Visitracker: Feature extraction by single-camera tracking for camera to camera matching

October 14, 2002

Niels P. Hietbrink
FNWI
Universiteit van Amsterdam

Visitracker: Feature extraction by single-camera tracking for camera to camera matching

Author:	Niels Hietbrink
Study:	Artificial Intelligence
Specialization:	Intelligent Autonomous Systems
Faculty:	FNWI
University:	Universiteit van Amsterdam
Location:	WCW, Kruislaan 403, Amsterdam
Date:	October 30, 2002
Supervisors:	Dr. Ir. B.J.A. Kröse
	Drs. W. Zajdel

Abstract

For fully automating surveillance, or in general, monitoring tasks, it is often required to be able to follow a person or object over multiple cameras. To do so, these objects have to be recognized and identified on differing backgrounds, lighting conditions and viewing angles. This thesis describes a method of gathering information about the appearance of an object, such that it can be recognized on different cameras.

A background subtraction method is used to find the foreground pixels in an image frame, after which these pixels are clustered and labeled. Then, features are extracted from these labeled pixel clusters. These features are used to track foreground objects on a single camera. While doing so, the features are also used to update the information a track collects of the object it is tracking. This information includes properties of the blob as well as statistical data on the tracking process. When tracking is complete, either when the object being tracked is no longer visible, or the tracking process does not receive any more input, the properties and statistics of each track are reported. This data can then be used to identify objects on different cameras.

Contents

List of Figures

List of Tables

List of Equations

Chapter 1

Introduction

A large part of the job of surveillance is following people or objects (collectively referred to as objects in the rest of the thesis) over multiple cameras. When an object moves out of the view of one camera, we expect it to appear at a different camera. We have no problem recognizing this object when it appears at this second camera. For a computer however, this is a far more complex task. To accomplish the task of tracking objects over multiple cameras we have split it into two subtasks. The first subtask, which is described in this thesis, is the tracking of objects on one camera. This will produce features, information about the appearance of an object. These features can then be used to accomplish the second subtask, which is the matching of objects on a camera with ones observed earlier on another camera.

As mentioned, this thesis focuses on the single camera object tracking. First, we will take a look at earlier research in this field and the approaches used. One of these approaches will be examined in more detail, as it will serve as the basis of our own system. The third section of this chapter will elaborate on the problem statement. The rest of the thesis covers the implementation of our system and the experiments done to test its suitability.

1.1 Context

This master's project is part of the Distributed Surveillance (DS) project. This is cooperation between FEL-TNO and the Intelligent Autonomous Systems group of the University of Amsterdam. The goal of the DS project is to develop a system capable of automatically tracking moving objects over multiple cameras. This system is supposed to be a set of distributed, possibly but not necessarily overlapping, intelligent sensors. Each sensor consists of a camera and a computer. Every sensor should be capable, based on extracted information, of tracking and, if possible identifying, objects it detects. Sensors can communicate so that objects can be tracked and followed throughout the entire system.

This thesis will focus on the tracking of moving objects using only one camera. The resulting information, gathered from this tracking process can then be used to track the objects as they move to other cameras. Camera images carry a large amount of information, such as position, size, shape, speed, direction and color about an object. After separating the foreground pixels from the background, these features can be extracted from the image to aid the tracking process. The goal of this masters project is to develop a system that can track objects within the view of one camera. This tracking uses features extracted from

a camera image, where objects have been separated by background subtraction as implemented by Frank Zwarthoed [16]. A track is a set of points (positions) at which an object was observed in subsequent image frames, together with characteristics (features) of the object, which are updated when a new point is added to the track. These characteristics will later be used to identify an object on a different camera. The trajectory of the object through the camera view can also be used to find the most likely camera the object is to pass next.

Other people have performed similar tasks. Schiele and Spengler [13] have designed a system in which multiple features are calculated for each blob (a cluster of foreground pixels, considered to contain the image of an object, further explained in section 2.1) after which they are integrated into one value. This is done using one of two combination systems. First, there is *Democratic Integration*, a self-adaptive multi-feature integration system where the different feature representations agree upon a common position estimation. In this system, not all features carry the same weight in the agreement. Redundant features (features that are not used in the position estimation, but are used to evaluate the tracking performance during the tracking) are used to update these weights for each of the features. Secondly, *Condensation*, a combination of multiple-hypotheses tracking and multi-feature integration based on a system by Isard and Blake [6] is used. This way, it is possible to track multiple objects simultaneously.

As an extension, Spengler and Schiele, together with Hannes Kruppa [12] describe an algorithm for switching between detection models. They illustrate this by tracking faces, based on a number of skin-color models. This model switching is used as a bootstrapping mechanism for failing features (failing features are features that change due to sudden environmental changes and therefore fail to provide an accurate position estimation. Using Democratic Integration, the only way to recover from this is by slowly adapting the used model, which decreases the system's reliability over a larger time), where a failing model is reset to a more appropriate one. In addition, it also provides a means of resetting the system when false positives cause deadlocks. False positives are hypotheses that are accepted while they are actually false. When this happens, the model will adapt to this hypothesis, even reinforcing it. Model switching will stop adapting when a different model is more appropriate, thus avoiding deadlocks.

Since we will be tracking persons, instead of parts of a person, it seems very hard to acquire good models for people wearing different types of clothing or being only half-detected. In addition, calculating the performance of every model, to find out which one fits the situation best is very time-consuming. While this algorithm might work well in well-defined tracking tasks, it is not suitable for general application, since you cannot track something that you have not modeled roughly first. While our task is to track objects in order to better describe their features, without first knowing them (which is required in order to use model-driven systems), this approach is not suitable for us.

A more suitable approach is described by Beymer [1]. Here, tracking is used to measure traffic parameters, such as vehicle counts, average speed and lane change frequencies. Because a problem with tracking vehicles is partial occlusion, not the entire car is tracked, but 'sub-features' are used (the specific sub-features

used in this system are corner features. Corners are detected when the (gray-scale) image gradient shows a steep increase or decrease both horizontally as well as vertically). Kalman filtering is used to decrease the effect of noisy image data.

To group the sub-features (necessary to find out which sub-features belong to the same vehicle) motion-based grouping is used. Here, point features moving rigidly together are grouped. As many frames as conceivably possible are used to make this robust. Because the decision to use background estimation had been taken before our tracking project started, this approach has not been considered.

1.2 The basis of our system

Intille, Davis and Bobick [4] have a situation that most resembles ours. They want to track multiple persons by using contextual information to decrease the information complexity. They use a closed-world assumption to do so. For a good understanding, it is necessary to know that they refer to tracks as objects. They are also using a background estimation algorithm to collect foreground pixels. They are firstly formed into clusters using a triple dilation, after which they calculate the bounding boxes of connected regions[1]. When two regions are sufficiently close together (5 pixels) they are merged into one region, after which a new bounding box is calculated. This is done exhaustively. Because large diagonally oriented clusters generally have a very large bounding box, they use an algorithm that estimates the smallest distance between the boundaries of two blobs for large clusters. The resulting connected regions are called blobs.

After this merging, features are collected for each blob, from the original image (the non-dilated image). The features used are size, position and average color. Then, matching between blobs and objects is done. Every object has four properties: average color, position, size and velocity. These properties are used to calculate four distances:

1. The two-dimensional distance between the average color of a blob and that of an object.
2. The two-dimensional distance between the position of a blob and the position of an object.
3. The two-dimensional distance between the expected position of a blob and its true position.
4. The one-dimensional difference between the size of an object's blob and the average size of all blobs in a track.

For each distance, a score matrix S is created, where S_{ij} is the match score of object *i* with blob *j*. These scores are normalized, so that $\sum S_{ij} = 1$. After weighting the four matrices, they are summed in one matching score matrix *M*. So then it is time to do the actual matching. This is separated in a number of steps. Important to know is, that objects can only enter or exit the video in the door area,

[1] Also see sections 2.2 and 2.3 for alternative methods.

part of the frame close to the door. To make sure that objects are not matched to blobs that are on the other side of the room, they use hard constraints to limit the matching.

Once the matching has been performed, the objects' properties and statistics are updated with the feature information of the new blob. The local-closed-world is used to determine whether to update certain properties. If the object is the only object in the local-closed-world, its properties can safely be updated, but if there are other objects in the same local-closed-world, this means the blob was probably a merging of two persons, so color information should not be updated and neither should the size estimation. Also, velocity information is unreliable in a multi-object closed-world, so this is not updated either.

1.3 Problem statement

The goal of this project is to produce features that describe an object as it is observed by a single camera. These features will be used to track an object within a network of cameras. It is possible to extract object features from one image. However, these features become more reliable when they are gathered from multiple images. The problem is to find out what specific features should be used to uniquely identify a single object in a series of image frames, such that it can be tracked. By tracking these objects, an arbitrary number of characteristics can theoretically be robustly determined. The goal can be separated in a number of different tasks:

- Implementation of a real-time blob-detection program
- Finding and implementing single-image feature extraction routines
- Implementing a tracking algorithm
- Producing results for use in the camera-to-camera tracking.

For better understanding it is important to make a distinction between tracking features and features collected by the tracking process. Tracking features are features that are extracted from a single blob and are used to match a blob to an existing track. Collected features are features that are collected over all the blobs that make up a track. Tracking features are necessary to be able to track at all and collected features are the goal of the tracking.

1.4 General layout of the thesis

The rest of this thesis describes a method for tracking moving objects using feature extraction. In chapter two, the method used for detecting foreground objects is described. This chapter also elaborates on two methods used to create consistent labels for fragmented foreground objects. Chapter three describes the feature extraction and track to blob matching. Because we have not tested our tracking software using people, but we have used robots, chapter four explains how these robots were built and programmed. Chapter five describes the

experiments we performed and it discusses the results from these experiments. Chapter six describes the implementation of the software and finally in chapter seven, the conclusions drawn from this project, as well as a discussion and future work is found.

Chapter 2

Object detection

In order to be able to track moving objects, you will first have to detect these objects in the camera images. There are several methods to do so. Examples of which are optic flow, temporal differencing and background subtraction [16]. The choice had previously been made to use the background subtraction method.

2.1 Background subtraction

For basic moving object detection, a background subtraction method by Frank Zwarthoed [16] is used. This method uses multi-dimensional Gaussian kernels to learn a per-pixel background model.

The background model uses normalized RGB (rgI):

$$r = \frac{R}{R+G+B}$$
$$g = \frac{G}{R+G+B} \quad (2.1)$$
$$I = R+G+B$$
$$\mathbf{c} = (r, g)^T . \quad (2.2)$$

The *I* (luminance) is ignored. In this way, shadows are eliminated as foreground objects, because they only differ from the background in color intensity, not in color tone.

The background model is used to make a decision on whether a pixel is part of the foreground or background. Combined with morphological operations (erosion, dilation) clusters of related, connected pixels, known as *blobs*, are formed. These blobs are labeled using a labeling algorithm by Horn[1]. This labeling assigns a number to each foreground pixel. Foreground pixels that are connected (connected pixels are pixels that are directly next to another foreground pixel in any direction) get the same number. The result is that every blob has its own label. Unfortunately, the Zwarthoed algorithm produces a very fragmented object detection, meaning that a real foreground object is represented as a number of blobs, all with their own label. To avoid them being handled as different objects, either morphological growing or label growing (explained in the following sections) is used to 'stick' blobs of fragmented objects together. An additional problem using EM mixture models for background subtraction is that

[1] See [16], chapter 2 for more information.

the kernels tend to 'grow' together. This always happens. The moment at which it becomes so severe that it starts to affect foreground detection depends on the color intensity of a pixel. Dark colored pixels produce more noise then light colored ones. This is caused by the non-linearity of the color system (normalized RGB, rgI).

To solve the problem of kernels growing together, we use the Bhattacharyya distance between two probability density functions (p.d.f.'s), as described in [18]. The Bhattacharyya distance between two p.d.f.'s p_1 and p_2 is defined as:

$$J_B = -\log \int (p_1(x)\, p_2(x))^{1/2} dx\,. \tag{2.3}$$

Since our distributions are normal: $p_1 = N(\mathbf{x};\boldsymbol{\mu}_1;\Sigma_1)$ and $p_2 = N(\mathbf{x};\boldsymbol{\mu}_2;\Sigma_2)$ the equation can be simplified to:

$$J_B = \frac{1}{4}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)^T (\Sigma_1 + \Sigma_2)^{-1} (\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1) + \frac{1}{2}\log\left(\frac{|\Sigma_1 + \Sigma_2|}{2(|\Sigma_1||\Sigma_2|)^{1/2}} \right). \tag{2.4}$$

First, the kernel with the highest prior probability is selected, and then all other kernels are checked for their distance to the kernel with the maximum prior probability. If for some kernel this distance is less than the set threshold, that kernel is not updated. The threshold is currently set to 2.5.
It is quite hard to tweak this threshold in a way that it avoids large amounts of noise and still produces acceptable foreground object detection.

2.2 Morphological growing

Morphological growing is used to combine the information of different blobs usually making up one object. If this method is not used, fragmented objects will be handled as being different foreground objects. Since we have to deal with fragmented objects in almost every image frame, this method is a necessity. After normal morphology (one erosion to remove single-pixel noise, followed by two dilations and one erosion), the binary picture is backed-up. Then, another five dilations are applied to the picture. This 'grown' picture is labeled and then the labels are copied to the backed-up picture. This backed-up picture now contains multiple blobs with the same label, which will be handled as being one foreground object (to extract features from these foreground objects only single-pixel information is used and the relevance of other pixels or their proximity is not taken into account).

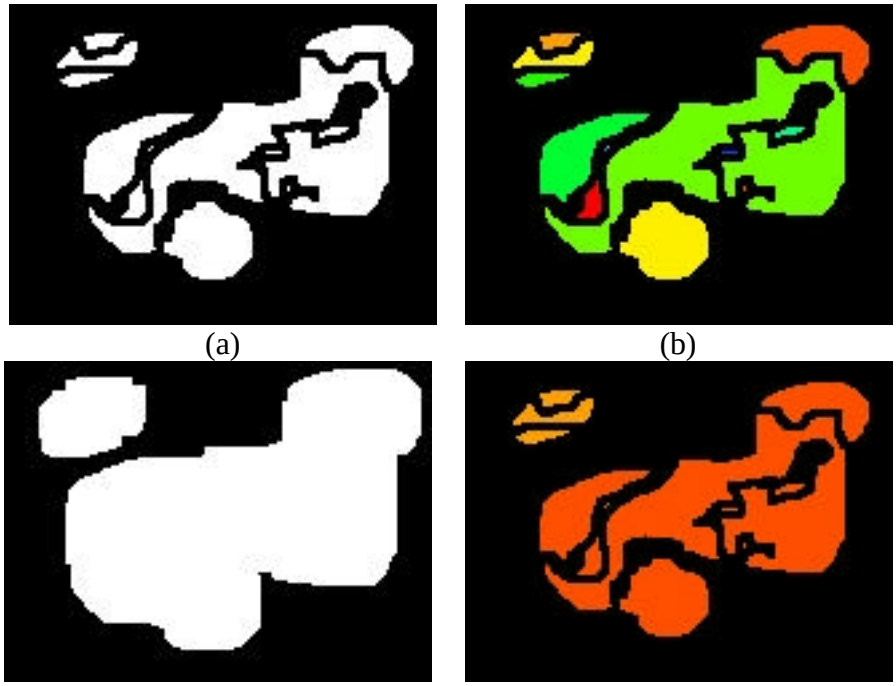


Figure 2.1: *Morphological growing. (a) The original, fragmented collection of blobs. (b) The labeling based on the original blobs. (c) The collection of blobs after morphological growing. (d) The resulting labeling.*

2.3 Label growing

Label growing is a variation on the morphological growing, described in the previous section. The process works as follows:

First, the binary foreground image, still including noise, is dilated twice. This “sticks” clusters together, possibly with noise pixels. This dilated image is then labeled. The labels are copied to the original binary foreground image, just like in the morphological growing algorithm. Then, using an erosion, the noise is removed from a copy of this image. Because the noise was also labeled, certain labels will vanish, while others will be retained. The noise with the same labels that are still present in this copy is brought back (all pixels with surviving labels are restored, including the ones that were removed by the erosion step). A further three dilations, followed by three erosions produces a series of blobs. Because certain parts might be sticking out because of noise being present close to the edge of an object, as is usually the case, another erosion and a dilation clips these parts.

This method produces far more constantly shaped blobs than the morphological growing method, because of the assumption that not all noise is random. This assumption means that noise, surrounded by, or very close to, an object usually is part of that object and the noise is caused by a failure of the background segmentation method, rather than a random variance in the image data.

Chapter 3

Tracking objects using feature extraction and probability matching

When an object passes a camera, we want to be able to extract its features in a robust way. One possibility of doing so is to track the object through consecutive image frames. A track is a collection of blob positions (in the image domain), along with other features of a blob of which is assumed they are produced by one single object. For tracking objects, it is necessary to have some way of comparing information stored in a track with the information of different blobs in the current image frame. Features provide this means of comparing. Based on features, probabilities of particular matches can be calculated, after which an assignment of blobs to tracks can be done. It is quite common to use Kalman filters for tracking moving objects, however since Kalman filtering assumes the motion can be described by some prior model, there is a problem. Human motion is difficult to model. People can suddenly stop and turn very sharp angles. Extreme speed changes are also possible (think about someone running into the image and suddenly stopping, turning around and running back). Because of this problem, we decided to track the blobs differently. This is described in the next sections.

3.1 Feature extraction

Features are numerical properties of pixels making up a blob, based on the pixel or spatial information that can be extracted from the blob. A number of features are extracted from the foreground blobs. The specific features we have selected are easily extracted from the image without the need of any kind of pre-processing of the image data. The features used are:

- Location of the center pixel of a blob, derived from the bounding rectangle.
- Average color of the blob, using normalized RGB (rgI), where the intensity channel I is ignored (further referred to as ‘rg’). The average color is computed for the red and green channel separately.

In a later stage, we also consider:

- Size of the blob relative to the image size.
- Average center location of the blob. The mean of all the coordinates of the pixels of a blob is computed. This is also known as the *center of gravity* of a blob.
- Covariance matrix of the pixel coordinates within the blob. This provides information about the pixel distribution of the blob.

The following sections describe the ways these features are extracted.

3.1.1 Center pixel location

Calculating the center pixel location is done by first establishing the dimensions of the bounding rectangle and calculating the center of this bounding rectangle[1]. The algorithm runs through the image top-down, left-to-right. The location of the first pixel with the given label is the initial top-left corner *and* the initial bottom-right corner of the bounding rectangle. When subsequent pixels with the same label are found outside the current bounding rectangle, the dimensions of the bounding rectangle are updated to include these pixels. Once the entire image is scanned and the dimensions and location of the bounding box are known, the center of the box is computed and rounded to the nearest pixel coordinates.

$$\mathbf{x}_c = \mathbf{x}_i + \frac{(\mathbf{x}_j - \mathbf{x}_i)}{2}, \tag{3.1}$$

where $\mathbf{x}_c = (x_{cx}, x_{cy})$ the pixel location of the center of the bounding rectangle, $\mathbf{x}_i$ is the pixel location of the top-left and $\mathbf{x}_j$ is the pixel location of the bottom-right corner of the bounding rectangle.

3.1.2 Average color

The mean color of a blob with label *l* is calculated using the following equation:

$$\boldsymbol{\mu}_c(l) = \frac{1}{|B_l|} \sum_{i \in B_l} \mathbf{c}_i, \tag{3.2}$$

where B_l is the set of pixels with label *l* in the image and $\mathbf{c}_i$ is the color vector for pixel *i*, as described in equation (2.2).

[1] See Appendix C for the algorithm

3.1.3 Blob size; zero order moment

Blob size *s* of a blob with label *l* is calculated in the following way:

$$s(l) = \frac{|B_l|}{N}, \tag{3.3}$$

where *N* is the total number of pixels in the image.

3.1.4 Center of gravity; first order moment

The average pixel center is calculated for the center of gravity feature using the following equation:

$$\boldsymbol{\mu}_x = \frac{1}{|B_l|}\sum_{i \in B_l} \mathbf{x}_i \,. \tag{3.4}$$

Here $\mathbf{x}_i$ is the location vector of pixel *i*.

The advantage of using this over the center pixel location feature is that when using the center of gravity feature, the computed location of the center pixel of a blob is less influenced by the dimensions of the blob, but more by the shape of it. When an object is stationary, the dimensions of its accompanying blob might still change radically, but the shape of the blob is more constant. This results in a more constant blob location computation.

3.1.5 Covariance matrix of pixel coordinates; second order moments

A covariance matrix is calculated for all pixels of a blob using the center of gravity of section 3.1.4 as the mean $\boldsymbol{\mu}_x$:

$$\Sigma(l) = \frac{1}{|B_l|-1}\sum_{i \in B_l}\begin{bmatrix} (x_{ix}-\mu_x)^2 & (x_{ix}-\mu_x)(x_{iy}-\mu_y) \\ (x_{iy}-\mu_y)(x_{ix}-\mu_x) & (x_{iy}-\mu_y)^2 \end{bmatrix}, \tag{3.5}$$

where x_{ix} is the x-coordinate of the location vector **x** of pixel *i* and μ_x is the x-coordinate of the average pixel center.

3.2 Calculating match probabilities

We are assigning blobs to tracks based on match probabilities. In order to calculate these match probabilities, distance measures are computed to be able to compare tracks to blobs. Converting distances into probabilities is done using a Gaussian probability density function. Because we assume that the features are independent, we can also assume that all the resulting probabilities are independent. We can combine the different probabilities by multiplication:

$$P(match_{ij} \mid i, j) = \prod_n P(match_{ij} \mid i, j, d_n), \tag{3.6}$$

where $match_{ij}$ is a match between track *i* and blob *j* and d_n is the distance for feature *n*. The probability of a match between track *i* and blob *j*, given track *i* and blob *j* equals the product of all probabilities of a match between track *i* and blob *j*, given track *i*, blob *j* and distance d_n.

The combined match probabilities are used to determine if one track and one blob are a match or not. The following distance measures are used.

3.2.1 Positional distance

Positional distance is the distance between the expected location of the center point of a blob and its actual location. To calculate the expected location of a new blob, we have:

$$\begin{aligned} \mathbf{?}(t) &= \frac{\mathbf{x}(t) - \mathbf{x}(t - \Delta t)}{\Delta t}, \\ \boldsymbol{\mu}(t + \Delta t') &= \mathbf{x}(t) + \mathbf{v}(t) \cdot \Delta t' \end{aligned} \tag{3.7}$$

where $\boldsymbol{\mu}(t + \Delta t)$ is the predicted position at timestamp $t + \Delta t$ and $\mathbf{v}(t)$ is the speed vector in pixels per millisecond between the last two points of the track. Δt is the amount of time between two frames. Note that this need not be constant. Now, a two-dimensional variant of the general Gaussian equation is used to calculate the match probability:

$$p(x) = \frac{1}{2\pi \left|\sum\right|^{\frac{1}{2}}} \exp\left(-\frac{1}{2} \left(\mathbf{x} - \boldsymbol{\mu}\right)^T \sum^{-1} \left(\mathbf{x} - \boldsymbol{\mu}\right) \right), \tag{3.8}$$

where $\mathbf{x}$ is the actual location of a blob, $\boldsymbol{\mu}$ the expected location and $\sum$ a covariance matrix. A diagonal covariance matrix $\sum$ is used:

$$\sum = \begin{pmatrix} \left(\frac{v(t)}{2}\right)^2 & 0 \\ 0 & \left(\frac{v(t)}{2}\right)^2 \end{pmatrix}. \tag{3.9}$$

3.2.2 Color distance

The color of a track is defined as the incremental mean of the mean colors of the blobs assigned to the track:

$$\boldsymbol{\mu}_k(t) = \frac{(N-1)\cdot\boldsymbol{\mu}_k(t-1)+\boldsymbol{\mu}_c}{N}, \tag{3.10}$$

where $\boldsymbol{\mu}_k(t)$ is the mean color of track *k* at time *t*, *N* is the length of the track and $\boldsymbol{\mu}_c$ is the mean color of the blob being assigned. Also, for the calculation of the color distance probability, we use equation (3.8). However, in this case the covariance matrix is not arbitrary. We are using an incremental update function for the covariance matrix:

$$\begin{aligned}\sigma_{ij}^2(t+1) &= \frac{1}{t}\left[\sum_k^{t+1}\Big(\big(x_i(k)-\mu_i(t+1)\big)\big(x_j(k)-\mu_j(t+1)\big)\Big)\right]\approx \\ &\approx \frac{1}{t}\Big((t-1)\sigma_{ij}^2(t)+\big(x_i(t+1)-\mu_i(t+1)\big)\big(x_j(t+1)-\mu_j(t+1)\big)\Big)\end{aligned}. \tag{3.11}$$

In this function, σ_{ij}^2 is the entry in the covariance matrix at row *i*, column *j*.

With a dimensionality of 2 (two color channels), the equation for calculating the color match probability is:

$$p(x) = \frac{1}{(2\pi)\left|\sum\right|^{\frac{1}{2}}}\exp\left(-\frac{1}{2}\left(\boldsymbol{\mu}_{c,blob}-\boldsymbol{\mu}_c(k)\right)^T\Sigma^{-1}\left(\boldsymbol{\mu}_{c,blob}-\boldsymbol{\mu}_c(k)\right)\right), \tag{3.12}$$

with Σ as expressed in equation (3.11).

3.2.3 Size distance

The size of a track is the expected size of the next blob assigned to the track. The expected size is computed as follows:

$$\begin{aligned} d(t) &= \frac{s(t)-s(t-\Delta t)}{\Delta t}, \\ \mu_s(t+\Delta t') &= s(t)+d(t)\cdot\Delta t' \end{aligned} \tag{3.13}$$

where *d(t)* is the difference in size between the last blob assigned to the track and the one before that. Since size is a scalar, we compute the match probability thus:

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(\frac{-(s-\mu_s)^2}{2\sigma^2}\right), \tag{3.14}$$

where $\sigma^2 = \left(\frac{d(t)}{2}\right)^2$, similar to the variance for the position.

Now that we have the different match probabilities, we can continue with the actual assignment process.

3.3 Assignment

Assignment is the linking of blobs to existing tracks. Now, we are possibly dealing with multiple blobs and multiple tracks. Because it is possible that straightforward matching (assigning the blob with the highest combined match probability to a track) does not work, for example when for two tracks the same blob has the highest match probability, the necessity arose to implement more elaborate assignment methods. Because all tracks are independent, we can simply calculate the overall match probability by taking the product of the appropriate combined match probabilities, as defined in section 3.2. Instead of looking at individual track-to-blob matches, we have to look at the overall match probability, because the matching of one blob to a track can affect the effectiveness of all other matches.

3.3.1 Matching

Our assignment problem can be seen as a special case of the weighted matching problem [8]. This involves finding the path with maximum weights in a bipartite graph $G = (V, U, E)$ (see figure 3.3). All tracks are nodes in *V*, all blobs are nodes in *U*. All nodes in *V* are connected to all nodes in *U*: $\forall i \forall j [v_i, u_j] \in E$. This will later be referred to as a *fully connected bipartite graph*. The weight of an edge w_{ij} is the probability for that match. A match set *M* on a graph *G* is a subset of edges in *E*, such that a node in *V* is connected to at most one node in *U*. There are a number of methods for solving this problem. For a match set M on graph G, all of the methods have the same goal, which is to maximize the assignment score, defined as $\sum w_{ij}$ for all $[v_i, u_j] \in M$. Because we are using probabilities instead of scores, we want to maximize the product of the assignment probabilities. This maximization guarantees a high overall match probability, which means the individual matches are all as high as they can be (with respect to the other matches). To realize this product maximization, we use log(P) instead of P, because maximizing $\sum \log(P)$ is the same as maximizing $\prod P$. This section explains how the different methods work and why one is more efficient than another.

Exhaustive matching

Exhaustive matching is by far the easiest way of solving the match problem. By enumerating all possible match sets and calculating their summed score, the best

match is found. It is easy to see that the order of this method is $O(n!)$. This takes a lot of time and since we want our algorithm to be able to run real-time (read: at least five frames per second on the system outlined in chapter 6), we clearly need a more efficient algorithm.

Assignment by Rangarajan and Shah

The assignment method by Rangarajan and Shah [10] is an approximate method and substantially more efficient than the exhaustive matching. Its order is $O(n^2)$. A match probability matrix is constructed. The row indexes are tracks (nodes *V* in the bipartite graph) and the column indexes are the blobs (nodes *U*). The matrix values are the match probabilities for the different track to blob matches. In every row (for every track), the highest score is selected. For all of these highest scores, the 'missed score' is calculated. The missed score is the sum of all scores that cannot be assigned by selecting this blob. The track with the lowest missed score is then selected to be matched with its highest scoring blob.

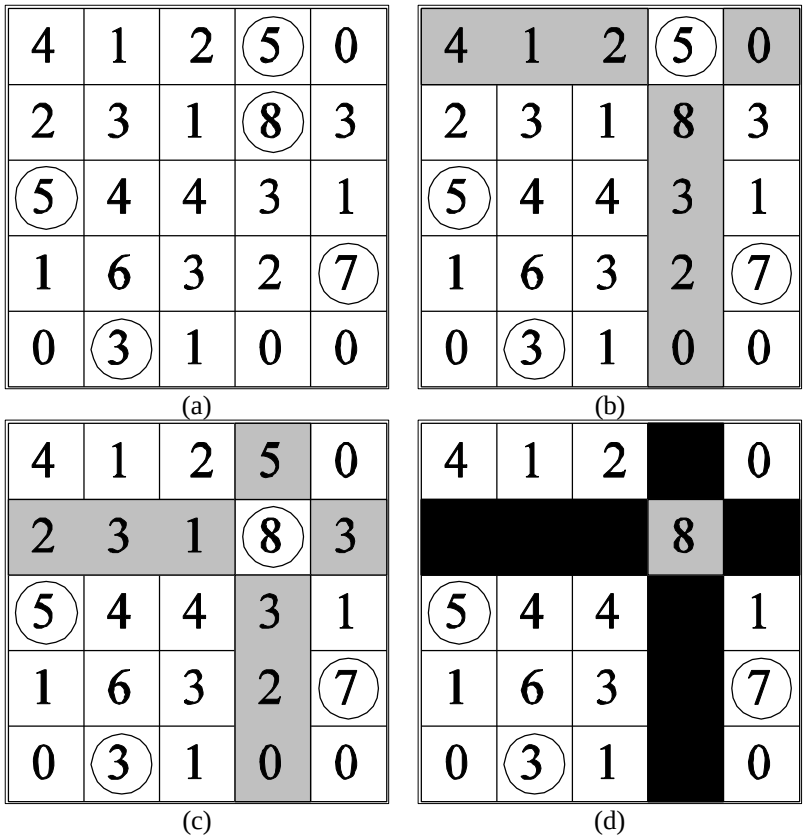


Figure 3.1: *Rangarajan assignment. (a) Score matrix with maximum scores per track circled. (b) Missed score for track 1 grayed. (c) Missed score for track 2 grayed. (d) After selecting blob 4 as a match for track 2, row 2 and column 4 are masked.*

By doing this, the amount of good alternatives (alternatives that would be considered useful in the assignment context) lost by this assignment is minimized. The respective row and column are masked and the process is restarted. This continues until all tracks are matched. Figure 3.1 illustrates the process.

Hungarian assignment algorithm

The Hungarian algorithm is the established method for graph matching problems. It is a rather complex algorithm to create an optimal match. In our problem we are dealing with a bipartite graph, which somewhat simplifies the problem. Still, its order is somewhat higher than that of the Rangarajan method: $O(n^3)$. If the number of tracks in the bipartite graph is not equal to the number of blobs, $|V| \neq |U|$, we need to add 'dummy' tracks or blobs to equalize the graph. To make sure they do not affect the matching process, all the edges these 'dummy' nodes are connected to have a match probability (and thus weight) of zero. The method as described in [17] works as follows:

A label *l(v)* is assigned to each node $v \in V$ and $u \in U$, such that $l(v) + l(u) \geq w(v,u)$. Such a labeling is called *feasible*. A feasible labeling always exists, namely $l(u) = 0$ and $l(v) = \max_{u \in U}\{w(v,u)\}$. Using this particular labeling, an equality subgraph $G_{=} = (V, U, E_{=})$ is constructed. The edges $E_{=}$ are the edges of E for which $l(v) + l(u) = w(v,u)$. Now the problem of finding an optimal match in G has been reduced to finding a perfect match in $G_{=}$, where every node in *V* is connected to a node in *U*. This is because a perfect matching in the equality subgraph corresponds to the optimal match in the original graph.

The algorithm starts with an arbitrary matching *M* in $G_{=}$. A matching is a set of edges that connect one node in *V* to one node in *U*. Nodes in *U* cannot be matched to more than one node in *V*. If this matching is perfect, the algorithm is completed. If not, starting at a node in *V* and not in *M*, an alternating path is constructed. An alternating path is a series of nodes, such that the edge connecting two subsequent nodes is alternating in and not in *M*. See figure 3.4 for an example. If the alternating path ends at a node in *U*, the matching can be augmented (changed). Then the algorithm restarts. If the alternating path ends at a node in *V*, it means the matching cannot be augmented and edges need to be added. Note that not all edges of *E* are in $E_{=}$. The edges are added in the following way:

$$\alpha_l = \min_{\substack{x \in S \\ y \in T}} \{l(x) + l(y) - w(x,y)\}, \tag{3.15}$$

where *S* is the set of nodes of *V* in the alternating path and *T* is the set of nodes of *U* in the alternating path. The edge causing α_l is added to the equality subgraph and the labels are updated:

$$\overline{l}(v) = \begin{cases} l(v) - \alpha_l \text{ if } v \in S \\ l(v) + \alpha_l \text{ if } v \in T \\ l(v) \quad \text{otherwise} \end{cases} . \tag{3.16}$$

A new matching is created. And the algorithm restarts.

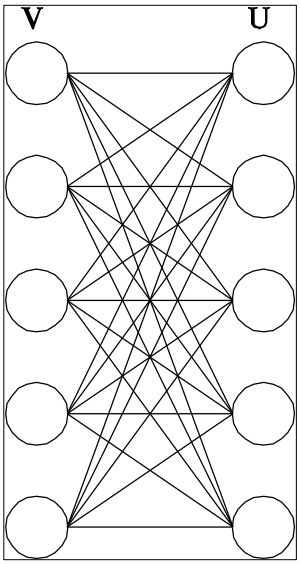
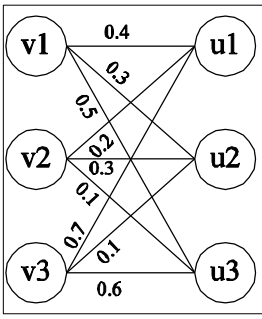


Figure 3.2: *Bipartite graph*



	u1	*u2*	*u3*
v1	0.4	0.3	0.5
v2	0.2	0.3	0.1
v3	0.7	0.1	0.6

Figure 3.3: *Translation from a weighted graph to a matrix*

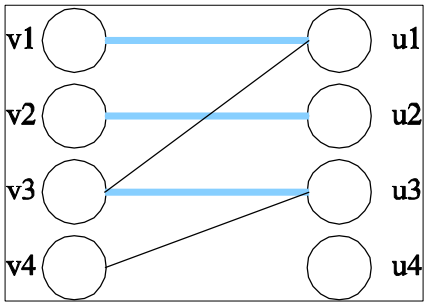


Figure 3.4: *Alternating path example. The current matching is denoted in blue. Starting at v4, a correct alternating path would be: {v4, u3, v3, u1, v1}. Since this alternating path does not end on a node of U, the current matching cannot be augmented.*

Because of the inefficiency of the exhaustive matching, we have decided not to use it. Both the Rangarajan and the Hungarian matching will be used in the assignment process.

3.3.2 Track extension

Track extension is the process of actually extending the existing tracks with new blobs, based on the matches returned by one of the match algorithms and creating new tracks if required. The match algorithm always returns a complete match, which means that all tracks are matched to a blob. A threshold is used to determine if a match probability is high enough to be accepted. Any match whose probability is below the threshold is discarded, leaving the track unmatched and a candidate for matching in a later stage. It is also possible that there are not enough blobs to match all tracks, or that there are more blobs then there are available tracks. In these cases, tracks need to be created or ended. Since merging of two objects is possible, assigning two tracks to the same blob is also considered. The extension steps are:

1. *Known track to unoccupied blob*. In this stage, blobs are assigned to their most likely track, based on their match probability. If there are multiple objects with the same blob as their candidate, it is needed to find out what the most profitable assignment would be. If you have a large number of objects, this can be a very time-consuming job. This is why we use the assignment systems mentioned above to solve this problem.
2. *Known track to occupied blob*. If it is not possible to assign a blob to a track in step 1, then try to assign a blob that was previously assigned to a different track. This should cover object merging. If the final assignment of this track results from this step, size and color information should not be updated.
3. *Blob without a track: create new track*. If there are still unassigned blobs, create new tracks for them.
4. *Track without a blob: remove track*. When a track didn't get a blob assigned in the previous steps, its corresponding blob has probably moved out of the image frame, so the object should be removed (since this can

also be caused by (partial) occlusion, the actual removal of the object is delayed for a number of frames).

3.4 Initialization

Our approach requires the track features to be initialized. Mean and standard deviation are initialized when the track is first created. Mean color is set to the mean color of the blob that created the track, the standard deviation is initialized high, since you don't know anything about it. Also, track size is set to the size of the same blob, the size difference is set to an initial value of 10% of the blob size.

3.5 Conclusions

The conclusions that can be drawn from this chapter are that there are multiple features to choose from, as well as multiple matching algorithms. The question now is: which combination of matching algorithms and features works best. Therefore the experiments will first look at the difference in performance between the Hungarian matching and the matching by Rangarajan and Shah and secondly take a look at the sensitivity of the tracking algorithm to the specific set of features used. Since we are using robots, we will first explain their construction and programming.

Chapter 4

The robots

To find out if the tracking software performs in an adequate way, tests have to be performed. These tests are set up to proof the different part of the tracking and feature acquisition. Different behaviors (for example constant and erratic movements) should not have a negative effect on the tracking performance. For these tests, we are using Lego Mindstorms robots. These robots are built using Lego and the RCX brick. The RCX brick is the Mindstorms computer. It has limited memory capacity and computing power, three connections for sensors and another three for actuators such as motors. For communication with a computer or another RCX brick, an infrared port is present (communication can be seen as a program transfer when the brick is not running, or the transmission and reception of data during the run of a program). After writing the program that describes the actions and interactions of the robot on a PC, it is downloaded using the supplied infrared interface. The program can then be run on the robot, which performs the tasks as programmed, optionally based on sensory input. The user has no control over the behavior of the robot once the program is downloaded.

The reason for choosing the Mindstorms robots is that their appearance can easily be altered as well as their behavior (behavior is defined as a series of motions). In order to get reproducible results, it is desirable to have objects with predictable behavior (human behavior is not considered predictable).

This section describes first the software used to program the robots. In section 2 the program itself is explained. Section 3 covers testing results.

4.1 Software

The software being used is Not Quite C (NQC). This is a near-C environment in which Lego Mindstorms robots can be programmed with considerable ease, albeit somewhat limited. Before programs written in NQC can be downloaded to the robot, the *firmware* has to be downloaded. Firmware is the actual operating system of the Lego robot. There are two versions of the firmware, RCX and RCX2. Note that the name of the version of the firmware is identical to the actual name of the Mindstorms brick (RCX).We are using the RCX2 firmware, because although it is slightly bigger then the original RCX firmware, leaving less space or user programs, it has largely increased functionality. In fact, on the original RCX firmware, most functions implemented in NQC are unavailable. The size of programs written in NQC is limited, because of the limited amount of memory in the RCX brick, the Mindstorms computer. There is a total of 32KB of memory in the brick, of which the firmware usually uses about 16KB, leaving a maximum

program size of 16KB. Usually, NQC programs are quite short, so they stay well within this limit.

There are other software platforms for the Mindstorms brick (RCX). Examples are:

- LeJOS, a Java implementation for the RCX. This makes it possible to write almost fully functional Java programs for the RCX. The disadvantage of this software is that Java programs tend to be much larger and much more complicated then NQC programs.
- LegOS, a Linux kernel running on the RCX, implementing full C and C++ compatibility. This was our first choice, but unfortunately we couldn't get the Linux-RCX cross-compiler working, so we abandoned this.

There are dozens more that are all more or less based on the three systems mentioned above. More information about NQC, LeJOS and LegOS can be found in their respective manuals and on the Internet.

4.2 Building the robots

To get devices that met our requirements (adaptable in appearance and behavior), we decided to build two physically different robots, running the same program. Because of these physical differences, their actual behaviors would be different as well. Another requirement was that they were capable of traversing an enclosed area without getting stuck in corners or against walls (for the testing of the software the robots had to be in the view area of the camera as the only moving objects while image sequences were recorded[1], so human intervention was not an option). This meant that collision detection had to be incorporated into the design.

The two Lego Mindstorms robots were built using only parts found in the Lego Mindstorms Invention Kit 2.0. This kit contains, apart from building parts, the RCX brick and an infrared tower for sending the programs to the brick. We based our designs on the Rover example from the building manual contained in the kit. The basic design involved a rover-like robot, consisting of two engines driving the wheels independently, a light sensor (pointed down and close to the surface) for measuring light and color changes (the light sensor in the kit can be used for both light and color change measurements) and a two-sensor bumper for collision detection on the front. The two sensors make it possible to distinct between a collision on the left and one on the right of the robot. One of the robots has caterpillar tracks and a geared transmission. This robot is quite slow, but is also very precise in its color measurements. The other one has only two wheels on the front, driven by the two motors and a skid pad on the back. The wheels are connected 1:1 to the engines, making this robot very fast and agile, but also very imprecise in its measurements (because of the high speed with which it moves). These differences make the two robots interact with the environment in completely different ways. The slow robot will never miss a crucial measurement,

[1] See chapter 5, section 2 for information about the recording of image sequences (videos).

whereas the faster robot will usually overshoot the measurement point and then try to solve a problem that is not at the position the robot is at. For convenience, both robots use the same programs.

4.2.1 Programming the robots

To test tracking accuracy, we need to test different behaviors. Speeds should be variable, as well as appearance and directional and speed changes. To accomplish this, we have two programs for the robots. One with which the robots move in a somewhat straight line (the actual line described is dependent on obstacles that have to be avoided and the torque the two motors provide. If this is not equal, then the line described will not be straight). Using this program, the speed at which the robot moves is quite constant, which makes tracking easy, since position and speed measurements can be used to predict the next likely position of the robot accurately. The second program moves the robots in a ‘straight’ line (as explained above) for a certain time (which can be a randomly determined number), then turns for a certain time and moves on ‘straight’ again. This second program is described in closer detail below:

We programmed the robots to be able to move over a blue-and-white tiled floor.
We wanted them to move in a way that simulates motion of humans in an environment where the direction of movement is not predetermined by the layout of the environment. Concretely, this means the robots’ motion is a set of straight-line movements and rotations of non-predetermined length and angle. This way of motion will further be referred to as random motion. Because the only way the robots can interact with their environment is to sense it, we have to use the environment to guide the robots. We decided to use the tiling on the floor for this purpose. Because the entire floor has the same white-and-blue tiles, using the edges between two differently colored tiles as sensory input to the program running on the robots does not make their behavior predetermined by the layout of the environment. The robots will behave in exactly the same way, no matter where they are placed on the floor, as long as the angle to the edges between the tiles is the same. The way in which the environment around the robots is used to guide them is:

1. Check sensor values. If one of the bumper sensors is on, handle the appropriate object avoidance. If the light sensor changed considerably (from light to dark or from dark to light), raise the number of edges (changes from a white to a blue tile or the other way around) detected by one.
2. If the number of edges detected is below the threshold (determined randomly), set the engines to full power forward. If the number of edges detected is at or above the threshold, set the engine's direction so that the robot turns either left or right (determined randomly) and leave them like that for a random period of time, reset the edge counter and determine a new threshold randomly. When the turn time has expired, move forward

again (note that during the time the robot turns, collision detection does not work).

Of course, the edge count threshold must have an initial value. This is also determined randomly. The program contains minimal and maximal values for this threshold.

To have accurate edge detection, appropriate light-sensor values have to be recorded for ‘light’ and ‘dark’. This can be done automatically in the following way: the robot is positioned at right angles to the edges between the tiles. The first two edges are measured and used to find minimal (dark) and maximal (light) values for the light sensor. Every time the sensor is polled, the difference to these dark and light thresholds is calculated. The smallest difference tells you whether the robot is on a light or a dark tile. The result of the last polling is retained and compared to the new one. If they differ (example: last polling was 'light', the new polling is 'dark'), an edge is detected and the edge counter is raised.

It is much easier to hard-code these sensor values into the program. For this purpose a small program is written that constantly reads the raw sensor value (the unscaled, direct sensor output) and displays this on the display on the RCX brick. When the robot is placed on a tile while this program runs, the sensor value returned by the light sensor can be read directly from the display. This value can be used in the actual robot program.

It is also possible to use constant calibration. That is, the polled light sensor value is first classified as either 'light' or 'dark' and then the closest threshold is updated by this value. A possible way of doing this is to average all measurements classified for this threshold. This could also be extended to a time-deprecating average, where influence of sensor pollings deprecates in time (of course, the first two measurements are the basis of the light-dark thresholds and when these don't differ a lot, for example because the second measurement was caused by a stain on the floor and not by an edge between tiles, this method will make sure that the thresholds will be somewhat reliable after some time, whereas the original calibration method would find many non-existent edges).

4.3 Testing

The robots have been tested on the aforementioned tiled floor. The two-step calibration method fails occasionally, causing the problem mentioned in the last section (detection of many non-existent edges) to occur. The calibration step is highly sensitive to a pre-determined value telling the robot how far sensor values should be separated to be considered an edge. The time-deprecating average method has not been tested, because it is much too elaborate for our purpose. To avoid the calibration problems, we have decided to use the hard-code option (during the recording of the testing videos, it is only needed to change these values once).

Chapter 5

Experiments

Experiments are used to test the tracking software as described in chapter 3. The tracking will be tested using different assignment methods, tracking features and robot behaviors. In this chapter the test setup will be explained first, after which the different tests are explained, as well as the results of the individual tests.

5.1 Setup

This section covers the design of the test environment as well as the equipment used for the experiments.

5.1.1 The room

The room we used as our test area has a blue and white tiled floor. A part of this floor has been fenced off and a white-ish, uniformly colored back wall is placed. Lighting conditions are kept constant during the experiments, to get an optimal background estimation model. People are not visible during the experiments. Since not all of the test area is within the view of the camera, it is possible to handle the robots without appearing before the camera.

5.1.2 The cameras

The cameras are situated in such a way that they look down on the testing area, since in real-life situations, surveillance cameras are often mounted in such a way that they look down onto the area surveilled. The cameras are so-called pan-tilt-zoom cameras, meaning that the pan, tilt and zoom of the camera can be controlled remotely, either by an infrared remote control, or a serial interface present on the back of the camera. This serial interface uses the VISCA protocol, which is a communication protocol for cameras developed by Sony. The VISCA interface enables a program to change all settings of the camera (not only pan, tilt and zoom, but also focus (auto focus in three modes or manual) and a number of more advanced functions. One of these more advanced functions is white balancing. White balancing is a process with which the total luminance (the sum of the red, green and blue values of the pixels) of all images is kept on or around a pre-defined value by means of the brightness of the image (often this value is

hard-coded into the camera). White balancing constantly updates the ambient luminance of the image. Sometimes this is done to the extreme, which causes areas to change color (light areas become white or dark areas turn black). This is a problem for the background estimation. The background model becomes very imprecise. For this reason, continuous white balancing (which is the default setting of the camera) is disabled and instead a one-push white balancing is used. One-push white balancing is one white balancing operation that is performed when the software requests it. We don't want the cameras to move while the software is running, so the software uses a pre-programmed pan-tilt-zoom situation in which the camera is positioned at the start of the program.

5.2 Testing the software

Software tests consist of runs in the lab, in which different situations are simulated using the two Lego Mindstorms robots. For the tests, pre-recorded videos are used to ensure reproducibility. These videos are recorded using a realistic simulated frame-rate of 10 frames per second. (this is the actual speed the software runs at on our test machine (see chapter 6) using the online system). Each video consists of 6000 frames, which is a total length of 10 minutes. The first 3000 frames are pure background, to make sure the background subtraction system has time to form an accurate model. The remaining 3000 frames contain repeated passes of one or two robots through the image. In one video, we use only a single robot, which moves in and out of the image repeatedly. In another we use two robots in the same fashion. The videos are presented to the tracking program using a ‘virtual frame grabber’, which simulates an actual frame grabber, but instead of reading from a camera device, this frame grabber reads from a file. This ensures absolute transparency, which allows for easy switching between online and offline use of the tracker.

5.2.1 Evaluation

The evaluation method analyzes the performance of our tracking system by first establishing the *ground truth*. We investigated the video sequences manually and recorded the begin- and endframes of a *true track*, as well as the number of tracks in the video. Each of the true tracks gets a label.

After this step, the video of tracking results of our algorithm is analyzed. This video shows tracks found by the tracking algorithm. These tracks will be referred to as *detected tracks*. Each frame in which one or more detected tracks are present, the true track they follow is determined by comparing the distance in pixels of the location of a new track point to the location of the different objects in the frame. The new track point receives the label of the true track that is closest to this point (each moving object represents a single true track). Note that within one detected track, different points can have different labels.

At the end of the result analysis, we have a set of detected tracks with their labeling. For each detected track, we can now determine which label occurs most frequently. By counting the number of occurrences of this label and dividing this by the total length of the track, we get the *label consistency*, the fraction of the feature information formed by the track that is reliable:

$$C(k) = \frac{maj(k)}{N_k}, \tag{5.1}$$

where *k* is the detected track for which the label consistency is calculated, *maj(k)* is the number of occurrences of the most frequently occurring label in track *k* and N_k is the total number of points in track *k*.

The second part of the evaluation compares the number of detected tracks to the number of true tracks. It is possible that the tracking algorithm ends a track prematurely or starts a track when no true track starts. This can be caused by an inaccuracy in the background estimation or by an error in the tracking in an earlier stage, such that the feature information used to match a track to a blob is corrupted too much to allow a positive match. This causes more tracks to be found then there are true tracks. For the same reasons it is also possible that a detected track connects two true tracks. In this case less tracks are found. The *track count error* is determined in the following way:

$$\begin{aligned} \Delta_K &= \left| K_d - K_t \right| \\ \varepsilon &= \frac{\Delta_K}{K_t} \end{aligned}, \tag{5.2}$$

where K_d is the total number of detected tracks in a video, K_t is the number of true tracks in a video.

Combination

The label consistency of an entire test can be found by averaging over all label consistencies:

$$\hat{C} = \frac{1}{K_d} \sum_{k=1}^{K_d} C(k) . \tag{5.3}$$

Now there are two measures for tracking quality: label consistency and number of detected tracks. Usually, optimizing the tracking performance is a trade-off between these two measures: when label consistency is optimized, the number of tracks usually suffers, because starting a new track is preferred over corrupting the label consistency. On the other hand, when the number of tracks is optimized, this could easily result in faulty labeling, simply to keep the track going.

5.2.2 The different tests

Two tests were carried out. The first test was aimed at determining which assignment method produced the best results. The method that scored best in this test was then used in the second test to establish which set of features could best be used to track the Lego robots. At first, the tracking is tested using only a single robot, moving in a straight line (as described in chapter 4) through the image. This is tracked using both the Rangarajan and Hungarian assignment methods, as described in chapter 3. Using each of these methods, the two available blob construction algorithms, described in chapter 2, (morphological growing and label growing) are also evaluated. The tracking features used in this stage are *expected position* and *average color*, as described in chapter 3. The same setup is used with three other videos. These videos contain one robot moving randomly, two robots moving in a straight line and two robots moving randomly.

When these tracking results are analyzed, the best performing system (either Rangarajan or Hungarian assignment) is selected for a second test. Here, the tracking features are tested to see which features can best be used to track moving objects. The remaining three features from chapter 3 are tested in this stage.

5.2.3 Testing assignment methods

In this stage, the two assignment methods (Rangarajan and Hungarian) are tested to determine which performs best in this particular task. This is combined with a test of the two blob construction algorithms. Here, the method used in the next test is also selected. Using four input videos, two assignment methods and two different blob construction algorithms, brings the total of videos to be evaluated at sixteen.

Tracking results

In the following tables, tracking results will be shown. Each table shows the results of tracking using one of the videos. A set of two numbers represents the {average label consistency / average track count error} of that particular algorithm.

Table 5.1: A single robot moving straight

	Morphological growing	**Label growing**
Rangarajan assignment	1.00 / 0.200	1.00 / 0.200
Hungarian assignment	1.00 / 0.200	1.00 / 0.200

Table 5.2: A single robot moving randomly

	Morphological growing	**Label growing**
Rangarajan assignment	1.00 / 1.500	1.00 / 1.625
Hungarian assignment	1.00 / 1.500	1.00 / 1.625

Table 5.3: Two robots moving straight

	Morphological growing	**Label growing**
Rangarajan assignment	0.965 / 0.077	0.932 / 0.154
Hungarian assignment	0.965 / 0.077	0.932 / 0.154

Table 5.4: Two robots moving randomly

	Morphological growing	**Label growing**
Rangarajan assignment	0.889 / 0.579	0.898 / 0.263
Hungarian assignment	0.929 / 0.368	0.902 / 0.211

5.2.4 Discussion of the first test results

The test results show, that for situations in which one single object is tracked, the two methods produce the same result. In the case of two robots, the different blob construction algorithms can produce slightly different blobs that have different center points and thus produce different track trajectories. In some cases this causes tracks to start and end at different frames and in different situations. Except for the last video: in this particular case, the robots moved very close to one another and actually touched each other frequently. This has somewhat upset the tracking algorithms, since the number of detected foreground objects did not always match the number of tracks in progress. It seems that only the data from the tests where two robots were used, produces any meaningful results for deciding which method produces the best results. Deciding which method is best, depends on what it is to be used for. In our case, the tracking results are used to identify an object on a different camera. Therefore, high label consistency is required (the more consistent track labeling is, the more of the feature information gathered in the track is from one single object). As mentioned before, feature

information is more reliable when it is gathered over a longer period of time, so the best method should also produce a number of tracks that is close to the true number of tracks. From the tables in the previous section, it can be seen that the Hungarian assignment algorithm has the highest label consistency. Judging from the difference in the track count error (where there are differences: tests 3 and 4), the morphological growing blob construction algorithm performs slightly better (in test 3 it produces half as many extra tracks as the label growing does, while the fourth test shows that morphological growing only produces 1.7 times as many extra tracks as label growing, which does not compensate for the difference in test 3). The combination of Hungarian assignment and morphological growing blob construction should therefore be theoretically most suitable for the task of tracking objects in order to gather information for their subsequent identification.

5.2.5 Testing feature sets

Here, the set of features best used for tracking moving objects is determined. For this test, the video which was also used for the fourth experiment of the first test (two robots moving randomly) is used again, since it is the video with the worst results and indeed the only video that shows different results for each algorithm combination. The previous stage has shown that expected position and average color describe an object fairly well and make it possible to track it. However, there are still a few points at which the tracking is far from optimal, such as the number of tracks being reported by the tracking algorithm. Using a different feature set may improve the tracking. In this stage, the basic feature set of expected position and average color is compared to all other possible combinations of features: expected position, average color, size, center of gravity and pixel covariance. For each of these combinations, the improvement or deterioration of the tracking performance is determined.

The table on the next page shows the results of the feature set tests. The numbers shown are the relative label consistency and track count error. They are computed in the following way:

$$\begin{aligned} \hat{C}_{2,s} &= C_{2,s} - C_1 \\ \hat{\varepsilon}_{2,s} &= \varepsilon_{2,s} - \varepsilon_1 \end{aligned}, \tag{5.4}$$

where $C_{2,s}$ is the average label consistency and $\varepsilon_{2,s}$ is the track count error of *test 2* when using feature set *s*. C_1 and ε_1 are the average label consistency and track count error respectively of test 1.

Table 5.5: Feature set test results

Feature set	Relative track consistency (+ better)	Relative track count error (- better)
Position (POS)	+0.019	±0.000
Color (COL)	-0.127	-0.210
Size (SIZ)	-0.091	-0.368
Center of gravity (COG)	+0.001	±0.000
Pixel covariance (COV)	-0.027	-0.157
POS + SIZ	-0.020	-0.052
POS + COG	+0.004	±0.000
POS + COV	-0.018	-0.052
COL + SIZ	-0.129	-0.263
COL + COG	-0.002	±0.000
COL + COV	-0.075	-0.210
SIZ + COG	-0.024	-0.052
SIZ + COV	-0.097	-0.263
COG + COV	-0.034	-0.052
POS + COL + SIZ	+0.003	±0.000
POS + COL + COG	+0.003	±0.000
POS + COL + COV	+0.003	±0.000
POS + SIZ + COG	-0.020	-0.052
POS + SIZ + COV	-0.028	-0.052
POS + COG + COV	-0.008	±0.000
COL + SIZ + COG	+0.003	±0.000
COL + SIZ + COV	-0.106	-0.315
COL + COG + COV	+0.003	±0.000
SIZ + COG + COV	-0.027	-0.052
POS + COL + SIZ + COG	+0.003	±0.000
POS + COL + SIZ + COV	+0.003	±0.000
POS + COL + COG + COV	+0.003	±0.000
POS + SIZ + COG + COV	-0.027	-0.052
COL + SIZ + COG + COV	+0.003	±0.000
POS + COL + SIZ + COG + COV	+0.003	±0.000

5.2.6 Discussion of the second test results

A feature set that performs better than the one used in the first test (expected position and average color) should optimally have an improvement in both average label consistency (denoted by a positive number in the table) and track count error (denoted by a negative number). The table shows that there is no feature set that does that. In fact, every feature set that does show an improvement in average label consistency, detects the same number of tracks as in the first test. Thus, judging from the table, this number of tracks is optimal with this particular

test video. There are improvements in the track count error in the table, but they are always accompanied by deteriorations in the average label consistency.

Something else the table shows, is that some positional expectation (either expected position or expected center of gravity) is necessary to have a reasonable average label consistency (one that does not deteriorate too bad) and indeed all feature sets that show improved performance have either expected position or expected center of gravity as one of their features.

Another conclusion that can be drawn from this table is that adding average color to a feature set is usually not a good idea, unless the feature set also has a positional expectation feature. All feature sets that do not have this positional expectation feature, but do have average color as a feature perform worse than the same feature set without average color. This is probably caused by the similarity of the color of the two robots (they are both yellow on top and black from halfway down). The average color feature makes it possible to “jump” from object to object when their color is similar. The positional expectation feature prevents this, because usually the other object (the one which is not currently tracked by this track) is too far away to be a serious candidate.

Finally, the table shows that a positional expectation feature is all that is needed to get a good tracking performance. The feature sets that only have either one or both of these positional expectation features perform better than the feature set of the first test.

Chapter 6

Implementation

6.1 System configuration

The system configuration follows from the initial project parameters and can be divided into three areas: hardware, operating system and software.

6.1.1 Hardware

- CPU: Intel Pentium III 1GHz
- Frame-grabber: Hauppauge WinTV GO
  Video digitizer: Connexant BrookTree BT878
  Inputs:
  · VHF/UHF RF input on Philips FI6748 TV tuner
  · Composite input
  Capabilities:
  · Image capture
  · Image overlay
  · Image clipping
  · Frame RAM
  · Image scaling
  Frames are captured as 24-bit RGB images. Maximum resolution of this card is 768x576 pixels. Scaling is done via hardware-implemented interpolation. Horizontal scaling is done via a 6-tap linear interpolation filter, for vertical scaling a 5-tap linear interpolation filter is used.
- VGA-display card: Matrox G400 AGP
- Camera: Sony EVI-D31
  Pan/tilt/zoom color camera featuring auto-focus, automatic white balancing and serial control interface. The serial (VISCA) control interface can be used to configure the camera as well as control its motion. This interface has been used to change the white balancing setting. The default setting is auto-white balancing, meaning it constantly adapts white balancing. Also see section 5.1.2 for more information about the VISCA interface.

6.1.2 Operating system

Red Hat Linux 7.0

- Kernel: 2.4.19
- Compiler: GNU gcc 2.96
- GlibC: 2.2.4 (currently, GlibC 2.2.5 is not supported, because of a bug in the libc.so library)
- Window manager: KDE 1.2.2

6.1.3 Software

The main implementation is done in C++. The executable has a *main loop* in which the different tasks, necessary for tracking, are performed. Initialization is done before the main loop. The tasks performed in the main loop are the following:

1. Grabbing an image frame.
2. Applying background subtraction to the image frame.
3. Applying morphological growing[1] or label growing[2].
4. Extracting features from the labeled foreground image in combination with the grabbed image frame.
5. Assigning foreground blobs to tracks.
6. Displaying the end result together with a number of intermediate results, produce output for the camera-to-camera tracking.

6.1.3.1 Before tracking

The image frame is captured in the form of an *Image<PixRGB<byte>>* class object (see appendix A for iLab classes). From this object, a C-array is created, which is used for all computations. The array is one-dimensional:

$$image = \{R_1, G_1, B_1, ..., R_n, G_n, B_n\}$$

After color space conversion to normalized RGB (rgI) (see section , the image array looks like this:

$$image = \{r_1, g_1, r_2, g_2, ..., r_n, g_n\}$$

This array is the input for the background estimation algorithm. The output of the background estimation algorithm is a one-dimensional C-array with foreground probabilities per pixel.

$$foreground = \{p_1, p_2, ..., p_n\}$$

This image is thresholded with a value of 0.6, resulting in a binary foreground image. The morphological operations (erosion, dilation) are applied on this image after which morphological growing is used to 'glue' fragmented objects together.

[1] Morphological growing is used to de-fragment objects. See section 2.2 for details.
[2] See section 2.3

The result is labeled according to Horn's algorithm[1].

6.1.3.2 Tracking

When the operations of the last section are completed, tracking can begin. The first step is to extract features from the labeled foreground regions. These features are then compared to properties of tracks that have been formed previously (if no tracks are present yet, features are only extracted to initiate new tracks). The resulting distance measures are converted to probabilities, of which the logarithm is taken as mentioned in chapter 3. A matrix is formed with combined probabilities:

$$M = \begin{pmatrix} p_{11} & \cdots & p_{1b} \\ \vdots & \ddots & \vdots \\ p_{t1} & \cdots & p_{tb} \end{pmatrix},$$

with b the number of available new blobs and t the number of tracks already formed.

This matrix is the input for the assignment algorithm. For the assignments we are using the Hungarian Algorithm[2]. Based on the output of this assignment algorithm, tracks are extended, new tracks are formed and tracks are terminated.

6.1.3.3 After tracking

After tracking, all that remains is to display the results and produce output for the camera-to-camera tracking. All C-arrays are converted to *Image* objects again, after which they can be displayed in an *XWindow* object. Tracks are directly drawn into the *Image* object. The list of tracks is analyzed to find out if any tracks were finished. If this is the case, their information is output in a text-file, to be used by the camera-to-camera tracking system.

[1] See Zwarthoed [16], section 2.4.2, pages 16-18 for an explanation of Horn's algorithm.
[2] See section 3.3.1

Chapter 7

Discussion and Conclusion

We have investigated the feasibility and performance of a feature-based visual object tracking system as a means of gathering information about objects in order to identify these objects in a multiple-camera environment. We have looked at different assignment methods as well as different blob construction algorithms and feature sets.

7.1 Discussion

Tracking moving objects is certainly not a straightforward process. There are a lot of uncertainties and the signal to noise ratio of different parts of the process is often hard to determine. Based on various approaches we created a tracking method using stationary cameras and an EM-based background subtraction system, to which we added a Bhattacharyya distance [18] measure to avoid the "growing together" of the background kernels. Because of the fragmentation of the foreground detection, we added two methods to grow these different fragments together to one blob: morphological growing and label growing.

The testing has shown that not many features are needed to get accurate tracking results. It has also shown that, when one robot is used, the two assignment methods in combination with any blob construction algorithm, produce identical results. This was expected, since choosing with only one option is not considered awfully difficult.

To have accurate tracking results, either expected position or center of gravity should be included as a feature, since it avoids jumps from one object to another when they are far apart.

Most of the feature sets produce more detected tracks than there are true tracks (in fact, in the tests of section 5.2.5 there is only one feature set that produces less tracks then there really are: average color combined with size). This means that the number of objects cannot be derived directly from the amount of detected tracks. The features gathered about the objects should be compared to decide which tracks describe the same object.

Which blob construction system should be used, depends on the particular situation. Morphological growing is most suitable if the background contains large evenly colored surfaces. When the background estimation fails to detect part of an object, the part is quite evenly detected as background, whereas when the background does not have large evenly colored surfaces, the failed part is usually fragmented itself. When this happens, label growing will produce a more constantly shaped blob. As a matter of fact, the blob shape produced by the label

growing algorithm over a number of frames is always more constant than a blob formed by morphological growing.

7.2 Conclusions

This thesis has shown that, although multiple features do make tracking objects more robust, it is not necessary to use a large number of them to get accurate tracking results. It can even make tracking results worse, since using too many features can “confuse” the assignment system, with deteriorating accuracy as a result. The described approach, using a small number of features and tracking multiple objects using the Hungarian assignment method works. An alternate method, the Rangarajan method, which does not necessarily produce the best assignment result, was shown to perform worse in situations where the differences in the probabilities for the assignments were very small. Although background estimation is not the optimal way of detecting foreground objects, it is sufficient for this tracking system. Since this system is supposed to be used to extract feature information about objects in order to track them in an environment with multiple cameras, the extracted features had to be sufficiently reliable, which means that the information making up the descriptive feature should only come from one single object.

7.3 Future work

In the future, the system could be improved by using a different background subtraction method, which incorporates pixel correspondence and inter-pixel relations to produce a more informative foreground blob. Also, different features could be used to track an object. The features we selected were only selected because they are easily computable and usually carry enough information to distinct between two objects (unless the two objects are twins wearing the same clothing).

Occlusion and partial occlusion should be detected and appropriately handled to bridge temporary (partial) invisibility of an object.

Since it is only necessary to produce a definite tracking result when an object has moved out of the image frame, it is possible to delay the final assignment decision. Multiple hypothesis tracking, where a number of 'good' results are kept and they are all expanded in the next frame, can be used. Only when this hypothesis tree has reached a certain size, a definite decision is made on the top-node and the rest of the tree is pruned. This approach makes it possible to 'see in the future' how well a certain assignment would work before actually doing it.

In the introduction, a method using sub-features was described to track traffic on a motorway. Using this system of grouping sub-features and tracking them can improve tracking performance. The only problem is that the grouping

method as used in [1] assumes rigid body motions, which is not applicable to humans. This means a different grouping method should be used to allow for the tracking of (corner-)features of non-rigid objects.

Appendix A

A.1 iLab classes

This section covers the classes used in VisiTracker that are part of the iLab Neuromorphic Vision C++ Toolkit. Classes found in this toolkit are:

Table A.1: iLab classes

Name	Description
Timer	Very precise timing class
Log	Implementing a logging system
Point2D	Class for a 2D point
PixRGB<T>	Definition of an RGB pixel with values of type T
Image<T>	Image class with pixels of type T
V4Lgrabber	Interface to the Video4Linux API
XWindow	Implementation of an X-Windows window frame

Appendix B

B.1 File structures

The software creates a number of different files with varying structures. This appendix explains the different structures and should provide all information necessary for reading the output of the software.

Image captures
Image captures are raw RGB images. This means that the red, green and blue values of the different pixels are simply dumped to a file. A correct raw RGB file has a size of $\boldsymbol{N = width \times height}$ bytes and looks like this:

$$\{R_1, G_1, B_1, ..., R_N, G_N, B_N\}.$$

Probability propagations
Files containing probability propagations (consecutive probability values for a set period of time) are text files. They consist of a frame number and a number of feature probabilities. It depends on which feature set is used, what the file looks like. The basic structure is:

$$\{frame_1, \{Basic\}_1, \{Extended\}_1, ..., frame_m, \{Basic\}_m, \{Extended\}_m\}.$$

For basic features, a one-frame structure looks like:

$$\{position, color, size\}$$

and for extended features:

$$\{alternate\_position, pixel\_\mathrm{cov}ariance\}.$$

Frame numbers are **int** numbers (signed whole numbers), all feature probabilities are **double** numbers (signed real numbers), written in a 7.5 format (7 digits total, of which 5 decimals).

Image videos
Image videos are series of raw RGB images in a wrapper containing information to accurately replay the recorded sequence. The video starts with a header:

C	**R**	**E**	**C**	**1**	**0**	w	w
w	W	h	h	h	h	fr	fr
fr	Fr	ts	#	#	#	#	#

- Bold “CREC10” is an identification string.
- ‘w’ is the image width (integer, 4 bytes)
- ‘h’ is the image height (integer, 4 bytes)
- ‘fr’ is the sequence frame rate (integer, 4 bytes)
- ‘ts’ is a Boolean, for timestamp inclusion.

After the header, the data starts. Images are a collection of a frame number (int), an optional timestamp (int) and an image, as described above. The end of a video can be detected by an ‘end of file’ (EOF) event.

Appendix C

C.1 Pseudo code

```
For X = 0 to image_width do
  For Y = 0 to image_height do
    If (label_image(X,Y) = requested label) then
    begin
      If X < current top_left x-coordinate then
        top_left x-coordinate = X;
      If X > current bottom_right x-coordinate then
        bottom_right x-coordinate = X;
      If Y < current top_left y-coordinate then
        top_left y-coordinate = Y;
      If Y > current bottom_right y-coordinate then
        bottom_right y-coordinate = Y;
    end;
box_width = (bottom_right x-coordinate) –
            (top_left x-coordinate);
box_height =(bottom_right y-coordinate) –
            (top_left y-coordinate);
center-x = (box_width / 2) + top_left x-coordinate;
Center-y = (box_height / 2) + top_left y-coordinate;
```

FigureC.1: *Pseudo-code describing the computation of the bounding box center.*

Bibliography

1. D. Beymer, P. McLauchlan, B. Coifman, J. Malik. A Real-time Computer Vision System for Measuring Traffic Parameters. *In Proc. of ther IEEE Conf. on Computer Vision and Pattern Recognition, 1997.* http://citeseer.nj.nec.com/beymer97realtime.html

2. Robert T. Collins, Alan J. Lipton and Takeo Kanade. A System for Video Surveillance and Monitoring. *In Proc. American Nuclear Society (ANS) Eighth International Topical Meeting on Robotics and Remote Systems, Pittsburg, PA, pages 497-501, April 25-29, 1999.*

3. Theo Gevers.Color Image Invariant Segmentation and Retrieval. *PhD Thesis, University of Amsterdam, 1996, pages 27-29.*

4. Stephen S. Intille, James W. Davis and Aaron F. Bobick. Real-Time Closed-World Tracking. *In Proc. of IEEE Computer Science Conference on Computer Vision and Pattern Recognition, Cambridge, MA, pages 697-703, June 1997.*

5. Stephen S. Intille and Aaron F. Bobick. Closed-World Tracking. *Proc. of the Fifth Int. Conf. on Computer Vision, pages 672-678, June 1995.*

6. Michael Isard and Andrew Blake. Condensation - Conditional density propagation for visual tracking. *International Journal of Computer Vision 29(1), pages 5-28, 1998.*

7. P. S. Maybeck. Stochastic models, estimation and control. Volume 1, chapter 1.*Academic Press, New York. 1979.*

8. Christos H. Papadimitriou, Kenneth Steiglitz. Combinatorial Optimization, Algorithms and Complexity. *Pentice-Hall, Inc, Englewood Cliffs, New Jersey, 1982.*

9. Hanna Pasula and Stuart Russel.Tracking many objects with many sensors. *In Proc. IJCAI-9 , pages 1160-1171, Stockholm 1999.*

10. K. Rangarajan and M. Shah. Establishing Motion Correspondence. *Computer Vision, Graphics and Image Processing, 54, pages 56-73, 1991.*

11. Timothy Huang and Stuart Russel. Object Identification: a Bayesian analysis. *In Artificial Intelligence 103, (1998), pages 77-93.*

12. Hannes Kruppa, Martin Spengler, Bernt Schiele. Context-Driven Model Switching for Visual Tracking. *In 9th International Symposium on Intelligent Robotic Systems, Toulouse, France, July 2001.*

13. Martin Spengler, Bernt Schiele. Towards Robust Multi-Cue Integration for Visual Tracking. *In 9th International Workshop on Computer Vision Systems 2001, pages 94-107, Vancouver, Canada, July 2001.*

14. Cor J. Veenman. Motion Correspondence, Image Segmentation and Clustering. *PhD Thesis, Technical University Delft, January 2002.*

15. W. Zajdel. Internal report on visual surveillance algorithms.

16. Frank Zwarthoed. Real-Time Object Detectie in Video met gebruik van Mixture Modellen voor achtergrondschatting. *Masters thesis, University of Amsterdam, April 17, 2002.*

17. Subhash Suri, Stephen Scott. Network algorithms (CS 541). *Source unknown, Fall 1995.*

18. Andrew Webb. Statistical Pattern Recognition. *Arnold Publishers, New York, 1999.*

Index

L

M

N

O

P

R

S

T

U

V

W