Tracking Faces using Active-Appearance-Models on calibrated web cams

Studienarbeit

Alexander Brendel
supervisor: *Sven Fleck*

GRIS, WSI für Informatik, Universität Tübingen
http://www.gris.uni-tuebingen.de/
alex_brendel@web.de

10.02.2005

Abstract

In today's demanding applications in the field of computer vision the tracking of objects is becoming more and more of interest. Be it in facility automation or in the development of biometrical identification processes as well as in security applications, a robust way of finding and further tracking of objects in sequences of images is an essential method to these approaches.

Inspired by some other workgroups around the world, this work deals with an approach to track the face of the user located in front of a computer. Web cams mounted on top of the monitor track the face of the user to extract parameters such as pose and location of the face in respect to the camera and thus the computer.

To implement a tracker that allows for some variability in shape and appearance of the object to be tracked, Active Appearance Models were chosen as fulfilling this need with the benefit of an already existing programmer's interface, the AAM-API developed by M. Stegmann.

Demanding a real time scheme to meet the needs of modern Human-Computer-Interfaces, the program developed to demonstrate the capabilities of the used algorithm has been written in C++ for Win32 in combination with some other libraries including OpenCV and DirectX 9.0a.

For being able to measure the quality of the algorithm's output, a stereo vision system operating on two calibrated web cams with two separate trackers was implemented. This allows for a direct comparison of two simultaneous tracks of the same object thus getting an estimate of the tracker's overall accuracy as well as evaluating the robustness of Active Apearance Models in a real time application.

Contents

1. Tracking in real time

Applications in tracking with methods applied by computer vision sciences are manifold. Human-Computer-Interfaces for clinical application, the use of cameras to classify the quality of production goods as well as capturing motions and emotions from video streams are just a few samples of a vast area concerned.

Crucial to many of these applications is the ability to deal with the data probe in real time, demanding for the use of sophisticated algorithms that comply with this need.

1.1 Inspiration

Some of the existing approaches to track rigid objects have been studied.

Martin and Horaud [1] developed a tracker able to make use of several cameras. The approach was to track ship parts as they would occur in production with several cameras from different angles and fields of view. The underlying physical model was derived by using CAD-models of the specific ship parts. Optimisation was done by computing difference vectors between model and example and afterwards rotating and translating the model's edges accordingly. This, of course, implies an object with clearly distinguishable edges. Though the setup seems to have worked rather exact, no statements were made suggesting a method to distinguish between visible edges and those occluded by the object or about real time capabilities of the presented method.

Another approach to track objects through video sequences has been made by Isard who implemented a real time hand tracker with Condensation [2], a statistical method formerly proposed by Blake and himself. The name stands for conditional density propagation, performing tracking by using a particle filter that statistically keeps track of several possible states and assigns probabilities to each one. A sample, the conditional density, is then propagated over time. The performance of the implementation by Isard is very good, but as the dynamical model is non-linear with some stochastic portion, it is very likely hard to analyse. Furthermore, the model is not robust against changes in distance and rotations.

1.2 Demands

The two approaches above led to demands that our program implementing a new tracker should fulfill.

Most important was the ability to track rigid objects in real time. As Java is known not to be the fastest programming language, the project was decided to be implemented in C++. This also guaranteed providing a possible solution to almost all the areas concerned with tracking today.

Another goal was to be able to track different kinds of objects without having to change the program. The modularity implied should make it easy to interchange the

model the program relied on, so that the use on different objects would be as easy as possible in future applications.

A qualitative approach to measuring the efficiency of the chosen algorithm was strived for by implementing a stereo vision system with two web cams. This offered an opportunity to compare two simultaneous tracks of the same object done by two different trackers. The demand for real time capabilities strengthened with this goal - now two trackers had to be fast at the same time.

1.3 Libraries

The base of the later program was thought to be implemented using the Open Computer Vision (OpenCV) library, as many functions already implemented there could have been of use. It had also already been used at the chair with some promising results in respect to its real time capabilities. The problems followed at hand as it became clear in early evaluation process of the library that it would not be possible to acquire two streams from two different web cams at a time, as long as one desired an automatic detection of the cameras. This error was also reported by other workgroups concerned with OpenCV programming. Another problem was the implementation of OpenCV on video for Windows (VfW) interfaces. These were reported to be relatively slow compared to newer interfaces.

Disappointed with the flaws of the above library, TLIB, a programmer's interface for computer vision issues, developed at the EPFL Lausanne, was inspected [3]. An overview revealed a fast yet easy to learn interface. It is promised to be very memory efficient and further development is given. The only negative argument arousing is the reduced functional range compared to OpenCV. This makes it suitable for educational purposes (these to meet it was designed), but not of general use. As one of the goals set was being able to implement a wide range of filters, the search for an alternative went on.

Some experiments with DirectX revealed a fast interface for capturing video from two cameras at a time as it is widely independent from old VfW interfaces. The only restriction encountered regarding the cameras was that no exact matches in the type description of the cameras are allowed. For example, two cameras of the type 'Logitech 3000' would be accessible only as one camera. Another point worth being mentioned is the highly complex programmer's interface which makes it a time consuming effort to get familiar with.

The implementation of a programmer's interface for tracking objects using the AAM model was done by Mikkel B. Stegmann of Denmark Technical University [4]. This interface written in C++ had been thoroughly tested in medical and in some non-medical environments on still images or sequences of images processed offline. It produced very remarkable results [5]. The implementation needs CLAPACK support for the Microsoft Vision SDK on which it relies.

The use of the intel Image Processing Library (IPL) became necessary because of the incompatibility concerning the conversion of image formats mentioned later in this work. intel discontinued the development of this library, the last version is IPL 2.5.

2. Active Appearance Models

Active Appearance Models (AAM) were first described by Cootes, Edwards and Taylor in 1998 [6]. An AAM is a statistical model describing an object's parameters concerning shape and appearance. Shape is to be understood as the outlining contours of the object plus some inner edges corresponding for example to facial features. Appearance describes the texture of the object in a shape free space.

The model becomes 'active' by being able to learn its statistical borders of representation in a one time training session. By learning a model from annotated images, one can prevent blind optimisation in run time, which would slow the online process down. Instead, it is possible to optimise similarities in the model offline significantly speeding up later convergence in the underlying high dimensional space of the model.

2.1 Interpretation by Synthesis

The AAM algorithm interprets images unseen before by searching for the best match between the current example and that state of the model that minimises the difference to it.

Other approaches to interpret previously unseen images were done, too. One by Turk and Pentland, called 'eigenfaces', was building a merely physical model by applying a principal component analysis (PCA) to raw data and analysing the difference between the projected model and the image. However, this showed to be invariant to slight deformations of the object which occurred by minimal tilting for example. Another idea were Active Shape Models (ASM) presented by Cootes et al. The model here represented shape only. As already being able to recognize flexible shapes, it did not deal with texture in any way.

AAMs combine both shape and texture, resulting in appearance. This promises to give robust results through measuring more dimensions in object space than other models up to now did.

2.2 Modelling appearance

The active appearance model is built by examining a number of annotated images showing the object to be modelled in a reasonable position. Appearance, in [6] misleadingly used in both senses as the sole textural information and for the combination of shape and texture, is here thought to be understood the first way.

In our case, images of three different faces were used. The images have to be paired with data files containing the coordinate data to corresponding landmarks on the object; this is what we call 'annotated' (see *figure 2.1*). Every image has to have the exact same number of annotations based on the same landmarks to produce a relevant model. Here it is advisable to use slight variations in the rotational orientation of the object to allow for a higher variability in the later model to that respect. This comes in handy, when two cameras in a stereo setup are used.

To extract the shape of an image the coordinates of the annotations have to be warped into a normalised frame. All shapes from the training session describe examples of the shape to be modelled. Performing a PCA on the data results in a formula to compute a shape x from a mean shape $\overline{x}$ knowing its parameters b_g.

$$x = \overline{x} + P_g b_g \tag{1}$$

P_g represents the modes of variation of the model.

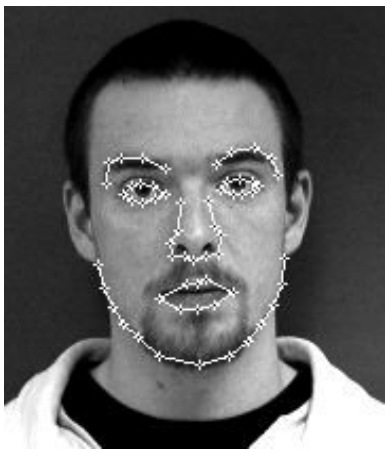

figure 2.1. annotated image with open and closed contours

Appearance is modelled by interpreting the textural information of the given images. For being able to coherently compare this data, all images are warped according to the shape data paired to the current image, so that texture can be represented in a shape-normalised frame. To reduce the effect of differing global lighting conditions, an optimisation is done.

$$g = (g_{im} - \beta) \cdot \alpha \tag{2}$$

This results in a scaled (α) and translated (β) grey image omissing the named effects. When performing yet another principal component analysis a formula describing appearance g is obtained.

$$g = \overline{g} + P_g b_g \tag{3}$$

The mean appearance $\overline{g}$ and the appearance parameters b_g are needed for computing a new shape g.

So far it is possible to model shape and appearance of an object. But, the main advantage of AAMs comes to count when both are combined. Now b_s and b_g are correlated data because shadows in texture, for example, are caused by shape as well as lighting conditions. Another principle component analysis minimizes those correlations when it is applied to each training example.

$$b = \begin{pmatrix} W_s b_s \\ b_g \end{pmatrix} = \begin{pmatrix} W_s P_s^T (x - \bar{x}) \\ P_g^T (g - \bar{g}) \end{pmatrix} \tag{4}$$

W_s is a matrix consisting of weighting values. These weights describe how image intensities vary with translations of shape-annotations. A direct comparison is obviously not reasonable. The specific weights are computed by a different process described in [6].

Now the combination of the two entitities shape and appearance results in a new active appearance model b.

$$b = Qc, \; Q = \begin{pmatrix} Q_s \\ Q_g \end{pmatrix} \tag{5}$$

Here Q are eigenvectors where c is the set of parameters.

The linear nature of the model allows for a direct formulation where P_i are the modes of variation and W_i the weight matrices.

$$\begin{aligned} x &= \bar{x} + P_s W_s Q_s c \\ g &= \bar{g} + P_g W_g Q_g c \end{aligned} \tag{6}$$

2.3 Synthesis of an example

Once the model is learnt and thus Q given, one can synthesise new images according to some model-parameters c. With (5) b is computed which gives b_s and b_g letting us compute a shape-free image with (3), using the mean appearance $\bar{g}$.

The shape, to which this shape-free image is then to be warped to, is gained by applying (1) under use of the mean shape $\bar{x}$.

2.4 Approximating a new example

When given a previously unseen image labelled with annotations, the model is able to generate a new approximation of the image. Whilst Q is gieb from former approximations, b is to be computed from the current example. Given both, it is possible to compute c with (5). Application of equations (6) gives the shape x and appearance g.

We now have to invert the lighting normalisation (2) done before. After applying the apropriate shape to all landmarks in the image, the grey-level image obtained through appearance can be projected onto the image according to shape (see *figure 2.2*).

figure 2.2. original imge (*left*), approximated and reprojected image (*right*)

2.5 AAM Search

The search algorithm is supposed to make a prediction concerning shape, appearance, position and pose of a face to search in a given, previously unseen image. An intuitive way of dealing with this is to efficiently adjust some or all model parameters, generating a new example. Then, a comparison of this example to the actual example found in the image is made. By minimising the difference between both, the algorithm and thus the model should converge and thus give a reasonable estimate.

The minimisation is done by altering certain model parameters. This seems to be hard to do at first glance, as it is a high-dimensional optimisation problem caused by many model parameters. On the other hand, the process of matching the model to a new example is quite similar every time. So one can learn how to solve this problem in advance, embedding the a priori knowledge on the optimisation into the process, rendering it more efficient and by that less time consuming at run time.

The question, which parameters are crucial in finding a good estimate of the example and how they can be efficiently varied during the search, has been dealt with in detail in [6]. Now that it is possible to predict the changes needed to make to achieve a better approximation of the example as the current one, an iterative method for the optimisation process may be formulated.

First evaluation of the error between current estimate of the sample g_s and the estimate of the model g_m, where the current estimate of the model parameters is c_0, has to be done. From this we get the current overall error thus letting us compute the predicted displacement δc of model parameters.

Now we are able to settle the difference that leads to a new estimate c_1 of the model parameters.

$$c_1 = c_0 - k\delta c, \text{ where } k \in \{1.5, 1.0, 0.5, 0.25, ...\} \quad (7)$$

After sampling the image at this new prediction, the new error vector is determined. If the new error is smaller than the last one, the new approximation is used for further processing, else the approximation is recomputed with the next choice of k.

Convergence is declared when the error does not improve any more.

3. AAM Tracking

We now come to the central chapter of this work, the tracking of faces with an AAM. The model used was made by M. Stegmann using three grey scale faces. Compared with the 88 hand labelled faces Cootes et al. used, this is a small number, but the model was feasable to produce the good results discussed below.

3.1 Implementation

Up to now a program called ‘aamTracker’ had been implemented, being able to acquire two live streams from two web cams mounted on top of the monitor. Different OpenCV filters were applicable by means of a DirectX callback filter called ProxyTrans. Saving the video streams to hard disk was an option.

The tracker was implemented on one stream only at first. The course of events is as follows: The program acquires the images and routes them to the callback filter where they are processed. Initialisation of the tracker is done on the first image using the StegmannInitialise function which predefines some initialisation parameters needed by the AAM-API. After that a frame-to-frame iteration is started, which is very much faster than the initialisation, until the error exceeds defineable boundaries. Then a reinitialisation takes place (see *figure 3.1*).

Figure 3.1. output of aamTracker

The first difficulty was to get all the needed libraries up and running. A sequential HowTo can be found in the appendix.

The main problem was dealing with the disjunctive image formats OpenCV and the AAM-API use. Converting an image from one side to the other, using a common format, as for example *.dib, both interfaces would support, was also not possible because of some incompatibility between Microsoft VisionSDK and DirectX 9.0a. Investigations revealed this to be a known problem existing with DirectX. This seems to be caused by two headers (d3d.h and qedit.h), both defining the same globally unique IDs (GUID) for two Direct3D devices, probably the web cams. The problem is known to exist from Direct3D version 7 on.

The workaround implemented was dividing the program into two partitions. One side only knows the AAM-related headers including VisionSDK; the other side only knows OpenCV-related headers and DirectX. The passing of images from one side to the other takes place by converting them to iplImages, a data type of the IPL, passing void pointers and reconverting them to a VisionSDK format and vice versa.

After having managed to find solutions to the above problems, the implementation of a second tracker on the second stream could be done. Here another instance of a CAAMTracker object is initialized so that both trackers run, each standalone, on different areas in memory. This was necessary to avoid overlapping of data in the callback function.

The options, to log the output of the tracker textually and being able to determine the error tolerance towards a new initialization manually, were implemented (see *figure 3.2*).

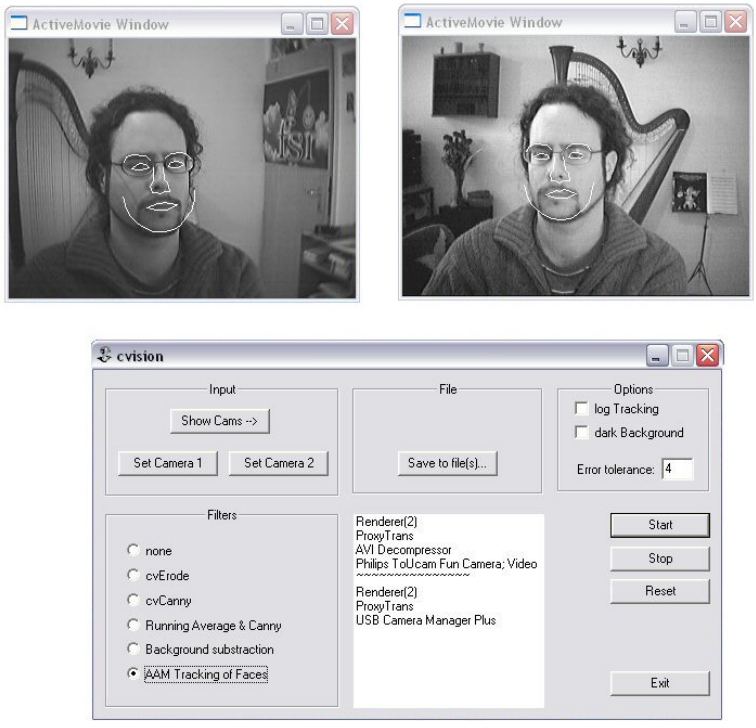

Figure 3.2. aamTracker tracking stereo

3.2 Excursion: Building a new model

Being able to build a customized model was one of the goals the tracker should be able to fulfill. Since the model lies outsourced in its own file, modularity is granted so far.

For building a model we need some images of the object to model. It is possible to acquire these by recording one video stream showing the object with aamTracker with no filters applied. It may be important to use the same resolution as the tracker later and a grey scale color scheme. The images can be extracted from the video file by calling ‘aamc sm *.avi‘.

To go on, one possibility is to use the AM_tools by Cootes. Use ‘am_markup’ to gain the point-contour relations and description of the contours. Compare [7] on how to do this.

Now the results are in Cootes’ format that is another than the one used by Stegmann. As we have to build the model with the AAM-API using ‘aamc b ..’, it is necessary to convert the result files to Stegmann’s format. As these are text files, conversion with Matlab for example should be quite easy.

By doing so, it is possible to create models to any kind of rigid object.

4. Evaluation of the Tracker

The development of the aamTracker was a step towards real time tracking of objects for many kinds of applications. But being now able to run two trackers simultaneously tracking the very same object, reveals new possibilities in evaluating the tracking algorithm implemented as well as the speed of the implementation itself.

4.1 Stereo Vision

A necessary task before being able to evaluate the quality of the tracker’s output is the calibration of both web cams.

This is possible by using the Matlab Calibration Toolbox [8]. Here it was used directly from Matlab but a C implementation is distributed with OpenCV as well.

First, the calibration pattern included in the toolbox has to be printed and mounted on a stand when possible. Each camera has to be calibrated by itself first according to the first example from the manual [9]. About twenty calibration images for each camera will be needed there for best results. These can for example be taken by using the snap shot function many cameras provide. It is most important that the pictures for both cameras are taken at the same view of the calibration setup. The resulting files `calib_results.mat` have to be renamed to `calib_results_left.mat` and `calib_results_right.mat` respectively.

Second, both cameras have to be calibrated together. By doing so, we gain information about the intrinsical as well as the extrinsical factors. This is done

according to [10]. The resulting file `Calib_Results_Stereo.mat` will later be used by the evaluation script.

4.2 qualitative verification

With the cameras calibrated, several rounds of simultaneous trackings were performed. Data recorded by aamTracker was the center of gravity of the face mesh as well as the pose of the face at each iteration step and for both trackers.

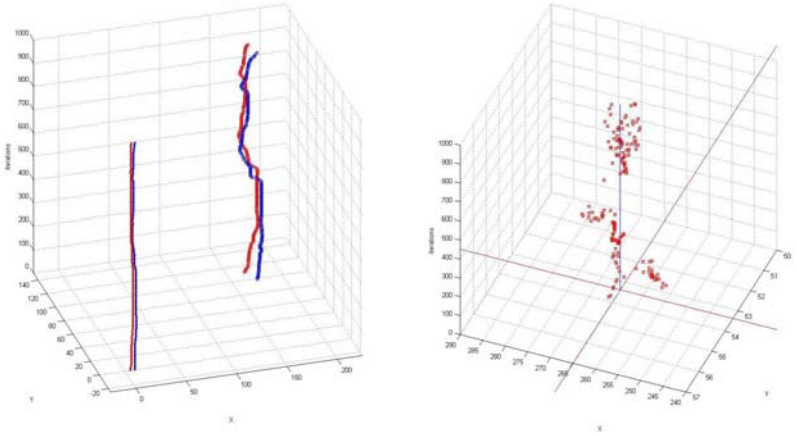

figure 4.1.left: trajectory of center of gravity *(back)*, trajectory of pose *(front)*

right: difference between rendered and actual trajectory for center of gravity

The resulting files were processed by a Matlab script using the calibration data obtained above. The script renders the trajectory of the first camera into the view of the second camera using the extrinsic parameters. The rendered trajectory can now directly be compared with the actual trajectories of the second camera (see *figure 4.1*).

The difference in both parameters pose and position is correlated with the uncertainty of the algorithm. Small values in the differences speak for robust and accurate tracking.

The differences resulting from those tracks were surprisingly good. The difference of tracking position was always smaller than 7 pixels horizontal and even smaller than 3 pixels vertical (see *figure 4.2*).

When regarding the size of each frame being 320 on 240 pixels, the percentual error computes to a maximum of 2.2% horizontal and 1.3% vertical.

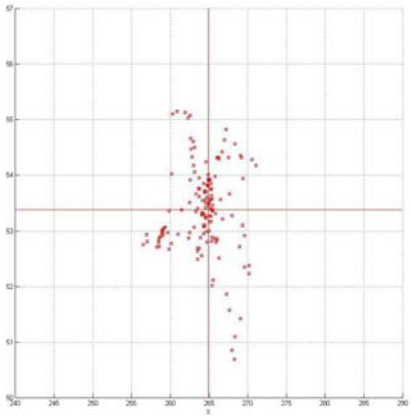

figure 4.2. resulting differences in an x-y-view

The angle measured by both cameras was also evaluated by taking the difference, here no calibration data was used (see *figure 4.3*).

By omissing calibration data a non-zero rotation between the two cameras is not regarded in this evaluation, causing the median not to be located in the origin as it would be intuitive.

A maximum difference of 1.7 degrees can be found here. The maximum angle appearing in the data probe is 11.8 degrees.

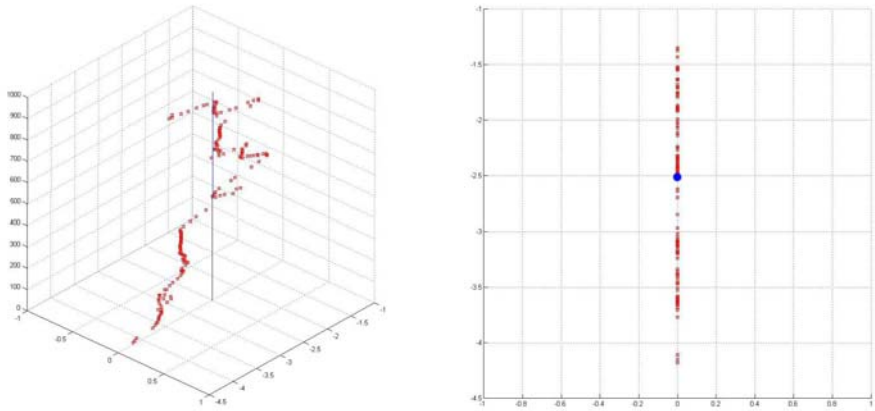

figure 4.3. left: trajectory of difference of angle

right: difference of angle with median (blue)

5. Discussion

5.1 Reached goals

We defined some goals above our implementation should fulfill. It was possible to meet even our most eager demands.

The implemented algorithm allows for using the program in real time applications as was expected. Meeting this demand with even two parallel trackers running was surprising even to us.

Implementing the program with C++ was a goal set by myself on the one hand to get into the ‘real time world’ and on the other hand to learn programming in this language as I didn’t have time for this in my studies up to that point.

We also met the demand to being able to track a variety of objects without having to change anything on the implementation itself. As the model file is separate from the program itself it can be easily interchanged.

5.2 Other aspects

The down side of the story are the many libraries necessary to be used.

The at first promising OpenCV was mostly set aside as the capturing of video on two streams was nothing to be easily solved. This is why DirectX came into talk. The unpredictable incompatibilities aroused by VisionSDK and DirectX made the situation quite complex. Now IPL was used to allow communication between VisionSDK and DirectX. The AAM-API also relied on CLAPACK which had to be added to VisionSDK as an extension.

Being a very good, yet poorly documented, programmer’s interface for dealing with AAM issues, the AAM-API still has some mostly minor bugs. The most serious one is an unhandled exception that is thrown when something with the initialisation goes wrong. This was hard to trace and impossible for us to cure. Another flaw is the absence of the projected mesh-grid in the video window when it would get partially occluded. So it is not easy to say if the tracker is still on it or if it already lost the object to be tracked.

At least being able to use more than one camera, VisionSDK still has its negative aspects. One being the incompatibility with DirectX, the other making it impossible to mount two cameras that have the exact same device description. Two cameras of different type can at least be mounted.

6. Possibilities for further development

Since the results are promising enough to be of interest for future extensions, some possibilities that evoked during the development of aamTracker will be discussed.

6.1 Variable model

Since the model is integrated into aamTracker in a modular way, it would be of great interest to test the algorithm on other objects to track. Being possible in principal, only one step is yet to be implemented, a way to generate custom models. The use of Cootes' AM_tools has been adressed in chapter 3.2. As explained there, it is possible to annotate images with a resulting data file in the wrong format. A script, for example in Matlab, has to be written to convert this data into the format the AAM-API is able to use to build a model. The latter format is well described in documentation of the programmer's interface.

Another option, making further development necessary, is trying to build evolutionary active appearance models. Evolutionary in the sense of being adaptive to changes in the object. One could think of this idea of evolution as producing a model from a reasonable starting definition of the object. This could be done by optimising the model like in the present training phase with the difference that it is not only done in advance to tracking but also in parallel to tracking the object. That is using free resources on the host computer to improve the model during run time.

6.2 Coupling the models of both cameras

In the present version of the program, fusion of the results of both trackers is done after recording the data. This rather passive fusion gives the opportunity to compare and evaluate the data. Another approach of interest would be an active fusion of results, meant as a coupling of the models through their iteration's results during run time. One could use the calibration data to build the differences in the center of gravity's positions regarding both trackers while running them. By doing so it would be possible to automatically reinitialise one tracker if both, its own error prediction and the difference of tracking data, have high values.

Another possibility, coupling both trackers in an even closer manner, would be to use calibration data again, computing an estimate of the objects position, derived from the data of the still accurate tracker. This could then be used in the reinitialisation of the other tracker, leading to a yet more robust tracking with significantly reduced initialisation time.

6.3 Combination with other algorithms

The combination of aamTracker with other algorithms could also be a promising possibility. M. C. Santana is developing a tracker based on skin tone detection called ENCARA2 designed to track iris [11]. The tracker also uses an OpenCV implementation of the Viola Jones face detector to gain an initial estimate for the search radius and position (see *figure 6.1*). As being pretty good, a possible improvement could be made when using aamTracker for the initial guess, since the current solution is fast but not as robust as it could be. Having access to the shape of the face and its position could give the good hints the iris tracker could need to converge in less time.

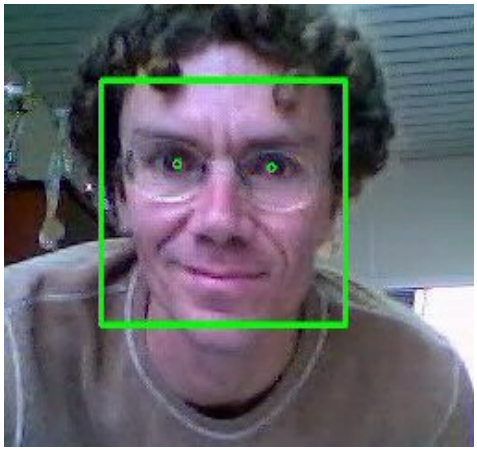

figure 6.1. M.C. Santana's ENCARA2

Literature

[1] F. Martin, R. Horaud: *Multiple-camera Tracking of Rigid Objects*.
INRIA, Rapport Récherche No. 4268, 2001

[2] M. Isard, A. Blake: *CONDENSATION - Conditional density propagation for visual tracking.*
Int. Journal of Computer Vision, 29, 1, pp. 5-28, 1998

[3] S. Grange: *TLIB overview.*
VRAI, EPFL, 2003. http://vrai-group.epfl.ch/tlib/

[4] Mikkel B. Stegmann: *The AAM-API: An Open Source Active Appearance Model Implementation.*
Medical Image Computing and Computer-Assisted Intervention - MICCAI 2003, pp. 951-952

[5] *AAM Tracking.*
IMM, Technical University Denmark, 2004.
http://www.imm.dtu.dk/~aam/tracking/

[6] Cootes, Edwards, Taylor: *Active Appearance Models.*
Lecture Notes in Computer Science, 1998

[7] Cootes: *How to start building models.*
http://www.isbe.man.ac.uk/~bim/software/am_tools_doc/index.html

[8] *Camera Calibration Toolbox for Matlab.*
http://www.vision.caltech.edu/bouguetj/calib_doc/

[9] *Calibrating one camera – first example.*
http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/example.html

[10] *Calibrating two cameras – fifth example.*
http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/example5.html

[11] Santana, Tejera, Gámez: *ENCARA: Real-Time Detection of Frontal Faces*
IUSIANI, Universidad de Las Palmas de Gran Canaria, 2003

Appendix

A.1 How to rebuild everything

This should also solve stream.h and highgui[d].lib problems.

DirextX

1. Open..*directxSDK_root**Samples**C++**DirectShow**BaseClasses**baseclasses.sln* with Visual Studio. Rebuild the both the release and the debug version with *'Erstellen->Batch erstellen...'.*
2. Copy the files strmbase.lib and strmbasd.lib from *..**directxSDK_root**Samples**C++**DirectShow**BaseClasses**Debug* and from *..**directxSDK_root**Samples**C++**DirectShow**BaseClasses**Release* to *..**OpenCV_root**lib.*
3. Correct the path to 'strmbase.lib' to suit path of Visual Studio. This can be done in "*Projekt->Einstellungen>Linker"* .
4. Open *..**directx_root**sdk**samples**C++**DirectShow**Filters**EZRGB24**ezrgb24.sln* and rebuild the both the release and the debug version. (This checks if the previous rebuild was successful.)

HighGUI

5. Open *..**OpenCV_root**_dsw**OpenCV.sln.* Go to *"Erstellen->Batch erstellen"* and deselect all but 'HighGUI', i.e. the Debug and Release versions. Don't select the 'HighGUI' MIL versions unless you really need them (you would need the extra MILibrary). Then rebuild (Neu Erstellen).
6. Open *..**OpenCV_root**lib* on the shell. Decapitalize the files 'HighGUI.lib' and 'HighGUId.lib' by renaming them to 'highgui.lib' and 'highguid.lib'.

The rest of OpenCV

7. Go to *"Erstellen->Batch erstellen"* again and select all this time but 'HighGUI', i.e. the Debug and Release versions. Don't select the 'HighGUI' MIL versions, 'MtlbWrps' or 'Hawk' either. Then rebuild (Neu Erstellen).
8. Add the include and library paths (see appendix) to the DV according to your system configuration (meaning paths here are to be understood as examples that does not necessarily match yours).
9. Then build ProxyTrans (Debug & Release) and copy the folders *..**OpenCV_root**_temp**ProxyTrans_Release* and *..**OpenCV_root**_temp**ProxyTrans_Debug to ..**OpenCV_root**lib.*

VisSDK

10. First install.
11. Add *..\VisSDK_root\lib* and *..\VisSDK_root\inc* to Visual Studio paths.
12. Rebuild all.

CLAPACK (needed for VisMatrix)

13. Download sources from http://www.netlib.org/clapack and install/unzip them.
14. Open the project and build BLAS, CLAPACK, F77 and I77 libraries (!use MFC in a shared DLL!).
15. Rename the libraries (*.lib) to Blas.lib, Clapack.lib, F77.lib, I77.lib for the release-versions and to BlasDB.lib, ClapackDB.lib, F77DB.lib, I77DB.lib for the debug-versions.
16. Copy these to *..\VisSDK_root\VisXCLAPACK* and build the VisXCLAPACK.dll.
17. Set path to it. Now VisMatrix can call the functions wrapped in here.

AAM-API

18. Download sources from http://www.imm.dtu.dk/~mbs/api/ and install/unzip them.
19. Add paths to *..\AAM-API_root\inc, ..\AAM-API_root\diva\inc* and *..\AAM-API_root\lib* to Visual Studio.
20. Rebuild all *..\AAM-API_root\aam-api-lib*.dsw.*
21. Rebuild Diva.
22. Rebuild AAMC.

IPL

23. Download sources and install/unzip them.
24. Add all *.dll's to path. The directory is *..\IPL_root\bin .*

A.2 How to get aamTracker running

Installation

1. aamTracker including all necessary dlls has to be copied to hard disk. Here ‚`RegisterProxyTrans.bat`' is to be executed. Set path to `IPLbin` if not done yet.
2. Both web cams (not of the same type) have to be connected to the PC using different USB hubs.
3. Start aamTracker.
4. Press „*Show Cams →*". Click on the left camera (your view), press „*Set Camera 1*". Click on the right camera, press „*Set Camera 2*".
5. Now press „*Start*" with no filter selected („*none*").
6. In some cases several hubs of the computer have to be tested until both cameras produce fluent video streams.
7. Now set both cameras to a resolutuin of 320 on 240 pixels, a framerate greater 20 fps and equal contrast.
8. The positions of the cameras on top of the monitor should be optimized in a way that during a test run with „*AAM Tracking of faces*" both cameras produce reasonable tracks. The cameras should not stand too far apart as the model supplied is trained on purely frontal views of faces.
9. Now calibration can take place.

Tracking

1. Start aamTracker.
2. Select both cameras (as in installation).
3. Set tolerance to a value between 3 and 6.
4. Press „*Save to file(s)…*" and follow the dialog boxes, leave „*log tracking*" unchecked for now.
5. Start the streams by pressing „*Start*". You see virgin streams.
6. Start tracking by selecting „*AAM Tracking of faces*".
7. When both trackers are initialised, start logging by enabling „*log tracking*".
8. Stop by pressing „*Stop*".
9. The results of both trackers are saved to `output1.txt` and `output2.txt`. Be sure to **make a copy before restarting the tracker**, else the files will be over-written without a warning!