Raset, soccer playing robot.

RAS1

Robin Hagblom   Jonas Karlsson

Abstract

Raset is the name of a soccer playing robot built for the course Robotics and autonomous systems (2D1426). This is the report of the construction and evaluation of that robot. It contains a detailed description of both it's hardware, software and game play tactics.

Contents

1 Introduction

We have as a part of the course Robotics and autonomous systems (2D1426) built a robot capable of playing a game of robot soccer. During the the process of building the robot we had to consider things like locomotion, image processing, localisation and path planning. A robot that just plays soccer may sound easy but the fact that the robot is supposed to be fully autonomous makes it a not neglectable task. We also encountered a wide range of malfunctioning hardware which made everything much harder.

2 Background

Initially we where given an EyeBot kit, consisting of 2 motors (with built in encoders and gear boxes), 1 servo, 1 EyeBot controller card and 1 EyeBot camera and a battery. The EyeBot kit is developed by The Univ. of Western Australia[2] and can be used to build a wide range of robots eg. walking, flying and wheeled. We were also given a workstation with Debian and gcc68 installed .This gave us a good platform to start building the robot.

2.1 The Game

The game was to be played on a specially designed table and consisted of two parts. The first part, as acted as a minimum requirement for the course, consisted of being able to score from various ball positions on the field and the second was a tournament where all the teams competed against each other. Most of the rules and conditions can be found in the rules[1] pdf.

3 The Robot

Our first goal was to make a simple and lowtech design and get it to work, then add functionality to get a more complex and advanced system. But because of a series of malfunctioning hardware the robot did not make as much evolving as we had hoped.

3.1 Hardware

The initial hardware was fundamental to the project but it was not enough, we needed more parts to make the robot complete.

We quickly decided that the robot should have a roller that gives the ball a backspin so that it would stay close to the robot, thus making the handling of the ball better. See figure 1. An

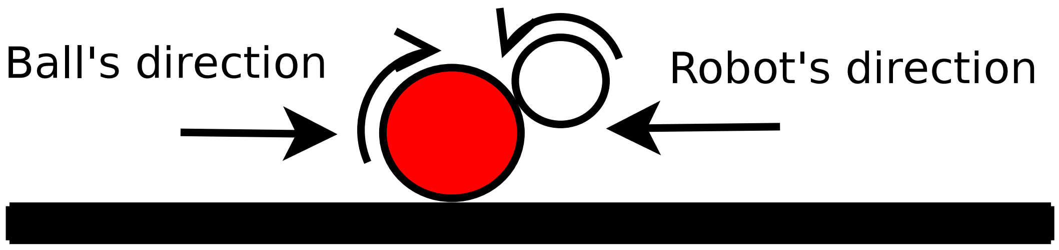

Figure 1: The backspin produced by the roller pushes the ball against the robot.

other taught we had from the start was to mount the camera at such an height and angle that the robot could see the whole field but nothing above the walls.

In figure 2 the placement of all the parts on the frame can be viewed.

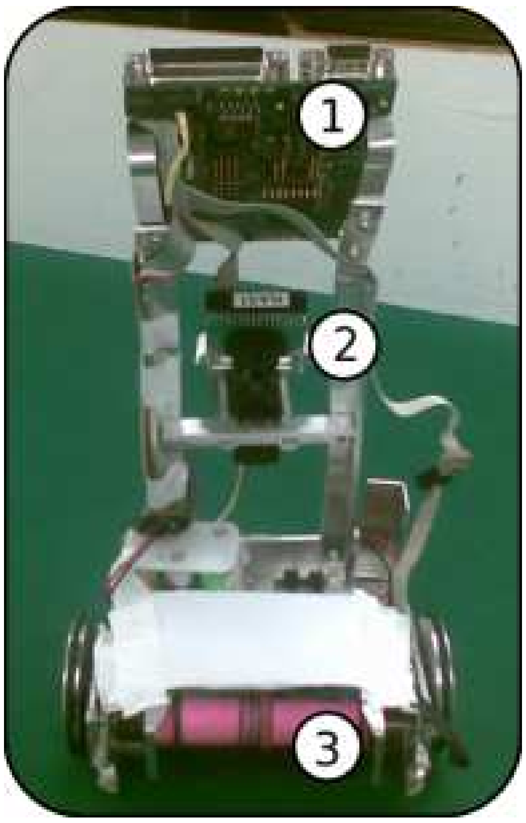

Figure 2: (1) EyeBot card, (2) Camera and servo, (3) Roller

3.1.1 EyeBot card

To be able to fit all the thing on the frame and still be able to have a free line of sight the EyeBot card had to be placed on the top. This made the robot top heavy but as far as we have seen it hasn't caused a problem. If it had been pushed diagonally backwards it might had fallen over. But as we did not qualify for the competition this was nothing we experienced.

Unfortunately our card suffered from an broken encoder which forced us to compensate for it in the code, more on that in the software section (3.2).

3.1.2 Roller

We mainly tried two kinds of rollers. The first version was thick and soft. It also had a sticky surface[1] which made the ball almost stick to the roller. This was not allowed in the competition but we figured that by then so much particles would have got stuck to the tape it would only be little sticky.

The second was a section of plastic tube covered whit a balloon, as can be seen in figure 2. None of them worked perfectly and at the end we decided to go without it. The robot was however still able to score goals.

3.1.3 Camera

As said before the placement of the camera was crucial and we wanted to mount it high enough to be able to see both the ball and the whole field. There was also some concern about the servo, were we going to use it or not. We did not have to move the camera vertically and the robot could

[1]inside-out tape

turn reasonably fast so initially there seemed to be no use for it. But we decided to go with the servo from start any way, so we didn't have to change the mounting later. The final code includes code for using the servo.

3.1.4 Frame

The main influence on the design of the frame was the placement of the camera, which made the robot kind of tall compared to other robots. Our first design was an even taller robot, but we soon decided to shorten it. The bottom plate

3.1.5 Wheels/Motors/Encoders

The wheels came from the start kit and at least one of them had to be replaced because the mounting was broken. Apart from that they seemed to work fine.

The motors suffered from some kind of error which all of a sudden made them stall and then again after some time run as usual. The motors also didn't turn equally fast which made the robot drive on a curve instead of a line. This was also something we had to compensate for.

3.1.6 Sensors

Our main sensor is the camera and it is also our only sensor, that is also why we made such an effort to give it as good view as possible of the field. Additional sensors might have improved the robot, but to us the camera was enough.

As a matter of fact we did have one more sensor, the encoders in the motors. This was something that was available through the hardware and nothing we added. Read about how we used this in section (3.2).

3.2 Software

The software Raset ran on was written in C with the C standard library and the RoBIOS Library Functions[3] and was compiled using gcc68 on a debian Linux machine. The program for running the robot could mainly be devided into four categories the menus3.2.1, the image processing3.2.2, the motor control3.2.3 and the scoring procedure3.2.4.

3.2.1 Menus

The menus are two simple menus shown on the LCD screen of the bot for configuring and controlling the bot during runtime. The first menu consists of four commands: setting white point compensation variables, sending pictures to the computer, setting the balls hue and starting the scoring procedure. While these options are shown, a self-updating black and white image of what the camera is currently seeing is also shown on the LCD screen. As described later, the white point compensation gets reset at each taken picture during the scoring procedure, but before the scoring procedure has started it must be set manually with the first menu option to be able to get an accurate image or accurate ball hue with the two other options in the first menu. This option calls the function described in the image processing section3.2.2 and will thus also be described later.

The second command, to send and take pictures is also very straightforward. The first time the option is selected the bot will take a full colour picture with the camera, using white point compensation and send it using the serial cable all following selections of the same option. Though this isn't very flexible, it was necessary since the robot only had four buttons and we wanted to avoid using submenus.

The third command, labeled "BALL" would quite simply take a picture with the camera and calculate the mean value of the hue of the center 100 pixels of the image to use as ball hue. This would of course only work if the ball was placed so that it would sit in the absolute center of the cameras view frustum. It turned out that the balls hue always was 42 as long as the white point was correctly configured, so most of the time we simply just used 42 as a default value without resetting the ball hue at every run.

The fourth and final command, labaled "GO" initiated the main scoring procedure and activated the second menu, which is shown for the entirity of what is left of the program's runtime. The second menu changed a lot during the programming of the bot as it was used mainly for debugging, but the final and most common application had four options. One option for taking full size colour pictures and storing them in the bot's memory, one for sending the last of these picture over the serial cable. Another for sending the same image but for one detail, it changed the colour of everything that had the ball's hue into white initially and purple later during development and a final option for terminating the program.

3.2.2 Image processing

Image processing was one of the main hurdles during the project since it is essential to the robot for deciding what to do since it was the only sensor we used. To be able to make informed decisions the robot has to classify the different parts of an image as different game elements, this is only possible if we have consistent images and a good algorithms for finding the different elements in the image. To begin with we had very nice and consistent images, a tad yellow tinted, but still very consistent. As our images were so nice and consistent we focused on the game element classifier. To begin with we wanted a fast algorithm so our robot could remain moving as much as possible and not have to stop to process images. To achieve this we decided that going through every pixel in the image every time would be a waste of processing power since the ball no matter how far away never is only one pixel wide. Instead we opted for a two pass algorithm were a first sweep would iterate through certain defined sample points and would initiate the second pass at all sample points that gave hue values within our ball hue threshold (the balls hue, 42 +-6). The second pass would then evaluate the hue of the pixels in a nine by nine square spread out over a square with twice the diameter of the ball at that specific row height, taken from a precalculated array. The algorithm would then return the mean X and Y coordinates of pixels with hues within threshold of the second sweep with the most pixels within the threshold.

This algorithm turned out to be very fast, especially after optimizations were made to first search close to the bot and only search far away unless nothing was found close, but it without a doubt had issues with balls at longer distances as the spread of the first sweep wasn't narrow enough further away to catch the ball at all times and it tended to slip through the grid. It definitely would have been possible to prebuild arrays and matrixes to adjust the sampling grid into an unstructured grid to compensate for this, but because of major time constraints, mostly due to trying to dealing with several hardware failures this never got done. Instead a quick'n'dirty slower algorithm was implemented where every pixel was evaluated columnwise and the column with the most hue values within the threshold was selected as the ball column. The Y value was also calculated by using the mean Y value of all "hits" from that column. This algorithm turned out to work slightly more reliably at medium distances and was thus used at the final lab 1 completion.

After some time during development the bot started giving very weirdly coloured images with all kinds of colour and hue distortions. These got worse and worse but we assumed they were some kind of hardware failures as they hadn't been there to begin with. A lot of time was used to try to fix this, but to no avail, finally we learnt that this was due to the autobrightness in the camera adapting to differing light conditions. It's quite odd that we never had this problem in the first half of the course, but we can only guess that we must have been very "lucky" with light conditions at that time. After learning about this we implemented white point compensation by taking pictures of white pieces of paper and then calculating what constants were needed to multiply the R, G and B values with so that the paper turned out white in the image. After some initial trouble we managed to perfect this so that we got very reliable images after setting our white point compensation variables. It turned out though that these variables varied greatly depending on where on the pitch we were, to such an extent that the ball hue would go from 42 to 150 depending on pitch location. After mounting a white piece of paper as shown2 and recalibrating the white point variables every time the robot takes a picture we finally managed to get accurate values during all light conditions.

As all outputted camera images were RGB and we only used hue values, we needed a fast and accurate pixelwise RGB to hue method. That it had to be fast was quite clear as it had to be used on every pixel we wanted the robot to evaluate. At first we borrowed this from a demonstration soccer eyebot program that was supplied by the course staff, but this method turned out to be insanely slow as it used lots of floating point multiplications, something the eyebot is very very very bad at. After replacing all float point operations with int operations at a 1024 times larger scale we could both multiply and divide very quickly as 1024 is two to the power of ten and scaling could be achieved simply by bitshifting 10 bits back and fourth.

3.2.3 Motor control

As we had to run with only one encoder for the last half of the development process we could not use the supplied RoBIOS[3] vwdrive. Instead we ran the motors at percentages of their max capability and counted the distance by use of the one working encoder. To increase distance reliability we chose to both motor brake by reversing the motors for a very brief moment when the robot reached it's target distance and slow down the engines when the robot started to get close to it's destination. To not overshoot our distances while the wheels were slowing down after power was cut we had to do this even though both these methods resulted in annoying problems. The motor resulted in very sharp stops which, while handling the ball would often result in the ball shooting away due to inertia, this could have been remedied had the second method worked well, but as our motors kept stalling at medium to low speeds we could never go below 35-40 percent of full motor speed. As the stalling speed would differ quite a lot depending on battery charge, an even greater value than needed had to be used. To reduce the risk of losing the ball, a special ball turn function was developed where the robot would turn only with it's right wheel (the wheel which had a working encoder) and simply drive double the distance to get a wider and slower turn than spinning on the spot. As only the right wheel could be used, left turns made the robot back up, usually losing grip of the ball, though still in a much better position to play it.

3.2.4 Scoring procedure

The scoring procedure is best illustrated by the flowchart below3, but can be divided into three main parts: classifying the target, moving to the target and searching for the target. These three parts all have two different and mutually exclusive modes in common, having the ball as a target and having the goal as a target. Generally the robot would be in the ball mode until it judges it has the ball and would then procede to the goal mode until it judges that it has lost the ball.

Every cycle will start by the bot trying to classifying the target. While doing this it will first take a picture with the camera, process the image as described above3.2.2 and depending if it finds the ball or not, either engage the moving to the target part or the searching for ball part. In the moving to the target part, the robot will first establish the range to the target, using the target's Y coordinate and a precalculated distance array. After this has been done the robot will judge if the target is straight ahead or if it has to turn to face the target. If it has to to turn to face the target, the bot will turn the calculated angle to the target and then continue to the next cycle. If the target is straight ahead, the robot will instead drive forward the distance supplied by the range finding array before jumping to the next cycle. This is unless the bot judges that it already is next to the target. If the target is the ball, it would then decide it has the ball and change target to the goal before jumping to the next cycle. If the target instead is the goal, it will decide that it must have scored and then jump to the next cycle, cycling through until the ball has been lost (taken away).

If the bot didn't find the target while processing the image and thus starts searching for the ball it will first turn the servo 45 degrees to the right and then try to find the ball there. If it didn't find the ball after turning the servo the whole robot turns to the left and jumps to the next cycle.

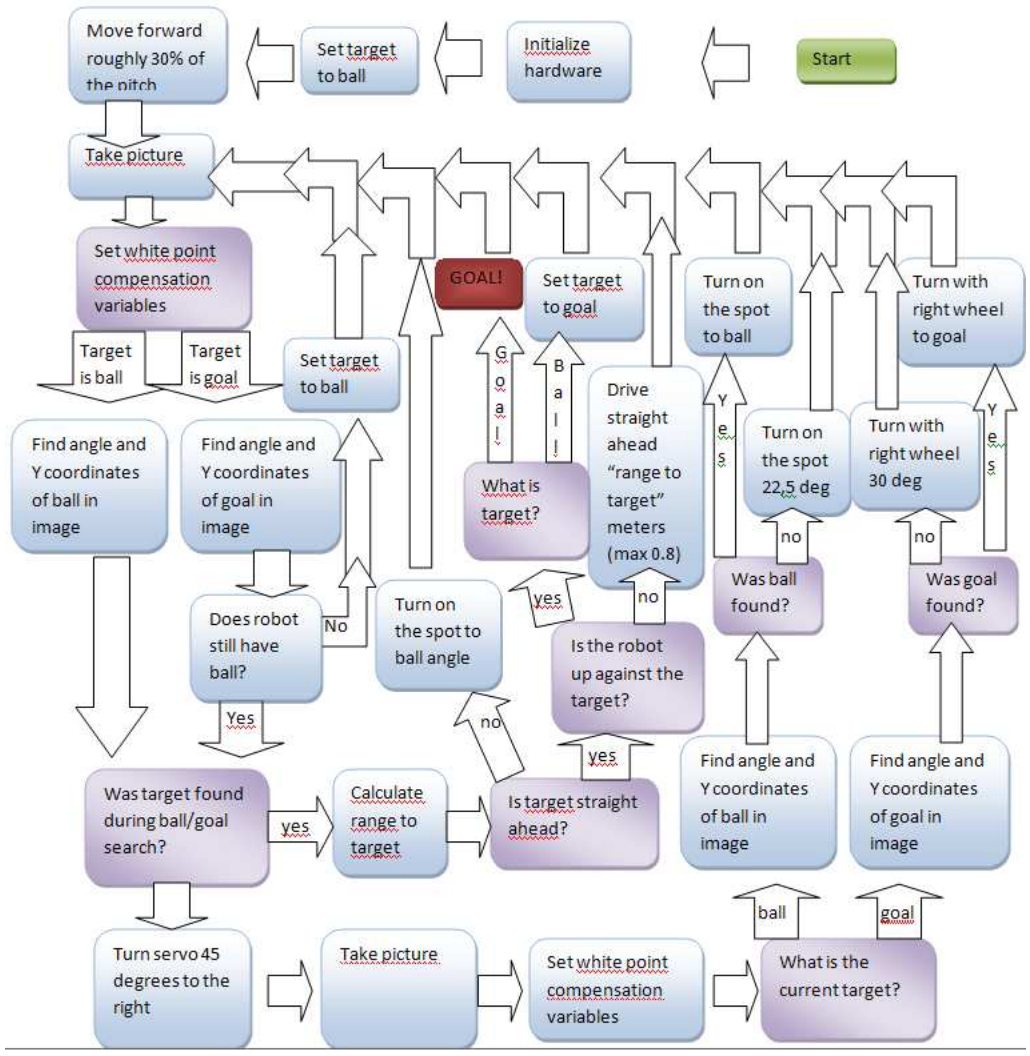
Start
Initialize hardware
Set target to ball
Move forward roughly 30% of the pitch
Take picture
Set white point compensation variables
Target is ball
Target is goal
Find angle and Y coordinates of ball in image
Find angle and Y coordinates of goal in image
Does robot still have ball?
No
Yes
Set target to ball
Turn on the spot to ball angle
GOAL!
Goal!
Ball!
What is target?
Set target to goal
Drive straight ahead "range to target" meters (max 0.8)
yes
no
Is the robot up against the target?
Was target found during ball/goal search?
yes
Calculate range to target
Is target straight ahead?
no
yes
Turn on the spot to ball
Yes
Turn on the spot 22,5 deg
no
Was ball found?
Turn with right wheel 30 deg
no
Was goal found?
Turn with right wheel to goal
Yes
Find angle and Y coordinates of ball in image
Find angle and Y coordinates of goal in image
ball
goal
Turn servo 45 degrees to the right
Take picture
Set white point compensation variables
What is the current target?

Figure 3: Flowchart for main loop

The flowchart is almost completely comprehensive, but for three minor details that were added as backup features to add to the overall robustness of the procedure. If the robot were for any reason ever to lose the ball, it will backup a short distance before the next cycle. This is for two reasons, first of all because lost balls often fall off to the right or left both out of the viewscope of the bot, unless it backs up a a bit. Secondly it helps getting out of the goal if the robot has scored and the ball is removed from the goal. To avoid infinite loops or searching when the ball is too far away, the robot will after having turned 360 degrees simply drive forward a small distance, hoping for better luck there. The last detail is a wall crashing fail-safe. If the robot were ever to drive for an unreasonable time in one direction without ever reaching it's destination the robot will immediately back up, turn 180 degrees and then complete it's current move.

4 Results

Despite all the faulty hardware we managed to get the robot to score. But we did not manage to qualify for the competition.

The placement of the camera proved to be a good decision and gave the results we strived for.

It was too bad that we had so many problems with the image processing that we couldn't utilize it's potential fully. The servo was first put on just in case we would find a use for it, but turned out to be quite useful in the ball searching part of the scoring procedure.

The roller didn't work well at all, the soft curved roller especially as it became totally useless when the illegal glue wore off.

5 Conclusions

If we had been more lucky and didn't have to struggle so much with the hardware the robot would likely been much more successful. Our decision to go lowtech first and then evolve had also worked better if we had made it work sooner. Now the robot stayed the same from the first prototype and did no evolving. Despite all hardware problems taking a whole lot of time, initially choosing to skip white point compensation as it didn't "seem" to be needed was a huge error that caused a whole lot of trouble and unneccessary work.

We never found out exactly why first one encoder broke and then why one encoder port on the eyebot broke, but most likely it was due to some kind of short circuit in our custom made extension cables for the motors. How any short circuit could have occured is a mystery as they cables looked alright, but it quite obviously is VERY important to make sure everything is isolated completely and throroughly.

The choice right at the end to switch to the quick and dirty ball finding algorithm also was a shame, with more time on refining the first sweep and less time on dealing with: broken hardware, stupid design decisions and incorrectly diagnosed faults we probably would have been able to make a bot could take pictures while driving and adjust it's course on the fly as initially intended.

Quite obviously a straight very thin roller with good grip is the most effective, something we realised way too late as our soft roller's effectiveness plumeted as the glue from the inside out tape got rubbed off.

The robot can use only the camera as only sensor and still be successful. Adding more sensors means more things to check in the code but also means more information from the environment. A proper sensor to make sure we had the ball would have been very helpful, but with our budget constraints this was never really an option.

Use white point compensation to handle different lighting conditions.

References

[1] Rules http://www.csc.kth.se/utbildning/kth/kurser/DD2426/rules/rules-rev1_2.pdf

[2] EyeBot http://robotics.ee.uwa.edu.au/eyebot/

[3] RoBIOS Library Functions http://robotics.ee.uwa.edu.au/eyebot/doc/API/library.html

A Source code

```
#include "eyebot.h"
#include <stdlib.h>
#include <math.h>
#include <stdio.h>


#define DRIVE

#define REALBALLRADIUS 2.5
#define RADIUSRES 4
```

```
#define GRIDROWS 36
#define GRIDCOLUMNS 48
#define WIDTH 176
#define HEIGHT 144
#define RGB 3
#define CELLHEIGHT HEIGHT / GRIDROWS
#define CELLWIDTH  WIDTH  / GRIDCOLUMNS
#define CAMHEIGHT 14.5
#define MINIMALHITS 4
#define LEFTMOTORCOMP 0.97//1.18
#define TURNRADIUS 0.0775
#define RIGHTWHEELTURN
#define CAMERA_Z_OFFSET 0.176
//#define DONOTDRIVE
#define GOAL_Y_INC 1
#define GOAL_X_INC 1

#define TURNSPEED 40


enum{BALL,BLUE,YELLOW};

BYTE BALL_LOW = 37;
BYTE BALL_HIGH = 45;

#define BALL_THRESH 7

#define BLUE_LOW 185
#define BLUE_HIGH 195

#define YELLOW_LOW  60
#define YELLOW_HIGH 80

#define BRAKE 2

#define SWIDTH 82
#define SHEIGHT 62
#define OURPI 3142
#define SATLIMIT 0.3

int g_target = BALL;
int g_goal = BLUE;
float x = 0;
float y = 0;
float dir = 0;
VWHandle g_vw;


int wht_mult_red1 = 1400;
int wht_mult_green1 = 1024;
int wht_mult_blue1 = 2100;

#define MIN(a,b) (a<b?a:b)
#define MAX(a,b) (a>b?a:b)
```

```
void floatmul(int* float3, int* arg);



int BALLRADIUS[GRIDROWS];

typedef struct {
  MotorHandle l;
  MotorHandle r;
  QuadHandle ql;
  QuadHandle qr;
} mhPair;

void setBallHue(BYTE img[HEIGHT][WIDTH][RGB])
{
    BYTE hue;
    int huesum;
    huesum = 0;
    int i,j;

    for(i=HEIGHT/2-5;i<HEIGHT/2+5;i++)
    {
        for(j=WIDTH/2-5;j<WIDTH/2+5;j++)
        {
            huesum+=RGBtoHue2(img[i][j][0],img[i][j][1],img[i][j][2]);
        }
    }

    hue = huesum/100;

    BALL_LOW = hue - BALL_THRESH;
    BALL_HIGH = hue + BALL_THRESH;
}

void setWhitePoint(BYTE img[HEIGHT][WIDTH][RGB])
{
    BYTE r,g,b;

    int i,j;
    int rsum,gsum,bsum;
    rsum=gsum=bsum=0;

    for(i=HEIGHT/2-5;i<HEIGHT/2+5;i++)
    {
        for(j=WIDTH/2-5;j<WIDTH/2+5;j++)
        {
            rsum+=img[i][j][0];
            gsum+=img[i][j][1];
            bsum+=img[i][j][2];
        }
    }

    r = rsum / 100;
    g = gsum / 100;
    b = bsum / 100;

    r -= 16;
    g -= 16;
    b -= 16;

    float maxval = MAX(r,MAX(g,b));

    wht_mult_red1 = (int)(((255.0)/(float)r)*1024);
    wht_mult_green1 = (int)(((255.0)/(float)g)*1024);
    wht_mult_blue1 = (int)(((255.0)/(float)b)*1024);

}


void SetWhitePointBot(BYTE img[HEIGHT][WIDTH][RGB])
{
    BYTE r,g,b;

    int i,j;
    int rsum,gsum,bsum;
    rsum=gsum=bsum=0;

    for(i=130;i<HEIGHT-1;i++)
    {
        for(j=70;j<105;j++)
        {
            rsum+=img[i][j][0];
            gsum+=img[i][j][1];
            bsum+=img[i][j][2];
        }

    }

    r = rsum / 100;
    g = gsum / 100;
    b = bsum / 100;

    r -= 16;
    g -= 16;
    b -= 16;

    float maxval = MAX(r,MAX(g,b));

    wht_mult_red1 = (int)(((255.0)/(float)r)*1024);
    wht_mult_green1 = (int)(((255.0)/(float)g)*1024);
    wht_mult_blue1 = (int)(((255.0)/(float)b)*1024);

}

void whitecompensate(BYTE *r, BYTE *g, BYTE *b)
{
    int r1,g1,b1;
    r1=*r-16;g1=*g-16;b1=*b-16;



    floatmul(&wht_mult_red1,&r1);
    floatmul(&wht_mult_green1,&g1);
    floatmul(&wht_mult_blue1,&b1);

    r1+=16;
    g1+=16;
    b1+=16;

    if(r1 < 255)
        *r = r1;
    else
        *r = 255;

    if(g1 < 255)
        *g = g1;
    else
        *g = 255;

    if(b1 < 255)
        *b = b1;
    else
        *b = 255;

}


void vStop(mhPair *m)
{
  MOTORDrive(m->l | m->r, 0);
}

void vAngleTurn(mhPair *m, int speed, float angleRad)
{
#ifdef VWWORKING
    VWDriveTurn(g_vw,angleRad,1);
    VWDriveWait(g_vw);
#else


    float meters = -(angleRad*TURNRADIUS);
     QUADODOReset(m->qr);

  if (meters < 0 || speed < 0)
  {

    /* Both should be negative for backwards driving. */
    if (speed > 0)
      speed = -speed;
    if (meters > 0)
      meters = -meters;
    while(QUADODORead(m->qr) > meters)
    {

#ifdef RIGHTWHEELTURN
      MOTORDrive(m->r, speed);
      MOTORDrive(m->l, -speed*LEFTMOTORCOMP);
#else
      MOTORDrive(m->l, -speed);
#endif

      OSSleep(1);
    }
#ifdef RIGHTWHEELTURN
    MOTORDrive(m->r, -speed);
#else
    MOTORDrive(m->l, speed*LEFTMOTORCOMP);
#endif
    OSSleep(BRAKE);

  }
  else {
    /* Forward driving */
    while(QUADODORead(m->qr) < meters) {


#ifdef RIGHTWHEELTURN
      MOTORDrive(m->r, speed);
      MOTORDrive(m->l, -speed*LEFTMOTORCOMP);
#else
      MOTORDrive(m->l, -speed);
#endif
      OSSleep(1);
    }

#ifdef RIGHTWHEELTURN
    MOTORDrive(m->r, -speed);
#else
    MOTORDrive(m->l, speed*LEFTMOTORCOMP);
#endif

    OSSleep(BRAKE);
  }
  vStop(m);


#endif
}
void vAngleBallTurn(mhPair *m, int speed, float angleRad)
{




    float meters = -(angleRad*TURNRADIUS*2);
```

```
#ifdef VWWORKING
    VWDriveCurve(g_vw, meters,angleRad,1);
    VWDriveWait(g_vw);
#else
    QUADODOReset(m->qr);

  if (meters < 0 || speed < 0)
  {

    /* Both should be negative for backwards driving. */
    if (speed > 0)
      speed = -speed;
    if (meters > 0)
      meters = -meters;



    while(QUADODORead(m->qr) > meters)
    {


#ifdef RIGHTWHEELTURN
      MOTORDrive(m->r, speed);
      //MOTORDrive(m->l, -speed*LEFTMOTORCOMP);
#else
      MOTORDrive(m->l, -speed);
#endif

      OSSleep(1);
    }


#ifdef RIGHTWHEELTURN
    //    MOTORDrive(m->r, -speed);
#else
    MOTORDrive(m->l, speed*LEFTMOTORCOMP);
#endif

    //OSSleep(BRAKE);

  }
  else {
    /* Forward driving */
    while(QUADODORead(m->qr) < meters) {


#ifdef RIGHTWHEELTURN
      MOTORDrive(m->r, speed);
      //MOTORDrive(m->l, -speed*LEFTMOTORCOMP);
#else
      MOTORDrive(m->l, -speed);
#endif
      OSSleep(1);
    }
```

```
#ifdef RIGHTWHEELTURN
    //MOTORDrive(m->r, -speed);
#else
    //MOTORDrive(m->l, speed*LEFTMOTORCOMP);
#endif

    //OSSleep(BRAKE);
  }
  vStop(m);

#endif

}
void vDriveDist(mhPair *m, int speed, float meters)
{
#ifdef VWWORKING
    VWDriveStraight(g_vw, meters, 0.25);
    VWDriveWait(g_vw);
#else
  QUADODOReset(m->qr);

  if (meters < 0 || speed < 0)
  {
    /* Both should be negative for backwards driving. */
    if (speed > 0)
      speed = -speed;
    if (meters > 0)
      meters = -meters;
    int mm = 0;
    while(QUADODORead(m->qr) > meters)
    {
      if (QUADODORead(m->qr) < (meters+0.25))
      {
    /* Slow down if we are close to goal distance, this is run cumulatively. */
          speed = (int) (speed * 0.95);
          if (speed > -30)
              speed = -30;
      }
      MOTORDrive(m->l | m->r, speed);
      MOTORDrive(m->l, speed*LEFTMOTORCOMP);

      int r = QUADODORead(m->qr);

      OSSleep(1);

      if(r == QUADODORead(m->qr))
      {
          mm++;
          if( mm > 10)
          {
              vStop(m);
              vDriveDist(m, 40,0.1);
              vAngleTurn(m,TURNSPEED,M_PI);
```

```
                break;
            }

        }
        else mm = 0;

      }
    }
    else {

        int n = 0;
      /* Forward driving */
      while(QUADODORead(m->qr) < meters) {
        if (QUADODORead(m->qr) > (meters-0.25)) {
          /* Slow down if we are close to goal distance, this is run cumulatively. */
          speed = (int) (speed * 0.95);
          if (speed < 35)
            speed = 35;
        }
        MOTORDrive(m->l | m->r, speed);
        MOTORDrive(m->l, speed*LEFTMOTORCOMP);

        int r = QUADODORead(m->qr);
        OSSleep(1);
        if((QUADODORead(m->qr) - r) < 1)
        {
            n++;
            if (n > 1000)
            {
                vStop(m);
                vDriveDist(m, 40,-0.1);
                vAngleTurn(m,TURNSPEED,M_PI);
                n = 0;
            }
        }
        else n = 0;
      }
      MOTORDrive(m->r, -speed);
      MOTORDrive(m->l, -speed*LEFTMOTORCOMP);

      OSSleep(BRAKE);
    }

    vStop(m);

#endif
}

#define TURN_LEFT -1
#define TURN_RIGHT +1

void vTurn(mhPair *m, int dir, float secs)
{
#ifndef VWWORKING
```

```
  if (dir < 0) {
    MOTORDrive(m->l, +40);
    MOTORDrive(m->r, -40);
  } else if (dir > 0) {
    MOTORDrive(m->l, -48); /* For some reason (friction?) it turns slightly less when turning rig
    MOTORDrive(m->r, +48);
  }
  OSSleep((int) (secs * 100));
  vStop(m);
#endif
}



void vInit(mhPair *m)
{
#ifndef VWWORKING

  m->l = MOTORInit(MOTOR_LEFT);
  m->r = MOTORInit(MOTOR_RIGHT);
  m->ql = QUADInit(QUAD_LEFT);
  m->qr = QUADInit(QUAD_RIGHT);

  if (   QUADGetMotor(QUAD_LEFT) != MOTOR_LEFT
         || QUADGetMotor(QUAD_RIGHT) != MOTOR_RIGHT
         || m->l == 0 || m->r == 0
         || m->ql == 0 || m->qr == 0){
    LCDPutString("vInit Err!\n");
    OSWait(200);
    exit(1);
  }
#endif
}

void vRelease(mhPair *m)
{
#ifndef VWWORKING
  MOTORRelease(m->l);
  MOTORRelease(m->r);
  QUADRelease(m->ql);
  QUADRelease(m->qr);
#endif
}


int camwidth = 0;
int camheight = 0;
int camfps = 0;

#define SERVO_CAM SERVO7
```

```
int RGBtoHue2(BYTE r, BYTE g, BYTE b)
{
  BYTE hue /*,sat, val*/, delta, max, min;

  //correct_colourB(&r,&g,&b);
  whitecompensate(&r,&g,&b);


  max   = MAX(r, MAX(g,b));
  min   = MIN(r, MIN(g,b));
  delta = max - min;
  hue =0;
  /* initialise hue*/

  /* val   = max;
     if (max != 0) sat = delta / max; else sat = 0;
     if (sat == 0) hue = NO_HUE;
  */
  if (2 * delta <= max) hue = NO_HUE;
  else {
    if (r == max) hue =  42 + 42*(g-b) / delta; /* 1*42 */
    else if  (g == max) hue = 126 + 42*(b-r) / delta; /* 3*42 */
    else if (b == max) hue = 210 + 42*(r-g) / delta; /* 5*42 */
    /* now: hue is in range [0..252] */
  }
  return hue;
}

void sendString2(int port, char* s )
{
  while (*s)
  {
    OSSendRS232( s, port);
    s++;
  }
}

void sendRGBData2(int port, BYTE *img )
{
    OSInitRS232( SER115200, NONE, port );


    unsigned int i,j;
    char temp[100];


    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)[j];
        g=((BYTE *)img)[j+1];
        b=((BYTE *)img)[j+2];
        whitecompensate(&r,&g,&b);
        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 );
        }
        sendString(temp);
    }

}


void showHueInterval(int low, int high, BYTE img [HEIGHT][WIDTH][RGB], BYTE highlight [HEIGHT][WI
{
    int row, column;
    BYTE r, g, b;
    int cr,cg,cb;
    BYTE hue, delta, max, min;

    for (row = 0; row < HEIGHT; row++)
    {
        for (column = 0; column < WIDTH; column++)
        {
            hue = RGBtoHue2(r,g,b);



            if (hue < high && hue > low )
            {
                highlight[row][column][0] = 125;
                highlight[row][column][1] = 255;
                highlight[row][column][2] = 0;

            }
            else
            {
                highlight[row][column][0] = (int)img[row][column][0];
                highlight[row][column][1] = (int)img[row][column][1];
                highlight[row][column][2] = (int)img[row][column][2];

            }
        }
    }

}

int withinHueInterval(int low, int high, BYTE pixelColor [3])
{
    int hue = RGBtoHue2(pixelColor[0],pixelColor[1],pixelColor[2]);
    return (hue > low && hue < high );
}

int hasBall(BYTE img[HEIGHT][WIDTH][RGB])
{
    int y,x,count;
    count = 0;
    for(y=100;y<116;y++)
    {
        for(x=55;x<130;x++)
        {
            if(withinHueInterval(BALL_LOW,BALL_HIGH,img[y][x]))
                count++;
        }
    }
    return count > 10;
}

void getWithinRGBImage(int low, int high, BYTE bwimage[82][62],BYTE colourimage[82][62][3], int c
{
    int x;
    int y;
    int colvalue = 0;

    for(y=0;y<SHEIGHT;y++)
    {
        for(x=0;x<SWIDTH;x++)
        {
            colvalue = colourimage[y][x][colour];
            if(colvalue > low && colvalue < high)
               bwimage[y][x] = 255;
            else
               bwimage[y][x] = 0;
        }
    }

}
void getWithinHueImage(int low, int high, image bwimage,colimage colourimage)
{
    int x;
    int y;

    for(y=0;y<SHEIGHT;y++)
    {
        for(x=0;x<SWIDTH;x++)
        {
            if(withinHueInterval(low, high, colourimage[y][x]))
               bwimage[y][x] = 255;
            else
               bwimage[y][x] = 0;
        }
    }
```

```
}


float getGoalAngle( BYTE img[HEIGHT][WIDTH][RGB] , int low, int high, int* Y)
{
    int x,y,hue;
    int sumX = 0;
    int meanX = 0;
    int count = 0;
    int maxY = 0;

    for(y=0;y<HEIGHT;y+=GOAL_Y_INC)
    {
        for(x=0;x<WIDTH;x+=GOAL_X_INC)
        {
            hue = RGBtoHue2(img[y][x][0],
                            img[y][x][1],
                            img[y][x][2]);

            if(hue > low && hue < high )
            {
                if(y>maxY)
                    maxY = y;
                sumX += x;
                count++;
            }

        }
    }
    *Y = maxY;
    if(count == 0)
        return -M_PI;

    meanX = sumX/count;

    return ((float)meanX*M_PI)/(4.0*WIDTH) - 0.3927;
}

float getBallAngle( BYTE img[HEIGHT][WIDTH][RGB] , int low, int high, int* Y)
{
    int row, col, count, tempcount, maxrow, sumrow, maxcol;




    count = MINIMALHITS;
    maxrow = -1;
    maxcol = -1;

    for(col = 0; col < WIDTH; col++)
    {
        tempcount = 0;
        sumrow = 0;
        for(row=0;row<HEIGHT;row++)
        {
            if (withinHueInterval(low, high, img[row][col]))
            {
                tempcount++;
                sumrow += row;

            }
        }
        if( count < tempcount)
        {
            maxcol = col;
            count = tempcount;
            maxrow = sumrow/count;
        }

    }

    *Y = maxrow;
    return ((M_PI*(float)maxcol)/((float)WIDTH*4.0))-0.3927; //0.3927 = PI/8

}

float getBallAngle2( BYTE img[HEIGHT][WIDTH][RGB] , int low, int high, int* Y)
{
    int row, col;
    int huetemp;
    int huenr;
    int meanX;
    int meanY;


    int closestX = -1;
    int closestY = -1;
    int closestNr = MINIMALHITS;
    int closestMeanX = 0;
    int closestMeanY = 0;
    int huetemp2;
    int rowNo = GRIDROWS/2;
    int rowstart;
    int rowend;
    int rowcolinc;
    int colstart;
    int colend;



    for(row = CELLHEIGHT/2 + HEIGHT/2; row < HEIGHT; row = row + CELLHEIGHT)
    {
        rowcolinc = BALLRADIUS[rowNo] / RADIUSRES;


        for( col = CELLWIDTH/2; col < WIDTH; col = col + CELLWIDTH)
        {
            if (withinHueInterval(low, high, img[row][col]))
```

```
{

    huetemp = 0;
    huenr = 0;
    meanX = 0;
    meanY = 0;
    int i,j;

    rowstart = row-BALLRADIUS[rowNo];
    rowend = row + BALLRADIUS[rowNo];
    colstart = col-BALLRADIUS[rowNo];
    colend = col - BALLRADIUS[rowNo];

    if(rowstart < 0) rowstart = 0;
    if(rowend > HEIGHT) rowend = HEIGHT;
    if(colstart < 0) colstart = 0;
    if(colend > WIDTH ) colend = WIDTH;

    for(i = rowstart;i <= rowend;i+= rowcolinc)
    {

        for (j = colstart; j <= colend; j+=rowcolinc)
        {
               huetemp2 = RGBtoHue2(img[i][j][0],
                                 img[i][j][1],
                                 img[i][j][2]);


               if(huetemp2 > low && huetemp2 < high)// && sat > SATLIMIT )
               {
                   huenr ++;
                   meanX += j;
                   meanY += i;
               }
        }

    }


    //huetemp = abs(huetemp-(25*42));
    if(huenr >= closestNr)
    {
        if(meanX > 0) meanX = meanX / huenr;
        if(meanY > 0) meanY = meanY / huenr;
        closestNr = huenr;
        //closestHue = huetemp;
        closestX = col;
        closestY = row;
        closestMeanX = meanX;
        closestMeanY = meanY;
    }
```

```
        }
        else
        {

        }

    }

    rowNo++;
}

int rowNo2 = 0;
int rowNoTemp;

if( closestMeanX == 0 )
{

    for(row = CELLHEIGHT/4 ; row < HEIGHT/2; row = row + CELLHEIGHT/2)
    {
        rowNoTemp = rowNo2/2;
        rowcolinc = BALLRADIUS[rowNoTemp] / RADIUSRES;


        for( col = CELLWIDTH/4; col < WIDTH; col = col + CELLWIDTH/2)
        {
            if (withinHueInterval(low, high, img[row][col]))
            {

                huetemp = 0;
                huenr = 0;
                meanX = 0;
                meanY = 0;
                int i,j;

                rowstart = row-BALLRADIUS[rowNoTemp];
                rowend = row + BALLRADIUS[rowNoTemp];
                colstart = col-BALLRADIUS[rowNoTemp];
                colend = col - BALLRADIUS[rowNoTemp];

                if(rowstart < 0) rowstart = 0;
                if(rowend > HEIGHT) rowend = HEIGHT;
                if(colstart < 0) colstart = 0;
                if(colend > WIDTH ) colend = WIDTH;

                for(i = rowstart;i <= rowend;i+= rowcolinc)
                {

                    for (j = colstart; j <= colend; j+=rowcolinc)
                    {
                        huetemp2 = RGBtoHue2(img[i][j][0],
                                             img[i][j][1],
                                             img[i][j][2]);
```

```
                            if(huetemp2 > low && huetemp2 < high)// && sat > SATLIMIT)
                            {
                                huenr ++;
                                meanX += j;
                                meanY += i;
                            }
                        }

                    }



                    if(huenr >= closestNr)
                    {
                        if(meanX > 0) meanX = meanX / huenr;
                        if(meanY > 0) meanY = meanY / huenr;
                        closestNr = huenr;

                        closestX = col;
                        closestY = row;
                        closestMeanX = meanX;
                        closestMeanY = meanY;
                    }



                }
                else
                {

                }

            }

            rowNo2++;
        }

    }



    *Y = closestMeanY;



    return ((M_PI*(float)closestMeanX)/((float)WIDTH*4.0))-0.3927; //0.3927 = PI/8
}


void floatmul(int* float3, int* arg)
{
    *arg = (*float3)*(*arg);
    *arg >>= 10;
```

```
}

int getHue(colimage img)
{
  int row, column;
  int count = 0;
  int my_hue = 0;
  int hue = 0;

  for (row = imagerows/2 - 2; row < (imagerows/2 + 3); row ++)
    for (column = imagecolumns/2 - 2; column < imagecolumns/2 + 3; column ++)
    {
      hue = RGBtoHue2(img[row][column][0], img[row][column][1],
         img[row][column][2]);
      if (hue != NO_HUE)
      {
        my_hue += hue;
        count ++;
      }
    }

  if (count != 0)
    my_hue /= count;


  return my_hue;
}

void getRanges(int ranges[HEIGHT])
{
    int y;
    for(y = 0; y<HEIGHT; y++)
    {
        ranges[y] = ((double)CAMHEIGHT-(double)REALBALLRADIUS)*tan(((double)(HEIGHT-y)*M_PI)/((do

    }


}

void fillRadii(int radii[GRIDROWS], int ranges[HEIGHT] )
{
    int row;
    double diagonal;
    for(row = 0; row<GRIDROWS; row++)
    {
        int a = (ranges[row*CELLHEIGHT + (int)CELLHEIGHT/2]);
        int b = ((double)CAMHEIGHT - (double)REALBALLRADIUS);
        diagonal = sqrt(a*a+b*b);
        radii[row] = ((double)(HEIGHT*4)/M_PI)*atan((double)REALBALLRADIUS/diagonal);
        if( radii[row] < RADIUSRES ) radii[row] = RADIUSRES;

    }
}
```

```
/**********************************************************************/
/** Start program.
    The main function controlls the several states of the
    program. It always shows the input of the camera, and starts the
    main menue, where you can set several parameters like the ball
    colour or the threshold for ball recognition.
    Before starting the ball search, an RGB tou hue lookup table can
    be built (option to do so when exitting SET-->COL menu).
    This will speed upp tracking by an order of magnitude.
*/
/**********************************************************************/

int main(void) {

    int port = SERIAL1;

    /** flags to exit loops around switch-statements */
    int exit_settings = FALSE;
    int exit_program = FALSE;
    int exit_settings2 = FALSE;
    int exit_colour = FALSE;
    int bTakenpic = FALSE;
    int bRGB = 0;
    int turnedAngle = 0;

    //int ballradii[GRIDROWS];
    int ranges [HEIGHT];
    getRanges(ranges);
    fillRadii(BALLRADIUS, ranges);

    LCDPrintf("%3d\n",'end' );

    // konfigurerar motorn

    mhPair* m = malloc(sizeof(mhPair));
    vInit(m);



    //#endif

    /** picture to work on */

    /** pictures for LCD-output */

    int key;
    BYTE servopos=136;
    BYTE servooldpos = 136;

    /** start initialisations */
    LCDMode(SCROLLING|NOCURSOR);  /* init lcd */
    LCDClear();
    Init_Cam();                  /* init camera */
    CAMGet(&camfps, &camwidth, &camheight);

    CAMSet(FPS15,0,0);

    BYTE img[HEIGHT][WIDTH][RGB];

    LCDSetPos(0,0);
    LCDPrintf("Warming up camera\n");

    int pic;
    for(pic =0;pic<100;pic++)
    {
        CAMGetFrameRGB((BYTE*)img);
        SetWhitePointBot(img);
    }


    CAMSet(FPS1_875,0,0);

    LCDPrintf("Camera warmed up\n");


    BYTE highlight[HEIGHT*WIDTH*RGB];
    BYTE takenpic[HEIGHT*WIDTH*RGB];
    colimage colourimage;
    image bwimage;

    ServoHandle cam_servo= SERVOInit(SERVO_CAM); // init servo
    SERVOSet(cam_servo,servopos);

    int count = 0;
    int rangeToBall;
    int ballY;
    int ballY2;

    int goalLow;
    int goalHigh;

    LCDSetPos(0,0);

    if(g_goal == BLUE)
    {
        goalLow = BLUE_LOW;
        goalHigh = BLUE_HIGH;
    }
    else if(g_goal == YELLOW)
    {
        goalLow = YELLOW_LOW;
```

```
        goalHigh = YELLOW_HIGH;
    }
    else
    {

        LCDPrintf("Error: invalid goal colour\n");
        return 0;
    }

///////////////////////////////////////////////
/*------------BEGIN OF MAIN LOOP-------------*/
///////////////////////////////////////////////

    LCDClear();
    LCDMenu("Wht", "SND ", "BALL","GO");

    int bStart = 0;
    int i;

    int bPictaken = 0;

    do
    {
        CAMGetFrame(bwimage);
        LCDPutGraphic(bwimage);

        key = KEYRead();

        switch(key)
        {
        case KEY1:
            LCDPrintf("Doing white point compensation\n");
            CAMGetFrameRGB((BYTE*)img);
            setWhitePoint(img);
            LCDPrintf("r:%d g:%d b:%d",wht_mult_red1, wht_mult_green1, wht_mult_blue1);
            break;

        case KEY2:
            if (bPictaken == 0)
            {
                CAMGetFrameRGB((BYTE*)takenpic);
                bPictaken = 1;
            }
            else
                sendRGBData2(port,(BYTE*)takenpic);

            break;

        case KEY3:
            LCDPrintf("Checking ball hue\n");
            CAMGetFrameRGB((BYTE*)img);
            setBallHue(img);
            LCDPrintf("ball hue:%3d",BALL_LOW+7);
            break;
        case KEY4:
            bStart = 1;
            break;

        }
    }while(bStart == 0);

    unsigned int nLow = 240;
    unsigned int nHigh = 260;
    LCDMenu("+", "-", "RGB", "END");

    vDriveDist(m, 40, 0.8);

    do
    {



        CAMGetFrameRGB((BYTE*)img);
        SetWhitePointBot(img);


#ifndef DONOTDRIVE

        float angle = 0;
        if( g_target == BALL)
        {

            angle = getBallAngle(img, BALL_LOW,BALL_HIGH, &ballY);
        }
        else
        {

            angle = getGoalAngle(img, goalLow,goalHigh, &ballY);




            if (!hasBall(img))
            {
                LCDPrintf("Lost ball");
                AUTone(10000,2);
                vDriveDist(m,40,-0.1);
                g_target = BALL;
                continue;
            }

        }
        float rangeMeters = 0;



        if (angle > -0.39 && angle < 0.39) // ~= +- PI/8
```

```
{

rangeToBall = ranges[ballY];
rangeMeters = ((float)rangeToBall)/100.0;

    turnedAngle = 0;

    if( angle < 0.06  && angle > -0.06 ) //should be 0.07 or 0.06
    {
        if(g_target == BALL)
            LCDPrintf("Ball at A:%2.2f R:%3i\n", angle, rangeToBall);
        if(g_target == g_goal)
            LCDPrintf("Goal at A:%2.2f R:%3i\n", angle, rangeToBall);


        if( rangeToBall > CAMERA_Z_OFFSET*100)
        {
            if( rangeToBall > CAMERA_Z_OFFSET*100+5)
            {
                if(rangeMeters > 0.8)
                    rangeMeters = 0.8;

                if( g_target == BALL)
                    LCDPrintf("Driving towards ball at %2.3fm\n",rangeMeters);
                else
                    LCDPrintf("Driving towards goal\n");
                vDriveDist(m,40,rangeMeters*0.90 - CAMERA_Z_OFFSET + 0.03);
                CAMGetFrameRGB((BYTE*)img);
                if(hasBall(img)) g_target == g_goal;


            }
            else
            {
                vDriveDist(m,40,0.10);
            }
        }
        else
        {
            if( g_target == BALL)
                LCDPrintf("Has ball at %2.3f!\n", rangeMeters);
            else
                LCDPrintf("GOAL!!!\n");

            AUTone(6000,4);
            g_target = g_goal;
        }
    }
    else
    {
        LCDPrintf("Turning %2.3f\n",angle);
        if(g_target == BALL)vAngleTurn(m,TURNSPEED,angle);
        else vAngleBallTurn(m,TURNSPEED,angle);
        turnedAngle = 0;
```

```
            }


        }
        else
        {
#ifdef DRIVE

            float newangle = -1;

            float servoangle;

            SERVOSet(cam_servo,136+45);
            OSSleep(10);

            CAMGetFrameRGB((BYTE*)img);
            SetWhitePointBot(img);

            if( g_target == BALL)
            {
                LCDPrintf("Searching for ball...\n");

                servoangle = getBallAngle(img, BALL_LOW,BALL_HIGH, &ballY);
                if(servoangle > -0.39 ) //0.3927 ~= PI/8
                {
                    vAngleTurn(m,40,(M_PI/4)+servoangle);
                    turnedAngle = 0;
                }
                else
                {
                    vAngleTurn(m,40,(-M_PI/8.0));
                    turnedAngle += M_PI/8.0;
                }
            }
            else
            {
                LCDPrintf("Searching for goal...");

                servoangle = getGoalAngle(img, goalLow,goalHigh, &ballY);
                if(servoangle > -0.39) //0.3927 ~= PI/8
                {
                    vAngleBallTurn(m,TURNSPEED,(M_PI/4.0)+servoangle);
                    turnedAngle = 0;
                }
                else
                {
                    vAngleBallTurn(m, TURNSPEED, -M_PI/6.0);
                    turnedAngle += M_PI/6.0;
                }

            }

            SERVOSet(cam_servo,136);
            if( turnedAngle > 2*M_PI )
                vDriveDist(m, 40, 0.8);

#endif

        }
#endif


        key = KEYRead();
        switch(key)
        {
        case KEY4:
            exit_program = TRUE;
            break;
        case KEY1:

            bTakenpic = TRUE;
            CAMGetFrameRGB((BYTE*)takenpic);
            break;
        case KEY2:

            if(bTakenpic == TRUE) sendRGBData2(port,(BYTE*)takenpic);
            break;
        case KEY3:


            if(bTakenpic == TRUE)
            {
                showHueInterval(BALL_LOW, BALL_HIGH, &takenpic, &highlight);
                sendRGBData2(port,(BYTE*)highlight);
            }
            break;
        default:
            break;
        }


        count ++;

    } while (!exit_program);

///////////////////////////////////////////////
/*------------END OF MAIN LOOP---------------*/
///////////////////////////////////////////////


  LCDPrintf("Program ended");


  SERVORelease(cam_servo);
  vRelease(m);
  return 0;
}
```