bra00].

Features

OpenCV encompasses a really widespread collection of Image Processing and Computer Vision tools. Starting from low-level functions for loading and accessing images up to elaborate Active Contour computations.
The areas covered by OpenCV are:

- Image creation and access
- Image arithmetic and logic operations
- Image filtering
- Linear image transformation
- Image morphology

- Color space conversion
- Image histogram and thresholding
- Geometric transformation (zoom-decimate, rotate, mirror, shear, warp, perspective transform, affine transform)
- Image moments

For a full overview of the features, one can have a look at Intel's OpenCV homepage [Int].

Features we actually used include

Background Differencing: Accumulate images and squared images, Running averages.
Used for background modelling and background subtraction

Thresholding: Binary, inverse binary, truncated, to zero, to zero inverse.
Used for obtaining binary images from greyscale images, e.g., to classify moving- and skin-pixels from greyscale difference image and greyscale skin probability image.

Histogram (recognition): Manipulation, comparison, backprojection.
Earth Mover's Distance (EMD).
Used for modelling skin hue and recognizing skin areas in person regions and for person color models.

Color space conversion: RGB↔HSV, RGB↔Greyscale, ...
Used for converting images to needed color spaces, e.g., HSV images for skin detection.

HighGUI: A GUI to display images opened via OpenCV's image access functions.

4.1.2 cURL

To capture images from our network cameras, which offer a CGI-interface via a HTTP-server, we decided to use cURL, a
'[...] command line tool for transferring files with URL syntax, supporting FTP, FTPS, TFTP, HTTP, HTTPS, TELNET, DICT, FILE and LDAP. [...]' [Haxa].
This tool provides a simple usage within C++, by just using system calls `system("curl ...")`, emulating command line interaction.
The call we used looks like this:

```
system("curl -u root:<password> -s
http://134.96.240.132:4444/axis-cgi/jpg/image.cgi?resolution=704x480
-o images/cam2.jpg");
```

which tells cURL to get data from the server `134.96.240.132`
(= `wwcam02.cs.uni-sb.de`), at port `4444` and folder
`/axis-cgi/jpg/image.cgi?resolution=704x480`, which actually starts a CGI request for the actual image with resolution 704×480 pixels.
We authenticate to the server as user `root` and with the known password.
The image data is stored in the local folder `images/cam2.jpg` and the flag `-s` tells cURL, to be silent, i.e., not to output (unpleasant) status messages to `cout`.

libcurl

A more elegant way of using cURL without UNIX-system calls would be to use libcurl [Haxb], which is a library, available for C, allowing to use features of the command line tool cURL in C/C++ programs.

4.2 Setup

Before we start to describe the actual implementation of our system, we want to shortly present the actual setup of our lab (SUPIE [BK]), which is relevant for MaMUT.

4.2.1 Room

As can be seen in figure 4.2, the main parts of our ubicomp environment, a small instrumented room are a shelf, where some products (digital cameras) are located. The shelf is also equipped with some loudspeakers, allowing to consult persons about the digital cameras inside.
On one wall, there is a big display which can be used to display all kinds of informations. On the ceiling, a movable beamer is mounted, which can project informations to the walls of our room.

4.2.2 Input: Cameras

The input of our tracker originates from from three mounted LAN-cameras of type AXIS 2130 PTZ (see figure 4.1).
These are Pan-Tilt-Zoom cameras which broadcast their captured images via Ethernet to a LAN and are also configurable by a Web-Interface. Therefore, each cam has a unique IP-address in the LAN and a domain name.

We use three cameras (see figure 4.2). Two mounted at corners of the room, to observe our room and track the position and movement of persons, the so called *room cameras* (macro-tracking).
The other camera is mounted above a shelf with products. This camera precisely tracks in detail the hands and arms of persons, who reach towards the products in the shelf, the so called *shelf camera* (micro-tracking).

Figure 4.1: **Left:** *Micro-tracking*: The shelf and the shelf camera. **Right:** One of our cameras.

Figure 4.2: **Top Left:** Image from a macro-tracking camera ($Room_1$). **Top Right:** Image from a macro-tracking camera ($Room_2$). **Bottom Left:** Image from a micro-tracking camera ($Shelf$).

4.2.3 Output: EventHeap

All applications who need information about the state of our room, e.g., PDAs displaying product informations, communicate over a common coordination infrastructure, the so called *EventHeap* [JF02], which will be the output destination for our tracking results. Details of the EventHeap and its usage by MaMUT can be seen in section 4.12.

4.2.4 Our Implementation

In the context we will implement a tracking system from scratch, adopting ideas from several sources.

4.3 Desired Results

To meet the requirements mentioned in section 1.1, we want our system to produce the following results for each frame of our video sequence, which then can be send to the EventHeap (see figure 4.3).

- *Number of persons* in the room $= n \Rightarrow person_0, \ldots, person_{n-1}$ (macro-tracking).
- For $i = 0, \ldots, n-1$: *Position* of $person_i$ in our room, represented as $2D$-coordinates of the ground plane, the floor of our room (macro-tracking).
- For $i = 0, \ldots, n-1$: Moving *direction* of $person_i$ (macro-tracking).
- For $i = 0, \ldots, n-1$: *Number of skin regions* for $person_i = k_i$ (micro-tracking)
- If $k_i > 0$: For $j = 0, \ldots, k-1$: *Position* of skin region j of $person_i$ in *shelf coordinates*, i.e., in a $2D$-coordinate system of the shelf front (micro-tracking).

4.4 Overview

First of all, we will give a brief overview of our system. To ease the understanding of the following, the reader may refer to figure 4.4.

To reach the desired results of our system described in section 4.3, we continuously *capture* images from n macro- and m micro-cameras, which do macro- and micro tracking as described in section 1.2.
With these images (to keep our figure simple we just set $n = m = 1$), we can detect *persons* in macro-images as well as *arms and hands* of persons reaching towards the shelf in micro-images.

These goals are reached by performing certain steps, which are partly equal for macro-

Figure 4.3: **Left:** Tracking result of one of the macro-tracking cameras: Person regions (blue and green, green denotes tracked persons at the shelf), Person ID's, moving direction and positions in room coordinates (orange). The 4 green markers denote the region in front of our shelf. **Right:** Tracking result of one of the micro-tracking cameras: Skin regions (red), moving direction, positions in shelf coordinates (orange) and ID of person the skin region most probably belongs to (red).

and micro-tracking.
First, we do a *background subtraction*, which yields a binary motion image, classifying motion in the images. With those images, we perform a *connected component labelling* to find connected *blobs*. These blobs now have to be merged, as usually we will observe more than one blob per person or hand. These merged blobs are grouped in *regions*, where we talk about person regions for macro- and hand-/arm regions for micro-tracking. For the micro-images, we will now have to do a *skin detection* based on histogram back projection in all found regions to detect hands, yielding so called skin regions.
Now, we can track skin- and person regions via *tracking* algorithms, mainly based on matching overlapping bounding boxes and color models. The results of this tracking tasks are stored in individual database structures (in our case *Tracking Database* and *Shelf Tracker*). Here, we also assign the position to those regions via *homography estimations*.
The next step is then to *fuse* tracking results of macro- and micro-tracking, i.e., assign detected hand regions to detected persons in our case.
These informations are all we were looking for and finally have to be stored to the *EventHeap* via *TrackerEvents*, coding all informations.

4.4.1 Overview of the Implementation

The actual implementation heavily uses functions of the OpenCV library [Bra00], which makes the code quite compact, we can get along with about 5000 lines of code in total.

All the parts of MaMUT which mainly deal with Image Processing tasks, like the back-

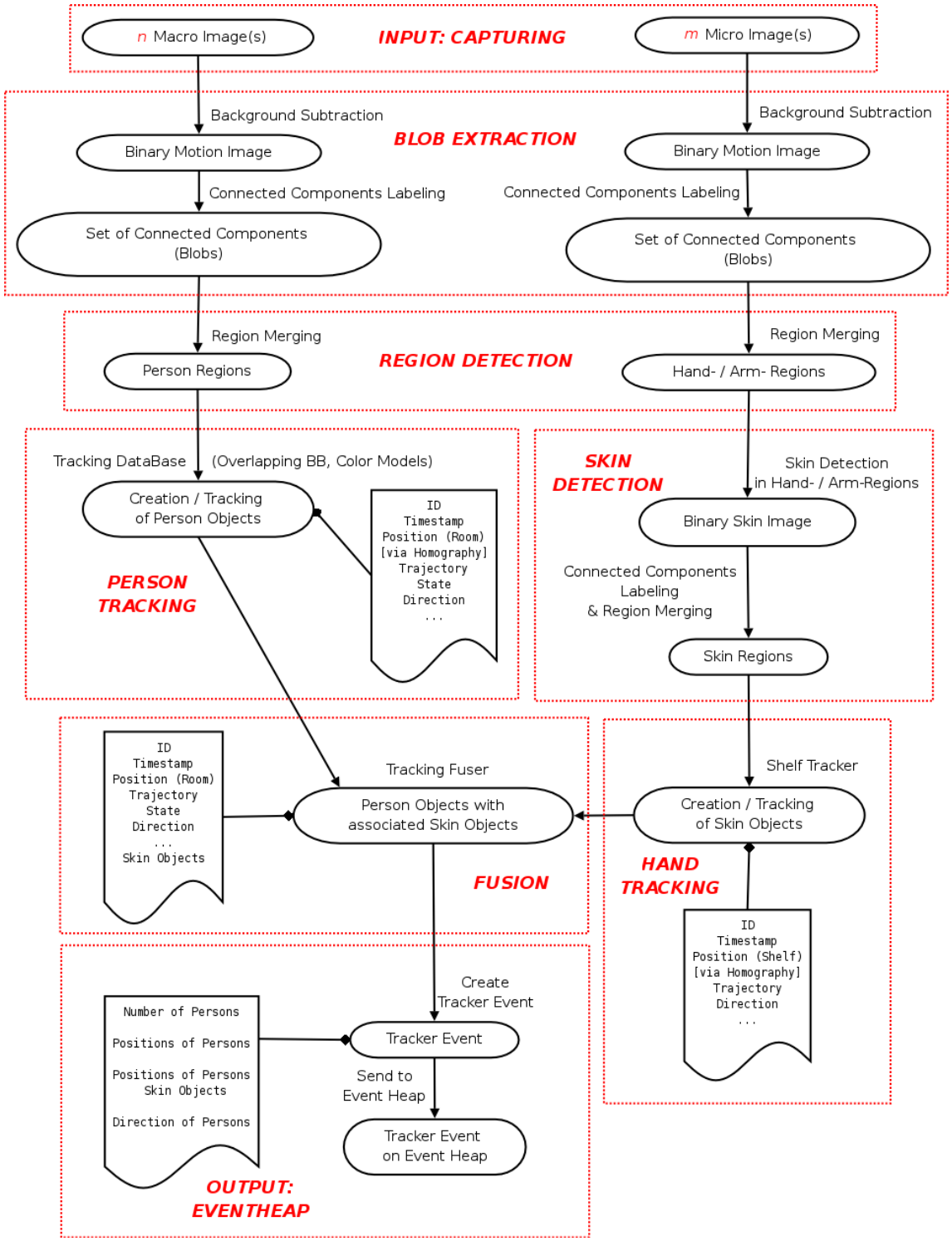

Figure 4.4: Overview of our system. Ellipses describe results, cambered rectangles attributes of associated objects, actions are denoted next to arrows and dotted red rectangles describe different modules.

ground subtraction, region merging, and so on are C/C++ functions stored in modules like `region_merging.cpp`.
All pure tracking parts, like the tracking database data structure, the EventHeap Output and so on are C++ classes, with header and implementation file, like `TrackingDB.h` and `TrackingDB.cpp`.

In our actual implementation, we used two macro-cameras ($room_1$ and $room_2$, *room cameras*) and one micro-camera ($shelf$, *shelf camera*). With the images from cameras $room_1$ and $room_2$ we will basically do the same things, as they are both macro-cameras. Now, we will present the implementation in detail.

4.5 Configuration Files

Before MaMUT can start its work, it has to know several configuration parameters, like the path of the input images, if we run in *HDD-mode* (loading images from harddisk instead of capturing live from the cameras), or all the threshold parameters.
These parameters are specified in a config file (`*.bcf`) which is passed to MaMUT as a command line parameter. For example: `./mamut ../files/BAIR.bcf`
The parsing of the file is done by a class called `ConfigFile`, which is available to download from [Wag]. For a detailed description, check out section C.2.

A commented example config file describing *all possible* (realtime- and HDD-) parameters is given in Appendix C.2.1.

4.6 Capturing: Network Cameras

As mentioned, MaMUT will use images from several mounted LAN-cameras of type AXIS 2130 PTZ.
In our case, the cameras are `wwcam01` for the micro-camera (*micro-tracking*) and `wwcam02` and `wwcam03` for the macro-cameras (*macro-tracking*), which we denote by camera $room_1$ and $room_2$, respectively.
Hereby it is important to note that the two used macro-cameras are *disjoint*, i.e., if we see a person in one camera image, this person can not be seen in the image from the other camera. Furthermore, they fulfill basically the same task, namely tracking persons, but camera $room_2$ does not observe any region of interest, like a shelf, so we do not need a shelf carpet (see section 4.11) there. All other important parameters, like parameters for the statistical background model or thresholds for region merging are the same for the two macro-cameras. The communication between them is actually established by using the same data structure to store tracking results.

The technical specifications of the cameras are as follows:

- *Resolution:* 704×480 or 352×240 or 176×112 pixels

- *FPS:* 12 fps @ 704 × 480, 30 fps @ 352 × 240 and 176 × 112
- *Images:* Motion- and Snapshot-images /Black and White or RGB-color in JPEG format, supporting 5 compression rates and timestamps

We use color (RGB) JPEG images with resolution of 352 × 240 pixels, which will give us enough details, and allow our algorithms to run fast enough for realtime tracking.
In MaMUT, we capture images via the CGI interface of HTTP-server of the cameras (for more detailed information, refer to the web page of the camera manufacturer [Axi]), which will offer some speed advantages, and make a point as we want to meet realtime requirements.
To do so, we use `cURL` [Haxa], a nice small command-line tool to get data from HTTP servers (for details see section 4.1.2).
It is accessed via a C++ system call:

```
system("curl -u root:<password> -s
http://134.96.240.132:4444/axis-cgi/jpg/image.cgi
?resolution=704x480 -o images/cam2.jpg");
```

which will store the actual image from the camera as `images/cam2.jpg`.
Unfortunately, we cannot use the data stream from the cameras directly, as this would require to manually decode the JPEG data to raw data, which then could be loaded as a `IplImage` object, the image class of OpenCV. But storing the JPEG images and loading them via the built-in load routine `cvLoadImage()`of OpenCV was found to be quite effective.

4.7 Person- and Skin-Region Detection

4.7.1 Background Subtraction

To obtain the Gaussian Mixture model of our background, i.e., an object of type `CvBGStatModel`, we just capture an image from our empty room `f`, specify some parameters `params` (see section 3.2.1) and do
`CvBGStatModel* bg_model = cvCreateGaussianBGModel(f, &params)`.
Interestingly, the adaptive Gaussian background model would even allow to start tracking if there are already persons in the room, but the results are better if we start with an empty room.

Empirically, we found the following parameters to work out well in our setting:
For the *room* cameras, we used

```
params.win_size      = 1000;     // win_size = 1/alpha
                                 // => alpha = 0.001
params.n_gauss       = 5;        // n_gauss = K
params.bg_threshold  = 0.8;      // bg_threshold  = T
params.std_threshold = 3.0;      // std_threshold = m
```

```
params.minArea       = 5;
params.weight_init   = 0.05;     // weight_init = w_init
params.variance_init = 60;       // variance_init = sigma_init
```

For the *shelf* camera, we used the *standard* parameters of OpenCV:

```
std_params.win_size      = 500;
std_params.n_gauss       = 5;
std_params.bg_threshold  = 0.7;
std_params.std_threshold = 2.5;
std_params.minArea       = 15;
std_params.weight_init   = 0.05;
std_params.variance_init = 30;
```

The actual background subtraction and model adaption is then done for all frames `f`, by just calling `cvUpdateBGStatModel(f, bg_model)`.
This yields the binary motion image as `bg_model->foreground`.

4.7.2 Filtering

To remove small noise blobs, we use OpenCV's median filter with a 3×3 mask `cvSmooth( .., .., CV_MEDIAN, 3, 0, 0)` for the room cameras and with a 7×7 mask for the shelf camera. The reason for using a larger mask for detecting skin regions at the shelf is that because of the large zoom-factor of the shelf-camera, we will face more noise here, but on the other hand we will also be more robust to stronger filtering.

4.7.3 Blob Extraction

OpenCV provides an efficient *connected component labeling algorithm* `CvSegmentMotion(...)`, which yields a sequence `CvSeq` of connected components `CvConnectdComp`. Such connected component objects encompass a `CvRectangle` object, which is just our bounding box.

4.7.4 Region Detection and Merging

The mentioned problem is now merging connected components (regions), which will most probable form a person.
To do so, we first remove every too small region from the sequence obtained by the connected component labeling, which will be due to noise. Therefore we just remove every region with area less than a threshold `TCC1` from the sequence. Then we start a merging method which checks in a nested `for`-loop every pair of remaining regions (`cc1`, `cc2`), if they should be merged.
To this end, we check if two regions' projections to the x-axis overlap or if the distances of their centers is less than a threshold `TD`. If this is the case, we merge them, i.e., we create a new `CvConnectedComp` object with a rectangle encompassing `cc1.rect` and

`cc2.rect` via
`CvMaxRect(cc1->rect,cc2->rect)`. This component is pushed to the end of the sequence and the two original components are removed.
After merging, we run another discarding function, which will remove small merged regions with area less than a threshold `TCC2` and regions with aspect ratio greater than 1.

Threshold Parameter Computation

The threshold parameters for region merging are not set statically, but initially computed according to the size of the input images $s := \mathbf{f}.width \cdot \mathbf{f}.height$ and a factor F via $T = s/F$.
Empirically we obtained the following factors for good thresholds:

For the *room-cameras*:
Step 1: First cleanup step to remove too large and too small regions.

- $T_{area-min} := s/700$
- $T_{area-max} := s/3$

Step 2: Final step to remove too large merged regions and regions with a strange aspect ratio ≤ 1.

- $T_{area} := s/17$
- $T_{dist} := 5$
- $T_{ratio} := 1$

For the *shelf-camera*:
Step 1: First cleanup step to remove too large and too small regions.

- $T_{area-min} := s/845$
- $T_{area-max} := s/2$

Step 2: Final step to remove too large merged regions and regions with a strange aspect ratio ≤ 1.

- $T_{area} := s/154$
- $T_{dist} := 5$

Important: If the according config parameter is set, it is also possible to adapt the threshold parameters online via trackbars!

Concrete values: In our case with an image size of 352×240 pixels, we obtained the following thresholds:

1. Room cameras:
   - $T_{area-min} = 14.98$
   - $T_{area-max} = 28160$
   - $T_{area} = 4969.41$
   - $T_{dist} = 5$
   - $T_{ratio} = 1$
2. Shelf camera:
   - $T_{area-min} = 99.98$
   - $T_{area-max} = 42240$
   - $T_{area} = 548.57$
   - $T_{dist} = 5$

4.8 Person Tracking

For tracking persons in consecutive frames, we mainly use overlapping bounding boxes and color models (see section 3.2.2).

Match Direction To robustify our system even more, we decided to also add a direction matching. This simple approach is inspired by the fact that due to heavy illumination changes, e.g., when persons are moving in front of a window, it can happen that even our sophisticated statistical background subtraction algorithm will fail and we can extract no moving regions during some (1 – 2) frames.
Hence, we will no longer have overlapping bounding boxes, as the movement through more than one frame might be too large.
To overcome this problem, we compute every time in which *direction* a person is moving, i.e., we just compare the x-coordinates of the tracepoints c_1 and c_2 of two consecutive bounding boxes.
The computation of the direction is then performed as follows:

$$direction(c_1, c_2) = \begin{cases} \textbf{Right} & \text{if } c_2.x \geq c_1.x \\ \textbf{Left} & \text{else} \end{cases}$$

This helps us to match a region to a person not only if the bounding boxes overlap, but also if the moving direction of the person would be preserved by the region.

4.8.1 Actual Tracking

We actually combine three approaches in a 3-step algorithm for *person tracking*:

1. Match newly detected regions with *moving* persons via *overlapping* bounding boxes **or** similar *color models*.

2. Match newly detected regions with *non-moving* persons via similar *color models* **or** *same direction*.

3. Else, we found a new person and have to add it to the database.

If we found a match, we update the person's status like bounding box, timestamp, tracepoint, and so on in accordance to the matched bounding box.

After this matching step is done for all newly found regions, we check which persons do not have been updated. For those persons we lost the track, and so we have two possibilities.
A first one is to set them *non-inside*, if the possibility of leaving the camera view existed, or if we lost their track for more than 5 frames.
The second possibility is to just set them *non-moving* and still update their status, if we still consider them as inside, as described above, and have not lost their track for more than 5 frames.

A final step consists of two checks.
The first one is to check if the *tracepoints projected to the ground plane* are actually located on the ground plane. This is needed because usually the lower parts of persons have low contrast to the floor. This yields too small bounding boxes which are still classified as persons, but do not encompass the feet region of a person and hence yield erroneous tracepoint coordinates when projected onto the ground plane.
This check simply assures that the second coordinate of our projected tracepoint is smaller or equal to 0, as our ground plane is the $x, -z$–plane of our room (see section 4.10).
The second check verifies if person regions did not became too large, i.e., too wide or too high. This can happen due to reflections next to persons, which lead to too large person regions. A good value found was to remove persons, whose regions are less than 10 pixels smaller than one of the actual image dimensions.

For *skin regions* we just match overlapping bounding boxes. Furthermore, we totally discard any non-matched skin region, instead of setting a non-moving state or something similar.

Forming Groups of Persons In MaMUT, such an approach does not seem to be too fruitful, as we want to distinguish which person interacts with an object, even if they stand close together. This requires us not to group close persons together and try our best to even delinate persons in such difficult situations.
Fortunately, our adaptive background subtraction algorithm gives quite robust results, even if persons are close. If one person stands still and the other one moves by, we almost have no problem, as the standing person is incorporated into the background model and does not disturb the tracking of the moving person.

4.8.2 Storing Tracking Results

The gathered informations have to be stored in a tracking database, which is in our case a class mainly containing a pointer to a vector of person objects `vector<Person*>* persons`. More details on this class can be found in Appendix C.4.

The tracking database provides a method for tracking the sequence of connected components at time t, extracted by the motion detection module via
`track(CvSeq* comp, int t, ..)`. In this method, every component is matched to all bounding boxes in the database. If it overlaps, i.e., it overlaps horizontally and vertically
`(vert_overlaping(r1, r2)) && (horiz_overlaping(r1, r2)`, or the color model matches to any person in the database, or the direction would match with the direction of any person
`same_direction(dir_person, compute_direction(..))`, the person object with index i is updated with the informations of the matching component via
`update_person(int index, CvRect r, ..)`. If no match is found we add a new person object for this component by calling `add_person(CvRect r, ..)`.
The matching of the color models, the color histograms, is done via the histogram intersection algorithm, available in OpenCV via
`double cvCompareHist(h1, h2, CV_COMP_INTERSECT)`. Empirically, histograms match if this function returns a value greater than 0.6.
In the mentioned update function, it is important to also update the color models. To do so, we accumulate stored and newly matched histograms and normalize afterwards via `cvNormalizeHist(h, 1.0)`
If we cannot update a person like described above, we lost the track and set the person non-moving via `remove_person(i,..)`.
It is interesting to note that we use a method `bool is_at_shelf(Person*)` to check if a person is standing in a region of interest. This method uses a quite sophisticated algorithm to test if a given point `x` is inside a general quadrangular, specified by its 4 corner points `x0,...,x3`: `in_quadrangle(x, x0, x1, x2, x3)`.
We use an algorithm originating from [OMH99], which we implemented in
`aux_functions.cpp`. Basically, it divides the quadrangular in two triangles and tests if the point lies in any of them. Hereby, it is important to note that the algorithm for

testing membership of a triangle traverses the triangle in a certain order. This forces us to give the 4 corner points in a certain order, namely starting in the lower left corner and continuing in counter-clockwise manner.

For convenience, the database also offers a method drawing the bounding boxes and trajectories of persons in an image, namely `draw_traces(IplImage* f)`.

A last thing is that our room cameras will *communicate* through the vector `persons`. All `TrackingDB` instances have a pointer to it and so they can get information about tracking results from other room cameras and store their own results in the same data structure.

4.9 Skin Detection

4.9.1 Skin Model

Before we start tracking with our shelf camera, we have to build a skin color model, which is represented by a hue-histogram of sample skin images.
So, we manually select small rectangular skin regions in sample images from our shelf camera with person's hands in it and save them as individual images.
At the initialization of MaMUT, we first create a one dimensional histogram
`CvHistogram* skin_model = cvCreateHist(1, ..)`, where the size is set to 30 bins (for the hue channel) and the range from 0 to 180, according to 0 and 180 degrees angles in the HSV color model.
Then, we compute the HSV version of every image via
`cvCvtColor( image, image_hsv, CV_BGR2HSV )`, compute the hue histogram of it and accumulate the histograms via `cvCalcHist( planes, skin_model, 1, 0 )` to our skin model.
After every step, we threshold the histogram to discard every bin which is smaller than $\frac{1}{40}$ of the total sample image pixels
(`cvThreshHist( skin_model, thresh )`), as we consider such hue values as due to noisy sample images.
Finally, after all sample images are processed we have to normalize the histogram such that the sum over all its bins is equal to 255, which is done via
(`cvNormalizeHist( skin_model, 255.0 )`). This is needed to create a greyscale skin probability image afterwards.

4.9.2 Skin Probability Image and Skin Classification

During the tracking, we compute a greyscale skin probability image.
Therefore, we compute the HSV version of the current frame, extract the hue channel in `f_hue` and set the image ROI according to a detected moving regions of the

`ShelfTracker` (which does micro-tracking) via `cvSetImageROI( f_hue, roi )`, where `roi` is just the `CvRect` object of a detected moving region.
Now, we can compute the histogram back projection with our skin model histogram (`cvCalcBackProject(plane, skin_prob_img, skin_model)`), yielding the greyscale skin probability image `skin_prob_img`. This image is then thresholded to classify skin pixels in a binary image (cf. binary foreground motion image).
A last step is to find connected components in the binary skin image, which is done in the same manner as for detecting moving regions. After connected components analysis, we also first discard too small regions and merge regions which are close enough together.

4.9.3 Thresholding, Filtering, Discarding and Merging

The threshold to binarize the skin probability image is computed according to the maximal value in $\mathcal{M}$:

$$T_{Skin} := \frac{3}{4} \cdot \max_{b \in \mathcal{M}} \mathcal{M}[b].$$

After that, we do a median filtering with a 3×3 mask.

A skin region area threshold $T_{Skin-Area}$ is used to discard too small skin regions. It is initially, like the thresholds for person regions, computed according to the image size $s = \mathbf{f}.width \cdot \mathbf{f}.height$ and a factor F via $T = s/F$:

$$T_{Skin-Area} := s/94.$$

Finally, to merge large skin regions which correspond to a hand or an arm, we use a distance threshold $T_{Skin-Dist}$, which is also computed according to the image size

$$T_{Skin-Dist} := s/1126.$$

To make the output nicer, we also use a height threshold $T_{Skin-Height}$ for skin regions. If we get too long skin regions, we set their height according to this value. It is computed as:

$$T_{Skin-Height} := s/1300.$$

Important: If the according config parameter is set, it is also possible to adapt the threshold parameters online via trackbars!

Concrete values: In our case with an image size of 352×240 pixels, we obtained the following thresholds:

- $T_{Skin} = 96.93$
- $T_{Skin-Area} = 898.72$

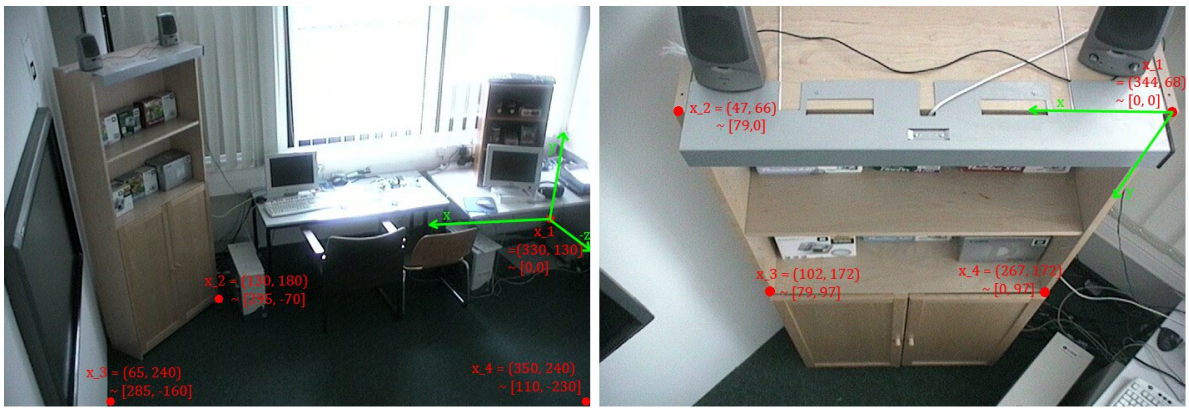

Figure 4.5: **Left:** Our instrumented room with the 4 used point correspondences $x_1, \ldots, x_4$ for camera $Room_1$ and coordinate axes. **Right:** The shelf in our room with the 4 used point correspondences $x_1, \ldots, x_4$ for the camera $Shelf$ and coordinate axes.

- $T_{Skin-Dist} = 75.03$
- $T_{Skin-Height} = 64$

4.10 Homography

The homography computation is done *before* the actual tracking task is started, and usually only once, as we used fixed cameras. We compute a homography to map image- to world (usually ground plane-) coordinates. Therefore, we use a separate program `./compute_homography`, which asks for 4 point correspondences $\underline{x_i} \leftrightarrow \underline{x_i'}$ between image- and world-coordinates to compute the projection matrix H.

The actual measurements, the resulting matrices and the implementation details can be found in Appendix C.6, but one can also refer to figure 4.5. The measuremts serve as input to our auxiliary program
`compute_homography <Measurements> <Matrix_H>` which computes matrix H according to the DLT algorithm sketched in section 3.3.2 and writes its result to a specified text file which will serve as input to our main program `./mamut <Matrix_H>`.

The resulting matrices H are used in the `Homography` objects of MaMUT, which offer a function
`CvPoint Homography::homography_projection (CvPoint x)` to transform image coordinates `x` to world coordinates `x_prime`.
Hereby, it is *essential* that we finally have to *normalize* the projection result $\underline{x_i'} := \frac{1}{w_i'} \underline{x_i'}$, i.e.,

```
x_prime.x = (int)(x_p_x / x_p_w);
x_prime.y = (int)(x_p_y / x_p_w);
```

Figure 4.6: The quadrangular region in front of our shelf which is used to check if a person stands in front of the shelf.

4.11 Fusion of Tracking Results

After we have tracked persons in our room from images of camera $Room_1$, which observes the shelf (macro-tracking) and skin regions at our shelf in images from the camera $Shelf$ (micro-tracking), we have to fuse these two results to finally be able to tell *which* persons hands we have detected at the shelf. Note that we do not have to implicitly fuse anything from camera $Room_2$, as it does not observe any areas of special interest.
In general, this approach can be extended to any number of room- / shelf-camera-pairs. The only thing that matters is that both cameras observe the same area of interest.

4.11.1 Approach

First of all, we check how many skin regions the shelf tracker has detected at the shelf. We denote this number by k and if $k = 0$, we are done.

If we have skin regions detected ($k > 0$), we now face the problem to *map skin regions to persons*. Therefore, the fuser asks the person tracker *how many persons* are standing in front of the shelf, which is obtained by just checking if the tracepoint of persons, which is for robustness reasons specified in image coordinates, is in an initially specified quadrangular region of our ground plane in front of the shelf (see figure 4.6). If we denote the number of persons at the shelf by n, we now do the following case distinction, after performing an initial *discarding* step. This step consists of two phases, where we discard each skin region which is outside of the shelf area in the first phase. Such skin regions have a center c_s in shelf coordinates, where $c_s.x < 0$ or $c_s.y < 0$. The

second phase is inspired by the idea that we only want to have one skin region per hand and per person. So we discard as many skin regions s.t. we obtain $k \leq 2 \cdot n$, i.e., we allow only two skin regions per person at the shelf. To decide which skin regions to discard, we just discard the smallest skin regions.

After discarding we can come to the mentioned case distinction:

- **If** $n = 0$, we cannot do anything as there is no person we can map our skin regions to.

- **If** $n = 1$, we can easily map *all* k skin regions to the lonely person at the shelf.

- **If** $n = 2$ **and** $k = 1$, we map the lonely skin region to the person *closest* to the shelf. To find the closest person, we search for those person whose tracepoint has the least distance to the center of the quadrangular shelf region (see figure 4.6).

- **If** $n = 2$ **and** $k > 1$, we do a *bipartition* of the skin regions. This means we search for two sets (skin clusters) of skin regions, which contain almost the same number of regions and where the distance of skin regions in a set is minimal.
  For this problem, there exist some sophisticated algorithms, known as *graph partitioning* algorithms.

  We use a simple algorithm inspired from the blob clustering of [KHM+00], which first computes the distances of each skin region to all others, and stores them in a (symmetric) $k \times k$ matrix D.
  We now search for the longest distance in D. If we find the maximum of D in $D_{i,j}$, we take the skin regions i and j as seeds for the two skin clusters.
  These clusters are then grown by just adding the skin region with the least distance to any skin cluster S. This is repeated until no skin regions are cluster-less (see figure 4.7).
  If we now have the bipartition of skin regions, we order the clusters from left to right, according to our shelf coordinate system. Therefore, we take into account the x-coordinate of the leftmost skin region of every skin cluster. If we now also order our $n = 2$ persons from left to right according to our room coordinate system, by comparing the x-coordinate of their tracepoints, we can do a simple $1 - to - 1$-mapping of the two skin clusters to the two persons.

- **If** $n \geq 3$, we do no fusing, as we consider the shelf as too crowded in this situation. If we had a larger shelf, it would also be possible to deal with three or even more persons by just refining the clustering algorithm to do a n-partition instead of a bipartition. The problem however is that persons which stand too close together will be classified as one moving region, which offers a big problem. A further discussion can be found in section 5.2.3.

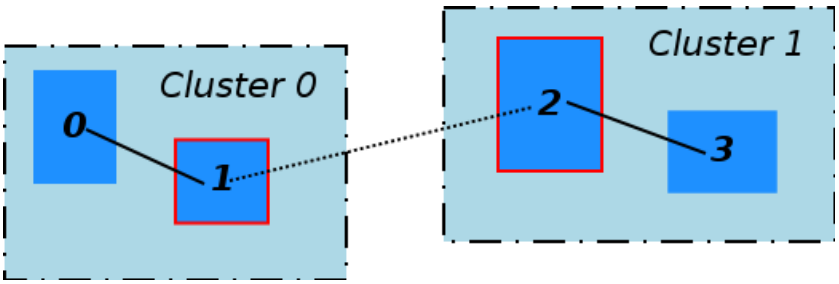

Figure 4.7: Example bipartition of 4 skin regions. The dotted line indicates that the longest distance of regions exists between region 1 and 2 (red). The solid lines indicate which regions were added to cluster 0 (because of close distance to region 1) or cluster 1 (because of close distance to region 2).

To implement above ideas, we use a tracking fuser class, which will process the results of person tracking and our shelf tracker, by mapping skin regions to persons, from which they most probably originate.
This is done by the `TrackingFuser` class, which has pointers to our `TrackingDB` and `ShelfTracker` objects and fuses the results via a `fuse(..)` method. This method will assign detected skin regions of the shelf tracker to persons in the `persons` vector of the `TrackingDB`. To do so, it also offers a method to do a bipartition of skin regions into two skin clusters: `vector<vector<SkinRegion*> > bipartition_skin()`.
For convenience, the fuser offers methods for printing its status to the console (`print(..)`) and for drawing the ID's of persons to the assigned skin regions (`draw(..)`).

4.12 Output: EventHeap

The EventHeap is a coordination model for coordinating interactions of applications running on devices which are part of an ubiquitous computing environment, like our instrumented room. It is similar to tuplespaces [CG92].
In this model, all participants communicate through a commonly accessible tuplespace, a storage where tuples can be written to and read from (either via a destructive or a non-destructive read).
These tuples are ordered type-value fields
`[(type_1,value_1), (type_2,value_2), ...]`, containing informations. In our case this will most probably be informations about the state of our room and the devices in it. If we want to read a tuple, we have to specify a template tuple, where fields to be matched contain concrete values and fields to be retrieved contain wildcards.

4.12.1 Tracker Events

As output for our tracker module we will write *Tracker Events* to the EventHeap, collecting the tracking results.
A typical tuple could look like this:

```
[
  (EventType, String, 'TrackerEvent'),
  (TimeToLive, Int, 1000),


  (Number_Of_Persons, Int, 2),

  (Position_Person_0, String, '< 257 , -169 > (*)'),
  (Number_Skin_Regions_Person_0, Int, 1),
  (Person_0_Skin_Region_0, String, '(47 , 120 )'),

  (Position_Person_1, String, '< 148 , -209 >'),
  (Number_Skin_Regions_Person_1, Int, 0),


  (TimeStamp, Int, 16)
]
```

This tuple would express that the *number of persons* in our room is 2, i.e., we have $person_0$ and $person_1$.
Furthermore, their *position* (the coordinates of their tracepoint on the ground plane ($x, -z$-plane) is given. If we have `(*)` appended to the position, this will mean that this person stands in front of the shelf. These are the informations gathered by the motion detection module, combined with the homography position estimation.

We also see how many skin regions are detected by the shelf tracker and if any, we get their positions in shelf coordinates. These informations originate from the skin detection and the fusion module.

The *TimeToLive* of our tuple is computed according to the computed *FPS-rate* of MaMUT, updated after every frame. This value tells us how many frames MaMUT can capture, process and output per second, and so we compute the TimeToLive in milliseconds as $TTL = 3 \cdot \frac{1000}{FPS-rate}[ms]$.
This yields that we (approximately) have three tuples per time on the EventHeap. This decision is based on the fact that we may have some useless results if MaMUT adapts to illumination changes or something similar. Usually three frames will then be enough to finish adaption and produce reasonable output again.

Finally, the *timestamp* of the frame of our video stream which was used for obtaining this results is given.

Chapter 5

Conclusions and Future Work

In this final chapter we want to summarize what we have done and learned during developing MaMUT. We state problems and possible solutions and ideas on how to enhance and extend MaMUT that would have exceeded the scope of this thesis by far.

5.1 Applications

Of course, one has to ask himself what will be the use of the data collected by our system. Before dealing with this problem in detail, we concern ourselves with the problem of *person identities*.
Basically, extracting number and position of persons and their hands is a big step towards multi-user distinction in ubicomp environments, but a final goal could be to assign personal informations, like the name, to every detected person. Remember that at the moment we just have an enumeration of anonymous users: $person_0, person_1, \ldots$.

As mentioned, MaMUT will basically extract for every frame of the image sequence how many persons are in the room and where they (macro-tracking) and their hands (micro-tracking) are located. These informations will then be published to a common communication infrastructure of our room, the *EventHeap* (see section 4.12).
Applications, which could use the informations originating from our system are for example:

Virtual Room Inhabitant: We have an application where a movable beamer projects a virtual room inhabitant (VRI), to the walls of our room [KSS05]. The VRI is supposed to assist and guide persons in the room. Therefore it has to follow the persons, which is only possible if we know their positions. With the informations of the micro-tracking, one could also let the VRI react to interaction with objects, like describing a product picked out of a shelf.

Consulting Customers at the Shelf: In our room is a shelf with digital cameras in it. Here, an audio system can tell customers details about the cameras, like price,

mega-pixels, focal length, and so on.
With the data of MaMUT, one could recognize if a person comes back to the shelf and continue the consultation, instead of starting it over and over again if we detect the removal of a product from the shelf, or we can even react to customers who do not pick any products, but just point towards them.

Integration into Indoor Positioning System: At the moment, we have an indoor positioning system [BS05], which can locate persons in our instrumented room via RFID-tags and according antennas or Infrared-senders and PDAs with IrDA (**I**nfra**r**ed **D**ata **A**ssociation) support.
With the help of MaMUT, one could also locate persons *without* any auxiliary devices.

Storing Persons Color-Profiles in UbisWorld: In our project UbisWorld [Hec], we can, among other things, model persons that are part of our environment, which encompasses informations about their name, gender and so on.
Our ideas is now that we could also add informations about the color model of persons which we create anyway during the tracking task. Therefore, we had to require our users to initially identify themselves when they enter the instrumented room, for example via a badge or something similar. But the problem is that most probably persons will change their clothes every other day, and so we could only store a *daily color model* for each person, which is only valid for one day or even a shorter period of time.

5.2 Conclusions

To conclude this thesis, we want to shortly recapitulate, what were our goals, which ones we have achieved and where there are still some unsolved problems.

5.2.1 What have we done

In section 1.3, we stated requirements to our system. In the following, we will discuss, which we have met and in what extent.

Result: Basically, we have developed a *person tracker* for our instrumented room and added a *shelf tracker*, which tracks hand- and arm-regions in front of our product shelf. Here, we also do a skin detection for finding hands of persons reaching towards products in our shelf.
All this was done with a '*come as you are*' philosophy in mind, wanting to be able to track person no matter how their outward appearance (clothes, skin color) is.
The results of these two, basically independent, modules are *fused* to finally obtain a result where we can tell which person has reached towards which object in the shelf. This is exactly the result we wanted.

In this context we should note that our approach can also be used for *displays* or other special areas of interest without any adaptations.

Integration into existing infrastructure: With the integration of an LAN-camera and an EventHeap interface to MaMUT, we met this requirement perfectly, but it costed quite a lot of time, because this requirement forced us to access the LAN cameras via their CGI-interface in our C++ programs, which we solved by using the command line tool `cURL` via UNIX system calls. Furthermore, we also had to find a solution for outputting tracking results to the EventHeap. This, we solved by using the HTTP interface of the EventHeap, for which we again used `cURL`.

Cameras: The three cameras we used and which are described in section 4.2 fulfill all the needs specified, like LAN access, frame rate or image resolution.

Realtime: Here we faced some problems, as our frame rate is limited to 4–5 FPS, which we will discuss in detail in section 5.2.3.

Robustness: Also this point posed, as expected, quite some challenges. Some we could solve, e.g., dealing to a certain degree with noisy images and resulting noisy motion images. We also can handle about three to four users in our relatively small lab, where two can stand in front of the shelf at a time. We judge these limits as quite reasonable.
Of course, we could not solve all problems in the context of this thesis. The interested reader may refer to section 5.2.3 for a detailled discussion of problems and possible solutions.

In conclusion, one can see that for developing our system, we partly adopted other approaches, like the statistical background subtraction, and newly invented ideas like the fusion of tracking results.

5.2.2 What have we learned

Now, it is time to summarize what we have learned while developing our tracking system.
The most important lesson we have learned is that Computer Vision is quite promising in the context of interacting with users in ubicomp environments, but also a very challenging field of Computer Science.
This can be explained by the fact that Computer Vision offers on the one hand the maybe ultimate ubiquitous interface to computer systems, as it allows to interact with users without any auxiliary means, by just detecting human movement, gestures and so on. On the other hand, nothing comes for free and Computer Vision systems force us to cope with lots of sensitive threshold parameters, robustness problems and computational highly complex algorithms, spoiling the needs of realtime applications. This last point bring us then to the problems we faced.

5.2.3 Problems

As mentioned, during developing and testing MaMUT, we encountered some problems we could not solve, or at least not totally satisfactory, in the scope of this thesis.
In some cases, the easiest remedy could come from just changing the settings in our room, which can be done quite easily here, but would pose big problems in an outdoor setting for example. Unfavorable lightning conditions, for example, can be handled by installing different lightning sources.

Realtime Needs

One big problem are the realtime needs of our system. We have three cameras from which we capture images of size 352×240 pixels, and where we want to have frame rates of about 10 FPS. This frame rate is needed by two factors. First, a realtime system should process *at least* 10 frames per second and beyond this, much of our approaches rely on the fact that our frame rate is quite high which yields comparably small changes from frame to frame.
This means we have to process $3 \cdot 10 \cdot 352 \cdot 240 = 2534400$ pixels per second. We have to keep in mind that we have to apply *several algorithms* (background subtraction, connected component labeling, region merging, skin detection, ...) for each image and some of them, like the background subtraction, have to process *every* pixel.
This is quite a challenge, even for state of the art systems and in spite of the fact that fast CPUs and memory becomes less and less expensive nowadays.

In fact, we did not manage to reach frame rates above 1–2 FPS under real-world conditions at a PC operating under *Debian/GNU Linux*, equipped with an *Intel Centrino* 1.4 *GHz* processor and 256 *MB RAM*. We guess that the main bottleneck is the small memory of our machine.
Measurements resulted that our machine needs about $750ms$ for just processing the images without capturing or output to the EventHeap, which yields that we approximately can only obtain 1.3 FPS.

Fusion

The fusing approach described in section 4.11.1 is quite simplistic and so naturally suffers from some problems.
Fist, we are in serious trouble if persons *cross their hands*. Then our algorithm will just do the wrong mapping. Also, if more than two persons stand in front of the shelf, the simple algorithm will most likely produce bad results.

Crowded Rooms

If our room is too crowded with persons, the delineation of all the persons is not possible. In our small room, we already face serious problems distinguishing persons if we have

more than three at the same time in the view of our camera.
This is caused by the fact that the persons will then have to stand such close together, that the region merging algorithm will merge all of their blobs to one or two big regions which it will discard because of too large area or strange aspect ratio.

Shadows

Artificial light sources which are punctiform, like spotlights, cause persons walking in our room to cast shadows. The problem is that these moving shadows are detected as moving blobs by the background subtraction algorithm and merged to person regions, which yields inexact, too large, bounding boxes (see figure 5.2.3).
One solution is to avoid punctiform lightning and rely on *diffuse light sources*, like natural sunlight, or spotlights pointing towards the ceiling. In the first case, putting some sunblinds in front of the windows will improve the situation further.

If we, nevertheless, want or have to use punctiform light sources, a background subtraction with *shadow removal*, like described in section 5.3.1 could solve this problem.

Illumination Changes

Altough we refrain from using punctiform lightning sources and use a quite sophisticated, adaptive, statistical background model, we have some problems if persons are for example walking along a window, which changes the illumination situation of the whole room (see figure 5.2). This change is not very serious, but it yields that we will have no useful background subtraction result for this and the next two to three frames (until the background model has been adapted to the new situation), resulting in losing the track of our tracked persons.
But this does not pose a big problem, as we will recover the track of persons after these few frames via our color models and just 'track' persons as non-moving during this few frames.

Static Persons

Another problem concerning background subtraction is the fact that persons will most probably stand still for quite some frames if they stand in front of the shelf, having a look at the products in the shelf (see figure 5.3).
The problem is that our adaptive background subtraction algorithm will incorporate static persons (or parts of them) into its background model, which yields that we do not find them anymore in the result of the background subtraction.
This problem is again partly solved by setting persons to non-moving and recovering their track as they start to move again.

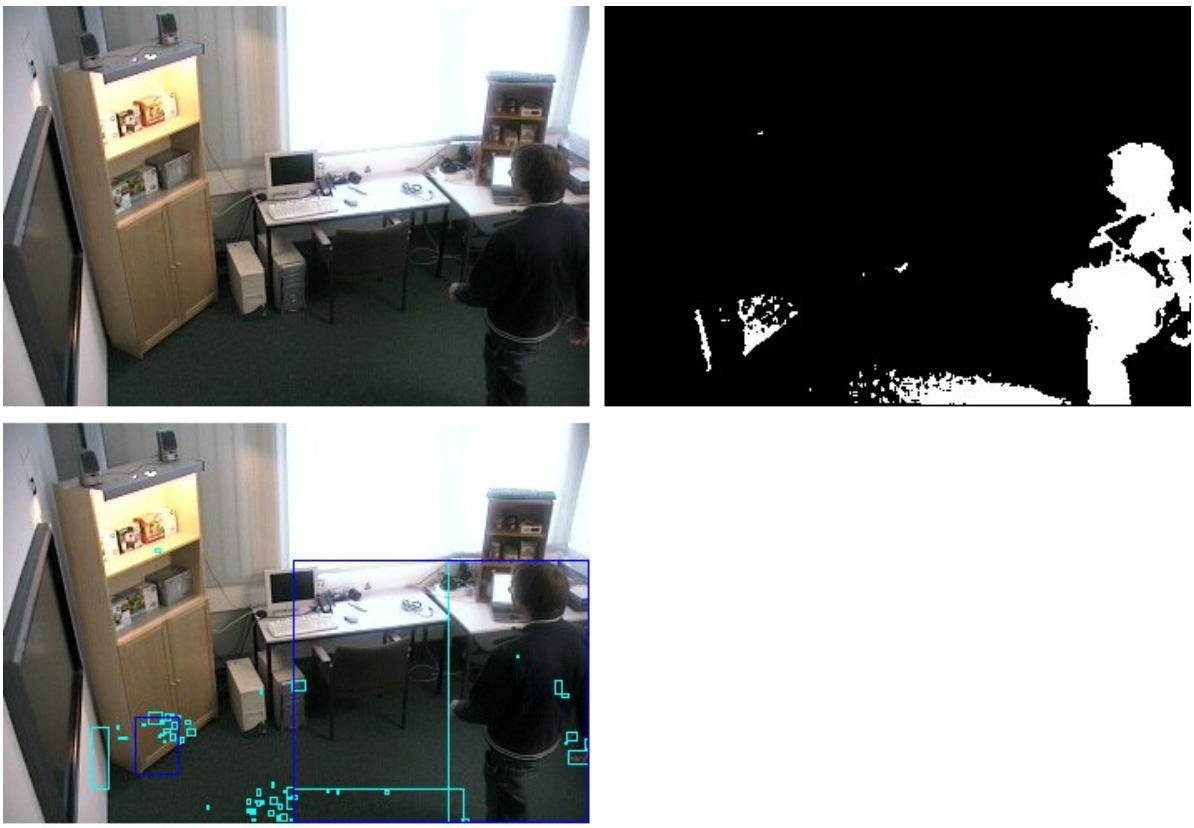

Figure 5.1: **Top Left:** Our room enlightened with a punctiform light source. **Top Right:** Result of background subtraction. **Bottom Left:** Result of person tracking has too large person regions, which also encompass the shadow cast by the person.

Figure 5.2: **Top Left:** Frame from one of the macro-tracking cameras. Person is walking across window. **Top Right:** Background model. **Bottom Left:** Result of background subtraction. **Bottom Right:** (Useless) result of region detection.

Figure 5.3: **Top Left:** Frame from one of the macro-tracking cameras. Persons are standing more or less static in front of the shelf. **Top Right:** Background model. Note the incorporation of the two persons into the model. **Bottom Left:** Result of background subtraction. **Bottom Right:** Result of region detection does *not* detect the two persons properly.

5.3 Possible Solutions to Mentioned Problems

As one could see from above, there are several points, where MaMUT could be improved. In this section we will collect ideas from other approaches, or ideas we could not realize in the scope of this thesis.

5.3.1 Other approaches to Motion Detection

This section covers possible improvements and alternatives to our motion detection module, i.e., background subtraction, blob- and region extraction and region merging.

Other Statistical Background Models

In [LHGT], another background modelling approach using Bayes decision rules for classifying moving objects from selected feature vectors is presented. It is also available in OpenCV's `cvaux` library, and so an evaluation with the Gaussian Mixture models [KB01] used in MaMUT may be interesting.

Shadow Removal in Background Subtraction Algorithms

A common problem is that most of the background subtraction algorithms cannot distinguish between moving shadows and persons casting them.
In [KB01] and [MJD+00], the authors use *chromatic color spaces*, like the RGB color space, where we can separate chromatic and brightness components. The idea is that a shadow pixel is darker than its background model, but only changes slightly in chromaticity. Such pixels should then *not* be classified as moving.
In [HHD98], the problem is easily solved by incorporating informations from the disparity images of the stereo-vision system. In disparity images, shadows cast on the ground will not differ from their background model.

As we do not use any stereo vision, maybe the best possibility would be to incorporate the approach of [KB01] in OpenCV. This will make sense, as the basic algorithm *without* shadow removal is already implemented in OpenCV's `cvaux` library function `cvCreateGaussianBGModel`.

Sophisticated Person Models

In MaMUT, we used the most simple person model, which is usually called *model-free*. This means we just use region- and blob-based-tracking combined with size and aspect-ratio thresholds and color information to recognize persons.

The RPT [Sie03] uses a really sophisticated model to distinguish moving regions which correspond to persons from non-person objects, like displays with changing content,

opened doors and windows, shadows of persons, and so on. Here, the idea is to teach the tracker a certain idea of the outward appearance of a person.

Active Shape Models:
The problem with these models is that we have a tradeoff between complexity and exactness. The RPT uses a more sophisticated model than MaMUT, actually a *2-D appearance model*, called *Active Shape model* ([Bau95],[BH95],[Bau96]) of persons, i.e. they teach the system the mean outline, the so called *mean shape* of a person in a $2D$-image.
To do so, they need a large set of test data, a set of pictures depicting a walking person, possibly recorded in a special environment where one can easily determine their shape via a simple edge detector (see Appendix A) or manually determine the person outlines. This training set is used for obtaining the Active Shape Model m, i.e. one wants to compute one adaptive 'mean' shape (person outline) $m(p)$ of a person in a $2D$ image. This shape is parameterized with a parameter p, so that it can be adjusted to the actual image parameters, like size, and obtain a representation $d = m(p)$.

Each of the n extracted shapes is represented by a vector of spline coefficients, so called control points (or landmark points as they are called in [Bau95],[BH95],[Bau96]) for a cubic B-spline and for obtaining a set of coefficient vectors. From this, one can compute the coefficients of the mean shape m, by just computing the mean coefficient vector.
The problem is that if one would try to match every region in every frame with the mean shape, one would get a high space and time complexity and destroying needs for realtime computations. Therefore one has to reduce the complexity of the model, i.e., reducing the dimensionality of the model by a certain approximation.
This approximation method is known in the literature under the notion of *Principal Component Analysis (PCA)* ([Ger99], sect.11.14), which we will not discuss any further in the context of this thesis.

Even more advanced person models

There exist even more advanced models relying on 3D information ([GD96], [SBF00]) which are even more robust and exact.
Here, 3D space models of persons are used, which are projected to the concrete setting. But either we need expensive, dedicated hardware (stereo cameras) or the computational complexity of these models does not allow realtime operation anymore and therefore we cannot fruitfully use them in our setting.

Advanced Region Merging, Blob Extraction

We use a very simple blob extraction algorithm, namely connected components labeling, i.e., we initially assume every connected component to be a region and then merge regions according to some simple criteria, like overlapping, distance, and so on.

More sophisticated algorithms, like in [KHM+00], use weighted graphs (blobs are nodes, arcs are weighted with Euclidean distance between two blob centroids) and minimum spanning tree algorithms on them to extract sets of blobs (clusters) which are most likely to belong to one person.

In OpenCV, there exist no functions for such algorithms, and hence a usage in MaMUT would require to implement the algorithms from scratch, which will be quite a lot of work.

Fortunately, Dave Grossman, a member of the OpenCV community has developed a blob extraction library available for download at [Sou].
The problem is that it is only available for Microsoft Windows at the moment, but as the code is open, one could try to port it to Linux and use it with MaMUT.

5.3.2 Improvements to Region Tracking

This section will show some improvements to our tracking module which tracks detected regions through subsequent frames.

Occlusion Handling with Stereo Systems

In the W4S tracking system [HHD98], a *stereo system* (SVM) is used to gather $2\frac{1}{2}$D information of the scene and robustify the tracking of person regions through sudden illumination changes, occlusions and merging and splitting of person groups. These are problems which also exist in MaMUT, but the question is if the use of a quite special system like SVM would make sense in our setting, especially, as W4S relies on monochromatic images, which would make a skin detection virtually impossible.
W4S detects *object interaction* via its sophisticated models of the human body. This approach could also be fruitful in MaMUT.

Color Histograms

In [MJD+00], the updating of the color histograms is done in an adaptive manner, using $P_t(x|i)$, the old, stored and $P_{t+1}(x|i)$, the newly acquired informations:

$$P_{t+1}(x|i) = \beta P_t(x|i) + (1 - \beta)P_{t+1}(x|i),$$

with a weight of $\beta = 0.8$.
This updating is not possible in OpenCV, but one could test if it leads to better results, than the cumulative update we use.
To distinguish persons as they form groups, which will lead to merging of their regions (bounding boxes), the authors of [MJD+00] use a *probabilistic occlusion reasoning* approach, based on color histograms, which could make sense to test in MaMUT. If persons come together in groups and occlude each other, this approach computes probabilities

of pixels belonging to a certain person, based on the color histogram of the persons. This approach sounds quite elegant, but will be hard to implement using the OpenCV library.

The *object interaction* detection in [MJD+00] relies on considering small new regions as dropped objects. Unfortunately, MaMUT will detect lots of small noise regions due to illumination changes, which will make this approach almost impossible in MaMUT.

Kalman Filtering

One thing to robustify tracking of person regions can be to use the widely accepted *Kalman Filtering* technique. In [Wik], we find the definition of a Kalman Filter as just 'an efficient recursive filter which estimates the state of a dynamic system from a series of incomplete and noisy measurements'.
In the context of person tracking, we can use a Kalman Filter to provide accurate and continuously-updated information about the position and velocity of our persons, given a sequence of (former) observations about their position, each of which includes some error.
In a person tracking application, we are at any time instant interested in tracking a person. To do so, information about the location, speed, and acceleration of the person is measured with a certain amount of corruption by noisy images and resulting corrupted person blobs and regions. The Kalman filter exploits the dynamics of the person movements, which govern its time evolution, to remove the effects of the noise and get a good estimate of the location of the person at the present time (filtering), at a future time (prediction), or at a time in the past (interpolation or smoothing).
Note: The above examples are inspired from the examples available in [Wik].

5.3.3 Possible Improvements to Skin Detection

Also our skin detection module could be improved further to deliver more robust classification of skin pixels.

Improvements to Histogram Back Projection

One thing to note is that for dark pixels, i.e., pixels with a low saturation, but also pixels with a quite high brightness it is very difficult to measure their hue value, because of the hexcone structure of the HSV model (see Glossary, Appendix A). A remedy could come from applying brightness and saturation thresholds to pixels before computing the histogram back projection, i.e., pixels with brightness or saturation values above these thresholds are not considered for the back projection.
In [Bra98], a value of 10% of the maximum pixel value is proposed.

Alternative to Histogram Back Projection: Bayesian Classifier

A second class of skin classification algorithm is based on Bayesian classification, like described in [AL04]. These approaches mainly obtain their skin probability image by computing in every pixel the probability of this pixel having skin color s, given its color value c, i.e., they compute the probability

$$P(s|c) = \frac{P(c|s) \cdot P(s)}{P(c)}$$

. The probabilities on the right side of the equation are obtained a priori on a small training set of images, where one manually classifies skin pixels. During operation the algorithm adapts itself to the given input of the last w frames $P_w(s|c)$, i.e., $P(s|c) = \gamma P(s|c) + (1-\gamma)P_w(S|c)$, using a parameters γ determining how strong the influence of new measurements is.
These approaches promise a higher robustness, especially due to their online adaption phase.
The problem is especially the offline training phase, where one has to manually mark skin pixels in test images. This could be quite difficult and time consuming to implement with OpenCV's small GUI library `highgui`.

5.3.4 Other approaches to Homography Estimation

In [AJN05], more than 10 different homography estimation techniques are described and evaluated, including the *DLT* algorithm, which we use in MaMUT.
Interesting in this context are algorithms which use more than 4 point correspondences to robustify the result. By the way, this is also the case in the Gold standard algorithm, an extension of the DLT-algorithm, which can also be found in [HZ04]. For further homography estimation techniques the reader may refer to [KS06] for an approach using more than one camera.

5.3.5 Approaches using ≥ 2 Cameras (Stereo Reconstruction)

A quite different approach would be to use sophisticated *stereo reconstruction* techniques, to get real $3D$-information from our environment, our instrumented room.
These approaches try to use stereo vision, inspired by the human vision system with two eyes. Hence, we have to use images from (at least) two calibrated (see Glossary, Appendix A) cameras. The main task is then to find corresponding points in the two images, which is computational quite complex and could spoil our realtime needs. By computing the disparity of all these point pairs, we can then compute depth informations for all pixels, which is just the missing third dimension.
A very good introduction to this complex field cam be found in the corresponding chapters in [HZ04] or [JKS95].

Papers using this approach in a comparable setting to ours are for example [RSL99], where the authors track the movement of cars on a parking lot using multiple video streams. Here, also a short description of the calibration task can be found.
In [FR02], the authors track persons in a smart room with multiple calibrated cameras, which compares quite well with to our task.

5.3.6 Stereo Cameras

OpenCv's auxiliary functions library `cvaux` comes with routines to calibrate and use stereo cameras to perform $3D$ tracking of objects. Those cameras have two lenses and output depth-images ($3D$-images) in realtime, which can then be processed by our tracker.
That this approach can be fruitful is shown in [KHM+00]. Here, two color stereo-vision Triclops cameras from Point Grey Research [Poi] are used. These cameras give $3D$ informations without any computations from the tracker module, which will be a huge benefit for determining the position of persons or skin objects, but they are unfortunately still quite expensive at the moment.

5.4 Extensions to our System

5.4.1 More Cameras

Theoretically, the number of cameras (for macro- and micro-tracking) is not restricted. For our setting, it was enough to have two room cameras for macro-tracking and one shelf camera for micro-tracking at the lonely shelf in our room.
Adding more (disjoint) room cameras is no problem, we just have to create individual database structures for each. For every shelf camera, we will need an individual fuser, which fuses informations from the room camera observing the shelf and the actual shelf camera.

One thing to keep in mind is that every additional camera increases the number of pixels to be processed in every frame, and so adding more and more cameras to MaMUT could spoil the realtime needs by increasing the network traffic, CPU load and memory usage.
A possible solution may be to use more than one computer to run our system, but distributing MaMUT to different computer systems, e.g., one computer for every room- / shelf-camera pair and maybe an additional computer for room cameras, which do not observe any shelf, and hence need no fusing.

5.4.2 Gesture Recognition

One thing that came to our mind when developing MaMUT, is to do gesture recognition with the micro-tracking cameras, as we anyway do skin detection to detect hands.
This could be used to distinguish *different interactions* with objects, like asking about the price or something similar, by different hand poses.

One nice thing is that the so called *POSIT*-algorithm [DD92], which can be used for gesture recognition is implemented in OpenCV.

5.4.3 Moving Cameras

Basically, our system is equipped with Pan-Tilt-Zoom (PTZ) cameras, i.e., cameras which are moveable. The advantage of such cameras could be that they allow us to follow certain users, or to have one camera observing different spots, depending on whether there are users in this area.
The big problem is that a homography estimation as we use it is only possible if one has static cameras, i.e., if we want to pursue the approach of moving cameras one has to think about another possibility to extract the position of users in the environment or to just allow *discrete steps* of our cameras. In the latter case, we then could store our homography matrices and positions of special areas of interest for micro-tracking for all the different camera positions.

Appendix A

Glossary - Basics in Image Processing and Computer Vision

We will use some notions and definitions from the field of Image Processing and Computer Vision. In the following we will give a short introduction to some basic notions like noise, filters, etc..

Noise in Digital Image is a very common kind of image degradation due to digitization and compression methods. As we will use JPEG images we mainly have the problem of noise which is created by the sub-sampling and quantization of color information by the JPEG compression. This compression method discards color information because brightness is more important for the human eye.
But also small defects in the CCD chip of our digital cameras can create so called impulse (salt and pepper) noise, where we obtain erroneous pixel values at certain pixels.

Image Filters may offer a remedy for dealing with noise.
Formally an image filter F is just function mapping an initial image $\mathbf{u}$ to another, filtered image $\mathbf{f}$, i.e. $\mathbf{f} = F(\mathbf{u})$.
We will deal with *local filters* which only take into account a small neighborhood of a pixel (e.g. a 3×3-mask or even just a single pixel) in $\mathbf{u}$ to determine the corresponding new pixel value in $\mathbf{f}$.

Median filtering is a local filtering which can have two different meanings, but the idea is always the same.
First, it can mean that we create an average image over n given images. Here, we average over an uneven number of images, so $n = 2m + 1$. Then for every pixel in the filtered image, we use the median over the n corresponding pixels in the n given images.
To compute the median, we order the n pixels with increasing value and take the middle pixel value as median.

Formally, if we have the n images $\mathbf{u_1}, \ldots, \mathbf{u_n}$ and want to obtain a filtered image $\mathbf{f}$:

$$pixels = \texttt{order(increasing, } \mathbf{u_1}(x,y), \ldots, \mathbf{u_n}(x,y)\texttt{);}$$

$$\mathbf{f}(x,y) = pixels[m].$$

The second meaning of a median filter can be that we average each pixel in a *single* image, by replacing its color value by the median of all color values in a $n \times n$ neighborhood, where n is uneven, $n = 2m + 1$, again.
The benefits of median filtering are that it is more robust against outliers (salt and pepper noise), as an arithmetic mean filter or something similar.

Closing is a morphological filter which performs a *erosion* followed by a *dilation*. These filters use a structuring element B which may be a $n \times n$, $n > 0$ mask or even a 'round' pixel mask.
Erosion replaces every pixel (x, y) in an image by the minimum pixel value in $B(x, y)$ and dilation replaces pixels by the maximum.
If we know close our binary motion image, we first apply an erosion, followed by a dilation, which will close any vacancies in blobs which are smaller or equal to B.

Edge Detection algorithms can be used to determine edges in greyscale or even color images. They search for strong changes in the pixel values in a small neighborhood (i.e., a large image gradient, see below), which indicate the presence of an edge.
Their problem is that they suffer a lot from noise.

Image Gradients are similar to gradients in analysis. As mentioned in section 3.1, we can describe images as functions $f : \Omega \longrightarrow \mathbb{R}$ and hence compute gradients.
Hereby, we have to note that usually $\Omega = [0, \mathbf{f}.width - 1] \times [0, \mathbf{f}.height - 1]$, i.e., we have for every pixel (x, y) gradients in $x-$ and $y-$ direction.

Color models are used to code color in images. One usually stores for every pixel of the image a triple, coding the color.

RGB color model Usually we use RGB color images. These images store for every pixel a triple $(r, g, b) \in [0, \ldots, 255]^3$ of red, green and blue color values (corresponding to 0 and 1 in a byte-wise coding) to describe all possible colors. This is a natural way of color coding, as computer monitors and even the human eye uses this 'technique'.
As a visualization, one uses a cube to describe RGB color (see figure A.1).
But for some reasons other color model may be more appropriate .

HSV color model HSV (Hue-Saturation-Value) stores a triple $(h, s, v) \in [0, \ldots, 360] \times [0, \ldots, 255]^2$.

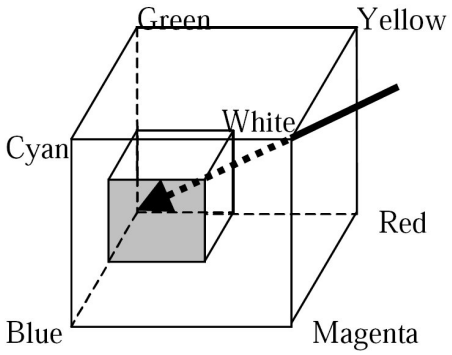

Figure A.1: RGB cube. **Author:** G. R. Bradski [Bra98]

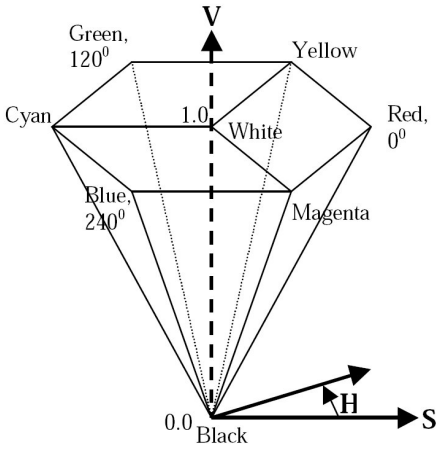

Figure A.2: HSV hexcone. **Author:** G. R. Bradski [Bra98]

The hue channel describes the color value as an angle between 0 corresponding to red and 360, which is red again. The saturation channel describes how concentrated the color is as a scalar value between 0 and 255 (corresponding to 0 and 1 in a byte-wise coding). Finally, the value channel describes the brightness of the color, also in the range from 0 to 255.
As a visualization, one uses a hexcone (see figure A.2).
This color model is extremely useful to analyze color information (hue), invariant from lightning changes, which we will use for skin color classification.

Histogram A histogram of an image can be thought of as a lookup table, where we store which color-value (or grey-value) occurs how many times in the image. So a color image yields three histograms, one for each color channel.
Usually, the domain of the histogram are discrete values in the range from 0 to 255 (its bins, as one says), representing all possible color values. But for efficiency reasons it may be useful to group certain bins together, e.g. just use bins from 0

to 64, where we combine every 4 bins. Naturally, the sum over all bin values of a histogram is supposed to be equal to the number of pixels in the image.

Homogeneous Point Coordinates If we want to describe points on a plane, e.g. on an image or points on the ground plane of our room, we basically only need 2 values $\vec{x} = (x, y)^T$. But as these points are projected from a real $3D$-scene onto the plane, we would like to express them in $\mathbb{R}^3$. Therefore, we exploit the fact that each point $\vec{x}$ on the plane will lie on the line described by $ax+by+x = 0$, hence we get an arbitrary scale factor w, leading to $\vec{x} = (x, y, w)^T$, which we usually set to $w = 1 \Rightarrow \vec{x} = (x, y, 1)^T$. This is motivated by the fact that we map homogeneous points to $2D$ points via $(x, y, w)^T \mapsto (x/w, y/w)^T$. In our case: $(x, y, 1)^t \mapsto (x, y)^T$.

Camera Calibration is the task to estimate the so called 6 *extrinsic* and 5 *intrinsic parameters* of cameras.
The *extrinsic parameters* denote the position of the world (room) coordinate system relative to the camera coordinate system.
We describe this relation by a translation $(t_1, t_2, t_3)^T$ followed by a rotation around the $x-$, $y-$ or $z-$axis by an angle ϕ_x, ϕ_y, ϕ_z.
The *intrinsic parameters* characterize the geometry of the image plane inside the camera, i.e. they are used to map an image from the $3D$-camera coordinate system to the $2D$-image coordinate system. This is necessary as

1. The origin of the image plane my be located in another point than the origin of the camera coordinate system. Usually, the image origin is in the upper-left corner, whereas the camera origin would be projected to the middle of the image plane. We say that the camera origin is located in $(x_0, y_0)^T$ of the image plane.
2. Pixels may have different dimensions in the two coordinate systems, we denote the pixel dimensions of the image by k_x and k_y.
3. The coordinate axis may not be parallel, but have an angle $\theta \neq \pi/2$.

These parameters are stored in matrices, which can be used to map points from one coordinate system to another.
The idea of mainly all existing camera calibration algorithms is it to investigate on an object of known size and shape and how correspondent points change if we move the object.

Appendix B

Algorithm Details

In this Appendix, we want to present details of algorithms presented in chapter 3.

B.1 Update Equation for Gaussian Mixture Model

As mentioned in section 3.2.1, we update at each frame our Gaussian distributions by some sophistacted update equations, we will now specify.

Let w_k^{N+1} be the new weight for distribution k, $(1 \leq k \leq K)$ at time level $N+1$,
μ_k^{N+1} be the new mean of the distribution k at time level $N+1$ and
$\sqcup_k^{N+1}$ be the new covariance matrix of the distribution k at time level $N+1$.

- For expected sufficient statistics update [Now91] (used at time $N < L$) we have:

$$\begin{aligned} w_k^{N+1} &= w_k^N + \frac{1}{N+1}(p(\omega_k \mid \vec{x}_{N+1}) - w_k^N)) \\ \mu_k^{N+1} &= \mu_k^N + \frac{p(\omega_k \mid \vec{x}_{N+1})}{\sum_{i=1}^{N+1} p(\omega_k \mid \vec{x}_i)}(\vec{x}_{N+1} - \mu_k^N) \\ \sqcup_k^{N+1} &= \sqcup_k^N + \frac{p(\omega_k \mid \vec{x}_{N+1})}{\sum_{i=1}^{N+1} p(\omega_k \mid \vec{x}_i)}((\vec{x}_{N+1} - \mu_k^N)(\vec{x}_{N+1} - \mu_k^N)^T - \sqcup_k^N) \end{aligned}$$

- For L-recent window update [MRG97] (used at time $N \geq L$) we have:

$$\begin{aligned} w_k^{N+1} &= w_k^N + \frac{1}{L}(p(\omega_k \mid \vec{x}_{N+1}) - w_k^N) \\ \mu_k^{N+1} &= \mu_k^N + \frac{1}{L}(\frac{p(\omega_k \mid \vec{x}_{N+1})\vec{x}_{N+1}}{w_k^{N+1}} - \mu_k^N) \\ \sqcup_k^{N+1} &= \sqcup_k^N + \frac{1}{L}(\frac{p(\omega_k \mid \vec{x}_{N+1})(\vec{x}_{N+1} - \mu_k^N)(\vec{x}_{N+1} - \mu_k^N)^T}{w_k^{N+1}} - \sqcup_k^N) \end{aligned}$$

B.2 *DLT* Algorithm

To understand the DLT algorithm mentioned in chapter 3.3, we need some mathematical background, we will give now.

All our considerations will result in how to actually compute the projection matrix H, s.t.

$$H\,\underline{x_i} = \underline{x_i'},\ \forall i = 1, \dots n.$$

By definition of the vector cross product $\times$, we can instead of searching for an H s.t., $H\,\underline{x_i} = \underline{x_i'}$, compute

$$\underline{x_i'} \times H\underline{x_i} = \underline{0}$$

.
Here, a simple linear solution is possible to find.

For notational convenience, we denote the 9 unknowns of H by $h_1, \dots, h_9$ respectively. So,

$$H = \begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{pmatrix}.$$

Furthermore, we will denote with h^{j^T} the $j-th$ $(j = 1, \dots 3)$ row of H, e.g., $h^{1^T} = (h_1, h_2, h_3)$.

This enables us to write

$h = \begin{pmatrix} h^{1^T} \\ h^{2^T} \\ h^{3^T} \end{pmatrix}$, and therefore,

$$H\underline{x_i} = \begin{pmatrix} h^{1^T}\underline{x_i} \\ h^{2^T}\underline{x_i} \\ h^{3^T}\underline{x_i} \end{pmatrix}.$$

This helps us to reformulate the cross product as

$$\underline{x_i'} \times H\underline{x_i} = \begin{pmatrix} y_i' h^{3^T}\underline{x_i} - w_i' h^{2^T}\underline{x_i} \\ w_i' h^{1^T}\underline{x_i} - x_i' h^{3^T}\underline{x_i} \\ x_i' h^{2^T}\underline{x_i} - y_i' h^{1^T}\underline{x_i} \end{pmatrix} = 0.$$

With $\underline{x_i}^T h^j = {h^j}^T \underline{x_i}$, we now can write

$$\begin{bmatrix} 0^T & -w_i'\underline{x_i}^T & y_i'^T \\ w_i'\underline{x_i}^T & 0^T & -x_i'\underline{x_i}^T \\ -y_i'\underline{x_i}^T & x_i'\underline{x_i}^T & 0^T \end{bmatrix} \begin{pmatrix} h^1 \\ h^2 \\ h^3 \end{pmatrix} = 0.$$

This has the form $A_i\, h = 0$, with $A_i \in \mathbb{R}^{3\times 9}$ and $h \in \mathbb{R}^{3\times 3}$, and will be written as

$$A_i\, h = 0.$$

Note: We should note that only the first two rows of A_i are linearly independent and so, for speeding up the algorithm, we could discard the third row and obtain

$$\begin{bmatrix} 0^T & -w_i'\underline{x_i}^T & y_i'^T \\ w_i'\underline{x_i}^T & 0^T & -x_i\underline{x_i}^T \end{bmatrix} \begin{pmatrix} h^1 \\ h^2 \\ h^3 \end{pmatrix} = 0.$$

This will speed up the algorithm, but also lead to not such exact estimations, and so we decided to use the algorithm without speedup.

From the n matrices $A_i \in \mathbb{R}^{3\times 9}$ (representing in its two linearly independent rows the two equations given by every point correspondence), we assemble a single matrix $A \in \mathbb{R}^{3n\times 9}$

$$A = \begin{pmatrix} A_1^1 \\ A_1^2 \\ A_1^3 \\ A_2^1 \\ A_2^2 \\ A_2^3 \\ \vdots \\ A_n^1 \\ A_n^2 \\ A_n^3 \\ 0 \dots 0 \end{pmatrix},$$

where A_i^j denotes the j-th row of A_i. Our problem can now be rewritten as solving

$$A\, h = 0,$$

where $h \in \mathbb{R}^{9\times 1}$ holds the 9 entries of our matrix H we are striving for, and hence, we search for a solution $h \neq 0$.
This solution can only be determined up to a scale factor, so to be deterministic, we

choose a norm of 1, i.e., $||h|| = 1$.

Usually, our point measurements are inexact, they will suffer from noise, i.e., we cannot get an exact solution, but strive for a best approximation. To do so, we search a vector h minimizing a suitable cost function, e.g., the norm $||A\,h||$. Further and more elaborate cost functions can be found in chapter 3.2 of [HZ04].
This amounts in minimizing the quotient $||A\,h||/||h||$.
One can show that this minimizer is just the (unit) eigenvector of $A^T\,A$ corresponding to the smallest eigenvalue.
Equivalently, we can use the unit singular vector corresponding to the smallest singular value of A, obtained by a *Singular Value Decomposition (SVD)* of A.
This SVD yields a decomposition of A as the product of three matrices U, D and V^T.

$$A = U\,D\,V^T\,,$$

where D is a diagonal matrix of the singular values s_i, i.e., $D = \mathrm{diag}(s_1, \ldots, s_n)$ and U and V^T are matrices, made up by the singular vectors $\underline{v_i}$.
For more informations on SVD, we refer the reader to the corresponding appendix section in [HZ04].
If we now order the singular values of D in a descending order, i.e.,
$s_1 \geq s_2 \geq \ldots \geq s_n$, we obtain h just as the last row of V^T, which is the last column of V. From h we can obtain H by above formula, H is just a 3×3 matrix built from the 9×1 matrix h.

B.2.1 Normalizing

The ideas above will only work if the origin of the world- and image-coordinate system will correspond, and if the coordinate axis have the same scale and orientation, which will literally never be the case in our applications.
Therefore, we need a *normalizing transformations* T and T', which normalize our points by
$\underline{\tilde{x_i}} = T\,\underline{x_i}$ and $\underline{\tilde{x_i'}} = T'\,\underline{x_i'}$, respectively.

Normalizing transformations T and T'

The goals of applying a transformation T are (for convenience, we will skip writing T and T' in the remainder and just use T. The formulae for T' are obtained by just substituting $\underline{\tilde{x_i}}$ with $\underline{\tilde{x_i'}}$)

1. Translation by a matrix T_t, to bring the *centroid* $\underline{c}$ of all points $\underline{x_i}$ to the origin.

2. Scaling by a matrix T_s, to make the *average distance* $\bar{d}$ of all points $\underline{x_i}$ to the origin equal to $\sqrt{2}$.

Hereby, the centroid is computed as $\underline{c} = \begin{pmatrix} (\sum_{i=1}^{n} x_i)/n \\ (\sum_{i=1}^{n} y_i)/n \end{pmatrix}$,

and hence,

$$T_t = \begin{pmatrix} 1 & 0 & -(\sum_{i=1}^{n} x_i)/n \\ 0 & 1 & -(\sum_{i=1}^{n} y_i)/n \\ 0 & 0 & 1 \end{pmatrix}.$$

The average distance to the origin is computed as

$$\bar{d} = (\sum_{i=1}^{n} \sqrt{x_i^2 + y_i^2})/n$$

.

Hence, the scaling factor s, normalizing the distance $\bar{d}$ to $\sqrt{2}$ is simply obtained via $s = \sqrt{2}/\bar{d}$.

And hence,

$$T_s = \begin{pmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & s \end{pmatrix}.$$

As we want to do a translation followed by a scaling we get $\underline{\tilde{x_i}} = T_s(T_t \underline{x_i})$ and hence $T = T_s\, T_t$.

A last thing to mention is that we finally apply our DLT algorithm to the normalized point set $\{\underline{\tilde{x_i}} \leftrightarrow \underline{\tilde{x_i'}}\}$, yielding a homography matrix $\tilde{H}$, which has to be re-normalized via

$$H = T'^{-1}\, \tilde{H}\, T.$$

Final note: A homography transformation $H\, \underline{x_i} = \underline{x_i'} = (x_i', y_i', w_i')^T$ will usually lead to $w_i' \neq 1$, which is not desired. So, we also have to normalize the results via $\underline{x_i'} := \frac{1}{w_i'}\, \underline{x_i'}$, to set $w_i' = 1$.

B.2.2 Actual *DLT* Algorithm

The ideas above are resembled in the simple *DLT* algorithm.

Get user input for n point correspondences: First, we obtain the measured point correspondences from the user, i.e., we have $\underline{x_i} \leftrightarrow \underline{x_i'}$, $\forall i = 1, \ldots, n$

Compute A_i for all $i = 1, \ldots, n$: This can be done directly from the obtained point correspondences $\underline{x_i} \leftrightarrow \underline{x_i'}$ with $\underline{x_i} = (x_i, y_i, w_i)^T$:

$$A_i = \begin{bmatrix} 0^T & -w_i' \underline{x_i}^T & y_i' \underline{x_i}^T \\ w_i' \underline{x_i}^T & 0^T & -x_i \underline{x_i}^T \\ -y_i' \underline{x_i}^T & x_i' \underline{x_i}^T & 0^T \end{bmatrix} \in \mathbb{R}^{2 \times 9}.$$

Resemble matrix A: The matrix A is resembled from the two independent equations each point correspondence gives us.

$$A = \begin{pmatrix} A_1^1 \\ A_1^2 \\ A_1^3 \\ A_2^1 \\ A_2^2 \\ A_2^3 \\ \vdots \\ A_n^1 \\ A_n^2 \\ A_n^3 \\ 0 \dots 0 \end{pmatrix}.$$

Note that we have: $A \in \mathbb{R}^{12\times 9}$ and $h \in \mathbb{R}^9$.

Solve system of equations for h via SVD: To solve the system of equations $A\,h = 0$ for h, where we strive for a non-trivial solution $h \neq 0$.
This is done a SVD of A.

Determine H from h: Remember that our final goal was to determine the projection matrix H.
Now, as we computed h we can compute H by the above mentioned formula

$$H = \begin{pmatrix} h_1 & h_2 & h_3 \\ h_4 & h_5 & h_6 \\ h_7 & h_8 & h_9 \end{pmatrix}.$$

where $h = (h_1, h_2, h_3, h_4, h_5, h_6, h_7, h_8, h_9)^T$.

B.2.3 More than 4 Point Correspondences

Of course, we could measure more than 4 point correspondences in our room. As one can see, this will result in an over-determined set of equations $A\,h = 0$.
Here, we will also strive for a approximate solution, like described above, which minimizes a suitable cost function. Also here, the requirement of $||h|| = 1$ will make sense and we remain with minimizing $||A\,h||/||h||$ via a SVD.
A discussion of different cost functions can be found in chapter 3.2 of [HZ04].

B.3 Conversion from RGB to HSV Color Space

Our skin detection uses HSV images, but our cameras only provide us with RGB images. A conversion mapping $F : \mathbb{R}^3 \longrightarrow \mathbb{R}^3, (r, g, b) \mapsto (h, s, v)$, can be found in [GW02]:

Let $max := \max\{r, g, b\}$ and $min := \min\{r, g, b\}$.

Then $F(r, g, b) = (h, s, v)$ with

$$h = \begin{cases} \text{undefined} = 0, & max = min \\ 60 \cdot \frac{g-b}{max-min} + 0, & \text{if } max = r \text{ and } g \geq b \\ 60 \cdot \frac{g-b}{max-min} + 360, & \text{if } max = r \text{ and } g < b \\ 60 \cdot \frac{b-r}{max-min} + 120, & \text{if } max = g \\ 60 \cdot \frac{r-g}{max-min} + 240, & \text{if } max = b \end{cases},$$

$$s = \begin{cases} 0, & \text{if } max = 0 \\ 1 - \frac{min}{max}, & \text{else} \end{cases},$$

$$v = max.$$

Appendix C

Implementation Details

In this Appendix, we want to present details of our implementation we described in chapter 4.

C.1 Classes and Modules

In this section we want to give a short presentation all of the classes and modules used in MaMUT.

`aux_functions.cpp:` This module contains all the small auxiliary functions we needed sometimes, and were not part of OpenCV, like
`double compute_distance_centers (CvPoint, CvPoint)` to compute the distance of two points.

`cam_capture.cpp:` This module is used to capture images from a specified camera, given the username, password and a folder for temporarily storing the captured images. This is done via calling
`IplImage* capture (int cam_nr, string userpass, string folder)`, which relies on the approach presented in section 4.6.

`compute_homography.cpp:` This file will be compiled to a small program which will compute a homography matrix H, given 4 point correspondences, like described in section 3.3.
The usage is `./homography <input(measurements)> <output(matrix H)>`.

`EHeapOutput.h / .cpp:` This class establishes a connection to the EventHeap for outputting tracking results, like described in section 4.12. The constructor assigns URL and port of the server running the EventHeap:
`EHeapOutput(string URL, int port)`. The class also offers a method to send `TrackerEvent`-objects to the EventHeap via
`bool write_to_EHeap(TrackerEvent*, ...)`.

`Homography.h / .cpp:` This class holds the computed projection matrix H, and offers a method to project points (`CvPoint homography_projection (CvPoint)`), like described in section 3.3.

`mamut.cpp:` This module contains the `main()`-method and therefore initializes and runs MaMUT, by calling modules and instantiating classes listed here.

`Person.h / .cpp:` This class represents the persons tracked. A detailed description is given in section C.4.

`region_detection.cpp:` This module is used to merge lots of initial blob regions (from the background subtraction and connected components labeling algorithms) to some person regions, via `CvSeq* detect_regions (... ,CvSeq* ConnComp, double TCC1_min, double ...)`, incorporating the specified thresholds. A description can be found in section 3.2.1.

`ShelfTracker.h / .cpp:` This class is similar to `TrackingDB`, except that it tracks skin regions of the shelf camera (micro-tracking) instead of person regions from the room cameras.

`skin_detection.cpp:` This module allows us to create a skin color model, i.e., a hue-histogram, via
`CvHistogram* create_skin_model (string skin_model_name, ...)`. Furthermore, it provides us with a function to find skin regions in person regions: `CvSeq* skin_detection(..., CvHistogram* skin_model, double TSkin, double ..., CvRect roi, CvSeq* SkinComp, ...)`, like described in section 3.4.

`SkinRegion.h / .cpp:` This class is quite similar to the `Person` class, but it represents the skin region objects of the `ShelfTracker`.

`statist_bg_subtr.cpp:` This module performs a statistical background subtraction (see section 3.2.1) via
`IplImage* stat_bg_subtr (IplImage*, CvBGStatModel* bg_model, ...)`.

`TrackerEvent.h / .cpp:` This class represents the results of our tracking task. In its constructor we specify all collected results:
`TrackerEvent (int number_of_persons, vector<Person*> persons, ..)`.
In addition, it offers a method to create a string which can be send to the EventHeap via an HTTP-request: `string get_HTTP_string(...)`. Details are given in section C.5.

`TrackingDB.h / .cpp:` This class holds the data structure for saving tracking results of the person tracker of the room cameras and offers methods to track persons like described in section 3.2.2. Details are given below.

`TrackingFuser.h / .cpp:` This class fuses the results of the `TrackingDB` and the `ShelfTracker` classes, by assigning the skin regions of the latter to the appropriate persons of the first.

C.2 Config Files

As MaMUT uses a config file to let the user specify configuration parameters, like paths of images, FPS-rate, and so on, the first thing to do in MaMUT is to parse the config file.
For convenience, we used a config file parser class available to download from [Wag].
If we `#include "ConfigFile.h"`, we can create an `ConfigFile` object, specifying the path to the file (`ConfigFile config ( "../files/WWCam02.bcf" )`).
This objects then allows us to simply read parameters into variables.
If we, for example, want to read the integer value for the FPS-rate, we just do
`int FPS = config.read<int>("FPS", 1);` where we also specified an optional default value of 1 for this parameter.

C.2.1 Example Config File

In the following we give a commented config file for MaMUT.

```
## --- GENERAL options -------------------------------------
TRACKBAR_MAIN       = true      # "=true" if we want to have
                                # trackbars to adjust
                                # MaMUT-Runtime-Parameters

TRACKBAR_THRESHOLDS = true      # "=true" if we want to have
                                # trackbars to adjust
                                # Thresholds-Parameters

HDD = true      # "=true" if we use images from HDD,
                # "=false" if we capture live from camera

BGSUBTR_T = false       # ="true" if we want to manually
                        # specify parameters for our
                        # statistical background subtraction

COUT = true     # "=true" if we want to print results
                # of tracking to console

MINIMAL_COUT = true     # "=true" if we want to have a minimal
                        # output of results to console.

WINDOW     = true       # "=true" if we want to output
                        # images in windows

AUX_WINDOW = false      # "=true" if we want to output ALL
```

```
                        # created images, like BG-Model,
                        # Skin Probability Image, ...

IMAGES = false  # "=true" if we want to save images to HDD

FPS = 6         # Frames per Second - rate,
                # one wants to achieve.
                # Should be between 1 and 12 !

STEP = true     # "=true" if we want to proceed from frame
                # to frame via a keypress,
                # "=false" if we proceed from frame to frame
                # after (1/FPS) seconds

EHEAP = true    # "=true" if we want to post results
                # of tracking to EHeap

## --- EHeap options ----------------------------------------
  EHEAP_URL  = http://wwplayer09.cs.uni-sb.de
                                      #URL of EHeap-Server
  EHEAP_PORT = 2486
                                      #Port of EHeap-Server

## --- Homography options -----------------------------------
  H_FILENAME__ROOM  = ../files/H__room.txt
                        # File containing Homography matrix H
                        # for the ROOM

  H_FILENAME__SHELF = ../files/H__shelf.txt
                        # File containing Homography matrix H
                        # for the SHELF

## --- Skin Detection options -------------------------------
  SKIN_MODELNAME__SHELF = ../images/skin_model/BAIR/skin_
                        # Images of our skin model.
                        # Give start of name, e.g., "skin_0"
                        # for skin_01.jpg, skin_02.jpg, ...

                        # and ...

  SKIN_SAMPLES_NUMBER__SHELF = 6
                        # number of samples to process

## --- Shelf Carpet (in IMAGE coordinates !!!) --------------
 # Marks regions of interest (like shelfs).
 # Will be projected to WORLD coordinates
 # in TrackingDB-Constructor !!!

  SHELF_CARPET_0_X = 135
  SHELF_CARPET_0_Y = 170
```

```
  SHELF_CARPET_1_X = 70
  SHELF_CARPET_1_Y = 215

  SHELF_CARPET_2_X = 145
  SHELF_CARPET_2_Y = 240

  SHELF_CARPET_3_X = 180
  SHELF_CARPET_3_Y = 180

## --- Threshold Parameter Scale options --------------------
# Threshold parameters are computed according to image sizes
# 's' and a scale factor 'f' via T = s / f. These factors 'f'
# have to be specified below.
# NOTE 1 : Increasing values of 'f' leads to smaller values
# of the threshold !
# NOTE 2 : 'f' has to be a double in general,
# i.e., append '.0' to every value !

#ROOM Cams
F_TCC1_MIN__ROOM = 5640.0   # Region Detection: Threshold for
                            # too small detected regions,
                            # at hysteresis step 1

F_TCC1_MAX__ROOM =    3.0   # Region Detection: Threshold for
                            # too large detected regions,
                            # at hysteresis step 1

F_TCC2__ROOM     =   17.0   # Region Detection: Threshold for
                            # too small detected regions,
                            # at hysteresis step 2

F_TD__ROOM       =    5     # Region Detection: Distance
                            # Threshold for adjacent regions

#SHELF Cam
F_TCC1_MIN__SHELF = 845.0   # Region Detection: Threshold for
                            # too small detected regions,
                            # at hysteresis step 1

F_TCC1_MAX__SHELF =   2.0   # Region Detection: Threshold for
                            # too large detected regions,
                            # at hysteresis step 1

F_TCC2__SHELF     = 154.0   # Region Detection: Threshold for
                            # too small detected regions,
                            # at hysteresis step 2

F_TD__SHELF       =   5     # Region Detection:
                            # Distance Threshold
                            # for adjacent regions
```

```
# -- Skin Detection ---
F_SKIN_PROB__SHELF    = 0.75    # Which skin probabilities will
                                # count as skin

F_SKIN_AREA__SHELF    = 94.0    # Threshold for too
                                # small skin regions

F_SKIN_DIST__SHELF    = 1126.0 # Distance Threshold for
                                # adjacent skin regions

F_SKIN_HEIGHT__SHELF = 1300    # Threshold for Skin Region height,
                                # too high regions are cropped !


## --- HDD options -----------------------------------------
IMAGE_LOCATION__ROOM  =  ../images/capture/BAIR/room/cam2_

IMAGE_LOCATION__SHELF = . ./images/capture/BAIR/room/cam1_
                # where we stored our (numbered) images,
                # e.g., image_001.jpg, image_002.jpg, ...
                # Give beginning of files, e.g., "cam2_"
                # EXAMPLE:
                # ../images/capture/cam2Duo/cam2_

INIT_TICK =  3          # Timestamp to start with
END_TICK  = 72          # Timestamp to finish

# --- Realtime options ------------------------------------
CAM__ROOM   =  2        # Number of Camera,
                        # n ==> 'wwcam0n.cs.uni-sb.de',
                        # n=1,2,3

CAM__ROOM_2 =  3        # If '-1' then no second room cam is
                        # attached !!!

CAM__SHELF  =  1

# COMMON Authenticating informations for the 2 cameras:
USER = root             # Username for authenticating to
                        # camera specified above
PASS = *****            # Password of user specified above

IMAGE_FOLDER = ../images/   #Folder to store captured image
```

C.3 Using OpenCV

After successfully installing OpenCV according to the mentioned documentation, all library functions are available if we include the OpenCV header and the header for auxiliary helper functions (very useful!):

```
#include "cv.h"
#include "cvaux.h"
```

If we want to use the built-in GUI for visualizing our results, we need in addition:

```
#include "highgui.h"
```

For further informations and help on particular problems with OpenCV, there exists a quite vital Yahoo Group for OpenCV [Yah], where one can search through former posts or post new questions oneself.
In addition, they offer some nice tutorials and HowTo's, e.g., how to build OpenCV projects with Eclipse or nice tutorials on camera calibration using the famous chessboard pattern (see Glossary, Appendix A).

C.4 The `Person` Class

```
class Person {

private:
 // Bounding Box of merged CC
 CvRect rect;

 // ID of person
 int id;

 // Detected skin regions for person
 vector<SkinRegion*> skin_regions;

 // Color model of persons appearance
 CvHistogram* color_model;

 // Timestamp, when person was tracked for the last time
 int t;

 // Timestamp, when person was tracked for the first time
 int t_0;

 // Tracking state
 bool moving;
 bool reliable;
 bool inside;

 // States if person is at shelf
 bool at_shelf;
```

```
 // Projection matrix for homography computation
 Homography* H;

 // Tracepoint of rectangle (footpoint) for trajectory
 CvPoint tracepoint;

 // Tracepoint in world coordinates
 CvPoint tracepoint_world;

 // Trajectory of bounding box tracepoints
 vector<CvPoint> trajectory;
 vector<CvPoint> trajectory_world;

 // Direction in which person is currently moving
 int dir;       // -1 = Left, 0 = None, 1 = Right

 // Denotes which camera tracks the person
 // element of {ROOM_1, ROOM_2}
 string CAM;


public:
 Person(CvRect r, int t, Homography* H,
         IplImage* f, IplImage* FG, string CAM);

 ~Person();

 // Add tracepoint to trajectory of person
 void add_to_trajectory(CvPoint c);
 void add_to_trajectory_world(CvPoint c);

 // Draw persons bounding boxes and traces in frame
 void draw_person(IplImage* f);
 void draw_trace(IplImage* f);

 // Create initial color model for person
 CvHistogram* init_color_model(IplImage* f,
                               IplImage* FG);

 // Update color model
 void update_color_model(IplImage* f, IplImage* FG);

 // Clears assigned skin regions,
 // needed before adding skin regions by Fuser
 void clear_skin_regions();

 ...
};
```

C.5 The TrackingDB Class

```
class TrackingDB {

private:
 // All tracked persons, same vector for all TrackingDB
 // classes for each room camera
 vector<Person*>* persons;

 // Connection to EventHeap
 EHeapOutput* EH;

 // Timestamp of current frame
 int act_time;

 // Projection matrix for homography computation
 Homography* H;

 // WORLD and IMAGE  coordinates of shelf carpet
 // If no carpet is needed, just pass an array which contains
 // only (-1, -1)
 CvPoint shelf_carpet[];
 CvPoint shelf_carpet_img[];

 // Denotes if we have a carpet or not
 bool has_carpet;

 // WORLD coordinate of the center of the shelf,
 // used to compute distances to shelf
 CvPoint shelf_center;

 // Denotes to which camera the DB is connected
 // element of {ROOM_1, ROOM_2}
 string CAM;

 // Update status of tracked person
 void update_person(int index, CvRect r, IplImage* f,
                    IplImage* FG, bool COUT);


 // Add a new person to track
 void add_person(CvRect r, IplImage* f, IplImage* FG, bool COUT);
 void add_person(Person* p, bool COUT);

 // Remove a person to track,
 // i.e., set them 'non-moving' or 'outside'
 void remove_person(int index, IplImage* f, bool COUT);

 // Match rectangle to all persons in database
 void match(CvRect r, IplImage* f, IplImage* FG, bool COUT);
```

```
 // Checks if color model of a ConnComp r and
 // a person (color_model_person) matches
 bool match_color_models
    (CvRect r, CvHistogram* color_model_person,
     IplImage* f, IplImage* FG);

 //Tests if a person is near the image border
 bool near_border(Person* p, IplImage* f);

 // Tests if a person stands in front of the shelf
 bool is_at_shelf(Person* person);


public:
 TrackingDB(int time, Homography* H, CvPoint shelf_carpet[],
            vector <Person*>* persons, string CAM);

 ~TrackingDB();

 // Track all ConnComp regions in comp at time t
 void track(CvSeq* comp, int t, IplImage* f, IplImage* FG, bool COUT);

 // Return all persons standing in front of shelf
 vector<Person*> get_persons_at_shelf();

 ...
};
```

Point	Image Coord.		Room Coord.	
	x'_i	y'_i	x_i	y_i
$\underline{x_1}$	330	130	0	0
$\underline{x_2}$	130	180	295	-70
$\underline{x_3}$	65	240	285	-160
$\underline{x_4}$	350	240	110	-230

Table C.1: The four measured point correspondences in our room for camera $Room_1$.

Point	Image Coord.		Room Coord.	
	x'_i	y'_i	x_i	y_i
$\underline{x_1}$	333	119	380	-447
$\underline{x_2}$	54	183	0	-447
$\underline{x_3}$	67	237	11	-355
$\underline{x_4}$	350	238	230	-267

Table C.2: The four measured point correspondences in our room for camera $Room_2$.

C.6 Measurements, Results and Implementation of the *DLT* Algorithm

C.6.1 Measured Point Correspondences

For the room, we obtained the following correspondences as shown in table C.1. Note that the ground plane of our room is the $x, -z$ –plane.
A last thing to notice is that the world coordinates are measured in centimeters, i.e., a point $[110, -230]$ is located $110cm$ in x- and $230cm$ in $-z$ –direction from the world origin $[0, 0]$.
Also for the second room camera, we will state the measurements and the results.

For the shelf, we got the corresponding as can be seen in table C.3.

Point	Image Coord.		Shelf Coord.	
	x'_i	y'_i	x_i	y_i
$\underline{x_1}$	344	69	0	0
$\underline{x_2}$	47	66	705	0
$\underline{x_3}$	102	172	79	97
$\underline{x_4}$	267	172	0	97

Table C.3: The four measured point correspondences at our shelf for camera $Shelf$.

C.6.2 *DLT* Algorithm

To implement the *DLT* algorithm we simply translated the sketched algorithm in C++-code. We store the points $\underline{x_i}$ and $\underline{x_i'}$ in arrays `double x[n]` and `double x_p[n]`. The normalizing transformation matrices are computed as

```
double T[9] =
  { s, 0.0, (-c.x*s),
   0.0, s, (-c.y*s),
   0.0, 0.0, s
   };

double T_prime[9] =
  { s_prime, 0.0, (-c_prime.x * s_prime),
    0.0,   s_prime, (-c_prime.y * s_prime),
    0.0,     0.0,  s_prime
  };
```

where the normalizing factors `s` and `c.x, c.y` are computed as shown in section 3.3.2. With them we transform our points via

```
for (int i=0; i<n; i++)  {
   x_tilde[i].x =
     T[0] * x[i].x + T[1] * x[i].y + T[2] * x[i].w;

   x_tilde[i].y =
     T[3] * x[i].x + T[4] * x[i].y + T[5] * x[i].w;

   x_tilde[i].w =
     T[6] * x[i].x + T[7] * x[i].y + T[8] * x[i].w;

   x_p_tilde[i].x =
     T_prime[0] * x_p[i].x + T_prime[1] * x_p[i].y
   + T_prime[2] * x_p[i].w;

   x_p_tilde[i].y =
     T_prime[3] * x_p[i].x + T_prime[4] * x_p[i].y
   + T_prime[5] * x_p[i].w;

   x_p_tilde[i].w =
     T_prime[6] * x_p[i].x + T_prime[7] * x_p[i].y
   + T_prime[8] * x_p[i].w;
}
```

So, we can assemble our matrix A as the array

```
double A_array[12*9] =
{ 0.0, 0.0, 0.0,
-x_p_tilde[0].w * x_tilde[0].x , -x_p_tilde[0].w * x_tilde[0].y,
-x_p_tilde[0].w * x_tilde[0].w,
 x_p_tilde[0].y * x_tilde[0].x ,  x_p_tilde[0].y * x_tilde[0].y,
 x_p_tilde[0].y * x_tilde[0].w,
```

```
// ------------------------------------------------------------
 x_p_tilde[0].w * x_tilde[0].x ,  x_p_tilde[0].w * x_tilde[0].y,
 x_p_tilde[0].w * x_tilde[0].w,
 0.0, 0.0, 0.0,
-x_p_tilde[0].x * x_tilde[0].x , -x_p_tilde[0].x * x_tilde[0].y,
-x_p_tilde[0].x * x_tilde[0].w,
// ------------------------------------------------------------
-x_p_tilde[0].y * x_tilde[0].x, -x_p_tilde[0].y * x_tilde[0].y,
-x_p_tilde[0].y * x_tilde[0].w,
 x_p_tilde[0].x * x_tilde[0].x, x_p_tilde[0].x * x_tilde[0].y,
 x_p_tilde[0].x * x_tilde[0].w,
 0.0, 0.0, 0.0,
// ------------------------------------------------------------
 ...                                                        ...
 ...                                                        ...
 ...                                                        ...
// ------------------------------------------------------------
 0.0, 0.0, 0.0,
-x_p_tilde[3].w * x_tilde[3].x , -x_p_tilde[3].w * x_tilde[3].y,
-x_p_tilde[3].w * x_tilde[3].w,
 x_p_tilde[3].y * x_tilde[3].x ,  x_p_tilde[3].y * x_tilde[3].y,
 x_p_tilde[3].y * x_tilde[3].w,
// ------------------------------------------------------------
 x_p_tilde[3].w * x_tilde[3].x ,  x_p_tilde[3].w * x_tilde[3].y,
 x_p_tilde[3].w * x_tilde[3].w,
 0.0, 0.0, 0.0,
-x_p_tilde[3].x * x_tilde[3].x , -x_p_tilde[3].x * x_tilde[3].y,
-x_p_tilde[3].x * x_tilde[3].w,
// ------------------------------------------------------------
-x_p_tilde[3].y * x_tilde[3].x, -x_p_tilde[3].y * x_tilde[3].y,
-x_p_tilde[3].y * x_tilde[3].w,
 x_p_tilde[3].x * x_tilde[3].x, x_p_tilde[3].x * x_tilde[3].y,
 x_p_tilde[3].x * x_tilde[3].w,
 0.0, 0.0, 0.0
};
```

The SVD of A is done via a OpenCV function `cvSVD( &A, D, U, VT, CV_SVD_V_T );`, where we have to make `CvMat` objects out of our arrays via
`CvMat A; cvInitMatHeader( &A, 12, 9, CV_64FC1, &A_array );`, for example.

C.6.3 Resulting Matrices H

The resulting matrix H is written to a text file in a format compatible with the `Homography` objects of MaMUT.
In our case the matrix H is for camera $Room_1$: H_{room_1}:

$$\begin{pmatrix} 0.011953387 & -0.028514934 & -0.237676287 \\ 0.008839702 & 0.053470653 & -9.868286717 \\ -0.000014811 & -0.000169101 & 0.019426561 \end{pmatrix}$$

For the camera $Room_2$: H_{room_2}:

$$\begin{pmatrix} -0.073631361 & 0.000225558 & 3.934816339 \\ -0.018459078 & 0.011917704 & 28.910878765 \\ -0.000029886 & -0.000336970 & -0.004047287 \end{pmatrix}$$

For the camera $Shelf$ we obtained as H_{shelf}:

$$\begin{pmatrix} -0.007814197 & -0.005785512 & 3.081498727 \\ -0.000100925 & 0.014987432 & -0.984426987 \\ -0.000001040 & -0.000124018 & 0.037757965 \end{pmatrix}$$

Bibliography

[AJN05] Agarwal, Anubhav ; Jawahar, C. V. ; Narayanan, P. J.: A Survey of Planar Homography Estimation Techniques / International Institute of Information Technology Hyderabad (Deemed University). 2005. – Forschungsbericht

[AL04] Argyros, A. A. ; Lourakis, M. I. A.: Real-Time Tracking of Multiple Skin-Colored Objects with a Possibly Moving Camera. In: *European Conference on Computer Vision*, 2004, S. Vol III: 368–379

[Axi] Axis Communications: *AXIS 213 PTZ Network Camera*. http://www.axis.com/products/cam_213/index.htm, . – [Online, last accessed 2006-11-02]

[Bau95] Baumberg, A.: *Learning Deformable Models for Tracking Human Motion*, University of Leeds, Diss., 1995

[Bau96] Baumberg, A.: Hierarchical Shape Fitting using an Iterated Linear Filter. In: *British Machine Vision Conference*, 1996, S. Model Fitting, Matching, Recognition

[BH95] Baumberg, A. ; Hogg, D.: An Adaptive Eigenshape Model. In: *Proceedings of the 6th British Machine Vision Conference* Bd. 1, 1995, S. 87–96

[BK] Butz, A. ; Krüger, A.: A mixed reality room following the generalized peephole metaphor. In: *To appear in IEEE Computer Graphics & Applications*

[Bra98] Bradski, G. R.: Computer Vision Face Tracking For Use in a Perceptual User Interface. In: *Intel Technology Journal* (1998), 2nd Quarter

[Bra00] Bradski, G.: The OpenCV Library. In: *Dr. Dobb's Journal of Software Tools* 25 (2000), November, Nr. 11, S. 120, 122–125

[BS05] Brandherm, B. ; Schwartz, T.: Geo Referenced Dynamic Bayesian Networks for User Positioning on Mobile Systems. In: Strang, T. (Hrsg.) ; Linnhoff-Popien, C. (Hrsg.): *Proceedings of the International Workshop on Location- and Context-Awareness (LoCA), LNCS 3479* Bd. 3479 /

2005, Springer-Verlag Berlin Heidelberg, 2005 (Lecture Notes in Computer Science), S. 223–234

[CG92] Carriero, N. ; Gelernter, D.: Coordination Languages and Their Significance. In: *Communications of the ACM* 35 (1992), February, Nr. 2, S. 97–107

[DD92] DeMenthon, D. F. ; Davis, L. S.: Model-Based Object Pose in 25 Lines of Code. In: *Image Understanding Workshop*, 1992, S. 753–761

[FR97] Friedman, N. ; Russell, S.: Image Segmentation in Video Sequences: A Probabilistic Approach. In: Geiger, Dan (Hrsg.) ; Shenoy, Prakash P. (Hrsg.): *Proceedings of the 13th Conference on Uncertainty in Artificial Intelligence (UAI-97)*. San Francisco : Morgan Kaufmann Publishers, August 1997, S. 175–181

[FR02] Focken, D. ; R.Stiefelhagen: Towards Vision-Based 3-D People Tracking in a Smart Room. In: *ICMI*, IEEE Computer Society, 2002, S. 400–405

[GD96] Gavrila, D. ; Davis, L. S.: 3-D model-based tracking of humans in action: a multi-view approach. In: *CVPR*, IEEE Computer Society, 1996, S. 73–80

[Ger99] Gershenfeld, N.: *The Nature of Mathematical Modeling*. New York, NY, USA : Cambridge University Press, 1999

[GW02] Gonzalez, R. C. ; Woods, R. E.: *Digital Image Processing*. 2nd. Addison-Wesley, 2002. – 295 S.

[Haxa] Haxx: *cURL groks URLs*. http://curl.haxx.se, . – [Online, last accessed 2006-11-02]

[Haxb] Haxx: *libcurl - the multiprotocol file transfer library*. http://curl.haxx.se/libcurl/, . – [Online, last accessed 2006-11-02]

[Hec] Heckmann, D.: *UbisWorld.org*. http://www.ubisworld.org, . – [Online, last accessed 2006-11-02]

[HHD98] Haritaoglu, I. ; Harwood, D. ; Davis, L. S.: W4S: A real-time system for detecting and tracking people in 2 1/2-D. In: *European Conference on Computer Vision*, 1998, S. I: 877

[HHD00] Haritaoglu, I. ; Harwood, D. ; Davis, L. S.: W4: Real-Time Surveillance of People and Their Activities. In: *IEEE Trans. Pattern Anal. Mach. Intell* 22 (2000), Nr. 8, S. 809–830

[HZ04] Hartley, R. I. ; Zisserman, A.: *Multiple View Geometry in Computer Vision*. Second. Cambridge University Press, 2004

[Int] Intel: *Open Source Computer Vision Library (Official Homepage).* http://www.intel.com/technology/computing/opencv/, . – [Online, last accessed 2006-11-02]

[JF02] Johanson, B. ; Fox, A.: The Event Heap: A Coordination Infrastructure for Interactive Workspaces. In: *WMCSA*, IEEE Computer Society, 2002, S. 83–93

[JKS95] Jain, R. C. ; Kasturi, R. ; Schunck, B. G.: *Machine Vision.* McGraw-Hill, 1995

[KB01] Kaewtrakulpong, P. ; Bowden, R.: *An Improved Adaptive Background Mixture Model for Realtime Tracking with Shadow Detection.* April 2001

[KHM+00] Krumm, J. ; Harris, S. ; Meyers, B. ; Brumitt, B. ; Hale, M. ; Shafer, S. A.: Multi-Camera Multi-Person Tracking for EasyLiving. In: *Workshop on Visual Surveillance*, 2000

[KJM03] Kapralos, B. ; Jenkin, M. ; Milios, E.: Audio-visual localization of multiple speakers in a video teleconferencing setting. In: *International Journal of Imaging Systems and Technology, special issue on Facial Image Processing, Analysis and Synthesis* 13 (2003), June, Nr. 1, S. 95–105

[KS06] Khan, S. M. ; Shah, M.: A Multiview Approach to Tracking People in Crowded Scenes Using a Planar Homography Constraint. In: *European Conference on Computer Vision*, 2006, S. IV: 133–146

[KSS05] Kruppa, M. ; Spassova, M. ; Schmitz, M.: The Virtual Room Inhabitant - Intuitive Interaction With Intelligent Environments. In: *Proceedings of the 18th Australian Joint Conference on Artificial Intelligence (AI05)*, 2005

[LHGT] Li, L. ; Huang, W. ; Gu, I. Y. H. ; Tian, Q.: Foreground object detection from videos containing complex background. In: *Proceedings of the 11th ACM International Conference on Multimedia (MM-03)*, S. 2–10

[MJD+00] McKenna, S. J. ; Jabri, S. ; Duric, Z. ; Rosenfeld, A. ; Wechsler, H.: Tracking Groups of People. In: *Computer Vision and Image Understanding* 80 (2000), October, Nr. 1, S. 42–56

[MRG97] McKenna, S. J. ; Raja, Y. ; Gong, S.: Object Tracking Using Adaptive Color Mixture Models. In: *Lecture Notes in Computer Science* 1351 (1997)

[Now91] Nowlan, S. J.: *Soft competitive adaptation: neural network learning algorithms based on fitting statistical mixtures*, Carnegie Mellon University, Diss., 1991. – CS–91–126

[OMH99] Orwant, J. ; MacDonald, J. ; Hietaniemi, J. ; Oram, A. (Hrsg.): *Mastering Algorithms with Perl*. O'Reilly & Associates, Inc., 1999

[Poi] Point Grey Research: *Advanced Digital Camera Technology Products*. http://www.ptgrey.com, . – [Online, last accessed 2006-11-02]

[RSL99] Romano, R. ; Stein, G. P. ; Lee, L.: Monitoring Activities from Multiple Video Streams: Establishing a Common Coordinate Frame / MIT Artificial Intelligence Laboratory. 1999 (AIM-1655). – Forschungsbericht

[SB91] Swain, M.J. ; Ballard, D.H.: Color Indexing. In: *IJCV: International Journal of Computer Vision* 7 (1991)

[SBB+05] Stahl, C. ; Baus, J. ; Brandherm, B. ; Schmitz, M. ; Schwartz, T.: Navigational- and Shopping Assistance on the Basis of User Interactions in Intelligent Environments. In: *Proceedings of the IEE International Workshop on Intelligent Environments (IE)*, 2005

[SBF00] Sidenbladh, H. ; Black, M. J. ; Fleet, D. J.: Stochastic Tracking of 3D Human Figures using 2D Image Motion. In: *European Conference on Computer Vision*, 2000, S. II: 702–718

[SG99] Stauffer, C. ; Grimson, W. E. L.: Adaptive Background Mixture Models for Real-Time Tracking. In: *Proceedings of the IEEE Computer Science Conference on Computer Vision and Pattern Recognition (CVPR-99)*, 1999, S. 246–252

[Sie03] Siebel, N. T.: *Design and Implementation of People Tracking Algorithms for Visual Surveillance Applications*, Department of Computer Science, The University of Reading, Reading, UK, Diss., March 2003

[SM01a] Siebel, N. T. ; Maybank, S. J.: The Application of Colour Filtering to Real-Time Person Tracking. In: *Proceedings of the 2nd European Workshop on Advanced Video-Based Surveillance Systems (AVBS'2001)*, 2001, S. 227–234

[SM01b] Siebel, N. T. ; Maybank, S. J.: Real-Time Tracking of Pedestrians and Vehicles. In: *International Workshop on Performance Evaluation of Tracking and Surveillance*, 2001

[SM02] Siebel, N. T. ; Maybank, S. J.: Fusion of Multiple Tracking Algorithms for Robust People Tracking. In: *European Conference on Computer Vision*, 2002, S. IV: 373 ff.

[Sou] Sourceforge: *OpenCV Blob Extraction Library*. http://opencvlibrary.sourceforge.net/cvBlobsLib/, . – [Online, last accessed 2006-11-02]

[Wag] Wagner, R.: *Configuration File Reader for C++*. http://www-personal.umich.edu/ wagnerr/ConfigFile.html, . – [Online, last accessed 2006-11-02]

[Wik] Wikipedia: *Kalman Filter*. http://en.wikipedia.org/wiki/Kalman_filter, . – [Online, last accessed 2006-11-02]

[WKB04] Wahlster, W. ; Krüger, A. ; Baus, J.: *Resource-adaptive Cognitive Processes*. 2004. – Finanzierungsantrag, BAIR: User Adaptation in Instrumented Rooms

[Yah] Yahoo Groups: *Open Source Computer Vision Library Community*. http://groups.yahoo.com/group/OpenCV/, . – [Online, last accessed 2006-11-02]