

Planar Surface Tracking and its Applications

SIMON GUSTAFSSON



**KTH Computer Science
and Communication**

Planar Surface Tracking and its Applications

S I M O N G U S T A F S S O N

Master's Thesis in Computer Science (20 credits)
at the School of Electrical Engineering
Royal Institute of Technology year 2007
Supervisor at CSC was Danica Kragic
Examiner was Jan-Olof Eklundh

TRITA-CSC-E 2007:096
ISRN-KTH/CSC/E--07/096--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.csc.kth.se

Abstract

In the field of computer vision, a large number of area trackers have been developed. Their general purpose is to track planar surfaces. Their success depends on the amount of suitable features present in the surface, but they can also be limited by variations in illumination as well as by distortions of the surface in other ways than the tracker was designed to follow.

This work compares a selection of implemented area trackers against each other. The first category of compared trackers is limited to tracking translations. A second category with first order optical flow trackers were also implemented, and their capabilities of tracking more complex planar transformations was investigated.

This thesis shows how to derive and implement a number of simpler trackers, as well as show certain applications where a number of trackers have to cooperate to achieve common goals. One of those applications shows how certain objects (in this case boxes) can be tracked by grouping a number of trackers together. Other applications include a tracker that can track severe perspective distortion of planar surfaces using trackers capable of only following affine transformations. Yet another application tracked stars in over 100 pictures of the night sky, greatly reducing the noise by cancelling the motion of the stars before summing all images together.

Följning av plana ytor samt relaterade applikationer

Sammanfattning

I exjobbet används flera datorseendemetoder för följdning av ytor på objekt. Mängden särdrag på den följda ytan bestämmer hur bra de presterar, men ytföljarna kan också begränsas av andra faktorer som belysningsvariationer samt förvrängningar av annan art än följaren var avsedd för.

Detta arbete jämför ett urval implementerade ytföljare mot varandra. Dels jämförs följare som enbart kan hantera translationer, och dels så jämförs följare som är baserade på första ordningens optiskt flöde. De senares förmåga att hantera komplexa modeller för den följda ytan testas också i praktiken.

Detta examensarbete visar utförligt hur man härleder och implementerar ett antal enkla följare baserade på optiskt flöde, samt presenterar ett antal applikationer där följare måste samarbeta för att nå gemensamma mål. En av applikationerna visar hur en box skulle kunna följas genom att använda sex följare. En annan applikation demonstrerar hur perspektivföljdning kan göras med en grupp följare som ensamma enbart klarar affin följdning. En ytterligare applikation demonstrerar hur bilder av natthimlen kan göras mindre brusiga, om stjärnornas rörelse i en mängd sekventiella bilder detekteras och kompenseras innan bilderna slås ihop.

Preface

The goal of this project has been revised since it was started. The original goal was to develop trackers and feature detectors for simple features such as lines and patches mainly in a two-dimensional setting, and then participate in the development of a Human Machine Collaborative System (HMCS).

The intended HMCS would have been a system where a user's ability to perform certain tasks could be improved by letting a robot and a user share the same physical tool. The tool would be equipped with sensors measuring the force applied by the user. The measured forces together with visual input from a camera fixated on the tool could then be filtered, possibly enhancing force and precision while reducing the amount of tremor of the human operator. The visual input would mainly be used to determine if the user tried to follow certain types of features, and the system could then apply virtual fixtures to enhance the user's performance at certain tasks.

Other people were going to contribute with software for a human-machine control framework. It would include capabilities of using the force sensors attached to the tool, as well as doing the actual path planning for the robot using only position and pose of the tool as input. Evaluation of different virtual fixtures (or other control laws) as well as automated classification of the user's desired actions was also determined to be outside the scope of my work.

My part of the project was only focused on computer vision, and on integrating that with the HMCS system, as well as evaluating the performance increase expected when visual feedback was available to the HMCS system.

Unfortunately, the robot available for this project had broken down before it was time to start working with the HMCS system, so that part of the work could never begin. After some time without the ability to work on the HMCS system, I started attaining regular courses. That meant almost completely postponing the thesis for half a year. After that, I worked on the parts that I considered as most interesting, including different area trackers as well as some useful code for vision-related applications.

When meeting my supervisor in November 2006, she recognised that a lot of work had been put in the area tracking department, and we selected a new final goal for the thesis. The goal involved making a simple prototype system capable of tracking boxes. The only requirements on the boxes was that their dimensions should be known, and that its sides should contain enough features to allow optical flow trackers to track any of the sides of the box.

This work will be made available at: <http://www.simong.se/papers>.

Contents

1	Introduction	1
1.1	A General Tracking System	1
1.2	Project Goals	2
1.3	Outline	2
2	Area Tracking	4
2.1	SSD Tracking	4
2.2	Short-circuited SSD Tracking	4
2.3	Optical Flow Tracking	5
2.4	Issues	6
2.4.1	Selection of Good Features to Track	7
2.4.2	Preprocessing	9
3	Post Processing	11
3.1	Prediction	11
3.1.1	Linear Prediction	11
3.2	Clustering	12
3.2.1	Clustering and Isometry Transformations	12
3.2.2	Clustering and Similarity Transformations	12
3.2.3	Clustering and Affine Transformations	13
3.2.4	Clustering and Perspective Transformations	14
4	Plane Tracking in Three Dimensions	15
4.1	Coordinate Systems	15
4.2	Retrieving Pose of Plane	15
4.2.1	Solving Pose by Knowing Tracker's Homography	17
4.2.2	Solving Pose by Knowing Corner Points	17
4.3	Post-processing the Solution	18
4.3.1	Building a Valid Solution	18
4.3.2	Minimizing Errors	19
4.4	An Alternative Way to Track the Pose of a Plane	19
5	Tracking of a 3D-box	21
5.1	Box Model	21
5.2	Individual Trackers	22
5.3	Tracker Handover	22

5.4	Discussion	23
6	Implementation	24
6.1	Image and Video	24
6.1.1	Text	24
6.1.2	Line	25
6.1.3	SImage	25
6.1.4	VideoSource	25
6.1.5	VideoSink	26
6.1.6	GrabberThread	26
6.1.7	MyScheduler	27
6.2	Feature Detectors	27
6.2.1	HoughTransform	27
6.3	Tracking and Clustering	27
6.3.1	LineTracker	27
6.3.2	SSDSlowTracker	28
6.3.3	SSDShortCircuitTracker	28
6.3.4	FlowTrackerCore	28
6.3.5	FlowTracker	29
6.3.6	PerspectiveTracker	29
6.4	Timing	29
6.4.1	RTC	29
6.4.2	Stopwatch	30
6.4.3	MeasureCycles	31
6.5	NCurses	32
6.5.1	OutputWrapper	32
6.5.2	OutputStreambuf	33
6.5.3	NCursesUser	33
6.5.4	NCursesClass	34
6.6	Additional Supporting Classes	34
6.6.1	ArgumentParser	34
6.6.2	MFileReader	35
6.6.3	MFileWriter	36
6.6.4	MY_CTRL_C_TRAPPER	37
6.7	Deprecated Classes	37
6.7.1	SMatrix	37
6.7.2	SSDFlowTracker	38
6.7.3	SSDFlowTrackerRect	38
6.7.4	ClusterPoints	39
7	Experimental Evaluation	40
7.1	Evaluation of Area Trackers	40
7.1.1	Time Consumption	40
7.1.2	Deviation from Expected Position	42
7.1.3	Step Response	42
7.1.4	Extension of Range for Optical Flow Trackers	43
7.1.5	Behaviour at Different Tracking Speeds	44

7.2	Applications	46
7.2.1	Star Photography	46
7.2.2	Iterative Tracking	47
7.2.3	Tracking of Three Dimensional Boxes	48
8	Conclusions	50
8.1	Regarding Area Trackers	50
8.2	Regarding Clustered Perspective Trackers	50
8.3	Regarding Star Photography	51
8.4	Regarding Tracking of Three Dimensional Boxes	51
8.5	Regarding Line Tracking and Detection	51
	Bibliography	53
A	Affine Approximations of Projective Transformations	55
B	Decomposition of a Projective Transformation	57
C	Optical Flow Estimation	59
C.1	Tracking Translations	60
C.2	Tracking Affine Transformations	62
C.3	Tracking Affine Transformations (two step approach)	65
C.4	Tracking Perspective Transformations	68
C.5	Intuitive Description	71
D	Notes about NOMAN	76
D.1	ImageShow2	76
D.2	CASMatrix	76
D.2.1	Multiplication	77
D.2.2	Temporary Objects	77
D.2.3	Using Optimised Libraries	78
D.2.4	Optimisation Flag	79
D.3	Resolving Compilation Issues with gcc 4.4.1 and FC6	79
D.3.1	libMath.so	79
D.3.2	SELinux	79
D.3.3	V4L2Grabber	79
E	Line Tracking and Detection	81
E.1	Line Tracking	81
E.1.1	Edge Tracking	81
E.1.2	Investigated Improvements	84
E.2	Line Detection – Hough Transform	87
E.2.1	Using 8x8 Patches	87

Chapter 1

Introduction

Feature detection and tracking are two interesting fields in computer vision. They are both used in a wide range of applications. One group of such applications rely on trackers to determine an optical flow field. That flow field can then be used for motion detection, video compression, video stabilization, and in some cases even for three-dimensional reconstruction of objects. Other groups of applications include human machine collaboration, vehicle tracking (both in traffic surveillance and for platooning), as well as general tracking of planes.

1.1 A General Tracking System

A general tracking system uses saved sequences or input from real time video as the source of frames to process. The system could either perform one tracking iteration for each tracker in each frame, or use more sophisticated ways to schedule when to run its different trackers. Which features to track could either get determined automatically (by feature detection), or manually by selection of features to track.

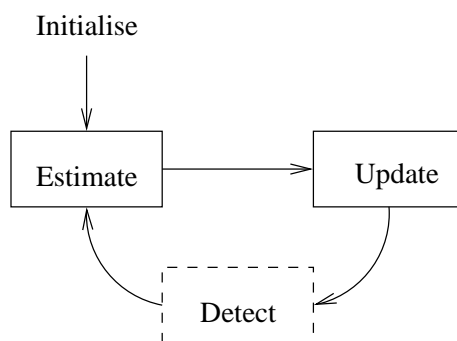


Figure 1.1: A general tracking system.

The tracking system would undergo the states depicted in figure 1.1. It is first initialised either manually by specifying features to track, or automatically by detecting features in the first frame. Then all trackers in the system are queried to estimate the new parameters of their tracked features. That estimation usually relies on assumptions about the tracked object, most commonly that the object would

retain its shape and appearance. This description do not apply for systems where features are not tracked, but rather detected in the entire frame for each frame, and then correlated to features detected in the previous frame. The third step in the image is to update the parameters of each tracker using the estimated values. Since both previous as well as current estimations of all tracked features are known, they can all be allowed to interact. As an optional step, new features could also be detected regularly, but depending on the application, detection of new features might be skipped.

1.2 Project Goals

Most of the effort in this work has been put into examining and experimenting with area trackers, even though line tracking and detection has been addressed briefly as well. Two main groups of area trackers have been investigated: correlation based trackers, and optical flow trackers. The correlation based ones were limited to tracking translations, while separate optical flow trackers were constructed for tracking of several different motion models.

This thesis also presents certain applications where a number of trackers had to cooperate to achieve common goals. One of those applications shows how certain objects (in this case boxes) can be tracked by grouping a number of trackers together. Other applications include a tracker that can track severe perspective distortion of planar surfaces using trackers capable of only following affine transformations. Yet another application tracked stars in over 100 pictures of the night sky, greatly reducing the noise by cancelling the motion of the stars and summing all images together.

During the course of this work, several support classes has been developed as well. They include classes for simulating a real time video source using saved sequences, classes simplifying the use of NCURSES, a command line argument parser, and several other reusable components.

1.3 Outline

Chapter 2 describes how correlation based SSD trackers work, and contains a brief overview of optical flow tracking. That chapter also discusses the process of determining good features to track, and finishes with two methods to combat illumination changes, which can have negative impact on optical flow trackers. Chapter 3 discusses prediction, as well as clustering of trackers, and demonstrates tracking of perspective transformations using a group of affine trackers.

Chapter 4 describes how knowledge about the camera and the tracked object can be used to retrieve the pose of a tracked plane, and it also introduces a possibly novel method to restrict the trackers movement to perspective transformations, at the same time as the trackers state contains the pose of the tracked plane. Chapter 5 explains how the pose of a box can be tracked by knowing the dimensions of the box and the internal camera parameters.

Chapter 6 briefly describes some of the developed C++ classes, which might be useful in other projects, and Chapter 7 begins with evaluation of trackers, and ends

with evaluation of certain applications. The applications include “Star photography”, “Iterative tracking” and “Tracking of three dimensional boxes”. Chapter 8 finally contain the conclusions of this thesis.

Appendix A contain relationships between affine and perspective transformations, and appendix B shows how to decompose a projective transformation. That is followed by appendix C, which shows how to mathematically derive several optical flow trackers, and also contains a separate more intuitive description of optical flow trackers.

Appendix D suggests some changes and improvements to NOMAN, and appendix E contains some work on line tracking and detection.

Chapter 2

Area Tracking

Area tracking is the process of tracking a specified area of an image in subsequent frames. The *constant brightness assumption* is the most important assumption for many types of area trackers. It states that the brightness of any point of a tracked patch will remain constant during tracking, and that only the positions of the points can change. A measure of deviation from the constant brightness assumption for two aligned images is the sum of squared differences (SSD). The SSD is equal to 0 for identical images, and gets higher the more different the images becomes.

$$SSD = \sum_{all\ x,y\ in\ patch} (I_1(x, y) - I_2(x, y))^2 \quad (2.1)$$

Where I_1 and I_2 denotes two separate images. This SSD equation can be used directly for tracking as in section 2.1 and 2.2. The SSD Equation can also be used to develop more complicated tracking algorithms, such as the one described in section 2.3.

2.1 SSD Tracking

To perform basic SSD Tracking, one has to determine some possible locations of the tracked area in the image. Then the SSD is calculated for each of those possible locations. The simplest scheme for choosing candidate locations is to use all locations within a specified radius of the trackers last position. The location with the best correlation (lowest SSD) is then assumed to be the location of the tracked area.

The most striking disadvantage with this method is that it is computationally expensive. It also lacks the ability of tracking objects which are rotated or scaled without much more extra effort.

2.2 Short-circuited SSD Tracking

One way of improving the performance of SSD tracking is to reduce the amount of unnecessary computations. One useful method for that is called loop short-circuiting[3]. It involves aborting computation of SSD:s when they get larger than

some previously calculated SSD. Since the lowest SSD corresponds to the best estimation of the position, continued evaluation of SSD:s whose sum already exceeds the previously lowest SSD is just a waste of time.

When calculating these SSD:s, it has been shown that the order in which the SSD:s are evaluated in gets important, as well as the order of the pixels involved in each sum.

2.3 Optical Flow Tracking

Optical flow trackers[10][9] provide a much faster way to track patches than the other trackers described in this paper. An optical flow tracker would not have to calculate all possible SSD values for all possible displacements. Instead, it would solve an equation system to directly estimate the movement of the tracked patch. The equation system could very easily be derived to allow tracking of most motion models, from translations up to arbitrary homographies¹. Optical flow trackers for several common motion models are derived in appendix C.

To achieve the high performance associated with optical flow trackers, there are a number of simplifications in their equation systems. The benefit of that is that most values can be precomputed before the tracker is used, making tracking a very fast operation. The drawback is that the simplifications make the tracker more sensitive to illumination variations, as well as decrease the trackable range to the subpixel range. Solutions to reduce the influence of illumination changes are discussed in section 2.4.2, and the solution that would increase the trackable range is simply to downscale² images during tracking. When the tracked image is down-scaled, the precision of the optical flow tracker is reduced. To maintain the original resolution, tracking could be committed by first tracking using strong downscaling, and then repeatedly track with a successively reduced amount of downscaling. It should be noted that the accuracy of optical flow trackers is much higher than the accuracy achieved by the trackers in sections 2.1 - 2.2, and when only translations are tracked, a downscaling by the factor of two or four gives comparable results.

For optical flow trackers, the two most important coordinate systems are the coordinate system used by the camera, and the coordinate system of the reference patch used by the tracker. As one tracker might use several reference patches with different amount of downscaling, there are even more coordinate systems in play.

An overview of the tracking process focused at the actions in different coordinate systems is depicted in figure 2.1. The homographies specified for each coordinate system shows the relationship to the rectified reference image used when initialising the tracker. To the left in the image is the camera coordinate system, and the homography H (which directly relates the pixels camera coordinates and rectified coordinates) completely describes the current state of the tracker in combination with the width and height of the rectified but not scaled image. The image from the camera is rectified and downscaled by the transformation $H^{-1}Q$, where Q depends on the amount of downscaling. If the deviations between the reference

¹Use of homographies and homogeneous coordinates allow all coordinate transformations used within this thesis to be described as matrix multiplications. Refer to [7] for details.

²Downscaling by the factor x would imply that only every x pixel is used in any direction.

and the current rectified and downscaled image is in the sub pixel range, solving the optical flow equation system yields the error estimate $Q\Delta M$, which simply corrects the homography in the rectified and downscaled coordinate system. If other parameters than a homography is used for the motion model, its corresponding homography must be constructed from the estimated error. Finally, the corrected homography has to be transformed into camera coordinates to get the new state of the tracked patch.

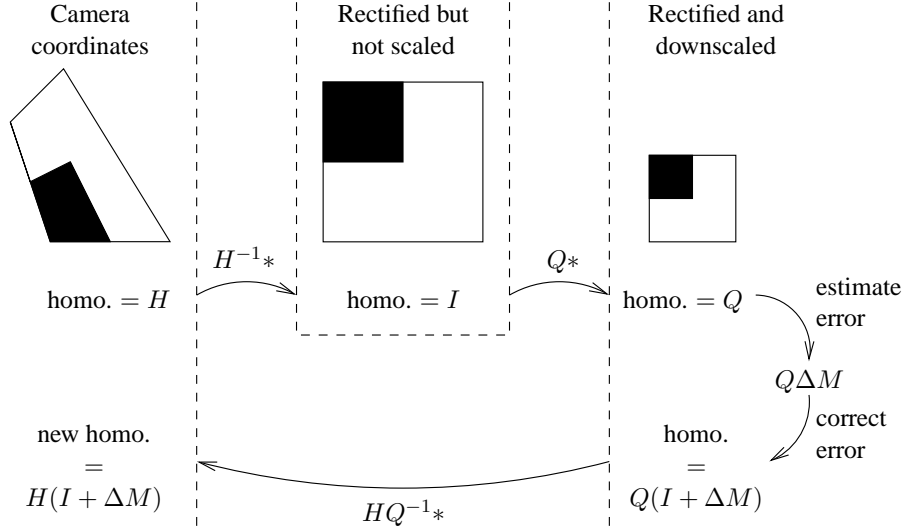


Figure 2.1: Tracking process overview showing all coordinate system transformations used by an optical flow tracker.

2.4 Issues

A small note is made here about prediction and downscaling, and the following subsections discuss selection of good features to track (2.4.1), as well as preprocessing of images to remove flickering (2.4.2).

When tracking fast movements, or tracking is computationally expensive, prediction and downscaling can be useful. Firstly, prediction can acquire better estimates of the current tracker state as described in section 3.1. This makes it possible to use a smaller candidate area in algorithms trying out possible matching positions one by one.

Secondly, downscaling is essential together with optical flow, as optical flow tracking only works reliably for movements in the sub-pixel range. Scaling the image down by for instance the factor 2 would mean that only every second pixel is used by the tracker in both x and y directions. That means that the motion of the tracked patch as seen by the tracker would be halved. Downscaling also reduces the amount of computations. Scaling the image down by the factor 2 would only pass every fourth pixel of the original image to the tracking algorithm, thereby saving 75% of the computation time in algorithms which scale linearly with the number of used pixels.

2.4.1 Selection of Good Features to Track

Selection of good features to track is important but not easy. The most common ideas seem to focus at determining when translational motion can be tracked without stumbling against the aperture³ problem. Two papers addressing this issue with the basis of eigenvalues are [15] and [12]. They both analyse the

$$A^T A = \sum \begin{bmatrix} I_x^2 & I_{xy} \\ I_{yx} & I_y^2 \end{bmatrix}$$

matrix that is used in optical flow tracking. The description in [15] talks about eigenvalues of the $A^T A$ matrix in the context of solving linear equation systems. They basically say that the eigenvalues must be larger than some minimum value (dependent on image noise), and that the matrix $A^T A$ should be well conditioned to make the equation system used while tracking solvable. Eigenvalues differing by several orders of magnitude would break the conditioning requirement, but was not handled anyway in [15]. They considered the largest eigenvalue as bounded since the bounded pixel values in the patch could not generate arbitrary large eigenvalues.

The description in [12] talks about the same $A^T A$ matrix, but they obtained that matrix by disregarding all temporal differences that could be considered when moving from equation (C.2) to equation (C.7) (both in appendix C). Their result was

$$ERR = \sum_{all \bar{x}} \left(\begin{bmatrix} \Delta x & \Delta y \end{bmatrix} A^T A \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \right) w$$

which gives approximate deviation from the constant brightness assumption for displacements Δx and Δy . The description in [12] talks about the eigenvalues of $A^T A$ in the context of analysing another simple corner detector. They had two points especially worth noting. Firstly, a contrast increase by p would increase the eigenvalues by p^2 , which have to be taken into consideration depending on preprocessing (section 2.4.2). Secondly, it was possible to use thresholds without explicitly calculating the eigenvalues. To be able to do that, the trace and the determinant of $A^T A$ should be considered:

$$Tr(A^T A) = \lambda_1 + \lambda_2 = \sum I_x^2 + \sum I_y^2$$

$$Det(A^T A) = \lambda_1 \lambda_2 = \sum I_x^2 I_y^2 - \sum I_{xy}^2$$

and then thresholding of trackable corner regions would be performed on the response

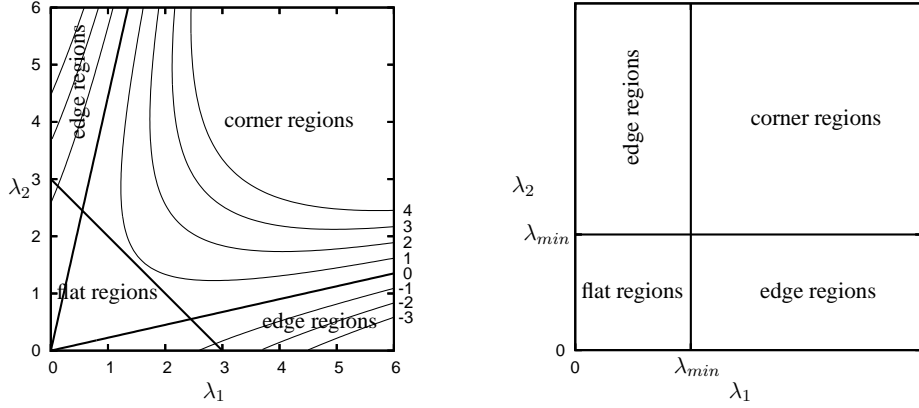
$$R = Det - k Tr^2 \quad (2.2)$$

where negative values of R corresponds to edge regions, and positive values to corner regions that are easy to track. Flat regions are detected by thresholding Tr , the sum of both eigenvalues.

A comparison between the threshold strategies in the two papers can be seen in figure 2.2. The only practical difference is that [15] disregards the conditioning of the matrix based on their practical experience, while [12] keeps that requirement.

³The aperture problem occurs when the tracked patch contains too little information to allow tracking in both x and y direction. The patch could be uniform, or contain edges in one direction only.

2. Area Tracking



(a) Thresholding of eigenvalues as performed by [12]. Iso-contour lines of equation (2.2) for $k=0.15$ are drawn.

(b) Thresholding of eigenvalues as performed by [15].

Figure 2.2: Feature thresholding based on eigenvalues. The main difference is that 2.2(b) disregards the conditioning of the matrix based on practical experience, while 2.2(a) keeps that requirement.

The additional possibility of adding a cornerity constraint to prefer features with many corners and curves is mentioned in [14], which would be highly beneficial for trackers not restricted to translational movements.

Unfortunately, the process of determining the thresholds used for determining if an area could be tracked is not straightforward. First of all, any weighting should be incorporated when calculating the eigenvalues used for thresholding, and preprocessing affecting the contrast of the image would have to be left out or be compensated for, as that changes the eigenvalues.

Another problem was that normalization of eigenvalues of different trackers with different sizes was not straightforward, possibly making it necessary to retrieve customized thresholds for all tracker sizes that would be used. For many applications, the size of automatically detected regions is predetermined, and only the initial problem of setting thresholds for fixed size trackers would have to be solved.

The attempted method of determining the thresholds for fixed size trackers was to simulate translational motion of an image while tracking many regions within it. The length of displacement was increased for each frame, and the direction of displacement was also changed wisely. Unidirectional displacement could falsely benefit regions containing edges, and erratic changes in displacement direction could give zero mean displacement between neighbouring frames, making it less demanding for the tracker.

Under those conditions, it would be possible to harvest “survival times” defined as the number of frames before a tracker deviated from its supposed position by more than some small constant. The “survival times” would correspond directly to the suitability of tracking the tracker’s associated patch.

The result of such experiments resulted in eigenvalues and “survival times” for a large number of trackers. When those data were evaluated, the difficulty of

selecting appropriate thresholds for eigenvalues became evident. The thresholds, which would correctly identify almost all good trackers was also letting over 25% of the worst trackers pass as good ones. When setting the thresholds such that all the worst trackers were blocked, many of the best trackers were not detected at all instead.

Discussion

The conclusion of my experiments is that it is possible to calculate eigenvalue thresholds for classifying patches as trackable. The drawbacks are that such classification sometimes makes mistakes, and that the thresholds have to be determined individually for each size of the tracked patches. For fixed size patches, the performance can be considered adequate in most cases.

Whenever possible, I suggest that other means should be used as well to find trackable patches. One such example would be background subtraction of static scenes with few moving objects, where traffic surveillance serves as an example. Another approach would be to “try out” trackers for a specified number of frames before acknowledging them as good trackers.

2.4.2 Preprocessing

Preprocessing of images before feeding them to different trackers can often be a useful tool. For instance, using gradient images or laplacian of images instead of the regular images can remove some of the influences from reflections, although the impact of high frequency noise makes that less suitable for normal image sequences[14].

Another possible task for preprocessing is to remove effects of flickering illumination between frames. Two methods, which does that has been tested, and the results can be seen in image 2.3. The first method was derived after observing a few histograms of a sequence of flickering frames: it was apparent that flicker in the illumination affected the image by multiplying all values in a frame by a factor specific for that frame. That observation could be written as

$$c_i * I_{frame\ i} = I_{frame\ 0}$$

This unknown factor c_i can be determined by forcing all mean intensities to the mean intensity of the first frame.

$$\frac{1}{N} \sum c_i * I_{frame\ i} = \frac{1}{N} \sum I_{frame\ 0}$$

By multiplying all pixel values in an image by the easily determinable c_i , the image intensities will be much closer to the intensities in the first frame.

The used camera was however not acting according to the model for the brightest 15% of the histogram. Therefore, histogram equalization⁴ was also tested. The decision to prefer histogram equalization over fixation of average was taken after considering the results in image 2.3(b). The behaviour of the current camera at the

⁴the histogram equalization used was using 16 bins, and linear interpolation

2. Area Tracking

highest intensities was just too inconsistent with the first model to allow it to be used.

Flicker compensation could be applied either locally to the tracked patch, or globally to the entire frame sent from the camera. Local flicker compensation introduces an inability to track patches lacking higher frequency content. As an example, consider the image $I(x, y) = \alpha x + \beta y$. Histogram equalization for any patches from that image would result in identical patches regardless of the translations in the image, thus making tracking impossible. Applying flicker compensation globally on the entire frame sent from the camera would be necessary to allow patches lacking high frequency content to be tracked.

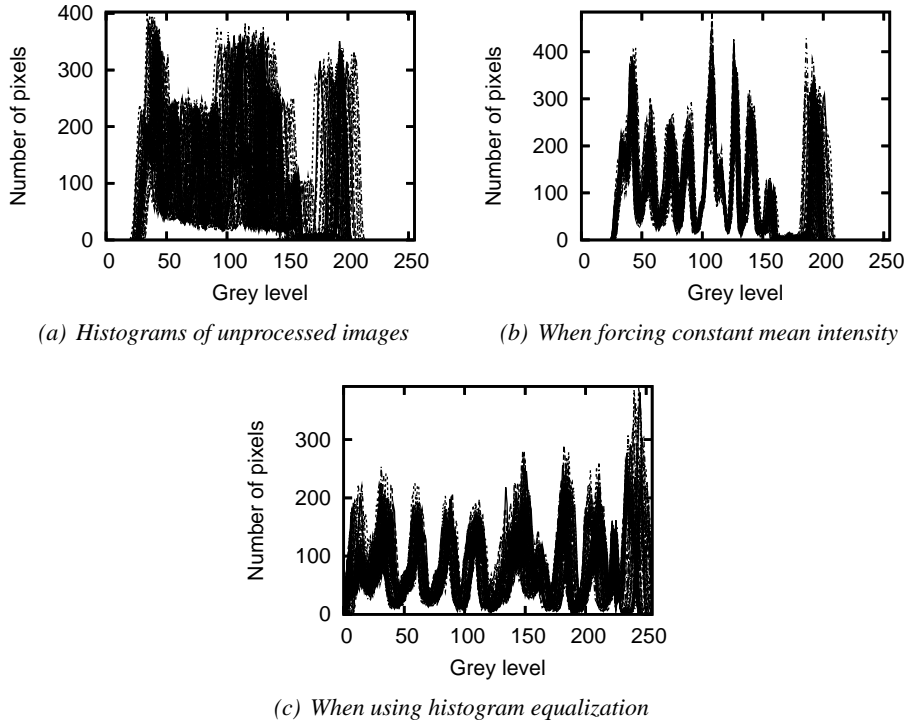


Figure 2.3: Test results of flicker compensation in a movie. The plots contain 75 histograms for 75 different frames, the camera was stationary, and the target consisted of 10 bars of different intensities.

Chapter 3

Post Processing

This chapter provides some information about possible post processing that can be applied to the results from tracking. First, the simple task of predicting future positions of trackers is addressed. After that, some means to cluster trackers to be able to combine their results are described.

Another form of post processing can be seen in chapter 4 (page 15), where information from an optical flow tracker is used to retrieve the three-dimensional pose of a plane.

3.1 Prediction

Some trackers have trouble tracking fast moving objects. If the fast moving object's acceleration is below some maximum level, prediction can be used to facilitate tracking. The prediction would use old states of the tracker to predict future states. The limit imposed on accelerations would then give a limit on the prediction error, making it possible to determine how “far away” the tracker has to be able to search for the tracked object.

3.1.1 Linear Prediction

Linear prediction is one of the simplest forms of prediction. Linear prediction uses the last and the current state of the tracker to estimate future states. The application of linear prediction to coordinates is straightforward:

$$\bar{\mathbf{x}}_{t+\tau} = \bar{\mathbf{x}}_t + \tau(\bar{\mathbf{x}}_t - \bar{\mathbf{x}}_{t-1})$$

When applying linear prediction to affine parameters describing more complex relations, one out of two roads can be taken. The first solution is to use the same principle as above: the affine parameter matrix \mathbf{A} can be updated using a differential matrix. This will however only work while the changes in the matrix \mathbf{A} are small (which they are going to be during optical flow tracking).

$$\mathbf{A}_{t+\tau} = \mathbf{A}_t + \tau(\mathbf{A}_t - \mathbf{A}_{t-1})$$

The second solution for updating affine parameters is to extract a suitable set of parameters from the affine matrix \mathbf{A} . These parameters could then be predicted

separately, and then recombined into the predicted matrix $\mathbf{A}_{t+\tau}$. This would give better responses when there are large changes in \mathbf{A} between each tracking cycle. The process of separating parameters for a projective transformation is described in appendix B, and that process can be simplified to handle less complex motion models as well. As my implementation mostly relies on Optical Flow tracking of slowly moving objects, I would not benefit from this second solution.

3.2 Clustering

It is possible to cluster trackers related by some measure by grouping the related trackers. The process of clustering can add to the usefulness and reliability of the tracking system. Trackers in each cluster would have to be processed together as a group to improve the performance of the tracking system.

Obvious benefits of clustering would be that one could update the state of all trackers in a group without having to run all individual trackers, as only a few are needed to determine the motion of the entire cluster. Another benefit would be that much more information would be available for predictors.

To determine which measure to group trackers according to, one first has to determine an appropriate model of the trackers. When limiting the model to allow only translations and rotations, the measure used to group trackers could be equidistance. Some reflections regarding the usefulness of clustering for a couple of motion models are mentioned in the following sections.

3.2.1 Clustering and Isometry Transformations

Isometry transformations are transformations, which preserve euclidean distances, and hence also angles and areas. Isometry transformations could thus include translations and rotations. For isometries, equidistance would be an ideal measure for clustering. Clustering could be implemented by calculating all distances between all trackers after each frame. That would require $\frac{1}{2}(N^2 - N)$ distances to be calculated, where N is the total number of trackers. The trackers whose distances between each other never change more then some threshold would then be clustered in the same group.

Low-pass filtering of the distances between all trackers, and clustering of trackers whose distances never deviate more then a threshold from the low pass filtered distances would allow slow deviations from the equidistance condition. Allowing for very slow deviations could for instance allow clusters to be preserved even when the patches are travelling along paths distorted by perspective or barrel distortion.

3.2.2 Clustering and Similarity Transformations

Similarity transformations are transformations allowing for translation, rotation, and isotropic scaling. During similarity transformations, only angles between lines, ratios of lengths, and ratios of areas are preserved. The proper ways to cluster trackers undergoing similarity transformations would be to use one of those invariants. For N trackers, there would be $\frac{1}{8}(N^4 - 2N^3 - N^2 + 2N)$ ratios of lengths to

calculate. For larger number of trackers, this would soon require too many computations to be practical (see table 3.1).

Table 3.1: relationship between number of trackers, and related number of unique lengths and length ratios.

<i>trackers</i>	<i>unique lengths</i>	<i>unique length ratios</i>
3	3	3
4	6	15
5	10	45
10	45	990
15	105	5460
20	190	17955

In many tracking applications, the tracked objects would not move fast towards or away from the camera. That means that the scaling that the tracked objects are undergoing is very slow compared to translations and rotations of the tracked object. This makes it possible to cheat in some situations, by using equidistance for clustering instead of ratio of lengths. That implementation could low-pass filter the distances between all trackers, and cluster the trackers whose distances never deviate more then a threshold from the low pass filtered distances. This would allow trackers to slowly break the equidistance condition, without splitting up the cluster. Clusters experiencing no distortion, or slow distortions, could hence be handled as an isometry transformation instead.

3.2.3 Clustering and Affine Transformations

Individual trackers trying to track patches undergoing affine transformations requires more information in the tracked patches then simpler trackers. The patches sometimes contain less information then necessary to correctly follow the affine transformations. If one would cluster some trackers undergoing the same affine transformation, one would see that all trackers and the cluster of trackers would undergo identical transformations¹.

This makes it possible to calculate the affine parameters for the cluster using only centre positions of the trackers, and then pass those parameters back to the trackers constituting the cluster. It would then be possible to follow affine transformations using a few trackers, which by themselves only would be able to follow similarity² transformations. If one would use trackers capable of following affine transformations, their parameters and the group's parameters could be fused. I believe that the fusion would result in more accurate tracking then possible without cooperation between trackers.

¹Different sub-regions of an affine transformation only differ in their displacement parameter.

²A similarity can be decomposed into scaling, rotation, and translation.

3.2.4 Clustering and Perspective Transformations

Clustering of affine trackers can be used to get information about perspective transformations. The main idea is that an affine transformation can approximate the effect of a perspective transformation in a small area. Hence, several small affine trackers can approximate the effects of a perspective transformation over a larger area. Approximation of a perspective transformation by an affine transformation is mathematically described in appendix A.

A basic implementation would calculate the perspective transformation by combining data from all individual affine trackers. Feedback from calculated perspective parameters could then get passed back to the affine trackers to keep diverging trackers from drifting away. Figure 3.1 shows an area undergoing a perspective transformation, which was tracked using a cluster of affine trackers.

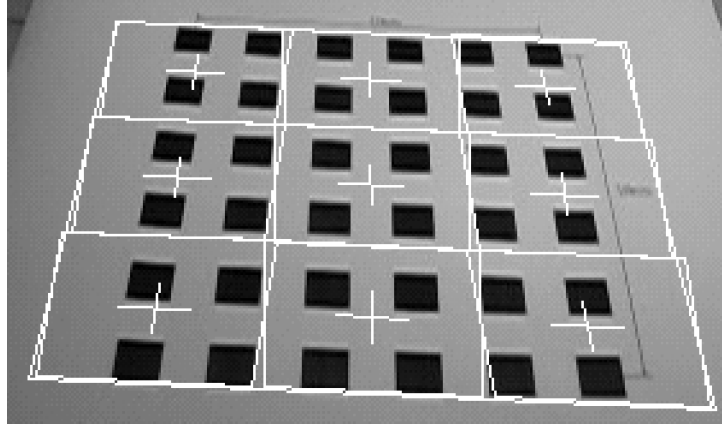


Figure 3.1: Perspective transformation tracked by a cluster of affine trackers.

Preliminary Results

So far, the performance of a clustered perspective tracker utilising a mesh of affine trackers seems to be very close to the performance of an optical flow perspective tracker. The most positive thing about the clustered perspective tracker is that it seems to converge slightly faster in each iteration than its optical flow counterpart.

On the negative side, the clustered tracker is currently a little bit less reliable than a single optical flow perspective tracker. To enhance the reliability, efforts should be made to ensure that none of the small trackers gets an impossible job (such as tracking something completely uniform). Weighting of affine trackers in the cluster based on the reliability of each tracker could also lead to a reliability improvement for the clustered tracker. Additionally, the reliability increases when an overlap is used between the affine trackers in the cluster, but on the expense of computation time. As the clustered tracker without overlap only saves roughly 10% of computation time, I would suggest that clustered trackers would not be used as a replacement of optical flow perspective trackers.

Chapter 4

Plane Tracking in Three Dimensions

The task of tracking a planar rectangular surface undergoing perspective transformations has already been solved (see appendix C.4). The next step would be to use knowledge about the tracked patch to determine its translation and rotation in three dimensional space. The approach described in [4] was used here because of its simplicity. It should be noted that better ways to estimate the pose exist [17][5], but they were not used. The only requirements for the method in [4] are that the internal camera parameters as well as the dimension of the tracked rectangular patch should be known. The Camera Calibration Toolbox for Matlab®[2] was used to estimate the internal camera parameters, as implementation of a camera calibration algorithm was considered outside the scope of this work. The following sections will outline how to estimate the pose of a tracked patch.

4.1 Coordinate Systems

The tracked rectangular patch in three dimensional space is shown in figure 4.1(a). To simplify calculations later on, we assume that the patch lies in the XY-plane (with Z-coordinates = 0). The (X_i, Y_i) coordinates in figure 4.1(a) are determined by the physical dimensions of the tracked patch. The calculated pose will show the displacement from this initial pose.

Figure 4.1(b) shows the tracked patch as seen by the camera. The corners (x_i, y_i) of the patch in image coordinates should be easy to retrieve from any area tracker.

4.2 Retrieving Pose of Plane

To retrieve the pose of the rectangular planar patch, one has to look at the relationship between the real world coordinates and the image coordinates for the corners

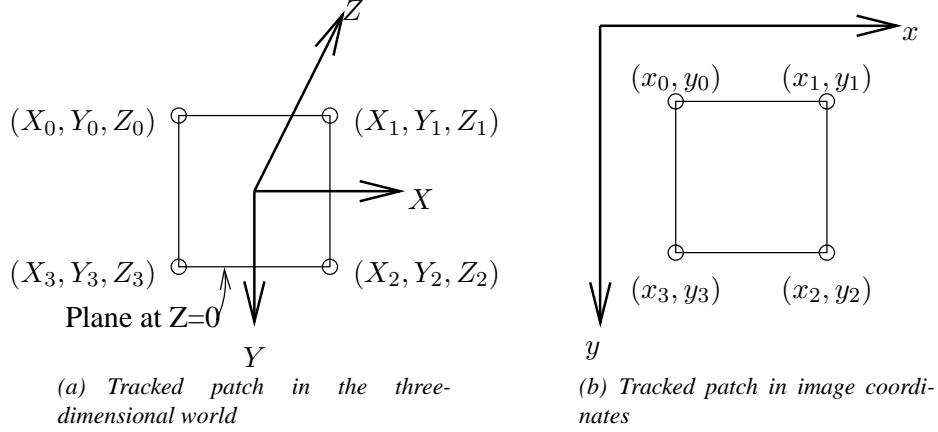


Figure 4.1: Patch in real world, and as seen by the camera.

of the tracked patch:

$$s \begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} = \mathbf{K} \mathbf{T} \begin{pmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{pmatrix}$$

Where the camera matrix \mathbf{K} contains the internal camera parameters:

$$K = \begin{bmatrix} a & b & c \\ 0 & d & e \\ 0 & 0 & 1 \end{bmatrix}$$

And \mathbf{T} is the unknown transformation of the patch in the real world, and both the image coordinates and the three dimensional coordinates are homogeneous. Since we have set the Z-coordinate of the tracked patch to zero, the equation above could be rewritten:

$$s \begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} = \mathbf{K} \mathbf{R}_{rot} \mathbf{T}_{trans} \begin{pmatrix} X_i \\ Y_i \\ 0 \\ 1 \end{pmatrix} = K \begin{bmatrix} | & | & | \\ \bar{\mathbf{r}}_1 & \bar{\mathbf{r}}_2 & \bar{\mathbf{t}} \\ | & | & | \end{bmatrix} \begin{pmatrix} X_i \\ Y_i \\ 1 \end{pmatrix} \quad (4.1)$$

$$\text{where } \mathbf{R}_{rot} = \begin{bmatrix} | & | & | & 0 \\ \bar{\mathbf{r}}_1 & \bar{\mathbf{r}}_2 & \bar{\mathbf{r}}_3 & 0 \\ | & | & | & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } \mathbf{T}_{trans} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Equation (4.1) could be further simplified by first multiplying it by the inverse camera matrix \mathbf{K}^{-1} , thereby yielding:

$$s \begin{pmatrix} x'_i \\ y_i \\ 1 \end{pmatrix} = \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ r_{31} & r_{32} & t_z \end{bmatrix} \begin{pmatrix} X_i \\ Y_i \\ 1 \end{pmatrix} \quad (4.2)$$

where

$$s \begin{pmatrix} x'_i \\ y'_i \\ 1 \end{pmatrix} = \mathbf{K}^{-1} \begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} \quad (4.3)$$

Equation (4.2) is a homography mapping the planar patch (in three dimensional space) into a patch in the (x'_i, y'_i) coordinate system. The most important part of that mapping is, that the translation and parts of the rotation matrix can be determined directly. The missing column \bar{r}_3 of the rotation matrix is also known, as it is the cross product of the first two columns.

Solving for pose (three-dimensional rotation and translation) requires the homography matrix from equation (4.2), and it can be estimated in two separate ways. Either by using the homography of a plane tracker, or by using corner points of the rectangular planar patch in both image coordinates and three dimensional coordinates.

4.2.1 Solving Pose by Knowing Tracker's Homography

I would highly recommend calculating the pose of the rectangular planar patch by using the homography of the tracker if available. Ensuring that compatible coordinate systems are used in all parts of equation (4.2) would make the matrix in that equation equivalent to the homography. That would mean that the homography commonly available from the tracker almost directly would give the pose of the tracked object when multiplied by \mathbf{K}^{-1} . Some steps shown in section 4.3 should however be applied to remove incorrect scaling and further inconsistencies in the parameters.

4.2.2 Solving Pose by Knowing Corner Points

The pose of the plane can also be estimated when the corner points of the rectangular planar patch in image coordinates and in three dimensional coordinates are known. The first step is to calculate the intermediate coordinates (x'_i, y'_i) from the image coordinates by using equation (4.3).

Then the homography can be calculated from the intermediate coordinates, allowing the pose to be estimated directly. First observe that equation (4.2) shows that each corner point gives an equation system like this:

$$\begin{cases} r_{31}X_i x'_i + r_{32}Y_i x'_i + t_z x'_i - r_{11}X_i - r_{12}Y_i - t_x = 0 \\ r_{31}X_i y_i + r_{32}Y_i y_i + t_z y_i - r_{21}X_i - r_{22}Y_i - t_y = 0 \end{cases} \quad (4.4)$$

Using all four corner points would generate eight equations with nine unknowns. It should be noted that the two columns from the rotation matrix are interdependent, making it possible to solve the equation system nevertheless.

The four corner points give four systems like (4.4) with a total of eight equa-

tions. They can be rewritten into the system (4.5):

$$\begin{bmatrix} -X_1 & -Y_1 & 0 & 0 & X_1 x'_1 & Y_1 x'_1 & -1 & 0 & x'_1 \\ 0 & 0 & -X_1 & -Y_1 & X_1 y'_1 & Y_1 y'_1 & 0 & -1 & y'_1 \\ \vdots & & & & & & & & \vdots \\ -X_4 & -Y_4 & 0 & 0 & X_4 x'_4 & Y_4 x'_4 & -1 & 0 & x'_4 \\ 0 & 0 & -X_4 & -Y_4 & X_4 y'_4 & Y_4 y'_4 & 0 & -1 & y'_4 \end{bmatrix} \begin{bmatrix} r_{11} \\ r_{12} \\ r_{21} \\ r_{22} \\ r_{31} \\ r_{32} \\ t_x \\ t_y \\ t_z \end{bmatrix} = \bar{\mathbf{0}} \quad (4.5)$$

This homogeneous system can be solved by performing a singular value decomposition, where the singular vector corresponding to the smallest eigenvalue corresponds to the correct solution. The solution yields the homography, which happens to be approximately equal (except scaling) to the pose parameters of the plane. Some steps shown in section 4.3 should however be applied to remove incorrect scaling and further inconsistencies in the parameters.

4.3 Post-processing the Solution

The solution yielding the camera's position relative to the plane in the real world have to be post processed to be of any value. As a start, the procedure from [4] was selected for constructing a valid solution and retrieving the full rotation matrix. As a second step, the calculated pose of the box was improved by minimizing the squared errors of the estimated patch corners.

4.3.1 Building a Valid Solution

The first step of post processing is to retrieve a valid solution. The steps described in [4] are followed. The unknown scaling of the homography is removed, and a valid rotation matrix is constructed. To remove the unknown scaling, the solution is scaled such that the two columns of the rotation vector yield unit length (approximately).

$$\begin{aligned} length &= \sqrt{\|\bar{\mathbf{r}}_1\| \|\bar{\mathbf{r}}_2\|} \\ \bar{\mathbf{r}}'_1 &= \bar{\mathbf{r}}_1 / length \\ \bar{\mathbf{r}}'_2 &= \bar{\mathbf{r}}_2 / length \\ \bar{\mathbf{t}}' &= \bar{\mathbf{t}} / length \end{aligned} \quad (4.6)$$

The second step before the solution could be considered valid is to ensure that the two calculated columns from the rotation matrix are perpendicular. This is done by first calculating

$$\begin{aligned} \overline{ctr} &= \bar{\mathbf{r}}'_1 + \bar{\mathbf{r}}'_2 \\ \overline{perp} &= \bar{\mathbf{r}}'_1 \times \bar{\mathbf{r}}'_2 \end{aligned}$$

$$\begin{aligned}\overline{dir} &= \overline{ctr} \times \overline{perp} \\ \overline{\mathbf{r}}_1'' &= \frac{\overline{ctr}}{\|\overline{ctr}\|} + \frac{\overline{dir}}{\|\overline{dir}\|} \\ \overline{\mathbf{r}}_2'' &= \frac{\overline{ctr}}{\|\overline{ctr}\|} - \frac{\overline{dir}}{\|\overline{dir}\|}\end{aligned}$$

and then normalizing $\overline{\mathbf{r}}_1''$ and $\overline{\mathbf{r}}_2''$ again. Finally, the third column of the rotation matrix is calculated as the cross product of the first two columns.

4.3.2 Minimizing Errors

Certain types of errors caused unwanted deviations in the solution yielded by the procedure above. The deviations could be caused either by errors in the coordinates in the 3D-world, by errors in the estimated screen coordinates of the corners of the tracked patch, or by problems with the algorithm itself. The sum of squared reprojection¹ errors for the models 3D-points was selected for minimization. Minimizing that measure produced a sounder solution than the initial estimate. The conjugate gradient method was chosen to do the minimization, because it converged faster than the steepest descent method during practical use.

To be able to more effectively minimize the problem, six parameters for determining the pose of the rectangular planar patch in the real world was isolated. Those parameters consisted of the three Euler angles describing the rotation matrix, and all three values of the translation vector.

4.4 An Alternative Way to Track the Pose of a Plane

Sections 4.2-4.3 described ways to recover pose information using a homography between certain three dimensional coordinates for the patch and the image coordinates for the patch. Those homographies describes the mapping of pixels between any two planes, and have eight degrees of freedom.

When a pinhole camera model is used, the perspective mapping of a plane is however limited to six degrees of freedom. In applications where the pose of a three-dimensional plane is tracked, the two additional degrees of freedom of homography trackers can turn out to be an actual disadvantage. The additional degrees of freedom in homographies make it possible for the tracked patch to be distorted in ways which makes it more difficult to determine the pose of the plane.

Some effort was made to find a simple optical flow tracker, which directly would track the pose of the plane. When no result was achieved that way, a homography tracker was modified to attain two wanted properties: restricting the freedom of the homography, such that the patch is consistent with a pinhole camera model; and enable the tracker to be queried about the pose directly. The underlying mathematics relies on equation (4.1), from which the homography can be isolated

¹reprojected 3D-coordinates into image coordinates

as:

$$H = K \begin{bmatrix} | & | & | \\ \bar{\mathbf{r}}_1 & \bar{\mathbf{r}}_2 & \bar{\mathbf{t}} \\ | & | & | \end{bmatrix}$$

Considering the change of the pose as estimated by the tracker to consist of a small change of rotation and an entirely new translation part:

$$H' = K \begin{bmatrix} | & | & | \\ \bar{\mathbf{r}}'_1 & \bar{\mathbf{r}}'_2 & \bar{\mathbf{t}}' \\ | & | & | \end{bmatrix} = K \left[\begin{bmatrix} 1 & -\alpha_3 & \alpha_2 \\ \alpha_3 & 1 & -\alpha_1 \\ -\alpha_2 & \alpha_1 & 1 \end{bmatrix} \begin{bmatrix} | & | \\ \bar{\mathbf{r}}_1 & \bar{\mathbf{r}}_2 \\ | & | \end{bmatrix} \begin{bmatrix} | \\ \bar{\mathbf{t}}' \\ | \end{bmatrix} \right]$$

The value of those equations would become clear as it can be seen that the translational part of the pose is given directly, while the incremental change in rotation can be determined by solving an over determined equation system:

$$\begin{bmatrix} 0 & r_{31} & -r_{21} \\ -r_{31} & 0 & r_{11} \\ r_{21} & -r_{11} & 0 \\ 0 & r_{32} & -r_{22} \\ -r_{32} & 0 & r_{12} \\ r_{22} & -r_{12} & 0 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix} = \begin{bmatrix} r'_{11} - r_{11} \\ r'_{21} - r_{21} \\ r'_{31} - r_{31} \\ r'_{12} - r_{12} \\ r'_{22} - r_{22} \\ r'_{32} - r_{32} \end{bmatrix}$$

The restriction of freedom for the homography tracker was conducted by determining the initial pose of the plane before tracking began, and then determine changes of the pose by looking at the incremental change of the homography. The restriction of the homography into a valid pinhole camera transformation is then carried out by recomputing the homography using the pose of the tracked plane.

Unfortunately, the current implementation of this alternative tracker performed much worse than plane trackers using homographies. If more work was put into this tracker, I believe that that might change.

Chapter 5

Tracking of a 3D-box

This chapter of the thesis is devoted to tracking of three dimensional boxes. Tracking was conducted by keeping separate plane trackers for each of the six sides of a box. Only one tracker at a time would be used for tracking, and a model of the box would then be used to determine the pose¹ of all other sides as well as the pose of the box. The calculated pose would then be used to select the most appropriate side to track in the next tracking iteration.

5.1 Box Model

The box is simply modelled as depicted in figure 5.1. The model has six sides, each associated with a separate tracker. The physical dimension of each tracked patch is given by the dimensions of that side, but the number of pixels used by the tracker is currently not determined before the tracker is created. The reason behind that was to create a tracker with as little interpolation as possible from the camera image used for initialising it.

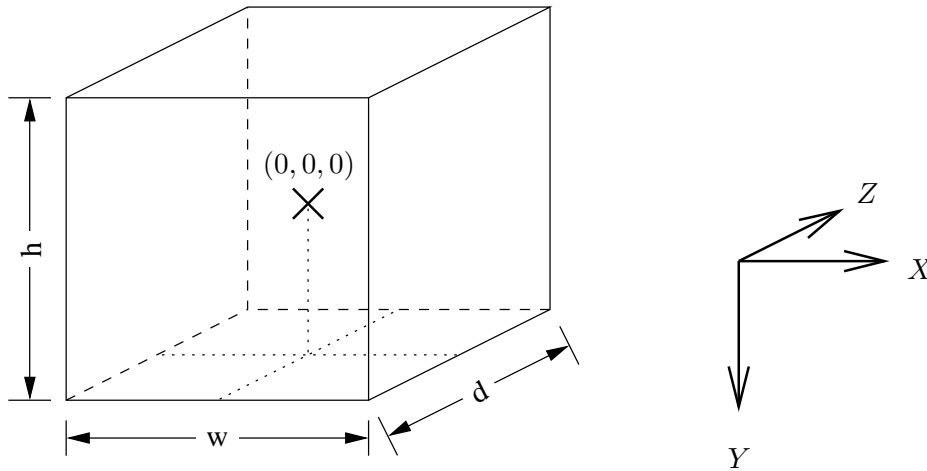


Figure 5.1: Box model. The origo for the box model is in the centre of the box.

¹To know the pose would mean that both position and orientation is known in the three dimensional space.

One important role of the model is to enable transformation of the pose of one of the sides of the box into the pose of the box. It is equally important to be able to retrieve the pose of an arbitrary side of the box from the pose of the box. Without those transformations, it would not be possible to change the tracked side of the box.

The model of the box could beside the measurements of the box also contain the textures for the sides of the box, but knowledge of the texture is not yet used. The currently implemented box tracker is initialised with the dimensions of the box and the four corner points of the front surface, and it then gathers textures for the other sides of the box as the box is rotated. It would be much easier to reliably track the box if the textures of its sides would be known in advance. Without any help from the textures, and without any help from structure-from-motion algorithms, deviations from the box can be seen when the box tracker switches to track other sides than the first side.

5.2 Individual Trackers

All six individual trackers would use optical flow tracking capable of tracking perspective distortions. Each tracker delivers a homography relating the tracked patch in image coordinates and real world coordinates. That homography is then used to estimate the pose of the tracked planar patch (see chapter 4).

One problem for the trackers was changing illumination, as the tracked side of the box might have to be tracked during a 90 degree rotation. The first countermeasure was to compensate for uniform changes in illumination (see section 2.4.2). This proved to be insufficient during some circumstances, especially if reflections occurred on the tracked side.

A successful ad hoc solution was to add background subtraction capabilities to the plane trackers. Normally, background subtraction would involve continuously updating the background image. When a plane is tracked, that would however become completely unnecessary since the supposed intensity values of the plane ideally would be known and constant at all times.

In the beginning of any tracking iteration, the intensity discrepancy as seen by the tracker would be due to deviations between the new state of the tracker and the last known state of the tracker. The threshold for a pixel could then be set to the absolute value of the gradient magnitude, as that corresponds to the largest expected intensity change for any single pixel displacement. To allow discrepancies larger than one pixel, that threshold was basically multiplied by some small factor. Another small factor was additionally added to allow for other types of intensity variations. There was not much experimenting to find the best strategy of thresholding, as the first tries went way better than anticipated.

5.3 Tracker Handover

The side of the box, which is facing the camera the most, is currently used as the tracked side. If a tracker handover should occur, it would happen as the last step

in a tracking iteration. That means that the previously tracked side just have been tracked, and the image used for tracking is used for constructing a new tracker for the next side (if that tracker was not previously initialised). When trackers hand over to each other, the most important part is that they use the model of the box to correctly determine the corner positions of the new tracker.

Previously, the side with the largest area as seen from the camera was used, but that suffered a few shortcomings in real life. The most evident drawback of using the area to decide when to shift the tracked face occurred for boxes with unequal sides. The area of the smaller side would not become larger then the area of the tracked side before excessive perspective distortions made tracking unreliable.

5.4 Discussion

There are possibly some additional improvements that could be added to this box tracker. First of all, thresholding with the purpose of removing reflections and reduce the impact of severe illumination changes did work beautifully. The thresholded image might be improved by erosions² and dilutions³ designed to remove isolated thresholded points, and removing isolated holes in regions that should be thresholded.

Another feature that would be useful in most practical situations would be the ability to initialise the trackers for all sides of the box in advance. Currently only the first face of the box is initialised, and the other sides are captured and initialised as the box rotates. The drawback of using the current tracker and box model to initialise new trackers is error propagation. It is currently not possible to rotate the box 270 degrees without loosing track, but that might possibly be partially remedied by choosing a better box with a higher contrast pattern.

²Erosion is an operation on binary images removing edge pixels and thereby shrinking all areas.

³Dilation is an operation on binary images adding edge pixels and thereby enlarging all areas.

Chapter 6

Implementation

This chapter briefly describes some of the most useful C++ classes developed during this work. The source code is heavily documented such that doxygen¹ can generate useful documentation.

6.1 Image and Video

Classes developed for image and video manipulation are described in this section. They include the classes `Text` and `Line` that store texts and lines, and they can be used for drawing their content into images. The class `SImage` stores images and contain many image manipulation functions. The header file for that class also contain several template functions for manipulating images whose pixels are stored sequentially in ordinary arrays of arbitrary types. The classes `VideoSink` and `VideoSource` are used for writing and reading video sequences to and from files on hard drives. The last two classes in this section, `GrabberThread` and `MyScheduler`, are both useful when developing applications where image acquisition is performed by a separate thread.

6.1.1 Text

The `Text` class is used for writing strings into images. It contains a font for the first 128 characters (i.e. some Swedish/German characters are missing), but it is still very useful. Typical usage involves writing names or numbers of trackers on top of displayed images.

Example Text usage

```
int w=320; int h=200;
unsigned char* image = new unsigned char[w*h];

Text t("This should get drawn with value 255");
t.draw(w/10, h/2, image, w, h, 255);
```

¹Doxygen is a widely used documentation generator, similar to javadoc but supporting much more languages.

6.1.2 Line

The Line class represents line segments by their two endpoints. The main usage of the class is to draw lines (both solid and dotted). This class performs clipping of lines outside the destination image before drawing, and is hence safe to use with unreliable input. This class additionally contains a limited set of geometric methods. Those methods can determine geometric length, normalized direction vector, angle of line, and distance to the line (when both endpoints extended infinitely).

6.1.3 SImage

SImage is a basic greyscale image class, with the capability of referencing small windows of other images without copying any data. The same header file also contains many useful image manipulation functions which operate on ordinary unsigned char* images, as well as some template functions which can operate on two-dimensional arrays of arbitrary types.

6.1.4 VideoSource

The VideoSource class is a wrapper interface, which provides a simple way of using the frame grabber, files, fifos, and other video sources.

The constructor `VideoSource(string pathname, int* w, int* h, int framerate)` can be used to choose input from the grabber card if the pathname is an empty string. If the pathname is a directory, all files in that directory are read as pgm-files in alphabetical order. If pathname specifies a multiple-frame pgm-file, that file would get used as the video source. The YUV4MPEG2 format is also detected and can be used. There is also support for reading data from both fifos as well as files, for instance to pipe images from mplayer² into some application using VideoSource.

The width and height parameters are passed to the grabber card to select the capture resolution. If the video is read from file(s), the width and height passed to the constructor gets overwritten with the values from the file used for the first frame.

The framerate parameter is used to select framerate for other sources then the grabber card. The implementation of framerates relies on usage of the RTC of the computer, and it must be configured to allow use of higher user interrupt frequencies, for instance by appending “echo 1024 > /proc/sys/dev/rtc/max-user-freq” to the start up script `rc.sysinit`.

```
Example VideoSource usage
/** Program that joins all files found in a directory to a
 * single multi-frame pgm.
 * @author Simon Gustafsson */

#include <iostream>
#include "VideoSource.hh"
#include "VideoSink.hh"

using namespace std;
```

²MPlayer is an open source media player.

6. Implementation

```
int main(int argc, char* argv[]){
    if(argc!=3){
        cout << "Usage: " << argv[0]
            << " input_directory_or_file output_file.pgm" << endl;
        return 1;
    }

    int w=640;
    int h=480;
    string destpath = argv[2];

    VideoSource* s =new VideoSource(argv[1],&w, &h);
    unsigned char* data = new unsigned char[w*h];
    VideoSink* sink=new VideoSink(destpath,w,h);

    if(!sink->OK()){ return 1; }

    while((sink->OK()) && (!s->isLast())){
        s->getGreyImage(data);
        sink->add(data,w,h);
    };

    if(!sink->OK()){ return 2; }

    delete sink;
    delete[] data;
    return 0;
}
```

Besides using image sequences captured with the grabber card, it is possible to convert movies into the appropriate pgm-format or YUV4MPEG2 format if mplayer is installed on the system. To extract all frames from an arbitrary movie as pgm-images, go into the destination directory, and run “mplayer /path/filename.avi -nosound -vo pnm:pgm” in the directory you would like to save all frames in. To convert all those images into a single multiple-frame pgm-file, use the example application above.

6.1.5 VideoSink

The VideoSink class stores a sequence of images as a multiple frame pgm-file. For a usage example, see the example code in section 6.1.4.

6.1.6 GrabberThread

The GrabberThread class is a thread class, which can run concurrently with the main application. Its sole purpose is to grab frames from files or video devices. All grabbed frames are stored in a circular buffer to allow the main thread to lag behind for short periods of intense processor usage.

```
Example GrabberThread usage
int w=320; int h=240;
GrabberThread gt(w,h,"video_source_file.pgm",25);
unsigned char* buff; //Should not be allocated separately
gt.start();
while(!gt.isLast()){
    if(gt.hasFrames()){ //Do we have a new frame to process
        buff=gt.getNextBuffer(); //Get buffer of new frame

        //maybe do some preprocessing of buffer
    }
}
```

```
    }

    //do part of what we want to do with buffer

    /* reschedule (we don't want to use the entire timeslice, since it is *
     * too long). Not rescheduling might also prevent GrabberThread from *
     * being picked to buffer frames, which could lead to dropped frames. */
    sched_yield();
}
gt.join();
```

6.1.7 MyScheduler

MyScheduler is a class used for cooperative multitasking. An application using it can be found in section 7.2.2. This class contains round robin queues of all callbacks that should be scheduled, and the main loop of applications should call this class repeatedly to run the next item in the queue. It contains one queue for items that should be executed only once for each new frame, and a separate queue for items which can run repeatedly.

6.2 Feature Detectors

6.2.1 HoughTransform

The HoughTransform class performs Hough transforms to detect line segments. If line segments are repeatedly computed in a window of known size, several values are precomputed to speed up the line detection process. More information about the Hough transform can be found in appendix E.2.

6.3 Tracking and Clustering

Classes developed for tracking and clustering are described in this section. The class LineTracker is a tracker for tracking line segments in images. It is followed by two classes, which use SSD tracking, namely SSDSlowTracker and SSDShort-CircuitTracker. Those classes are followed by classes for optical flow tracking. FlowTrackerCore is useful for developing entirely new optical flow trackers, and the FlowTracker class contain abilities for tracking patches with several different motion models. The last class in this section is the PerspectiveTracker class, which uses a cluster of several affine trackers to track perspective distortions.

6.3.1 LineTracker

The LineTracker class is used for basic line tracking, and can track both high contrast lines as well as keep track of their extension. Appendix E.1 describes how it works in more detail.

6.3.2 SSDSlowTracker

The SSDSlowTracker class implements a basic SSD³ tracker. This tracker also happens to be the computationally most expensive of the trackers implemented, hence the name SSD*Slow*Tracker. Its use should be limited to comparison against other trackers. The tracker can only track translations of the tracked patch, and is designed to track square patches of size $2R + 1$, where R can be seen as the radius of the patch. The principle behind it is described in section 2.1.

```
Example SSDSlowTracker usage
int w,h; int radius=20;
VideoSource g("video.pgm", &w, &h);

/* Allocate buffer _AFTER_ construction of VideoSource, since      *
 * width and height altered to match values in the file "video.pgm". */
unsigned char* image = new unsigned char[w*h];

g.getGreyImage(image); //get first frame

/* Create square tracker with centre at (w/2, h/2), *
 * and sides of length 2*radius+1 */
SSDSlowTracker ssd(image, w, h, w/2, h/2, radius);
ssd.setMaxSearchDev(5); //Setting max searching distance to 5 pixels

while(!g.isLast()){        // repeat until last frame
    g.getGreyImage(image); // Get next frame
    ssd.track(image,w,h); // use SSDSlowTracker to track patch
}
```

6.3.3 SSDShortCircuitTracker

The SSDShortCircuitTracker class is also an implementation of a basic SSD⁴ tracker. Contrary to the SSDSlowTracker, this tracker uses loop short-circuiting (described in section 2.2) to reduce the computational load while tracking. Basic usage of this class is identical to basic usage of the SSDSlowTracker class. The example code for SSDSlowTracker (section 6.3.2) applies for this class as well after changing the tracker class.

6.3.4 FlowTrackerCore

The FlowTrackerCore class contains the most important data structures and algorithms needed to perform optical flow tracking. It is purposely designed to be “universal” such that as many different optical flow trackers as possible could use this for their tracking.

It can handle weighting, and evaluation of parameters in either a single step, or in two steps. It can also perform basic thresholding to handle occlusions or reflections better if thresholds are provided. It simply evaluates optical flow parameters by using motion templates passed to this class. Keeping track of the current state of a tracker or higher level functions are not performed by this class, and should be handled by the classes which are using this class.

³Sum of Squared Differences

⁴Sum of Squared Differences

Two feature of this class makes it special: it can be switched between several types of tracking by the user, and its storage type is specified as a template parameter.

Interestingly, more motion templates can be added after initialisation, resulting in a possibility to enhance tracking capabilities as time goes by. The idea was to make it possible to have a separate thread that first sets up a translation tracker, and then adds the ability to handle scaling, rotation, up to perspective transformations when the CPU is idle. That additional thread was however not implemented.

6.3.5 FlowTracker

The FlowTracker class was implemented to simplify everyday use of the FlowTrackerCore class. It presents a simple interface, common to all area trackers developed here. It can very simply be configured to track anything from translations up to perspective distortions. Currently this class performs one tracking iteration by beginning to track at the coarsest level, and going through all scales up to and including the finest before returning (when -1 supplied to the track() function).

6.3.6 PerspectiveTracker

The PerspectiveTracker class demonstrates clustering of several trackers, and allows tracking of perspective transformations. The individual trackers would not have been able to do that without treating them as a cluster. See section 3.2.4 for more information.

6.4 Timing

This section describes classes related to timing. The first class described, the RTC class, is designed for generating periodic interrupts faster than the system clock, to allow more precise control over when functions are brought back from sleep. The other two classes described here, Stopwatch and MeasureCycles, are designed for measuring time with very high precision.

6.4.1 RTC

The RTC class waits for periodic interrupts from the Real Time Clock (RTC). The RTC class is used by the GrabberThread class (see section 6.1.6), which is responsible for serving frames as similar as possible to a live system when stored data is used instead of live input. That required that a thread could be woken up from sleep at regular intervals, at least up to 30 times per second. Kernels using a 100Hz software clock (default for 2.4.x kernels) would be unable to wake up sleeping processes with a resolution better than 10 ms, which would mean that the available processing time for each frame would fluctuate by as much as $\pm 33\%$. The RTC class makes it possible to circumvent that problem, and makes it possible to sleep for shorter periods of time even if the kernel is unable to do that.

Different kernels use different default system clocks, and the situation for x86 based kernels follows: the default for 2.4.x was 100Hz; the default starting from

2.6.0 was 1000Hz; from 2.6.13, the software clock is a kernel configuration parameter which could be 100, 250 (the default) or 1000. Additionally, some vendors have adjusted the system clock to other values as well. For kernels with fast system clocks, `nanosleep` would be suited for sleeping with a high accuracy⁵, but other kernels requires other means such as using the RTC to be able to sleep with a high resolution.

A few details have to be explained. To begin with, the class uses `/dev/rtc` to be able to receive periodic interrupts, and that requires that `/proc/sys/dev/rtc/max-user-freq` is set to a high enough value. That can be achieved by putting `echo 1024 > /proc/sys/dev/rtc/max-user-freq` into `/etc/rc.local` or any similar startup script. The “max-user-freq” set previously determines the highest frequency of periodic interrupts that an application running in user space can request. The RTC class has been verified to function correctly with periodic interrupts up to 8192Hz, but for normal video processing, values of 256Hz or 512Hz would be more suitable. As a side note, the class resorts to using `nanosleep` when `/dev/rtc` is unavailable, or the requested interrupt frequency was higher then `max-user-freq`.

6.4.2 Stopwatch

The Stopwatch class simplifies measuring of execution time for selected pieces of code. It is designed to be used for code called repeatedly. Code called only once should preferably be measured by using the `MeasureCycles` class instead, as there is no need to gather statistics for non-repetitive functions.

The Stopwatch class returns four different parameters for each stopwatch: Its name, and its minimum, mean, median and maximum execution time in either clock cycles or milliseconds. Three properties of this class should be kept in mind when using it:

Firstly, since the class provides statistics for median execution times, it has to store all individual timings. That would require allocation of more and more memory whenever the last allocated buffer gets filled. So Stopwatch should not be used in code designed to run indefinitely.

Secondly, timing should not be nested, in the sense that one timer measures execution of some block of code containing another timer. If that would be done, reallocation of the array containing all timings would now and then add irrelevant clock cycles to the code executed and measured.

Thirdly, to make the measurements as easy as possible to add to existing code, a decentralized way to identify and keep track of all timings was employed. Strings passed to their constructors identify the stopwatches, and that identity is kept even when a stopwatch is destroyed and created several times. Measurements between the start of and the stop of a stopwatch can be printed either as the number of clock cycles spent in execution, or as the number of used milliseconds.

Example Stopwatch usage

```
Stopwatch t1("int=int*int");  
Stopwatch t2("float=float*float");
```

⁵man `nanosleep` currently incorrectly states a 10 ms resolution, it really is 1/"software clock"

6. Implementation

```
int int1=5; int int2=6; int int3=7;
float float1=5.01; float float2=5.02; float float3=5.07;

for(int i=0; i<50; i++){
    t1.start();
    for(int j=0; j<100; j++){ int1=int2*int3; }
    t1.stop();
    t2.start();
    for(int j=0; j<100; j++){ float1=float2*float3; }
    t2.stop();

    Stopwatch t3("Overhead"); // Works, even though created and destroyed
    t3.start();                // 50 times before the results are printed.
    t3.stop();
}
cout << Stopwatch::getStat() << "\n" << t1.getStat_as_ms() << "\n";
```

6.4.3 MeasureCycles

The MeasureCycles class measures CPU clock cycles using the “rdtsc” instruction available in protected mode for Pentium Pro and later and in all newer AMD processors.

Since this instruction executes in parallel with other instructions, some other people prefer to execute a cpuid instruction before and after executing each rdtsc instruction, to wait for the processor to completely finish all current instructions in the pipelines before issuing the rdtsc instruction. That adds a significant overhead, and it was decided that it would be better to repeat the measurements, then to add an overhead of several hundred clock cycles.

```
Example MeasureCycles usage
/* To measure cycles during 1 second, to be able to calculate timing
 * in milliseconds (only done for first instantiation of the class). */
MeasureCycles c;

/* Alternative initialisation to specified core frequency without
 * measuring, avoiding the 1-second delay otherwise imposed. */
// MeasureCycles c(2000000000);

typedef MeasureCycles::ull ull;

ull start, stop;
start = c; //equivalent to c.getCycles();
call_to_function_to_measure();
stop = c;

cout << "Clocks used: " << start-stop
      << " approximate time: " << c.as_ms(start-stop) << "ms\n";
```

The added overhead of measurement on my compiled version⁶ was never lower than 28 cycles, but would occasionally get as high as 255 cycles. Those fluctuations are mostly related to caching, pipelining, branch prediction, out-of-order execution, and other features of the CPU. For very long measurements, scheduling and interrupts are going to be a problem, making it difficult to measure anything else than the total time of execution. In those occasions, measurements of max,

⁶Amd 64 3000+ socket 754, gcc version 3.3.2 20031022, profiling not enabled

min, median and average time makes more sense, and the Stopwatch class could be used for that.

For my CPU, the overhead equates to measurement errors normally close to 14 ns, occasionally as high as 128 ns. As a comparison, gprof which could be used to measure execution time of functions, uses a sampling period of 10 ms, resulting in errors up to 10 ms or 20 000 000 cycles for one-shot measurements on the same CPU. In addition to that, the overhead from enabling profiling for very short functions is sometimes even larger than the execution time of the functions themselves.

6.5 NCurses

It was discovered that one large bottleneck of many tracking implementations was their slow output of status information to the terminal. One promising solution would be to use ncurses to output that information to the terminal, instead of having output written to the scrolling terminal handled by the X server. After some basic ncurses functionality was implemented, it was however discovered that a large part of the issue was caused by excessive use of “`cout << ... << endl;`”. The `endl` was not only causing a new line to be written, but also flushed all buffered data to the screen, instead of handling writes to the screen at more appropriate occasions. The simplest way around it would be to always use “`cout << ... << “\n”;`” instead.

By the time of that discovery, a working quite transparent wrapper for use of ncurses had been implemented. The remaining increase of performance for applications using this output method caused ncurses support to stay in the project.

The class `OutputWrapper` is the central class for a programmer adding ncurses support. It can be used to redirect `cout` and `cerr` to an ncurses display, and if the output from the program already is displayed as a few lines with repetitive information, only a command to flush output to the screen has to be issued each time the virtual page should get written to the physical screen. See section 6.5.1 for an example.

6.5.1 OutputWrapper

`OutputWrapper` is a class that simplifies writing to the terminal without scrolling, with simple switching between normal output to `stdout` and output to an ncurses terminal. The most practical aspect is the ability to steal `cout` and `cerr`, redirecting their output to the ncurses screen without having to care for anything else. The only thing to remember is to replace “`... << endl`” with “`... << “\n”`” in all outputs, or calling “`((OutputStreambuf*) rdbuf())->disableSync()`” for the `OutputWrapper` class, to avoid updating the screen when not wanted.

Example `OutputWrapper` usage

```
#include "OutputWrapper.hh"
#include <iostream>

using namespace std;
```


6. Implementation

```
int main(){
    OutputWrapper out;

    //redirect cout and cerr to OutputWrapper
    out.setMode(OutputWrapper::NCURSES | OutputWrapper::STEAL_BOTH);

    //process data, use paged output
    static int max_bottles=100; //=INT_MAX;
    for(int i=max_bottles; i>0;i--){
        cout << i
            << " bottles of beer on the wall " << "\n"; //redirected above
        cerr << i << " bottles of beer. "; //also redirected
        out << "Take "; //always available
        if(i==1) cout << "it"; else cout << "one";
        cout << " down and pass it around - " << "\n";
        if(i==1) {cerr << "NO MORE bottles of beer on the wall!!"; }
        else { cout << i-1 << " bottles of beer on the wall. "; }
        out.newPage(); //flush page to screen and clear temporary page
        sleep(1);
    }
}
```

6.5.2 OutputStreambuf

OutputStreambuf is an implementation of a Streambuf class, which is used to output data to ncurses windows. Direct use of this Streambuf is discouraged, as the OutputWrapper allows easier interfacing, as well as the ability to directly use io-manipulators when outputting data through that class.

The OutputStreambuf can however be initialised to output data only to a specific rectangular area of the screen, but even then it should preferably be wrapped by using the Streambuf as an argument to the constructor of an OutputWrapper. Note that each window has to be flushed to the screen separately. Also note that ncurses behaviour is unspecified when overlapping areas are used, so overlapping between different windows should never be forced by a user.

6.5.3 NCursesUser

NCursesUser is a helper class that should be inherited by classes using ncurses, to provide a callback which is called when ncurses is enabled or disabled globally, and optionally to provide a callback for terminal resize events

Make sure that the callback functions is not using NCursesClass in such a way that infinite loops could occur, at least by not letting the NCURSES_OFF() and NCURSES_ON() call NCursesClass::forceMode().

```
Example derivation from NCursesUser
#include "OutputWrapper.hh"

class OneNCursesUser : private NCursesUser{
public:

    OneNCursesClass(){
        //the callback functions are registered automatically just
        //before the body of the constructor begins to execute.

        if(NCursesClass::isActive()){
            //do something special if ncurses active
        }else{
            //avoid doing bad things if ncurses not activated yet.
        }
    }
}
```

6. Implementation

```
    }  
}  
  
private:  
    //Callback functions  
    void NCURSES_OFF(){  
        //put the class into some state avoiding use of ncurses functions  
    }  
    void NCURSES_ON(){  
        //put the class into some state allowing use of ncurses functions  
    }  
    void NCURSES_RESIZE(){  
        //when this is called, the terminal has been resized.  
        int newx = NCursesClass::getWidth();  
        int newy = NCursesClass::getHeight();  
    }  
};
```

6.5.4 NCursesClass

NCursesClass is a helper class for coordination between NCURSES, Output-Streambuf and OutputWrapper. It should normally not be used by application programmers directly.

6.6 Additional Supporting Classes

This section described some useful support classes developed during this work. They contain a very simple command line argument parser named ArgumentParser, classes for reading and writing data to m-files (for use by Octave or Matlab), and a class that simplifies testing for Ctrl-C key presses.

6.6.1 ArgumentParser

The ArgumentParser is a basic class, which facilitates parsing of command line arguments. It provides parsing of integers, floating point numbers, switches and string arguments. Switches can be either 0 or 1, depending on their absence or presence at the command line. All other variables keep their values when they are absent on the command line, providing an easy implementation of default values. The class automatically documents all possible command line arguments and their default values. Additional documenting strings can be inserted to further describe the use of command line arguments.

```
Example ArgumentParser usage  
#include "ArgumentParser.hh"  
#include <iostream.h>  
  
using namespace std;  
  
int main(int argc, char* argv[]){  
  
    int on;           //binary switches can not have default values  
    int off;  
    int one_int = 2;           //default value  
    double one_double = 3.2;  //default value  
    string one_string = "";    //default value  
    string unparsed_1 = "";    //default value
```

6. Implementation

```
string unparsed_2 = "some default string"; //default value

ArgumentParser p;
p.addHelp("This is some test program for argument parsing\n");
p.addHelp("Try it with some of these binary switches:");
p.addSwitch("-on",&on,"To turn something on...");
p.addSwitch("-off",&off,"To turn something off...");
p.addHelp("\nTry some other switches, expecting numbers and strings:");
p.addInt("-one_int",&one_int,"Stored into local variable one_int");
p.addDouble("-one_double", &one_double,"This is some number" );
p.addString("-one_string", &one_string,"comment...");
p.addHelp("\nIt can catch arguments not preceded by tokens:");
p.addUnparsed(&unparsed_1,"only this has comment, not the other");
p.addUnparsed(&unparsed_2);

if(argc==1){
    cout << p.getHelp() << endl;
    exit(0);
}
p.parse(argc,argv);

cout << "on=" << on
    << "\noff=" << off
    << "\none_int=" << one_int
    << "\none_double=" << one_double
    << "\none_string=" << one_string
    << "\nunparsed_1=" << unparsed_1
    << "\nunparsed_2=" << unparsed_2
    << endl
    << "-----"
    << endl
    << "parse errors: \n"
    << p.getErrors()
    << endl;
}
```

6.6.2 MFileReader

The MFileReader class is used for reading numbers stored as vectors in m-files⁷. It has no knowledge of any other aspect of Octave then vectors, and will not be able to process files containing commands, or any other type of matrices then one-dimensional vectors.

The class is designed to completely ignore everything after the line

```
%====[end of data (MFileReader won't read past this line)]====
```

which automatically got added by MFileWriter. Below that line, any kind of commands or even garbage can be inserted without interfering with MFileReader.

Example MFileReader usage

```
MFileReader m("results.m");

for(int i=0; i<m["x"].size(); i++){
    cout << "x[" << i << "]=" << m["x"][i] << ", "
        << "xx[" << i << "]=" << m["xx"][i] << endl;
}
vector<double> x = m["x"]; //to improve performance, get access
                           //to the vector directly, instead of
                           //having to look up the vector
                           //through m["x"] during each access.
```

⁷m-files are files containing Matlab (or Octave) data and commands.

6.6.3 MFileWriter

The MFileWriter class provides a way of writing vectors to an m-file. The write to the specified file is postponed until the destruction of the class, and until then all added vectors are stored in memory. The vectors to be written are stored one number at a time as can be seen in the example code below.

It is also possible to add any number of arbitrary strings, which will get appended to the end of the m-file. That facilitates writing applications, which in addition to storing valuable data, also could append all commands needed to process and evaluate that data in either Matlab or Octave.

```
Example MFileWriter usage
MFileWriter m("results.m",
               "%example of squaring a number, where xx(i)=x(i)*x(i)");
for(int i=1; i<4; i++){
    m.addToVector("x",i);
    m.addToVector("xx",i*i);
}
m.appendString("deviations=xx-x.*x;");
m.appendString("max_abs_deviations=max(abs(deviations))");
m.appendString("plot(x, xx, \";computer;\", x, x.*x, \";octave;\");");
```

The code above would create this m-file:

```
results.m
%Example of squaring a number, where xx(i)=x(i)*x(i)

%=====[start of data]=====

x = [1;
2;
3];

xx = [1;
4;
9];

%====[end of data (MFileReader won't read past this line)]====

%=====[Appended commands]=====

deviations=xx-x.*x;
max_abs_deviations=max(abs(deviations))
plot(x, xx, ";from computer;", x, x.*x, ";according to octave;")

%=====[end of appended commands]=====
```

Running the m-file in Octave would produce the expected plot, as well as printing the line `max_abs_deviations = 0`, indicating that the algorithm in octave gave the same answer as the algorithm of the C program. This example might seem trivial, but the principle is very useful for checking implementations first written in Octave (or Matlab) against their reimplementations in C.

As a side note: For those using Octave, the installation of octave-forge is recommended, as this adds many additional commands. This class does not depend of the presence of any commands at all, but users of this class might want to use functions not yet included in the standard Octave distribution.

6.6.4 MY_CTRL_C_TRAPPER

MY_CTRL_C_TRAPPER is a class that intercepts the SIGINT signal (Ctrl-C), and makes testing for it quite easy. If desired, it can be initialised to call the previous SIGINT handler after a specified number of SIGINT signals.

```
Example MY_CTRL_C_TRAPPER usage
MY_CTRL_C_TRAPPER ctrlc(3); //intercept Ctrl-C 3 times before calling
                             //previous SIGINT handler.
                             //Use 0 to never call previous SIGINT handler.
for(int i=0; i<INT_MAX; i++){
    sleep(1);
    if(ctrlc.pressed()){ break; }
}
```

6.7 Deprecated Classes

This section describes some of the classes, which for various reasons were not completely finished or got replaced by other classes but still deserve a special mention in the report.

6.7.1 SMatrix

This class is deprecated – use arrays or CASMatrix instead.

There was a number of reasons why something else then CASMatrix⁸ was wanted for matrix calculations. The first reason for developing SMatrix was that CASMatrix lacked the ability to define the storage type of its elements. Other reasons included the inefficiencies in the multiplication routine in CASMatrix (see section D.2.1), the poor performance when handling small objects (see section D.2.2), the fact that CASMatrix by default always is compiled with optimisations disabled, and that licensing restrictions⁹ prohibits distribution of CASMatrix in source format.

The most valuable insight from the development of this class might be the optimised multiplication routine. Some comparative measurements¹⁰ can be found in tables 6.1 and 6.2. Note that the performance increase of using SMatrix::mul_and_assign with optimisations over using CASMatrix::operator* without optimisations is 9.5 times faster execution when multiplying 10x10 matrices, and that the difference grows slightly for larger matrices. Hopefully the lessons learned could be integrated into CASMatrix, as suggested in section D.2.

The reasons for deprecating this class in this project was that matrix inverse and singular value decomposition never got implemented, resulting in the need of going through CASMatrix for that functionality. Another reason was that the data was stored according to the less normal FORTRAN convention. The choice of the less popular FORTRAN ordering made the class more efficient in this particular project, where more matrices with many rows was used then matrices with

⁸the already developed candidate in NOMAN, which is a local source code repository.

⁹CASMatrix uses some code from “Numerical recipes”, which cannot be redistributed freely in source format.

¹⁰All Measurements in core cpu cycles on an Athlon 64 3000+ with 512 KB L2-cache.

Table 6.1: Comparison of multiplication when using *SMatrix* or *CASMatrix*, when both compiled with optimisations enabled (`-O2 -DNDEBUG`). Values in parenthesis compare speed with that of *CASMatrix*, and the other value is the measured median number of clock cycles for the multiplication.

Compiled with optimisations enabled			
matrix dimension	<i>CASMatrix</i> ::operator*	<i>SMatrix</i> ::operator*	<i>SMatrix</i> ::mul_and_assign
3x3	1346 (100%)	1292 (104%)	378 (356%)
5x5	3269 (100%)	1697 (193%)	822 (398%)
10x10	17849 (100%)	7907 (226%)	5938 (301%)
100x100	13244775 (100%)	3490105 (380%)	3414891 (388%)
1000x1000	(100%)	(479%)	(480%)
3333x3333	(100%)	(612%)	(614%)

Table 6.2: Comparison of multiplication when using *SMatrix* or *CASMatrix*, when both compiled without optimisations enabled. Values in parenthesis compare speed with that of *CASMatrix*, and the other value is the measured median number of clock cycles for the multiplication.

Compiled with optimisations disabled			
matrix dimension	<i>CASMatrix</i> ::operator*	<i>SMatrix</i> ::operator*	<i>SMatrix</i> ::mul_and_assign
10x10	56425 (100%)	23657 (239%)	21323 (265%)
100x100	50330099 (100%)	16990850 (296%)	16810706 (299%)
1000x1000	(100%)	(430%)	(453%)
3333x3333	(100%)	(496%)	(492%)

many columns. Yet another reason for not using this class was that lazy evaluation to remove unnecessary use of temporary objects (see section D.2.2) was not implemented. Without lazy evaluation, users of this class would have to call many different specialized functions of the class to get even more additional performance increases, instead of being able to write arithmetic matrix expressions as usual.

6.7.2 SSDFlowTracker

This class is deprecated – use FlowTracker instead.

The *SSDFlowTracker* class is an optical flow tracker. The class can track two dimensional translations, translations combined with rotation, or full affine motion.

6.7.3 SSDFlowTrackerRect

This class is deprecated – use FlowTracker instead (unless the dynamic selection of proper pyramid levels is very important).

The *SSDFlowTrackerRect* class can track two dimensional translation, translations combined with rotation, or even full affine motion. This class dynamically selects a proper pyramid level based on how large the deviation of the tracker is

between each new frame. Additionally, it treats iterated tracking of the same frame specially: it can track at finer pyramid levels while iterating, and switch back to the coarser scale when a new frame is received.

6.7.4 ClusterPoints

This class do not provide a suitable API for use beyond experimentation.

ClusterPoints was a helper class used when testing clustering of points based on equidistance in subsequent frames. The code in `testClustering.cc` could be examined for usage hints and code from that file should preferably be transferred to the ClusterPoints class. Additionally, ClusterPoints currently only provides text output of identified clusters, which is of little use beyond testing.

Chapter 7

Experimental Evaluation

These experiments were carried out on a computer with the following specifications:

Table 7.1: Specification of test system

CPU	AMD Athlon™64 Processor 3000+ (2GHz core, 512 KB cache)
RAM	1024 MB PC3200
MBoard	Abit NF8

When the code was compiled and linked with profiling information and analysed with gprof, the resolution was not accurate enough. Therefore, all time measurements below were collected using the `rdtsc` instruction and knowledge of the core frequency. The Stopwatch class described in section 6.4.2 was constructed to handle these time measurements.

7.1 Evaluation of Area Trackers

This section briefly describes the results of measurements performed while evaluating the area trackers.

7.1.1 Time Consumption

These measurements allow comparison of time consumption to be performed between some of the implemented trackers. It also shows the difference between interpolation types, as trackers using nearest neighbour (NN) interpolation were measured separately from those who use linearly interpolated values (IP). The measurements could also be used to determine the maximum number of simultaneous trackers allowed.

Time measurements for any tracking iteration for the optical flow tracker FlowTracker are located in table 7.2. Table 7.3 contains measurements for SSDSlowTracker, and table 7.4 have measurements for a short-circuited tracker.

These results could be summarised by stating that optical flow trackers by far outperform the other correlation based SSD trackers in term of computational efficiency. It is also clear that the impact of interpolating pixel values instead of

7. Experimental Evaluation

Table 7.2: Time for tracking measurements for a FlowTracker with dimension 81x81, using every fourth pixel only in each direction

Tracker type	Warp type ^c	Optimised ^a (ms)	Not optimised ^b (ms)
Translation	NN	0.0502	0.1063
	IP	0.0606	0.1183
Translation & rotation	NN	0.0722	0.1347
	IP	0.0816	0.1528
Affine (two step)	NN	0.0857	0.1768
	IP	0.0956	0.1949
Perspective (one step)	NN	0.0980	0.2273
	IP	0.1042	0.2382

^aMedian time for tracking if compiled with optimisations enabled “-O2”

^bMedian time for tracking if compiled without optimisations

^cNN = Nearest Neighbour’s value, IP = Interpolated pixel value

Table 7.3: Time for tracking measurements for an SSDSlowTracker with dimension 81x81

Max search distance (pixels)	Optimised ^a (ms)	Not optimised ^b (ms)
5	2.584	17.28
10	9.743	65.16

^aMedian time for tracking if compiled with optimisations enabled “-O2”

^bMedian time for tracking if compiled without optimisations

Table 7.4: Time for tracking measurements for an SSDShortCircuitTracker with dimension 81x81

Max search distance (pixels)	Optimised ^a	min time (ms)	average time (ms)	max time (ms)
5	YES	0.472	2.655	5.38
	NO	1.165	6.938	15.51
10	YES	1.089	5.885	12.89
	NO	2.634	15.22	31.89

^aWhether class compiled with optimisations enabled “-O2”

using nearest neighbour interpolation is very small, so there is no real reason to avoid using the better interpolation. The only real surprise was that short-circuited tracking in some circumstances performed worse than SSD tracking without short circuiting, but only when both were compiled with optimisations enabled.

7.1.2 Deviation from Expected Position

Tests for measuring the deviation of trackers from their supposed position has been constructed. Those tests used an image displaced with known displacements, and the positions of trackers following parts of the image were saved to be processed later. More about those experiments can be found in the later parts of section 2.4.1.

7.1.3 Step Response

This section will demonstrate how a small perturbation is handled by the tracker. It is a good “sanity-check” of trackers, and can find errors such as incorrect centre-points of trackers or other initialisation errors. By design, optical flow trackers should respond very quickly to changes as long as the changes are small.

The optical flow tracker was repeatedly tracking an identical image. The tracker was however displaced one pixel in x and y direction from its initial position in the first iteration, so the results demonstrates how the tracker converges towards its initial position. Figure 7.1 shows a typical convergence back to the original position. For some patches, the tracker converges back to the initial position. For some other patches, such as the one used in figure 7.1, the tracker instead converges to a position very close to the original position. Note that the tracked image was downscaled, so the deviation as perceived by the tracker is 0.1 pixels, not 0.2 pixels.

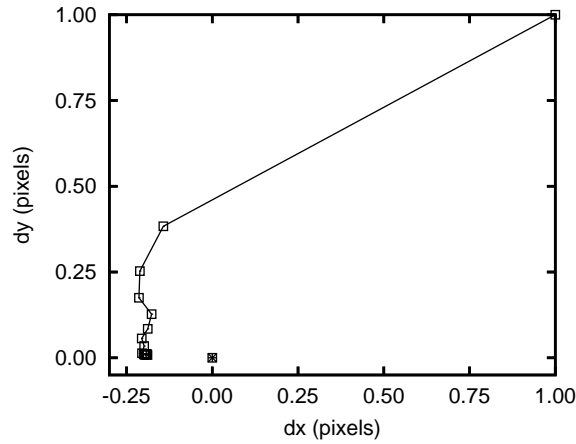


Figure 7.1: Typical path of tracker converging back toward true position after large displacement (affine tracker using linear interpolation in warp, downscaling 1).

The distance from the initial position to the tracker’s current position is also plotted in figure 7.2. That figure clearly shows the difference between using nearest neighbour interpolation and linear interpolation. Note that the tracked image was downscaled by the factor two, so the deviation as seen from the tracker is half of the values in those plots.

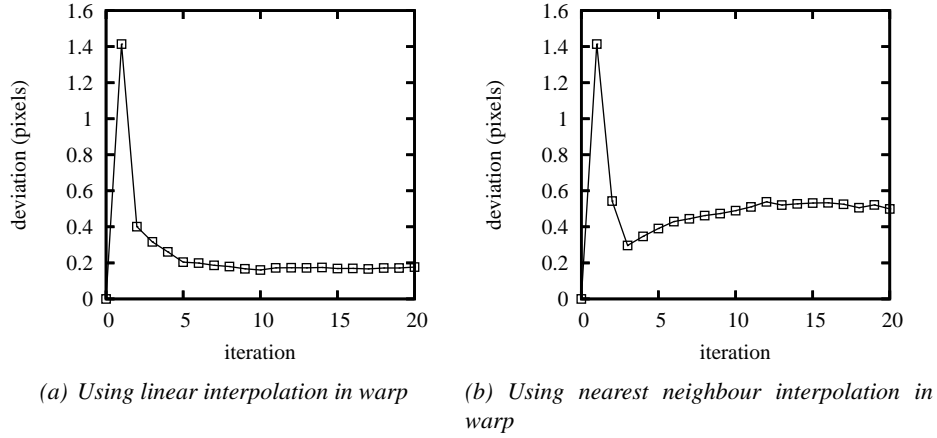


Figure 7.2: Deviation in pixels plotted against tracking iteration for tracker whose position was forced off target. The two figures show the difference caused by different means of warping.

7.1.4 Extension of Range for Optical Flow Trackers

Since optical flow trackers by design is limited to tracking relatively small motions, a few ways to make them handle larger motions could be desired. A couple of methods extending their range are described here:

Subsampling the image to track is very common in combination with these trackers. Subsampling by a factor always increases the trackable range by the same factor. At the same time, subsampling reduces the amount of information in the patch, making it easier to loose the tracked patch when too little useful information remains in the patch.

Blurring the image to track is another way to increase the trackable range. It works by removing high contrast areas in the images, and produces larger areas with gradients. Unfortunately, it does not provide a directly proportional relation between change of parameter and estimation of parameter for the full range of trackable motion. The lack of proportionality is however not a deal breaker, as iterated tracking still yields the correct result fairly quickly.

A third way of increasing the trackable range is by making the differentiating kernel wider. If no blurring were applied, this way of increasing the trackable range would maintain a directly proportional relationship between the applied displacement and the measured displacement. At the same time, this method is more sensitive to noise and experiences more difficulties with textured areas then trackers using the smallest differentiating kernel.

Figure 7.3 shows the relationship between a real-world displacement of a tracker, and the tracker's calculated displacement after one iteration. The influence of different amounts of blurring, and different lengths of the differentiating kernel can be seen.

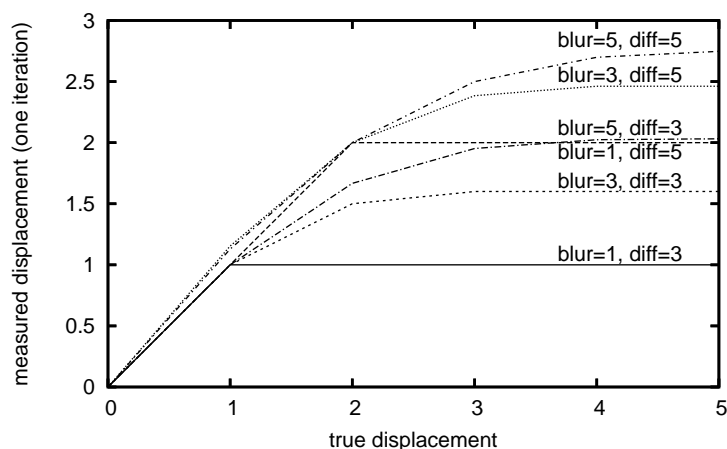
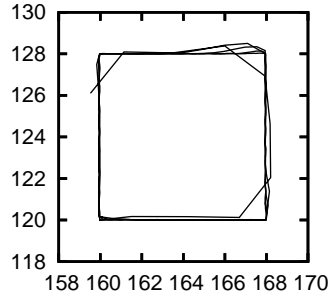


Figure 7.3: Measured displacement versus true displacement. “blur” denotes the width of the blurring kernel, and “diff” denotes the width of the differentiating kernel. The tracked object was a vertical edge.

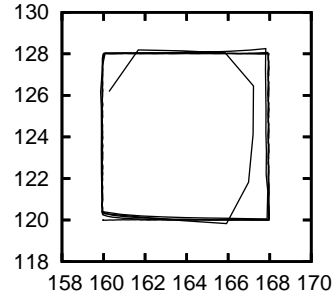
7.1.5 Behaviour at Different Tracking Speeds

Here, an image was translated in a square pattern for five revolutions. Each revolution was performed with increasing speed. The results (image 7.4) are very good at lower speeds (within fractions of a pixel). First at the highest speed, the tracker without prediction started to loose accuracy.

Experience shows that the more parameters a tracker tries to estimate, the worse it performs at these tests. The reason behind that is the interdependency between all parameters solved by the tracker, and the fact that small estimation errors are unavoidable. It would hence be inexpedient to use trackers tracking more complex motion models than necessary.



(a) Downscaling 2 and prediction used



(b) Downscaling 2 and no prediction used

Figure 7.4: Tracked centre coordinates of a patch in an image translated at speeds of 0.1, 0.25, 0.5, 1 and 2.2 pixels/frame. One square path was followed at each of those speeds, and the slowest speeds generated the paths most similar to a square. As expected, tracking starts to deteriorate at 2.2 pixels/frame, since that corresponds to movements >1 pixel/frame in the downscaled image.

7.2 Applications

7.2.1 Star Photography

One application taking advantage of sub-pixel resolution tracking could be photographing star constellations in the night sky. If that is done with an ordinary digital camera instead of through a telescope, the amount of light entering the camera could be too small to get sufficient image quality. One solution available besides acquiring a more sensitive camera includes using longer exposure times. A motorised support for the camera might then be called for, since the prolonged exposure time otherwise would blur the moving stars.

The solution tested in this section used a noisy low sensitivity camera¹ mounted on a standard tripod. About 130 consecutive pictures were taken of the night sky using eight second exposure, aperture f/2.8, and a sensitivity of ISO200. Those camera settings resulted in individual pictures where the brightest stars were visible, but many stars were not perceivable in the noise.

If a fixed camera were capturing several images of a static scene, it would be trivial to decrease the impact of noise by averaging a large number of captured images. When it comes to decreasing noise by averaging several images of a moving scene, the difficulties increase. First of all, one has to model the deviation of the objects from their original positions. Secondly, artefacts such as distortions introduced by the imaging process (most notably projective distortions, barrel distortion, and distortions caused by the atmosphere) might have to be dealt with. When the movements of the objects in the image frames are known, those objects could be warped back into their original positions in the first frame, making the process of averaging noise out of the images trivial.

The process employed to determine the motion of the sky is outlined in this section. The used model of the movements of stars was really simple: It is assumed that the only possible deviations of the stars are caused by rotation. In the first acquired image, bright spots were automatically selected as features for optical flow tracking. Those features were tracked, and only features equidistant to each other throughout all frames were used to reverse the motion of the stars in all frames. The required condition that the distance $d_{i,j}$ between $star_i$ and $star_j$ should be constant throughout all frames is valid for all motions that can be modelled as any combination of rotation and displacement. The motion of the equidistant stars reveals the centre of rotation, and then the angle of rotation is determined by following the star furthest away from the centre of rotation. The current implementation does however require an approximate centre of rotation to be entered manually to determine the true centre of rotation. After the motion of the stars was reversed in all images, all frames were summed together, resulting in a brighter image with a considerable decrease of noise. Much fainter stars than before could be detected in the enhanced image as seen in figure 7.5.

¹A Panasonic FZ20.

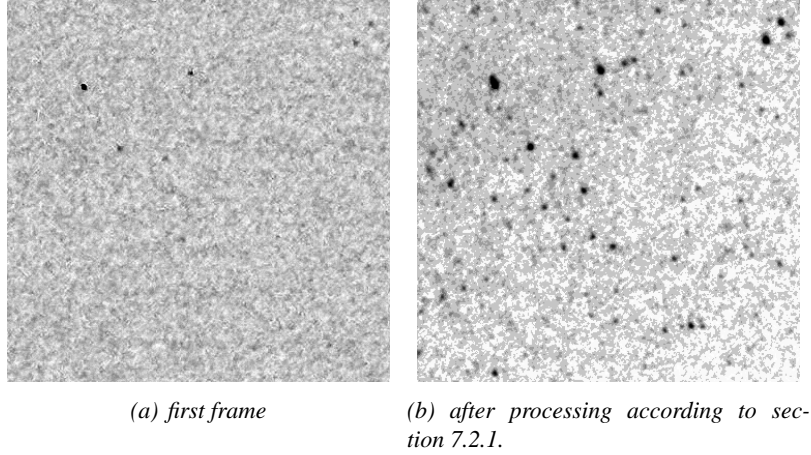


Figure 7.5: Virtually elongated shutter time by averaging several motion compensated exposures (100% crops of inverted images, with levels changed to reveal more details)

7.2.2 Iterative Tracking

Iterative tracking has the tendency of increasing the accuracy of tracking, especially when optical flow trackers are used. A test application for experimenting with iterative tracking was developed. The results of iterating tracking could be seen in figure 7.1. That figure indicates that iteration generates better results, especially when linear interpolation is used internally by an optical flow tracker.

The main goal of this application was to allow the trackers to be iterated as much as possible without having to drop frames, or making other applications unresponsive. To achieve that, a design with two separate threads was used. Both threads are depicted in figure 7.6.

The sole purpose of the first thread is to acquire images. That thread enters a sleeping state until a new image is captured. Currently, it works when using images stored on a hard drive, but whether it works with specific video hardware also depends on the kernel drivers for those particular capture devices. In general, the kernel should reschedule tasks whenever a thread waits for a device, so in general, the proposed solution should work.

The second thread of the application is responsible for processing all captured frames. The core idea is to use cooperative multitasking, where round robin is used to select which tracker to run next. The tracker tracking the longest time ago would always be the next one executed. Additionally, the kernel is given the opportunity to reschedule some other task every time a tracker has been executed. That would ensure that the frame capturing thread could be woken up without having to wait for an entire time slice to expire, at the same time as other tasks is given the opportunity to function even though close to 100% of the CPU is used for iterated tracking.

To prevent excessive waste of the CPU, there are methods that individual trackers could use to disable further iteration for itself until the next frame is received, but the current trackers has not been modified to use that possibility yet.

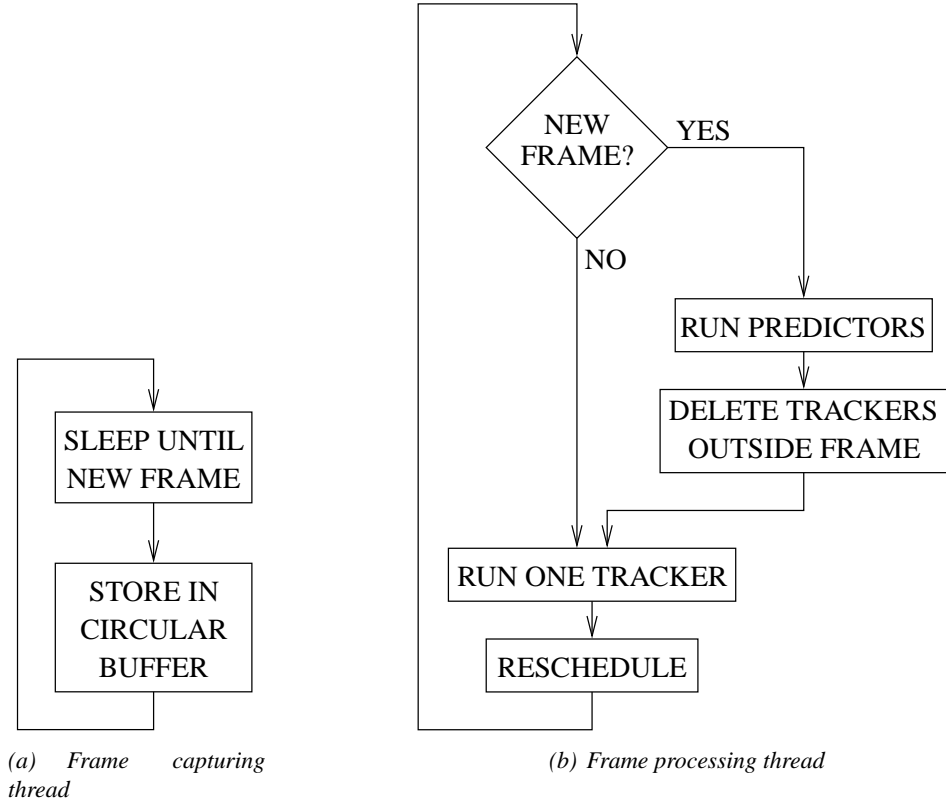


Figure 7.6: The two threads in the application

7.2.3 Tracking of Three Dimensional Boxes

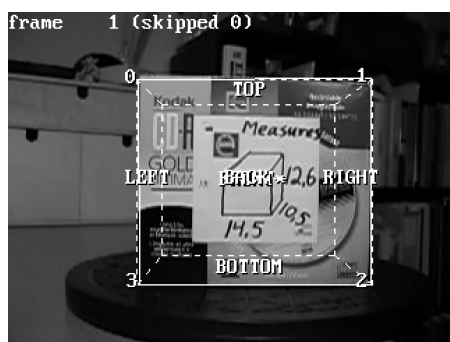
An application to test tracking of three dimensional boxes was created. The implemented box tracker was described in chapter 5, and some results from its evaluation are presented here.

The idea was to use a calibrated camera and a box of known dimensions, making it possible to determine the pose of the box by knowing the corners of one side of the box. In theory, it would be possible to track the box as it is rotated even if the tracker only was initialised with the first side of the box.

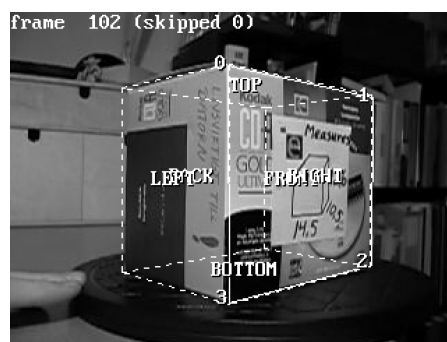
The optimistic attempt of only initialising the tracker with the first side of the box turned out to not work in practice. Figure 7.7 shows a few snapshots from these experiments. Figure 7.7(a) shows the box when it was initialised, and 7.7(b) shows the box right before using the box model to initialise the tracker for the left side. It can be seen that the deviation between the box model and the physical box still is small although it is present. Figure 7.7(c) shows a larger deviation between the physical box and the box model, which is propagated since the back tracker of the box now is going to be initialised in the wrong way. Figure 7.7(d) finally shows when the tracker (solid lines) starts to loose tracking completely.

It turned out that the somewhat optimistic goal of only having to initialise tracking of the first side of the box, and let the tracker automatically initialise the other sides whenever needed was not viable in practice. Small errors are amplified by

7. Experimental Evaluation



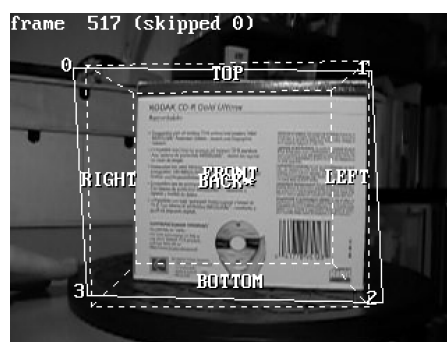
(a) Box and model, early frame.



(b) Box and model, before switching to left side.



(c) Box and model, before switching to back side.



(d) Box and model, about to loose track.

Figure 7.7: Tracking of a 3D box. Solid lines show the borders of the current tracker. Dotted lines show a projection of the box model.

perspective effects, and reliable tracking would require some method to ensure that a tracker is initialised correctly. I would recommend initialising all six sides of the tracker manually. The deviations between the tracked box and the box model would then be similar to the deviation in image 7.7(b). Alternatively, targets of known positions and dimensions could be added to all sides of the box, to allow automatic alignment of the box model before the box tracker initialises new trackers for new sides.

Chapter 8

Conclusions

8.1 Regarding Area Trackers

A variety of area trackers were implemented, and their time consumption was compared to each other in section 7.1. Optical flow trackers turned out to be the fastest trackers as expected. They are about 40 times faster than their corresponding SSD trackers, while providing resolution in the sub-pixel range. On the other hand, optical flow trackers have to be iterated somewhat when the tracker is jumping larger distances (see section 7.1.3) to settle over the best fit, but optical flow trackers are nevertheless far superior in regard of time consumption.

An interesting property of the optical flow trackers was the close to 10% difference in time consumption caused by two different ways of warping the input image, leading to a selection of the slightly slower interpolating warp algorithm causing a slight increase in precise tracking of small motions. Other possible choices of warping algorithms could have been nearest-neighbour warping, or two-pass warping.

It is also noteworthy that short-circuited SSD trackers exhibited improvements in their average tracking time over the slow SSD tracker when neither was optimised, whereas the slow SSD tracker sometimes is slightly faster than the short circuited tracker when both are compiled with optimisations enabled. This indicates that one might have to consider worse performance from the short-circuited tracker than expected at the beginning of this work. The short circuited tracker could however be preferred if the complexity of the design is not a threat, and spare computational time is useful for the application, even if only to decrease power consumption. In mission critical systems, the computational resources have to be large enough to handle the worst case of almost no difference in speed between using loop short-circuiting and ordinary SSD tracking.

8.2 Regarding Clustered Perspective Trackers

The idea of clustering several small trackers to track motion models which the individual trackers would not be capable of looked promising, but did not deliver any real performance increase. The test implementation only gave a 10% gain in

computation efficiency when affine trackers was used to track perspective distortions (section 3.2.4). I would therefore not recommend the added complexity of these clustered trackers over using perspective optical flow trackers directly.

8.3 Regarding Star Photography

Merging of several images of the night sky to reduce noise was tried because of a pure personal interest. Other noisy image sequences might be merged as well to reduce noise if a suitable selection of initial trackers could be done (the star photography application automatically selected which stars to follow). The result was a palpable decrease of image noise, and many stars previously invisible for the naked eye could be seen in the merged image.

The star photography application was also thought to provide excellent statistics of time consumption of different algorithms and functions. It turned out that `gprof`¹ only sampled which function the program was occupying every 10 ms, which was far too slow to generate useful statistics when each tracker finished in much less than a millisecond. The use of the `rdtsc` instruction was then investigated and made it possible to measure time consumption of sections of code at much higher accuracy, with a resolution² of a single CPU core clock cycle.

8.4 Regarding Tracking of Three Dimensional Boxes

An attempt to track the pose of three dimensional boxes was made without initialising more than the tracker for the front surface of the tracked box. During practical use, it appeared as if the deviation between the model and the physical box was doubled each time the system initialised a tracker for a new surface. That makes it necessary to either initialise all sides of a tracked box before tracking is started, or provide some other mean of the tracking system to determine the corner positions for each new side of the box.

8.5 Regarding Line Tracking and Detection

An edge tracker based on the approach in [9] was constructed. It was extended to be able to handle tracking of lines, and the ability to determine endpoints of the tracked line was also added (section E.1). Endpoint detection was not as reliable as desired, but worked nevertheless in many circumstances.

Additionally, a line detector using the Hough Transform was implemented. That detector precomputed and saved several intermediate values, such that repeated use of the detector for equally sized patches would be faster. It is also shown that detecting lines using the Hough transform for full frame video nowadays is very close to possible in real time even with unoptimised code. By using 8x8 patches, the number of additions to an accumulator was reduced 72 times for images with a resolution of 320x240 pixels (section E.2.1). Future work should be

¹see the `gprof` man pages for more information

²Although accuracy could be expected to be in the order of 28-255 clock cycles.

8. Conclusions

done to merge those individual segments into continuous line segments to gain real benefits in line detection.

Bibliography

- [1] S. Benhimane, E. Malis, “Real-time image-based tracking of planes using efficient second-order minimization”, *Proceedings of 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems*, September 2004, pp. 943-948
- [2] Jean-Yves Bouguet, “Camera Calibration Toolbox for Matlab®”, last visited 2007-02-22:
http://www.vision.caltech.edu/bouguetj/calib_doc/
- [3] Scott A. Brandt, Christopher E. Smith, Nikolaos P. Papanikolopoulos, “The Minnesota Robotic Visual Tracker: A Flexible Testbed for Vision-Guided Robotic Research”, *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, 1994, pp. 1363-1368.
- [4] David Claus, “Camera Location”, last visited 2007-01-17:
<http://www.robots.ox.ac.uk/~dclaus/cameraloc/cameraloc.htm>
- [5] Paul D. Fiore, “Efficient Linear Solution of Exterior Orientation”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol 23, no 2, February 2001 pp. 140-148
- [6] Agner Fog, “1. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms”, last visited 2007-02-20:
<http://www.agner.org/optimize>
- [7] James D. Foley et al., “Computer Graphics. Principles and Practice. 2nd Edition in C”, Addison-Wesley, 1996, ISBN 0-201-84840-6
- [8] Rafael C. Gonzalez, Richard E. Woods, “Digital Image Processing”, Second edition, Prentice-Hall, 2002, ISBN 0-201-18075-8
- [9] Gregory D. Hager & Kentaro Toyama, “X Vision: A Portable Substrate for Real-Time Vision Applications.”, *Computer Vision and Image Understanding*, 69(1):23-37, 1998.
- [10] G. Hager, P. Belhumeur, “Real-Time Tracking of Image Regions with Changes in Geometry and Illumination”, last visited 2007-01-17:
http://www.cs.brown.edu/courses/cs143/GuestLectures/Joe_Mundy/hager-belhumeur.pdf

BIBLIOGRAPHY

- [11] G.D. Hager, P.N. Belhumeur, “Efficient region tracking with parametric models of geometry and illumination”, *IEEE Trans. Pattern Anal. Mach. Intell.* v20 i10, October 1998, pp. 1025-1039.
- [12] Chris Harris and Mike Stephens, “A Combined Corner and Edge Detector”, *Alvey88*, pp. 147-152
- [13] R. Hartley and A. Zisserman, “Multiple View Geometry in Computer Vision”, Cambridge University Press, 2000, p. 22, ISBN 0-521-62304-9
- [14] D. Kragic and H. I. Christensen, “Tracking Techniques for Visual Servoing Tasks”, *Proceedings of the IEEE International Conference on Robotics & Automation*, April 2000, pp. 1663-1669.
- [15] J. Shi and C. Tomasi, “Good Features to Track”, *1994 IEEE Conference on Computer Vision and Pattern Recognition (CVPR’94)*, 1994, pp. 593 - 600
- [16] Bjarne Stroustrup, “Programmeringsspråket C++”, 3’rd edition, p.787-789, ISBN 0-201-67504-8
- [17] Roger Y. Tsai, Thomas S. Huang, “Estimating Three-Dimensional Motion Parameters of a Rigid Planar Patch, II: Singular Value Decomposition”, *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol ASSP-30, no 4, August 1982, pp. 525-534
- [18] “AMD Athlon™ Processor x86 Code Optimization Guide”, Publication #2207, Rev. K, Advanced Micro Devices, Inc. February 2002
- [19] “Software Optimization Guide for AMD64 Processors”, Publication #25112, Rev.3.06, Advanced Micro Devices, Inc. September 2005

Appendix A

Affine Approximations of Projective Transformations

This chapter describes how to derive affine approximations to perspective transformations. Affine transformations are capable of approximating the effects of a perspective transformation. The approximation works best for smaller areas, and the steps to derive the approximation around some arbitrary point x_0, y_0 are described below.

Consider a perspective transformation

$$\begin{bmatrix} w'x' \\ w'y' \\ w' \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

which we want to approximate as an affine transformation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \approx \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

The approximative affine transformation should have some properties in common with the perspective transformation:

- Equivalent midpoints x'_0, y'_0
- Identical behaviour in an infinitesimal surrounding to the midpoints \implies can be achieved by equating derivatives in that point

Those properties listed above gives us the following equations from the affine transformation:

$$\begin{aligned} \frac{\delta x'}{\delta x} &= a_{11} \\ \frac{\delta x'}{\delta y} &= a_{12} \\ \frac{\delta y'}{\delta x} &= a_{21} \end{aligned}$$

$$\begin{aligned}\frac{\delta y'}{\delta y} &= a_{22} \\ t_x &= x'_0 - (a_{11}x_0 + a_{12}y_0) \\ t_y &= y'_0 - (a_{21}x_0 + a_{22}y_0)\end{aligned}$$

Note that there are no higher order derivatives available from the affine transformation, telling us that the approximation problem can be seen as a first order Taylor series expansion of the perspective transformation.

Combining the equations above with the following equations obtained from the perspective transformation

$$\begin{aligned}\frac{\delta x'}{\delta x}(x'_0, y'_0) &= (h_{11} - x'_0 h_{31})/w'_0 \\ \frac{\delta x'}{\delta y}(x'_0, y'_0) &= (h_{12} - x'_0 h_{32})/w'_0 \\ \frac{\delta y'}{\delta x}(x'_0, y'_0) &= (h_{21} - y'_0 h_{31})/w'_0 \\ \frac{\delta y'}{\delta y}(x'_0, y'_0) &= (h_{22} - y'_0 h_{32})/w'_0\end{aligned}$$

would then give us an affine approximation of the perspective transformation valid in the point x'_0, y'_0 and its closest surrounding:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \approx \begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

where the parameters (given x_0, y_0) would take the following values:

$$\begin{aligned}w'_0 &= h_{31}x_0 + h_{32}y_0 + h_{33} \\ x'_0 &= (h_{11}x_0 + h_{12}y_0 + h_{13})/w'_0 \\ y'_0 &= (h_{21}x_0 + h_{22}y_0 + h_{23})/w'_0 \\ a_{11} &= (h_{11} - x'_0 h_{31})/w'_0 \\ a_{12} &= (h_{12} - x'_0 h_{32})/w'_0 \\ a_{21} &= (h_{21} - y'_0 h_{31})/w'_0 \\ a_{22} &= (h_{22} - y'_0 h_{32})/w'_0 \\ t_x &= x'_0 - (a_{11}x_0 + a_{12}y_0) \\ t_y &= y'_0 - (a_{21}x_0 + a_{22}y_0)\end{aligned}$$

Appendix B

Decomposition of a Projective Transformation

As mentioned in [13], a projective transformation can be decomposed into a chain of transformations:

$$H = H_S H_A H_P = \begin{pmatrix} s\mathbf{R} & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{pmatrix} \begin{pmatrix} \mathbf{K} & \mathbf{0} \\ \mathbf{0}^\top & 1 \end{pmatrix} \begin{pmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{v}^\top & v \end{pmatrix} = \begin{pmatrix} \mathbf{A} & v\mathbf{t} \\ \mathbf{v}^\top & v \end{pmatrix}$$

where $\mathbf{A} = s\mathbf{R}\mathbf{K} + \mathbf{t}\mathbf{v}^\top$, and \mathbf{K} is an upper triangular matrix fulfilling $\det(\mathbf{K}) = 1$.

That decomposition is valid as long as $v \neq 0$, and is unique as long as s is chosen positive. The algorithm chosen for this decomposition is described below:

1. if($h_{33} < 0$) $H = -H$
2. $v = h_{33}$
3. $\mathbf{v} = [h_{31} \ h_{32}]^\top$
4. $\mathbf{t} = \frac{1}{v} [h_{13} \ h_{23}]^\top$
5. $\mathbf{L} = s\mathbf{R}\mathbf{K} = \begin{pmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{pmatrix} - \mathbf{t}\mathbf{v}^\top$ (\mathbf{L} determined from known variables).
6. Writing out the newly acquired $\mathbf{L} = s\mathbf{R}\mathbf{K}$, we can set up an equation system:

$$\mathbf{L} = \begin{pmatrix} | & | \\ \mathbf{l}_1 & \mathbf{l}_2 \\ | & | \end{pmatrix} = s \begin{pmatrix} | & | \\ \mathbf{r}_1 & \mathbf{r}_2 \\ | & | \end{pmatrix} \begin{pmatrix} a & b \\ 0 & c \end{pmatrix}$$

Here, \mathbf{r}_i and \mathbf{l}_i are the two columns of the rotation matrix and the \mathbf{L} matrix of the equation above.

$$\begin{cases} s\mathbf{r}_1 a = \mathbf{l}_1 \\ s\mathbf{r}_1 b + s\mathbf{r}_2 c = \mathbf{l}_2 \end{cases}$$

Using $|\mathbf{r}_i| = 1$, we get

$$sa = |\mathbf{l}_1|$$

and hence

$$\mathbf{r}_1 = \frac{\mathbf{l}_1}{sa}$$

continuing, multiplying the second equation of the system to the left by \mathbf{r}_1^\top , using some properties of the rotation matrix, namely that $|\mathbf{r}_i| = 1$ and that the two columns are orthogonal (i.e. $\mathbf{r}_i^\top \mathbf{r}_j = 0$ when $i \neq j$), we get

$$sb = \mathbf{r}_1^\top \mathbf{l}_2$$

now, using

$$s\mathbf{c}\mathbf{r}_2 = \mathbf{l}_2 - sb\mathbf{r}_1$$

taking the norm of both sides gives

$$sc = |(\mathbf{l}_2 - sb\mathbf{r}_1)|$$

and we then also get

$$\mathbf{r}_2 = \frac{1}{sc}(\mathbf{l}_2 - sb\mathbf{r}_1)$$

and hence the rotation matrix has been determined. We also know the entries of the \mathbf{K} matrix up to scale, but since we know that $\det(\mathbf{K}) = ac = 1$, we can determine s and the correct \mathbf{K}

$$s = \sqrt{sa \ sc}$$

$$\mathbf{K} = \frac{1}{s} \begin{pmatrix} sa & sb \\ 0 & sc \end{pmatrix}$$

All entries in the decomposition have now been determined.

Appendix C

Optical Flow Estimation

Construction of optical flow trackers relies on the *constant brightness assumption*. That assumption states that the brightness of any point of a tracked object should remain constant during tracking. Only the positions of the points are allowed to change. For an individual point with position $\bar{\mathbf{x}}(t)$ at time t in an image $\mathbf{I}(t)$, the constant brightness assumption for a single point can be expressed as (C.1):

$$\frac{dI(\bar{\mathbf{x}}(t), t)}{dt} = 0 \quad (\text{C.1})$$

The simplest and most straightforward way to derive the equations necessary for optical flow tracking does actually not include any equations derived from the SSD equation (2.1) directly. It does instead involve calculations of *ERR* (C.2), the summed squared deviation from the constant brightness equation (C.1) above. Note that when certain conditions¹ are met, the SSD equation (2.1) and the *ERR* equation (C.2) will give identical values. The weighting function $w(\bar{\mathbf{x}})$ is inserted to accommodate for weighting of pixels if needed.

$$ERR = \sum_{all \ \bar{\mathbf{x}}} \left(\frac{dI(\bar{\mathbf{x}}(t), t)}{dt} \right)^2 w(\bar{\mathbf{x}}) \quad (\text{C.2})$$

A thoroughly explained derivation of an optical flow tracker for tracking of translations will be found in the next section, and briefer examples for tracking more complex movements will be presented in the subsequent sections. [9] and [10] also contain examples of simple optical flow trackers.

All optical flow trackers in the following sections share the property that the equations for optical flow tracking was determined by differentiating equation (C.1), and then putting the result in the *ERR* equation (C.2). Then the minima of *ERR* was determined by differentiating *ERR* with respect to each parameter to determine, and then equating the resulting equations to zero. Solving the resulting equation system would give an approximation of the parameters which would minimise the *ERR* equation.

¹Requirements for ERR equal to SSD: Identical placement of the points used for evaluation of the sums; Time derivatives evaluated by comparison of first and current intensity for points; Omission of variable τ which would scale *ERR* if time between frames not set to 1; Use of identical weighting of pixels in both SSD and *ERR* calculation.

Coordinate Systems

One important property to consider about optical flow trackers is that they only can track small changes of the tracked patch relative to the used reference patch. That means that the camera image has to be warped using the last known tracked parameters into an image close to the reference image before tracking. Then the trackers resulting correction parameters would need to be warped before they applies to the patch as seen by the camera. Figure 2.1 (page 6) shows those coordinate system transformations together with additional transformations needed if the tracked patch is downscaled by the tracker.

C.1 Tracking Translations

This is a thoroughly explained derivation of a tracker capable of following translations. Starting by differentiating equation (C.1) yields

$$\frac{dI(\bar{\mathbf{x}}(t), t)}{dt} = \frac{\delta I(\bar{\mathbf{x}}(t), t)}{\delta x} \frac{\delta x}{\delta t} + \frac{\delta I(\bar{\mathbf{x}}(t), t)}{\delta y} \frac{\delta y}{\delta t} + \frac{\delta I(\bar{\mathbf{x}}(t), t)}{\delta t} \quad (\text{C.3})$$

Making the following substitutions in the formula above while letting τ denote the interval between consecutive frames

$$\begin{aligned} \frac{\delta I(\bar{\mathbf{x}}(t), t)}{\delta t} &\approx \frac{I(\bar{\mathbf{x}}(t + \tau), t + \tau) - I(\bar{\mathbf{x}}(t), t)}{\tau} \\ \frac{\delta x}{\delta t} &\approx \frac{\Delta x}{\tau} \\ \frac{\delta y}{\delta t} &\approx \frac{\Delta y}{\tau} \end{aligned}$$

yields

$$\begin{aligned} \frac{dI(\bar{\mathbf{x}}(t), t)}{dt} &\approx \frac{\delta I(\bar{\mathbf{x}}(t), t)}{\delta x} \frac{\Delta x}{\tau} + \frac{\delta I(\bar{\mathbf{x}}(t), t)}{\delta y} \frac{\Delta y}{\tau} + \\ &\quad \frac{I(\bar{\mathbf{x}}(t + \tau), t + \tau) - I(\bar{\mathbf{x}}(t), t)}{\tau} \end{aligned} \quad (\text{C.4})$$

Now an assumption that unfortunately limits trackable movements to less than one pixel (unless downscaling or excessive blurring is used) has to be done. It involves assuming that the intensities at frame $\mathbf{I}(t)$ is close to the intensities in frame $\mathbf{I}(t_0)$. This will be true as long as the tracker did not loose the tracked object in the last tracking iteration, and the brightness of the tracked points remained constant. The consequences of the assumption is that $I(\bar{\mathbf{x}}(t), t) \approx I(\bar{\mathbf{x}}(t_0), t_0)$, $\frac{\delta I(\bar{\mathbf{x}}(t), t)}{\delta x} \approx \frac{\delta I(\bar{\mathbf{x}}(t_0), t_0)}{\delta x}$, and that $\frac{\delta I(\bar{\mathbf{x}}(t), t)}{\delta y} \approx \frac{\delta I(\bar{\mathbf{x}}(t_0), t_0)}{\delta y}$. That makes it possible to rewrite equation (C.4) as

$$\begin{aligned} \frac{dI(\bar{\mathbf{x}}(t), t)}{dt} &\approx \frac{\delta I(\bar{\mathbf{x}}(t_0), t_0)}{\delta x} \frac{\Delta x}{\tau} + \frac{\delta I(\bar{\mathbf{x}}(t_0), t_0)}{\delta y} \frac{\Delta y}{\tau} + \\ &\quad \frac{I(\bar{\mathbf{x}}(t + \tau), t + \tau) - I(\bar{\mathbf{x}}(t_0), t_0)}{\tau} \end{aligned} \quad (\text{C.5})$$

Introducing some notations to make the expressions more compact

$$\begin{aligned}
 I_x &= \frac{\delta I(\bar{\mathbf{x}}(t_0), t_0)}{\delta x} \\
 I_y &= \frac{\delta I(\bar{\mathbf{x}}(t_0), t_0)}{\delta y} \\
 I &= I(\bar{\mathbf{x}}(t_0), t_0) \\
 I_{warp} &= I(\bar{\mathbf{x}}(t + \tau), t + \tau) \\
 w &= w(\bar{\mathbf{x}})
 \end{aligned} \tag{C.6}$$

makes it possible to write the *ERR* equation (C.2) as

$$ERR = \frac{1}{\tau} \sum_{all \bar{\mathbf{x}}} (I_x \Delta x + I_y \Delta y + I_{warp} - I)^2 w \tag{C.7}$$

To find the minimum of *ERR*, one has to differentiate it with respect to the variables to determine, namely Δx and Δy , and then set the resulting derivatives to zero as in equation (C.8). The variable τ was also set to 1 to facilitate comparison with results derived from the SSD equation.

$$\begin{aligned}
 \frac{\delta ERR}{\delta \Delta x} &= 2 \sum_{all \bar{\mathbf{x}}} I_x (I_x \Delta x + I_y \Delta y + I_{warp} - I) w = 0 \\
 \frac{\delta ERR}{\delta \Delta y} &= 2 \sum_{all \bar{\mathbf{x}}} I_y (I_x \Delta x + I_y \Delta y + I_{warp} - I) w = 0
 \end{aligned} \tag{C.8}$$

In the two equations above, only $(\Delta x, \Delta y)$ is unknown, and everything except I_{warp} was known when the tracker was constructed. One could rearrange those equations to

$$\sum_{all \bar{\mathbf{x}}} \left(w \begin{bmatrix} I_x I_x & I_x I_y \\ I_y I_x & I_y I_y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \right) = \sum_{all \bar{\mathbf{x}}} \left(w \begin{bmatrix} I_x (I - I_{warp}) \\ I_y (I - I_{warp}) \end{bmatrix} \right) \tag{C.9}$$

which also can be expressed as

$$A^T W A \mathbf{c} = A^T W \mathbf{b} \tag{C.10}$$

by selecting

$$\begin{aligned}
 A &= \begin{bmatrix} | & | \\ I_x & I_y \\ | & | \end{bmatrix} \\
 W &= \begin{bmatrix} \diagdown & & \mathbf{0} \\ & w & \\ \mathbf{0} & & \diagdown \end{bmatrix} \\
 \mathbf{c} &= \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} \\
 \mathbf{b} &= \begin{bmatrix} | \\ (I - I_{warp}) \\ | \end{bmatrix}
 \end{aligned}$$

The movement $\mathbf{c} = (\Delta x, \Delta y)^T$ can easily be solved as

$$\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \mathbf{c} = (A^T W A)^{-1} A^T W \mathbf{b}$$

The residual, which is the change of intensity which could not be explained by the solved parameters, can be computed as

$$\mathbf{r} = \sqrt{W} \mathbf{b} - (\sqrt{W} A) \mathbf{c}$$

From the residual, it is easy to calculate ERR , which is the summed squared deviation from the constant brightness equation:

$$ERR = \mathbf{r}^T \mathbf{r}$$

If it instead would be more interesting to calculate the SSD, that could be done in two slightly different ways.

Firstly, assuming that all pixels in the tracked patch on average would contribute with the same amount to the SSD, it could be evaluated as $SSD = \mathbf{r}^T \mathbf{r} / (\sum w)$. That method has the side effect that errors further away from the centre of the patch would not contribute as much to the SSD as errors close to the centre of the patch (with Gaussian weighting). Whether that is good or bad is difficult to determine, but as the error ERR which the tracker minimises is directly proportional to this way of calculating the SSD, it could be considered an advantage.

Secondly, it is also possible to determine the SSD by ignoring weighting at all, when the residual is calculated as $\mathbf{r} = \mathbf{b} - A\mathbf{c}$. That would make the calculation as simple as $SSD = \mathbf{r}^T \mathbf{r}$. That result could get normalized by dividing it by the number of pixels in the tracked patch if wanted.

Solution steps

Here, the precalculated matrices are $(A^T W A)^{-1} A^T W$, $\sqrt{W} A$, and \sqrt{W} . Note that W is diagonal, and the reduced number of computations needed when multiplying it with other matrices is used in the table. N in the tables is the total number of pixels in the tracked patch.

<i>Calculation</i>	<i>multiplications</i>	<i>additions</i>	<i>subtractions</i>
$\mathbf{b} = (I - I_{warp})$			N
$\mathbf{c} = ((A^T W A)^{-1} A^T W) \mathbf{b}$	$2N$	$2N - 2$	
$\mathbf{r} = \sqrt{W} \mathbf{b} - (\sqrt{W} A) \mathbf{c}$	$3N$	N	N
<i>total sum (excluding residual):</i>	$2N$	$2N - 2$	N
<i>total sum (including residual):</i>	$5N$	$3N - 2$	$2N$

It is quite notable that only one subtraction, two multiplications, and two additions per pixel are needed to solve for translation using optical flow.

C.2 Tracking Affine Transformations

In this section, equations for tracking an affine transformation are derived. Incremental changes to the affine transformation is described by two functions, f and g ,

such that the warping of points (u, v) into (u', v') are described by the following equation:

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} f(a_{11}, a_{12}, t_u, u, v) \\ g(a_{21}, a_{22}, t_v, u, v) \end{bmatrix} = \begin{bmatrix} a_{11}u + a_{12}v + t_u \\ a_{21}u + a_{22}v + t_v \end{bmatrix} \quad (\text{C.11})$$

Note that u and v denotes affine positions relative to the centre of the tracked patch in the initial frame. Most often the patch in the initial frame is considered to have an identity transformation, such that u and v coordinates coincides with x and y coordinates relative to the centre of the tracked patch. That would in turn make t_u and t_v relative to the (x, y) coordinate system as well.

Also note that the process used are going to solve for incremental changes, as the optical flow algorithm always compares the current frame with the frame just before the current frame. (Actually, assuming that the neither the tracker lost track of the target, nor that the constant brightness assumption broke, the frame just before the current frame could for our intents and purposes be considered identical to the very first frame).

Making the assumptions required to get equation (C.5), and using the notations introduced as equations (C.6) makes it possible to write ERR as

$$ERR = \frac{1}{\tau} \sum_{all \bar{\mathbf{x}}} (\mathcal{R} + I_{warp} - I)^2 w \quad (\text{C.12})$$

$$\text{where } \mathcal{R} = \left(I_u \Delta t_u + I_v \Delta t_v + u I_u \Delta a_{11} + v I_u \Delta a_{12} + u I_v \Delta a_{21} + v I_v \Delta a_{22} \right)$$

Here, I_u, I_v coincides with I_x, I_y if the patch in the initial system had its (u, v) system coinciding with the (x, y) coordinate system.

Setting τ to 1 and locating minimum of ERR by derivating (C.12) with respect to the parameters to find yields this equation system:

$$\begin{aligned} \frac{\delta ERR}{\delta \Delta t_u} &= 2 \sum_{all \bar{\mathbf{x}}} I_u (\mathcal{R} + I_{warp} - I) w = 0 \\ \frac{\delta ERR}{\delta \Delta t_v} &= 2 \sum_{all \bar{\mathbf{x}}} I_v (\mathcal{R} + I_{warp} - I) w = 0 \\ \frac{\delta ERR}{\delta \Delta a_{11}} &= 2 \sum_{all \bar{\mathbf{x}}} u I_u (\mathcal{R} + I_{warp} - I) w = 0 \\ \frac{\delta ERR}{\delta \Delta a_{12}} &= 2 \sum_{all \bar{\mathbf{x}}} v I_u (\mathcal{R} + I_{warp} - I) w = 0 \\ \frac{\delta ERR}{\delta \Delta a_{21}} &= 2 \sum_{all \bar{\mathbf{x}}} u I_v (\mathcal{R} + I_{warp} - I) w = 0 \\ \frac{\delta ERR}{\delta \Delta a_{22}} &= 2 \sum_{all \bar{\mathbf{x}}} v I_v (\mathcal{R} + I_{warp} - I) w = 0 \end{aligned}$$

This system can be written shorter as

$$A^T W A \mathbf{c} = A^T W \mathbf{b} \quad (\text{C.13})$$

where

$$\begin{aligned} A &= \begin{bmatrix} | & | & | & | & | & | \\ I_u & I_v & uI_u & vI_u & uI_v & vI_v \\ | & | & | & | & | & | \end{bmatrix} \\ W &= \begin{bmatrix} \diagdown & & \mathbf{0} \\ & w & \\ \mathbf{0} & & \diagdown \end{bmatrix} \\ \mathbf{c} &= \begin{bmatrix} \Delta t_u \\ \Delta t_v \\ \Delta a_{11} \\ \Delta a_{12} \\ \Delta a_{21} \\ \Delta a_{22} \end{bmatrix} \\ \mathbf{b} &= \begin{bmatrix} | \\ (I - I_{warp}) \\ | \end{bmatrix} \end{aligned}$$

and the solution would simply be

$$\mathbf{c} = (A^T W A)^{-1} A^T W \mathbf{b}$$

where $(A^T W A)^{-1} A^T W$ can be precomputed. The residual, which is the change of intensity which could not get explained by the solved parameters, can be computed as

$$\mathbf{r} = \sqrt{W} \mathbf{b} - (\sqrt{W} A) \mathbf{c}$$

From the residual, it is easy to calculate ERR , which is the summed squared deviation from the constant brightness assumption:

$$ERR = \mathbf{r}^T \mathbf{r}$$

Solution steps

Here, the precalculated matrices are $(A^T W A)^{-1} A^T W$, $\sqrt{W} A$, and \sqrt{W} . Note that W is diagonal, and the reduced number of computations needed when multiplying it with other matrices is used in the table. N in the tables is the total number of pixels in the tracked patch.

<i>Calculation</i>	<i>multiplications</i>	<i>additions</i>	<i>subtractions</i>
$\mathbf{b} = (I - I_{warp})$			N
$\mathbf{c} = ((A^T W A)^{-1} A^T W) \mathbf{b}$	$6N$	$6N - 6$	
$\mathbf{r} = \sqrt{W} \mathbf{b} - \sqrt{W} A \mathbf{c}$	$7N$	$5N$	N
<i>total sum (excluding residual):</i>	$6N$	$6N - 6$	N
<i>total sum (including residual):</i>	$13N$	$11N - 6$	$2N$

C.3 Tracking Affine Transformations (two step approach)

Tracking of affine transformations using the steps in the previous section are not considered to be the best way of tracking affine transformations in real life. [9] describes a different approach, where displacement and rotation are determined in a first step, before the other parameters are determined. The reason behind that is very simple.

During normal circumstances, displacement and rotation of the tracked object changes more often and faster than scaling and shear of the same object. At the same time, determination of scaling and shear suffer more easily from lack of details in the tracked patch, making them more prone to contain errors. If all the affine parameters were solved in one step, the errors introduced in the scaling and shear parameters would automatically introduce corresponding errors in the displacement and rotation parameters. To promote a better solution of displacement and rotation, the solution was split into two steps. The first step only solves for displacement and rotation, while the second step only solves for scaling and shear.

Construction of this two-step tracker begins by looking at the equation for this type of affine transformation. Normally the incremental transformation would be described by:

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = \Delta A \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, \text{ where } \Delta A = \begin{bmatrix} \Delta a_{11} & \Delta a_{12} & \Delta t_u \\ \Delta a_{21} & \Delta a_{22} & \Delta t_v \end{bmatrix}$$

This time, we consider rotation, scaling and shear separately. Rotation R can during small incremental updates be approximated linearly by ΔR :

$$\Delta R = \begin{bmatrix} 0 & \alpha & 0 \\ -\alpha & 0 & 0 \end{bmatrix}, \text{ where } \alpha \approx 0$$

Incremental scaling and shear could in turn be described as

$$\Delta S = \begin{bmatrix} s_u & \gamma & 0 \\ 0 & s_v & 0 \end{bmatrix}, \text{ where } s_u \approx 1, s_v \approx 1, \text{ and } \gamma \approx 0$$

And incremental translation could be described as

$$\Delta \mathbf{t} = \begin{bmatrix} 0 & 0 & t_u \\ 0 & 0 & t_v \end{bmatrix}$$

The incremental affine transformation described by $\Delta R + \Delta S + \Delta \mathbf{t}$ could thus be described by two functions, f and g , such that the warping of points (u, v) into (u', v') are described by the following equation:

$$\begin{bmatrix} u' \\ v' \end{bmatrix} = \begin{bmatrix} f(u, v, \alpha, s_u, \gamma, t_u) \\ g(u, v, \alpha, s_v, t_v) \end{bmatrix} = \begin{bmatrix} s_u u + (\alpha + \gamma)v + t_u \\ -\alpha u + s_v v + t_v \end{bmatrix} \quad (\text{C.14})$$

Differentiating $I(u', v') = I(f(u, v, \alpha, s_u, \gamma, t_u), g(u, v, \alpha, s_v, t_v))$ just as in equation (C.3), and using the notations introduced in equations (C.6) yields

$$\frac{dI}{dt} = \frac{dI(f(u, v, \alpha, s_u, \gamma, t_u), g(u, v, \alpha, s_v, t_v))}{dt} \approx$$

$$\begin{aligned}
 & \frac{1}{\tau} I_u (s_u \Delta u + (\alpha + \gamma) \Delta v + v \Delta \alpha + u \Delta s_u + v \Delta \gamma) + \\
 & \frac{1}{\tau} I_v (-\alpha \Delta u + s_v \Delta v - u \Delta \alpha + v \Delta s_v) + \\
 & \frac{1}{\tau} (I_{warped} - I)
 \end{aligned} \tag{C.15}$$

Here, I_u, I_v again coincides with I_x, I_y if the patch in the initial system had its (u, v) system coinciding with the (x, y) coordinate system.

Assuming small differential changes, we can set $s_u = 1, s_v = 1, \alpha = 0$ and $\gamma = 0$ to linearise the system around that point. Then inserting (C.15) into the *ERR* equation (C.2) while setting τ to 1 yields:

$$\begin{aligned}
 ERR = \sum_{all \bar{\mathbf{x}}} & \left(I_u (\Delta u + v \Delta \alpha + u \Delta s_u + v \Delta \gamma) + \right. \\
 & \left. I_v (\Delta v - u \Delta \alpha + v \Delta s_v) + \right. \\
 & \left. (I_{warped} - I) \right)^2 w(\bar{\mathbf{x}})
 \end{aligned} \tag{C.16}$$

Differentiating equation (C.16) with respect to each parameter to solve for yields this system for minimizing *ERR*.

$$A^T W A \mathbf{c} = A^T W \mathbf{b} \tag{C.17}$$

where

$$\begin{aligned}
 A &= \begin{bmatrix} | & | & | & | & | & | \\ I_u & I_v & (vI_u - uI_v) & uI_u & vI_v & vI_u \\ | & | & | & | & | & | \end{bmatrix} \\
 W &= \begin{bmatrix} \diagdown & & \mathbf{0} \\ & w & \\ \mathbf{0} & & \diagdown \end{bmatrix} \\
 \mathbf{c} &= \begin{bmatrix} \Delta t_u \\ \Delta t_v \\ \Delta \alpha \\ \Delta S_u \\ \Delta S_v \\ \Delta \gamma \end{bmatrix} \\
 \mathbf{b} &= \begin{bmatrix} | \\ (I - I_{warp}) \\ | \end{bmatrix}
 \end{aligned}$$

If the system is solved directly, *ERR* will be minimised, but then less than optimal values are more likely to be chosen for displacement and rotation. The reason is that scaling and shear are more prone to be solved slightly incorrect. When solving for those parameters and for displacement and rotation at the same time, the errors in scaling and shear would induce corresponding errors in displacement and rotation, thereby degrading their accuracy.

By solving the equation system in two steps, the accuracy of the displacement and rotation parameters can get enhanced. The first step solving only for displacement and rotation:

$$L^T W L \mathbf{d} = L^T W \mathbf{b} \quad , \text{resulting in} \quad \mathbf{d} = (L^T W L)^{-1} L^T W \mathbf{b}$$

where

$$L = \begin{bmatrix} | & | & | \\ I_u & I_v & (vI_u - uI_v) \\ | & | & | \end{bmatrix}$$

$$\mathbf{d} = \begin{bmatrix} \Delta u \\ \Delta v \\ \Delta \alpha \end{bmatrix}$$

Then the second step solves for scaling and shear while using the result from the first step above:

$$R^T W L \mathbf{d} + R^T W R \mathbf{e} = R^T W \mathbf{b}$$

$$\text{resulting in} \quad \mathbf{e} = (R^T W R)^{-1} R^T W (\mathbf{b} - L \mathbf{d})$$

where

$$R = \begin{bmatrix} | & | & | \\ uI_u & vI_v & vI_u \\ | & | & | \end{bmatrix}$$

$$\mathbf{e} = \begin{bmatrix} \Delta s_u \\ \Delta s_v \\ \Delta \gamma \end{bmatrix}$$

Solution steps

Here, the precalculated matrices are $(L^T W L)^{-1} L^T W$, L , $(R^T W R)^{-1} R^T W$, R , and \sqrt{W} . Note that \sqrt{W} is diagonal, and the reduced number of computations needed when multiplying it with other matrices is used in the table. N in the tables is the total number of pixels in the tracked patch.

<i>Calculation</i>	<i>multiplications</i>	<i>additions</i>	<i>subtractions</i>
$\mathbf{b} = (I - I_{warp})$			N
$\mathbf{d} = (L^T W L)^{-1} L^T W \mathbf{b}$	$3N$	$3N - 3$	
$M = L \mathbf{d}$	$3N$	$2N$	
$\mathbf{e} = (R^T W R)^{-1} R^T W (\mathbf{b} - M)$	$3N$	$3N - 3$	N
$\mathbf{r} = \sqrt{W} (\mathbf{b} - M - R \mathbf{e})$	$4N$	$2N$	$2N$
<i>sum (only translation and rotation):</i>	$3N$	$3N - 3$	N
<i>total sum (excluding residual):</i>	$9N$	$8N - 6$	$2N$
<i>total sum (including residual):</i>	$13N$	$10N - 6$	$4N$

Comparing the number of operations required for tracking between this tracker, and the other affine tracker in section C.2 reveals almost no difference in the number of computations when both the solution and the residual is computed. If the residual is not needed, the other tracker does however have noticeably fewer operations. Another interesting property made clear in the table is the low number of operations needed when solving for translation and rotation only.

C.4 Tracking Perspective Transformations

It is possible to extend optical flow tracking to be able to handle perspective transformations as well, as will be demonstrated in this section. A perspective transformation could be described by

$$\begin{aligned} \begin{bmatrix} u' \\ v' \end{bmatrix} &= \begin{bmatrix} f(m_{11}, m_{12}, m_{13}, m_{31}, m_{32}, m_{33}, u, v) \\ g(m_{21}, m_{22}, m_{23}, m_{31}, m_{32}, m_{33}, u, v) \end{bmatrix} \\ &= \begin{bmatrix} (m_{11}u + m_{12}v + m_{13})/(m_{31}u + m_{32}v + m_{33}) \\ (m_{21}u + m_{22}v + m_{23})/(m_{31}u + m_{32}v + m_{33}) \end{bmatrix} \end{aligned} \quad (C.18)$$

Note that u and v denotes affine positions relative to the centre of the tracked patch in the initial frame. Most often the patch in the initial frame is considered to have an identity transformation, such that u and v coordinates coincides with x and y coordinates relative to the centre of the tracked patch.

Also note that the nine entries m_{ij} only can be determined up to an unknown scale factor, making it possible to fix entry m_{33} to 1, while scaling the other entries accordingly (this requires that $m_{33} \neq 0$, if that can not be expected, an alternative would be to normalize M such that $\det(M) = 1$).

Making the assumptions needed earlier to get equation (C.5), and using notations introduced in equations (C.6) makes it possible to write ERR as

$$ERR = \frac{1}{\tau} \sum_{all \bar{x}} (\mathcal{R} + I_{warp} - I)^2 w \quad (C.19)$$

where

$$\begin{aligned} \mathcal{R} &= I_u \left(\frac{1}{C_3} (u\Delta m_{11} + v\Delta m_{12} + \Delta m_{13}) \right. \\ &\quad \left. + \frac{C_1}{C_3} (u\Delta m_{31} + v\Delta m_{32}) \right) \\ &\quad + I_v \left(\frac{1}{C_3} (u\Delta m_{21} + v\Delta m_{22} + \Delta m_{23}) \right. \\ &\quad \left. + \frac{C_2}{C_3} (u\Delta m_{31} + v\Delta m_{32}) \right) \end{aligned} \quad (C.20)$$

Here, I_u, I_v again coincides with I_x, I_y if the patch in the initial system had its (u, v) system coinciding with the (x, y) coordinate system. Also note that equation (C.20) above contains these abbreviations:

$$\begin{aligned} C_1 &= -(m_{11}u + m_{12}v + m_{13})/C_3 \\ C_2 &= -(m_{21}u + m_{22}v + m_{23})/C_3 \\ C_3 &= m_{31}u + m_{32}v + 1 \end{aligned}$$

where m_{ij} at initialisation usually would correspond to an identity matrix. Setting τ to 1 and locating minimum of ERR by derivating (C.19) with respect to the parameters to find yields this equation system:

$$\frac{\delta ERR}{\delta \Delta m_{11}} = 2 \sum_{all \bar{x}} \frac{I_u}{C_3} u (\mathcal{R} + I_{warp} - I) w = 0$$

$$\begin{aligned}
 \frac{\delta ERR}{\delta \Delta m_{12}} &= 2 \sum_{all \bar{\mathbf{x}}} \frac{I_u}{C_3} v (\mathcal{R} + I_{warp} - I) w = 0 \\
 \frac{\delta ERR}{\delta \Delta m_{13}} &= 2 \sum_{all \bar{\mathbf{x}}} \frac{I_u}{C_3} (\mathcal{R} + I_{warp} - I) w = 0 \\
 \frac{\delta ERR}{\delta \Delta m_{21}} &= 2 \sum_{all \bar{\mathbf{x}}} \frac{I_v}{C_3} u (\mathcal{R} + I_{warp} - I) w = 0 \\
 \frac{\delta ERR}{\delta \Delta m_{22}} &= 2 \sum_{all \bar{\mathbf{x}}} \frac{I_v}{C_3} v (\mathcal{R} + I_{warp} - I) w = 0 \\
 \frac{\delta ERR}{\delta \Delta m_{23}} &= 2 \sum_{all \bar{\mathbf{x}}} \frac{I_v}{C_3} (\mathcal{R} + I_{warp} - I) w = 0 \\
 \frac{\delta ERR}{\delta \Delta m_{31}} &= 2 \sum_{all \bar{\mathbf{x}}} \left(\frac{I_u C_1}{C_3} u + \frac{I_v C_2}{C_3} u \right) (\mathcal{R} + I_{warp} - I) w = 0 \\
 \frac{\delta ERR}{\delta \Delta m_{32}} &= 2 \sum_{all \bar{\mathbf{x}}} \left(\frac{I_u C_1}{C_3} v + \frac{I_v C_2}{C_3} v \right) (\mathcal{R} + I_{warp} - I) w = 0 \quad (C.21)
 \end{aligned}$$

This system can be written shorter as

$$A^T W A \mathbf{c} = A^T W \mathbf{b} \quad (C.22)$$

where

$$\begin{aligned}
 a_1(\mathbf{x}) &= u I_u(\mathbf{x}) / C_3(\mathbf{x}) \\
 a_2(\mathbf{x}) &= v I_u(\mathbf{x}) / C_3(\mathbf{x}) \\
 a_3(\mathbf{x}) &= I_u(\mathbf{x}) / C_3(\mathbf{x}) \\
 a_4(\mathbf{x}) &= u I_v(\mathbf{x}) / C_3(\mathbf{x}) \\
 a_5(\mathbf{x}) &= v I_v(\mathbf{x}) / C_3(\mathbf{x}) \\
 a_6(\mathbf{x}) &= I_v(\mathbf{x}) / C_3(\mathbf{x}) \\
 a_7(\mathbf{x}) &= \left(\frac{I_u C_1}{C_3} u + \frac{I_v C_2}{C_3} u \right) \\
 a_8(\mathbf{x}) &= \left(\frac{I_u C_1}{C_3} v + \frac{I_v C_2}{C_3} v \right) \\
 A &= \begin{bmatrix} | & | & | & | & | & | & | & | & | \\ a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 & a_8 & \\ | & | & | & | & | & | & | & | & | \end{bmatrix} \\
 W &= \begin{bmatrix} \diagdown & & \mathbf{0} \\ & w & \\ \mathbf{0} & & \diagdown \end{bmatrix}
 \end{aligned}$$

$$\mathbf{c} = \begin{bmatrix} \Delta m_{11} \\ \Delta m_{12} \\ \Delta m_{13} \\ \Delta m_{21} \\ \Delta m_{22} \\ \Delta m_{23} \\ \Delta m_{31} \\ \Delta m_{32} \end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix} (I - I_{warp}) \end{bmatrix}$$

and the solution would simply be

$$\mathbf{c} = (A^T W A)^{-1} A^T W \mathbf{b}$$

The residual could be calculated as well, and is equal to

$$\mathbf{r} = \sqrt{W}(\mathbf{b} - A\mathbf{c})$$

Solution steps

Here, the precalculated matrices are $(A^T W A)^{-1} A^T W$, A , and \sqrt{W} . Note that \sqrt{W} is diagonal, and the reduced number of computations needed when multiplying it with other matrices is used in the table. N in the tables is the total number of pixels in the tracked patch.

<i>Calculation</i>	<i>multiplications</i>	<i>additions</i>	<i>subtractions</i>
$\mathbf{b} = (I - I_{warp})$			N
$\mathbf{c} = (A^T W A)^{-1} A^T W \mathbf{b}$	$8N$	$8N - 8$	
$\mathbf{r} = \sqrt{W}(\mathbf{b} - A\mathbf{c})$	$9N$	$7N$	N
<i>total sum (excluding residual):</i>	$8N$	$8N - 8$	N
<i>total sum (including residual):</i>	$17N$	$15N - 8$	$2N$

Issues

The abbreviations $C_i(u, v)$ used while creating the precalculated matrices need some discussion. Since each $C_i(u, v)$ contains elements from the homography, they add yet another approximation to the tracker. During tracking their values depend on the pose of the tracked plane and they are not constant as assumed by the construction of the tracker.

Analysing the problem further revealed that the influence of C_3 can be regarded as a weighting, which would not affect the tracking except for the few pixels where the current sign of $C_3(u, v)$ differ from the sign of the initial $C_3(u, v)$. The real issue is that C_1 and C_2 affects the tracker, but mainly the m_{31} and m_{32} parameters. The problem is not visible while observing a tracker in progress, but most likely it affects at least the convergence rate of these trackers.

In practice, the current tracker works well when the tracked surface is not rotated away from the camera too much. This approximation is however not good

enough in some applications when the tracked surface is rotated up to 90 degrees relative to the camera. There are three ways to handle this. The chosen option was to do nothing. Alternatively, the precalculated matrices of the tracker can be recalculated whenever the surface has been angled more then some constant angle away from the position which was used the last time the matrices was updated. That would count as a quick and dirty fix and should preferably be avoided. The third way of handling these elements is by completely redesigning the tracker, and possible alternatives may be found in both [11] and [1].

C.5 Intuitive Description

Earlier in this chapter, relationships necessary for calculating optical flow was derived. Those derivations were focused entirely at finding solutions that would minimise the deviations from the constant brightness assumption. As a consequence, the resulting equations for optical flow tracking were justified only mathematically.

To make up for the lack of intuitiveness in the mathematical derivations, this section is going to try to intuitively describe how optical flow tracking works by providing some examples. To avoid making this example too cluttered with details, the translation tracker from section C.1 is going to be used as an example. That tracker is tracking only two parameters, Δx and Δy . Expanding the results for any number of parameters is however straightforward. The system to solve for optical flow was described in equation (C.9). That equation system is repeated here without weighting for reference:

$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_y I_x & \sum I_y I_y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} \sum I_x (I - I_{warp}) \\ \sum I_y (I - I_{warp}) \end{bmatrix} \quad (C.23)$$

Each individual summation is performed for all pixels \bar{x} of the tracked patch, I is the original patch, and I_{warp} is the image obtained by warping the current image using the last known tracker parameters (x and y). I_x and I_y is the x and y -derivatives of the original patch, and can be considered to be two motion templates.

One Parameter

If the tracker was supposed to only track translations in the x -direction, the equation above could be written as

$$\Delta x \sum I_x I_x = \sum I_x (I - I_{warp}) \quad (C.24)$$

That equation above could be visualised for particular cases, such as the one in figure C.1 where the warped image was translated one pixel to the right. Putting the images from figure C.1 into equation (C.24) would result in

$$\begin{bmatrix} \text{Patch 1} \end{bmatrix} * \begin{bmatrix} \text{Patch 2} \end{bmatrix} \Delta x = \begin{bmatrix} \text{Patch 3} \end{bmatrix} * \begin{bmatrix} \text{Patch 4} \end{bmatrix}$$

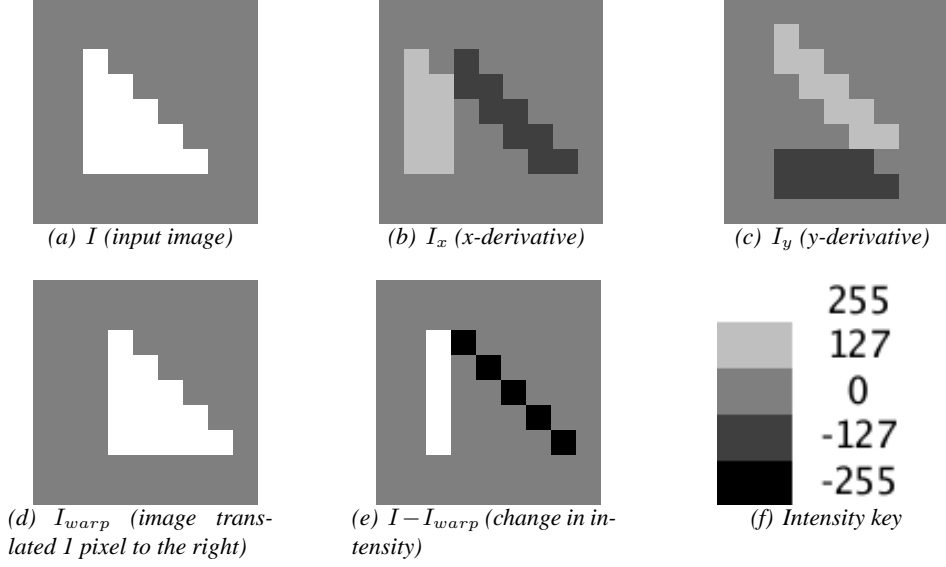


Figure C.1: Image I and derivatives, and I_{warped} which is translated one pixel in positive x -direction.

Observe that unpadding convolution $(*)$ of two images is identical to the summation of the product of pixels at the same position in both images. Performing that convolution yields

$$292\,612.5\Delta x = 292\,612.5 \implies \Delta x = 1$$

which agrees with the one pixel displacement of I_{warped} . Looking at $I - I_{\text{warped}}$ for fractional displacements in figure C.2 makes it quite clear that the resulting Δx is directly proportional to the displacement in the image, as the size of the areas involved in the convolutions is directly proportional to the displacement.

When central difference is used to calculate derivatives, Δx would get values directly proportional to the displacement for translations up to one pixel. For translations between one and two pixels, the calculated Δx would have the right sign, but not the right magnitude. For translations of two pixels or more, the calculated Δx would be completely uncorrelated to the translation in the image, unless the image contain gradients spanning larger areas.

The parameter $\sum I_x I_x$ from equation (C.24) is a constant, originating from the formal derivation of this particular optical flow tracker. That constant ensures as this example showed, that the solved optical flow parameter is close to identical to the motion of the tracked object. It is known that a motion template multiplied by the magnitude of its parameter would equal the change in intensity caused by that motion (with small variations depending on how the derivatives were calculated). That would imply that

$$I_x \Delta x \approx (I - I_{\text{warped}}) \quad (\text{C.25})$$

Multiplying each pixel by I_x in the equation above, followed by summing both sides would yield equation (C.24), thereby verifying $\sum I_x I_x$ as the correct constant.

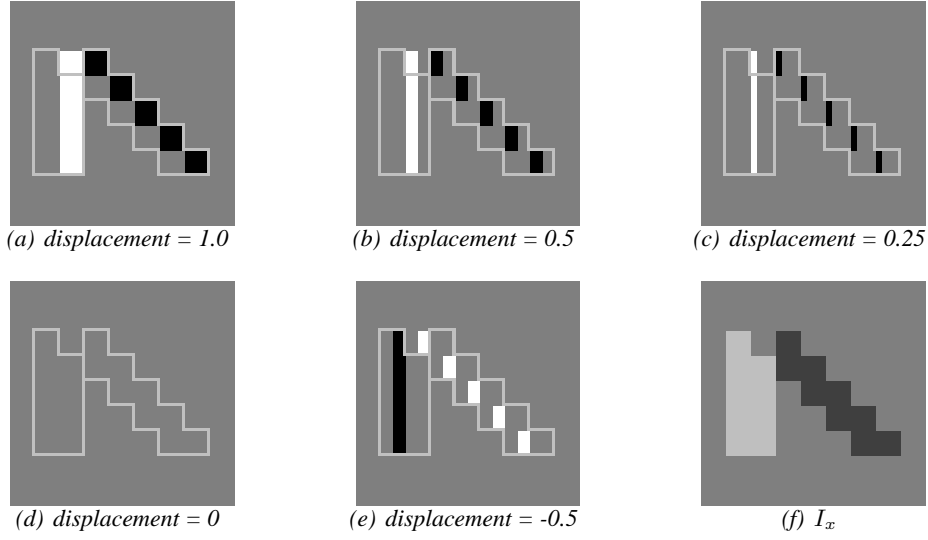


Figure C.2: $I - I_{\text{warp}}$ for a few different displacements. The outlined regions is the part shared with I_x . Only that shared part contributes to $\sum I_x(I - I_{\text{warp}})$ in equation (C.24).

An observant reader might assume that the motion template I_x (figure C.1(b)) has to be identical to $I - I_{\text{warp}}$ (figure C.1(e)). The deviation between those two images derive from the fact that a central difference was used to calculate I_x . To maintain correct results from algorithms, the central difference must be acquired by convolving the input image with the kernel $\begin{bmatrix} 0.5 & 0 & -0.5 \end{bmatrix}$, as the created motion template otherwise would correspond to a two pixel displacement.

Two Parameters

Back to the example, one sooner or later is faced by solving for more than one parameter. Assuming that a second parameter could be calculated in the same manner as a single parameter, that would give us these two equations for calculating both parameters:

$$\begin{aligned} (\sum I_x I_x) \Delta x &= \sum I_x (I - I_{\text{warp}}) \\ (\sum I_y I_y) \Delta y &= \sum I_y (I - I_{\text{warp}}) \end{aligned}$$

This solution is going to work poorly, or not at all depending on the tracked patch.

The problem is that motions described by any single parameter will introduce errors into the calculation of the other parameters. In our case, the extension of the tracker to track Δy would result in incorrect evaluations of Δx . That problem is illustrated below where Δx is calculated for a patch translated one pixel in the positive y-direction. The wanted result would be no change in Δx , but there was a big change. Figure C.3 shows intermediate values for a translation in y-direction.

Here equation (C.24) was used to calculate Δx without knowledge about y-translations for the patch in figure C.3. That patch was translated 1 pixel in the positive y-direction.

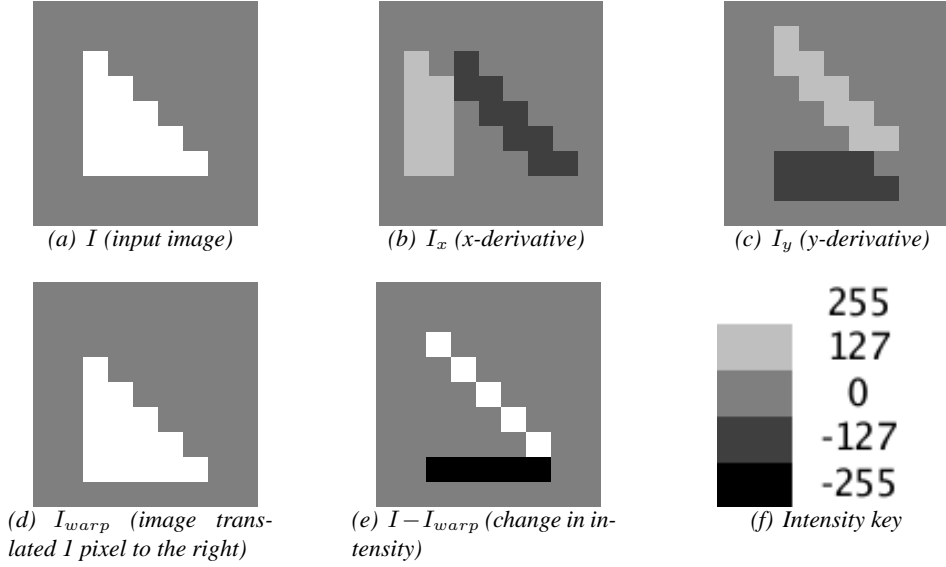


Figure C.3: Image I and derivatives, and I_{warp} which was translated one pixel in positive y -direction.

Performing the calculation above yields

$$292\ 612.5\Delta x = -130\ 050 \implies \Delta x \approx 0.44$$

As can be seen, Δx would not equate to zero, as the undetermined motion in y -direction interacts with the motion template used to determine motion in the x -direction.

Making the assumption that the true translation in y -direction would be known, one would face the problem of determining the influence of a known y -displacement on the calculated value of the x -direction. The equation system (C.23) (from the theory in section C.1) holds the key to these interdependencies. The relevant part is repeated here for reference:

$$\Delta x \sum I_x I_x + \Delta y \sum I_x I_y = \sum I_x (I - I_{warp})$$

Still pretending that we know Δy , we could directly calculate Δx from the equation above:

$$\Delta x \sum I_x I_x = \sum I_x ((I - I_{warp}) - I_y \Delta y)$$

There is a simple way to think of that correction. It simply states that you could subtract $I_y \Delta y$ from any difference image to cancel out distortions caused by the motion Δy . The same holds for any other motion template multiplied by its parameter as long as the motion compensated for is in the sub-pixel range.

Another understanding of correlation could be found by looking at the meaning of the factor $\sum I_x I_y$. Evaluating that for the triangle example used throughout this section yields:

$$\sum I_x I_y = \begin{array}{|c|} \hline \text{Image 1} \\ \hline \end{array} * \begin{array}{|c|} \hline \text{Image 2} \\ \hline \end{array} = \sum \begin{array}{|c|} \hline \text{Image 3} \\ \hline \end{array} = -130\,050$$

As seen in the image above, $\sum I_x I_y$ leaves us with a large negative number in this case, which occurred mostly because of a large common diagonal portion where the motion templates had opposite values. Any response in the difference image $I - I_{\text{warp}}$ would affect both parameters in that common region, regardless of which underlying motion that caused the response.

Setting up an equation system for simultaneous solution of Δx and Δy makes it possible to calculate a solution which is compensated for dependencies between the parameters. For that system, it is preferable to keep the parameters located as in equation (C.23).

$$\begin{bmatrix} \sum I_x I_x & \sum I_x I_y \\ \sum I_y I_x & \sum I_y I_y \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} \sum I_x (I - I_{\text{warp}}) \\ \sum I_y (I - I_{\text{warp}}) \end{bmatrix}$$

All parameters from the equation above should now be familiar:

$\sum I_x I_x$ - purely a constant scaling factor

$\sum I_x I_y$ - constant factor which compensates for mutual interference between motion templates.

Δx and Δy - parameters to calculate

I_x Motion template for motions in x-direction

I_y Motion template for motions in y-direction

$I - I_{\text{warp}}$ difference image

Appendix D

Notes about NOMAN

NOMAN is a shared code repository for work at the *Computer Vision and Active Perception Laboratory* (CVAP) and the *Centre for Autonomous Systems*, both part of the *Royal Institute of Technology* (KTH). NOMAN mostly contains reusable C++ classes, and is a valuable resource. During my usage of NOMAN, some of those classes has been expanded with new features or improved in other ways.

The following sections include suggestions for improvements to NOMAN. I suggest that at least the critical changes should get merged into the NOMAN repository. Besides that, a large portion of code developed while working on this thesis (see chapter 6) could also be considered suitable for inclusion in NOMAN.

D.1 ImageShow2

This class suffers from a severe bug, which leads to memory corruption (and segmentation faults). The constructor for the class sets the dimension of the image to 320x240 pixels and allocates the necessary amount of memory. If an image larger than 320x240 is drawn, a function for resizing the visible window is called, but its memory is not enlarged to accommodate larger images. Additionally, the initially allocated memory is not freed on destruction of the class.

Besides the bugs, I added another function that seemed useful. I modified the class to make it possible to query for mouse clicks in a non-blocking manner by always enabling ButtonPress events, and polling for them in the new checkPointer function.

Those modifications can be found in the files `ImageShow2_mod.cc` and `ImageShow2_mod.hh`, which probably should get merged into the NOMAN tree.

D.2 CASMatrix

One of the most frequently used matrix operation during this work was the matrix multiplication operator. The matrix multiplication function however suffers from two different performance bottlenecks, one in the multiplication algorithm, and another caused by temporary objects. Luckily, both bottlenecks can be corrected.

D.2.1 Multiplication

The first problem lies in the implemented multiplication routine:

```
//current multiplication loop in CASMatrix
CASMatrix res(r1.q_rows(), rhs.q_cols());
res = 0.;

for( i = 0 ; i < r1.q_rows() ; i++ )
    for( j = 0 ; j < rhs.q_cols() ; j++ )
        for( int k = 0 ; k < r1.q_cols() ; k++ )
            res.elem[i][j] += r1.elem[i][k] * rhs.elem[k][j];
```

When observing the code above, it is clear that the index variable `k` changes in each iteration, and that forces `rhs.elem[k][j]` to access memory in a non-linear manner. Accessing memory non-linearly is never a good idea according to [6] [19], [18] and my own measurements. If linear access of data would be used instead, special hardware prefetch circuits¹ would move data from the slow main memory into fast cache memory even before the microprocessor could know that it might need the data.

The performance of the multiplication routine was improved greatly by changing the indexes such that linear access of all the three matrices is achieved.

```
//suggested multiplication loop in CASMatrix
res = 0.;
for( i = 0 ; i < r1.q_rows() ; i++ )
    for( int k = 0 ; k < r1.q_cols() ; k++ )
        for( j = 0 ; j < rhs.q_cols() ; j++ )
            res.elem[i][j] += r1.elem[i][k] * rhs.elem[k][j];
```

Additionally, it is possible to improve performance slightly more for larger matrices by avoiding using two-dimensional arrays in the innermost loop (see `SMatrix_implementation.hh`). It might not be preferably to try and optimise `CASMatrix` too much, as the results could prove to be architecture dependent. It should be noted that the majority of the time used when multiplying small matrices lies in the creation of temporary objects, and that that issue also should be addressed to achieve the best performance possible. For reference, some comparisons between the uncompleted `SMatrix` class and the `CASMatrix` class can be found in section 6.7.1.

D.2.2 Temporary Objects

When using `CASMatrix` objects to calculate expressions such as $A = b * c$, that code would generate a temporary object containing the result of the multiplication before the results are copied into `A`. The longer the expression, the more temporary variables would be generated. The temporary variables add a significant overhead to calculations when the matrices are small. Multiplying two 3x3 matrices for instance uses less then one third of the time for the multiplication – the majority of

¹Early P4 microprocessors had four prefetch units, allowing an absolute maximum of four different arrays to be prefetched simultaneously when they are all accessed linearly.

the time is eaten up by creating, copying and destroying the temporary variable. If a simpler operation than matrix multiplication is considered, the proportion between time eaten up by the temporary variable and useful processing time would be much larger.

The effect of bypassing the use of a temporary variable can roughly be seen by comparing the two rightmost columns in table 6.1 on page 38. Even though that data is not for CASMatrix, the effects should be similar.

There exists a way to get rid of temporary variables. The use of common arithmetic expressions could be replaced by calling corresponding temporary-free functions. As an example, `dest=a*b;` could be evaluated by calling the temporary-free `mul_and_assign(CASMatrix& dest, const CASMatrix& a, const CASMatrix& b);`. Even if only the most common expressions (such as `A=b*c;` `A=b+c;` `A=b-c;`) would be implemented directly, it would lead to a large performance increase when small matrices are used.

The addition of a large number of strangely named functions would however make it difficult for users to use the class efficiently. Luckily, there is a way around that through the use of template meta programming. If lazy evaluation² were used as described in [16], the compiler would translate arithmetic expressions directly into calls to the appropriate temporary-free functions. Remaining unhandled arithmetic expressions would get their sub expressions evaluated as far as possible without temporary variables. The compiler would not resort to using overloaded arithmetic functions (which might use temporary objects) until the remaining expression cannot be handled by any specialized temporary-free function.

I would presently not recommend the implementation of a scheme for removing temporary objects because of the added complexity. I do however see a need for knowledge about this issue, as some (like myself) happily started using the convenient CASMatrix class without too many thoughts on possible performance penalties.

D.2.3 Using Optimised Libraries

I would like to draw a little attention to the fact that both AMD and Intel provides highly optimised versions of BLAS³, which together with LAPACK⁴ would be suitable for performing more of the functions of CASMatrix. There also exists efforts such as ATLAS⁵, which target more architectures as well.

Previously, I would have stated that it would not be a good idea to include dependencies on many additional packages into NOMAN. But currently, both BLAS and LAPACK is needed to perform singular value decompositions, and I hope the maintainers of CASMatrix are thinking about either removing those library dependencies, or are thinking about using the optimised libraries to improve performance in other parts of the class as well.

²also known as closure objects

³Basic Linear Algebra Subprograms

⁴Linear Algebra PACKage

⁵Automatically Tuned Linear Algebra Software

D.2.4 Optimisation Flag

Currently, no optimisation is enabled when compiling any of the functions in `NOMAN/src/Math` into the library `libMath`. I would suggest that `-O2` should be enabled by default. Compiling `CASMatrix` with optimisations and using an improved multiplication routine would result in approximately ten times the performance for matrix multiplications.

The only reason against using optimisation might be that enabling `-O2` could make debugging of classes using `CASMatrix` more difficult.

Another solution could be to define a few of the mostly used functions in the h-file. Then those functions would be optimised in the same way as any other application which depends on `CASMatrix`. It would pose a problem to select which functions to include in the h-file, and only slow commonly used functions (with little code) would be best suited for inclusion (unless everything is moved into the h-file).

D.3 Resolving Compilation Issues with gcc 4.4.1 and FC6

This list might not be complete, but have some notes about compiling and using NOMAN with newer compilers and libraries.

D.3.1 libMath.so

Building this library was not straightforward because of newly added dependencies to the `libblas-3`, `liblapack` and `libg2c` libraries. More recent GCC compilers (most probably from 4.x.x and later) ship with `libgfortran` instead of `libg2c`, which made it necessary to edit `"NOMAN/src/Math/Makefile"` to reflect those changes. It would be nice if the necessary tests could be performed by a configure script.

D.3.2 SELinux

As SELinux has started to be common in many distributions, it is necessary to know how to run NOMAN if SELinux is enabled. If the message "Error: cannot restore segment prot after reloc: Permission denied" gets displayed when trying to run certain NOMAN applications, the "Memory Protection" policy for SELinux would have to be changed to "Allow all unconfined executables to use libraries requiring text relocation that are not labeled `textrel_shlib_t`". It should also be possible to change the security context for the libraries built by NOMAN by issuing `"chcon -t textrel_shlib_t NOMAN/lib/*.so"`. Both ways should make it possible to use NOMAN together with SELinux.

D.3.3 V4L2Grabber

This class relied on the obsolete define `HAVE_V4L2` to get the preprocessor to generate valid code. The code would not compile without `HAVE_V4L2`, so it would be preferable to modify `V4L2Grabber.cc` to simply define `HAVE_V4L2` for those kernels (2.6.18 or later) that needs it:

D. Notes about NOMAN

```
//Since current kernels don not have HAVE_V4L2 defined any more,  
//and V4L2 is included in the kernel source  
//see "http://www.kernel.org/pub/linux/kernel/v2.6/ChangeLog-2.6.18"  
#include <linux/version.h>  
#if LINUX_VERSION_CODE >= KERNEL_VERSION(2, 6, 18)  
#define HAVE_V4L2  
#endif
```


Appendix E

Line Tracking and Detection

During the initial phase of this work, one of the goals involved integrating a vision part into a human machine collaborative system (HMCS). The environment where the HMCS was going to operate would include visual clues in the form of lines hinting preferred tool¹ paths. That would require detection as well as tracking of line segments in the vicinity of the tool.

Due to unexpected hardware failure, the HMCS system was never implemented, and line detection and tracking is no longer a goal of this work. Therefore, the developed line tracker as well as the implemented line detection algorithm are described in this appendix rather than in the main text of the thesis.

E.1 Line Tracking

The developed line tracking algorithm was based on the edge tracking algorithm described in [9]. Their edge tracking algorithm is described in the first section, and latter sections describes how it was extended to allow tracking of lines.

E.1.1 Edge Tracking

One computationally effective and easy to conceptualise way to track edges was described in [9]. Their methodology was to use known information of the edge (last known position and angle) when estimating position and angle in the current frame. Thereby, the current frame could be preprocessed to facilitate determination of the position and angle of the edge. Further on, the current angle of the edge will be denoted by θ_t .

If the approximate state of the edge in the input image is known, those parameters could be used to rotate the part of the image containing the edge. Then an almost vertical edge should be found.

Figure E.2 shows the edge in the input image (figure E.1) after rotation by θ_{t-1} , which was the last known angle of the edge. If the rotation since the previous frame was small, that edge would be almost vertical.

¹The tool is attached to a robotic arm, has a mounted camera, and is moved by measuring the applied force from the user. The applied force would be processed by a computer, which determines how the robotic arm should be moved.



Figure E.1: Unrotated edge in input image.



Figure E.2: Edge rotated by last known angle θ_{t-1} (preprocessed image).

The purpose of getting a rotated cut-out of the edge, as they did in [9], was to facilitate the estimation of the state of the edge. The first step is to estimate the angle of the rotated line as well as its displacement along the normal direction of the line. After that, a transformation is used to retrieve the angle and position of the unrotated tracked edge in the input image.

The estimation of parameters for the rotated (almost vertical) edge was in [9] carried out by first creating two slightly rotated² images of the rotated cut-out of the edge (as in figure E.3).

One important step is then carried out in all three images. The purpose of that step is to determine the position and response of the strongest vertical edge in all three images. To find vertical edges, one can differentiate the image in the horizontal direction. The differentiated image would contain edge points as either larger positive or larger negative values (as in figure E.4(b)). To determine the position of one vertical edge, one could then sum all columns in the differentiated image, and the column with the greatest absolute response is the position of the strongest edge (figure E.4(c)). For increased accuracy, the average position acquired from the three differently rotated images is used as the position of the edge.

²shear in just one direction can approximate this rotation

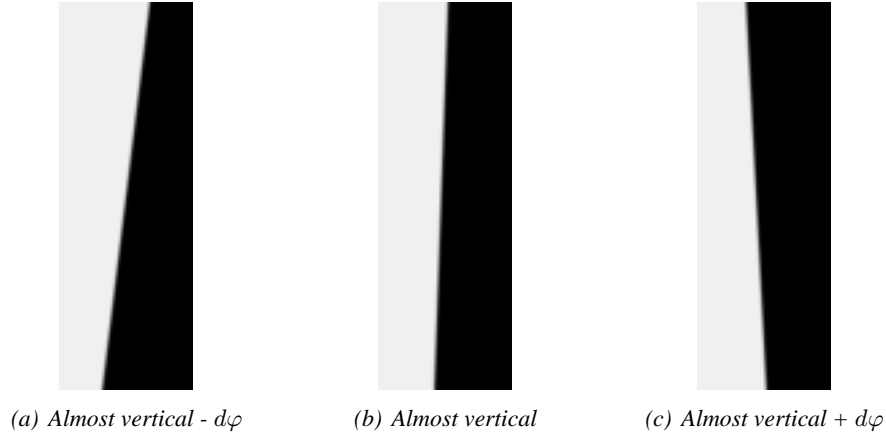


Figure E.3: The edge rotated by three different amounts.

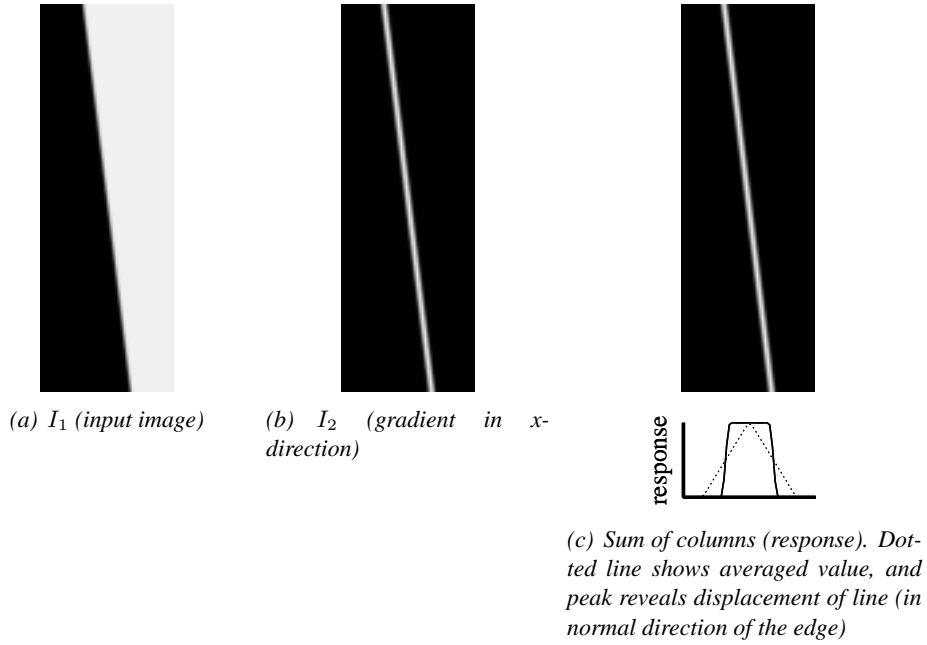


Figure E.4: Intermediate results during edge tracking

Now that the position of the rotated edge is known, the angle of that edge has to be determined as well. This was solved in [9] by using the greatest responses in the three differently rotated images. By fitting a quadratic curve to these three values (as in figure E.5), they estimated which angle $d\theta$ that should give the maximum response and hence correspond to a completely vertical edge. Knowing the angle of the edge and displacement in the normal direction of the edge, one can calculate

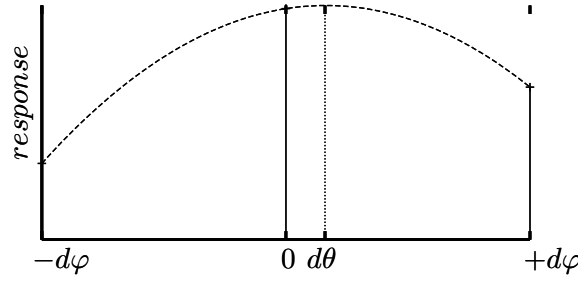


Figure E.5: Interpolation of angle by fitting of a 2'nd degree polynomial to responses at $-d\varphi$, 0, and $+d\varphi$.

the real angle and position of the edge:

$$\begin{pmatrix} x_t \\ y_t \\ \theta_t \end{pmatrix} = \begin{pmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{pmatrix} + \begin{pmatrix} -ds \sin(\theta_{t-1} + d\theta) \\ ds \cos(\theta_{t-1} + d\theta) \\ d\theta \end{pmatrix}$$

, where ds = displacement of edge in the normal direction of the edge, and $d\theta$ = deviation in angle. Subscripts $t - 1$ denotes that those variables are the ones corresponding to the previous frame.

E.1.2 Investigated Improvements

Three improvements over the implementation in [9] has been investigated: modifying the algorithm to track lines instead of only edges, determining endpoints of lines, and improving angle estimation. Those improvements are discussed in the following sections:

Ability to Track Lines

The algorithm described in [9] had to be modified to be able to track lines. The problem was that summing of columns to determine displacement in the normal direction of the line would not work reliable. Lines slightly slanted in the intermediate images would sum gradients from both sides of the line as columns of the image are summed. That results in a response close to zero for the middle of the line (as figure E.6(a) illustrates).

That can be avoided in a few ways. The gradient image could be preprocessed in such a way that no cancelling occurs for columns containing both sides of the line. Alternatively, the centre of the line could be considered to be in the middle between the two peaks in the response with opposite values (figure E.6(b)). That is however not a good option since all information from the middle of the line will get lost, and the tracking will be much more susceptible to noise. Increased noise sensitivity would also occur if the centre of the line was considered to be on either peak in figure E.6(b) because of the cancellations which would be present.

To avoid increased noise vulnerability, only preprocessing of the gradient image to eliminate cancellation is an option. When only one edge of the line is going

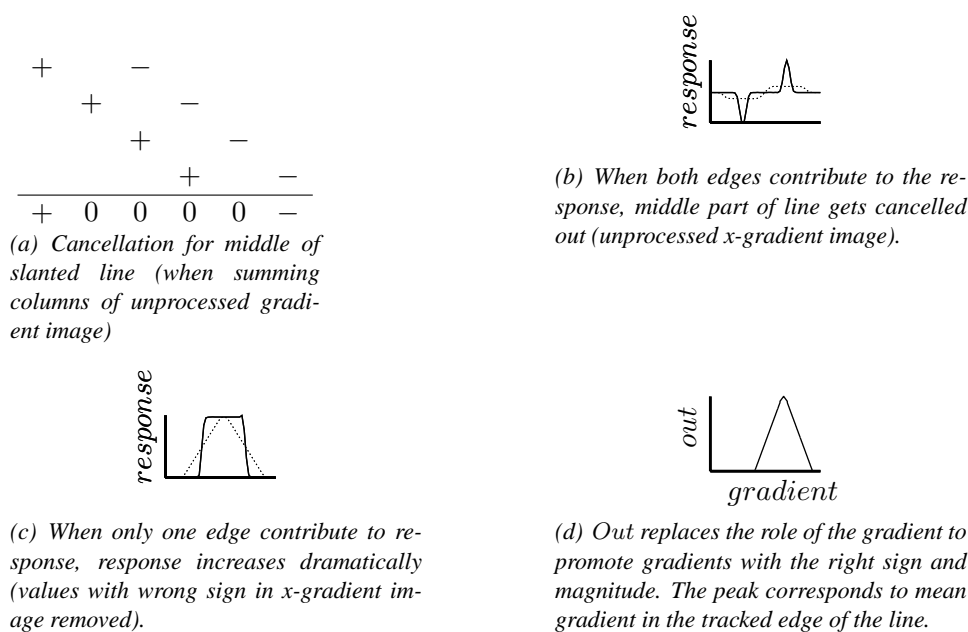


Figure E.6: Preprocessing of the x-gradient image could increase the response significantly for slanted lines.

to be tracked, preprocessing by removing all gradients with the wrong sign from the intermediate image could be performed. Better yet would be to only propagate gradients close to the mean gradient of the tracked edge of the line. Both alternatives would generate responses similar to the one in figure E.6(c). I chose to map gradients according to figure E.6(d), such that gradients belonging to the tracked edge of the line is heavily promoted.

The drawback of this solution is that the gradient image has to be computed. Previously, the operations of summing columns, and taking derivatives could be interchanged, resulting in a need for taking derivatives of only one row. Now, derivatives have to be evaluated for all pixels in a box fitted around the line.

Determining Line Extension

Determination of endpoints for line segments was not done in [9], but was implemented here by looking at an image derivated in the normal direction of the line (as in figure E.7). By summing all rows, it would be possible to determine the extension of the line. Note that gradients again have to be preprocessed to prevent cancelling of the line when summing.

This procedure is more sensitive to noise then the procedure used to determine displacement of the line in its normal direction. Therefore, some restrictions have been placed on the endpoint determination, such as not allowing endpoints to move more then a certain number of pixels in a tracking iteration. It should be noted that [9] avoided handling of the extension of lines completely, and I currently suggests that more work should be put into endpoint detection before relying on the method presented here.

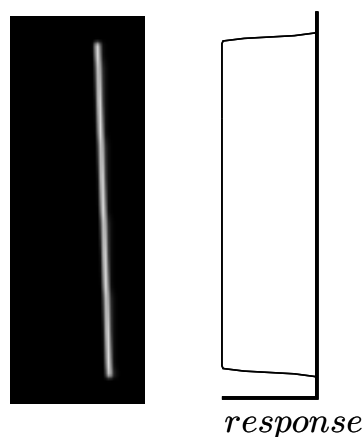


Figure E.7: Determination of endpoints by summing of gradients in normal direction of line (along rows in warped image). Note that the gradient image was preprocessed to reveal only one side of the line.

There are other ways to determine line extension as well. They should have been investigated if the visual input to the line tracking algorithm was of such a nature that endpoint determination could be problematic. One of the simpler candidates would be to filter the neighbourhood of the expected endpoint with certain kernels. Since the direction of the line is known, something similar to Canny Edge Detection could be performed without having to use any additional rotated versions of the edge detection kernel. Another alternative might be to put smaller area trackers on the line edges, tailored to respond to displacement, rotation and scaling only. They could be linked to the line tracker, and aid in determining the parameters for the line. It might actually be possible to use an area tracker to perform the entire line tracking operation.

Better Estimation of Line Angle

A better estimator for line angle has not been implemented even though some effort was made at finding one. in spite of that, some of the results was interesting:

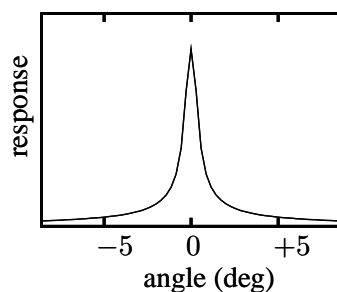


Figure E.8: Typical response versus angle for rotated edge

Intuitively, finding the angle by fitting of a quadratic function for three mea-

surement points seemed like a bad solution. The supposed reason was that the shape of the response (figure E.8) was too “spiky”. The thought was that it would be difficult to estimate the maxima of that function knowing only three equally spaced points.

However, results from simulations using a uniformly distributed angle of the edge from -2 to 2 degrees revealed that the medium estimation error was only 0.7 degrees (even though the worst case estimation error was 1.15 degrees). That corresponds to an error of 1.2 (or 2) pixels for a 100 pixel long edge, and that was considered as good enough since iteratively tracking the line would lead to even better estimates.

Alternative ways to calculate the angle could be employed, such as splitting the line into several line segments. That would perform better, as edge centres are much easier to determine than edge angles.

E.2 Line Detection – Hough Transform

Tracking lines with the procedure outlined above would require the position of the line to be known in advance. One algorithm designed to locate lines is the Hough transform[8]. It is a transform which determines the location of line segments in a binary image by first transforming edge positions (x, y) into all possible parametrisations for all lines that could pass through that point. In my case the (ρ, θ) parametrisation is used, with $\theta \in [0, \pi]$, and $\rho \in [0, d]$ (where $d = \text{diagonal of image}$). The relation between the (x, y) - and (ρ, θ) -parametrisation is $\rho = -x \sin \theta + y \cos \theta$.

The main data structure used by the transform is an accumulator, holding the number of occurrences of possible combinations between ρ and θ . The size of this accumulator determines the resolution of the transform. If one would like to detect lines within a pixel of their true position in an image, one would have to be able to separate roughly $N_\rho \approx d$ different ρ values, and $N_\theta \approx \pi / \arctan(1/d)$ different θ values (where $d = \sqrt{(\text{image width})^2 + (\text{image height})^2}$).

The transform stores all possible (ρ, θ) -parameters for all on-pixels in one accumulator, to make it possible to locate which parametrisations that acquired the largest amount of votes. Table E.1 below summarizes the number of additions to the accumulator for some image sizes and presumed ratios of pixels belonging to lines.

E.2.1 Using 8x8 Patches

Applying the Hough transform on a large image has some limitations. To begin with, it would perform poorly at finding shorter line segments. Those segments could easily get masked by noise and other features in the image. At the same time, the Hough transform by itself will not reveal the extension of the line, requiring scanning along each parametrised line to find the segments constituting lines. Another limitation is the computational requirements, which even today makes it less useful for real time processing of video segments.

Table E.1: Number of accumulator inserts for differently sized images and different amounts of edges.

Image Size	Accumulator		Accumulator inserts		
	N_ρ	N_θ	100% edges	10% edges	one edge-pixel
8x8	12	36	2 304	231	36
10x10	15	45	4 500	450	45
50x50	71	222	555 000	55 500	222
80x80	114	355	2 272 000	227 200	355
320x240	400	1 257	96 537 600	9 653 760	1 257
640x480	800	2 513	771 993 600	77 199 360	2 513

Table E.1 shows the rapid increase of computational burden as the algorithm is used to locate line segments in larger and larger images. Clearly, the algorithm is not suited for processing of large images directly, and another approach has to be taken.

My suggestion is to apply the Hough transform on several smaller overlapping regions, and then merge the detected line segments. That would reduce the amount of necessary computations dramatically. An algorithm for joining the smaller line segments from neighbouring 8x8 patches would however have to be devised, and that has not been implemented in this work. The principle does however sound simple: join line segments in neighbouring 8x8 patches as long as no endpoints of lines to be joined deviates longer then some given constant from the new line.

Consider, for example, a subdivision of an area of 80x80 pixels into 181 ($= 10 * 10 + 9 * 9$) smaller 8x8 regions, thereby reducing the worst case number of inserts to $181 * 2\,304 = 417\,024$ (a reduction by the factor 5). In the same way, the worst case number of inserts could be reduced by a factor of 18 for a region of 320x240 pixels. Limiting inserts into the accumulator by requiring a similar direction of the parameterisation compared to the direction of the local gradient in the image would also reduce the inserts by an additional factor of at least four. The gain of using gradient directions would not lie directly in the reduced number of inserts, but rather in the reduced complexity of finding the most valid peak in the accumulator, and hence finding the corresponding line. In total, the number of inserts can be reduced by a factor of at least 72 in the 320x240 case – providing a substantial increase of performance. The factor of avoided inserts for differently sized images is tabulated in table E.2

Table E.2: Factor of avoided inserts for differently sized images.

<i>Image Size</i>	<i>by using 8x8 regions</i>	<i>by using 8x8 regions and edge gradient direction</i>
8x8	1.0	4
32x32	2.5	10
64x64	4.5	18
80x80	5.4	22
320x240	18.0	72
640x480	35.4	142

TRITA-CSC-E 2007:096
ISRN-KTH/CSC/E--07/096--SE
ISSN-1653-5715