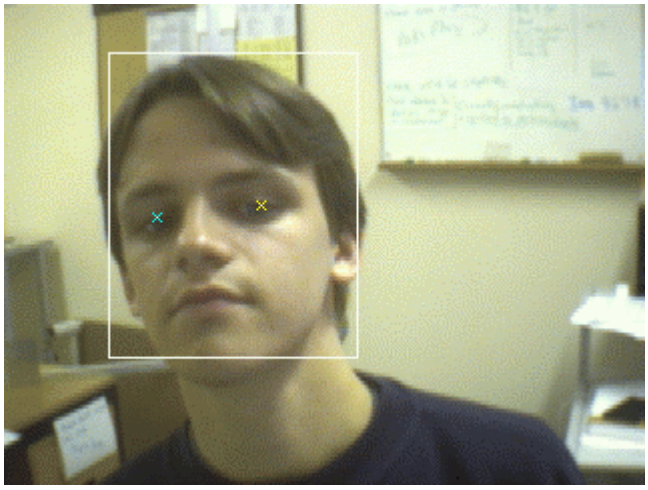
Dunedin, September-December 2000

Jens Willems

Keeping an eye on eyes

A report on the design of a head and eye position tracker for FTVR purposes



On behalf of:
University of Twente, Enschede, the Netherlands
Supervised by prof. Dr. Ir. A. Nijholt

On behalf of:
University of Otago, Dunedin, New Zealand
Supervised by Dr. K. Novins, Dr. B. McCane

Preface

This project was initiated as a practical training period, to allow me to gain working experience outside the University of Twente before finishing my degree.

Because of my interest in visiting and working in New Zealand, I wanted to do an assignment with the University of Otago. My interest in computer graphics and virtual reality made me join the computer graphics research group of this University for doing a project. Research on FTVR systems was being done at the time, using glasses for detection of the eye position, however there was a lot of interest for developing a system that could work without the use of these glasses. Therefore I decided to start this research project.

I would like to thank Mr Kevin Novins and Mr Brendan McCane for their supervision and support throughout the project, and Sui-Ling Ming Wong for proofreading and correcting my report several times (and of course for the cool surprise). Further thanks go out to the rest of the Graphics Research group, it was great to be working with you guys!

Last but not least I want to thank my girlfriend Mariël for supporting me in going to New Zealand, though it meant missing me these months. I know you had a hard time, and I couldn't have done it either without your regular backup!!

Jens Willems

Summary

An eye-tracking system for future use in Fish-Tank Virtual Reality (FTVR) systems is developed. The aim is to achieve reasonable eye-tracking results within real-time constraints, on a standard PC-like machine. The tracking is done in 2 dimensions only, as the system used can see no depth.

The tracker starts off by reducing the image size of the captured image, thus reducing the amount of calculations. Using image subtraction and edge detection the coordinates of a bounding box of the user's face are determined. After this, a number of colour processing techniques is used to determine the positions of areas within this box that have the colour features of eyes. Finally, a positional calculation is performed, using common knowledge about face geometry as well as information about the previously detected eye coordinates, to determine which of the found areas are most likely to be eyes.

The results of implementing each step taken are described, including necessary refinements to initial algorithms to improve performance. Furthermore, optimisation steps taken to increase the overall speed of the system are discussed, keeping in mind the real-time constraint. The image subtraction step combined with the edge detection step performs well after a number of optimisations, almost never resulting in a mistaken bounding box. The colour processing required more development before performing at an acceptable rate. Finally, the positional calculation, as could have been expected, performs quite well on faces viewed from up front but starts making mistakes when the user turns his head. It also appears that dark features in the face, such as dark hair or eyebrows, as well as dark clothing, may sometimes fool this part of the system.

The performance of the system, both in accuracy and speed, is discussed. First, the performance of the head tracking system alone is evaluated and then the performance of the complete system including the eye tracking part. The head tracker appears to be more stable, less likely to misdetect, but the eye tracker is more accurate when detecting correctly. Processing time of the complete tracking is just fast enough to keep the system tracking at 20 frames per second.

We conclude from this research that the head tracker, as developed by us, performs satisfactorily. Our eye tracker, however, requires improvement to function reliably with an FTVR system. We do expect that implementing an eye tracker, instead of just a head tracker, will pay off in an FTVR system as its higher accuracy will increase the realism of the intended illusion. We therefore recommend research on improving the eye tracking system, and also to enhance the system to work with two cameras, enabling the computer to calculate depth as well.

Table of contents

1 Introduction

The use and importance of virtual reality in today's society is growing rapidly. Not only in computer games, but also in industrial applications where a virtual design can give a good impression of a product without actually having to construct it.

But to take a good look at a certain object you need more than just a single image. You may want to walk around your possible new car, look at it from different angles. An image on a computer screen isn't going to do.

Fish Tank Virtual Reality (FTVR) aims to overcome this problem. It is a concept based on the idea that what you see on the screen should appear as if it were a real object behind the glass of your fish-tank monitor, allowing you to see different views from different angles. It would require the computer to know the position of your eyes and adjust the image on your screen accordingly. This requires a system that tracks the eyes of the computer user in space. The aim of the project resulting in this report was to create this eye-tracking system.

This report gives a complete description of the development of the eye tracking system. It starts with giving an exact definition of the purpose of the system to be developed and the requirements it will have to meet.

In chapter 3, a complete overview is given of the way the eye tracker was developed, describing the structure of each step taken in the eye tracking process as well as its *intended* result.

Subsequently, in chapter 4, a detailed description is given of the *achieved* results. Every process step is described in detail: the way it initially was implemented, the way we refined it for better performance and the way we optimised the algorithms for higher speeds. This chapter concludes with an overview of how the final eye-tracking system works. A rough estimation of the processing speed is also given here, describing the contribution of each step taken to the process.

Next, we discuss the performance of the system as a whole. In chapter 5, the effect of environmental factors is discussed, and an overview is given on how quickly and how accurately both head tracking and eye tracking perform. This chapter concludes with giving a brief description on how well both systems perform when used in a simple FTVR system.

We conclude this report with an overview of the project results and point out some recommendations for future research.

2 Problem definition

FTVR creates images that appear to be 3D objects in a box behind the computer screen instead of just flat images on it [REK95]. For this illusion to succeed, the user should be able to see the bottom or sides of an object when looking from corresponding angles at the screen. The image should thus appear to rotate when the user moves his head, just like when walking around a real object standing on a table. To create such an illusion, the VR system should be able to track the position of the head (or even better, the eyes) of the user to change the view of the object accordingly. Some systems exist that achieve this using, for example, VR head gear or glasses of some sort. Experimentation on such a system is currently being performed at the Graphics Lab at Otago University [MAS97, TRE00].

From this experiment, another research question arises: would it be possible for a computer to 'see' the user in front of the screen? That is, can the computer determine the user's position in space without this position actively being 'given away' by a device worn by the user? This is the main question of my research project. The problem definition that fits this question is:

> **Develop a set of routines that is able to track, in real time, the position in space of a user's head or eyes, without the user having to wear an active tracking device.**

The project started with developing a system that performs only a 2-dimensional tracking. We equipped the system with one camera, allowing it only to calculate coordinates in the 2D system given in the image. Allowing the system to see depth as well requires equipping it with a second camera. This extension may be a subject of future research.

We didn't have to start entirely from scratch, because, as mentioned before, a similar system that uses an active tracking device was already set up at the lab, so we were able to use this system as a starting point. Furthermore, research on tracking a computer user has already been performed in the past [ERI99, BAK99], so the task was actually to apply (and maybe extend) this research to the existing software.

Initially, we only used the video capturing and screen display functions provided in the original software; the actual tracking is programmed from scratch. The goal was to implement it in the form of a 'unit' consisting of a set of procedures that can be called by other programs (such as the existing FTVR system at Otago University). The reason for this is that a stand-alone eye tracker will not be very useful as a program; the source code is of much more use as part of a larger system. With this in mind, no serious attention was given to building an interface around it, only a very simple program was created to be able to test the essential routines. Instead, we decided to make the various routines as generally applicable as possible and also as transparent as possible, paying extra attention to correct and clear in-line comments, to allow future programmers to easily understand how to use them.

3 Research method

3.1 Introduction

3.1.1 Equipment

To enable a computer to 'perceive' its user's whereabouts, it needs to be equipped with some sort of sensor that is able to perceive its environment (and the user within it). The software developed here is based on a video capture system. This gives the computer input in the form of subsequent images (video) which it can process. The equipment we started with consisted of a single Unix-based computer equipped with a video capture card and a webcam positioned on top of the user's monitor. The webcam can be set to capture at various resolutions; Because of the speed constraint, we used the lowest possible capturing resolution, 320x200.

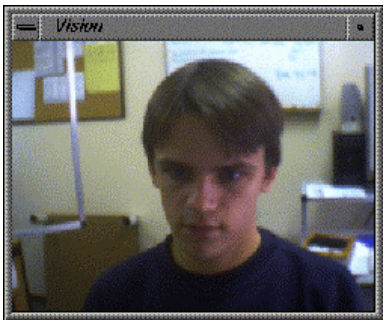


Figure 1: image captured by webcam

With this configuration the computer has a frontal view of the user (Figure 1), enabling it to determine the user's position in two dimensional directions. Possible future extensions of the project include equipping the system with two cameras, thus giving it a 'stereo view' and allowing the computer to compute depth.

3.1.2 Tracking algorithm

Now that we have equipped the computer with an eye, how do we make it see?

Our goal is to track the user's position as he moves. To achieve this, we need to grab a single image from the webcam, process it to find out where the user is at that moment, then grab the next and find out where the user is at the next moment, etc. The essence of tracking a user is to repetitively find out where he is in a single moment. The space in which we search for the user is relatively small, as the user will have to be in front of the screen to see the Virtual Reality. Therefore we need to search for the user in an approximate frontal view only.

Now the problem is reduced to processing a single image in a relatively short time. How do we get a computer to analyse a large blob of data called an image and transform it into a set of neat coordinates?

3.1.3 Optimisations

It is important that the resulting software be able to track the user's eyes in real time, so that the computer has an accurate and up-to-date position of the user. To have the computer give a real-time impression of the eye-tracking results, a reasonable frame rate is needed. 24 frames per second (fps) is the frame rate used for movies. We tried to achieve this frame rate, but initially we accepted a frame rate of about 20 fps. This frame rate can give the user a reasonable perception of the eyes being tracked real-time. Furthermore, if we achieve this frame rate with the current equipment, a future system with more processing capability should, of course, be able to perform the calculations fast enough to achieve the desired 24 fps (The Unix machine we used for the eye tracking has only a 180 MHz processor, which is far from the state of the art machines already).

Unfortunately, this frame rate restriction limits the possibilities for calculating eye positions, as the computer needs to be able to perform the analysis at least 20 times

per second. To overcome this, we introduce several optimising routines into the software, either to limit the amount of calculations needed or to speed up the calculations.

The programming strategy used for the software was:

- implement a single part of the algorithm;
- test if it works at all;
- consider if its value is significant (it may work perfectly, yet not return significant additional data);
- consider the possibilities of refining the calculation to become more accurate;
- consider if the frame rate is still acceptable, and if not try to optimise the newly implemented part;
- if necessary, alter other parts of the algorithm to speed up the system.

3.2 Frame size reduction

This part of the algorithm performs no eye tracking by itself, but it is introduced because it reduces some unwanted noise effects generated by the image subtraction introduced in §3.3, and it speeds up some eye tracking steps. Both noise and speed effects are discussed in the following paragraphs, wherever they apply.

The algorithm simply consists of reducing the image size, averaging the colour values of a number of pixels. The advantage of this algorithm as opposed to capturing an image of smaller size was that this way a single noise pixel will be "averaged out" by its neighbours' values. Furthermore, the results of various algorithms can be transformed back to the original frame, using this frame for algorithm steps yet to come. By doing this, precision of the results can be preserved.

3.3 Image subtraction

The first technique we implement is image subtraction. It was also applied by Andrew Erickson et al. [ERI99]. The idea is to capture a single calibration image, which should be an image of the background (with *no* user in it). After this, when the tracking starts, the computer compares the image being processed (*with* the user in it) to the calibration image, and 'subtracts' them. The dots of the same colour are made black (because they should belong to the background). The dots of different colour are considered part of the user's image, so these stay in the colour they are. Ideally, the resulting image will contain only the image of the user surrounded by black.

3.4 Face edge detection

This step of the algorithm tries to find a small rectangle within the image that contains the user's eyes. The image resulting from the subtraction step can roughly be divided into two areas: a black area and a non-black area containing the user's image. This feature is used to detect the x- and y-coordinates of the edges of the face. Calculating the average x- and y-coordinates of the non-black area gives us a spot that is likely to be somewhere in the centre of the user's image. Using this spot we can trace the image in x- and y-directions until we reach a black pixel (the pixel is made black during the subtraction step, so we know its exact colour values).

3.5 Colour filtering

After determining the face edges, we now have a box containing a human face. Certain colour features of faces will be used to locate the eyes. This part of the tracking algorithm concentrates on the colour intensity features of a human face, as was also done by Vera Baki_ [BAK99]. It is based on the fact that the colour intensity of the area around the eyes is relatively low (i.e. it is relatively dark) with respect to the rest of the face. This is due to the eyes being positioned deep into the face, so they are often shaded.

We then implement a colour filtering algorithm that identifies a number of relatively dark areas within the facial region, expecting the eyes to be somewhere within two of these areas.

3.6 Neighbourhood analysis

After finding a number of dark areas we can try, for example, to calculate the 'average position' of each dark area, by calculating the averages of the x- and y-coordinates of every pixel in a dark area. This is the approach taken by Baki_. However, a problem arises when a dark area consists of more than just an eye. For example, when areas of dark hair and eye regions appear interconnected in the image. In this case, calculating the average position would probably result in one entirely wrong average instead of in two averages, one of which is the right one.

To bypass this problem, we perform a neighbourhood analysis that calculates a score for each pixel, based on the intensity values of its *surrounding* pixels, determining how 'centrally located' it is within such a dark area. The main idea is that pixels along the edge of such an area get low scores, whereas pixels with lots of 'significant' pixels around them will get a higher score.

The intended effect can best be described with a simplified example. Consider two dark regions interconnected like the halves of an hourglass (Figure 2). We want the centre of each region, but calculating the average of the connected parts will result in a point in the thin centre of the hourglass (the blue dot). When considering the neighbourhood of each pixel in the region however, the blue dot will get a low scores as it is surrounded by lots of unimportant pixels (as shown by the blue circle). The pixels in the centre of each part of the hourglass (the green dots) will get high scores.

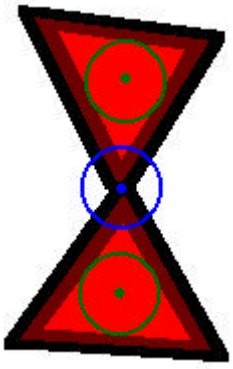


Figure 2: intended effect of neighbourhood analysis

After this calculation we drop the pixels with the lowest scores, the black and dark-red areas in the example. Thus we get rid of 'hourglass-like' connections in dark regions. Ideally, we should now have a number of dark blobs, smaller because the edges were stripped, and also not interconnected anymore. So now we can continue on the path shown by Baki_, calculating average positions of each blob found. The result, ideally, is a small number of single pixels (order of magnitude: 20 or less), some of which may be noisy background pixels, some are probably the user's hair, mouth or possibly dark clothes, and two are the eyes.

The intended effect of this algorithm may appear to be quite similar to the effect of the erosion algorithm used in image morphology. The erosion algorithm scans the edges of a blob, removing everything within a certain distance of the edge, thus

reducing the size of the blob and removing small details along the edges (See [GON92] for an exact description). There is, however, one important difference: the neighbourhood analysis described here will *not* completely wipe out small blobs inside the image, as opposed to the erosion algorithm. If a blob consists of only one pixel, this algorithm will keep the pixel in the list, whereas the erosion algorithm would remove it. Because the eyes are quite often detected as relatively small blobs, the erosion algorithm can not be used: it would then wipe out the crucial candidates.

3.7 Positional calculation

After performing this neighbourhood analysis a small number of eye candidates will be left in the image. Now we use an algorithm that calculates the positions of a set of ideal eyes relative to the position of the facial region. This position is then compared with the positions of the candidates found, using the positional difference to calculate a score. The further away a candidate spot is from the ideal position, the lower its score. Furthermore, we use the position of the previous set of eyes detected, assigning a higher score to the candidate positions closer to the last set of eyes detected.

This calculation is first performed for the left eye only. The spot found to be the left eye is then used as a reference point for calculating the right eye. Thus the knowledge of a certain distance existing between both eyes is incorporated.

3.8 Overview

The positional calculation is the final one, leaving us with a set of coordinates within the image captured by the webcam, indicating the position of the user's eyes. The eye-tracking is now complete. A small overview of the entire tracking process is given in Figure 3. The chain displayed is executed for every frame analysed: a frame is captured using a video capturing routine. It is then reduced to a smaller size, increasing both speed and accuracy. The image is then subtracted and edge detection is performed, resulting in a small part of the image containing the face. This facial region is then subject to an inverted colour filtering, leaving only the darkest parts of the face available for further processing. On these dark parts a neighbourhood analysis is performed, removing large parts of the dark areas' edges, while never entirely removing blobs. This leaves us with a small amount of candidates, which are then subject to a positional calculation resulting in the two sets of coordinates indicating where the eyes should be.

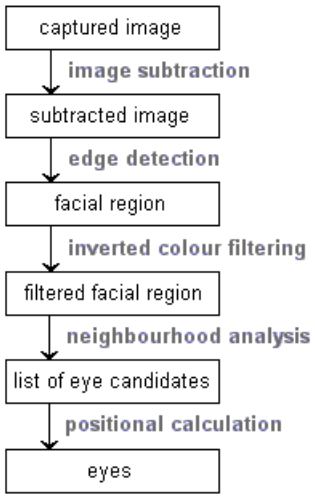


Figure 3: overview of tracking process

4 Research results

4.1 Introduction

This chapter describes in detail the results of the various research and programming steps described in chapter 3. Appendix 8.1 shows the contents of a readme.txt file included with the source code, giving an overview of the file structure of the code. In every paragraph we will refer to the corresponding routine in the source header file, which can be found in appendix 8.2.4 (except for the frame size reduction routines). The source code can be found in the corresponding unit, which is in appendix 8.3.3. The way these units can be used is shown in appendices 8.3.5 and 8.3.7.

4.2 Frame size reduction

The step of frame size reduction is not part of the eye tracking itself. However, it is introduced because it has a positive effect on the accuracy of the edge detection, as will be explained in §4.4, as well as on the speed of the overall process. The corresponding routines are `reduce` and `reduceDouble` in appendix 8.2.5.

4.2.1 Initial Algorithm

The algorithm divides the image into 4-pixel squares, and calculates the average colour values of each square. This results in one set of colour values used for one pixel. The image size is thus reduced to a quarter of its original size.

4.2.2 Effect

The reduction step improves performance significantly, especially in accelerating the program. The overhead generated by the extra pre-processing step is completely annulled by the acceleration due to the reduced amount of calculations needed. The reduction step even managed to accelerate the initial program from an average 22 fps to an average 29 fps, a visible acceleration of about 32 %. However, the 29 fps appeared to be a hardware constraint induced by the video capturing card not being able to capture frames at a higher speed. A significant number of calculations could be added to the algorithm before the framerate dropped below 29 fps again.

As expected, the performance of the face edge detection is also improved. The amount of incorrectly detected edges is reduced considerably, as can be seen on screen when visualising the result of the tracking process.

As the reduced frame appears to be still large enough to slow down the processing speed considerably at certain eye-tracking steps, a double reduction is performed to reduce the frame to $1/16^{th}$ of its size. This reduces initial subtraction precision but, by averaging out certain values in later steps and transforming coordinates back to the original frame, the precision can be kept within an acceptable margin.

4.3 Image subtraction

The corresponding routine is `subtract_image`.

4.3.1 Initial algorithm

The first subtraction algorithm we applied simply compared the images per pixel:

```
For (every pixel in captured image) do
      If (pixel's colour = calibration pixel's colour)
```

```
            Paint black (processed pixel);
```

This appeared not to work the way we wanted. Instead of the background being eliminated, only a small amount of background dots turned black, resulting in a 'bad quality movie' where small black stains flashed across the screen.

4.3.2 Refinement

The reason for this is easy to explain: because of continuous light changes in the room, due to, for example, lamps (which actually flicker at a high frequency) or moving curtains, two corresponding dots of subsequently captured images are seldom of *exactly* the same colour. However, if they both belong to an unchanged background, they are probably very *alike*. So the trick is to eliminate all pixels that are *not too different* from the calibration image. To determine this slight difference, we must split the pixel's 32-bit value into its four channel values (Red, Green, Blue and Transparency) and compare each channel of the pixel. If the difference of each channel is below a certain threshold, the pixels are considered the same and are subtracted from the image (made black). The resulting algorithm:

```
For (every pixel in captured image) do
   {
      For (every channel in pixel) do
         If (channel - calibration pixel's channel) <= threshold
            Set flag;
      If (all channel flags set)
      Paint black (processed pixel);
   }
```

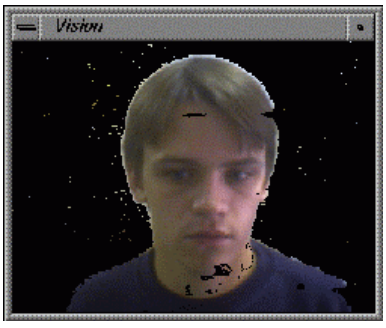


Figure 4: result of image subtraction

Experimentation can be done to determine below what threshold the pixels should be subtracted. It appeared that a threshold of between 25 and 30 results in a subtracted image that was reasonably useful (Figure 4, using a threshold of 28). Furthermore, applying the image reduction as discussed in §4.2, averaging colour values, lowers the total colour difference between calibration frame and tracked frame, allowing an even lower threshold to be used.

4.3.3 Optimisation

This, of course, has a large negative effect on the speed of the algorithm. The computer we used could not keep the frame rate above the required 20.

However, some experimentation confirmed an assumption we made that the transparency value is always set to zero by the capturing card, so ignoring this value does not affect performance at all. Therefore, we could reduce the amount of evaluations needed by $1/4^{th}$, raising the frame rate a little. Nevertheless, it was still

under 20, and with the image subtraction being only an initial part of the calculation, we needed to speed it up a lot more.

The next optimising step taken involves skipping calculations for every second pixel. The results for the first pixel determines whether or not *both* pixels are made black. Applying this method increases the frame rate rather significantly. At this point, a frame rate of about 25 fps can be achieved.

The largest optimising step however was to implement the frame size reduction discussed in §4.1, raising the processing speed to the capture-card maximum of 29 fps.

4.4 Face edge detection

The corresponding routine is `detect_edges`.

4.4.1 Initial algorithm

At first, we simply calculate the average position of the user's image and start tracing outward in three directions: left, right, and up. Algorithmically, for detecting left edge:

```
Set pointer to
   coordinates (average X, average y);
While not (pointed pixel's colour = black)
   Move pointer 1 pixel left;
Get pointer's current X - coordinate;
```

For right and up, an analogous algorithm is used. Because the bottom of the image always contains part of the user's body, the bottom of the face can not easily be calculated the same way. However, acceptable results can be achieved by calculating the bottom edge as a function of the top edge and the difference between left and right edge. The result can be visualised by drawing vertical and horizontal lines across the image at the calculated positions. (see Figure 5).

Figure 5: edge detection

4.4.2 Refinement

The result we achieved appeared to be highly unstable: the noise present in the subtracted image caused the lines to jump across the image, even across the face when a single black pixel appeared on it.
Two solutions come up to reduce this unwanted effect. The first is to reduce, by some way, the noise that causes the effect. This is another one of the reasons the frame size reduction discussed in §4.1 was introduced. The second solution is to refine the tracing algorithm to spot whether the black pixel indicates the face edge or is a noise pixel. We introduce both solutions to the algorithm because the separate solutions don't increase performance satisfyingly, but the combination does.

To refine the tracing step in the algorithm without slowing down too much, we introduce a second trace pointer. The outline in Figure 6 shows how this works, the orange blob representing the user and the black spots representing some noise present in the user's image. Originally, we used one pointer tracing from the average position of the image (the red dot). We now start using two, one placed a small number of pixels (say, 3) before the average value and one placed a small number after. Both pointers then start tracing in parallel, 6 lines apart (the white lines in the image, from the average position outward). They stop tracing only if *both* pointers indicate a black pixel. This way, if a black pixel is a noise pixel, or maybe even part of a small noise blob (as at the blue cross in Figure 6), the other pointer will most probably point at a non-black pixel and thus the trace will continue. However, if the black pixel is an edge, the other one is probably also black, and if not, the next or second next probably will be (the white crosses). The algorithm describing this, tracing right:

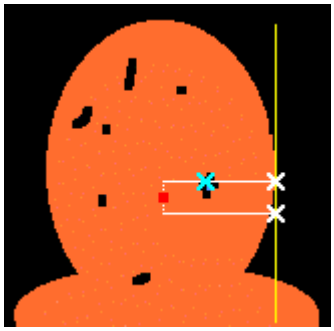


Figure 6: outline edge detection (refined algorithm)

```
Set pointer1 to coordinates (average X, average Y - scan width)
Set pointer2 to coordinates (average X, average Y + scan width)
While not   ((pointer1's pixel's colour = black) and
             (pointer2's pixel's colour = black))
      Inc (both pointers' X - coordinate);
Get pointers' current X - coordinate;
```

The same procedure is used for tracing upward and to the left, to detect these face edges. Large noise blobs are not corrected by this refinement, but it does reduce the chance of miscalculation considerably. To stabilise the edge detection even more, we decided to introduce another trace pointer.

Unfortunately, the results of the edge detection process were still not satisfactory. The problem was that the colour of a certain area in the background appeared to be close to skin colour. This resulted in a large, horizontally stretched noise area through the face as the computer calculated this area to be part of the background. The outcome of this was an unstable edge detection as both pointers sometimes crossed the noise, and then detected it incorrectly as an edge. This can be remedied by inverting the algorithm, ie. start tracing from the outer edges instead of from the centre, and searching for non-black pixels instead of black ones. Figure 7 shows this. As the noise blobs outside the user's image are almost never as big as the one we encountered inside the user's image, this again improves performance considerably.

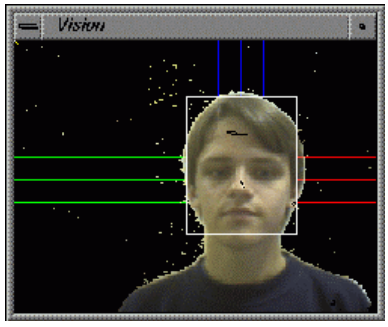


Figure 7: example output refined edge detection algorithm

4.4.3 Optimisation

Most parts of this tracking step are hard to optimise, as all steps have to be taken one by one. However, one optimisation can be made. The pixels relevant for calculation of the "average position" of the user in the image (the red dot in Figure 6) are exactly those identified by the image subtraction step. Therefore, we can integrate the calculation of the average position in the subtraction step, thus saving us an evaluation for every pixel in the image; we simply keep track of the positional data of a pixel if it is not to be subtracted. The resulting calculation takes up so little time that no noticeable drop in tracking speed occurs.

4.4.4 Reducing search space

After performing the face edge detection, we have a bounding box giving a fairly accurate position estimate of the user's head in the image. We now know the rest of the image is irrelevant, because the final targets we are looking for are the eyes and they ought to be somewhere inside the box. Therefore, after detecting the face position we can limit our calculations to the part of the image within the detected edges (from now on called the facial region), thus reducing the amount of calculations needed.

4.5 Colour filtering

The colour analysis routines are `VC_analysis` and `HC_analysis`; the filtering routine is `inv_red_filter`.

4.5.1 Preliminary analysis

To determine a useful filter for identifying the dark regions within the face, we start by analysing the colour features of the face. This is done by taking the colour value for each pixel of one of the colour channels (red, green and blue) and then calculating the average channel value of one *horizontal* line in the facial region. This average value is calculated for each horizontal line in the facial region, and the average values are displayed graphically in the image itself, on the corresponding horizontal line, the calculated average value used as an offset from the left edge of the image. The result is an in-line graph, directly adapting itself to changing facial region contents (see Figure 8).

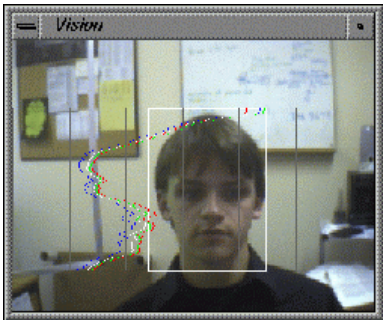


Figure 8: colour analysis performed on facial region

This analysis can be performed for each channel, giving a complete colour analysis of the facial region. It is not used in the final eye-tracking program because the results are too noisy and subject to too many changes. However, features of the analysis results appear very useful in determining the colour processing functions discussed next.

An analog analysis of every vertical line was also performed, but this appeared not to give any useful additional results.

4.5.2 Initial algorithm

As the graphs in Figure 8 show, the colour intensity in the horizontal lines around the eyes shows a little 'dip': the intensity drops there, because the eyes are relatively dark spots in the face. Experiments confirm this average intensity change around the eyes. Because of this, we perform an intensity filtering in the facial region. For efficiency

reasons we use only one channel, because the changes are mostly equivalent for the 3 channels, as Figure 8 shows. We picked the red channel, as this channel's intensity changes were more extreme than the other channels' changes. (Baki_ also uses this channel). As a consequence, a clearer outline of the eye regions can be detected. The initial filtering filters out all but the red channel, and furthermore filters out all pixels with a red intensity value *above* a certain threshold (as the *darkest* pixels are the interesting ones). The coordinates of the remaining ones are stored in a list, because for future calculations only these pixels need to be processed. Algorithmically:

```
For every pixel in Facial region do
      {
         set green to 0;
         set blue to 0;
         if red > FILTER_THRESHOLD then
            set red to 0;
         else
            {
               set red to maximum;
               insert_in_list(pixel)
            }
      }
```

To make the effect more visible in the process we make the remaining pixels entirely red (maximum channel value), so that the darkest pixels become the brightest ones (except for the black ones, of course). Figure 9 shows the result of this inverted colour filtering. As this figure shows, the eyes are visible quite clearly as two large blobs of red near the centre of the facial region.

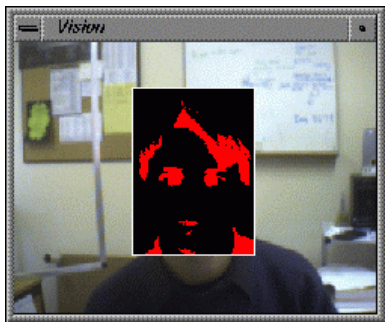


Figure 9: inverted colour filtering

4.5.3 Refinement

As opposed to the filtering that was initially performed, we decided not to turn the important pixels plain red, as shown in Figure 9, but to invert them, so that the computer could still distinguish between darker and brighter pixels.

Another refinement step in the inverted colour filtering is to use a variable filtering threshold. When the user moves closer to the camera or away from the camera, the overall average colour intensity of his face is affected. Although the computer cannot see real depth, this movement can be deduced by the size changes of the facial region. If the facial region becomes smaller, the user probably moves away from the camera. Therefore, the size of the facial region can be used as input for a function determining the threshold value of the colour filtering routine.

As this refinement step is not very reliable however, we can replace it with a variable-threshold algorithm that keeps track of the number of non-black pixels left after

filtering. The threshold is then simply adjusted for the next frame if this number is too low or too high. This thresholding algorithm appears to be a lot more reliable, after determining a suitable number interval.

Adding these refinements we finally get the following algorithm:

```
For every pixel in Facial region do
            {
               set green to 0;
               set blue to 0;
               if red > var_FILTER_THRESHOLD then
                  set red to 0;
               else
                  {
                    set red to (maximum – red);
                    insert_in_list(pixel);
                  }
            }
if ( # list elements > MAX_NO)
   set var_FILTER_THRESHOLD   to var_FILTER_THRESHOLD-1;
else if ( # list elements < MIN_NO)
   set var_FILTER_THRESHOLD   to var_FILTER_THRESHOLD+1;
```

4.5.4 Optimisation

This part of the algorithm appears to be the slowest one in the entire eye-tracking process, lowering the frame speed by at least 5 fps (It is the first step that causes the process speed to drop below the capture card's maximum again, about 5 fps below it). The framerate resulting after adding this step to the process is about 24 fps. It is however also the most important and it performs fast enough to keep the framerate above the initial requirement of about 20. The main reason for its slowness is the fact that during this step the candidate list is created, allowing the future steps to skip all non-important pixels and performing the relatively time-expensive neighbourhood analysis and following routines on only the pixels in this list. However, the list is still relatively long at the point of colour filtering, covering every pixel dark enough not to be filtered out. This corresponds with about 100 elements. A list this size has to be created over again for every single frame. It is therefore a relatively slow step, but it does result in speeding up the subsequent steps.

4.6 Neighbourhood analysis

The corresponding routines are `centrality_score` for calculating the scores, and `remove_least_centrals` for removing the edges. The algorithm for calculating the average position of each blob is `blob_averages` (based on a floodfill algorithm).

4.6.1 Initial algorithm

The neighbourhood analysis is performed to remove 'hourglass-like' connections between blobs that are part of more than one feature in the face. The initial neighbourhood analysis algorithm checks for every pixel left in the list, the pixel itself and the eight surrounding pixels. The intensity values of every pixel's red channel are averaged and the result is stored in the central pixel's green channel, both to save memory, as this channel isn't being used anyway, and, more importantly, to visualise the result of the process. Algorithmically:

```
For (every pixel in list) do
    Set green to average (red value of eight surrounding
pixels);
```

After performing this calculation on every pixel in the list, leaving all other pixels a score of zero, we perform a conditional check for each pixel, comparing it with the neighbourhood scores of its neighbouring pixels. If any of the pixels checked has a higher score than the one being processed (this meaning the pixel in the original frame has more *dark* around it), the processed one is dropped from the list of candidates, as a more centrally located pixel exists close to this one. The algorithm:

```
For (every pixel in list) do
    If (green value of a neighbouring pixel)
      > (this pixel's green value)
        Remove_from_list(this pixel);
```

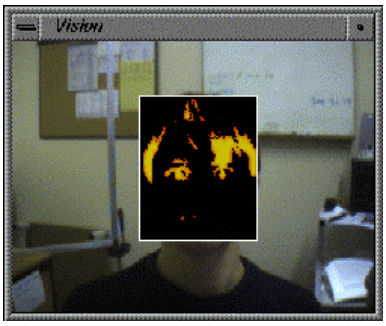


Figure 10: result neighbourhood analysis

Figure 10 shows an example of the neighbourhood analysis performed after the inverted colour filter described in 4.5. Every pixel that was turned red there gets a score which is stored in the green channel. Therefore, a high score will turn the pixel yellow (red + green) while low scores leave the pixel relatively reddish. As the image shows, the edges of each region are less yellow and more red than the centres are.

4.6.2 Refinement & optimisation

Our experiments showed that too many of the neighbourhood scores were identical, resulting in too many connections between blobs that should have been dropped. We therefore decided to extend the calculation of the neighbourhood score to encircle a larger neighbourhood. Both a 5x3 (5 wide and 3 high) and a 5x5 pixel neighbourhood calculation were tested, the latter one being – as could be expected – the better, yet slower one.

A remarkable experiment, however, was to apply this neighbourhood analysis on the reduced frame instead of the original one. This requires the filtering process also to be applied on the reduced frame, as the filtering process returns the initial list of pixels to

be analysed. However, this resulted in no noticeable performance loss. This frame shifting results in both faster and better calculations. Faster, because once again the amount of calculations is diminished, for both filtering and neighbourhood analysis, but also better because a calculation is performed on a 5x5 pixel square *of 4x4 pixel averages*, thus more or less performing neighbourhood analysis on a 20x20 pixel square. The results of the neighbourhood analysis improve significantly because of this.

4.7 Positional calculation

The corresponding routine is `get_eyes`.

4.7.1 Initial algorithm

Initially the ideal position of the eyes is calculated, with respect to the facial region. This ideal position is assumed to be at 50% of the facial region in Y direction, and at 30% and 70% for each eye respectively, in X-direction. This corresponds with eye positions as perceived by the computer when the user is looking straight at the screen, without any tilting or turning of his head in any direction.

These ideal coordinates are then compared with the coordinates of every eye candidate left in the list. The two candidates which are closest to both ideal coordinates, distance at this moment measured as X-distance + Y-distance, are selected as the winners.

4.7.2 Refinement

As can be expected, the initial algorithm does not perform satisfactorily. The main reasons for this are that so far no difference between left eye and right eye is recognised and that distance between both eyes is not yet incorporated.

The first enhancement is therefore to make two passes along the list of candidates: first search for the left eye, then use its coordinates to locate the right eye. We can now use our knowledge that the right eye should be located at more or less the same height as the left eye, and a certain distance to the right of it. This information is added in the search for the correct candidate.

Another improvement that can be made was the use of weight factors added to, for example, vertical and horizontal difference. Testing showed that users are more likely to turn their head slightly sideways and less likely to turn them up or down. However, above and below the eye, some deceptive candidates usually still exist, for example mouth, nostrils or hair. It therefore turned out to be very useful to assign a higher penalty to vertical deviation than to horizontal deviation from ideal eyes.

The final refinement added to this calculation is the incorporation of the previous detection results. These results are added to the formula, again assigning a higher penalty to a higher deviation.

The final formula, two passes with all factors included:

```
For every candidate do
   {
      Left_eye_score = Initial_value
                         - WF1*X_difference_ideal_left
```

```
                        - WF2*Y_difference_ideal_left
                        - WF3*X_difference_previous_left
                        - WF4*Y_difference_previous_left;
   }
LEFT_EYE = best candidate;
For every other candidate do
   {
      Right_eye_score = Initial_value
                        - WF1*X_difference_ideal_right
                        - WF2*Y_difference_ideal_right
                        - WF3*X_difference_previous_right
                        - WF4*Y_difference_previous_right
                        - WF5*X_diff_from_ideal_dist(LEFT_EYE);
   }
RIGHT__EYE = best candidate;
```

Where WF1-WF5 are the corresponding weight factors. The coordinates of LEFT_EYE and RIGHT_EYE are then returned.

4.8 Overview final eye tracking process

After performing the positional calculation, the eye tracking is complete. We will now give a brief overview of the structure of the final tracking process as a whole:

- first, we capture an image from the webcam. The webcam is set to capture images at a resolution of 320 x 200 pixels, 32 bits colours. This image is stored in memory. The capturing can be done at a framerate of at most 29 fps.
- The image stored in memory is first reduced in size: a new image is created of $1/16^{th}$ the original image size (80 X 60), each pixel having the average colour value of its corresponding 4x4 pixel square in the original image.
- After this, image subtraction is performed on this reduced size image. The image is compared with a calibration image containing the view of the webcam without a user therein. Every pixel is compared, and pixels with a colour value difference less than a set threshold value are made black, leaving a (reduced size) image with only the user therein.
- Using the subtracted image, edge detection is performed. From the left edge of the image, three pointers start scanning for non-black pointers. When all three pointers have found a non-black pixel the edge of the user's head is found. This procedure is repeated for the right and upper edges. Their results are used for determining the bottom edge. We now have the coordinates of a box containing the user's head in the image. This box is shown in Figure 11. *Head detection is now complete*. All following calculations, in search for the user's eyes, are now restricted to this box, the facial region.

- On the facial region we apply a colour intensity filter. Every pixel's colour intensity is checked against a *variable* threshold. This threshold depends roughly on the size of the bounding box, and is adjusted after detecting every frame, aiming to leave about 100 pixels in the box. These pixels are added to a list of potential candidates.
- A neighbourhood analysis checks for every pixel in this list, how close it is to an edge of a blob, removing edges where possible without completely erasing blobs. As a result, previously interconnected blobs containing multiple facial features are not connected anymore. The blobs still existing are removed from the list and replaced by a single pixel at each blob's average position. This reduces the list size to about 10.
- Finally, a positional calculation is performed on the pixels still in the list, assigning a score to each candidate depending on distance to an ideal eye position, distance to the other eye and distance to its previously detected position. The one with the highest left eye score is specified as left eye and the one with the highest right eye score is specified as right eye, as shown in Figure 11. *The eye detection is now complete*.

Figure 11: Example tracking result

5 Overall Performance

5.1 Environment sensitivity

To achieve one of the main purposes of the project, tracking the user's eyes *without the user having to wear an active tracking device*, we use a webcam to record his movements. The images recorded by the webcam are of course subject to changes in the user's environment. An ideal eye tracking system is completely unaffected by anything happening in the environment. But because a change in the environment will always affect the image recorded by the web camera, the tracking results will always be affected.

The eye tracking system we developed appeared to be able to operate in different environments as long as not too many changes occur during tracking. The calibration followed by image subtraction filters out virtually all background in the image, as long as it does not change during running. However, changes while running, such as turning off a light switch or another person walking by in the background, do affect the head-tracking part significantly, and as the eye-tracking relies on accurate head-tracking, this is also affected severely. This means that, in general, the background can be of any colour or pattern, but some kind of lighting control, for example by blinding windows, is preferable. Persons or objects moving by in the background should also be avoided, though the process should be able to track correctly again after the background noise has disappeared.

As the tracking relies on comparing the recorded image with a pre-recorded background image, moving the camera itself while tracking, or between calibration and tracking, will have a severe negative effect on the tracking process that can usually only be undone by recalibrating.

5.2 Tracking accuracy performance

During the system development the various algorithms were subject to extensive testing and adapting to get them performing as optimal as possible. Most of the tests were done by running the algorithm and displaying the results in the image itself, thus getting an indication of the algorithm's strengths and weaknesses. This is a very good way of finding out whether or not the performance of the algorithm is sufficient as well as finding wrong program reactions to specific user movements.

However, this testing method makes it very hard to quantify results. The only way we could express tracking results numerically was to record a tracking sequence and then to manually count frames and tell in how many frames the eyes were correctly tracked. This method however appeared to be highly unreliable as the tracking process can be sensitive to environmental changes. Furthermore, the eye-tracking part is dependent on specific user movements. A different head turn in a second test run appeared to affect accuracy. During development, the most algorithm refinement steps were therefore taken to allow the tracker to deal correctly with various head movements. These environment dependencies made it virtually impossible to give a numerical indication of how well the tracking process performs. This is why we decided not to quantify the performance of the eye tracker but give an impression in natural language instead, as this will probably give the reader of this report much more insight on the performance of the system.

The tracking process can roughly be divided into two parts: the head tracking and the eye tracking. Both parts will be discussed separately.

5.2.1 Head tracking accuracy performance

The final head tracking process as described in chapter 4 does of course rely on the environment restrictions given above. However these are usually not too difficult to maintain, and given these restrictions the head tracking process performs quite well. The improvements discussed in §4.3.2 and §4.4.2 as well as the frame size reduction discussed in §4.1 are essential for the head tracking process to succeed, but the combination results in accurate head tracking for all but very exceptional movements. Assuming the user will keep his head visible to the camera and will not tilt his entire body to one side a lot more than the regular computer user does, the head tracking performs accurately almost permanently. We therefore consider the development of a head tracker a success.

5.2.2 Eye tracking accuracy performance

The final eye tracking process relies a lot less directly on the environmental restrictions given above, but it is completely dependent on accurate head tracking, as it uses the tracked head coordinates as a starting point from which to continue. Therefore, if the head tracking fails, the eye tracking will virtually always fail as well. Due to this dependency the eye tracking can, of course, never be more accurate than the head tracker. It was therefore also a motivation for us to pay special attention to tracking the head accurately.

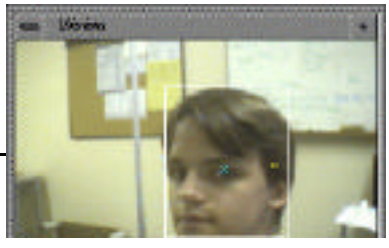
The colour intensity filter will of course be affected by lighting changes, but the use of a variable filtering threshold as discussed in §4.5.3, originally intended to make up for (much more significant) intensity changes in the user's face as he moves, also diminishes this effect.

Tests point out that the following process, neighbourhood analysis, is reasonably capable of filtering out irrelevant pixels in the image while still keeping both eyes in the list. However, it does not always succeed as the eyes sometimes appear to be too much a part of another blob in the filtered region.

An important weakness of the eye tracking algorithm appears to be the positional calculation. One of the reasons is of course that it has to rely on its preceding algorithms, which appeared not always to get satisfactory results. Furthermore, it relies for a significant part on the user having his face positioned toward monitor or camera. It is very sensitive to head turning by the user.

The main problem we encountered however, was one we encountered several times throughout the eye-tracking part of the project: the user face that is captured by a webcam is subject to many changes and uncertainties. Some examples include the colour of the user's hair or clothes, or, more subject to change: whether or not the mouth is open, or the nostrils are visible or not. This kind of uncertainty is responsible for certain dark areas in or close to the face, that are sometimes there, sometimes not. If a certain feature is always there, you can use its presence. If it's never there, you can ignore it. But if it's *sometimes* there, how do you deal with it? It appears very hard to take into account the fact that it is *unknown* whether or not a certain feature is present.

The overall performance of the eye tracking system turned out to be relatively successful when the user keeps looking



at the screen while he moves. However, turning the head away could quite easily, though not always, confuse the system, into pointing at ears or hair instead of eyes, as Figure 12 shows. The system then recovers when the user turns his head back in a straight position.

5.3 Process speed performance

We performed some tests on the speed of each part of the final algorithm, taking into account the real-time constraint. It appears that the set of calculations performed to track the eyes in a single frame is just small enough to keep the process going at the initial real-time goal of 20 fps. Table 1 shows what every process step contributes to the total processing time of a single frame, as far as we could make a fairly accurate estimation. The first line of capturing an image is shown in grey as this step is actually performed by the image capture card and not by the main process. It therefore takes place in parallel to the subsequent steps. We know that the first three steps (reducing image size, subtracting the image and edge detection) can together be done at a speed higher than the image capturing takes place. Adding colour filtering causes the process to need 13 ms per frame more, so this part takes up at least 13 ms. The neighbourhood analysis needs another 4 ms to complete. The positional calculation (actually some arithmetic performed on only a very small amount of data) does not add any significant time to the calculation.

Process step	Estimated step time (ms/f)	New total process time (ms/f)	New framerate (fps)
Capturing an image	34	34	29
Reducing image size	< 34	34	29
Subtracting image	< 34	34	29
Edge detection	<< 34	34	29
Colour filtering	> 13	47	24
Neighbourhood analysis	4	50	20
Positional calculation	< 1	50	20

Table 1: calculation time per process step

5.4 Applying the tracker

The ultimate motive for developing an eye tracker, as stated in chapter 2, was to apply it to an FTVR system, allowing a user to look at an image as if it were a real object behind the screen. We therefore decided to perform a small experiment to get an idea of how well the developed eye tracking system would work when actually used with such a system. We developed a simple openGL program that could generate a 3D scene adjusting the view to eye coordinates as given by the eye tracker. This code can be found in appendix 8.2.7. As the openGL system can only use one viewpoint for creating the image, for this test we used only one generated eye coordinate, telling the user to close his other eye. This 3D system was first tested using only the head tracker, generating some eye coordinate using only the detected face edges. After this, we used the eye tracking system as a whole.

5.4.1 Applying the head tracker

It appears that the system using only head tracking functions quite well. The object projected on the screen is able to create the illusion reasonably, without making too many mistakes. However, head turning can of course get the scenery somewhat skewed as the eyes are not exactly where the head tracker calculated they should be.

5.4.2 Applying the eye tracker

The application that uses the entire tracking system is able to generate a more realistic 3D view of the scenery presented, as far as the detection is correct. However, since the head turning can cause the eye tracker to detect the eye incorrectly quite often, this application turned out to be quite unsteady, with the entire scene sometimes flashing across the screen, corresponding with wrongly detected eye coordinates jumping across the recorded image.

5.4.3 Conclusion

Comparing these results we come to the conclusion that tracking the eyes can give a more realistic impression of an object on a screen than merely tracking the position of the head. However for this eye tracking to succeed, a system detecting eyes more accurately than the one we developed in this project, would be required. Furthermore, our eye tracker, using one camera, can only calculate the user's position in 2 dimensions, the computer does not have enough information to derive how far away the user is from the camera. To apply our eye tracker we therefore had to assume the user's head was at a certain fixed distance from the camera. When using two camera's instead of one, a system can be developed that is also capable of deriving this distance and can therefore calculate the user's exact position in 3 dimensions. This would allow the FTVR system to generate an even more realistic view.

6 Conclusions and recommendations

In this report we have described the development of a tracking system that is able to track the head and eyes of a person using images recorded by a webcam.

The program we have developed is able to track the user's eyes at an acceptable speed, though not a very high one, without using extraordinarily high-end equipment. We can therefore expect the system to run more smoothly on a system with higher capacity. When only head tracking is performed, the system is able to run smoothly with our equipment.

The head tracking part of the system performs quite well. Using the image subtraction method, the user's head can be detected accurately most of the time and in any environment, as long as this environment's lighting conditions are kept constant and there is no background movement. We therefore consider the development of the head tracking system a success.

The eye tracking system as it is now performs all right, detecting the eyes correctly most of the time, as long as the user does not turn his head. However, a better eye tracking system would probably be required for use in a real FTVR system. Furthermore, a system that does allow head turning to a greater extent would be a considerable improvement, especially for the intended application of an FTVR system. This would most probably make eye tracking preferable over head tracking as it does enable an FTVR system to create realistic images more accurately.

To achieve this greater tolerance, maybe the colour intensity filter combined with the positional calculation could be advanced. This would probably result in more sophisticated pattern recognition techniques. However, we deliberately decided to keep these techniques simple to achieve the real-time constraint. Therefore applying more advanced techniques will require more processing power. Another option would be to try developing an alternative technique. When colour filtering is still used, a higher processing resolution may boost performance, but once again this requires more processing power.

Finally, to make an FTVR application work optimally, accurate eye tracking in 3D space is required. Therefore we recommend researching the use of two cameras instead of one, thus allowing the computer to calculate the user's position in 3D space instead of in 2D space.

7 References

[BAK99] Baki_, V. et al., *Menu selection by facial aspect*, Department of computer science and Engineering, Michigan State University, 1999. http://www.cse.msu.edu/~bakicve1/faces/

[ERI99] Erickson, A. et al., *Face and Eye Tracking*, Proceedings of Image and Vision Computing New Zealand, Landcare Research New Zealand, 1999, pp.139-144.

[GON92] Gonzalez, R. C. et al., *Digital Image Processing*, Addison-Wesley, Reading, Massachusetts, USA, 1992.

[MAS97] Mason, T. et al., *The virtual display case*, Computer Graphics International, IEEE Computer Society Press, Los Alamitos, California, 1997, pp. 200-204.

[REK95] Rekimoto, J., *A Vision-Based Head Tracker for Fish Tank Virtual Reality - - VR without head gear* --, IEEE Virtual Reality Annual International Symposium (VRAIS) '95 Proceedings), 1995, pp. 94-100. http://www.csl.sony.co.jp/person/rekimoto/papers/vrais95.pdf

[TRE00] Treadgold, M. et al., *Correcting Brightness Discontinuities in Fish Tank Virtual Reality Systems*, Proceedings of Image and Vision Computing, New Zealand, 2000. http://www.cs.otago.ac.nz/gpxpriv/public_html/homepage.html

8 Appendices

8.1 Overview source code structure

```
 ********** readme.txt ************

This readme file contains a brief overview of the source structure of the
eye tracking source code created by Jens Willems.

besides this readme, these 15 files should be present (if they're not, it
won't work!):

Makefile

alldefs.h

capture.c
capture.h

draws.c
draws.h

gltest.c
gltest.h

main.c

tracker.c
tracker.h

util.c
util.h

vision.c
vision.h

We will start off with a diagram showing all my units and their dependencies
(leaving out the standard libraries and the openGL libraries). After this we
will list all units with a short description of the type of contents you can
find in them. For a description of each routine you can find detailed
comments
in each header file (the *.h files).

the dependency diagram:      layer description:

    alldefs                  "atomic" basics
       |
       |
capture    util              elementary utility routines
       |
       |
vision     draws             core eye tracking and drawing routines
       |
       |
     tracker                 main eye tracking unit
       |
       |
     gltest                         sample application with graphics
       |
```

```
       |
     main                    the program that runs it all...
```

Alldefs is at the top, needing no includes at all, and main at the bottom, directly or indirectly using everything. Every unit in the diagram uses only units from a layer higher than the one it is in. The exact dependencies are shown in the table below.

INCLUDES / UNIT	alldefs	util	capture	vision	draws	gltest	tracker
capture	X						
util	X						
draws	X	X					
vision	X	X	X				
tracker	X		X	X	X		
gltest	X		X				X
main	X	X	X			X	X

In the list below every unit's contents are described shortly.

alldefs.h
At the top of the structure, here all types and global constants are defined.
This unit is included by every other program so all newly defined types are known everywhere, as well as all fixed program parameters.

capture.h & capture.c
this unit contains the video capturing and window manipulation routines: everything you need to grab a frame from a video capture card and pass it on to the eyetracker in the correct format, and everything needed to create a window and display any frame in it.

util.h & util.c
this unit contains a set of small utility routines used throughout the source.
Most routines defined here are only a few lines and perform elementary actions
such as: positioning a a pointer at a certain position in a frame, returning a
certain pointer's position, format conversion routines, standard list-manipulation routines, a swap-function etc.

**** vision.h & vision.c ****
this is the most important unit, containing all routines that perform the various parts of the head- and eye tracking calculation routines, from image subtraction to positional calculation, as well as some analysis routines

draws.h & draws.c
In this unit all routines for drawing the head- and eyetracking and analysis results in any frame are defined. It also contains the framesize reduction routines.

tracker.h & tracker.c
this unit fits the core routines together to form a single routine for tracking one frame, one for drawing the tracking results and initialisation

and ending routines. To simply track frames with the existing code, capture a frame and call these routines whenever desired.

gltest.h & gltest.c
in this unit an openGL scenery is defined and the eye tracker is applied to generate the viewpoint openGL should use for drawing. When this is used, openGL libraries are required (check the includes).

main.c
this is the actual program code that is run after compilation. There is an example loop available tracking a number of frames and displaying the tracking
results in a window. The call that initiates the example openGL scenery is also
there.

8.2 Source header files

8.2.1 Alldefs.h

```
/******* alldefs.h *******

  this file contains all definitions of types and global constants used
  during the eye tracking. The constants are mostly parameters for certain
  tracking steps.
*/

#define NIL '\0'
#define TRUE 1
#define FALSE 0

typedef double HR_TIMER; // for efficiency calculations

/* a couple of macros performing the corresponding mathematical routines */
#define MAX(a,b) (((a)>(b))?(a):(b)) // maximum of 2 numbers
#define MIN(a,b) (((a)<(b))?(a):(b)) // minimum
#define ABS(x) ( (x)<0?(-(x)):(x) )  // absolute value


// all tracking specific type definitions:

/* a frame, this is a single image capture by the webcam, it has a
framehandle
   that points to a memory area containing the image data. the xsize and
ysize
   determine the respective dimensions and therefore also the size of the
   memory block */
typedef char frameHandle;
typedef struct {int xsize, ysize; frameHandle *handle;} frame;

// a single colour channel =1/4th of the colour data of a single pixel
typedef char channel;

/* a set of coordinates to indicate the position of a box in an image.
   standardised values, so can be applied to any image */
typedef struct {int AvgX, leftX, rightX, AvgY, topY, bottomY;}allEdges;

/* a set of dimensions and a handle, containing information about the window
   in which the results can be displayed */
typedef struct {int xsize, ysize; long handle;} windowInfo;

/* a datastructure to be used for storing analysis data when performing
colour
   analyses. */
typedef struct {int red[320], green[320],blue[320];}cData;

/* a linked list datastructure for storing pointers to certain pixels in a
   certain image. The pointer points to a certain area, but the X and Y
values
   are standardised. */
typedef struct listElt {channel *cPtr; int X; int Y; struct listElt *next;
  int pScore;} listElt;
#define LIST_ELT_SIZE 20 // for allocating memory
```

```
/* a structure used to store the X and Y coordinates of a pair of eyes.
   Standardised values. */
typedef struct {int leftX, rightX, leftY, rightY;}setOfEyes;



// a set of global constants (tracking parameters):

/* what framesize reduction to use while processing.
   0 = no reduction;
   1 = single reduction(1/4th  original framesize)
   2 = double reduction(1/16th original framesize) (default)
*/
#define PROCESS_REDUCTION 2

/* whether to display the reduced frame or the original one. Reduced one can
   only be displayed (would only be interesting anyway :) if
PROCESS_REDUCTION
   is set to 1 or 2 */
//#define OUTPUT_FRAME redProcFrame
#define OUTPUT_FRAME procFrame

// image subtraction:
#define CAL_FRAMES_NO 50    // no of frames used for calibration sequence
#define FRAMES_NO 500       // no of frames displayed while tracking
#define COLOUR_THRESHOLD 15 /* Maximum colour difference to be considered
                            identical */
#define X_PIXEL_SKIP 0      // skip # of pixels after calculating one

// face edge detection (standardised values):
#define FACE_SCAN_WIDTH 20   // distance (# pixels) between face tracers
#define Y_CORRECTION -36    // distance to be compensated up from average

/* inverted colour filtering (standardised values) these determine the
   list-length the filtering routine should be aiming for */
#define OUTPUT_LIST_MAX 550
#define OUTPUT_LIST_MIN 500


/* this defines what capture screen size  should be used to capture the
   frames. */

//#define FULL_CAPTURE_SCREEN
//#define BIG_CAPTURE_SCREEN
//#define MED_CAPTURE_SCREEN
#define ORIGINAL_CAPTURE_SCREEN
//#define HALF_ORIGINAL_CAPTURE_SCREEN
//#define HALF_SQUARE_PAL_CAPTURE_SCREEN
  //if using this change the Y_AXIS_PIXEL_ADJUST to 1.0
//#define FULL_SQUARE_PAL_CAPTURE_SCREEN


#if defined(FULL_CAPTURE_SCREEN)
  #define VIDEO_CAP_SIZE_X 640
  #define VIDEO_CAP_SIZE_Y 480
  #define VIDEO_CAP_ZOOM 1.0
  #define VIDEO_CAP_ZOOM_NUMERATOR 1 /* top bit */
  #define VIDEO_CAP_ZOOM_DENOMINATOR 1 /* bottom bit */
```

```
#endif
#if defined(BIG_CAPTURE_SCREEN)
  #define VIDEO_CAP_SIZE_X 576
  #define VIDEO_CAP_SIZE_Y 432
  #define VIDEO_CAP_ZOOM 0.9
  #define VIDEO_CAP_ZOOM_NUMERATOR 9 /* top bit */
  #define VIDEO_CAP_ZOOM_DENOMINATOR 10 /* bottom bit */
#endif
#if defined(MED_CAPTURE_SCREEN)
  #define VIDEO_CAP_SIZE_X 512
  #define VIDEO_CAP_SIZE_Y 384
  #define VIDEO_CAP_ZOOM 0.8
  #define VIDEO_CAP_ZOOM_NUMERATOR 4 /* top bit */
  #define VIDEO_CAP_ZOOM_DENOMINATOR 5 /* bottom bit */
#endif
#if defined(ORIGINAL_CAPTURE_SCREEN)
  #define VIDEO_CAP_SIZE_X 320
  #define VIDEO_CAP_SIZE_Y 240
  #define VIDEO_CAP_ZOOM 0.5
  #define VIDEO_CAP_ZOOM_NUMERATOR 1 /* top bit */
  #define VIDEO_CAP_ZOOM_DENOMINATOR 2 /* bottom bit */
#endif
#if defined(HALF_SQUARE_PAL_CAPTURE_SCREEN)
  #define VIDEO_CAP_SIZE_X 384
  #define VIDEO_CAP_SIZE_Y 288
  #define VIDEO_CAP_ZOOM 0.5
  #define VIDEO_CAP_ZOOM_NUMERATOR 1 /* top bit */
  #define VIDEO_CAP_ZOOM_DENOMINATOR 2 /* bottom bit */
#endif
#if defined(FULL_SQUARE_PAL_CAPTURE_SCREEN)
  #define VIDEO_CAP_SIZE_X 768
  #define VIDEO_CAP_SIZE_Y 576
  #define VIDEO_CAP_ZOOM 1.0
  #define VIDEO_CAP_ZOOM_NUMERATOR 1 /* top bit */
  #define VIDEO_CAP_ZOOM_DENOMINATOR 1 /* bottom bit */
#endif


/* standardisation values. MUST be REALS. based on captured frame size;
   higher values mean higher precision. current precision is maximum useful,
   because this is captured frame size. Higher precision would not make
sense.
*/
#define STD_X_SIZE (float) VIDEO_CAP_SIZE_X
#define STD_Y_SIZE (float) VIDEO_CAP_SIZE_Y
```

8.2.2 Capture.h

```
/*       ***** capture.h *******

  this unit contains the various routines for capturing frames from the
video
  capture card, as well as the routines used for window manipulation.

*/


void init_capture(void);
/* performs initialising sequence for capturing from capture card */


frame capture_frame(void);
/* captures a single frame */
void finish_with_frame(void);
/* gets rid of captured frame, thus preparing memory block for capturing
next
   frame. MUST always be called before capturing next frame */

void capture_end(void);
/* ends capturing sequence, freeing memory area used for capturing. Called
   once after capturing the last frame. */

int capture_image_xsize(void);
// returns x size of frame being captured at the moment
int capture_image_ysize(void);
// returns y size of frame being captured at the moment


windowInfo init_window(char *title);
// Initialises a window, and returns its handle
void display(frame dispFrame, windowInfo win);
// copy frame data into specified window, thus displaying it
void close_window(windowInfo win);
// close given window
```

8.2.3 Util.h

```
/*          ***** util.h *****

   this file contains several small routines that are used throughout the
   tracking routines. They perform small standard calculations for working
   with frames and datastructures.
*/


/****** a set of timing routines ******/

double current_time(void);
// returns current time in seconds

HR_TIMER hrGetTime(void);
// returns current time in milliseconds

void hrWait(HR_TIMER time);
// waits for specified amount of time (in milliseconds)



/* A set of coordinate calculation routines used to be able to:
   - apply calculations in a certain frame to a different-sized frame;
   - quickly retrieve a pixel's coordinates or retrieve a pointer to a pixel
     on the given coordinates
 */

channel *pos_ptr(frame pFrame, int X, int Y);
// returns a pointer to position (X,Y) in frame pFrame

int std_X_pos(frame pFrame, channel *pPtr);
// returns standardised X coordinate of pPtr in pFrame
int std_Y_pos(frame pFrame, channel *pPtr);
// returns standardised Y coordinate of pPtr in pFrame

int standardise_X(frame pFrame, int X);
// transforms pframe's X-coordinate to standard X-coordinate
int standardise_Y(frame pFrame, int Y);
// transforms pframe's Y-coordinate to standard Y-coordinate

int apply_X(frame pFrame, int X);
// transforms standard X-coordinate to pframe's X-coordinate
int apply_Y(frame pFrame, int Y);
// transforms standard Y-coordinate to pframe's Y-coordinate

void colour_pixel(channel *pPtr, char R,char G,char B);
// paints *pPtr in colour values R,G,B

/* ***************** */



/* here follows a set of procedures that perform standard list-operations to
   my tracking-specific :) linked list
*/

listElt **createElt(channel *newChannel, int newCX, int newCY, int
newScore);
// creates a list element holding given values and returns a pointer to it

void insertInList(listElt **list, listElt *insElt);
// inserts *insElt at the head of *list

void insertSorted(listElt **list, listElt *insElt);
// inserts *insElt in *list, sorted by value of score field
```

```
void removeFromList(listElt **list, channel *rChannel);
/* removes first element (if one exists) that satisfies
   (listElt->cPtr)==rchannel
   from list */

int is_member(listElt *list, channel *mChannel);
/* checks if the element listElt which satisfies (listElt->cPtr==mChannel)
is
   in list */



void swap(int *a, int *b);
/* swaps values of variables a and b */


int find_blob(frame bFrame, listElt *list, channel *bPtr, listElt
**blobList);
/* IN  bFrame    : frame to be processed
       *list     : list of pixels to be considered
       *bPtr     : target pixel
       **blobList: -empty- list of pixels
   OUT bFrame    : processed frame (all pixels in blob have blue channel set
to
                   value 255)
       **blobList: list of all pixels in blob
       return    : flag indicating whether (1) or not (0) *bPtr is part of a
                   blob. (if so, its neighbors are also considered)

   find_blob finds an entire blob of connected pixels that are inside *list,
   starting its search at pixel *bPtr in frame bFrame. All pixels belonging
to
   the blob found are returned in **bloblist.

   this is actually a recursive floodfill algorithm, using membership of
   *bloblist instead of their being a certain colour.
*/
```

8.2.4 Vision.h

```
/*               ***** vision.h *****

   This unit contains the various routines called for the actual eye
tracking.
   the main program calls these procedures sequentially to perform the
   tracking as a whole.

*/


frame calibrate(windowInfo capWin,int calFramesUsed);
/* IN  capWin       : window Information
       calFramesUsed: number of frames used for calibration sequence
   OUT return       : calibrationframe

   calibrate captures and displays a sequence of frames, giving the user
time
   to remove himself out of the computer's view. Then it returns a frame
   containing the user's background. This frame is an average of the last
two
   frames captured.
*/


allEdges subtract_image(frame calibrationFrame,
                frame procFrame,
                int cThreshold,
                int xPixelSkip);
/* IN  calibrationFrame: calibration frame as produced by calibrate
procedure
       procFrame       : frame to be subtracted
       cThreshold      : Colour threshold: maximum difference in colour
                         channel value to be considered identical (and thus
                     subtracted)
       xPixelSkip      : number of pixels to be skipped after 1 pixel
                         calculation
   OUT procFrame       : subtracted image
       return          : allEdges structure; ONLY AvgX and AvgY
                         fields are valid!

   subtract_image performs image subtraction on procFrame, using
   calibrationFrame as reference: If the difference of each colour value in
   both frames of a pixel is below cThreshold, pixels are considered part of
   the background and made black. After calculating one, xPixelSkip pixels
are
   skipped before next is calculated.
*/


void detect_edges(frame resultFrame, allEdges *edges, int scanWidth);
/* IN  resultFrame: frame containing subtracted user image
       edges      : alledges structure. Only AvgX and AvgY fields need to be
                    valid. These should apply to resultFrame.
       scanWidth  : The distance between scanning pointers
   OUT edges      : allEdges structure. Now all fields are valid and give
the
                    coordinates of the bounding box of the user's face,
                  hereafter called the facial region

   calculate_edges uses the subtracted image and the average values in the
   allEdges structure to calculate the values for the remaining fields in
the
   same structure (indicating the edges that form a box around the user's
   face). All values are standardised.
```

```
*/


listElt *inv_red_filter(frame calcFrame, allEdges fEdges, int *threshold);
/* IN  calcFrame     : frame to be filtered
       fEdges        : area within frame to be filtered (usually facial
region)
       *threshold    : filtering threshold to be used.
   OUT calcFrame     : frame with filtered area
       *threshold    : threshold used during NEXT call of inv_red_filter.
       return        : all remaining non-black pixels

   inv_red_filter sets all but red channel-values in fEdges area to zero.
   for the red channel, each value above *threshold is also set to zero. red
   values below *threshold are inverted (making them bright instead of
dark).
   the function returns a list containing every remaining non black pixel,
   this is a (still quite large) list of potential eye candidates.
   The value of *threshold is adjusted for use in the next call, if this
call
   resulted in a too large or too small list.
*/


void VCAnalysis(frame pFrame, allEdges *edges, cData *Analysis);
/* IN  pFrame   : frame to be analysed
       edges    : allEdges structure containing coordinates of box to be
                  analysed
   OUT *Analysis: datastructure containing analysis results

   VCAnalysis performs VERTICAL colour analysis on the area defined by
   allEdges in frame pFrame. For every horizontal line in the box and for
   each colour channel the average channel value in this line is calculated
   (thus resulting in 4 averages, one for each channel). Results of this are
   returned through *analysis.
*/


void HCAnalysis(frame pFrame, allEdges *edges, cData *analysis);
/* IN  pFrame   : frame to be analysed
       edges    : allEdges structure containing coordinates of box to be
                  analysed
   OUT *Analysis: datastructure containing analysis results

   HCAnalysis performs HORIZONTAL colour analysis on the area defined by
   allEdges in frame pFrame. For every vertical line in the box and for
   each colour channel the average channel value in this line is calculated
   (thus resulting in 4 averages, one for each channel). Results of this are
   returned through *analysis.
*/


void centrality_score(frame tFrame, listElt *maxPixels);
/* IN  tFrame    : frame to which pixels in *maxpixels apply
       *maxPixels: list of pixels
   OUT tFrame    : frame, with centrality scores stored in green channels,
for
                   each pixel in maxPixels
       *maxPixels: list of pixels

   centrality_score performs a neighbourhood analysis on every pixel in
   *maxpixels. For each pixel a score is calculated that roughly indicates
   how bright it and its neighbours are. Thus, if a pixel has lots of bright
   neighbours, it gets a high score. The result of this calculation is
   stored as the green channel value of the corresponding pixel.
*/
```

```
void remove_least_centrals(frame rFrame, listElt **centrals);
/* IN  rFrame   : frame to which pixels in *centrals apply
       *centrals: list of pixels
   OUT *centrals: list of pixels

   remove_least_centrals performs a check on the pixels in *centrals,
checking
   whether or not they have neighbours with higher scores. If they have,
they
   are removed as the neighbours will be more centrally located.
*/


void blob_averages(frame bFrame, listElt **pixelList);
/* IN  bFrame    : frame to which pixels in *pixelList apply
       *pixelList: list of pixels
   OUT bFrame    : frame, with colour values changed for pixels in
*pixellist
       *pixelList: list of pixels

   blob_averages  replaces every set of connected pixels in pixellist by a
   single pixel corresponding with the average X and Y position of the
   set. All these pixels get their blue colour channel set to 255.
*/


void get_eyes(allEdges gEdges, listElt **gPixels, setOfEyes *lastEyes);
/* IN  gEdges   : coordinates of facial region and average face position
       *gPixels : list of pixels
       *lastEyes: X and Y coordinates of PREVIOUSLY detected eyes
   OUT *gPixels : list of pixels
       *lastEyes: the X and Y coordinates of the detected eyes

   get_eyes uses coordinates in gEdges and the previously found eye
coordinates
   to calculate a score for each pixel in *gPixels indicating its
probability
   of being an eye. The result of these calculations is returned through
   *lastEyes AND as the first two elements of *gPixels.
*/


void calc_ideal_eyes(allEdges dEdges, setOfEyes *idealEyes);
/* IN  dEdges    : facial region coordinates
   OUT *idealEyes: the resulting ideal coordinates of the eyes

   calc_ideal_eyes uses only the facial region coordinates to calculate the
   position of whereabouts the eyes should be therein. These are halfway the
   facial region vertically, and at 30% and 70% horizontally (This procedure
   is only used for experiments on how to calculate the real eyes).
*/
```

8.2.5 Draws.h

```
/*               ***** draws.h *****

   This unit contains the framesize reduction routines and the routines
   called for displaying various intermediate and final results of eye
   tracking processes.
*/


frame reduce (frame pFrame);
/* IN  pFrame  : frame to be reduced
   OUT return  : reduced frame

   reduce reduces the size of pFrame to 1/4th its original size, taking the
   average colour value of every 4 pixel square. Thus the amount of noise in
   the resulting frame is reduced (averaged out) and calculation time of the
   following procedures is reduced.
*/


frame reduceDouble (frame pFrame);
/* IN  pFrame  : frame to be reduced
   OUT return  : reduced frame

   reduceDouble is almost identical to reduce, except that it reduces pFrame
   to 1/16th its size (effect is identical to applying reduce twice, hence
   the name).
*/


void draw_box (frame dFrame, allEdges dEdges);
/* IN  dFrame: frame in which the edges are to be drawn
       dEdges: the structure containing the edge data
   OUT dFrame: frame, with the edges drawn

   draw_box draws edges given in dEdges in dFrame
*/


void draw_V_analysis(frame dFrame, allEdges dEdges, cData Analysis);
/* IN  dFrame  : frame in which analysis is to be drawn
       dEdges  : contains coordinates where analysis is to be drawn
       Analysis: contains analysis data
   OUT dFrame  : frame with analysis data drawn

   draw_V_analysis draws the vertical analysis data given in Analysis at
   coordinates given in dEdges into frame dFrame
*/

void draw_H_analysis(frame dFrame, allEdges dEdges, cData analysis);
/* IN  dFrame  : frame in which analysis is to be drawn
       dEdges  : contains coordinates where analysis is to be drawn
       Analysis: contains analysis data
   OUT dFrame  : frame with analysis data drawn

   draw_H_analysis draws the horizontal analysis data given in Analysis at
   coordinates given in dEdges into frame dFrame
*/
```

```
void draw_cross(frame cFrame, int X, int Y,
                channel R, channel G, channel B);
/* IN  cFrame  : the frame in which cros is to be drawn
       X,Y     : standardised coordinates of position where cross is to be
                 drawn
       R,G,B   : RGB colour values of cross to be drawn
   OUT cframe  : frame, with crosses drawn

   draw_cross draws a cross with colour (R,G,B) at position (X,Y) in cframe.
*/
```

8.2.6 Tracker.h

```
/*     ******* tracker.h *********

  this unit contains the basic routines needed for performing eye tracking.

  to track eyes, simply define the variables needed by each procedure used,
  then call the procedures.

   About the procedure descriptions:
   IN descriptions describe the REQUIRED contents of the variables BEFORE
     the START of each procedure; Unless explicitly described otherwise,
     these contents MUST be valid at this point
   OUT descriptions describe (if changed) the contents of the variables
     AFTER procedure TERMINATION. These contents WILL be valid at this point
     (eventual 'return' variables describe the contents of the function
return)

  */


void initialise_tracking (windowInfo *capWin,
                      frame *calFrame,
                      int *ftr_threshold,
                      setOfEyes *eyes,
                      int processReduced);
/* IN  processReduced: a boolean indicating whether or not frame reduction
is
                       applied
   OUT *capWin       : a record containing the info concerning a now
                       initialized window
       *calFrame     : a calibration frame
       *redCalFrame  : the reduced size version of the same calibration
frame
       *ftr_threshold: an initial value for the colour filtering threshold.

   initialise_tracking initialises all variables that need initialisation
   before the eye tracking can be started. It also takes care of the initial
   calibration required.

   call this procedure at least once before starting eye tracking, and
   if desired at points where recalibration might be necessary.
*/


frame track_frame(frame calFrame,
                int *ftr_threshold,
                setOfEyes *prevEyes,
                allEdges *resultEdges,
                int processReduced);
/* IN  calFrame      : calibration frame
       *ftr_threshold: colour filtering threshold to be used for THIS frame
       *prevEyes     : previously detected eye coordinates
       processReduced: a boolean indicating whether or not frame reduction
is
                       applied (MUST have same value as was passed to
                     initialise_tracking)
   OUT *ftr_threshold: colour filtering threshold to be used for NEXT frame
       *prevEyes     : newly detected eye coordinates
       *resultEdges  : coordinates of detected face edges
       **candidates  : list of all candidates detected, first two are eyes

   track_frame performs eye tracking on a single frame, passing on the
   detected values of face edges and eye coordinates for arbitrary use.
   It also adjusts the colour filtering threshold if necessary for the next
   call of track_frame.
```

```
   call this procedure once for each frame thad needs to be tracked.
   */

void draw_results(frame drawFrame,
                  allEdges resultEdges, setOfEyes theEyes);
/* IN  drawFrame  : the frame in which the tracking results are to be drawn
       resultEdges: the head detection results
       *candidates: the eye detection results
   OUT drawFrame  : the frame, with the results drawn

   draw_results draws the given tracking results in the given frame.

   call this whenever visualising tracking results is desired. If so,
usually
   per-frame
   */
```

8.2.7 Gltest.h

```
/*      ****** gltest.h ******

   this unit contains a set of routines using openGL to draw a simple
   scenery in a window, from a viewpoint that is derived from the tracked
   eye-coordinates. Only init_graphics needs to be called from the main
   program, it will execute the other routines and the eye-tracker when
needed.
*/

void draw_model(void);
/* draws a simple model (some cubes on a plane) in a virtual space */

void calculate_all(void);
/* calculates the position of the screen with respect to the eye coordinates
   by calling the eye tracker
*/

void graphics_display(void);
/* displays the scenery */

void init_graphics(frame calFrame,
                int *ftr_threshold,
                setOfEyes theEyes,
                int processReduced);
/* main opengl procedure. initialises opengl. sets calculation and display
   routines, then enters openGL main loop
*/
```

8.3 Complete source code

8.3.1 Capture.c

```
/********** capture.c ************

  This contains routines related to getting a frame from the O2's default
  video-in port, as set by videopanel(1)

   It also contains some window manipulation routines
*/

#include <stdlib.h>
#include <stdio.h>
#include <gl/gl.h>
#include <dmedia/vl.h>
#include <time.h>

#include "alldefs.h"

static VLServer svr;
static VLPath path;
static VLNode src, drn;
static VLControlValue val;
static VLBuffer buffer;
static VLInfoPtr info;
static char *dataPtr;
static int c;
static int cap_xsize;
static int cap_ysize;


int capture_image_xsize(void)
/* Return the horizontal size of the captured frame in pixels */
{
  return(cap_xsize);
}

int capture_image_ysize(void)
/* Return the vertical size of the captured frame in pixels */
{
  return(cap_ysize);
}

void init_capture(void)
/* Initialisation stuff for the video capture.  Uses whatever the default
   input is, as set by videopanel(1) */
{
  foreground();

  /* Connect to the daemon */
  if (!(svr = vlOpenVideo(""))) {
    printf("Error opening video.\n");
    exit(1);
  }

  /* Set up a drain node in memory */
  drn = vlGetNode(svr, VL_DRN, VL_MEM, VL_ANY);

    /* Set up a source node on any video source  */
  src = vlGetNode(svr, VL_SRC, VL_VIDEO, VL_ANY);

  /* Create a path using the first device that will support it */
  path = vlCreatePath(svr, VL_ANY, src, drn);

  /* Set up the hardware for and define the usage of the path */
```

```
  if ((vlSetupPaths(svr, (VLPathList)&path, 1, VL_SHARE, VL_SHARE)) < 0) {
    printf("Error setting up path\n");
    exit(1);
  }

    /* Set the packing to RGB */
  val.intVal = VL_PACKING_RGB_8;
  vlSetControl(svr, path, drn, VL_PACKING, &val);

  /* try and set the video zoom */
  val.fractVal.numerator = VIDEO_CAP_ZOOM_NUMERATOR;
  val.fractVal.denominator = VIDEO_CAP_ZOOM_DENOMINATOR;
/*   val.xyVal.x = VIDEO_CAP_ZOOM; */
/*   val.xyVal.y = VIDEO_CAP_ZOOM; */
  vlSetControl(svr,path,drn,VL_ZOOM,  &val);

  /* Get the video size */
  vlGetControl(svr, path, drn, VL_SIZE, &val);
  cap_xsize = val.xyVal.x;
  cap_ysize = val.xyVal.y;
  /* Create and register a buffer for 1 frame */
  buffer = vlCreateBuffer(svr, path, drn, 1);
  if (buffer == NULL) {
    printf("Error creating buffer\n");
    exit(1);
  }
  vlRegisterBuffer(svr, path, drn, buffer);

  /* Begin the data transfer */
  if (vlBeginTransfer(svr, path, 0, NULL)) {
    printf("error beginning transfer\n");
    exit(1);
  }

}

frame capture_frame(void)
/* This captures a frame, and returns a pointer to its location in memory.
   Uses 0xAABBGGRR format in memory */
{
  frame newFrame;
    /* Wait for a frame */
  do {
    /*    info = vlGetNextValid(svr, buffer); */
    info = vlGetLatestValid(svr,buffer);
  } while (!info);
  /* Get a pointer to the frame */
  dataPtr = vlGetActiveRegion(svr, buffer, info);
  newFrame.handle = dataPtr;
  newFrame.xsize  = capture_image_xsize();
  newFrame.ysize  = capture_image_ysize();
  return(newFrame);
}



void finish_with_frame(void)
/* Finished with frame, unlock the buffer */
{
  vlPutFree(svr, buffer);
}



void capture_end(void)
/* Tidy-up stuff to do before exiting program */
{
  /* End the data transfer */
```

```
  vlEndTransfer(svr, path);

    /* Cleanup before exiting */
  vlDeregisterBuffer(svr, path, drn, buffer);
  vlDestroyBuffer(svr, buffer);
  vlDestroyPath(svr, path);
  vlCloseVideo(svr);
}


/* window manipulation routines */

windowInfo init_window(char *title)
// Initialise a window, and return its info
{
  windowInfo win;

  /* Set up and open a GL window to display the data */
  foreground();
  prefsize(VIDEO_CAP_SIZE_X,VIDEO_CAP_SIZE_Y);
  win.handle = winopen(title);
  win.xsize = VIDEO_CAP_SIZE_X;
  win.ysize = VIDEO_CAP_SIZE_Y;
  RGBmode();
  pixmode(PM_TTOB, 1);
  gconfig();
  return(win);
}

void display(frame dispFrame, windowInfo win)
// Copy RGBA data into a specified window
{
  /* Write the data to the screen */
  winset(win.handle);
  lrectwrite(0,0, dispFrame.xsize-1, dispFrame.ysize-1,
            (ulong *)dispFrame.handle);
}

void close_window(windowInfo win)
// Close window
{
  winclose(win.handle);
}
```

8.3.2 Util.c

```
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <math.h>
#include <pthread.h>
#include <stdio.h>

#include "alldefs.h"

/***** a set of timing routines ******/

double current_time(void)
// returns the current time in seconds (resolution of microseconds)
{
  struct timespec ts;
  clock_gettime(CLOCK_REALTIME, &ts);

  return ( ts.tv_sec+(0.000000001*ts.tv_nsec) );
}


HR_TIMER hrGetTime(void)
// returns the current time in milliseconds (resolution of microseconds)
{
  struct timespec ts;
  clock_gettime(CLOCK_REALTIME, &ts);

  return ( 1000.0*ts.tv_sec+(0.000001*ts.tv_nsec) );
}


void hrWait(HR_TIMER time)
/* This function waits for a specified time (in milliseconds)
   Use with caution. This is very demanding on cpu time */
{
       HR_TIMER initTime=hrGetTime();
       while ((hrGetTime()-initTime)<time)
       {
       }
}



/* A set of coordinate transformation routines used to be able to:
   - apply calculations in a certain frame to a different-sized frame;
   - quickly retrieve a pixel's coordinates or retrieve a pointer to a pixel
     on the given coordinates
 */

channel *pos_ptr(frame pFrame, int X, int Y)
// returns a pointer to position (X,Y) in frame pFrame
{
  int appliedX=X*(pFrame.xsize/STD_X_SIZE);
  int appliedY=Y*(pFrame.ysize/STD_Y_SIZE);
  return pFrame.handle+4*appliedX+4*(appliedY*pFrame.xsize);
}

int std_X_pos(frame pFrame, channel *pPtr)
// returns standardised X coordinate of pPtr in pFrame
{
  return (((pPtr-(pFrame.handle))%(pFrame.xsize*4))/4) /
    (pFrame.xsize/STD_X_SIZE);
}

int std_Y_pos(frame pFrame, channel *pPtr)
// returns standardised Y coordinate of pPtr in pFrame
```

```
{
  return ((pPtr-(pFrame.handle))/(pFrame.xsize*4)) /
    (pFrame.ysize/STD_Y_SIZE);
}

int standardise_X(frame pFrame, int X)
// transforms pframe's X-coordinate to standard X-coordinate
{
  return X/(pFrame.xsize/STD_X_SIZE);
}

int standardise_Y(frame pFrame, int Y)
// transforms pframe's X-coordinate to standard Y-coordinate
{
  return Y/(pFrame.ysize/STD_Y_SIZE);
}

int apply_X(frame pFrame, int X)
/* transforms standard X-coordinates to coordinates that apply to pFrame
   (so it is the inverse of standardise-X) */
{
  return X*(pFrame.xsize/STD_X_SIZE);
}

int apply_Y(frame pFrame, int Y)
/* transforms standard Y-coordinates to coordinates that apply to pFrame
   (so it is the inverse of standardise-Y) */
{
  return Y*(pFrame.ysize/STD_Y_SIZE);
}

void colour_pixel(channel *pPtr, char R,char G,char B)
// paints pixel pointed to by pPtr in colours R,G,B
{
  *(pPtr+3) = R;
  *(pPtr+2) = G;
  *(pPtr+1) = B;
}

/* here follows a set of procedures that perform standard list-operations to
   my tracking-specific linked list
*/

listElt **createElt(channel *newChannel, int newCX, int newCY, int newScore)
/* creates a list element sets its next field to NIL and returns a pointed
to
   it
*/
{
  listElt *newElt = malloc(LIST_ELT_SIZE);
  newElt->cPtr    = newChannel;
  newElt->X       = newCX;
  newElt->Y       = newCY;
  newElt->pScore  = newScore;
  newElt->next    = NIL;
  return &newElt;
}



void insertInList(listElt **list, listElt *insElt)
// inserts insElt at the head of *list
{
  insElt->next = *list;
  *list = insElt;
}
```

```
void insertSorted(listElt **list, listElt *insElt)
/* inserts insElt in list, sorted by value of insElt->score
 */
{
  listElt *posPtr = *list;
  if ((posPtr == NIL) || (insElt->pScore>(*list)->pScore))
    {
      insElt->next = *list;
      *list = insElt;
    }
  else
    {
      while ((posPtr->next  != NIL) &&
            (insElt->pScore<posPtr->next->pScore))
       posPtr = posPtr->next;
      insElt->next = posPtr->next;
      posPtr->next = insElt;
    }
}



void removeFromList(listElt **list, channel *rChannel)
/* removes first element (if one exists) that satisfies
   (listElt->cPtr)==rchannel
   from list */
{
  listElt *Ptr1= *list, *Ptr2;
  if ((*list)->cPtr == rChannel)
    {
      Ptr1 = (*list)->next;
      free(*list);
      *list = Ptr1;
    }
  else
    {
      Ptr2 = (*list)->next;
      while ((Ptr2!=NIL)&&(Ptr2->cPtr != rChannel))
       {
         Ptr1 = Ptr2;
         Ptr2 = Ptr2->next;
       }
      Ptr1->next = Ptr2->next;
      free(Ptr2);
    }
}

int is_member(listElt *list, channel *mChannel)
/* checks if the element listElt which satisfies
   (listElt->cPtr==mChannel)
   is in list */
{
  listElt *checkPtr = list;
  while ((checkPtr !=NIL) && (checkPtr->cPtr != mChannel))
    checkPtr = checkPtr->next;
    //now either checkPtr==NIL or points to element searched for
  return (checkPtr != NIL);
}



void swap(int *a, int *b)
//swaps values of variables a and b
{
  int swap = *a;
  *a = *b;
  *b = swap;
```

```
}



int find_blob(frame bFrame, listElt *list, channel *bPtr, listElt
**blobList)
/* finds all pixels belonging to blob which pixel *bptr is part of.
   This is actually a recursive floodfill-algorithm, applied to my eye-
tracking
   system.
*/
{
  int newBlob = 0;
  listElt **elt;
  if ( (*(bPtr+1)!=255) && (is_member(list,bPtr)) )
    {
      *(bPtr+1) = 255;
      elt = createElt(bPtr,std_X_pos(bFrame,bPtr),
std_Y_pos(bFrame,bPtr),0);
      insertInList(blobList, *elt);
      if (bPtr>bFrame.handle)
       find_blob(bFrame, list, bPtr-4, blobList);
      if (bPtr<bFrame.handle+(bFrame.xsize*bFrame.ysize*4))
       find_blob(bFrame, list, bPtr+4, blobList);
      if (bPtr>bFrame.handle+bFrame.xsize*4)
       find_blob(bFrame, list, bPtr-bFrame.xsize*4, blobList);
      if (bPtr<bFrame.handle+bFrame.xsize*(bFrame.ysize-1)*4)
       find_blob(bFrame, list, bPtr+bFrame.xsize*4,blobList);
      newBlob = 1;
    }
  return newBlob;
}
```

8.3.3 Vision.c

```
/************ vision.c ***********/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "alldefs.h"
#include "capture.h"
#include "util.h"



frame calibrate(windowInfo capWin, int calFramesUsed)
     /* displays frames in given window for about 2 seconds;
       returns 'average value' of last 2 frames as reference frame */
{
  int i;
  char c;
  frame calFrame,calFrame2, defCalFrame;
  channel *cf1Ptr,*cf2Ptr,*defPtr;
  printf("press enter to start calibration\n");
  getchar();
  calFrame2.handle = malloc(capWin.xsize*capWin.ysize*4);
  calFrame2.xsize = capWin.xsize;
  calFrame2.ysize = capWin.ysize;
  defCalFrame.handle = malloc(capWin.xsize*capWin.ysize*4);
  defCalFrame.xsize = capWin.xsize;
  defCalFrame.ysize = capWin.ysize;
  while (c != 'c')
    {
      for (i=0; i<calFramesUsed;i++)
       {
         calFrame = capture_frame();
         display(calFrame, capWin);
         finish_with_frame();
       }
      calFrame = capture_frame();
      calFrame2 = capture_frame();
      cf1Ptr=calFrame.handle;
      cf2Ptr=calFrame2.handle;
      defPtr=defCalFrame.handle;
      for (i=0;
i<capWin.xsize*capWin.ysize*4;i++,cf1Ptr++,cf2Ptr++,defPtr++)
       *defPtr = ((*cf1Ptr)+(*cf2Ptr))/2;
      display(defCalFrame,capWin);
      finish_with_frame();
      printf("press Enter to calibrate again or c to continue...\n");
      c = getchar();
    }
  return defCalFrame;
}



allEdges subtract_image(frame calibrationFrame,
                    frame procFrame,
                    int cThreshold,
                    int xPixelSkip)
     /* Processes given frame:
       - Performs image subtraction
       - calculates average position of user */
{
  int x,y, channelNo,colorCheck;

  long Xtotal=0, Ytotal=0, NoOfDots=0,
```

```
    AvgX=0, AvgY=0;

  allEdges edges;

  channel *cChannelPtr; // traces calibration frame per channel
  channel *pChannelPtr; // traces process frame
  channel *rChannelPtr; // traces result frame

  frame resultFrame;

  resultFrame.handle = calloc(procFrame.xsize*procFrame.ysize*4,1);
  resultFrame.xsize=procFrame.xsize;
  resultFrame.ysize=procFrame.ysize;

  cChannelPtr = calibrationFrame.handle;
  pChannelPtr = procFrame.handle;
  rChannelPtr = resultFrame.handle;
  // pixel-by-pixel image subtraction
  for(y=0;(y<(procFrame.ysize)); y++)
    {
      for (x=0;(x<(procFrame.xsize)); x+=xPixelSkip+1)
       {
         colorCheck = 0;
         for (channelNo=0;channelNo<4;channelNo++)// 4 colour channels
           {
             if (ABS(*cChannelPtr-*pChannelPtr)<=cThreshold)
              colorCheck++;
             cChannelPtr++;
             pChannelPtr++;
           }
         rChannelPtr+=4;
         if (colorCheck < 4) // not equal enough --> part of user image
           {
             memcpy(rChannelPtr-4,pChannelPtr-4,4*(xPixelSkip+1));
             Xtotal +=x; // for calculating average position of non-black
pxs.
             Ytotal +=y;
             NoOfDots++;
             }
         pChannelPtr += 4*xPixelSkip;
         cChannelPtr += 4*xPixelSkip;
         rChannelPtr += 4*xPixelSkip;
       }
    }

  // average position of non-black pixels
  if (NoOfDots !=0)
    {
      edges.AvgX = standardise_X(procFrame,(Xtotal/NoOfDots));
      edges.AvgY = standardise_Y(procFrame,(Ytotal/NoOfDots));
    }
  else edges.AvgX=edges.AvgY=0;
  memcpy(procFrame.handle, resultFrame.handle,
        procFrame.xsize*procFrame.ysize*4);
  free (resultFrame.handle);
  return edges;
}



void detect_edges(frame resultFrame, allEdges *edges, int scanWidth)
     /* performs edge-detection on subtracted image: scans from outer edges
        to first non-black pixels encountered */
{
  int AvgX    = apply_X(resultFrame,edges->AvgX),
      AvgY    = apply_Y(resultFrame,edges->AvgY),
      leftX   = 0,
      rightX  = resultFrame.xsize-1,
```

```
    topY    = 0,
    bottomY,
    std_xsize=standardise_X(resultFrame, resultFrame.xsize);
channel *calcPtr1, *calcPtr2, *calcPtr3;
// left vert. line: start from left edge of frame
calcPtr1 = pos_ptr(resultFrame, 0,
                (edges->AvgY)+Y_CORRECTION-scanWidth);
calcPtr2 = pos_ptr(resultFrame, 0,
                (edges->AvgY)+Y_CORRECTION);
calcPtr3 = pos_ptr(resultFrame, 0,
                (edges->AvgY)+Y_CORRECTION+scanWidth);
while((leftX<resultFrame.xsize)           &&
     (((*(calcPtr1+1)==0) &&
       (*(calcPtr1+2)==0) &&
       (*(calcPtr1+3)==0)   )||  // while ONE of ptrs black
      ((*(calcPtr2+1)==0) &&
       (*(calcPtr2+2)==0) &&
       (*(calcPtr2+3)==0)   )||
      ((*(calcPtr3+1)==0) &&
       (*(calcPtr3+2)==0) &&
       (*(calcPtr3+3)==0))  )  ) // move ALL right...
  {
    *(calcPtr1+2)=*(calcPtr2+2)=*(calcPtr3+2)=255;
    calcPtr1 += 4;
    calcPtr2 += 4;
    calcPtr3 += 4;
    leftX++;
  }

// right vert. line: start from right edge of frame
calcPtr1 = pos_ptr(resultFrame, std_xsize-1,
                (edges->AvgY)+Y_CORRECTION-scanWidth);
calcPtr2 = pos_ptr(resultFrame, std_xsize-1,
                (edges->AvgY)+Y_CORRECTION);
calcPtr3 = pos_ptr(resultFrame, std_xsize-1,
                (edges->AvgY)+Y_CORRECTION+scanWidth);
while( (rightX>0) &&
     (((*(calcPtr1+1)==0) &&
       (*(calcPtr1+2)==0) &&
       (*(calcPtr1+3)==0)   )||  // while ONE of ptrs black
      ((*(calcPtr2+1)==0) &&
       (*(calcPtr2+2)==0) &&
       (*(calcPtr2+3)==0)   )||
      ((*(calcPtr3+1)==0) &&
       (*(calcPtr3+2)==0) &&
       (*(calcPtr3+3)==0))  )  ) // move ALL left...
  {
    *(calcPtr1+3)=*(calcPtr2+3)=*(calcPtr3+3)=255;
    calcPtr1 -= 4;
    calcPtr2 -= 4;
    calcPtr3 -= 4;
    rightX--;
  }

// top hor. line: start from top of frame in the middle of detected
// left & right, original average is adjusted
AvgX = standardise_X(resultFrame, (leftX+rightX)/2);
calcPtr1 = pos_ptr(resultFrame,AvgX-scanWidth,0);
calcPtr2 = pos_ptr(resultFrame,AvgX,0);
calcPtr3 = pos_ptr(resultFrame,AvgX+scanWidth,0);
while((topY<resultFrame.ysize-1)   &&
     (((*(calcPtr1+1)==0) &&
       (*(calcPtr1+2)==0) &&
       (*(calcPtr1+3)==0))  ||       // while ONE of ptrs black
      ((*(calcPtr2+1)==0) &&
       (*(calcPtr2+2)==0) &&
       (*(calcPtr2+3)==0))  ||
      ((*(calcPtr2+1)==0) &&
```

```
        (*(calcPtr2+2)==0) &&
        (*(calcPtr2+3)==0)   ))     )
    {
      *(calcPtr1+1)=*(calcPtr2+1)=*(calcPtr3+1)=255;
      calcPtr1 += resultFrame.xsize*4;
      calcPtr2 += resultFrame.xsize*4;
      calcPtr3 += resultFrame.xsize*4;
      topY++;                          // move ALL down...
    }
 /* now topY should contain top edge of blob
       (= y position of top hor. crosshair */
  bottomY = MIN(topY+((rightX-leftX)*1.25),resultFrame.ysize-1);
  edges->leftX  = standardise_X(resultFrame, leftX);
  edges->rightX = standardise_X(resultFrame, rightX);
  edges->topY   = standardise_Y(resultFrame, topY);
  edges->bottomY= standardise_Y(resultFrame, bottomY);
  edges->AvgX   = AvgX;  //already standardised
  edges->AvgY   = (edges->topY+edges->bottomY)/2;
  // (edges.avgX is already defined)
}



listElt *inv_red_filter(frame calcFrame, allEdges fEdges,
                    int *threshold)
     /* sets all but red colour channels in frame to zero and also red
values
       that are higher than threshold. Then remaining values are inverted to
       bring UP DARKER shades, and all remaining pixels are returned */
{
  int i=0, j=0, x,y,noOfElts=0,
    boxSizeX = apply_X(calcFrame,(fEdges.rightX-fEdges.leftX)),
    boxSizeY = apply_Y(calcFrame,(fEdges.bottomY-fEdges.topY)),
    realLeftX= apply_X(calcFrame,fEdges.leftX),
    realTopY = apply_Y(calcFrame,fEdges.topY);
  listElt *nonBlacks = NIL, **newElt = NIL;
  channel *cChannelPtr = pos_ptr(calcFrame, fEdges.leftX, fEdges.topY);
  for (y=0;y<boxSizeY;y++, cChannelPtr+=(calcFrame.xsize-boxSizeX)*4)
    {
      for (x=0;x<boxSizeX;x++,cChannelPtr +=4)
       {
         *cChannelPtr= *(cChannelPtr+1)= *(cChannelPtr+2)=0;
         //ch 3 (r) stays
         if ((*(cChannelPtr+3)<*threshold) && (*(cChannelPtr+3)>0))
           {
             *(cChannelPtr+3)= 255 -( *(cChannelPtr+3)); // inversion
             newElt = createElt(cChannelPtr, //create listelement for pixel
                           standardise_X(calcFrame,realLeftX+x),
                           standardise_Y(calcFrame,realTopY+y) ,
                           0);
             insertInList(&nonBlacks, *newElt);
             noOfElts++;
           }
         else *(cChannelPtr+3) = 0;
       }
    }
  if (noOfElts > apply_X(calcFrame,OUTPUT_LIST_MAX)) (*threshold)--;
  else if (noOfElts <apply_X(calcFrame,OUTPUT_LIST_MIN)) (*threshold)++;
  return(nonBlacks);
}



void VCAnalysis(frame pFrame, allEdges *edges, cData *analysis)
     /* performs extensive colour analysis (vertical) on given frame */
{
  int x,y,framesize=pFrame.xsize*pFrame.ysize*4,
      firstY=apply_Y(pFrame,(*edges).topY),
```

```
      lastY =apply_Y(pFrame,(*edges).bottomY),
      firstX=apply_X(pFrame,(*edges).leftX),
      lastX =apply_X(pFrame,(*edges).rightX),

      Pixels[240],

      TotRedValue[240],
      TotGreenValue[240],
      TotBlueValue[240],

      AvgWhiteValue[240],
      CorrWhiteValue;

  channel *pChannelPtr=pos_ptr(pFrame,edges->leftX,edges->topY);

  memset(Pixels,0,960);
  memset(TotRedValue,0,960);
  memset(TotGreenValue,0,960);
  memset(TotBlueValue,0,960);
  memset(analysis->red,0,1280);
  memset(analysis->green,0,1280);
  memset(analysis->blue,0,1280);

  for (y=firstY;y<lastY;y++)
    {
      for(x=firstX;x<lastX;x++)
       {
         if (*(pChannelPtr+3) != 0 ||
             *(pChannelPtr+2) != 0 ||
             *(pChannelPtr+1) != 0 ||
             *(pChannelPtr)   != 0)
           {
             Pixels[y]++;
             TotRedValue[y]  += *(pChannelPtr+3);
             TotGreenValue[y]+= *(pChannelPtr+2);
             TotBlueValue[y] += *(pChannelPtr+1);
           }
         pChannelPtr+=4;
       }
      if(Pixels[y]!=0)
       {
         analysis->red[y]= (TotRedValue[y]/Pixels[y]);
         analysis->green[y] = (TotGreenValue[y]/Pixels[y]);
         analysis->blue[y]  = (TotBlueValue[y]/Pixels[y]);
       }
      pChannelPtr = pos_ptr(pFrame, edges->leftX,y);
    } //y direction loop
}



void HCAnalysis(frame pFrame, allEdges *edges, cData *analysis)
     /* performs extensive colour analysis (horizontal) on given frame */
{
  int x,y,framesize=pFrame.xsize*pFrame.ysize*4,
      firstY=apply_Y(pFrame,(*edges).topY),
      lastY =apply_Y(pFrame,(*edges).bottomY),
      firstX=apply_X(pFrame,(*edges).leftX),
      lastX =apply_X(pFrame,(*edges).rightX),

      Pixels[320],

      TotRedValue[320], TotGreenValue[320], TotBlueValue[320];

  channel *pChannelPtr=pos_ptr(pFrame,edges->leftX,edges->topY);

  memset(Pixels,0,1280);
```

```
  memset(TotRedValue,0,1280);
  memset(TotGreenValue,0,1280);
  memset(TotBlueValue,0,1280);

  memset((*analysis).red,0,1280);
  memset((*analysis).green,0,1280);
  memset((*analysis).blue,0,1280);

  for (x=firstX;x<lastX;x++)
    {
      for(y=firstY;y<lastY;y++)
       {
         if (*(pChannelPtr+3) != 0 ||
             *(pChannelPtr+2) != 0 ||
             *(pChannelPtr+1) != 0 ||
             *(pChannelPtr)   != 0)
           {
             Pixels[x]++;
             TotRedValue[x]  += *(pChannelPtr+3);
             TotGreenValue[x]+= *(pChannelPtr+2);
             TotBlueValue[x] += *(pChannelPtr+1);
           }
         pChannelPtr+=pFrame.xsize*4;
       }
      if(Pixels[x]!=0)
       {
         (*analysis).red[x]   = (TotRedValue[x]/Pixels[x]);
         (*analysis).green[x] = (TotGreenValue[x]/Pixels[x]);
         (*analysis).blue[x]  = (TotBlueValue[x]/Pixels[x]);
       }
      pChannelPtr = pos_ptr(pFrame, x, edges->topY);
    } //x direction loop
}



void centrality_score(frame tFrame, listElt *maxPixels)
     /* performs neighbourhood analysis on every pixel in maxpixels */
{
  listElt *pixelPtr = maxPixels;
  channel *checkPtr;
  int x,y, totV;

  while ((pixelPtr != NIL))
    {
      if (((pixelPtr->cPtr-tFrame.handle)>(tFrame.xsize*8+8)) &&
         ((pixelPtr->cPtr-tFrame.handle)<(tFrame.xsize*(tFrame.ysize-1)*8)-
8))
       {
         checkPtr = pixelPtr->cPtr-tFrame.xsize*8-8; // 2 pixels up & 2 left
         for (y=0; y<5;y++)    // check 5x5 pixel block around pixelptr
           {
             for (x=0; x<5;x++)
              {
                // totV += *(checkPtr+3);
                *((pixelPtr->cPtr)+2) += (*(checkPtr+3))/25;
                checkPtr +=4;
              }
             checkPtr += (tFrame.xsize-5)*4;
           }
         // *((pixelPtr->cPtr)+3) ) = 0;
       }
      pixelPtr = pixelPtr->next;
    }
}
```

```
void remove_least_centrals(frame rFrame, listElt **centrals)
     /* uses results of neighbourhood analysis to remove non-centrally
located
       pixels from *centrals */
{
  listElt *pixelPtr = *centrals;

  while (pixelPtr != NIL)
    // compare 3x3 pixel square around pixel
    {
       if((*((pixelPtr->cPtr)+2)  < *((pixelPtr->cPtr)-2-rFrame.xsize*4)) ||
         (*((pixelPtr->cPtr)+2)  < *((pixelPtr->cPtr)+2-rFrame.xsize*4)) ||
         (*((pixelPtr->cPtr)+2)  < *((pixelPtr->cPtr)+6-rFrame.xsize*4)) ||
         (*((pixelPtr->cPtr)+2)  < *((pixelPtr->cPtr)-2 )) ||
         (*((pixelPtr->cPtr)+2)  < *((pixelPtr->cPtr)+6 )) ||
         (*((pixelPtr->cPtr)+2)  < *((pixelPtr->cPtr)-2+rFrame.xsize*4)) ||
         (*((pixelPtr->cPtr)+2)  < *((pixelPtr->cPtr)+2+rFrame.xsize*4)) ||
         (*((pixelPtr->cPtr)+2)  < *((pixelPtr->cPtr)+6+rFrame.xsize*4))   )
        {
          removeFromList(centrals,pixelPtr->cPtr);
          *(pixelPtr->cPtr+3) = 0;
        }
       pixelPtr = pixelPtr->next;
    }
}



void blob_averages(frame bFrame, listElt **pixelList)
     /* calculates average position of each blob formed by pixels
       in *pixellist */
{
  listElt **newList=malloc(4),
          *pixelPtr =*pixelList,
          *blobs,
          *removePtr,
          **newElt;
  int totX, totY, blobSize,check;

  *newList =NIL;
  while (pixelPtr != NIL)
    {
      blobs = NIL;
      if (find_blob(bFrame, pixelPtr, pixelPtr->cPtr, &blobs) == 1)
             {
        totX=0;
        totY=0;
        blobSize=0;
        while (blobs != NIL)
          {
            totX += blobs->X;
            totY += blobs->Y;
            blobSize++;
            removePtr = blobs;
            blobs = blobs->next;
            free(removePtr);
          }
            totX = totX/blobSize;
            totY = totY/blobSize; // these are STANDARD X & Y values
            newElt = createElt(pos_ptr(bFrame,totX,totY),
                          totX,
                          totY,
                          0);
            insertInList(newList, *newElt);
      }
      removePtr = pixelPtr;
      pixelPtr  = pixelPtr->next;
      free(removePtr);
```

```
    }
  *pixelList = *newList;
}



void get_eyes(allEdges gEdges, listElt **gPixels, setOfEyes *lastEyes)
     /* calculates a score for each pixel in list indicating its probability
       of being a left or right eye */
{
  listElt **newList = malloc(4),
          *gPtr     = *gPixels,
          *removePtr, **newElt;
  int xScore    = 0,
      yScore    = 0,
      finalScore= 0,
      boxSizeX = gEdges.rightX-gEdges.leftX;
  *newList=NIL;
  // first loop: calculate score, highest one should  be left eye (onscreen)
  while (gPtr!= NIL)
    {
      /* calculate scores, insert in list sorted by 'lefteye'  score */
      xScore     =    ABS(gEdges.leftX+0.3*boxSizeX-gPtr->X);
      yScore     = 2*(ABS(gEdges.AvgY-gPtr->Y));
      finalScore = 100-yScore-xScore
                   -3*ABS(gPtr->X-lastEyes->leftX)
                  -2*ABS(gPtr->Y-lastEyes->rightX);
      newElt = createElt(gPtr->cPtr, gPtr->X, gPtr->Y, finalScore);
      insertSorted (newList,*newElt);
      removePtr = gPtr;
      gPtr = gPtr->next;
      free(removePtr);
    }
  if (*newList != NIL)
  // list is sorted -> first element has highest score. Now calculate new
score
  // of all other elements which is RELATIVE to first one
    *gPixels=NIL; //already should be, but for insurance...
  gPtr = *newList;
  if (gPtr != NIL)
    {
      lastEyes->leftX = gPtr->X; // store found X & Y values for next frame
      lastEyes->leftY = gPtr->Y;
      gPtr = gPtr->next; // skip 1st one (left eye)
      /* second loop: use coordinates of left eye to determine probability
        of all other pixels to be the right eye. left eye score is
discarded*/
      while (gPtr!= NIL)
       {
         /* calculate scores, insert in list sorted by 'righteye'  score */
         xScore     =    ABS(gEdges.leftX+0.7*boxSizeX-gPtr->X);
         yScore     = 3*(ABS(gEdges.AvgY+15-gPtr->Y));
         finalScore = 100-yScore-xScore
                      -3*ABS(gPtr->X-lastEyes->rightX)
                      -2*ABS(gPtr->Y-lastEyes->rightY)
           //x-coordinate difference constraint, high difference drops score
                      -(2* (0.4*boxSizeX)-(gPtr->X - (*newList)->X) )
         //y-coordinate constraint.
                      -(3*(ABS((*newList)->Y - gPtr->Y)));
         newElt = createElt(gPtr->cPtr, gPtr->X, gPtr->Y, finalScore);
         insertSorted (gPixels,*newElt);
         gPtr = gPtr->next;
       }
       // add left eye at head of new list, first two are now the eyes
      newElt = createElt((*newList)->cPtr,(*newList)->X,
                     (*newList)->Y,(*newList)->pScore);
      insertInList(gPixels,*newElt);
    }
```

```
  if ((*gPixels != NIL) && ((*gPixels)->next !=NIL))
    {
      lastEyes->rightX = (*gPixels)->next->X;
      lastEyes->rightY = (*gPixels)->next->Y;
      if (lastEyes->rightX < lastEyes->leftX)
       //if right eye is left of left eye, swap 'em
       {
         swap(&(lastEyes->leftX), &(lastEyes->rightX));
         swap(&(lastEyes->leftY), &(lastEyes->rightY));
       }
    }
}



calc_ideal_eyes(allEdges dEdges, setOfEyes *idealEyes)
     /* calculates the positions of ideal eyes (halfway face in Y
       direction, 30% and 70% in X direction) */
{
  int partBoxSizeX= 0.3*(dEdges.rightX-dEdges.leftX);

  idealEyes->leftX  = dEdges.leftX +partBoxSizeX;
  idealEyes->leftY  = dEdges.AvgY;
  idealEyes->rightX = dEdges.rightX-partBoxSizeX;
  idealEyes->rightY = dEdges.AvgY;
}
```

8.3.4 Draws.c

```
/************* draws.c ***************/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#include "alldefs.h"
#include "util.h"

frame reduce (frame pFrame)
     /* divides both xsize and ysize by 2, taking an average colour
       value for each new pixel */
{
  int x,y, ch;
  channel *evenChannel, *oddChannel, *newChannel;

  frame newFrame;
  newFrame.xsize  = pFrame.xsize / 2;
  newFrame.ysize  = pFrame.ysize / 2;
  newFrame.handle = malloc(newFrame.xsize*newFrame.ysize*4);

  evenChannel = pFrame.handle;
  oddChannel  = pFrame.handle+(pFrame.xsize*4);
  newChannel  = newFrame.handle;

  for (y=0;y<newFrame.ysize;y++)
    {
      for (x=0;x<newFrame.xsize;x++)
       {
         for (ch=0;ch<4;ch++)
           {
             *newChannel=(*evenChannel+*(evenChannel+4)+
                        *oddChannel +*(oddChannel +4) )/4;
             newChannel ++;
             evenChannel++;
             oddChannel ++;
           }
         evenChannel +=4; //second x-pixel already counted
         oddChannel  +=4;
       }
      evenChannel +=pFrame.xsize*4;
      oddChannel  +=pFrame.xsize*4; //skip the other line
    }
  return newFrame;
}



frame reduceDouble (frame pFrame)
     /* divides both xsize and ysize by 4, taking an average colour
       value for each new pixel */
{
  int x,y, ch;
  channel *Channel0, *Channel1, *Channel2, *Channel3, *newChannel;

  frame newFrame;
  newFrame.xsize  = pFrame.xsize / 4;
  newFrame.ysize  = pFrame.ysize / 4;
  newFrame.handle = malloc(newFrame.xsize*newFrame.ysize*4);

  Channel0 = pFrame.handle;
  Channel1 = pFrame.handle+(pFrame.xsize*4 );
  Channel2 = pFrame.handle+(pFrame.xsize*8 );
  Channel3 = pFrame.handle+(pFrame.xsize*12);
```

```
  newChannel  = newFrame.handle;

  for (y=0;y<newFrame.ysize;y++)
    {
      for (x=0;x<newFrame.xsize;x++)
       {
         for (ch=0;ch<4;ch++)
           {
             *newChannel=  // average of 4x4 pixel square channel
              (*Channel0+*(Channel0+4)+*(Channel0+8)+*(Channel0+12)+
               *Channel1+*(Channel1+4)+*(Channel1+8)+*(Channel1+12)+
               *Channel2+*(Channel2+4)+*(Channel2+8)+*(Channel2+12)+
               *Channel3+*(Channel3+4)+*(Channel3+8)+*(Channel3+12) )/16;
             newChannel ++;
             Channel0++;
             Channel1++;
             Channel2++;
             Channel3++;
           }
         Channel0 +=12; //2nd,3rd & 4th x-pixel already counted
         Channel1 +=12;
         Channel2 +=12;
         Channel3 +=12;
       }
      Channel0 +=pFrame.xsize*12;      //skip 4 channels * 3 lines
      Channel1 +=pFrame.xsize*12;
      Channel2 +=pFrame.xsize*12;
      Channel3 +=pFrame.xsize*12;
    }
  return newFrame;
}



void draw_box (frame dFrame, allEdges dEdges)
     // draws given edges in given frame
{
  channel *edge1Ptr, *edge2Ptr;
  int y,
      boxSizeX = apply_X(dFrame, dEdges.rightX-dEdges.leftX+1),
      boxSizeY = apply_Y(dFrame, dEdges.bottomY-dEdges.topY);
  // top hor. line
  edge1Ptr = pos_ptr(dFrame,dEdges.leftX,dEdges.topY);
  memset(edge1Ptr,255,boxSizeX*4);
  // bottom hor. line
  edge2Ptr = pos_ptr(dFrame,dEdges.leftX,dEdges.bottomY);
  memset(edge2Ptr,255,boxSizeX*4);
  // avg hor. line
  //edge1Ptr = pos_ptr(dFrame,dEdges.leftX,dEdges.AvgY);
  //memset(edge1Ptr,255,boxSizeX*4);

  // vertical lines
  edge1Ptr = pos_ptr(dFrame,dEdges.leftX,dEdges.topY);
  edge2Ptr = pos_ptr(dFrame,dEdges.rightX,dEdges.topY);
  for (y=0; y<boxSizeY; y++)
    {
      // left vert. line
      colour_pixel(edge1Ptr,255,255,255);
      edge1Ptr += (dFrame.xsize*4);
      // right vert. line
      colour_pixel(edge2Ptr,255,255,255);
      edge2Ptr += (dFrame.xsize*4);
    }
}



void draw_V_analysis(frame dFrame, allEdges dEdges, cData analysis)
```

```
     /* draws analysis results of vertical colour analysis in frame */
{
  int x,y, whiteValue, corrWhiteValue;
  channel *pixelPtr = pos_ptr(dFrame, 0, dEdges.topY);

      for(y=dEdges.topY;y<dEdges.bottomY; y++)
       {
         //red graph
         colour_pixel(pixelPtr+(analysis.red[y]*4),255,0,0);
         // green graph
         colour_pixel(pixelPtr+(analysis.green[y]*4),0,255,0);
         // blue graph
         colour_pixel(pixelPtr+(analysis.blue[y]*4),0,0,255);

         whiteValue= (analysis.red[y]+analysis.green[y]+analysis.blue[y])/3;
      // White Graph = avg.value *4 (channels)
         corrWhiteValue = whiteValue*4;
         colour_pixel(pixelPtr+corrWhiteValue,255,255,255);

         pixelPtr += dFrame.xsize*4;
         // gridlines
         for (x=0; x<=255;x+=50)
           {
             colour_pixel(pos_ptr(dFrame,x,y),100,100,100);
           }
       } //y-loop

}



void draw_H_analysis(frame dFrame, allEdges dEdges, cData analysis)
     /* draws analysis results of vertical colour analysis in frame */
{
  int x,y, whiteValue,
      realLeftX = apply_X(dFrame, dEdges.leftX),
      realRightX= apply_X(dFrame, dEdges.rightX);
  channel *pixelPtr=pos_ptr(dFrame,dEdges.leftX,dFrame.ysize),
          *pPtr2;

  for(x=realLeftX;x<realRightX; x++)
    {
      //red graph
      pPtr2 = MAX( (pixelPtr-(analysis.red[x]*dFrame.xsize*4)),
                (dFrame.handle+x*4));
      colour_pixel(pPtr2,255,0,0);
      // green graph
      pPtr2 = MAX( (pixelPtr-(analysis.green[x]*dFrame.xsize*4)),
                (dFrame.handle+x*4));
      colour_pixel(pPtr2,0,255,0);
      // blue graph
      pPtr2 = MAX((pixelPtr-(analysis.blue[x]*dFrame.xsize*4)),
               (dFrame.handle+x*4));
      colour_pixel(pPtr2,0,0,255);

      whiteValue = (analysis.red[x]+analysis.green[x]+analysis.blue[x])/3;
      // White Graph = avg.value RGB
      pPtr2 = MAX((pixelPtr-(whiteValue*dFrame.xsize*4)),
                (dFrame.handle+x*4));
      colour_pixel(pPtr2,255,255,255);

      pixelPtr += 4;
      // gridlines
      for (y=dFrame.ysize; y>=0;y-=50)
       {
         colour_pixel(pos_ptr(dFrame,x,y),100,100,100);
       }
    } //x-loop
```

```
}



void draw_cross(frame cFrame, int X, int Y,
             channel R, channel G, channel B)
     /* draws a cross with colour R, G, B around position (X,Y), in
       frame cFrame */
{
  channel *rPixelPtr;
  int i=0;

  // middle of cross (always works)
  rPixelPtr = pos_ptr(cFrame, X,Y);
  colour_pixel(rPixelPtr, 255, 255, 255);
  //topleft of cross (only if not to close to upper & left edge)
  if ((X>1)&&(Y>1))
    {
      rPixelPtr = pos_ptr(cFrame, X-1, Y-1);
      colour_pixel(rPixelPtr,R,G,B);
      rPixelPtr = pos_ptr(cFrame, X-2, Y-2);
      colour_pixel(rPixelPtr,R,G,B);
    }
  //bottomleft of cross (only if....)
  if ((X>1)&&(Y<cFrame.ysize-2))
    {
      rPixelPtr = pos_ptr(cFrame, X-1, Y+1);
      colour_pixel(rPixelPtr,R,G,B);
      rPixelPtr = pos_ptr(cFrame, X-2, Y+2);
      colour_pixel(rPixelPtr,R,G,B);
    }
  //topright of cross
  if ((X<cFrame.xsize-2)&&(Y>1))
    {
      rPixelPtr = pos_ptr(cFrame, X+1, Y-1);
      colour_pixel(rPixelPtr,R,G,B);
      rPixelPtr = pos_ptr(cFrame, X+2, Y-2);
      colour_pixel(rPixelPtr,R,G,B);
    }
  //bottomright of cross
  if ((X<cFrame.xsize-2)&&(Y<cFrame.ysize-2))
    {
      rPixelPtr = pos_ptr(cFrame, X+1, Y+1);
      colour_pixel(rPixelPtr,R,G,B);
      rPixelPtr = pos_ptr(cFrame, X+2, Y+2);
      colour_pixel(rPixelPtr,R,G,B);
    }
}
```

8.3.5 Tracker.c

```
/******* tracker.c *******/

#include <stdio.h>
#include <stdlib.h>

#include "alldefs.h"
#include "capture.h"
#include "vision.h"
#include "draws.h"

static cData vColourData, hColourData;



void initialise_tracking (windowInfo *capWin,
                      frame *calFrame,
                      int *ftr_threshold,
                      setOfEyes *eyes,
                      int processReduced)
  // init sequence
{
  frame redCalFrame;
  printf("Initialising...\n");
  init_capture();
  *capWin = init_window("Vision");
  // calibration
  *calFrame = calibrate(*capWin, CAL_FRAMES_NO);
  // generate noise reduction calibration frame
  switch (processReduced)
    {
    case 1:
      redCalFrame = reduce(*calFrame);
      *calFrame = redCalFrame;
      break;
    case 2:
      redCalFrame = reduceDouble(*calFrame);
      *calFrame = redCalFrame;
    }
  *ftr_threshold = 75; //initial value of threshold
  eyes->leftX =
  eyes->leftY =
  eyes->rightX=
  eyes->rightY=0; // initial eye positions
  printf("screensize processed: %d x %d\n",capWin->xsize, capWin->ysize);
}



frame track_frame(frame calFrame,
                int *ftr_threshold,
                setOfEyes *prevEyes,
                allEdges *resultEdges,
                int processReduced)
  // the tracking itself
{
  frame procFrame, redProcFrame, RPframe2, RPframe3, *procFramePtr;
  allEdges noseFindEdges;
  listElt *candidates=NIL;

  procFrame = capture_frame();
  switch (processReduced)
    {
    case 0:
      procFramePtr = &procFrame;
      break;
```

```
    case 1:
      redProcFrame = reduce(procFrame);
      procFramePtr = &redProcFrame;
      break;
    default:
      redProcFrame = reduceDouble(procFrame);
      RPframe2 = reduceDouble(procFrame);
      procFramePtr = &redProcFrame;
    }

  *resultEdges = subtract_image(calFrame, *procFramePtr,
                          COLOUR_THRESHOLD,
                          X_PIXEL_SKIP);

  detect_edges(*procFramePtr, resultEdges, FACE_SCAN_WIDTH);

     // horizontal and vertical analyses. not part of actual tracking;
     // they run correctly only on original frame.
  // VCAnalysis(procFrame, resultEdges, &vColourData);
  // HCAnalysis(procFrame, resultEdges, &hColourData);

  candidates = inv_red_filter(*procFramePtr, *resultEdges, ftr_threshold);

  centrality_score(*procFramePtr, candidates);
  remove_least_centrals(*procFramePtr, &candidates);

  blob_averages(*procFramePtr, &candidates);

  get_eyes(*resultEdges, &candidates, prevEyes);

  //  calc_ideal_eyes (*resultEdges, prevEyes);
  return(OUTPUT_FRAME);
}



void draw_results(frame drawFrame,
               allEdges resultEdges, setOfEyes theEyes)
  // draw tracking results
{
  draw_box(drawFrame, resultEdges);

  /* these display analysis result. !!! run ONLY when OUTPUT_FRAME is set to
     procFrame (= original frame) otherwise they screw up */
  // draw_V_analysis(drawFrame, resultEdges, vColourData);
  // draw_H_analysis(drawFrame, resultEdges, hColourData);

  // draw_ideal_eyes(resultEdges,drawFrame);
  draw_cross(drawFrame, theEyes.leftX, theEyes.leftY, 0,255,255);
  draw_cross(drawFrame, theEyes.rightX, theEyes.rightY, 255,255,0);
}
```

8.3.6 Gltest.c

```
/*   ****** gltest.c ******

  this unit contains some routines for creating an openGL scenery based on,
  and reacting to, the eye-tracker, giving the eventual fish-tank VR
illusion
  this project was all about :)

*/

#include <GL/glx.h>
#include <GL/gl.h>
#include <GL/glut.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>
#include <gl/image.h>

#include "alldefs.h"
#include "capture.h"
#include "tracker.h"

// distance units are cm's. distances are not completely accurate.

/* static variables, YUCK! I hoped to get a better way of passing parameters
   but openGL won't let me... At least they're only visible within this unit
   init_graphics will take the corresponding values from the main process
and
   copy them into these */

// openGL variables...

static double frustumLeft  = -12.0,
              frustumRight =  12.0,
              frustumBottom= -11.0,
              frustumTop   = 11.0,

              I=0,

              eyeX = 0, // these will later contain the user's eye position
              eyeY = 0,
              eyeZ = 0;

// eyetracking variables...

static int *ftr_th, procReduced;

static frame *cFrame;

static setOfEyes previousEyes;



void dolighting(void)
     // sets a certain lighting to the scenery
{
  GLfloat light0_ambient[] = {0.5, 0.5, 0.5, 1.0};
  GLfloat light0_diffuse[] = {0.75,0.75,0.75,1.0};
  GLfloat light0_specular[] = {0.75,0.75,0.75,1.0};
  GLfloat light0_position[] = {50.0, 50.0, 0.0, 1.0};
  /* final 1.0 = non-directional */

  GLfloat light1_ambient[] = {0.5, 0.5, 0.5, 0.75};
```

```
  GLfloat light1_diffuse[] = {0.75,0.75,0.75,0.75};
  GLfloat light1_specular[] = {0.75,0.75,0.75,0.75};
  GLfloat light1_position[] = {-10.0, 100.0, 1000.0, 0.75};

  glLightfv(GL_LIGHT0, GL_AMBIENT, light0_ambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, light0_diffuse);
  glLightfv(GL_LIGHT0, GL_SPECULAR, light0_specular);
  glLightfv(GL_LIGHT0, GL_POSITION, light0_position);

  glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
  glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
  glLightfv(GL_LIGHT1, GL_SPECULAR, light1_specular);
  glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
  glEnable(GL_LIGHTING);
}



void draw_model(void)
     /* draws a model in the scenery, consisting of a number of cubes on a
       plane */
{

  GLfloat white  [] = {1.0, 1.0, 1.0, 1.0};
  GLfloat red    [] = {1.0, 0.0, 0.0, 1.0};
  GLfloat green  [] = {0.0, 1.0, 0.0, 1.0};
  GLfloat blue   [] = {0.0, 0.0, 1.0, 1.0};
  GLfloat yellow [] = {1.0, 1.0, 0.0, 1.0};
  GLfloat magenta[] = {1.0, 0.0, 1.0, 1.0};
  GLfloat cyan   [] = {0.0, 1.0, 1.0, 1.0};
  GLfloat gray   [] = {0.5, 0.5, 0.5, 1.0};
  GLfloat desk   [] = {0.7, 0.7, 0.7, 1.0};

  dolighting();

  glPushMatrix();             // origin of model

  glTranslated(-2.0, -20.0, -41.0);  //to far edge of desk
  glMaterialfv(GL_FRONT,GL_DIFFUSE,white);
  glutSolidCube(4.0);

  glTranslated(4.0,0.0,0.0);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,cyan);
  glutSolidCube(4.0);

  glTranslated(0.0,4.0,0.0);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,green);
  glutSolidCube(4.0);

  glTranslated(-4.0,0.0,0.0);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,yellow);
  glutSolidCube(4.0);

  glTranslated(-1.0,1.0,-5.0);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,red);
  glutSolidCube(6.0);

  glTranslated(1.0,-5.0,1.0);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,magenta);
  glutSolidCube(4.0);

  glTranslated(4.0,0.0,0.0);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,blue);
  glutSolidCube(4.0);

  glTranslated(0.0,4.0,0.0);
  glMaterialfv(GL_FRONT,GL_DIFFUSE,gray);
  glutSolidCube(4.0);
```

```
  glPopMatrix();  // back to origin

  glPushMatrix();          // store origin
  glTranslated(0,-22,0); // down to desk height (?)
  glRotated(-90.0, 1,0,0);  // face down to desk
  glMaterialfv(GL_FRONT,GL_DIFFUSE,desk);
  glRectd(-63,47,55,-28);  // draw rectangle representing desk
  glPopMatrix();           // and return to origin


  //  glPopMatrix();

}



void calculate_all(void)
     /* transforms the coordinates found in the images to coordinates in the
       virtual openGL space */
{
   // first track the eyes, then calculate their coordinates
  frame currentFrame;
  allEdges faceEdges;
  currentFrame = track_frame(*cFrame,
                         ftr_th,
                         &previousEyes,
                         &faceEdges, procReduced);
  finish_with_frame();

    /* transform eye coordinates into openGL coordinates (sort of).
       requires some geometry which is at the moment oversimplified. The
       multiplication factor is the actual distance in space corresponding
       with 1 pixel in the image. this depends on your Z distance.
    */
  eyeX  = ((previousEyes.leftX)-160)*0.0718;
  eyeY  = ((previousEyes.leftY)-120)*0.0718;
  eyeZ  = -50;//-10*sin(0.03*I); // generates a nice sinoid movement
  I++;
  frustumLeft  = -12.0+eyeX;
  frustumRight =  12.0+eyeX;
  frustumBottom= -11.0+eyeY;
  frustumTop   =  11.0+eyeY;
  glutPostRedisplay();
}


void graphics_display(void)
     /* adjusts the openGL scenery settings according to the calculations
       from calculate_all and redraws the scenery */
{
  float distanceToScreen=50.0,
        screenWidth     =24.0,
        screenHeight    =22.0;

    glClear(GL_COLOR_BUFFER_BIT| GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Set up the viewing matrix...
    glFrustum(frustumLeft, frustumRight, frustumBottom, frustumTop,
             distanceToScreen, 150.0);
    // (left, right, bottom, top, near, far)

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    // now transform view from camera to monitor:
```

```
    //translate current position from user's eye to origin...

    glRotated(15.0, 1, 0, 0); // 15 degrees around X-axis (monitor tilt)
    glTranslated(0,-24,0);      // cam. center to mon. center

    // Then draw the model...
    glTranslated(eyeX, eyeY, eyeZ);
    draw_model();

    // And since we're double buffering...
    glutSwapBuffers();
    glFlush();
}

void keyboard_stuff(unsigned char key, int mouseX, int mouseY)
     // processes keyboard keys pressed while OpenGL scene is running
{
  if (key =='x')
    exit(-1);
}


void init_graphics(frame calFrame,
                int *ftr_threshold,
                setOfEyes theEyes,
                int processReduced)
// Initialise the displayer.  Open window, get GL set up.
{
  char Args[12];
  int command;
  int i;
  FILE *f;

  /* set the local statics to the parameter values */
  cFrame  = &calFrame;
  ftr_th = ftr_threshold;
  previousEyes = theEyes;
  procReduced=processReduced;

  glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGBA|GLUT_DEPTH);
  glutInitWindowSize(800,800);  // this is on heff's monitor 24.0x22.0cm
  glutInitWindowPosition(240,112); //centers window on screen
  glutCreateWindow("Vision  (press X to exit)");

  glClearColor(0.0,0.0,0.0,1.0); // =black, fully non-transparent
  glMatrixMode(GL_PROJECTION);

  /* these two keep each other alive. After display is finished, gl will be
     idle, thus calling the tracker. after that is finished, calculate_all
     orders GL to redisplay (using new values of course) */
  glutIdleFunc(calculate_all);
  glutDisplayFunc(graphics_display);

  glutKeyboardFunc(keyboard_stuff);
  glEnable(GL_DEPTH_TEST);

  glEnable(GL_BLEND);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHT1);
  glShadeModel(GL_SMOOTH);

  glutMainLoop();
}
```

8.3.7 Main.c

```
/****** main.c *****

  some simple example code of what an eye tracker program could look like.
  shows how to display results, and how to apply em in an openGL scenery.
  */

#include <stdio.h>

#include "alldefs.h" // tracking parameter & type definitions
#include "util.h"    // small utility routines
#include "capture.h" // video capturing routines
#include "tracker.h" // main tracking routines
#include "gltest.h"  // opengl routines


void main(void)
{
  // declare these variables:
  windowInfo capWin;
  frame calFrame, redCalFrame, dispFrame;
  int ftr_threshold;
  setOfEyes foundEyes;
  allEdges faceEdges;
  listElt *eyesList;
  char c;

  int i;
  HR_TIMER time1, time2;

  // call init routine. initialises variables where needed, and calibrates.
  initialise_tracking(&capWin,
                  &calFrame,
                  &ftr_threshold,
                  &foundEyes,PROCESS_REDUCTION);


  /* call this loop to display tracking results in a camera window
  */
  time1 = hrGetTime();
  for(i=0; i<FRAMES_NO; i++)
    {
      dispFrame = track_frame(calFrame,
                          &ftr_threshold,
                          &foundEyes,
                          &faceEdges,
                          PROCESS_REDUCTION);

      draw_results(dispFrame, faceEdges, foundEyes);

      display(dispFrame, capWin);

      finish_with_frame();
    }
  time2 = hrGetTime();
  time2 -= time1;
  if (time2 != 0)
    {
      printf("time to display %d frames: %7.0f ms\n", i,(time2));
      printf("average display time 1 frame: %3.2f ms\n",
i,(time2)/FRAMES_NO);
      printf("average framerate: %16.2f fps\n", ((1000*FRAMES_NO) /
(time2)));
    }
    /**/
  close_window(capWin);
```

```
  /* call this routine to apply eyetracking to an OpenGL scenery (defined in
     gltest.h).
     init_graphics will initiate a loop, calling track_frame like in the
     loop above, not drawing and displaying but using the results instead.

  init_graphics(calFrame, &ftr_threshold, foundEyes,
              PROCESS_REDUCTION);
  */

  capture_end();
}
```