Rasdalf Purple

Gandalf Grey's less famous cousin

Daniel Kallin
Mikael Sundberg
Jannis Tsiroyannis
Linus Wiklund

21 Augusti 2007

Abstract

A report on the mechanical and programmatical construction of a autonomous robot designed to play soccer as part of the examination in the course 2D1426 *Robotics and Autonomous Systems* given at KTH in spring 2007.

Contents

Chapter 1

Preface

1.1 Introduction

This report details the construction of the robot Rasdalf Purple. Rasdalf Purple is a robot designed to compete in robot soccer by a variation of the RoboCup rules. We, as authors of this report and builders of the robot have set out to maintain a simple and spartan design philosophy. We have tried to equip the robot with the bare essentials and as little else as possible. The report is structured so as to make it easy to look up specific parts without having to go through large amounts of text. This is done both in the interest of maintaining our design philosophy and to be as useful a resource to future teams as possible.

Chapter 2

Hardware design

2.1 Design philosophy

The physical design of the robot did not follow any detailed plan. Instead it had more of a iterative nature with components added or redesigned when needed. This was not as inefficient as it might sound. It caused almost every part of the robot to be constructed at least twice due to redesigns but we also maintained a high degree of flexibility.

2.2 Camera

We decided early on to mount the camera as high up as possible and tilted downwards to avoid seeing anything outside the field and getting an optimal situation for distance determination. Since we barely saw the goals when situated at the far end of the field and only saw half of the ball when we possessed it the calibration of the camera was very delicate but the excellent vision and distance determination was clearly worth it.

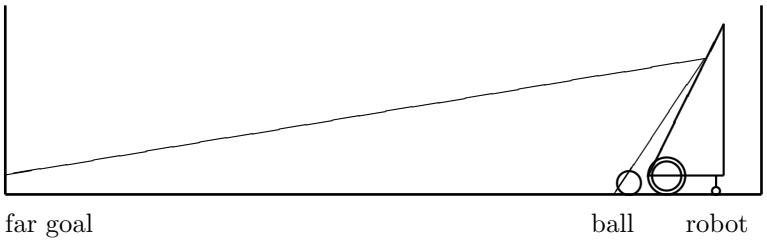

Figure 2.1: Qualitative schematic of robot view frustrum

2.3 Ball handling

During the construction of the robot we came to the conclusion that the primary issue regarding ball handling would be how to turn the robot around without

losing the ball. There is also the secondary issue of how to stop the ball when the robot stops driving forward. The original plan for how the robot would handle these problems was twofold. First of we decided that the best way to turn with the ball would be to keep the ball as close as possible to the robot's center of rotation and turn the robot around the ball rather than with it. This is the primary reason the robot has its two powered wheels placed in the absolute front of the robot.

The second part of the plan involved a revolving rod placed across the front of the robot often called a 'dribbler'. The dribbler's job is to apply backspin to the ball effectively keeping the ball from rolling away from the robot. At the day of the competition the dribbler was only partially operational and therefore disabled during the actual competition. The placement of the wheels proved an adequate if not perfect solution to the ball handling problem.

2.4 Robot picture

Figure 2.2: The final robot hardware design

Chapter 3

Software design

3.1 Color classification

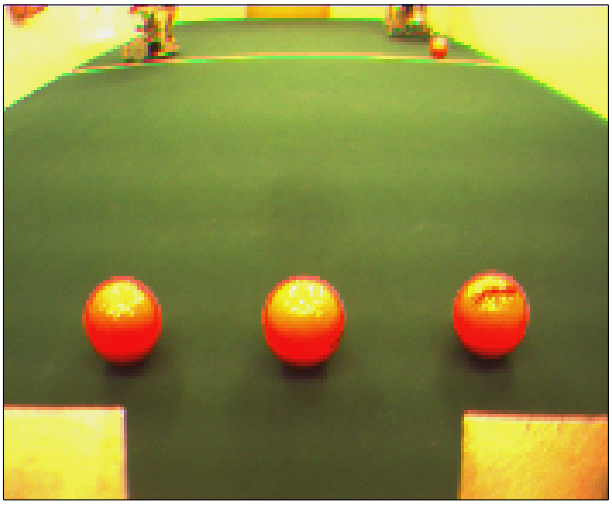

Figure 3.1: Sample camera image.

To make it easier for us Rasdalf's world was color-coded. The colors it should know about was:

- Blue and Yellow - Goals
- Orange - Ball
- White - Walls and lines on the playing field
- Green - Playing field
- Purple - Opponents

But as can be seen in our images below, the walls come in a rather white-yellow color, the ball is orange and yellow and the opponents come in a multitude of colors. More information about the color codes and other tournament rules can be

found in the Rules document at `http://www.csc.kth.se/utbildning/kth/-kurser/DD2426/rules/` (15/6/2007, *KTH CSC 2D1426: Project competition rules*).

3.1.1 Physical color samples referencing

The color sampling method which we used during Lab1 was very simple and quite unique. When we discussed and tried to analyze the problem of color based object classification we identified the following difficulties:

- The camera's autobrightness function caused the same physical color to span a long range of different brightness values.
- The lighting changed during day time and night time in the laboratory and was also expected to change when the field was moved to the tournament hall.

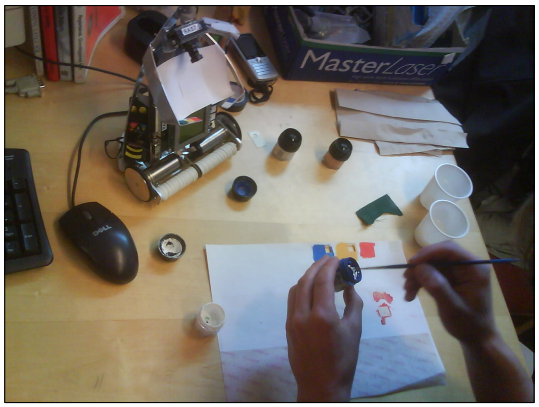

Figure 3.2: Making the "LUT"

We realized that these difficulties could be alleviated by moving the color references away from software - where it would be stored as a LUT or as threshold values - and out into the physical world in front of the camera. We mounted arms on the robot which held small patches colored in the same colors as the objects in the soccer game. These samples experienced the same lighting fluctuations and their colors were automatically processed by the camera's autobrightness function. Our algorith sampled these color references and used those references in the pixel classification process. A pixel's color to be classified was compared with the average color within a sample by calculating a numerical 'distance' between the two colors. If the distance was greater than a certain threshold it was considered *unclassified* and otherwise it was given the class of the color sample that was the closest. We tried different metrics to calculate the distance.

Color metrics

- Max color component deviation. $d = \max(|r - r_0|, |g - g_0|, |b - b_0|)$.
- Euclidean distance in RGB space. $d = \sqrt{(r - r_0)^2 + (g - g_0)^2 + (b - b_0)^2}$.

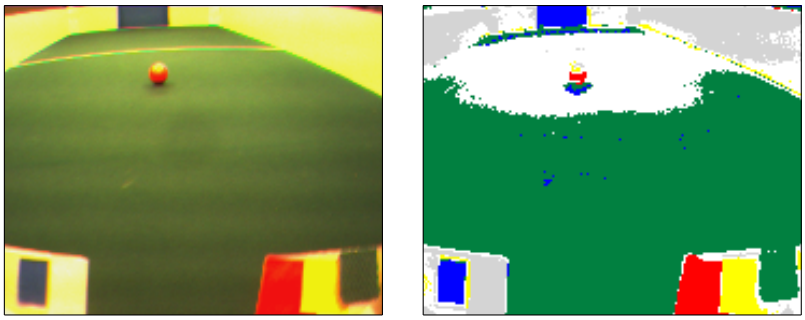

Figure 3.3: Physical color referenced classification. *Left*) Camera output. *Right*) Image pixels classified.

We considered other variations of this reference approach. Several pixels of the references were sampled and averaged together. But more information could have been retained. The variations inside the reference patches would be stored in some fashion to increase the robustness of the classifications. We also discussed comparing the color distance between different samples as a measure of how well the camera performed at the current conditions.

3.1.2 HSV classification

After finishing Lab1 we realised that the color classification code was not always working as it should be. For example the robot was functioning better during the evenings and late afternoons than during the mornings and lunch-time. So we descided on doing some experimenting on classification in the HSV color space with hard-coded thresholds.

HSV color space

The HSV color model have some advantages over the RGB model. The color is a single coordinate in the HSV model, the other two describing saturation and brightness. Since every object on the soccer field is coded in bright colors it should therefore be possible to differentiate between different objects by comparing this single coordinate. A drawbacks was that one color - white - is not uniquely defined in HSV space.

Calibrating tresholds

Working with most of the test-pictures we had taken earlier on we started out classifying the thresholds by hand in all the pictures that was deemed not to be extreme cases of camera screw-up, trying to avoid overlaping ranges as much as possible. This code was then pasted into the pixel classification function, tested on the sample images and then fine-tuned. This procedure was then redone a few times with new images from the robot posing different situations. This seemed to work pretty good so we decided on using it, and we did not have to do any recalibration while in the tournament room.

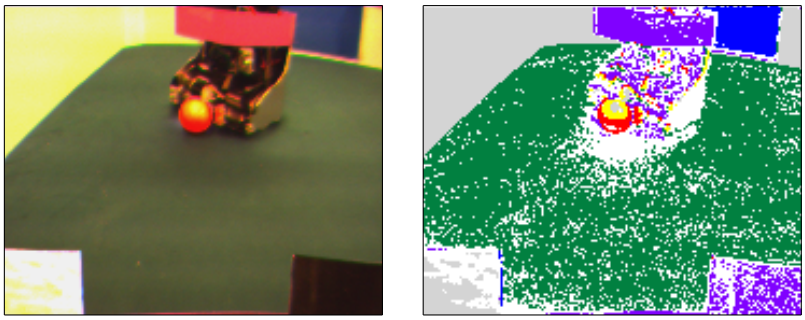

Figure 3.4: HSV color space classification. *Left*) Camera output. *Right*) Image pixels classified.

3.2 Object recognition

3.2.1 Ball detection

Horizontal edge detection

Our first idea of how to handle ball recognition was to go through the camera image and note the leftmost and rightmost pixels that was classified as being ball pixels. The idea was that the coordinates of these pixels would tell us everything we needed to know about the ball. The mean of the coordinates could be considered the center of the ball and was used to calculate the angle between the the robot's bearing and the direction of the ball. The difference between the leftmost and rightmost coordinates would also tell us the apparent width of the ball in the image, from which we could determine the distance to the ball. This was done using a precomputed lookup table. This approach failed due to imperfections in the camera. Noise in the image often led to pixels at random locations being classified as orange. Also intersections between wall and floor showed a tendency to have a false orange tinge in the images.

Neighbourhood verification

To address the problem of noisy images we moved on to the idea that a pixel should not be considered part of the ball unless it was surrounded by other pixels that had similar color. We tried several implementations for this. For example we checked a square around the pixel in question and only classified it as ball if more than 75% of the square's pixels also had the correct color. This was not entirely unsuccesfull but still had problems similar to the naive approach.

Coherent blob identification

Another idea was that to find the ball it is not enough to look at separate orange pixels. We instead decided to look at groups of pixels. This was done by first finding any orange pixel in the image and then recursively checking all its neighbours. This method would for every given image return one or more groups of orange pixels which could then be compared in size, position and proportions to decide the most likely, if any, to be the ball.

Coordinate averaging

When we got the coordinate transformation described in section 3.6 working we no longer needed accurate measurements of the apparent size of the ball to localize it. But we still needed a very precise measurement of location of the center of the ball. The solution was simply to consider the ball's apparent center to be the averaged coordinates of all the pixels that were classified as *ball*. Any noise would only have a negliable effect on the averaged coordinate.

The U-method

One of the ball recognition algorithms we tried involved trying to find a U of orange pixels. When finding an orange pixel it basically tried to go as far down finding new orange pixels as it could, then started going to the right, and afterwards it went upwards. It did not get that much of an effort getting it to work satisfyingly though, and in the tests we did it almost never found the ball. This might often have been because the ball-pixels was a mix of yellow and orange pixels, and not always U-shaped in long ranges. The reason we decided on trying an U-shape and not an O-shape was that the top of the ball often got classified as being yellow. This algorithm could probably have worked if it had been given enough effort and time. We did not have the time though.

3.2.2 Goal detection

Corner detection

The goal search pattern was simplified by the camera's position. We never saw outside the field and we always had the goal along the top edge of the camera image whenever we faced one of the goals. The goal finding algorithm relied on the wall-floor intersection finding algorithm. (See Figure 3.5). Our camera placement made sure that if a goal was visible some part of it would occupy the top edge of the screen. We therefore only searched a thin scanline at the top of the camera image. When we had detected the leftmost and rightmost goal pixels we identified the wall/floor edges a few pixels outside the identified goal edges.

Largest Pixel-Class Group Method

This goal finding algoritm found the largest coherent pixel group of the same color. It did allow for some gaps though. The gap-size was specifyable so that there could be one or more pixels of a different color in the gap. This method was tried when we were only using RGB-color space for classification, and therefore there was a pretty large amount of miss-classification and camera "errors". It was also rather slow since it had to go through a large amount of pixels.

3.2.3 Opponent detection

When it came to detecting the opponent, time was a problem. There simply wasnt any time left for us to implement any of the functions we had planned that would be using opponent detection. So when it was time for competition our robot could see the opponents' purple color really well. She just didnt care.

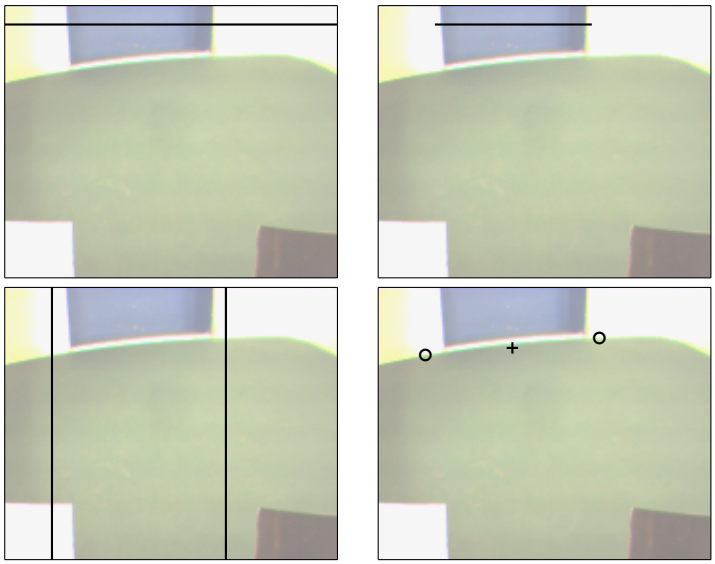

Figure 3.5: The goal finding process. *Top Left*) Goal pixel search in horizontal scanline. *Top Right*) Goal edges found. *Bottom Left*) Floor edge search in vertical scanlines outside goal edges. *Bottom Right*) Floor edge points (o) found. Goal center ($+$) approximated as their centroid.

3.2.4 Wall detection

When searching for the wall-floor intersection along a vertical scanline we at first tried a fixed brightness threshold approach. The algorithm assumed it had found the intersection when the pixel brightness fell below a certain threshold (See Figure 3.6 for a brightness plot). This was not robust enough in different auto brightness compensation levels so we resorted to a brightness derivative threhold approach. The brightness derivative was approximated with a difference with a step length greater than one pixel. When this difference was greater than a certain threshold the scan stopped and reported that an intersection had been found.

3.3 Kinematics

We used the built in Vω-interface but after some testing we realized that it had some problems. For example if the robot started a new `drive-straight` while one was already running. The robot started to move really slow. We solved this by mostly using `setspeed` and calculating the time it should take for the robot to travel the wanted distance in the speed we had set. This worked really well as-long as the batteries were charged. When the batteries started to get discharged the robot started to move slower making our predefined calculations wrong and therefore the distance became too short. Unfortunately we did not have time to solve this in a satisfying way. Instead we used the not so good drive-straight method when we wanted to be sure we traveled the correct distance. Which became clear in the tournament where our initial move went totally wrong.

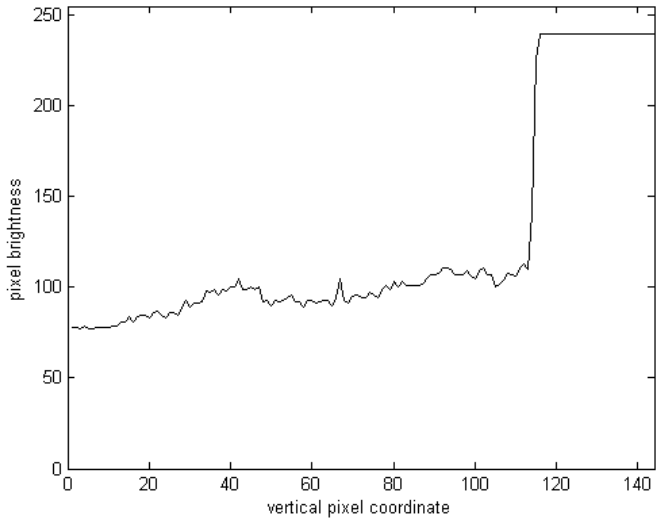

Figure 3.6: Pixel brightness along vertical scanline.

Another disadvantage the Vω-interface is that if used, the camera can not take pictures while the robot is moving. At least we did not get it to work in a way that met our standards.

3.4 Memory

The RoBIOS' Vω-interface has a built in odometric functions that (allegedly) could keep track of the position of the robot during motion relative its initial postion. We wrote a simple shell around these functions for use in our scoring algoritms. The motivation being that once the goal would have been sighted its location could be stored in memory and later retrieved. The robot would then be able to perform much more advanced ball searches since there would be no need to start from scratch when relocating the goal once the ball was retrieved. The robot would be able to move into an optimal scoring pose as soon as it saw the ball using the memorized position of the goal as a reference.

The memory was not designed to place objects in a global map but would instead memorize only the goals position in the robots local reference frame. It also provided a function to flush (to forget) the memory that was to be called

Unfortunately we never had time to actually use the memory functions. We can therefore not assess the Vω-odometry's accuracy but we can make some (unflattering) assumptions considering the overall poor performance of the Vω-interface.

3.5 Fixed point math

During development we fought alot with strange return values and weird results in general. After some research we came to the conclusion that floats are evil.

That conclusion made it clear to us that we needed functions so we could use integers instead of floats. So we created our *millimath.c* library containing trigonometric values and some needed math functions. Doing all calculations and return values in integers. The only problem this caused was that since we used the built in Vω-interface that wants floats we had to compensate for that when calling those functions. Another advantage of using integers instead of floats is that its faster to compute integers. But as far as we know our (lack of) programming skills made that advantage go away.

3.6 Coordinate transformation

Rasdalf works with several different two-dimensional coordinate systems; camera image coordinates and local reference frame coordinates. The *transform.c* module provide transformation functions from the image coordinate frame to the local frame.

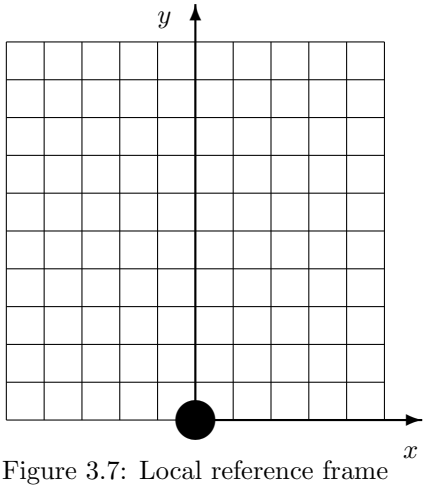

Figure 3.7: Local reference frame

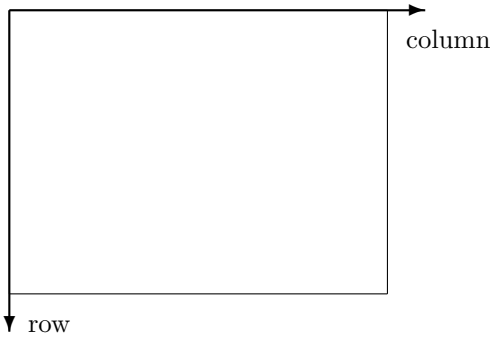

Figure 3.8: Image coordinate system

This transformation relied on the one to one correspondence between coordinates in the 2D floor plane and 2D coordinates in camera images. Real world

objects are not points and the objects on the soccer field were not strictly confined to the floor. But the assumption that they were simplified our coordinate transformations greatly whilst contributing only minor errors. And the greatest benefit was that with a algorithm capable of deducing distance to single points there was no need for robust (and thus complex) object size recognition routines.

A point (x_i, y_i) in image coordinates is approximately translated to a point (x_w, y_w) in the local reference frame by equations 3.1 and 3.2. The adjustment of the six parameters in these equations are breifly described in section 3.8.1.

$$x_i, y_i, x_w, y_w \in [0, 1]$$
$$h : \text{height of the robots camera}$$
$$\beta_h : \text{angular width of the camera's FOV}$$
$$\beta_v : \text{angular height of the camera's FOV}$$
$$\alpha_h : \text{angle between local y-axis and left edge of the FOV}$$
$$\alpha_v : \text{angle between horizontal plane and top edge of the FOV}$$
$$\gamma : \text{angular offset}$$

$$y_w = \frac{h}{\tan(\alpha_v + \beta_v y_i)} \tag{3.1}$$

$$x_w = y_w \tan(\alpha_h + \beta_h(\gamma + x_i)) \tag{3.2}$$

3.6.1 Triangulation

To get the robot to figure out its position we tried to use triangulation based on the goal posts. But since we used a rather naive approach the robot could not be further out on the sides of the field than the right- or leftmost goal post. If this was satisfied we got a rather good location estimate of the robot in our test cases. What we did was to estimate the distance to one of the goal posts and the angle to both of them. Since we knew the global position of both of the goal posts and the distance between them we could then approximate the x and y distances in the global frame from one of the posts to the robot if above constraints were satisfied. But since we did not have a way of knowing if above constraints were satisfied and no time to figure it out, this was never used on the robot.

3.7 State engine

In the state engine each state was implemented as a single function in the *ai.c* module. Each state function returned the next state that the robot should enter as its return value.

Attack

The **attack** state is a special initial state designed to quickly deny the opponent the ball in the very beginning of the game. It consisted of a series of hard coded movements that were executed blindly. No sensor readings are collected while in the **attack** state since we decided we couldn't afford to wait for the auto-brightness level to stabilize. Driving without sensor readings requires precise knowledge of the environment, knowledge which is only availible at the start of

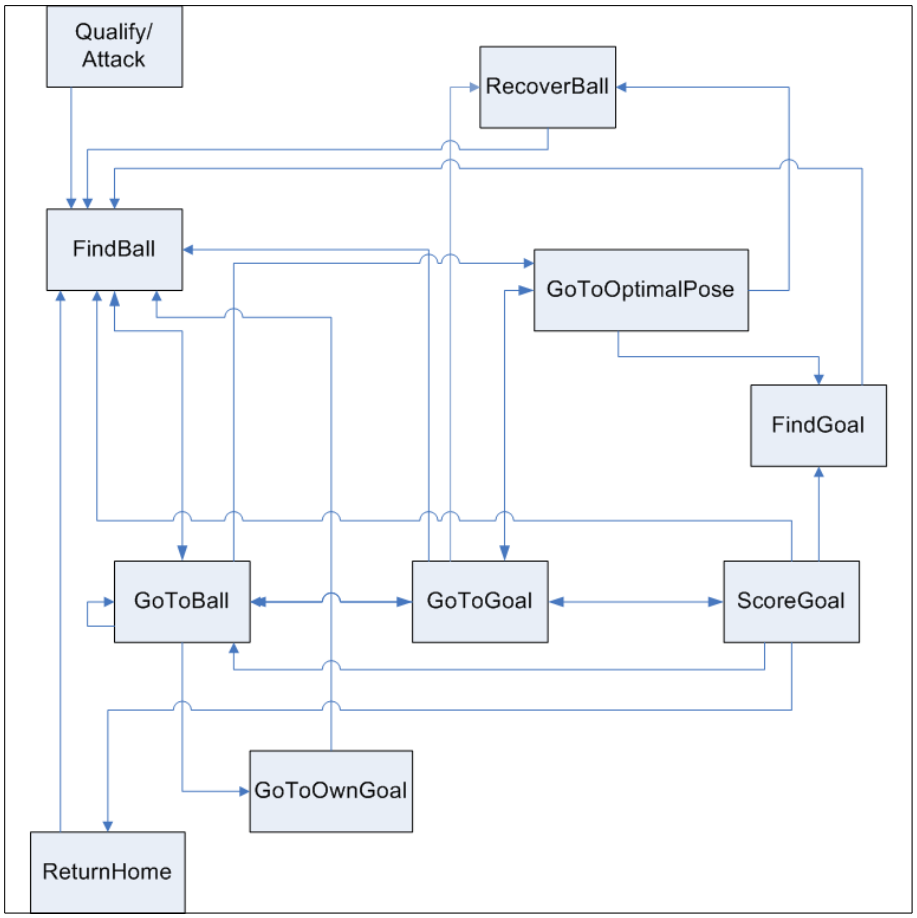

Figure 3.9: State engine flow chart. Note that some arrows are double directed.

the game when the rules define the locations of the robot, the opponent and the ball. At the start of the game the ball is located in between the two robot players. Because of this the robot must move in an S-shape to avoid handing the opponent the ball (the opposite of the desired behaviour). The **attack** state also provided our robot with a comically aggressive behaviour for increased audience appeal.

In practice this behaviour failed. It successfully aquired the ball during tests but in the tournament it failed to hit the ball. We are baffled by this discrepancy that might be attributed to any or several of a score of reasons. Fluctuating battery voltage levels, encoder errors and programming errors are possible culprits.

Hassle

This state was supposed to disturb (hassle) our opponent and possibly steal the ball. It was not implemented in our robot because of our lack of a opponent distance determination function. The state machine was supposed to invoke this state whenever the ball and the opponent was seen togheter. It would then

aim at the opponent, drive towards her and stop a few centimeterns before collision. It was then supposed to turn 90° and start to make tight laps around the opponent at high speed. The opponents vision would thus be greatly disturbed, ideally triggering any avoidance routines in its ai, and if it tried to move hopefully knocking the ball out of its despicable grip.

Qualify

During the qualification a different inital behaviour than that offered by the **attack** state was desired. This state was intended as an alternative. It only consisted of a short drive closer to the ball before it handed control over to the **FindBall** state.

FindGoal

This state does almost the same thing as the **FindBall**-state. Except it looks for the opponent goal instead of the ball, and when it finds it it stops, returning the **FindBall**-state. After turning a full 360° it turns back until facing the wall furthest away and drives in that direction, then start looking for the opponent goal again. It was never used during the competition, since the point in using it was as an aid for the robot memory which was never used.

FindBall

This state was supposed to be invoced if the robot had lost the ball. It should then try to find the ball as soon as possible. It tries to do so by turning step by step until it sees the ball. While turning it also takes notice of the two goals and the longest wall distance. It it turns a full 360° and does not see the ball, it tries to reposition itself. The thought here was that if it can't see the ball it should probably try to go somewhere else, so if it had found both goals it drives to the goal farthest away. If it did not see both goals, it instead tries to drive one third of the distance to the most distant wall. In testing this seemed to work well enough, but in the competition we could never really see it doing any driving as a result of this function. Eventually it always found the ball, and then it return the **GoToBall**-state.

GoToBall

The **GoToBall** function has the simple job to go to the ball when it is visible. It does this by simply turning to the ball and going forward untill it has the ball. When it has the ball it calls the apropriate state depending on if it can se the opponents goal, its own goal or no goal at al.

GoToGoal

The GoToGoal state is called when the robot had the ball. The problem was since our memory functions wasn't used, the robot didn't always know where the goal was. And since we didn't have a roller it was really hard to search for the goal without dropping the ball. We tried to solve this by driving in a circle instead of rotating. Togheter with our **LostBall** state it worked ok atleast. When the goal was found the robot minimised the angle and drove closer until

it was close enough to call the **ScoreGoal**-state. The GoToGoal state was also supposed to avoid the opponent while driving towards the goal. But since we ran out of time we skipped that part.

GoToOptimaPose

This state represents the robot knowing where the ball and the enemy goal is, and attempting to move into a position and orientation (pose) from where it can score a goal . This is done by calculating the vector from the enemy goal to the ball within the robot's frame and then adding a short fixed distance to this vector. The robot will then turn and move to the coordinates representing the enemy goal's position plus the new vector. When the robot gets to the desired position it will turn once more to face the ball and enemy goal. This sequence of actions is executed blindly to avoid confusion from bad camera input. This proved an effective approach in most cases but it also had its shortcomings. For example the algorithm tends to be tedious when the robot tries to move to an optimal pose when it is already very close to the ball. On rare occasions the robot also crashed into walls while perfroming this maneuver, although this could probably have been avoided.

GoToOwnGoal

Rasdalf entered this state when he had to go towards his own goal to get the ball. If he did not have the ball and it was not in his way, he went straight towards the own goal, otherwise he tried to go around the ball towards his own goal.

ScoreGoal

The **ScoreGoal**-state is called by **GoToGoal** when the robot is in possession the ball and can see the goal straight ahead. This state has a simple behaviour. It drives three times the distance measured to the goal in high speed. Followed by 200mm in even higher speed. This to make sure we don't lose the ball and that we don't drive too short. When done it assumes it scored a goal and calls the **ReturnHome**-state.

ReturnHome

This state was one of the funniest to look at. The idea is that when the robot thinks it had scored, it should get back to its own side of the field as soon as possible. We tried five or six different ways of doing this, and spent lots and lots of hours on it. Trying different ways of figuring out how the robot was positioned on the field when scoring to get the best way of getting back home. Well, this final way didn't even use the camera, it just blindly relied on hitting the walls in the right way, which it pretty much always did. What it did was to drive out of the (supposed) goal in an arc, and then drive straight into the wall. It then kept driving into the wall a time long enough so that it should always have hit the wall. It then at least knew what side of the field it was on (or it could have know that at least, but well, it's memory-less). It then drove backwards a few seconds, turned, drove towards the own goal and then turned again so that it should pretty much be facing the center of the field. This turned

out to work pretty well, except for when it though that it had scored and wasn't really close to the opponent goal.

RecoverBall

The **recoverball** state encapsulated one of the most powerful, visually impressive yet simple behaviours our robot was capable of. Designed to be called whenever the robot didn't see the ball when it only recently did. The ball recovery algorithm then drove backwards a bit with the assumption that the ball had rolled off the edge of our ball scoop. Immediately after this it initiated the **findBall** state. It is a very simple behaviour but for an audience which can see how the ball slowly rolls closer and closer to the edge of the scoop to finally roll off it is both relieving and humorous to see the robot suddenly stop - as if startled - quickly back up and recover the recently lost ball.

3.8 Diagnostics and utility programs

3.8.1 Coordinate transform calibration

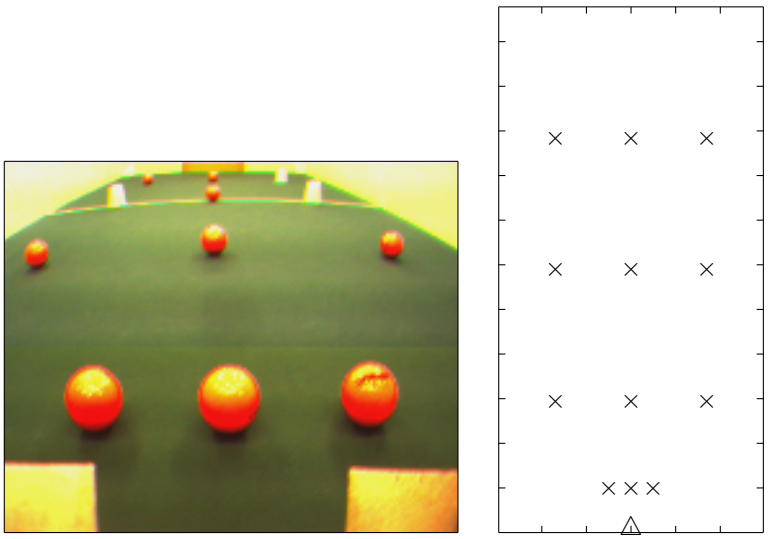

Figure 3.10: Calibration setup. *Left*) Calibration image with reference object setup. *Right*) Known positions of calibration objects ($\times$) in the robots ($\triangle$) local reference frame.

The image coordinate to local frame coordinate transformation was handled by the equations 3.1 and 3.2. The parameters $h, \beta_h, \beta_v, \alpha_h, \alpha_v, \gamma$ in these equations was either measured directly of the robot (the camera height h) or calibrated using a special MATLAB program (see Appendix 5.2.1). We placed 12 reference objects on carefully measured positions in front of the robot and stored camera images using our image capture software (Appendix 5.2.4). We then manually extracted the apparent positions in the image coordinate frame and recorded these in our MATLAB calibrator. The calibration was an iterative

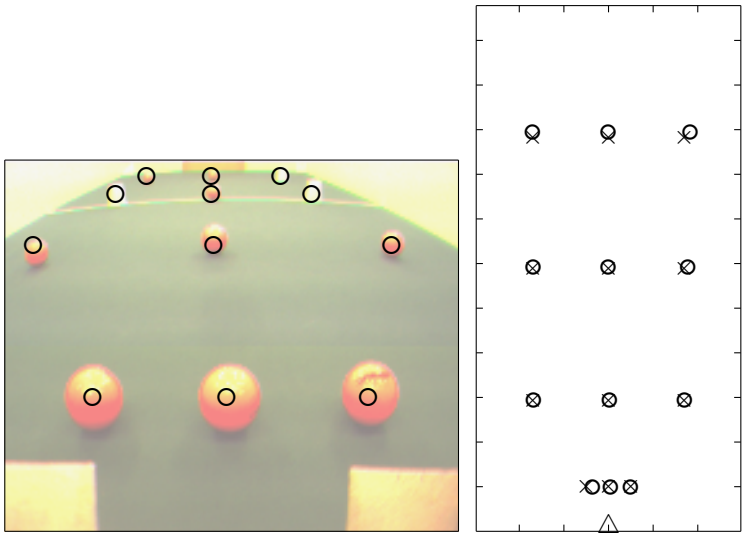

Figure 3.11: Coordinate transform. *Left*) Extracted object positions from image data (○). *Right*) Calculated positions from image data (○) superposed on known object positions (×) in the robots (△) local reference frame.

process with approximated initial values for the parameters. In each iteration the values were manually adjusted, something which worked very well and was quick thanks to the visual output of the software (see Figure 3.10 and 3.11).

3.8.2 Image capture

Our image capture software was developed a bit late into the development. We tried some software we found in the robios archives but since it used the very useless `colimages` we realised we had to make something on our own. It is a fairly simple program based on another program found on an old homepage for this course (*http://www.nada.kth.se/kurser/kth/2D1426/code/getAndSendBigRGBData.txt*). The program grabs ten frames and stores them in a vector to send them when the robot is connected to the PC again. Since no one had the time to make a program for the computer to properly and cleanly receive all ten files we had to manually either `cat` it to a file or use the `dl` program for each picture taken.

3.8.3 BMP parser

In the early stages of experimentation we used computer generated images to try different image analysis algorithms. In order to read these images a 24-bit uncompressed BMP parser was written. We chose the BMP format because it is a fairly simple one. There is a 40 byte header from which the dimensions of the image are decoded, followed by row after row of pixel data, starting from the bottom of the image and moving up. Every row of data is padded with enough empty bytes at the end to make its size evenly divisble by four. Source code for this parser can be found near the end of this report.

3.8.4 PPM reader

We decided to make our own ppm-file reader to be able to actually see what the robot "sees" after classifying its image. One can get the ppm-file from the robot in two different formats, one is plain text ascii, one is a binary format. You get the ascii one by cat:ing the serial port to a file (eg `cat /dev/ttySx > filename.ppm` where `x` is the serial port number) and the binary one by using the dl program supplied with the eyebot (with the `-u` parameter). Our ppm-file reader uses the binary format. Some things to think about while making your own ppm-file reader:

- Don't forget the null-char att the end of file. ;)
- Don't put any null-chars before end of file (eg. use 0x01 or such as black).
- Remember to analyze every pixel, you do want the whole image. (Though it might be a really good idea not to analyze the whole image on the eyebot, see Section 3.2 for more information).

Feel free to use our ppm-file reader as a first try. It's dead ugly but hey, it works! Using this we were able to actually see what our program regarded as being different objects, which helped us alot with our manual HSV-calibration and with comparing between our cheat-sheet color classification method and the hard-coded HSV.

3.8.5 Radar

A tool which was used during development and tuning of the perception and environment analysis algorithms was the program *radar.c*. It polled our image analysis and object recognition routines and displayed what they had deduced about the surrounding environment in a radar-like fashion on the robots tummy-display (Figure 3.12).

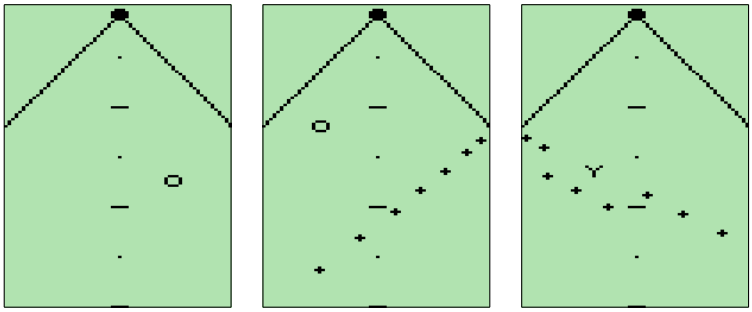

Figure 3.12: Three displays of the *radar.c* program. The robot is the topmost black dot and the lines are its view frustrum. A ruler is always drawn with marks each 250th mm. Symbols shown are ∘ (ball), Y (yellow goal) and + (wall).

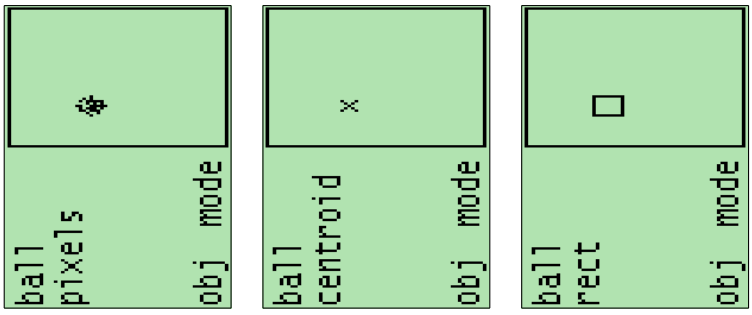

Figure 3.13: The three display modes of the *vision.c* program.

3.8.6 Vision

Correctly classifying colors as object turned out to be a significant part of the project. During the development our color classifier we made extensive use of the *vision.c* program and its required code structures inside *camera.c*. The Vision program was simply a shell that instructed the Camera module to visually display the images from the camera after they had been processed. The program offered a simple interface where the user could choose which object the algorithm should display and in which fashion it should present the information (Figure 3.13).

Object classes that could be visualised

- Ball
- Blue goal
- Yellow goal
- Floor
- Wall
- Opponent
- Unclassified

Availible visualization modes

- Pixels : only displayed the pixels that were classified as the selected object.
- Centroid : displayed a cross at the mean pixel coordinates.
- Rect : displayed a bounding rectangle enclosing the classified pixels.

3.8.7 Workstation color classifier

In order to be able to tweak the robot's color classification a small program was written for the PC that would take an image (from the robot's camera or some other sources) and classify every pixel in the image by the exactly same criteria that the robot used. The program would then alter the image, for example

changing every pixel it classified as ball to black. By looking at this image we could then determine how well the clasification criteria for that color worked. This method was used extensively and greatly streamlined the calibration of the classification algorithm.

Chapter 4

Conclusion

4.1 Advice for future course participants

There are thing we wish someone had told us before we started on this project. And there are things which we are really proud of having found out. Some of these findings we graciously offer to our successors. Love your robot. It is your mirror image.

- Use visualizing diagnostics programs.
- Implement your own drive interface. The RoBIOS Vω interface cause grave interference with the camera.
- Place the camera at the same level as the top of the field walls and as far back on the robot chassis as possible. Avoid looking above the field.

Chapter 5

Appendix - source code

5.1 Robot onboard software

5.1.1 Makefile

```
CC = gcc68
CP = cp -f
RM = rm -f

SOURCES = $(wildcard *.c)
TARGETS = $(SOURCES:%.c=%.hex)

all:
  $(CC) $(LDFLAGS) -o bot.hex ai.c camera.c kinematics.c main.c memory.c transform.c millimath.c

radar:
  $(CC) $(LDFLAGS) -o radar.hex radar.c ai.c camera.c kinematics.c memory.c transform.c millimath.c

vision:
  $(CC) $(LDFLAGS) -o vision.hex vision.c ai.c camera.c kinematics.c memory.c transform.c millimath.c

clean:
  $(RM) *.o
  $(RM) *.hex
  $(RM) *.elf
  $(RM) *~
```

5.1.2 Main

```
#include "eyebot.h"
#include <stdio.h>
#include "constants.h"
//---------------------------
int main(void){
  MemInit();
  KinInit();
  CamInit();

  int goal;  // opponent goal
  LCDPrintf("Which Goal?\n");
  LCDMenu("blue","","  ye","llow");
  int k = KEYGet();
  if(k==KEY1)
    goal = BLUE_GOAL;
  else
    goal = YELLOW_GOAL;

  LCDPrintf("Which state?\n");
  LCDMenu("atck"," "," ","qual");
  k = KEYGet();
  int state;
```

```
  if(k==KEY1)
    state = ATTACK;
  else
    state = QUALIFY;

  AISoccer( goal, state );
}
//---------------------------
```

5.1.3 AI

```
#include "eyebot.h"
#include <stdio.h>
#include "constants.h"
#include "ai.h"

int goal; // the opponents goal
int ourGoal;

extern VWHandle vw;

//---------------------------
void AISoccer(int opponentGoal, int initialState){

  goal = opponentGoal;
  int state;

  if (goal == YELLOW_GOAL) {
    ourGoal = BLUE_GOAL;
  }
  else {
    ourGoal = YELLOW_GOAL;
  }

  printf("Pause\n");
  OSWait(100);
  LCDClear();

  // this is the main state machine
  // each state-function performs the state-behaviour and
  // returns which state the robot should enter
  state = initialState;

  while(1){
    printf("--------\n");
    switch(state){
      case QUALIFY:
        printf("Qualify\n");
        state = AIQualify();
        break;
      case ATTACK:
        printf("Attack\n");
        state = AIAttack();
        break;
      case HASSLE:
        printf("Hassle\n");
        state = AIHassle();
        break;
      case GOTOBALL:
        printf("GoToBALL\n");
        state = AIGoToBall();
        break;
      case GOTOGOAL:
        printf("GoToGoal\n");
        state = AIGoToGoal();
        break;
      case GOTOOPTIMALPOSE:
        printf("GoToOP\n");
        state = AIGoToOptimalPose();
        break;
      case GOTOOWNGOAL:
        printf("GoToOwnGoal\n");
        state = AIGoToOwnGoal();
        break;
```

```
      case FINDBALL:
        printf("FindBall\n");
        state = AIFindBall();
        break;
      case FINDGOAL:
        printf("FindGoal\n");
        state = AIFindGoal();
        break;
      case RETURNHOME:
        printf("ReturnHome\n");
        state = AIReturnHome();
        break;
      case SCOREGOAL:
        printf("ScoreGoal\n");
        state = AIScoreGoal();
        break;
      case RECOVERBALL:
        printf("RecoverBall\n");
        state = AIRecoverBall();
        break;
      default:
        printf("--ERROR--\n");
        state = GOTOBALL;
    }
  }
}
//---------------------------
// initial ball finding improvement
int AIQualify(){
  KinSetSpeed(750);
  KinSpeedyDrive(400);
  return FINDBALL;
}
//---------------------------
// blind very quick inital move
int AIAttack(){
  OSWait(100);
  KinSetSpeed(750);
  KinSpeedyDrive(1200);
  KinSpeedyDrive(-300);
  return FINDBALL;
}
//---------------------------
// circulate around the opponent in tight circles
// sadly not implemented
int AIHassle(){
}
//---------------------------
// search for the ball
// Vi brjar med att snurra max ett varv. p varvet s undersker vi om vi ser
// motstndarmlet, eget ml eller boll. hittar vi ngt av mlen s lagra avstnd.
// hittar vi ej boll s kr mot det ml som r lngst bort lite fint.
// vi avslutar nr vi str vnda mot bollen med ngn vinkel
int AIFindBall(){
  KinSetTurn(MILLI_PI);

  OSWait(50);
  CamTakePicture();
  CamAnalyze();

  int distToOppGoal = 0, distToOwnGoal = 0;
  int angToOppGoal = 0, angToOwnGoal = 0;
  int antalSnurr = 0;
  int snurrVinkel = MILLI_PI/4;
  int snurrTillOppG = 0, snurrTillOwnG = 0;

  int distToWall = 0;
  int snurrTillWall = 0;
  int distToWallTmp = 0;

  while(1) {
    distToWallTmp = CamGetWallDistanceAtColumn(0);
    if (distToWallTmp > distToWall) {
      snurrTillWall = antalSnurr;
```

```
      distToWall = distToWallTmp;
    }

    if (CamBallVisible()) {
      return GOTOBALL;
    }
    else if (AIGoalVisible(goal)) { // the global "goal" is the opponent goa
      distToOppGoal = AIGetGoalDistance(goal);
      snurrTillOppG = antalSnurr;
    }
    else if (AIGoalVisible(ourGoal)) { // goal has been found
      distToOwnGoal = AIGetGoalDistance(ourGoal);
      snurrTillOwnG = antalSnurr;
    }
    else if (2*MILLI_PI < antalSnurr*snurrVinkel) {
      // if Rasdalf has rotated a full 360 degrees
      if (distToOppGoal < distToOwnGoal && distToOppGoal != 0 && distToOwnGoal != 0) {
        // distance to home goal was largest so turn towards it
        KinSetTurn(2*MILLI_PI);
        KinTurn(-(antalSnurr-snurrTillOwnG)*snurrVinkel+angToOwnGoal);
        KinSetTurn(MILLI_PI);
        KinSpeedyDrive(3*distToOwnGoal/4);
      }
      else if (distToOppGoal > distToOwnGoal && distToOppGoal != 0 && distToOwnGoal != 0) {
        // distance to opponent goal was largest so turn towards it
        KinSetTurn(2*MILLI_PI);
        KinTurn(-(antalSnurr-snurrTillOppG)*snurrVinkel+angToOppGoal);
        KinSetTurn(MILLI_PI);
        KinSpeedyDrive(3*distToOwnGoal/4);
      }
      else {
        int minvackravariabel = (int)(-(antalSnurr-snurrTillWall)*snurrVinkel);
        KinSetTurn(2*MILLI_PI);
        KinTurn(minvackravariabel);
        KinSetTurn(MILLI_PI);
        KinDrive(distToWall/3);
      }
      // restart search. reinitialize all variables
      distToOppGoal = 0, distToOwnGoal = 0;
      angToOppGoal = 0, angToOwnGoal = 0;
      antalSnurr = 0;
      snurrTillOppG = 0, snurrTillOwnG = 0;
      distToWall = 0;
      snurrTillWall = 0;
      distToWallTmp = 0;
    }
    KinSetTurn(2*MILLI_PI);
    KinTurn(snurrVinkel);
    KinSetTurn(MILLI_PI);
    antalSnurr++;
    OSWait(50);
    CamTakePicture();
    CamAnalyze();
  }
}
//---------------------------
int AIReturnHome(){
  KinSpeedySetSpeed(-2000,0);
  OSWait(20);
  KinSpeedySetSpeed(-1000,3*MILLI_PI/2);
  OSWait(80);
  KinSpeedySetSpeed(2000,0);
  OSWait(100);
  KinSpeedySetSpeed(1000,0);
  OSWait(60);
  KinSpeedySetSpeed(500,0);
  OSWait(80);
  KinSpeedySetSpeed(-500,0);
  OSWait(50);
  KinSpeedySetSpeed(0,MILLI_PI);
  OSWait(55);
  KinSpeedySetSpeed(2000,0);
  OSWait(80);
  KinSpeedySetSpeed(0,-MILLI_PI);
```

```
  OSWait(10);
  KinSpeedySetSpeed(2000,0);
  OSWait(80);
  KinSpeedySetSpeed(1000,0);
  OSWait(30);
  KinSpeedySetSpeed(0,MILLI_PI);
  OSWait(70);
  KinSpeedySetSpeed(0,0);

  // flush the robots odometric memory. Memory was sadly never used.
  MemFlush();

  return FINDBALL;
}
//-------------------------
int AIScoreGoal(){
  KinSetSpeed(800);

  CamTakePicture();
  CamAnalyze();

  if (!CamBallVisible()){
   return FINDBALL;
  }
  if (!AIGoalVisible(goal)){
   return FINDGOAL;
  }

  if ( abs(AIGetGoalAngle(goal))<MILLI_PI/16 ){
    while( CamHasBall2() && AIGetGoalDistance(goal)>100){
      KinSpeedyDrive( 3*AIGetGoalDistance(goal) );
      OSWait(55);
      CamTakePicture();
      CamAnalyze();
    }
    KinSetSpeed(1500);
    KinSpeedyDrive(200);
    AUBeep();
    return RETURNHOME; // RECOVERBALL
  }
  return GOTOGOAL;
}
//--------------------------
int AIFindGoal(){
  KinSetTurn(MILLI_PI);

  OSWait(55);
  CamTakePicture();
  CamAnalyze();

  int antalSnurr = 0;
  int snurrVinkel = MILLI_PI/4;

  int distToWall = 0;
  int snurrTillWall = 0;
  int distToWallTmp = 0;


  while(1) {
    distToWallTmp = CamGetWallDistanceAtColumn(0);
    if (distToWallTmp > distToWall) {
      snurrTillWall = antalSnurr;
      distToWall = distToWallTmp;
    }

    if (AIGoalVisible(goal)) { // opp goal
      return FINDBALL;
    }
    else if (2*MILLI_PI < antalSnurr*snurrVinkel) {
      // if Rasdalf has rotated a full 360 degrees
      int minvackravariabel = (int)(-(antalSnurr-snurrTillWall)*snurrVinkel);
      KinSetTurn(2*MILLI_PI);
      KinTurn(minvackravariabel);
      KinSetTurn(MILLI_PI);
```

```
      KinDrive(distToWall/3);

      // restart search. reinitialize all variables
      antalSnurr = 0;
      distToWall = 0;
      snurrTillWall = 0;
      distToWallTmp = 0;
    }
    // rotate and look for ball
    KinSetTurn(2*MILLI_PI);
    KinTurn(snurrVinkel);
    KinSetTurn(MILLI_PI);
    antalSnurr++;
    OSWait(55);
    CamTakePicture();
    CamAnalyze();
  }
}
//---------------------------
// go to a goal WITH the ball
int AIGoToGoal(){
  int goalAngle = 0;
  int goalLocalX = 0;
  int opponentVisible = 0;
  int goalDistance = 0;
  KinSetSpeed(1000);
  OSWait(55);
  CamTakePicture();
  CamAnalyze();
  if (!CamBallVisible()){
    return FINDBALL;
  }
  if (!AIGoalVisible(goal)){
    CamTakePicture();
    CamAnalyze();
  }
  KinSetSpeed(100);
  KinSetTurn(MILLI_PI/8);
  if(CamHasBall2() && !AIGoalVisible(goal)){ //letar ml med boll
    //HR SKA DEN SVNGA EFTER ML UTAN ATT TAPPA BOLL
    KinSpeedySetSpeed(500, MILLI_PI/4);
    OSWait(30);
    KinSpeedySetSpeed(0, 0);
    return GOTOBALL;
    OSWait(55);
    CamTakePicture();
    CamAnalyze();
  }

  if(!CamHasBall2()){
    OSWait(200);
    return RECOVERBALL;
  }
  KinSetSpeed(500);
  KinSetTurn(MILLI_PI/16);
  goalAngle=AIGetGoalAngle(goal);
  while(AIGoalVisible(goal) && goalAngle>(MILLI_PI/36)){ //siktar mot ml
    CamTakePicture();
    CamAnalyze();
    if(!CamHasBall2()){
      OSWait(200);
      return RECOVERBALL;
    }
    goalAngle=AIGetGoalAngle(goal);
    KinSpeedyTurn(-goalAngle);
    OSWait(55);
  }
  //NU BORDE VI RRA OSS MOT MLET, MEN TNK OM MOTSTNDAREN KOMMER!!!
  while(1){
    CamTakePicture();
    CamAnalyze();
    goalAngle = AIGetGoalAngle(goal);
    goalDistance = AIGetGoalDistance(goal);
    goalLocalX = AIGetGoalLocalX(goal);
```

```
    opponentVisible = 0; // kr utan opponent frst

    if(!CamHasBall2()){
      return RECOVERBALL;
    }
    if (goalDistance<900 && !opponentVisible && abs(goalLocalX)<=100){
      return SCOREGOAL;
    }
    if(abs(goalLocalX)>100){ // aim at goal
      KinSpeedyTurn(-goalAngle/2);
      OSWait(50);
    }
    if(goalDistance>900){ // go closer to goal
      KinSetSpeed(50);
      KinSpeedyDrive(200);
      OSWait(50);
      return GOTOBALL;
    }

  }

}

//---------------------------
// Om bollen ligger mot vrt ml k mot vrt ml litegrann!
int AIGoToOwnGoal() {
  KinSetSpeed(800);
  KinSetTurn(800);

  OSWait(55);
  CamTakePicture();
  CamAnalyze();

  if (AIGoalVisible(ourGoal) && !CamHasBall2()) {
    KinDrive(AIGetGoalDistance(ourGoal)/2);
  }
  else {
    KinTurn(3*AIGetGoalAngle(ourGoal));
    KinDrive(AIGetGoalDistance(ourGoal)/3);
    KinTurn(-6*AIGetGoalAngle(ourGoal));
    KinDrive(AIGetGoalDistance(ourGoal)/4);
    KinTurn(-6*AIGetGoalAngle(ourGoal));
  }

  return FINDBALL;
}

//---------------------------
int AIGoToBall(){
  OSWait(50);
  CamTakePicture();
  CamAnalyze();

  int ballAngle = CamGetBallAngle(), goalAngle = AIGetGoalAngle(goal);

  if( CamHasBall() ) {
    printf("Har boll!\n");
    return GOTOGOAL;
  }
  if(CamBallVisible() && AIGoalVisible(goal) && abs(ballAngle-goalAngle)<MILLI_PI/36 ) {
    printf("Liten vinkel!\n");
    return GOTOGOAL;
  }
  if( !CamBallVisible() ) return FINDBALL;
  if( AIGoalVisible(goal) ) return GOTOOPTIMALPOSE;
  if(CamBallVisible() && AIGoalVisible(ourGoal)) {
    printf("Kor mot eget!\n");
    return GOTOOWNGOAL;
  }

  // at this point the ball is visible and the goal is unknown
  KinSetSpeed(500);

  KinSetTurn(500);
```

```
  ballAngle = CamGetBallAngle();
  if( abs(ballAngle) > MILLI_PI/32 ){
    KinTurn( -ballAngle );
    return GOTOBALL;
  }
  else {
    KinSetSpeed( constrain( CamGetBallDistance()/2-100 , 50, 1000) );
    KinDrive( CamGetBallDistance()/2 );
    return GOTOBALL;
  }
}
//---------------------------
int AIGoToOptimalPose(){
// Den kr fr mkt nr den str bra!!!!
  OSWait(55);
  CamTakePicture();
  CamAnalyze();

  int goalX,goalY;
  if( AIGoalVisible(goal) ){
    goalX = AIGetGoalLocalX(goal);
    goalY = AIGetGoalLocalY(goal);
  }
  else if( MemGoalKnown() ){
    goalX = MemGetGoalLocalX();
    goalY = MemGetGoalLocalY();
  }
  else {
    // this should not really occur
    return FINDGOAL;
  }

  if( CamHasBall() && abs(AIGetGoalLocalX(goal))<130 && AIGetGoalDistance(goal)<900 ){
    return SCOREGOAL;
  }

  int displace;
  if(goalX<0) displace = 200;
  else displace = -200;
  int ballX = CamGetBallLocalX();
  int ballY = CamGetBallLocalY();
  int vecX = ballX-goalX;
  int vecY = ballY-goalY;
  if(vecY==0) vecY = 1;
  int vecMag = integerSqrt(vecX*vecX + vecY*vecY);
  int targetX = ballX + vecX*200/vecMag;
  int targetY = ballY - abs(vecY)*200/vecMag;
  int targetDist = integerSqrt(targetX*targetX + targetY*targetY);
  targetDist = min(750,min(targetDist,abs(targetX)+abs(targetY)));

  int angle = -milliAtan2(targetX,targetY);
  int finalAngle = -angle - milliAtan2( ballX-targetX , ballY-targetY );

  KinTurn( angle );
  KinDrive( targetDist - 100 );
  KinTurn( finalAngle );

  OSWait(55);
  CamTakePicture();
  CamAnalyze();
  if(CamBallVisible()){
    if(CamHasBall()) {
      return GOTOGOAL;
    }
    else {
      return GOTOBALL;
    }
  }

  printf("op8\n");
  return RECOVERBALL;
}
//---------------------------
```

```
// recover a very recently lost ball
int AIRecoverBall(){
  KinDrive(-200);
  return FINDBALL;
}
//---------------------------
// wrap Cam Goal funcs
int AIGoalVisible(int g) {
  if (g == YELLOW_GOAL) {
    return CamYellowGoalVisible();
  }
  else if (g == BLUE_GOAL) {
    return CamBlueGoalVisible();
  }
  else {
    return -1;
  }
}
//---------------------------
// wrap Cam Goal funcs
int AIGetGoalAngle(int g) {
  if (g == YELLOW_GOAL) {
    return CamGetYellowGoalAngle();
  }
  else if (g == BLUE_GOAL) {
    return CamGetBlueGoalAngle();
  }
  else {
    return -1;
  }
}
//---------------------------
// wrap Cam Goal funcs
int AIGetGoalDistance(int g) {
  if (g == YELLOW_GOAL) {
    return CamGetYellowGoalDistance();
  }
  else if (g == BLUE_GOAL) {
    return CamGetBlueGoalDistance();
  }
  else {
    return -1;
  }
}
//---------------------------
// wrap Cam Goal funcs
int *AIGetGoalLeft(int g) {
  if (g == YELLOW_GOAL) {
    return (int*)CamGetYellowGoalLeft();
  }
  else if (g == BLUE_GOAL) {
    return (int*)CamGetBlueGoalLeft();
  }
  else {
    return NULL;
  }
}
//---------------------------
// wrap Cam Goal funcs
int *AIGetGoalRight(int g) {
  if (g == YELLOW_GOAL) {
    return (int*)CamGetYellowGoalRight();
  }
  else if (g == BLUE_GOAL) {
    return (int*)CamGetBlueGoalRight();
  }
  else {
    return NULL;
  }
}
//---------------------------
// wrap Cam Goal funcs
int AIGetGoalLocalX(int g) {
  if (g == YELLOW_GOAL) {
```

```
    return CamGetYellowGoalLocalX();
  }
  else if (g == BLUE_GOAL) {
    return CamGetBlueGoalLocalX();
  }
  else {
    return -1;
  }
}
//---------------------------
// wrap Cam Goal funcs
int AIGetGoalLocalY(int g) {
  if (g == YELLOW_GOAL) {
    return CamGetYellowGoalLocalY();
  }
  else if (g == BLUE_GOAL) {
    return CamGetBlueGoalLocalY();
  }
  else {
    return -1;
  }
}
```

5.1.4 Camera

```
#include <stdio.h>
#include "eyebot.h"
#include "constants.h"
#include "camera.h"

extern int goal;
int dispMode;
int dispObject;
// sample pixel distribution
int topXres, bottomXres, topYres, bottomYres;

unsigned char imgBuffer[XRES*YRES*3]; // eyebot

// objekt representation
/*
* The objects positions on the screen are stored as a rect defined by LEFT,TOP,RIGHT,BOTTOM;
* A mean of the x & y coordinates of all the pixels
* A counter of the number of pixels classified as the object type
*/
int ballRect[4];
int ballCentroid[2];
int ballPixelCount;
int ballColor[3];

int blueGoalRect[4];
int blueGoalCentroid[2];
int blueGoalLeft[2];
int blueGoalRight[2];
int blueGoalPixelCount;
int blueGoalColor[3];

int yellowGoalRect[4];
int yellowGoalCentroid[2];
int yellowGoalLeft[2];
int yellowGoalRight[2];
int yellowGoalPixelCount;
int yellowGoalColor[3];

int opponentRect[4];
int opponentCentroid[2];
int opponentPixelCount;
int opponentColor[3];

int wallColor[3];
int floorColor[3];

//------------------
void CamInit(){
    CAMInit(NORMAL); // eyebot
```

```
    CamSetDispMode(NOTHING);
    CamSetDispObject(BALL);
    CamSetPixelDistribution(2,2,10,5);
}
//------------------
void CamTakePicture(){
    CAMGetFrameRGB (imgBuffer); // eyebot
}
//------------------
void CamSetPixelDistribution(int topX, int topY, int bottomX, int bottomY){
  topXres = topX;
  bottomXres = bottomX;
  topYres = topY;
  bottomYres = bottomY;
}
//------------------
void CamSetDispMode(int mode){
    dispMode = mode;
}
//------------------
void CamSetDispObject(int obj){
    dispObject = obj;
}
//------------------
void CamCycleDispMode(){
    dispMode++;
    dispMode = dispMode % DISPMODE_RANGE;
    if(dispMode==NOTHING) dispMode = NOTHING+1;
}
//------------------
void CamCycleDispObject(){
    dispObject++;
    dispObject = dispObject % OBJECT_RANGE;
}
//------------------
void clearObjectData(int rect[], int centroid[], int pixelCount[], int color[]){
    rect[LEFT] = XRES;
    rect[TOP] = YRES;
    rect[RIGHT] = 0;
    rect[BOTTOM] = 0;
    centroid[X] = 0;
    centroid[Y] = 0;
    color[RED] = 0;
    color[GREEN] = 0;
    color[BLUE] = 0;
    *pixelCount = 0;
}
//------------------
void normalizeCentroid(int* centroid, int* pixelCount){
    if(*pixelCount!=0){ // undvik division med noll
      centroid[X] /= *pixelCount;
      centroid[Y] /= *pixelCount;
    }
}
//------------------
void adaptObjectData(int x, int y, int pixelSize,  int* rect, int* centroid, int* pixelCount){
    *pixelCount += pixelSize;
    rect[LEFT]   = min(rect[LEFT],x);
    rect[RIGHT]  = max(rect[RIGHT],x);
    rect[TOP]    = min(rect[TOP],y);
    rect[BOTTOM] = max(rect[BOTTOM],y);
    centroid[X] += x*pixelSize;
    centroid[Y] += y*pixelSize;
}
//------------------
int xToRow(int x){
  return (x*63)/XRES;
}
//------------------
int yToCol(int y){
  return 126-(63*y)/YRES;
}
//------------------
void drawPixel(int x, int y, int color){
```

```
    int row = xToRow(x);
    int col = yToCol(y);
    // do not draw outside the screen area
    if(row!=0 && row!=63 && row!=64 && row!=62 && col!=yToCol(0) && col!=yToCol(175))
      LCDSetPixel( row, col, color);
}
//------------------
void drawCrosshair(int x, int y){
    LCDLine( yToCol(y)-3, xToRow(x)  , yToCol(y)+3,xToRow(x)  , 1 );
    LCDLine( yToCol(y)  , xToRow(x)+2, yToCol(y)  ,xToRow(x)-2, 1 );
}
//------------------
void drawRect(int l, int t, int r, int b, int color){
    LCDLine( yToCol(t), xToRow(l), yToCol(b), xToRow(l), color );
    LCDLine( yToCol(b), xToRow(l), yToCol(b), xToRow(r), color );
    LCDLine( yToCol(b), xToRow(r), yToCol(t), xToRow(r), color );
    LCDLine( yToCol(t), xToRow(r), yToCol(t), xToRow(l), color );
}
//------------------
void drawSilouette(int x, int y, int pixelClass){
  if(dispMode==SILOUETTE) drawPixel(x,y,(dispObject==pixelClass));
}
//------------------
void drawVisualization(){
    if( dispMode!=NOTHING ){
      if( dispMode!=SILOUETTE ) LCDArea(yToCol(143)+1,xToRow(0)+1,yToCol(0)-1,xToRow(175)-1,0);
      LCDSetPos (0,0);
      if( dispObject==BALL )         LCDPrintf("ball   ");
      if( dispObject==YELLOW_GOAL )  LCDPrintf("yellow ");
      if( dispObject==BLUE_GOAL )    LCDPrintf("blue   ");
      if( dispObject==OPPONENT )     LCDPrintf("enemy  ");
      if( dispObject==WALL )         LCDPrintf("wall   ");
      if( dispObject==FLOOR )        LCDPrintf("floor  ");
      if( dispObject==UNCLASSIFIED ) LCDPrintf("unknown");

      LCDSetPos (1,0);
      if( dispMode==SILOUETTE ) LCDPrintf("pixels ");
      if( dispMode==CENTROID )  LCDPrintf("centre ");
      if( dispMode==RECT )      LCDPrintf("rect   ");

      if(dispMode==CENTROID){
        if(dispObject==BALL) drawCrosshair( ballCentroid[X], ballCentroid[Y] );
        if(dispObject==YELLOW_GOAL) drawCrosshair( yellowGoalCentroid[X], yellowGoalCentroid[Y] );
        if(dispObject==BLUE_GOAL) drawCrosshair( blueGoalCentroid[X], blueGoalCentroid[Y] );
        if(dispObject==OPPONENT) drawCrosshair( opponentCentroid[X], opponentCentroid[Y] );
      }
      else if(dispMode==RECT){
        if(dispObject==BALL)
          drawRect( ballRect[LEFT],ballRect[TOP],
          ballRect[RIGHT],ballRect[BOTTOM],1 );
        if(dispObject==YELLOW_GOAL)
          drawRect( yellowGoalRect[LEFT],yellowGoalRect[TOP],
          yellowGoalRect[RIGHT],yellowGoalRect[BOTTOM],1 );
        if(dispObject==BLUE_GOAL)
          drawRect( blueGoalRect[LEFT],blueGoalRect[TOP],
          blueGoalRect[RIGHT],blueGoalRect[BOTTOM],1 );
        if(dispObject==OPPONENT)
          drawRect( opponentRect[LEFT],opponentRect[TOP],
          opponentRect[RIGHT],opponentRect[BOTTOM],1 );
        }
      // finally draw a nice border around the display
      drawRect(0,0,175,143,1);
    }
}
//------------------
// performs a vertical scanline scan for the wall/floor intersect
int verticalWallFloorScan(int x){
  int res=2, y, top=2;
  int r,g,b,value, brightValue;

  r = imgBuffer[(top*XRES+x)*3];
  g = imgBuffer[(top*XRES+x)*3+1];
  b = imgBuffer[(top*XRES+x)*3+2];
  brightValue = max( r, max(g,b) );
```

```
  for (y=0; y<120; y+=res ){
    r = imgBuffer[(y*XRES+x)*3];
    g = imgBuffer[(y*XRES+x)*3+1];
    b = imgBuffer[(y*XRES+x)*3+2];
    value = max( r, max(g,b) );
    if( value*175/100 < brightValue )
      break;
    brightValue = value;
  }
  return max( 4, y );
}
//------------------
void CamAnalyze(){

    clearObjectData( ballRect, ballCentroid, &ballPixelCount, ballColor);
    clearObjectData( yellowGoalRect, yellowGoalCentroid, &yellowGoalPixelCount, yellowGoalColor);
    clearObjectData( blueGoalRect, blueGoalCentroid, &blueGoalPixelCount, blueGoalColor);
    clearObjectData( opponentRect, opponentCentroid, &opponentPixelCount, opponentColor);

    int xres = topXres, yres = topYres;
    int x, y, pixelClass, pixelSize, c, i;
    int goalStripTop = 2, goalStripBottom = 5, goalStripRes = 2;

    // look for goal
    for (y=goalStripTop; y<goalStripBottom; y+=goalStripRes ){
      for (x=4; x<XRES-4; x+=goalStripRes ){
          pixelClass = getPixelClass(x,y);
          pixelSize = xres*yres;
          if(dispObject==BLUE_GOAL || dispObject==YELLOW_GOAL) drawSilouette(x,y,pixelClass);
          if( pixelClass == YELLOW_GOAL )
            adaptObjectData( x, y, pixelSize, yellowGoalRect, yellowGoalCentroid, &yellowGoalPixelCount);
          else if( pixelClass == BLUE_GOAL )
            adaptObjectData( x, y, pixelSize, blueGoalRect, blueGoalCentroid, &blueGoalPixelCount);
      }
    }

    // look for ball and opponent
    for (y=goalStripBottom; y<YRES; y+=yres ){
      for (x=4; x<XRES-4; x+=xres ){
        // exclude corners where color samples reside
        if( !(x<38 && y>115) && !(x>130 && y>118) ){
          pixelClass = getPixelClass(x,y);
          pixelSize = xres*yres;
          if(dispObject!=BLUE_GOAL && dispObject!=YELLOW_GOAL) drawSilouette(x,y,pixelClass);
          if( pixelClass == BALL && y > 5 && x > 10)
            adaptObjectData( x, y, pixelSize, ballRect, ballCentroid, &ballPixelCount);
          else if( pixelClass == OPPONENT )
            adaptObjectData( x, y, pixelSize, opponentRect, opponentCentroid, &opponentPixelCount);
        }
      }
      xres = topXres + (bottomXres-topXres)*y/YRES;
      yres = topYres + (bottomYres-topYres)*y/YRES;
    }

    // centroid-vektorn r hittills bara en summa och mste
    // delas med pixelantalet fr att bli ett medelvrde
    normalizeCentroid( ballCentroid, &ballPixelCount );
    normalizeCentroid( yellowGoalCentroid, &yellowGoalPixelCount );
    normalizeCentroid( blueGoalCentroid, &blueGoalPixelCount );
    normalizeCentroid( opponentCentroid, &opponentPixelCount );

    // goal distance calculation

    int verticalScanDisplacement = 5;
    int minimumGoalWidth = 25;

    blueGoalLeft[X]  = max( 0, blueGoalRect[LEFT]-verticalScanDisplacement );
    blueGoalRight[X] = min(XRES-1, blueGoalRect[RIGHT]+verticalScanDisplacement );
    int blueGoalWidth = blueGoalRect[RIGHT]-blueGoalRect[LEFT];
    if( blueGoalWidth < minimumGoalWidth){
      blueGoalLeft[Y]  = blueGoalWidth/3;
      blueGoalRight[Y] = blueGoalWidth/3;
    }
```

```
    else if(blueGoalLeft[X]==0 || blueGoalRight[X]==XRES-1){
      blueGoalLeft[X]  = blueGoalCentroid[X];
      blueGoalLeft[Y]  = blueGoalWidth/3;
      blueGoalRight[X] = blueGoalLeft[X];
      blueGoalRight[Y] = blueGoalLeft[Y];
    }
    else {
      blueGoalLeft[Y]  = verticalWallFloorScan( blueGoalLeft[X] );
      blueGoalRight[Y] = verticalWallFloorScan( blueGoalRight[X] );
    }

    yellowGoalLeft[X]  = max( 0, yellowGoalRect[LEFT]-verticalScanDisplacement );
    yellowGoalRight[X] = min(XRES-1, yellowGoalRect[RIGHT]+verticalScanDisplacement );
    int yellowGoalWidth = yellowGoalRect[RIGHT]-yellowGoalRect[LEFT];
    if( yellowGoalWidth < minimumGoalWidth){
      yellowGoalLeft[Y]  = blueGoalWidth/3;
      yellowGoalRight[Y] = blueGoalWidth/3;
    }
    else if(yellowGoalLeft[X]==0 || yellowGoalRight[X]==XRES-1){
      yellowGoalLeft[X]  = yellowGoalCentroid[X];
      yellowGoalLeft[Y]  = verticalWallFloorScan( yellowGoalCentroid[X] );
      yellowGoalRight[X] = yellowGoalLeft[X];
      yellowGoalRight[Y] = yellowGoalLeft[Y];
    }
    else {
      yellowGoalLeft[Y]  = verticalWallFloorScan( yellowGoalLeft[X] );
      yellowGoalRight[Y] = verticalWallFloorScan( yellowGoalRight[X] );
    }

    // memory interaction

    if(AIGoalVisible(goal))
      MemMemorizeGoal( AIGetGoalLocalX(goal), AIGetGoalLocalY(goal) );

    drawVisualization();
}
//------------------
int getPixelClass (int x, int y){
  int r = imgBuffer[(y*XRES+x)*3];
  int g = imgBuffer[(y*XRES+x)*3+1];
  int b = imgBuffer[(y*XRES+x)*3+2];

  int imax = max(r, max(g,b));
  int imin = min(r, min(g,b));

  int value = imax*100/255;
  int hue = 0;
  int sat = 0;

  if (imax == imin) {
    return UNCLASSIFIED;
  }
  if (imax == r && g >= b) {
    hue = 60*(g-b)/(imax-imin);
  }
  else if (imax == r && g < b) {
    hue = 60*(g-b)/(imax-imin)+360;
  }
  else if (imax == g) {
    hue = 60*(b-r)/(imax-imin)+120;
  }
  else if (imax == b) {
    hue = 60*(r-g)/(imax-imin)+240;
  }
  else {
    return UNCLASSIFIED;
  }
  if (imax != 0) {
    sat = 100-100*imin/imax;
  }

  if( hue > 0 && hue < 20 && sat > 80) return BALL;
  else if( hue > 21 && hue < 65 && sat > 86) return YELLOW_GOAL;
  else if( hue > 200 && hue < 245 ) return BLUE_GOAL;
```

```
  else if( ( (hue > 25 && hue < 85 && sat < 85) || (hue == 0 && sat == 0) ) && value > 80 )
    return WALL; // lite specialkod ;)
  else if( (hue >= 0 && hue < 11 && sat < 80) || (hue > 354 && hue < 361))
    return OPPONENT;
  else if( hue > 60 && hue < 140 && sat < 55 ) return FLOOR;

}
//==========================================
//==========================================
int CamGetWallDistanceStraightAhead(){
  return CamGetWallDistanceAtColumn(0);
}
int CamGetWallDistanceAtColumn(int x){
  x += XRES/2;
  x = min( XRES-1, max(0, x) ); // no out of bound errors
  int y = verticalWallFloorScan(x);
  int xL = TransformScreenToLocalFrameX(x,y);
  int yL = TransformScreenToLocalFrameY(x,y);
  return integerSqrt(xL*xL+yL*yL);
}
//------------------
int CamHasBall(){
    int x,y=YRES-2,res=2,ballCounter=0;
//    for(y=YRES-4;x<YRES-2;x+=res){
    for(y=YRES-4;y<=YRES-2;y+=res){
      for(x=38;x<113;x+=res){
        if( getPixelClass(x,y)==BALL ) ballCounter++;
      }
    }
    return ballCounter>4;
    return (abs(CamGetBallLocalX())<50) && (CamGetBallLocalY()<200);
}

int CamHasBall2(int goal) {
  if (!CamBallVisible()) {
    return 0;
  }
  if (CamGetBallLocalY()/3+abs(CamGetBallLocalX()) < 100) {
    return 1;
  }
  else if ((CamBallVisible() && AIGoalVisible(goal)) ||
  abs(CamGetBallAngle()-AIGetGoalAngle(goal))<MILLI_PI/36 )
    return 1;
  else {
    return 0;
  }
}

int CamBallVisible(){
    return (ballPixelCount > 1) && (ballPixelCount < 1000);
}
int CamGetBallAngle(){
    return milliAtan( CamGetBallLocalX()*1000/CamGetBallLocalY() );
}
int CamGetBallDistance(){
  int x = CamGetBallLocalX();
  int y = CamGetBallLocalY();
  return integerSqrt(x*x+y*y);
}
int CamGetBallLocalX(){
    return TransformScreenToLocalFrameX(ballCentroid[X],ballCentroid[Y]);
}
int CamGetBallLocalY(){
    return TransformScreenToLocalFrameY(ballCentroid[X],ballCentroid[Y]);
}
//------------------
int CamBlueGoalVisible(){
    return blueGoalPixelCount>80;
}
int CamGetBlueGoalAngle(){
    return milliAtan( CamGetBlueGoalLocalX()*1000/CamGetBlueGoalLocalY() );
}
int CamGetBlueGoalDistance(){
  int x = CamGetBlueGoalLocalX();
```

```
  int y = CamGetBlueGoalLocalY();
  return integerSqrt(x*x+y*y);
}
int CamGetBlueGoalLocalX(){
    int x1 = TransformScreenToLocalFrameX(blueGoalLeft[X],blueGoalLeft[Y]);
    int x2 = TransformScreenToLocalFrameX(blueGoalRight[X],blueGoalRight[Y]);
    return (x1+x2)/2;
}
int CamGetBlueGoalLocalY(){
    int y1 = TransformScreenToLocalFrameY(blueGoalLeft[X],blueGoalLeft[Y]);
    int y2 = TransformScreenToLocalFrameY(blueGoalRight[X],blueGoalRight[Y]);
    return (y1+y2)/2;
}
int *CamGetBlueGoalLeft(){
  return blueGoalLeft;
}
int *CamGetBlueGoalRight(){
  return blueGoalRight;
}

//------------------
int CamYellowGoalVisible(){
    return yellowGoalPixelCount>80;
}
int CamGetYellowGoalAngle(){
    return milliAtan( CamGetYellowGoalLocalX()*1000/CamGetYellowGoalLocalY() );
}
int CamGetYellowGoalDistance(){
  int x = CamGetYellowGoalLocalX();
  int y = CamGetYellowGoalLocalY();
  return integerSqrt(x*x+y*y);
}
int CamGetYellowGoalLocalX(){
    int x1 = TransformScreenToLocalFrameX(yellowGoalLeft[X],yellowGoalLeft[Y]);
    int x2 = TransformScreenToLocalFrameX(yellowGoalRight[X],yellowGoalRight[Y]);
    return (x1+x2)/2;
}
int CamGetYellowGoalLocalY(){
    int y1 = TransformScreenToLocalFrameY(yellowGoalLeft[X],yellowGoalLeft[Y]);
    int y2 = TransformScreenToLocalFrameY(yellowGoalRight[X],yellowGoalRight[Y]);
    return (y1+y2)/2;
}
int *CamGetYellowGoalLeft(){
  return yellowGoalLeft;
}
int *CamGetYellowGoalRight(){
  return yellowGoalRight;
}

//------------------
int CamOpponentVisible(){
    return opponentPixelCount>6;
}
int CamGetOpponentAngle(){
    return TransformScreenToLocalFrameX(opponentCentroid[X],500);
}
int CamGetOpponentDistance(){
    return 500;
}
int CamGetOpponentLocalX(){
    return TransformScreenToLocalFrameX(opponentCentroid[X],500);
}
int CamGetOpponentLocalY(){
    return 500;
}
//------------------
```

5.1.5 Constants

```
#ifndef CONSTANTS_H
#define CONSTANTS_H

#define TRUE 1
#define FALSE 0
```

```
#define PI 3.14159265
#define MILLI_PI 3142

// the states
#define ATTACK 0
#define HASSLE 1
#define GOTOBALL 2
#define GOTOOPTIMALPOSE 3
#define FINDGOAL 4
#define RETURNHOME 5
#define SCOREGOAL 6
#define GOTOPOSE 7
#define GOTOGOAL 8
#define RECOVERBALL 9
#define FINDBALL 10
#define QUALIFY 11
#define GOTOOWNGOAL 12

#define X 0
#define Y 1
#define ANGLE 2
#define UNCERT_X 3
#define UNCERT_Y 4
#define UNCERT_ANGLE 5

#define RED 0
#define GREEN 1
#define BLUE 2

#define LEFT 0
#define TOP 1
#define RIGHT 2
#define BOTTOM 3
#define XRES 176
#define YRES 144

// vision diagnostics program constants
#define OBJECT_RANGE 7

#define UNCLASSIFIED 0
#define BALL 1
#define YELLOW_GOAL 2
#define BLUE_GOAL 3
#define WALL 4
#define FLOOR 5
#define OPPONENT 6

#define DISPMODE_RANGE 4

#define NOTHING 0
#define SILOUETTE 1
#define CENTROID 2
#define RECT 3
#define FIELD_WIDTH 1186
#define FIELD_HEIGHT 2356
#define FIELD_HALF_WIDTH 593

#endif
```

5.1.6 Kinematics

```
#include <stdio.h>
#include "eyebot.h"
#include "constants.h"
#include "kinematics.h"

/*
* the Kinematics module works with integer milliRadians and milliMeters
*/

float speed;
float turn;
VWHandle vw;
//------------------
```

```
void KinInit(){
  speed = 0.5;
  turn = 0.5;

  vw = VWInit(VW_DRIVE,1);
  VWStartControl(vw,7,0.3,7,0.1);
}
//------------------
VWHandle KinGetVWHandle(){
  return vw;
}
//------------------
void KinSetSpeed(int speedParam){
  speed = speedParam*0.001;
}
//------------------
void KinSetTurn(int turnParam){
  turn = turnParam*0.001;
}
//------------------
void KinDrive(int distance){
  int cnt = 0;
  VWDriveStraight(vw,(float)distance*0.001,speed);
  // stall detection
  while(VWDriveDone(vw) == 0 && cnt < 26 && VWStalled(vw) == 0) {
    OSWait(20);
    cnt++;
  }
}
//------------------
void KinTurn(int angle){  // milliradianer pos. -> counter-clockwise
  int cnt = 0;
  VWDriveTurn(vw,(float)angle*0.001,turn);
  // stall detection
  while(VWDriveDone(vw) == 0 && cnt < 26 && VWStalled(vw) == 0) {
    OSWait(20);
    cnt++;
  }
}
//------------------
void KinSpeedyDrive(int distance){ // milli
  if(distance<0)
    VWSetSpeed(vw,-speed,0);
  if(distance>=0)
    VWSetSpeed(vw,speed,0);
  OSWait(abs((float)distance*0.001/speed*100));
  VWSetSpeed(vw,0,0);
}
//------------------
void KinSpeedyTurn(int angle){  // milliradianer pos. -> counter-clockwise
  if (angle<0)
    VWSetSpeed(vw,0,-turn);
  if (angle>=0)
    VWSetSpeed(vw,0,turn);
  OSWait(abs((int)(100*(float)angle*0.001/turn)));
  VWSetSpeed(vw,0,0);
}

//------------------
void KinSpeedySetSpeed(int s, int a) {
  VWSetSpeed(vw,(float)s*0.001,(float)a*0.001);
}
```

5.1.7 Memory

```
#include "eyebot.h"
#include <stdio.h>
#include "constants.h"
#include "memory.h"

int goalPosition[2];
int goalKnown;

//-----------------
```

```
void MemInit(){
  MemFlush();
}
//-----------------
void MemFlush(){
  goalPosition[X] = 0;
  goalPosition[Y] = 0;
  goalKnown = FALSE;
}
//-----------------
void MemMemorizeGoal(int x, int y){
  goalPosition[X] = x;
  goalPosition[Y] = y;
  VWSetPosition( KinGetVWHandle(), 0.0, 0.0, 0.0 );
  goalKnown = TRUE;
}
//-----------------
int MemGoalKnown(){
  return goalKnown;
}
//-----------------
int MemGetGoalLocalX(){
  PositionType vwRobotPosition;
  VWGetPosition( KinGetVWHandle(), &vwRobotPosition );

  int rx = (int)(1000*vwRobotPosition.x);
  int ry = (int)(1000*vwRobotPosition.y);
  int rphi = (int)(1000*vwRobotPosition.phi);

  int gx = goalPosition[X] - rx;
  int gy = goalPosition[Y] - ry;

  goalPosition[X] = ( milliCos(-rphi)*gx - milliSin(-rphi)*gy ) /1000;
  goalPosition[Y] = ( milliSin(-rphi)*gx + milliCos(-rphi)*gy ) /1000;

  return goalPosition[X];
}
//-----------------
int MemGetGoalLocalY(){
  PositionType vwRobotPosition;
  VWGetPosition( KinGetVWHandle(), &vwRobotPosition );

  int rx = (int)(1000*vwRobotPosition.x);
  int ry = (int)(1000*vwRobotPosition.y);
  int rphi = (int)(1000*vwRobotPosition.phi);

  int gx = goalPosition[X] - rx;
  int gy = goalPosition[Y] - ry;

  goalPosition[X] = ( milliCos(-rphi)*gx - milliSin(-rphi)*gy ) /1000;
  goalPosition[Y] = ( milliSin(-rphi)*gx + milliCos(-rphi)*gy ) /1000;

  return goalPosition[Y];
}
//-----------------
int MemGetGoalAngle(){
  return -milliAtan2(goalPosition[Y],goalPosition[X]);
}
//-----------------
```

5.1.8 Fixed point math

```
#include <stdio.h>
#include "eyebot.h"
#include "constants.h"
#include "millimath.h"

// milliCosLUT is a maps x in centiRadians to 1000*cos(x)
int milliCosLength = 158;
int milliCosLUT[] = {1000,1000,1000,1000,999,999,998,998,997,996,995,994,993,
  992,990,989,987,986,984,982,980,978,976,974,971,969,966,964,961,958,955,952,
  949,946,943,939,936,932,929,925,921,917,913,909,905,900,896,892,887,882,878,
  873,868,863,858,853,847,842,836,831,825,820,814,808,802,796,790,784,778,771,
  765,758,752,745,738,732,725,718,711,704,697,689,682,675,667,660,652,645,637,
```

```
  629,622,614,606,598,590,582,574,565,557,549,540,532,523,515,506,498,489,480,
  471,462,454,445,436,427,418,408,399,390,381,372,362,353,344,334,325,315,306,
  296,287,277,267,258,248,238,229,219,209,199,190,180,170,160,150,140,130,121,
  111,101,91,81,71,61,51,41,31,21,11,1};
int milliAtanLUT[] = {0,10,20,30,40,50,60,70,80,90,100,110,119,129,139,149,159,
  168,178,188,197,207,217,226,236,245,254,264,273,282,291,301,310,319,328,337,
  346,354,363,372,381,389,398,406,415,423,431,439,448,456,464,472,480,487,495,
  503,510,518,526,533,540,548,555,562,569,576,583,590,597,604,611,617,624,631,
  637,644,650,656,662,669,675,681,687,693,699,704,710,716,722,727,733,738,744,
  749,754,760,765,770,775,780};
//------------------
int milliCos(int x){
  x = abs( x/10 );
  while( x>314 )
    x-=3142;

  if(x<158 && x>=0)
    return milliCosLUT[ x ];
  else if(x>=158 && x<315)
    return -milliCosLUT[ x-158 ];
  else return 0;
}
//------------------
int milliSin(int x){
  return milliCos(x+MILLI_PI/2);
}
//------------------
int milliTan(int x){
  return milliSin(x)*1000/milliCos(x);
}
//------------------
int milliAtan(int x){
  if(x<0)
    return -milliAtan(abs(x));
  else if(x<1000)
    return milliAtanLUT[ x/10 ];
  else
    return MILLI_PI/2 - milliAtanLUT[ 100000/x -1 ];
}
//------------------
int milliAtan2(int y, int x){
  if(y<0) return -milliAtan2(-y,x);
  if(x<0) return MILLI_PI - milliAtan(-y*1000/x);
  if(x>0) return milliAtan(y*1000/x);
  return MILLI_PI/2;
}
//------------------
int integerSqrt(int x){
  int i;
  for(i=0;i<3000;i+=300)
    if(x < i*i)
      for(i-=270;i<3000;i+=30)
        if(x < i*i)
          return i;
  return 3000;
}
//------------------
int min(int a, int b){
  if(a>b) return b;
  else return a;
}
//------------------
int max(int a, int b){
  if(a<b) return b;
  else return a;
}
//------------------
int constrain(int x, int lower, int upper){
  if(x<lower) return lower;
  if(x>upper) return upper;
  return x;
}
//------------------
int abs(int x){
```

```
  if(x<0) return -x;
  else return x;
}
```

5.1.9 Radar

```
#include "eyebot.h"
#include <stdio.h>
#include "constants.h"

extern int goal;

int robot[6];
int ball[6];
int blueGoal[6];
int yellowGoal[6];
int opponent[6];
int optimalPose[6];
int memory[6];

int iconNone[]  = {0,0,0,0,0,  0,0,0,0,0,  0,0,0,0,0,  0,0,0,0,0,  0,0,0,0,0};
int iconB[]     = {0,1,1,1,0,  0,1,0,0,1,  0,1,1,1,0,  0,1,0,0,1,  0,1,1,1,0};
int iconY[]     = {1,0,0,0,1,  0,1,0,1,0,  0,0,1,0,0,  0,0,1,0,0,  0,0,1,0,0};
int iconO[]     = {0,1,1,1,0,  1,0,0,0,1,  1,0,0,0,1,  1,0,0,0,1,  0,1,1,1,0};
int iconCross[] = {1,0,1,0,1,  0,0,1,0,0,  1,1,1,1,1,  0,0,1,0,0,  1,0,1,0,1};
int iconBlob[]  = {0,1,1,1,0,  1,1,1,1,1,  1,1,1,1,1,  1,1,1,1,1,  0,1,1,1,0};
int iconMark[]  = {0,0,0,0,0,  0,0,1,0,0,  0,1,1,1,0,  0,0,1,0,0,  0,0,0,0,0};
int iconSquare[]= {1,1,1,1,1,  1,0,0,0,1,  1,0,0,0,1,  1,0,0,0,1,  1,1,1,1,1};

extern yellowGoalPixelCount,blueGoalPixelCount;

//-----------------
int sx(int x){
  return 32-x*32/500;
}
//-----------------
int sy(int y){
  return 127-y*127/1500;
}
//-----------------
void drawIcon(int* icon, int x, int y){
  int i,j;
  for(i=0;i<5;i++)
    for(j=0;j<5;j++)
      LCDSetPixel(sx(x)+i-2,sy(y)-j+2,icon[i+5*j]);
}
//-----------------
void clearIcon(int x, int y){
  drawIcon( iconNone, x, y );
}
//-----------------
void computeOptimalPose(){
  int goalX,goalY;
  if( AIGoalVisible(goal) ){
    goalX = AIGetGoalLocalX(goal);
    goalY = AIGetGoalLocalY(goal);
  }
  else if( MemGoalKnown() ){
    goalX = MemGetGoalLocalX();
    goalY = MemGetGoalLocalY();
  }
  else {
    return;
  }
  int displace;
  if(goalX<0) displace = 200;
  else displace = -200;
  int ballX = CamGetBallLocalX();
  int ballY = CamGetBallLocalY();
  int vecX = ballX-goalX;
  int vecY = ballY-goalY;
  int vecMag = integerSqrt(vecX*vecX + vecY*vecY);
  optimalPose[X] = ballX + vecX*200/vecMag;
  optimalPose[Y] = ballY - abs(vecY)*200/vecMag;
```

```
}
//-----------------
int main(void){

  LCDClear();
  int k;
  CamInit();
  MemInit();
  KinInit();

  robot[X] = 0;
  robot[Y] = 50;

  int wallPointsX[10];
  int wallPointsY[10];

  while(1) {

    k = 4711;
    k = KEYRead();
    if(k==KEY1){
      KinDrive(200);
      KinTurn(MILLI_PI/8);
    }

    CamTakePicture();
    CamAnalyze();

    clearIcon( robot[X], robot[Y] );
    clearIcon( ball[X], ball[Y] );
    clearIcon( blueGoal[X], blueGoal[Y] );
    clearIcon( yellowGoal[X], yellowGoal[Y] );
    clearIcon( optimalPose[X], optimalPose[Y] );
    clearIcon( memory[X], memory[Y] );
    int j;
    for(j=0;j<10;j++)
     clearIcon( wallPointsX[j], wallPointsY[j] );

    //--update memory--
    if (CamBallVisible()){
      ball[X] = CamGetBallLocalX();
      ball[Y] = CamGetBallLocalY();
    }
    if (CamBlueGoalVisible()){
      blueGoal[X] = CamGetBlueGoalLocalX();
      blueGoal[Y] = CamGetBlueGoalLocalY();
    }
    if (CamYellowGoalVisible()){
      yellowGoal[X] = CamGetYellowGoalLocalX();
      yellowGoal[Y] = CamGetYellowGoalLocalY();
    }
    if (MemGoalKnown()){
      memory[X] = MemGetGoalLocalX();
      memory[Y] = MemGetGoalLocalY();
    }

    computeOptimalPose();

    //--draw local frame map--
    drawIcon( iconBlob, robot[X], robot[Y] );
    if (CamBallVisible())
       drawIcon( iconO, ball[X], ball[Y] );
    if (CamBlueGoalVisible())
      drawIcon( iconB, blueGoal[X], blueGoal[Y] );
    if (MemGoalKnown())
      drawIcon( iconSquare, memory[X], memory[Y] );
    if (CamYellowGoalVisible()){
      drawIcon( iconY, yellowGoal[X], yellowGoal[Y] );
      drawIcon( iconCross, optimalPose[X], optimalPose[Y] );
    }

    //--draw detected wall--
    int x,y;
    for(j=0;j<10;j++){
```

```
      x = 2+17*j;
      y = verticalWallFloorScan( x );
      wallPointsX[j] = TransformScreenToLocalFrameX(x,y);
      wallPointsY[j] = TransformScreenToLocalFrameY(x,y);
      drawIcon( iconMark, wallPointsX[j], wallPointsY[j]);
    }

    //--draw local frame grid--
    LCDLine(sy(robot[Y]),sx(robot[X]),sy(600),sx(-480),1);
    LCDLine(sy(robot[Y]),sx(robot[X]),sy(600),sx( 480),1);
    int grid[] = {1500,1250,1000,750,500,250};
    int dash[] = {25,10,25,10,25,10};
    int i;
    for(i=0;i<6;i++)
      LCDLine(sy(grid[i]),sx(-dash[i]),sy(grid[i]),sx(dash[i]),1);

    LCDSetPos(0,0);
    LCDPutString( "   \n" );
    LCDPutString( "   \n" );

    LCDSetPos(0,0);
    LCDPutInt( yellowGoalPixelCount );
    LCDPutString( "\n" );
    LCDPutInt( blueGoalPixelCount );
  }
}
//-----------------
```

5.1.10 Transform

```
#include <stdio.h>
#include "eyebot.h"
#include "constants.h"
#include "transform.h"

// localYLUT maps every other screen Y coordinate to a y-coord in the local frame
int localYLUT[] = {3163,2541,2122,1822,1595,1418,1276,1159,1061,979,907,845,791,
  743,700,662,627,595,567,540,516,494,473,453,435,419,403,388,374,361,348,337,
  325,315,305,295,286,277,268,260,252,245,237,230,224,217,211,205,199,193,187,
  182,177,172,167,162,157,152,148,144,139,135,131,127,123,119,115,111,108,104,
  100,97};

// kiloTanXLUT maps every other screen X to 1024 * the corresponding tan(angle)
int kiloTanXLUT[] = {-776,-752,-728,-705,-683,-661,-639,-618,-597,-577,-557,
  -537,-518,-499,-480,-461,-443,-425,-407,-390,-372,-355,-338,-321,-305,-288,
  -272,-256,-240,-224,-208,-192,-176,-161,-145,-130,-114,-99,-84,-68,-53,-38,
  -23,-8,8,23,38,53,68,84,99,114,130,145,161,176,192,208,224,240,256,272,288,
  305,321,338,355,372,390,407,425,443,461,480,499,518,537,557,577,597,618,639,
  661,683,705,728,752,776,-794,-770,-746,-722,-699,-677,-655,-633,-612,-591,
  -570,-550,-530,-511,-491,-472,-454,-435,-417,-399,-381,-364,-346,-329,-312,
  -295,-278,-262,-245,-229,-213,-196,-180,-164,-149,-133,-117,-101,-86,-70,-54,
  -39,-23,-8,8,23,39,54,70,86,101,117,133,149,164,180,196,213,229,245,262,278,
  295,312,329,346,364,381,399,417,435,454,472,491,511,530,550,570,591,612,633,
  655,677,699,722,746,770,794};

//------------------
int TransformScreenToLocalFrameX(int screenX, int screenY){
  int localY = localYLUT[ screenY/2 ];
  int localX = localY*kiloTanXLUT[ screenX/2 ]/1024;
  return localX;
}
//------------------
int TransformScreenToLocalFrameY(int screenX, int screenY){
  return localYLUT[ screenY/2 ];
}
//------------------
```

5.2 Diagnostics and utility software

5.2.1 Coordinate transform calibration

```
function calibrate
% MATLAB coordinate transformation calibration
clf

width = 1186;
height = 2356;
row1 = height*3/4;
row2 = height/2;
row3 = height/4;
row4 = 200;
col1 = 254;
col2 = width/2;
col3 = width-254;

ballPos = ones(4,12);

ballScreen = [55 80 107 43 80 119 11 81 150 34 86 141;
    6 6 6 13 13 13 33 33 33 92 92 92];

%-- lift all balls 20mm above the field --
ballPos(3,:) = 20;

%-- position the first nine balls --
ballPos(1,1:3) = col1;
ballPos(1,4:6) = col2;
ballPos(1,7:9) = col3;
ballPos(2,[1 4 7]) = row1;
ballPos(2,[2 5 8]) = row2;
ballPos(2,[3 6 9]) = row3;

ballPos(3,:) = 0.1*ballPos(2,:);

%-- position the three closest balls --
ballPos(1,10:12) = width/2 + [-100 0 100];
ballPos(2,10:12) = 200;

topAngle = 0.06;
bottomAngle = pi*0.35;
fovOffset = 0.045;
fovWidth = pi*0.42;
fovStart = -fovWidth/2;

%-- screen to world transform --
balls = ballScreen;
for i=1:12
  balls(2,i) = 190/tan( topAngle + bottomAngle*balls(2,i)/144 );
  angle = fovStart + fovWidth*(fovOffset + balls(1,i)/176);
  balls(1,i) = width/2 + balls(2,i)*tan( angle );
end

subplot(2,2,[1 3])
hold on
plot(ballPos(1,:),ballPos(2,:),'xr')
plot(tmp(1,:),tmp(2,:),'ob')
title 'red: true, blue: calculated'
axis([0 width 0 height]);

subplot(2,2,2)
hold on
plot(ballScreen(1,:),144-ballScreen(2,:),'ob')
title 'camera image'
axis([0 176 0 144]);
```

5.2.2 Ball finding

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

#include "ballFinder.h"

#define COLORTHRESHOLD 50
#define XRES 176
#define YRES 144

unsigned char* rgbdata;
char* bitmap;

int findBall(unsigned char* data, int subLevels, int* leftBorder, int*
 rightBorder, int* topBorder, int* bottomBorder)
{
// bsta vrden tills nu
int bestLb = -1;
int bestRb = -1;
int bestTb = -1;
int bestBb = -1;

rgbdata = data;
bitmap = (char*)calloc(XRES*YRES, sizeof(char));

for (int x = 0; x < XRES; x+=subLevels)
for (int y = 20; y < YRES; y+=subLevels)
examineArea(x,y, &bestLb, &bestRb, &bestTb, &bestBb);

free (bitmap);
if (bestLb == -1 || bestRb == -1 || bestTb == -1 || bestBb == -1) // om inget omrde hittades
return 0;
*leftBorder = bestLb;
*rightBorder = bestRb;
*topBorder = bestTb;
*bottomBorder = bestBb;
return 1;
}

int orangePixel (unsigned char r, unsigned char g, unsigned char b)
{
if ( r - COLORTHRESHOLD > g && r - COLORTHRESHOLD > b )
return 1;
return 0;
}

void checkNext(int x, int y, char* bitmap, int* leftBorder,
   int* rightBorder, int* topBorder, int* bottomBorder)
{
if ( x < XRES && x >= 0 && y < YRES && y >= 0 && bitmap[y*XRES+x] == 0)
{
bitmap[y*XRES+x] = 1; // kom inte tillbaka
if ( orangePixel( rgbdata[(y*XRES+x)*3], rgbdata[((y*XRES+x)*3)+1], rgbdata[((y*XRES+x)*3)+2] ) )
{
// kolla om punkten r mer extrem n tidigare kanter
if ( x > *rightBorder )
*rightBorder = x;
if ( x < *leftBorder )
*leftBorder = x;
if ( y > *bottomBorder )
*bottomBorder = y;
if ( y < *topBorder )
*topBorder = y;

// kolla alla nya nrliggande punkter
checkNext (x+1, y, bitmap, leftBorder, rightBorder, topBorder, bottomBorder); // hger
checkNext (x-1, y, bitmap, leftBorder, rightBorder, topBorder, bottomBorder); // vnster
checkNext (x, y-1, bitmap, leftBorder, rightBorder, topBorder, bottomBorder); // upp
checkNext (x, y+1, bitmap, leftBorder, rightBorder, topBorder, bottomBorder); // ner
}
}
}
```

```
void examineArea(int x, int y, int* bestLb, int* bestRb, int* bestTb, int* bestBb)
{
int leftBorder = x;
int rightBorder = x;
int topBorder = y;
int bottomBorder = y;

checkNext (x, y, bitmap, &leftBorder, &rightBorder, &topBorder, &bottomBorder);

// ta bort vldigt sm orangea omrden
if ( rightBorder - leftBorder < 3 ||
 bottomBorder - topBorder < 3 )
return;

// ta bort vldigt stora orangea omrden
if ( rightBorder - leftBorder > 40 ||
 bottomBorder - topBorder > 40 )
return;

// ta bort orangea omrden med konstiga proportioner
if ( (rightBorder - leftBorder) < 0.6*(bottomBorder - topBorder) ||
 (bottomBorder - topBorder) < 0.6*(rightBorder - leftBorder))
return;

// om detta omrde r "bttre" (strre) n tidigare funna, s betrakta detta som bollen
if ( (rightBorder - leftBorder) > (*bestRb - *bestLb) &&
 (bottomBorder - topBorder) > (*bestBb - *bestTb) )
{
//printf("Area passed tests..\n");
*bestLb = leftBorder;
*bestRb = rightBorder;
*bestTb = topBorder;
*bestBb = bottomBorder;
}
}
```

5.2.3 BMP parser

```
#include "camera.h"

#include <stdio.h>
#include <windows.h>
#include <iostream>
#include <fstream>

using namespace std;

void main ()
{
// bitmap header
BITMAPFILEHEADER file_header;
BITMAPINFOHEADER info_header;
RGBTRIPLE rgb_triple;

ifstream in ("test.bmp", ios::in | ios::binary);

// kolla om filen ppnades
if (!in)
{
cout << "Kunde inte ppna filen" << endl;
system("pause");
return;
}

// ls headers
in.read( (char *)&file_header, sizeof(file_header) );
in.read( (char *)&info_header, sizeof(info_header) );

// skapa array fr bilddata
unsigned char * data;
data = new unsigned char[info_header.biHeight * info_header.biWidth * 3];

cout << "Skapar bildarray, " << info_header.biWidth << "x" << info_header.biHeight << endl;
```

```
for (int y = info_header.biHeight-1; y > -1; --y) // BMP lagras upp och ner
{
for (int x = 0; x < info_header.biWidth; ++x)
{
in.read( (char *)&rgb_triple, sizeof(rgb_triple) );
data[(y*info_header.biWidth+x)*3+0] = rgb_triple.rgbtRed;
data[(y*info_header.biWidth+x)*3+1] = rgb_triple.rgbtGreen;
data[(y*info_header.biWidth+x)*3+2] = rgb_triple.rgbtBlue;
}
in.ignore( info_header.biWidth % 4 ); // ignorera padding
}
in.close();

// Analys
CamTakePicture(data);
CamAnalyze(2);

delete [] data;
system("pause");
}
```

5.2.4 PPM reader

```
// Usage instructions:
// ppmread infile.ppm > outfile.ppm
// that is, it tries to read param1 as the file to be analyzed
// and writes the outfile to STDOUT
// Dont be stupid and overwrite files you want to keep! ;)

#include <stdio.h>
#define LINESIZE 100000
#define XRES 176
#define YRES 144

#define UNCLASSIFIED 0
#define BALL 1
#define YELLOW_GOAL 2
#define BLUE_GOAL 3
#define WALL 4
#define FLOOR 5
#define OPPONENT 6

unsigned char *imgBuffer;  // ppmread

void CamAnalyze(){

  int x, y, r, g, b, pixelClass;

  // Classify each pixel and print it
  for (y=0; y<YRES; y++ ){
    for (x=0; x<XRES; x++ ){
      r = imgBuffer[(y*XRES+x)*3];
      g = imgBuffer[(y*XRES+x)*3+1];
      b = imgBuffer[(y*XRES+x)*3+2];
      pixelClass = getPixelClass(r,g,b);

      if( pixelClass == BALL ) {
        printf("\xff\x01\x01");
      }
      else if( pixelClass == YELLOW_GOAL ) {
        printf("\xff\xff\x01");
      }
      else if( pixelClass == BLUE_GOAL ) {
        printf("\x01\x01\xff");
      }
      else if( pixelClass == OPPONENT ){
        printf("\x80\x01\xff");
      }
      else if( pixelClass == WALL ) {
        printf("\xd4\xd4\xd4");
      }
      else if( pixelClass == FLOOR ){
        printf("\x01\x80\x40");
      }
```

```
      else {
        printf("\xff\xff\xff");
      }
    }
  }
}
//------------------
int getPixelClass (int r, int g, int b){
  int imax = max(r, max(g,b));
  int imin = min(r, min(g,b));

  int value = imax*100/255;
  int hue = 0;
  int sat = 0;

  if (imax == imin) {
    return UNCLASSIFIED;
  }
  if (imax == r && g >= b) {
    hue = 60*(g-b)/(imax-imin);
  }
  else if (imax == r && g < b) {
    hue = 60*(g-b)/(imax-imin)+360;
  }
  else if (imax == g) {
    hue = 60*(b-r)/(imax-imin)+120;
  }
  else if (imax == b) {
    hue = 60*(r-g)/(imax-imin)+240;
  }
  else {
    return UNCLASSIFIED;
  }
  if (imax != 0) {
    sat = 100-100*imin/imax;
  }

  if( hue > 0 && hue < 20 && sat > 80) return BALL;
  else if( hue > 21 && hue < 65 && sat > 86) return YELLOW_GOAL;
  else if( hue > 200 && hue < 245) return BLUE_GOAL;
  else if( ((hue > 25 && hue < 85 && sat < 85) || (hue == 0 && sat == 0) ) && value > 80) return WALL;
  else if( (hue >= 0 && hue < 11 && sat < 80) || (hue > 354 && hue < 361)) return OPPONENT;
  else if( hue > 60 && hue < 140 ) return FLOOR;
  else {
    return UNCLASSIFIED;
  }
}
//------------------
int min(int a, int b){
  if(a>b) return b;
  else return a;
}
//------------------
int max(int a, int b){
  if(a<b) return b;
  else return a;
}
//------------------
int main(int argc, char *argv[]) {
  FILE *infile;
  char fname[40];
  unsigned char line[LINESIZE];

  imgBuffer = malloc(sizeof(line));

  int lcount = 1;

  /* Open the file.  If NULL is returned there was an error */
  if((infile = fopen(argv[1], "rb")) == NULL) {
    printf("Error Opening File.\n");
    exit(1);
  }

  // remove first three lines, we dont care about them
```

```
  fgets(line, sizeof(char)*16, infile);
  fgets(line, sizeof(char)*16, infile);
  fgets(line, sizeof(char)*16, infile);

  while( fgets(line, sizeof(line), infile) != NULL ) {
    /* Get each line from the infile */
    imgBuffer = realloc(imgBuffer, sizeof(line)*lcount);

    int i = 0;
    while(i < LINESIZE) {
      imgBuffer[(lcount-1)*LINESIZE+i] = line[i];
      i++;
    }
    lcount++;
  }
  fclose(infile);  /* Close the file */

  // Print neccesary chars and then the analyzed file
  printf("P6\n");
  printf("176 144\n");
  printf("255\n");
  CamAnalyze();
  // ppm-files should end with null-char
  printf("\0");
}
```

5.2.5 Picture saver

```
#include "eyebot.h"
#define PI 3.14159

int knapp;
int bildnr=0;
typedef BYTE bigcolimage[144][176][3];
bigcolimage img[10];
int i,j;
char temp[100];
VWHandle vw;

void sendString(char* s )
{
  while (*s)
  {
    OSSendRS232( s, SERIAL1);
    s++;
  }
}

void getBigRGBData()
{
  AUBeep();
  if (bildnr<10){
    CAMGetFrameRGB((BYTE *)img[bildnr]);
    bildnr++;
    printf("bildnr:%d\n",bildnr);
  }
  else{
    sendBigRGBData();
  }
}

void sendBigRGBData(){
  VWSetSpeed(vw, 0, 0);
  bildnr--;
  OSInitRS232( SER115200, NONE, SERIAL1 );
  while(bildnr>0){
    LCDClear();
    LCDMenu("","send","","end");
    LCDPrintf("bildn: %d\n", bildnr);
    LCDMenu("","send","","end");
    knapp=KEYGet();
      if(KEY4==knapp)
    exit(0);
    if(KEY2){
```

```
      printf("bildnr:%d\n", bildnr);
      sprintf(temp,
       "P6\n"
       "176 144\n"
       "255\n");
      sendString(temp);
      for(i=0, j=0; i < 144*176; i++, j+=3){
        BYTE r,g,b;
        r=((BYTE *)img[bildnr])[j];
        g=((BYTE *)img[bildnr])[j+1];
        b=((BYTE *)img[bildnr])[j+2];
        temp [0]=r;temp [1]=g;temp [2]=b;temp [3]=0;
        if ((i&0x3ff) == 0){
          LCDClear();
          LCDPrintf(" %4d of %4d\n", i, 144*176 );
          LCDPrintf("bildnr: \n%d",bildnr);
        }
          sendString(temp);
      }

      bildnr--;
      LCDClear();;
    }
  }
}

int main(){
  int i = CAMInit(NORMAL);
  CAMSet(FPS7_5,0,0);
  while(1){
    getBigRGBData();
    OSWait(150);
  }
}
```

5.2.6 Vision

```
#include "eyebot.h"
#include "constants.h"
#include <stdio.h>

int main (void){
  int k;

  CamInit();

  LCDMenu("mode"," obj","","");
  CamSetDispMode(SILOUETTE);

  while(1) {
    k = KEYRead();
    if( k==KEY1 ) CamCycleDispMode();
    if( k==KEY2 ) CamCycleDispObject();

    CamTakePicture();
    CamAnalyze();
    OSWait(10);
  }
}
```