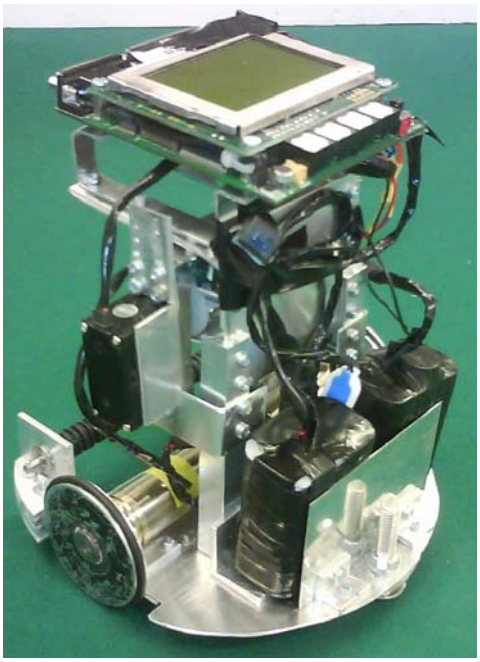
Claes “The LUT” Ohlsson

a robot created in the course
Robotics and autonomous systems



Författare:

Paulino Guerra	670117-0117	guerra@kth.se
Linh Huynh	790307-0485	linh@kth.se
Anders Lindström	820203-0253	anli02@kth.se
David Teppans	820731-0098	teppans@kth.se

Handledare:
Patric Jensfelt
Anders Dovervik

Table of contents

Abstract

This project report explains the work and concepts behind the creation of the robot Claes “The LUT” Ohlsson. During the spring of 2007 a project group consisting of four people spent lots of time with both construction of the actual robot and coding of the software to control the robot. The idea and main goal was to make the robot able to compete well in football games against other robots at a tournament.

Claes “The LUT” Ohlsson competed in the tournament and won a bronze medal after some exciting football games. What made the robot successful is explained and discussed in this report and it also contains detailed outlines of the construction and coding of the robot.

Introduction

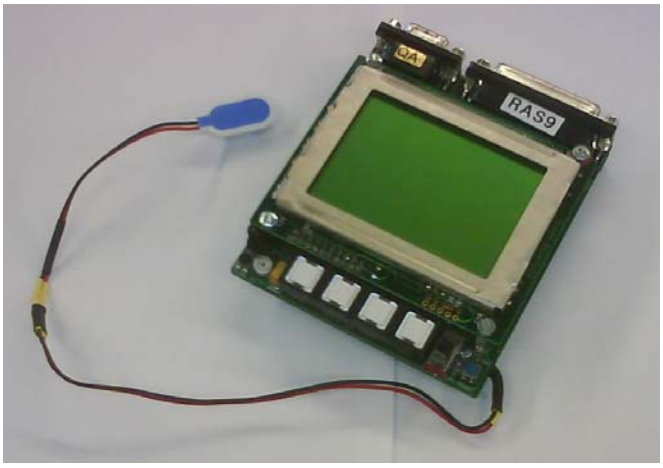
The Team

The team that performed this project was David Teppans, Anders Lindström, Linh Huynh and Paulino Guerra. All project members have attended the computer science program at KTH and chose this course out of interest in robotics. The project members are all genuine programmers with little experience of electronics and construction.

Available Hardware

For the construction of the robot "Claes 'The LUT' Ohlsson" we received the following hardware and got access to these materials:

- An EyeBot controller board
  This board has a 32-bit Motorola 68332 microcontroller processor with 35 MHz. The board also have 2 MB RAM, 512 KB flash ROM, a digital camera interface, digital and analog I/O pins, a built-in speaker under the screen, a microphone and a large LCD with input buttons. The platform has pre-installed the operating system "RoBIOS" (version 6.5). "RoBIOS" is created for the 68332 microcontroller. It provides basic functions to control mobile robots; LCD output (text and graphic), serial input /output, digital color camera input, system timer, multitasking, audio output, basic image processing, loadable user programs, servo and motor control, various sensor input, radio communication and key input (input/output system).



The CPU

- A camera
  The camera is a color camera called "EyeCam". EyeCam is a digital camera to be used directly on an EyeBot or with an adapter on a PC. The camera has auto brightness based in the CMOS technology and it has several different resolutions. We used the resolution 176*144 pixels.

The Camera

- 3 sheets of aluminium

- An R/C servo
  The servo is a Hitec Hs-422.

The HS-422 Servo

- Two wheels

The Wheel

- Two motors
  The motors have in-built encoders that are used for calculating the distance the wheels have moved and controlling its velocity.

The Engines

- Batteries and a charger

Battery pack and batteries

We had access to a small workshop room with tools and accessories to construct the robot.

Tournament

One of the group goals was to participate in a tournament.
The tournament had three parts; the first part was the qualification, the second was the group play and the third part was the finals.

Qualification

In the qualification the robot had two minutes to do as many goals as possible. The referee would always put the ball somewhere on the center line and the robot would start in its own goal. When a goal was made the referee would once again put the ball

somewhere on the center line while the robot continues from its current position. The robot will have to find the ball again to score.

Group play

For the group play, only the robots that got a good result in the qualification round competed. These robots were divided into two groups. Each group consisted of four robots. All robots in the same group played one match versus each and everybody else in the group. The winner of a match got three points while the loser got zero points. If a match ended in a draw, each robot got one point. The two best of each group continued to the finals.

Finals

In the finals the best two robots of each group participated, that means the first and second respectively. The first of group one met the second of group two and the second of group one met the first of group two. These matches are known as the semi finals. The losers of the semi finals met each other to fight for the third and fourth position and in the final the winners of the semi finals fought for the first and second place respectively.

Robot Design

Contruction Rules

There were several rules regarding the construction of the robot. These rules were written to have each robot compete under the same conditions.

One of the biggest restrictions for the design of the robot was that it must fit within an 18 centimeter cylinder. Possible tentacles or sensory antennas do not affect this radius though. This rule in combination with the fact that each team were handed only two wheel motors, limited the design of the robot to such an extent that all robots more or less had the same design basis.

If tactile sensors were to be installed, they must not be electrically conducting. This could lead to short circuits in the opponent robot if our sensors were to touch any of their components.

The robot's most important exteroceptive sensor was the camera. Since the robot's decisions are mostly based on what the camera can see, the opponents may not resemble any of the objects on the football-field, apart from just an opponent. There may not be any colors of shapes that look like a football or perhaps a goal. This ultimately led to the robot being aluminum colored with all parts not being the body-frame to be black. Also our robot had a light array for diagnostics, which had to be removed during competition, since it could resemble the red color of a ball and the yellow color of one of the goals.

The team wanted to avoid specular reflection on the football, since this often resulted in almost an entirely yellow color when the ball was close to the camera. The group thought of the idea of an onboard light source to light up the surface of the ball. This was to avoid the yellow light from the fluorescent lights above the football field to interfere, and change the color of the ball. But a light source of any kind could also interfere with the opponents' camera, and this could of course not be allowed.

To effectively control the ball, it would be a good idea to have the robot engulf the ball, and hold it within the robot. This is not allowed since the opponent can not take back the ball. Therefore there is a clearly specified rule to handle this situation. This rule stated that no more than 25% of the balls diameter may be within the robot at any time. This meant that without a contraption that makes the ball spin backwards towards the robot, it will be very difficult to turn, and still maintain control of the ball.

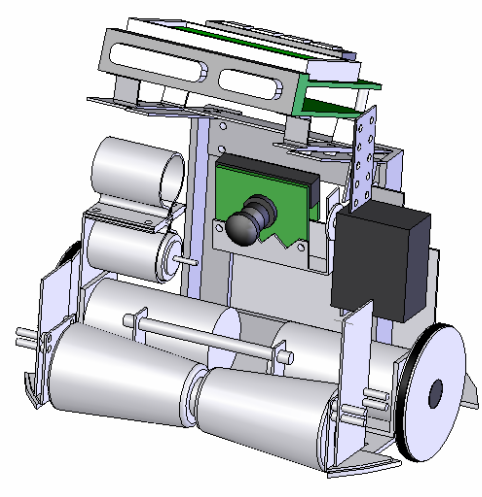
Design Idea

When the first designs of the robot was created, obviously all the previous specified rules were needed to be taken into account. The project group realized immediately that these rules limited the creative process greatly. Just the fact that the robot has to fit within a circle with an 18cm diameter, limited the alternative robot designs. Also, since differential drive was the most preferable locomotion method, almost all robots were of circular character. This was due to the fact that utilizing maximum surface area on the robot to optimally be able to fit all the equipment and electronics was of importance. Furthermore limited materials and limited availability of tools constrained the construction.

A model of the imagined robot was created in SolidWorks[1] to get an idea about the layout of the robot. The general opinion was to create a stable robot with good weight properties to easily, yet robust, move across the football-field. Also the concept of a roller in the front of the robot was purposeful. The roller keeps a backspin on the ball which enables the robot to maintain control of the ball even if turning or reversing.

Construction

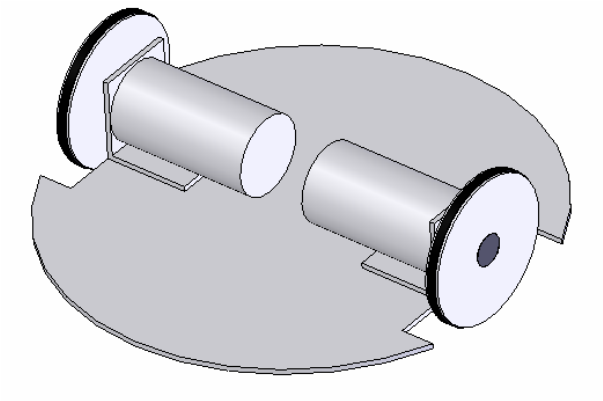
The provided materials affected the modeling in such extent that complete reconstruction of the model was performed several times. The model was also of help to create the individual parts of the robot.



SolidWorks model of the robot

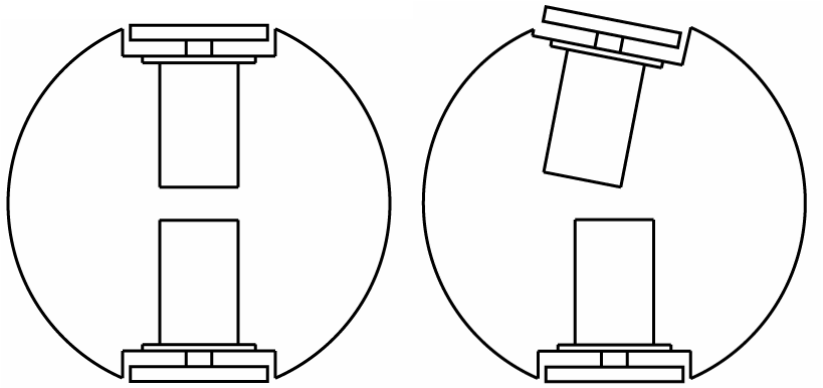
First the circular-base concept of the robot was decided, after that the discussion about whether to put the wheels above or below the robot-base took place. We observed that if the robot-base where below the wheel-base; the robot would get more stability as well as more volume to construct on.

[1] SolidWorks 2005, ©2007 SolidWorks Corporation



The base with mounted engines and wheels

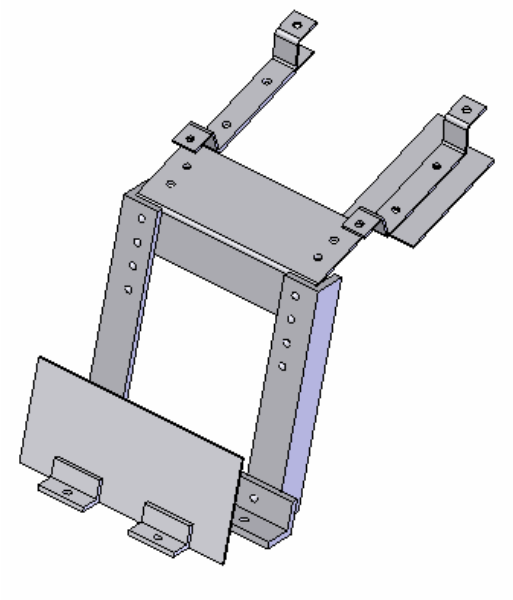
When constructing the physical drive system of the robot, the team realized that all measurements needed to be very accurate. A misalignment of the wheels or a non symmetric layout of them would result in odd movement with a risk of necessarily more difficult algorithms to compensate for this.



If the wheels are not mounted collinear the robot might not drive as expected

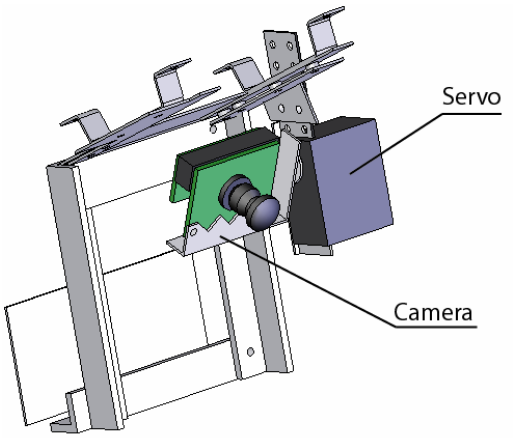
A small alignment tool was created to more accurately be able to fasten the engine attachments correctly.

To avoid cable and robot mechanism congestions, the decision of a slender and relatively tall CPU mounting was considered a good idea. Furthermore the need for component proximity was determined to be a vital property since it would facilitate easier connectivity of cables and peripherals. The fact that the engine cables were of limited range put the constraint on the CPU height above the robot-base. Since the project group did not want to create extension cables for the engines, due to the fact that if this is not done correctly it could result in engine failure or damage, the CPU mounting was designed with variable height.



The CPU holder

One of the most essential aspects of good robot-performance is the camera. The complete reconstruction of camera mountings and camera position were carried out several times. This was done since the rollers changed frequently and new ideas were thought of regarding field of view and camera angles.
The first idea was to utilize a servo to have the camera turn horizontally. This would facilitate searching for objects without turning the robot. Later the project group realized that there were no efficiency gain regarding turning the camera in respect to turning the entire robot. The decision was made to have the camera turn vertically to be able to adapt the field of view in aspect of distance to an object. First the camera was more or less just mounted randomly on the provided servo, which resulted in a flipped image since the camera was up side down. When this was corrected the discovery that the higher the camera is mounted, and the further back it is located, the better the resulting view of the field would be. Also when the robot moved, small vibrations occurred. These propagated through the robot which made the camera image blurry and distorted. A more robust and stable camera-mounting solved this problem.

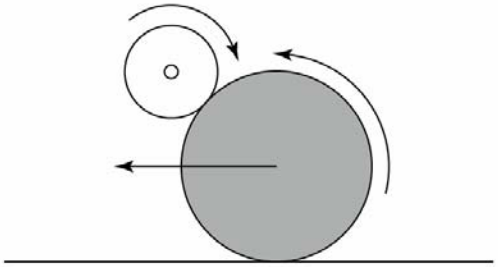


The camera mounting with servo

The construction went fairly smoothly, apart from the fact that finding a suitable roller was identified as a major problem. This resulted in uncertainty of the final robot design, which led to uncertainty when writing the software. The software should obviously be more or less written according to the robot-design. The roller problem is presented in more detail in the chapter *Rollers.*
Several design features differs from the final robot design. This is due to the fact that when a problem was identified, an alternative solution had to be found. Both roller mountings, roller engine mounting and CPU mountings differs somewhat from the original SolidWorks Model.

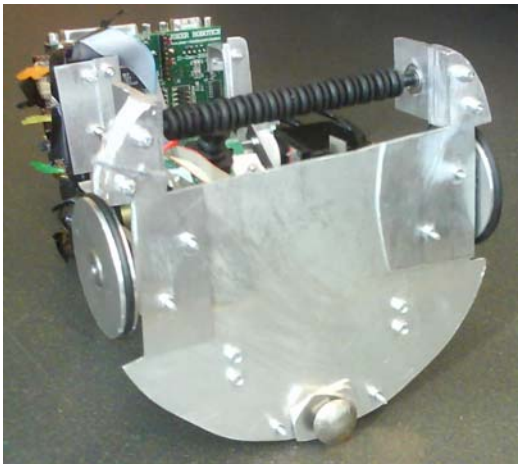
Rollers

The concept of a roller on the robot, gives the robot the functionality of maintaining the control of the ball when turning and reversing by applying backspin on the ball. Materials for a roller should have good grip against the plastic golf-ball. The roller itself should be as rotational symmetric as possible to avoid bouncing of the ball against the roller. If this were to occur, only a small fraction of the roller surface would touch the ball and thus the roller will loose its backspin ability. The roller axis mountings are equally important since if they begin to vibrate during runtime, the same effect will occur.



The principle of backspin on the ball

At an early stage, the team noticed that the robot base were a bit too thin to support a roller. It was also possible that the roller mountings would got bent if the robot were to collide with something. The project group installed reinforcements to resolve this issue.



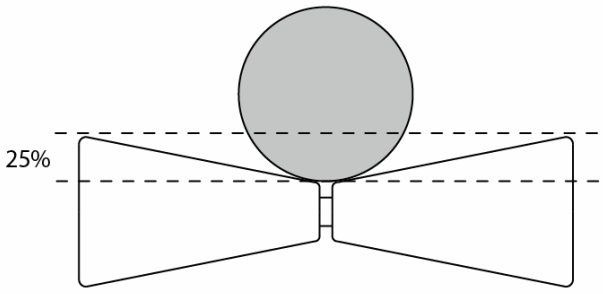
The reinforcements under the base plate

When starting designing the robot, a possible roller material were found fairly fast. This unicorn of rollers consisted of two hard-rubber door-stoppers. The material was very hard with very good grip against the golf ball plastics. They were slightly conical which made the ball roll inwards towards the center of the rollers, as well as perfect rotational symmetric. Not long after the installation did the project group notice that these rollers did not work at all. They were too hard and vibrated horrendously. This was a result of the fact that the inner circumference of the roller only allowed for a wide-diameter axis. A wide axis can be difficult to mount with perfection and low friction. The vibrations of

these rollers were so immense that the robots nuts and bolts were loosened at a steady rate. Also the rollers were mounted to low, which did not force the ball into its grip. The concerns of the possibility that our robot would tare some other robot apart, or roll up onto it and thrash its construction, made us take the decision to change this roller.

The two door stoppers mounted on PVC tubing

The conical property did also show to be worthless, since it requires a very steep angle to maintain control of the ball whilst turning. Since there was a limitation on how much of the ball which was allowed inside the robot periphery at any time, there were no room for edges on the side to stop the ball from rolling out on the side.



There is no room for stopping-edges on the side if the ball starts to roll sideways

Below is a brief explanation and walkthrough of different rollers that were mounted and tested. None of these following rollers worked as intended.

This first roller consists simply of PVC plastic tubing. It did not have any traction at all and it vibrated quite heavily whilst rolling. It did not allow for backspin on the ball.

PVC tubing

Vulcanized tape coating on foam cylinder mounted on PVC tubing. Good grip, unsymmetrical, vibrating.

Foam cylinder on PVC tubing. Moderate grip, symmetrical, vibrating.

Tennis racket hand grip tape. Good grip, unsymmetrical, vibrating. Too hard.

Electric tape on foam cylinder. No grip, unsymmetrical vibrating.

Soft foam on steel rod. Good grip, unsymmetrical, too soft.

Smelted glue on steel rod. Good grip, symmetrical, no vibration. This was a good candidate, but it was too hard.

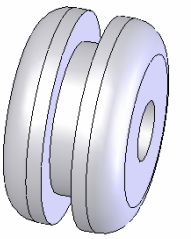
Big fishing jig, threaded on steel rod. Very high grip, no vibration, unsymmetrical.

Small fishing jigs threaded on steel rod. Very high grip, no vibration, unsymmetrical.

Double sided tape on PVC tubing. Due to rule violation where the ball would get stuck permanently on this roller, it was not tested further.

Vulcanized tape on steel rod. High grip, symmetrical. Too hard.

When the team tested the last roller for grip and traction, the team figured that a satisfactory solution was found. The rubber band between the roller-engine and roller was noticed to swing along the axis on the roller whilst operating. This often led to that the rubber band came loose, which meant that the roller didn't work anymore. To resolve this, a small rubber gasket with a groove was fitted to the roller, so that the rubber band should always stay in place.



A small rubber gasket which normally is used to splice tubing of different diameters.

When the gasket was fitted, the project team noticed that as soon as the ball touched the bus ring, the ball got a very fast and stable spin. We then created a roller made up entirely of small rubber gasket.

The final roller with 15 rubber gaskets

And this was the winning concept. A small, high-grip, rotational symmetric, rubber roller, that did not lose the rubber band when operating. This was the roller that was utilized during the competition.

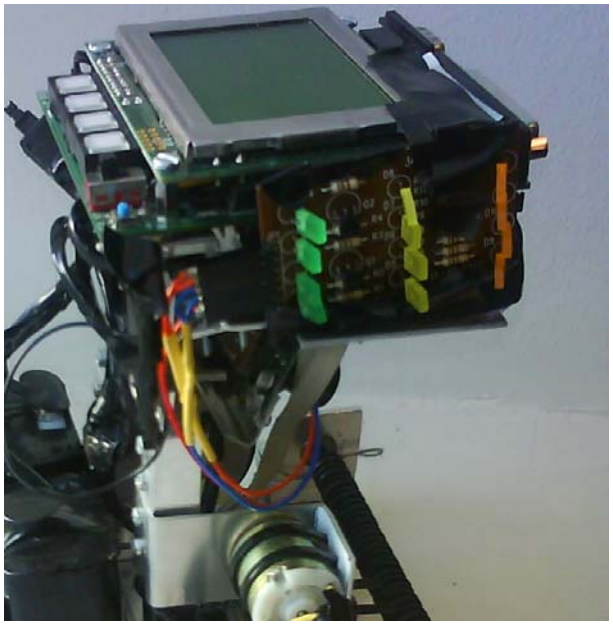
Regarding the roller engine, we chose a small and fast engine. The engine was driven by 6 volts to get sufficient spin on the roller, as well as maintaining the force of the spin when the roller touched the ball. The engine was operated by a manual switch which turned it on and off.
When the rubber band broke or came off, the engine became unloaded, which made it smell like burnt plastics. Therefore the project group figured that if the power were not turned off fast enough when there was no rubber band, it was a high possibility that the engine would fail. This was a risk the team was willing to take.
The rubber band was first a regular rubber band with good elasticity. This form of drive band tended to snap fairly often, which often resulted in the pain of fitting a new one. Later a rubber hair band was fitted. This type of band consists of a rubber band coated

with elastic wool cloth. Only problem was that a knot had to be made to tie the ends together, which sometimes could kick away the ball from the roller.

All the materials for the rollers were bought at the “Clas Ohlson” store. The project group traveled to this store no less than ten times looking for materials. We went to Clas Ohlson so often that we finally considered Clas Ohlson was our saviour regarding the construction of the robot, and therefore we baptized the robot “Claes Ohlsson”.

Leds

When the robot was placed on the football field, with the objective to score, it of course did not always do as expected. Sometimes it drove away from the ball and into the wall, and sometimes it found the ball and drove it into the wall. The project group identified that it was difficult to read the debug output on the LCD screen. This was due to the fact that when the robot turns, you needed to run around the field to be able to read on the screen. Also it was difficult to provide relevant feedback on the screen. Lots of printouts on the screen will also affect the speed of the algorithm.
To solve this conundrum the team constructed a light diode array. It consisted of three colors which would display which state the robot currently was in.

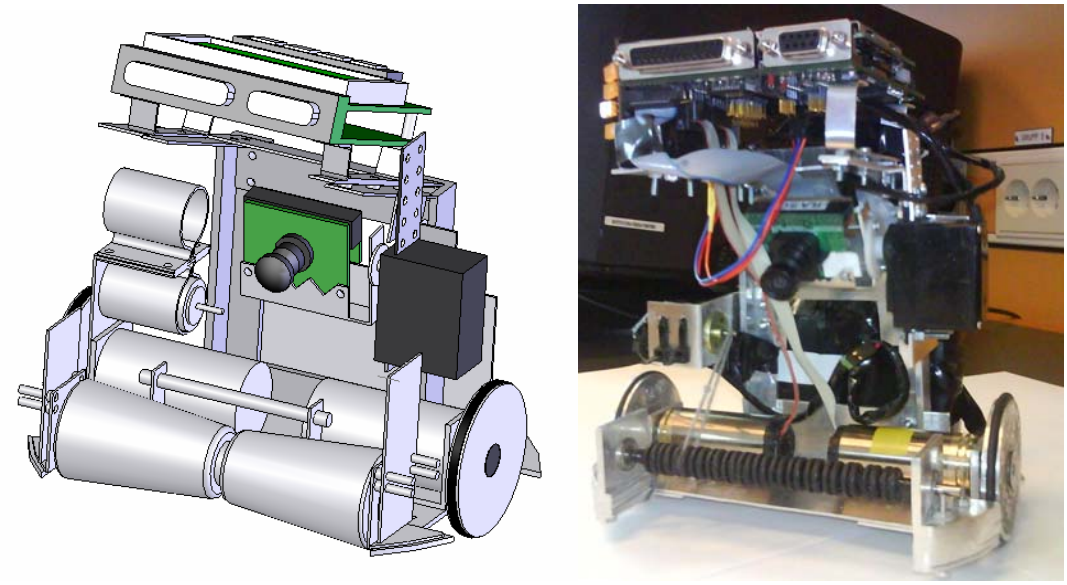


The diode-array which displayed the different robot states

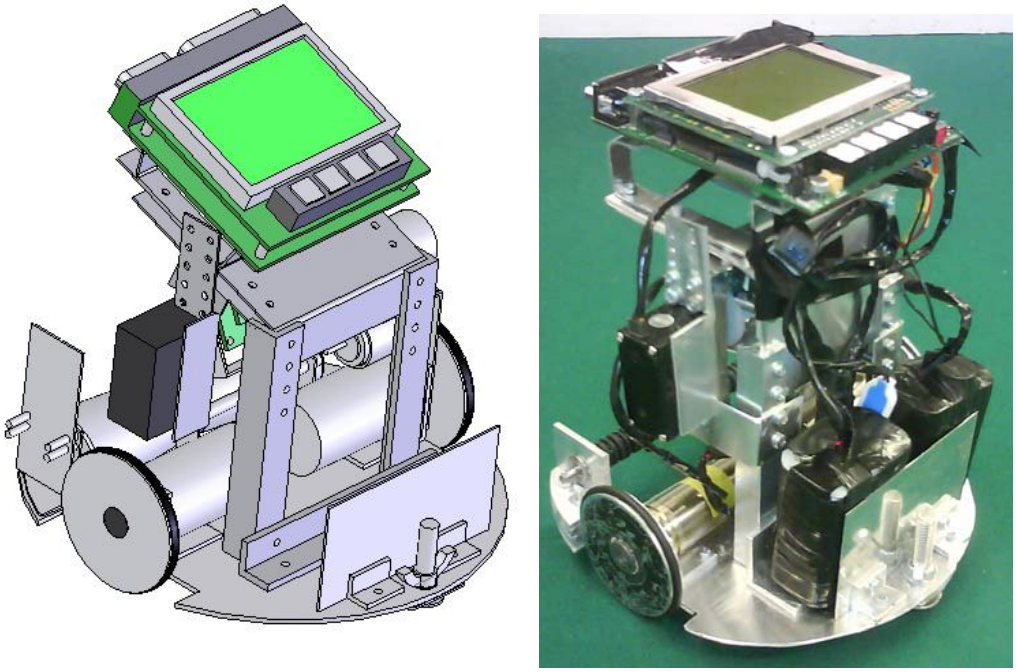
These lights helped tremendously during testing and debugging since they clearly showed which state the robot was in. They were removed during the competition to avoid confusion for the other robots.
The diodes displayed if the robot successfully saw the target goal, if it had identified the location of the ball and if the ball was considered as held by the rollers.
In retrospect this was one of the major time savers whilst writing the code.

Finished robot

The finished robot did resemble the original idea, even though plenty of features had been changed. The project group considered the construction to be both robust and capable of performing its task. Below is a comparison between the original model and the finished robot.



Front view. The rollers has changed, as well as the roller engines

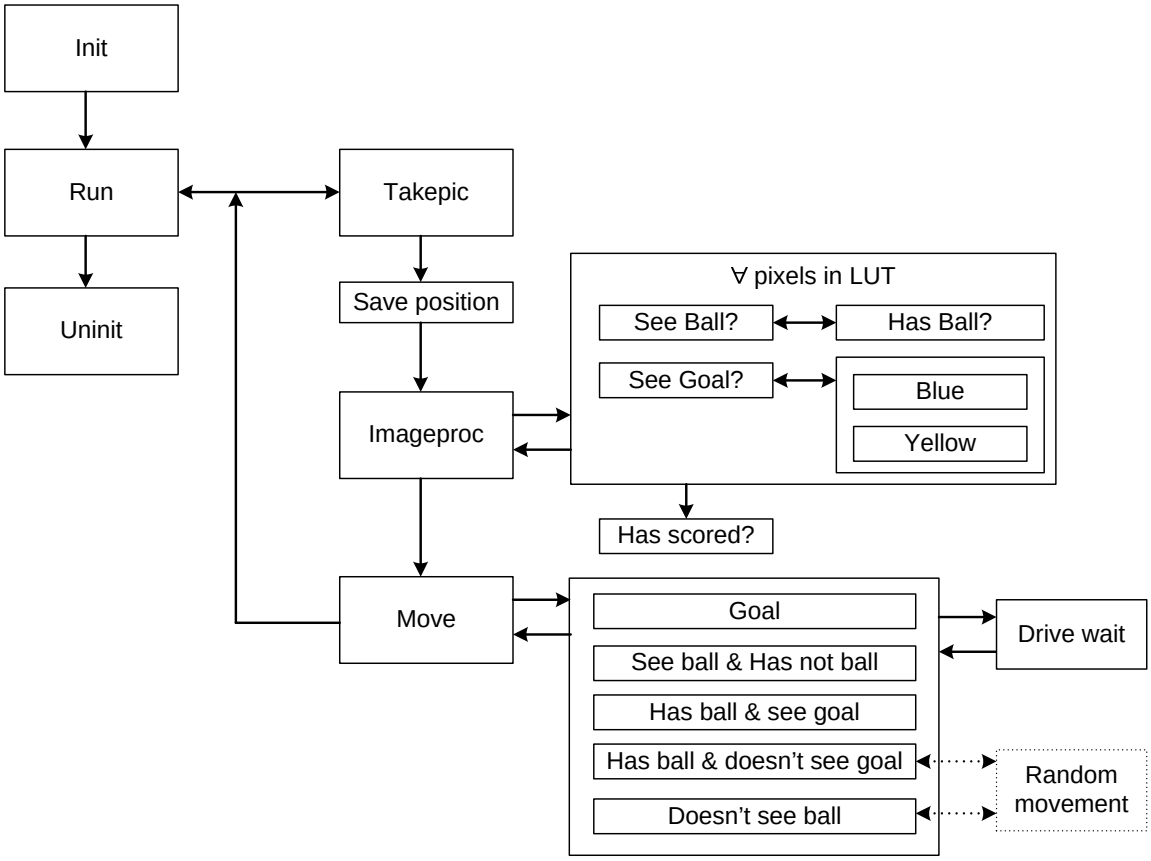


Back view. The model resembles the robot closely.
The CPU mounting has been heightened and tilted downwards.
Also, the cpu holding bracket was changed last minute due
to violation of the 18 cm diameter rule.

Robot Programming

General idea

The project group created, at an early stage, a flow chart over the general software design. Below is a brief overview of the design.



- **Init**
  Sets up lookup tables and initializes the camera, servos and engines.

- **Run**
  Main loop, listens for key inputs.

    - **Takepic**
      Takes a picture and stores it in memory

    - **Save Position**
      Since the image processing might take time, the current position and angle is stored so that the movement decisions can be compensated for any movement that has occurred during the image processing.

- **ImageProc**
  Takes an image as input and sets different Booleans given what the image contains.
  Different Booleans are: HAS_BALL, SEE_OUR_GOAL, SEE_TARGET_GOAL, SEE_OPPONENT, SEE_BALL, GOAL
  Depending on what was specified as target goal (blue or yellow) in the init phase, different target_goal routines will be carried out.

  - **∀ pixels in LUT**
    For all the X,Y coordinates stored in a lookup table, a classification is made. Each coordinate is tested to see whether it is a ball a goal or nothing important.

  - **Has scored?**
    If the ball and goal is seen, an identification whether the ball is located inside the goal is made.

- **Move**
  Given the different identifications made during the image processing, different movement patterns will be carried out. Each decision is compensated for the movement that has occurred since "Save position".

  - **Goal**
    A victory sound is played, and the robot returns to home.

  - **Has ball & doesn't see goal**
    The robot will begin searching for the goal in such a way that it still maintains control of the ball.

  - **Has ball & see goal**
    Takes out a heading towards the center of the target goal.

  - **See ball & Doesn't have ball**
    Takes out a heading towards the ball. Depending on how far the ball is estimated to be, different speeds are applied.

  - **Doesn't see ball**
    If the robot can't see the ball, it will try to find it by rotating around its axis.

    - **Random movement**
      If, after a complete turn of the robot, the ball is still not visible, a chaos movement is made to change the location of the robot to perhaps get the ball in sight.

- **Drive wait**
  When the robot has decided how to move, the code execution is halted until the movement is complete. If the robot is found to be stuck, the previous movement decision is cancelled and a different movement combination is made to escape.

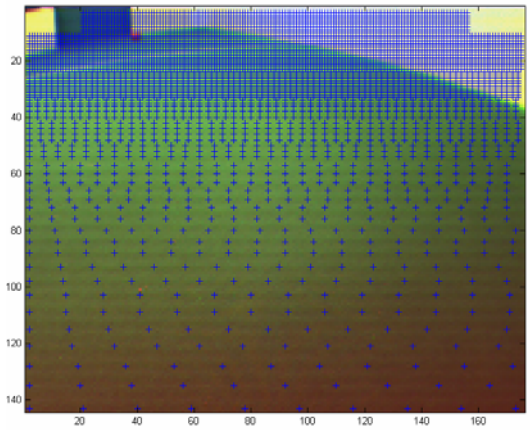
- **UnInit**
  Engines, camera and other necessary allocations are released.

Image processing

The robot was built with only the supplied camera as an external sensor of the environment, so the image processing was very important to make the robot behave as intended. Objects that need detection are the ball (red golf ball), the goals (blue and yellow in color) and the opponents that are dressed with a purple band of paper around them. All the objects had different detection routines programmed for them, the way the robot detects the blue goal is very different from the way that the ball is detected. Due to lack of time the opponent detection routine was discarded.

Sampled Image

Images from the camera had a resolution of 176x144 pixels. It could use lower resolutions for faster handling but the larger size was chosen. The reason for this was mainly that it felt better to have more pixels available when searching for the ball when it was far away. Searching every pixel in the image is usually an unnecessarily cumbersome task. For example it's highly likely that many close by pixels are red from the ball if the ball is close the robot. Instead of searching through all 25344 pixels the algorithms used a lookup table (LUT) called samples lut. This lookup table contained 5143 pixels from various positions in the image. The distance between used pixels were lower depending on how far up in the image they were, since objects become smaller the further away they are. Also the pixels in the upper corners of the image were discarded, this because of the



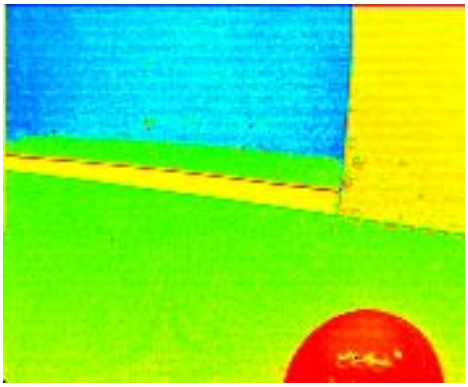
The lookup table for pixels in images, the samples lut

cameras down viewing angle that made it sometimes see areas above the walls of the football fields in the corners.

Color detection

Most algorithms with purpose of detecting various objects on the football field used color values in some way, and they usually work something like this; detect a pixel with color value within a specified interval then check that an average color value over close by pixels are within the interval as well. How near the close by pixels need to be or in what form they are taken from around the first pixel is varying both with what objects are being detected and where in the image they are detected. Color values can be directly extracted from the image taken by the camera. The color values of a pixel are represented in the image as a triple of Red, Green and Blue commonly abbreviated as RGB. RGB values are great for some sorts of comparisons, like how much more of the red component is represented compared to the blue component, but is unsuited for others. How to represent a range of colors from dark red to an orange shade of red is not easy in the RGB color space. Also, if the luminance changes then the RGB values will proportionally change to higher or lower values, making the comparisons even harder. Here is where the HSV (Hue, Saturation and Value) color space becomes very useful. The hue represents the color type and is usually in the range between 0 and 360. Color types have values ranging from red towards orange and yellow, then towards green etc. Saturation means the color intensity and value represents the brightness. The hue is a great tool to effectively analyze images in search for certain colors. Intensity and brightness does not really affect if the pixel represent the ball or not and those parameters also varies greatly with lighting and other settings around the football field. Hue values are therefore, at least in theory, the perfect color parameter to analyze in color detection algorithms.

A lookup table was created that converts RGB values into its hue value since the hue value requires some calculations based on the RGB values from the image. To avoid a LUT with 256^3 elements, the RGB values are discretisized down to 16^3 different colors.



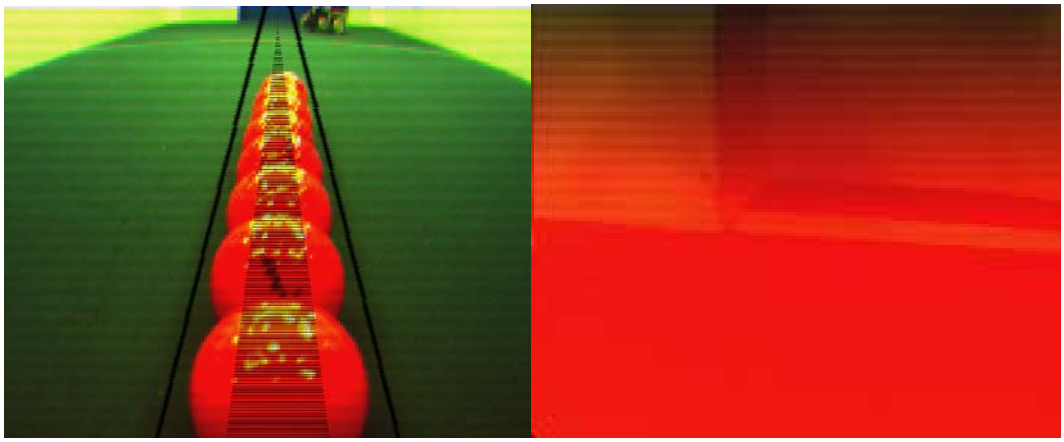
Hue values of a typical RGB image from the camera

Each object has a defined hue interval shown in the picture.
From top to bottom; ball, blue goal, yellow goal (without edge detection algorithm).

Ball detection

The ball detection depends on its distinct red color, few other things on the field are frequently getting as many red pixels close to each other. Color values in RGB and HSV were used to create color intervals that hopefully only the ball occupied. The ball detection routine used shape recognition in a processing time efficient way, a bit simplified from what could have been done with more processing power. A ball shape was required to have a certain extent on the horizontal axis, an extent that varies in size depending on how far away the ball is (in effect what y-value the ball pixels have).

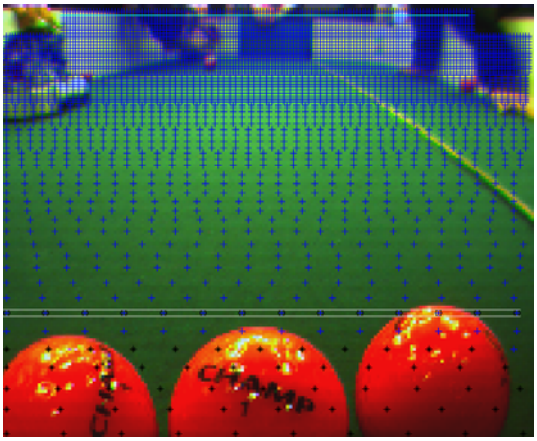
Why did we not just require a minimum horizontal extent value for the ball? Reason for this was more general problems with the camera. Sometimes the camera could produce an image containing a lot of extra red pixels (or other colors for that matter) that would make the robot see balls on nearly any pixel. The maximum size of the ball made these images easier to cope with. The vertical extent of the ball varied significantly as well with its y-value, for example a ball is red and round in shape while being in lower parts and up until the middle of the image, while being more like a thick horizontal line of red pixels higher up in the image. In the vertical extent the ball was just required to have at least a y-value dependant value of ball pixels.



Maximum and minimum ball horizontal extent.
This avoids detecting balls in corrupted images like the one to the right.

It is important to detect if the ball is close enough to the robot so that it’s possible to affect the balls motion with movement of the robot. This concept we named has the ball.

A difficulty with trying to detect whether the ball is actually in possession of the robot was the low height of the camera. The camera's low height resulted in that small changes in the ball's position gave very small changes in the y-value of ball detection, especially when the ball was close to the robot. If the camera had been mounted higher up it would be easier to detect small changes in the balls position, if it was actually against the rollers or two centimeters away. This difficulty can seem minor and insignificant but it affects how well the robot can act with the ball. When the robot turns, if the has ball routine performs its duty correctly, then the ball will follow the robots motion and stick with the turn. If the routine reports that the robot has the ball even if it is a small distance away, then it would lose control and sight over the ball in a few seconds of turning motion. So how was this algorithm defined then? There were many ways to solve this problem including adding different sensors, like IR detection just outside the roller, or using the image in clever ways. The project group chose a solution that acted on the information of the find ball routine. If the ball was found below a certain limit line (a y-value) in the image then the algorithm checked whether the balls red pixels could be found on the limit line. If the balls pixels intersected the limit line then the ball was out of reach and therefore not in possession. So if the ball was detected strictly below the limit line then and only then was it considered to be in possession. This limit line posed some constraints on the construction and calibration of the robot. For example if the cameras position or viewing angle was disturbed from contact with another robot then the has ball routine could be totally broken until the camera was reset into its regular position. To ensure that the robot could cope with this problem the camera was mounted on a well secured servo that could reset its vertical angular position into a known state. The servo was periodically reset into this position during robot operation.



The balls to the left is considered "has ball" but the ball to the right is not.

Blue goal detection

Detection of the blue goal was based strictly on color values, in RGB and HSV together. Since nothing else on the field is blue in general the algorithm was straight forward to code. The top line of pixels in the images was scanned for more or less blue pixels. After some blue pixels were found then the extent of the goal in the horizontal axis was to be

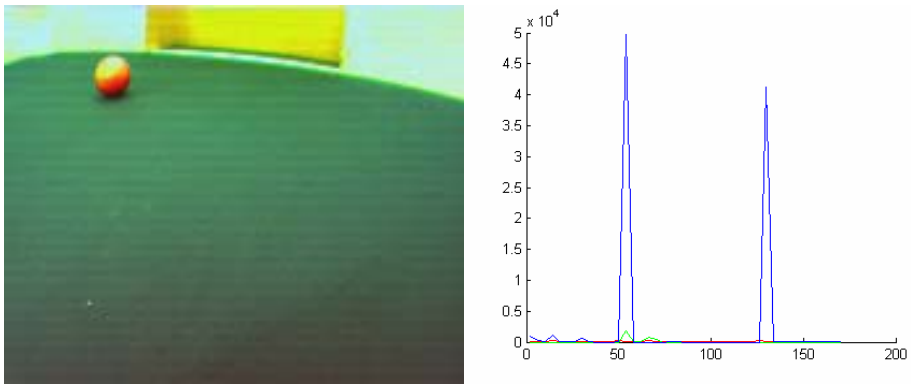
determined, which was done by searching in both directions until no more blue pixels could be found. This was important for how the robot would drive towards the goal so that it didn't hit the nearby goal posts or walls.

Yellow goal detection

The algorithm for detecting the yellow goal did undergo some changes during the project. It started out as outlined above for the blue goal but that solution was not satisfactory for various reasons. Main reason for its failure was that the walls could get just about the same color values as the goal in nearly any image (not dependant on robot pose on the field). The project group tried many different color value ranges, in both RGB and HSV, without ever getting rid of the problems, so a new approach was implemented. Instead of depending on the exact color values, the color value changes were used instead. The algorithms content can most easily be explained as regular edge detection with some color checks so that no irrelevant edges were used. Color value ranges were made much more liberal to cope with anything from orange to white color values. But also, instead of just checking the pixels colors, the image had to have distinct edges of color values.

Edges, which could be viewed as the goal posts, were calculated by measuring squared differences in RGB values between adjacent pixels on the x-axis. How adjacent the pixels had to be was set to a maximum of three pixels. The edges then had to have a quite large minimum extent on the y-axis with the whole extents mean square difference in RGB values within a certain range. The range that was used to detect the goal posts was measured by testing lots of images taken by the camera in various light settings and poses. Whenever two goal posts were found then the goal was naturally assumed to lie in between them.

This algorithm was superior to the previous algorithm although some problems still exist with various "imaginary" goal posts found in corners or the walls.

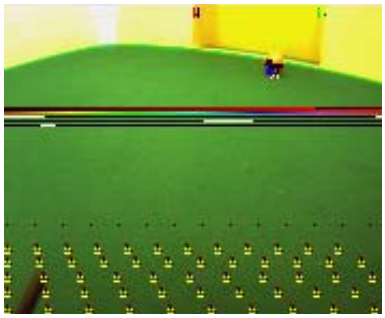


The two edges in the left image generates a high squared distance in the RGB color space, shown in the graph to the right

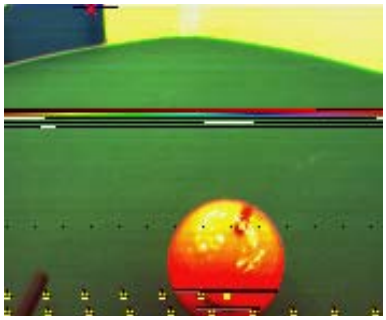
Diagnostics

To effectively analyze the classification algorithms lots of pictures were taken with the robot and sent to the computer. These pictures were modified and plotted with pixels in

different colors depending on the classification. For instance, blue pixels were inserted where the robot found the ball. Below is a brief description of how this worked. Also a small taste of how the algorithm works is provided.



The algorithm has found a yellow goal, and illustrates this with a green and a red vertical line. This can be seen in the image to the left. Also the algorithm has detected the ball. This is shown with a blue square. But the image displays more than just the location of a detection. Valuable information is stored for diagnostic purposes. RGB, HUE, object width, minimum size, average HUE are some of the things that are embedded in the RGB values of the pixels adjacent to a classification. In this manner it is easy to sample the pixels when viewing the image with and picture viewer and extract interesting information. This is why there are black, yellow, red and other colored pixels in the vicinity of the balls detected position.
In the lower part of the image are the sample pixels, illustrated with yellow corners, that has been tested for “Has ball”. Apparent is that no “Has Ball” has been found. These pixels are also fitted with diagnostics pixels close by.



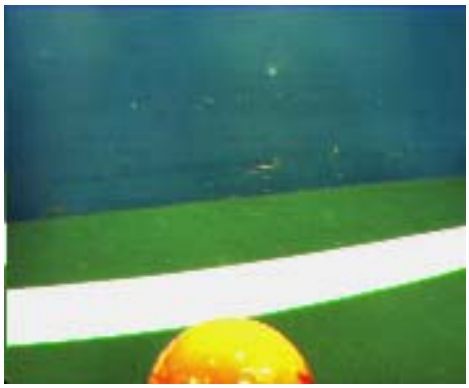
In the picture to the right a detection of the blue goal can be seen. It is illustrated with a black line and a red square, where the red square represents the center of the goal. Obviously this detection did not function optimally, since it is too far to the right of the goal-center. In the lower part of the image the “Has Ball” detection routine can be viewed. The algorithm has scanned the image from the lower left corner, writing yellow dots along the way. On the 7:th try on the first line it detected a potential ball. This detection was discarded since the detected width was not wide enough. Gray and black lines illustrate how extended the detection is horizontally. Since no ball is considered found, the algorithm continued and detected yet another candidate on the line above. This time the detected width did match the criteria and the ball was considered found. A filled yellow square illustrates the horizontal center of the ball. But this ball is not considered “Has Ball” since it touches the black dotted limit line above. Since a ball was found, there is no need to continue, and the algorithm stop.

The horizontal lines, in the middle of both pictures above, are an illustration of how HUE and RGB are calculated by the algorithm. Also in which intervals each object resides. See chapter color detection for a similar explanation.

Difficulties

Even though the ball is usually red, the colors change to a large extent depending on external light (like sunlight from windows or other kinds of light sources) and from what

direction the ball is viewed upon. It was apparent that just checking for red pixels would not be satisfactory. The main problem was that the ball became more and more yellow whenever reflections from other light sources reflected into the camera. This problem became more and more evident when the ball came closer to the robot.

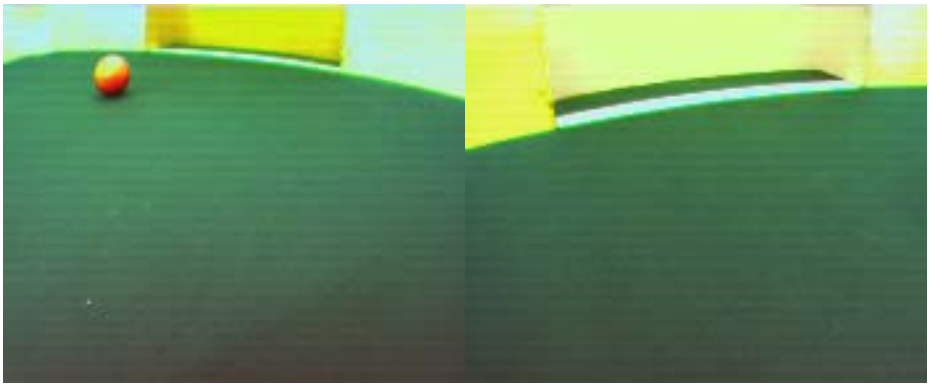


The ball sometimes turns yellow when close to the camera

To cope with the yellow balls some modifications was made to the ball detection algorithm to allow more yellow pixels closer to the robot, so depending on the y-value that was examined more or less yellow pixels were accepted.

Ball detection could also fail if other things on the field generated red pixels in the images; wires from other robots, the transition from green surface to blue or yellow goal and sometimes even the irregularities in the white walls. Shape detection for the ball algorithm effectively removed most of these errors.

The main problem with the early version of find yellow goal routine was that the walls usually could be seen as belonging to a yellow hue value. With the camera's tendency to often generate images differently over time, depending on where the robot are and the external lights, it became apparent that the yellow color could not safely be used alone to detect the yellow goal. When the robot came close to the yellow an even stranger color setting appeared; the yellow goal which often had a near orange shade of yellow changed its brightness and thus became more yellow and white, while the goal posts and nearby walls became more orange. This resulted in that any color dependant algorithm that we could write would actually detect the goal outside the real goal at both sides and try to steer clear of the detected wall in the center. A solution to this could be that the robot's movement should keep its position away from the goal. This would mean that the robot would have to kick the ball into the goal instead of pushing it into the goal. Another solution could be to actually ignore the last images taken when approaching the goal and act on earlier information.



Notice the odd colors in the right picture.
The walls outside the goal are more yellow than the goal itself.

The project group noticed that the camera produced some strange images sometimes. Reasons for these images varied but some regularity could be found among them. When the robot had low battery power then the images became all gray and without specific color information. This was considered a major problem for the robots functionality in the tournament because it had to function well over many matches for a considerable time. The project group bought additional batteries to ensure that the robot could have fully loaded batteries at each match.

Other interesting images appeared depending mostly on the light settings in the robot laboratory room (but sometimes for no apparent reason at all) where the images sometimes got overly colored with red, purple or yellow. The major problem with those pictures, which were rather frequent, was that the ball could be detected in many erroneous positions on the football field. This was fixed with the introduction of the shape detection algorithm for ball detection explained above.

Image taken with low battery to the left
Strange color values for no apparent reason in the middle and right images

A problem with image processing in general has been the calibration of color value ranges for different light settings. Light settings change mainly with how much sunlight is added to the football field and how many, and what type of, external light sources are active. The project group mostly calibrated the color value ranges at night without any additional light sources. Light sources active then were only the fluorescent lights that were mounted on the football field. The fact that the calibrations were done this way ensured that no conflicting calibrations for different settings were used so that for example the ball was seen during the night but the blue goal only during the day. Luckily enough these calibrations worked good most of the time, but did real poor at some settings like morning sun light through a back window. Since the tournament was held in an external light free environment the calibrations worked perfectly when they were needed the most.

Locomotion

The robot was built with both wheels attached to the robot body on opposite sides, mounted along an axis running almost through the middle. The axis was put a bit in front of the middle so that the center of gravity would be behind the wheel axis. A screw was put in the back part of the robot to give it another point of contact with the ground to allow static stability. The wheels are both individually controlled from the program enabling a differential drive locomotion.

Differential drive is a commonly used locomotion technique which, at least for round shaped robots, have a high maneuverability with small problems of getting stuck. Another good feature of differential drive locomotion is that it is relatively easy to control the motion of the robot.

The motors of the robot uses encoders to send information to the EyeBot platform about how far they have turned. These “ticks”, sensed at a high enough resolution, can then be used to calculate how far the robot has driven and what its pose is. EyeBot comes with an api called “V-Omega Driving Interface” that can make use of this information to control the robot. Many functions of high level are available in this api like DriveStraight, DriveTurn that take as input quantities like what speed and how far. To ease up the controlling program the code can run functions as DriveWait to wait until we actually have driven as far as wanted.

Difficulties in V-Omega Driving Interface

During programming against the V-Omega Driving Interface and testing how the robot behaved some problems and difficulties were identified.

There are essentially two ways to move the robot using the V-Omega Driving Interface; The SetSpeed function or DriveX functions (where X is either Straight, Turn or Curve). DriveX functions have ending conditions which can be checked against with DriveWait (or DriveDone) while the SetSpeed functions does not. For definitions of the mentioned

functions read the declarations in the Eybot api. Typical pseudo code for using the DriveX functions are:

```
DriveStraight(input parameters for how long and how fast)
DriveWait()
```

How this difference in semantics affects the robot control is that DriveX will always end up in a condition, which at some earlier time seemed like a good idea. If the program is really slow, meaning the frequency of decisions is slower, then the robot will at least end up in poses which have been decided to be good at some earlier time. If the robot instead uses the SetSpeed function then the suitability of the robots pose will be much more dependant on the execution speed of the program. As an example, if the programmers think that a speed of 0.5m/s is a good choice when driving against the ball then if the program would, after some modification, become half as fast then the robot would have gone twice as long before next decision. This can of course be a minor problem if the ratio of distance traveled and program speed is relatively small. Because of this issue the control program did get dependencies between different parts of the code structure that were not intended. For example if the image processing for determining if the robot had the ball was changed to be more specific and demanding the drive against goal routine could break up resulting in loosing the ball.

According to this argument, why would one use SetSpeed instead of the DriveX functions? The reason for this robot was that the SetSpeed command generally resulted in smoother motion. Smoother motion is essential for keeping the ball while moving (especially without roller).

The project group used a mix of these functions; SetSpeed when in control of the ball and DriveX commands while driving around without possession of the ball.

A problem encountered with the DriveWait command was that it would wait indefinitely until the wheels have moved the specified distance. For instance, if the robot is trying to drive straight into the wall a certain distance then it will just sit there for a long time (or indefinitely if the wheels are not slipping any on the surface). The alternative to DriveWait is to write a loop with semantics as:

```
while (Not DriveDone()) { if(Stalled()) signal stalled }
```

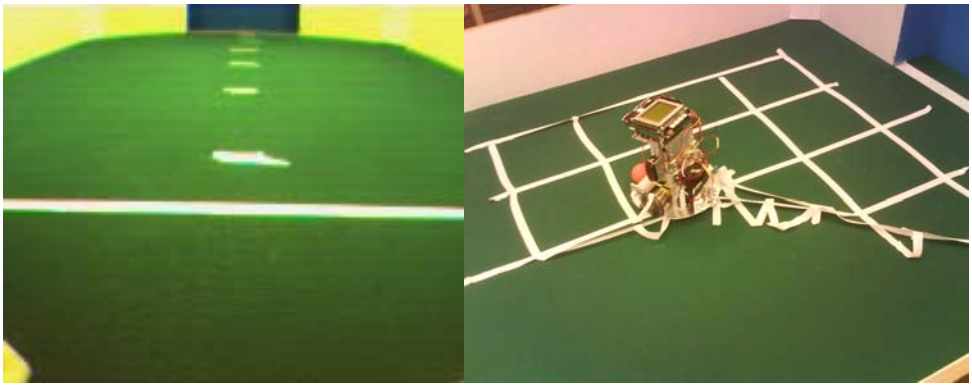
This solution would be satisfactory if only the stalled function did work properly which it did not. Without an easy check against whether the robot is stalled it became harder for the project group to use the V-Omega Driving Interface properly. Reasons for why the Stalled function did not perform correctly could be that the motor encoders were not sending the ticks properly but it was not investigated further than just identifying that the function did not work as intended.

Localization

Localization on the football field was performed sparsely by the robot. Important concepts were distances and angles to objects in the environment. The ball and goals were used to navigate and position the robot on the soccer field. Angles where the ball and goals were last seen gave the robot sufficient information about the environment to perform its tasks. Pose tracking through the V-Omega Driving Interface then enabled the

robot to compare angles and decide which direction to turn when searching for a lost ball. Distances were measured according to a function deciding on the cameras pixel positions. Higher up in the image naturally meant that the object was further away, with error in estimations increasing with the y-value of the pixel (because each pixel then represents a larger distance). The function measuring the distance was created with a least squares function approximator using polynomial base functions. Coefficients were determined with data gathered from images containing marks at certain known distances from the robot on the field. A lookup table was initiated on startup and used for the distance function to ensure that it executed quickly.

Distances to objects were used for example when deciding the speed of the robot; the robot drove at a lower speed when approaching the ball (to ensure it doesn't bounce off) compared to when the robot was further away from the ball. Since the project group had some problems with the roller of the robot it was important to treat the ball as something fleeting.



Marks on the field for distance approximations to the left.
Trashing another group's grid world to the right (tactical move)

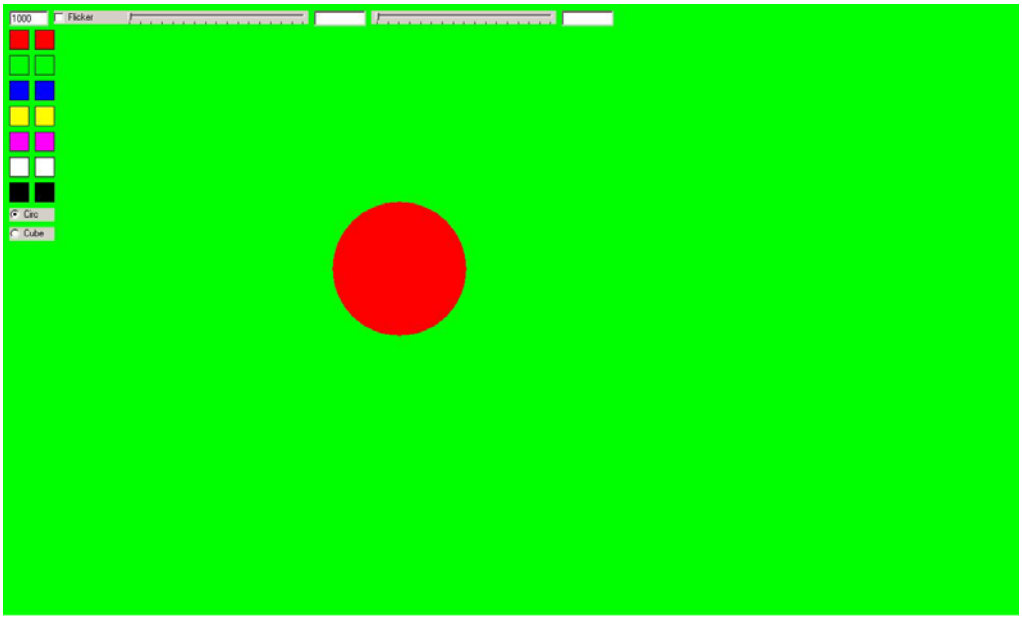
Additional Software Developed

To effectively design the robot software, the team created some tools that could help the process.

External Software

Ball simulator

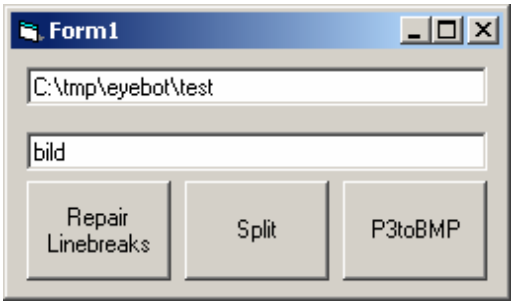
When the robots object detection routines was tested, the team wanted to make sure the pictures taken were correct and not occluded by errors or odd effects. To be sure to have perfect pictures, a small application was created in visual basic. It could simulate any background color, specified by its RGB or HUE value. It could also simulate a square or a circle with any color and size. When the robot was placed in front of the computer screen, it was easy to see if the robot were tracking the ball correctly when the ball was moved or the colors were varied.



Simulating a red ball against a green background

P3toBMP converter

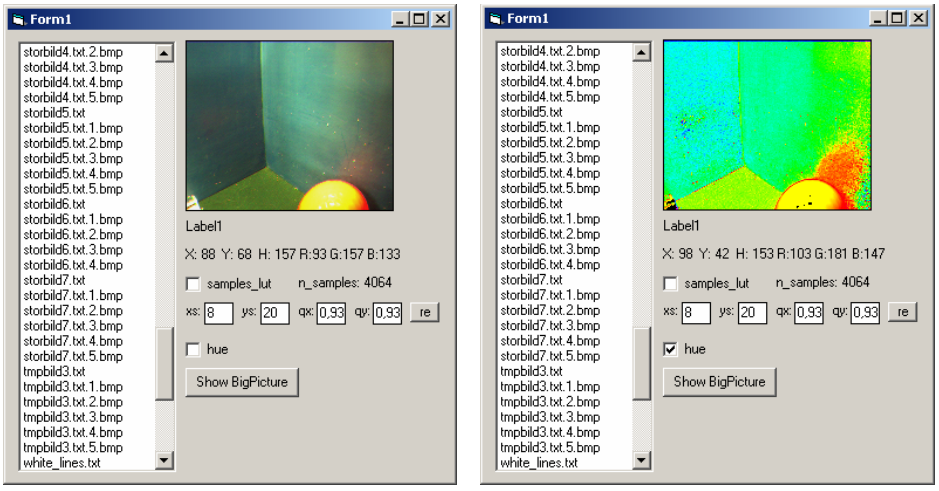
When pictures were taken with the robot, they could be transferred to the computer by a communications cable. The transferred picture format was a plain text format called P3. This format could only be opened by a limited variety of image processing software. When several pictures were taken between transfers, they were all stacked together in the same file. To view the separate images, they needed to be separated. Also, since the project group utilized several different computers, installed with UNIX, Linux and Windows, the line breaks were needed to be repaired in some situations. To deal with these problems, this P3toBMP application was created. It splits stacked files, repairs the line breaks if they are not correct and converts all of the separated images to correctly encoded 24-bit Bitmap Images.



Application to repair, split and convert images

Picture Analyzer

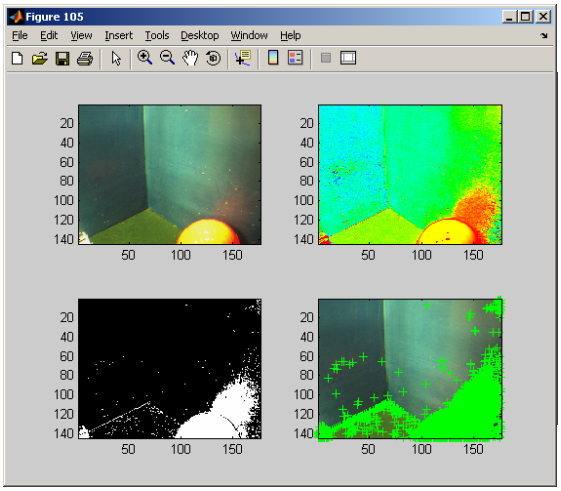
When pictures were taken, it was often interesting to see why the robot acted in a certain way. Therefore a Picture analyzer was created to easily inspect the pictures. It is capable of image zooming, HUE spectrum conversion and sampling. Also the application had a feature that plotted the samples_lut (See image processing) given different parameters.



Analyzes pictures, sample colors, inspect HUE spectrum, and more

Matlab – Image classifier

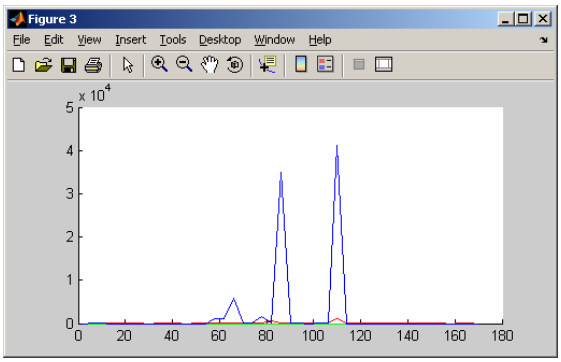
When writing the object detection algorithms for the robot, it was clear that the colors were of big importance. To effectively tweak the color intervals this Matlab code was written. It could load a picture, P3 or BMP, and given some parameters display where certain pixels were classified as an object. These parameters could then be transferred directly to the code for the robot.



Ball classification

Matlab – Algorithm analyzer

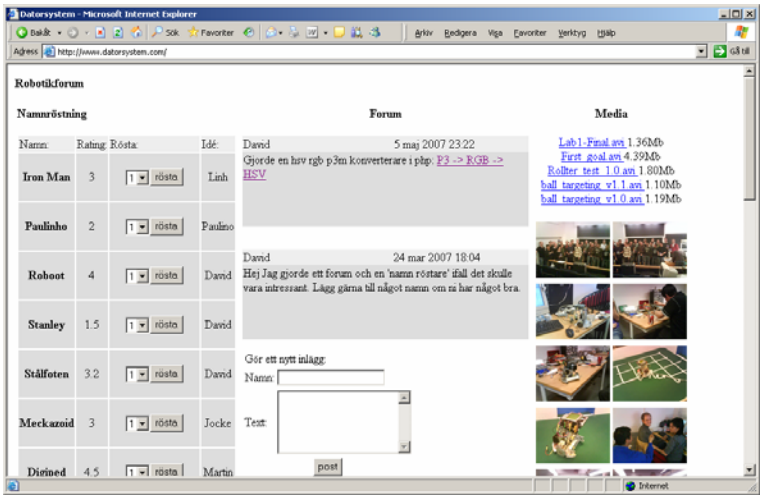
The algorithms in the robot contain lots of tweakable variables. There are parameters for ball detection, goal detection, LUT tables and much more. To see how all these parameters came together a Matlab script was written. It simulated the robots algorithms precisely, and given the same parameters it behaved exactly in the same way as the robot. With Matlab it was very easy to visualize what the robots decision would be.



Edge detection graph

Web synchronizer

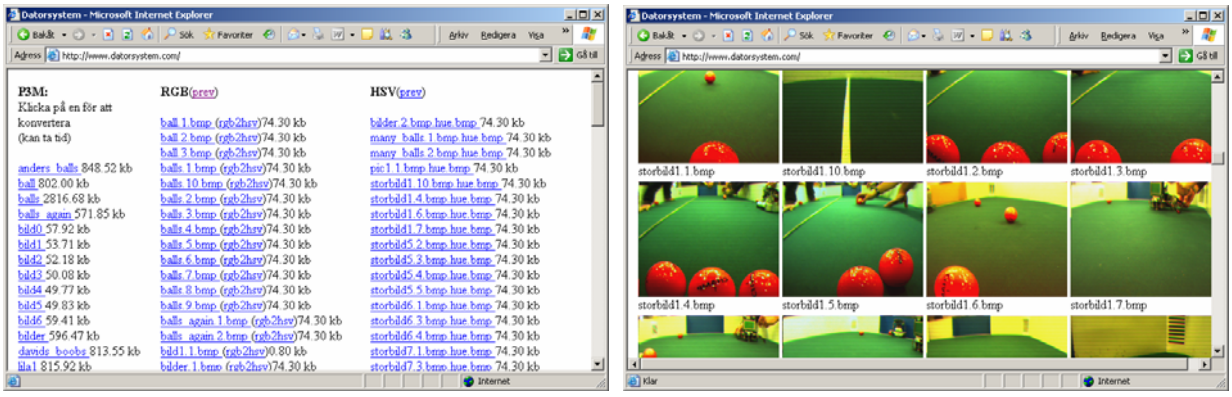
The team created a webpage allowing for easier communication between project members and sharing of information. The page contained a forum, a voting feature and media collecting abilities. It also had the ability of making sub-pages if any additional information was needed to be shared. Everything was written in PHP with a SQL database.



Project homepage

Web P3toBMP

To allow all the members to access all pictures taken, and share them without worrying about the different platforms, this page was written. When a picture was taken with the robot, it could be uploaded to this page, converted and stored online.



P3 to RGB bmp to HUE (HSV) bmp. Online visualization of pictures

Internal Software

Internal software refers to additional software that was written to be executed on the robot. These files had mostly diagnostic or helping purposes.

Speed test (speed_test.c)

Speed test is program for testing the robot's ability to keep the ball with rollers in various speeds when turning and driving straight. In usage the speed can be incremented and decremented with 0.05m/s while driving straight and PI/8 rad/s while turning. The LCD displays the current speed. In this manner it was easy to find the velocities that was optimal for moving the robot with the ball still in control

Take pictures (takepic.c)

This file was copied from the EyeBot site and modified to fit our purposes. It takes pictures with the camera and sends them to the computer. During the nocturnal hue tweaking period several pictures were taken and then sent to the computer in batches.

Extras (extras.c)

Contains helpful little functions such as
rand_float(), set_lights() and hue2rgb().

Music (robomusic.c)

This file contains all the musical compositions that were made for the robot’s different events. Such as the music played when a goal was made.

Discussion and results

"Claes `The LUT' Ohlsson" classified very comfortably for the group game. In this qualifying stage the robot demonstrated to have great skills of good scorer. It did one of the fastest goals of the championship. In the group play it took the first place and met the second of the other group. In the semi finals football proved again to be a game with strange mechanics; Claes Ohlsson lost a close game which could just as well have gone the other way. After the loss in the semi finals Claes Ohlsson played the next game in a convincing manner, resulting in a bronze medal.

The project group have overall been very satisfied with the project and course in general. Few courses or projects have engaged us as much during our studies at KTH. Many hours have been spent on this 5-point course which actually made it feel more like a 10-point course.

Claes “The LUT” Ohlsson went from some pieces of aluminum and components into a relatively functional robot with robust features and control. Some more features would have felt good and fun to implement but for various reasons they have been removed or not even started.

The EyeBot platform, together with the supplied materials, provided many useful functions and peripherals that made construction and coding of the robot much easier than what it had been otherwise. But during the project part of the course it became evident that the robot was constructed and coded quite far away from the complexities and methods discussed during the lectures. The reason for this was mainly due to the computational power of the EyeBot controller. Even if an advanced robot would have been optimized and coded in a time and memory efficient manner it probably would not be able to compete well with faster reacting robots. For this discussion it is also relevant as to how well performing the robot actually have to be. The tournament, which feels like a fun and appealing goal, could perhaps take more into account the techniques that are used, to promote usage of higher level reasoning and robotics methods more. Since the robots would then most likely be playing at lower speeds and update frequencies some other judgment would most likely have to be used.

Prior the tournament all teams should have gotten more information about the rules and how they could be applied. E.g the rule for repair was used for tactical reasons by some teams during the preliminary round and at the actual tournament. A robot could be taken off the field for 30 seconds placed in the defensive corner that is farthest away from the ball. If the robot was slow (and stupid) this was a preferable choice after each scoring, since the robot could be put back facing the ball (instead of getting stuck in the goal). Therefore this rule should preferably be renamed to indicate its actual usage.

For training, it would be a good idea to have a “course robot” programmed by the course leader or to have the best robot from previous year, to compare against. This might improve the quality of the robots for the tournament. Many ideas came up during the tournament when it was too late to make improvements.

The course should consist more of sensors and electronics. Consider for instance a team consisting of only software engineers and no electric engineer. Some further guidance would have been appreciated by the course assistant or during some extra seminars or lectures since not all are good at electronics.

Attachments

The Software

robot.c

```
#include "eyebot.h"
#include "robomusic.h"
#include "takepic.h"
#include "extras.h"
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>

#define min(a,b) (a<b?a:b)
#define max(a,b) (a>b?a:b)
#define abs(a)   (a<0?-a:a)

#define PI 3.14159
#define FALSE 0
#define TRUE 1
#define MAX_SPEED 0.2
#define TURN_SPEED PI/4
#define MAX_CHAOS_DIST 0.3
#define NO_HUE 255

#define MIN_TURN_ANGLE 2*PI/60
#define MIN_TURN_SPEED 2*PI/60

/* Global boolean */
int HAS_BALL, IS_GOAL, SEE_OUR_GOAL, SEE_TARGET_GOAL, SEE_YELLOW_GOAL;
int SEE_BLUE_GOAL, SEE_OPPONENT, SEE_BALL, GOAL;
int TARGET_IS_BLUE;

bigcolimage img;

/* Resolution of the camera */
int img_x = 176;
int img_y = 144;

/* Only process a subsample of number of pixels in image */
int n_samples;

/* Set when ball and/or goal are found */
int ball_x;
int ball_y;
int ball_h;
int blue_x;
int blue_y;

/* Look-up tables */
BYTE X[6500];
BYTE Y[6500];
BYTE hue_lut[4096]; // for 16 color values of r, g and b
float speed_lut[144];
float dist_lut[144];
int tolerance_y_lut[144];
int tolerance_x_lut[144];
int ball_width_min_lut[176];
int ball_width_max_lut[176];

/* Function pointers needed for dynamic choice of goal */
int (*is_target_goal_col)(BYTE, int, int, int) = NULL;
```

```
int (*is_target_goal_hue)(BYTE) = NULL;
int (*col_func)(BYTE,int,int,int) = NULL;

/* Take picture, store it in global variable img */
void take_pic()
{
  int ret_value = CAMGetFrameRGB((BYTE*)img);

  if (ret_value != 0) {
    printf("camera not initialised\n");
    return;
  }
}

/* Calculates hue according to the RGB-values
 * from imageproc.c written by Birgit Graf, Thomas Braunl
 * modified by Mattias Bratt
 */
BYTE calc_hue(int r, int g, int b)
{
  BYTE hue /*,sat, val*/, delta, vmax, vmin;

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

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

  /* val   = max;
     if (max != 0) sat = delta / max; else sat = 0;
     if (sat == 0) hue = NO_HUE;
  */


  // hmm:  if (2 * delta <= vmax) hue = NO_HUE;

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

}


/* Some pixels in right and left upper corner are removed
 * since we don't want to process pixels showing anyting else
 * but the football field. The camera is placed in such a way that in
 * some angle the corner will cover the surrounding of the football
 * field and its wall.
 */
inline int ignore_pixel(int x, int y)
{
  if((x > 155 || x < 20) && y < 10)
    return TRUE;

  return FALSE;
}

/*
 * Setup all the LUTs
*/

/* Create LUT's containing sample pixels
 * X, Y starts at lower left corner
```

```
 */
void setup_samples_lut() {

  BYTE y,x;

  float ys = 8;    // start interval, y
  float xs = 20;   // start interval, x
  float qy = 0.93; // how much we remove of the prev interval between pixels
  float qx = 0.93; // same but for x

  int y_offset = 1;
  int max_y = img_y-1;
  int max_x = img_x-1;

  int i = 0;
  for (y=1; y <= max_y-1-y_offset; y = y + (BYTE)ys+1) {
    ys = (ys*qy);
    xs = (xs*qx);

    for (x=1; x <= max_x-1; x = x + (BYTE)xs+1) {
      if(!ignore_pixel(x,max_y-y))
        {
          X[i] = x;
          Y[i] = max_y-y;
          i++;
        }
    }
  }

  n_samples = i;
}

/* Due to light issues the images can sometime be quite red.
   These LUTs contains the max and min width for the ball for
   different y-coordinates in the image, thus minimizing the risk of
   missclassifying a set of red pixels as ball.
 */
void setup_ball_width_lut()
{
  int y;
  for (y=0; y < img_y+1; y++)
    {
      ball_width_min_lut[y] = ((y*1024)/4)/1024 - 2;   // (int)((float)(y)/4.0-2.0);
      ball_width_max_lut[y] = ((y*1024)/2)/1024 + 8;   // (int)((float)(y)/2.0+8.0);
    }
}

/* A LUT not used.
   Is intented to be a distance look-up table. Was created by placing markings
   on the field in the middle of the robot's view and captured in image.
   Matlab made the estimated functions below.
*/
void setup_dist_lut()
{
  int y;

  /*0-20 cm */
  for (y = 144; y > 60; y--)
    dist_lut[y] = (49.3017 - 0.6275*y + 0.0022*y*y)*0.01;

  /*20-60 cm */
  for (y = 60; y > 18; y--)
    dist_lut[y] = (160.2243 - 5.3469*y + 0.0503*y*y)*0.01;

  /* allt som är "för långt bort", sätt till 0.5 m */
  for (y = 18; y > 0; y--)
    dist_lut[y] = 0.5 ;

}

/* LUT for converting RGB to HUE
```

```
 It is discretisized with 16 step. (16^3 instead of 256^3)
*/
void setup_hue_lut()
{
  int num_col_vals = 16;
  int col_incr = 256/num_col_vals;

  int r,g,b;
  BYTE hue;

  for(r = 0; r < 256; r += col_incr)
    for(g = 0; g < 256; g += col_incr)
      for(b = 0; b < 256; b += col_incr)
      {
        hue = calc_hue((BYTE)r,(BYTE)g,(BYTE)b);

        hue_lut[(r/col_incr) + ((g/col_incr)*16) + ((b/col_incr)*256)] = hue;
      }
}

/* Doesn't work as intended, should depend more on the distance to ball
 * Now - the farther away the higher speed
 */
void setup_speed_lut()
{
  int y;
  float k1 = 0.05;
  float k2 = 0.10;
  float k3 = 0.15;

  //0-20 cm
  for (y = 144; y > 60; y--)
    //speed_lut[y] = (49.3017 - 0.6275*y + 0.0022*y*y)*0.01*k1;
    speed_lut[y] = k1;

  //20-60 cm
  for (y = 60; y > 18; y--)
    //speed_lut[y] = (160.2243 - 5.3469*y + 0.0503*y*y)*0.01*k2;
    speed_lut[y] = k2;

  for (y = 18; y > 0; y--)
    speed_lut[y] = k3;

}

/*
 * Used when calculating average hue of an area when an interesting
 * pixel is found. Different width and height depending on the
 * location in the image. The farther up in the picture, the lower
 * tolerance.
 */
void setup_tolerance_lut()
{
  int ys;

  for (ys = 0; ys < img_y+1 ; ys++)
    {
      tolerance_x_lut[ys] = (int)(6.0*(float)(ys)/144.0);
      tolerance_y_lut[ys] = (int)(4.0*(float)(ys)/144.0);
    }
}


/*************end of setting up LUTs*********************************/

/*
 * Calculate hue value for specified pixel from img
 * Uses hue_lut.
*/
int img_hue(bigcolimage* img, BYTE x, BYTE y)
{
```

```
  int r = (*img)[y][x][0];
  int g = (*img)[y][x][1];
  int b = (*img)[y][x][2];

  int num_col_vals = 16;
  int col_incr = 256/num_col_vals;

  return hue_lut[(r/col_incr) + ((g/col_incr)*16) + ((b/col_incr)*256)];
}

/*
 * Calculate average hue value for specified pixel from img
*/
int avg_hue(bigcolimage* img, int xs, int ys)
{
  int min_x = 1;
  int max_x = img_x-2;
  int min_y = 1;
  int max_y = img_y-2;

  int tolerance_x = (int)(6.0*(float)(ys)/144.0);
  int tolerance_y = (int)(4.0*(float)(ys)/144.0);

  int x;
  int y;
  int num = 0;
  int sum = 0;
  for(x = max(xs-tolerance_x,min_x); x <= min(xs+tolerance_x,max_x); x++)
    for(y = max(ys-tolerance_y,min_y); y <= min(ys+tolerance_y,max_y); y++)
    {
      sum += img_hue(img,x,y);
      num++;
    }

  return (int)(sum/num);
}

/*
 * Calculate average hue value for a line surrounding
 * the specified pixel for finding goal
*/
int avg_goal_hue(bigcolimage* img, int x, int y)
{
  int min_x = 1;
  int max_x = img_x-2;

  int j;
  int num = 0;
  int sum = 0;
  for(j = max(x-8,min_x); j < min(x+8,max_x) ; j++)
    {
      sum += img_hue(img,j,y);
      num++;
    }
  return sum/num;
}

/*
 * Functions for testing hue values
 */
int is_ball_hue(BYTE hue)
{
  return (hue < 46 && hue > 39);
}

int is_has_ball_col(BYTE hue, int r, int g, int b)
{
  int a = (hue < 68 && hue > 39) && ((b+g)*50) < r*85;
  int b2 = (b < 60) && (r > 120);
  int c = (hue < 95 && hue > 62) && (r >= g) && (r > 100) &&  (g > 100) && (g >= b) ;
```

```
  return (a && b2) || c;
}

int is_ball_col(BYTE hue, int r, int g, int b)
{
  int a = (hue < 68 && hue > 39) && ((b+g)*50) < r*85 && (g < b+60);
  int b2 = (b < 60) && (r > 120);

  return (a && b2);
}

int is_blue_goal_hue(BYTE hue)
{
  return hue > 155 && hue < 210;
}

int is_blue_goal_col(BYTE hue, int r,int g,int b)
{
  return is_blue_goal_hue(hue) && (r*100 < b*85) && (g*100 < b*85) && (r < 70);
}

int is_yellow_goal_hue(BYTE hue)
{
  return hue > 65 && hue < 85;
}

int is_yellow_goal_col(BYTE hue,int r,int g,int b)
{
  return is_yellow_goal_hue(hue) && (b < 30) && (g < 200);
}

/*
 * Calculate mean pixel for an object searched for.
 * If non existed FALSE is return.
 */
int is_obj_within(int (*col_func)(BYTE,int,int,int), int i, int min_width, int max_width)
{
  int x = X[i];
  int y = Y[i];

  int width = 0;
  int xx;

  int plus_count = 0;
  int minus_count = 0;

  if (min_width < 0)
    min_width = 0;

  if (x - min_width < 0)
    min_width = min_width - x;
  if (x + min_width > img_x)
    min_width = img_x - x;

  for(xx = x; xx <= img_x-2; xx++)
    {
      BYTE r = img[y][xx][0];
      BYTE g = img[y][xx][1];
      BYTE b = img[y][xx][2];

      if((*col_func)(avg_hue(&img,xx,y),r,g,b) == FALSE)
        {
          break;
        }

      width++;
      plus_count++;
    }
```

```
  for(xx = x-1; xx >= 1; xx--)
    {
      BYTE r = img[y][xx][0];
      BYTE g = img[y][xx][1];
      BYTE b = img[y][xx][2];

      if((*col_func)(avg_hue(&img,xx,y),r,g,b) == FALSE)
        {
          break;
        }

      width++;
      minus_count++;
    }

  if(width < min_width || width > max_width)
    {
      return FALSE;
    }

  int mean = x + (int)((plus_count-minus_count)/2.0);
  return mean;
}

/* Checks for yellow edges in image
 * uses x, and x+1, so make sure it doesnt test wrong pixels
 * For more information, see project report
 */
int is_yellow_scarf_dx(int x, int y, int is_scarf_dist)
{
  int d = 2;
  BYTE r1 = img[y][x][0];
  BYTE g1 = img[y][x][1];
  BYTE b1 = img[y][x][2];

  BYTE h1 = calc_hue(r1,g1,b1);
  if(!is_yellow_goal_hue(h1) || is_blue_goal_hue(h1))
    return FALSE;

  BYTE r2 = img[y][x+d][0];
  BYTE g2 = img[y][x+d][1];
  BYTE b2 = img[y][x+d][2];

  BYTE h2 = calc_hue(r2,g2,b2);
  if(!is_yellow_goal_hue(h2) || is_blue_goal_hue(h2))
    return FALSE;

  int dist = 0;
  dist += (r1-r2)*(r1-r2);
  dist += (g1-g2)*(g1-g2);
  dist += (b1-b2)*(b1-b2);

  return dist > is_scarf_dist;
}

/* is_scarf_dist - what determines wether it is a scarf between two
 * pixels. Squared distance in RGB-space is used.
 * fault_piuxels_tol - how many pixels have to be scarf along the
 * y-axis.
 */
int find_scarf_over_y(int x, int y_min, int y_max, int is_scarf_dist)
{
  int y;
  int scarf_xs = 0;

  for(y = y_min; y <= y_max; y++)
    {
      if(is_yellow_scarf_dx(x,y,is_scarf_dist))
        scarf_xs++;
    }
```

```
  float fault_pixels_tol = 0.8;

  return scarf_xs >= fault_pixels_tol*(y_max-y_min);
}

/* Returns the mid point of the yellow goal
 * Uses scarfs
 */
int find_yellow_goal_mid()
{
  int y = 4;

  int vert_y_min = 2;
  int vert_y_max = 5;

  int scarf_dist = 5000;

  int scarf1 = -1;
  int scarf2 = -1;

  int hoppDist = 2;

  int x;
  for(x = 20; x <= 155 - hoppDist - 1; x=x+hoppDist)
    {
      // If the first pixel is scarf, then check the column
      if(is_yellow_scarf_dx(x,y,scarf_dist))
        {
          if(find_scarf_over_y(x,vert_y_min,vert_y_max,scarf_dist))
            {
              // have scarf
              if(scarf1 > 0)
                {
                  scarf2 = x;
                  break;
                }
              if(scarf1 < 0) scarf1 = x;
            }
        }
    }

  if(scarf1 >= 0 && scarf2 >= 0)
    {
      return (scarf1+scarf2)/2;
    }

  if(scarf1 >= 0)
    {

      BYTE r1 = img[y][scarf1 ][0];
      BYTE g1 = img[y][scarf1 ][1];
      BYTE b1 = img[y][scarf1 ][2];

      BYTE r2 = img[y][scarf1 + 2][0];
      BYTE g2 = img[y][scarf1 + 2][1];
      BYTE b2 = img[y][scarf1 + 2][2];


      int h1 = r1 + g1 + b1;
      int h2 = r2 + g2 + b2;

      if (h1 > h2)
        {
          scarf1 = min(scarf1 + 30,img_x-2);
        }
      else
        {
          scarf1 = max(scarf1-30,2);
        }

     return scarf1;
```

```
    }

  return FALSE;
}

/* Image processing
 */
void img_proc() {

  BYTE x,y, hue,r,g,b;
  HAS_BALL = FALSE;
  IS_GOAL = FALSE;
  SEE_TARGET_GOAL = FALSE;
  SEE_OUR_GOAL = FALSE;
  SEE_YELLOW_GOAL = FALSE;
  SEE_BLUE_GOAL = FALSE;
  SEE_OPPONENT = FALSE;
  SEE_BALL = FALSE;
  GOAL = FALSE;

  ball_x = -1;
  ball_y = -1;
  ball_h = -1;

  int i = 0;
  int tmpVar = 0;
  int xk = 0;
  int yk = 0;

  if(!TARGET_IS_BLUE)
    {
      blue_x = find_yellow_goal_mid();
      if(blue_x)
        SEE_TARGET_GOAL = TRUE;
    }

  int j;
  int see_balle = FALSE;
  for (j = 70; j < 84; j++)
    {
      x = X[j];
      y = Y[j];

      r = img[y][x][0];
      g = img[y][x][1];
      b = img[y][x][2];
      see_balle = is_has_ball_col(avg_hue(&img, x, y),r,g,b);
      if (see_balle)
        {
          break;
        }
    }

  int ball_width_min = 0;
  int ball_width_max = 0;

  for(i = 0; i < n_samples; i++)
    {
      x = X[i];
      y = Y[i];
      r = img[y][x][0];
      g = img[y][x][1];
      b = img[y][x][2];
      hue = img_hue(&img, x,y);
      tmpVar++;
      ball_width_min = ball_width_min_lut[y];
      ball_width_max = ball_width_max_lut[y];

      // determine has_ball (and see_ball if so)
      if(!see_balle && !HAS_BALL && (i >= 0 && i <= 54))
        {
```

```
          int found_ball_x =
is_obj_within(&is_has_ball_col,i,ball_width_min,ball_width_max);

          if(found_ball_x >= 25 && found_ball_x <= 140)
            {
              HAS_BALL = TRUE;
              SEE_BALL = TRUE;
              ball_x = found_ball_x;
              ball_y = y;
              ball_h = hue;
            }
        }


      // determine see_target_goal (if the targetgoal is blue, yellow is handled
elsewhere)
      if(i > 4870 && !SEE_TARGET_GOAL)
        {
          if (TARGET_IS_BLUE)
            {
              if((*is_target_goal_hue)(hue) &&
(*is_target_goal_col)(avg_goal_hue(&img,x,y),r,g,b))
                {
                  SEE_TARGET_GOAL = TRUE;
                  int rq;
                  int gq;
                  int bq;

                  BYTE xq,h2;
                  for(xq = x; xq < img_x-2; xq++)
                    {
                      rq = img[y][xq][0];
                      gq = img[y][xq][1];
                      bq = img[y][xq][2];

                      h2 = avg_hue(&img,xq,y);
                      if(!(*is_target_goal_col)(h2,rq,gq,bq))
                        break;
                    }
                  blue_x=(BYTE)((x+xq)/2);
                  blue_y=y;
                }
            }

        }


      // determine see_ball
      if(i < 4600 && !SEE_BALL && is_ball_col(hue, r, g, b) )
        {
          int found_ball_x = is_obj_within(&is_ball_col,i,ball_width_min,ball_width_max);

          if(found_ball_x)
            {
              SEE_BALL = TRUE;
              ball_x = found_ball_x;
              ball_y = y;
              ball_h = hue;
            }
        }

      if (SEE_BALL && SEE_TARGET_GOAL)
        break;
  }


  /* check if a goal has been made and set GOAL */
  if(HAS_BALL && (*is_target_goal_hue)(avg_hue(&img,88,72)))
    {
      BYTE yq;
      for(yq = ball_y; yq >= 80; yq--)
```

```
        {
          if(!is_ball_hue(avg_hue(&img,ball_x,yq))
&& !(*is_target_goal_hue)(avg_hue(&img,ball_x,yq)))
            {
              break;
            }
        }
      if (yq >= 80)
        {
          GOAL = TRUE;
        }
    }
}

/* Calculate mean hue of the image for detection of erroneous colors
 * in images.
 * Not used.
 */
int img_mean() {
  double tr=0;
  double tg=0;
  double tb=0;
  int i;
  int j;

  for (i = 0; i<145; i+=4) {
    for (j = 0; j<177; j+=4) {
      tr += img[i][j][0];
      tg += img[i][j][1];
      tb += img[i][j][2];
    }
  }
  return calc_hue((int)(tr/(176*144)),(int)(tg/(176*144)),(int)(tb/(176*144)));
}

/* Get current angle of rotation for robot
 */
float get_cur_fi(VWHandle vw)
{
  PositionType p;
  VWGetPosition(vw, &p);
  return p.phi;
}

/* Correct angle for minimize movement.
 */
inline float cor_angle(float angle)
{
  if(angle < -PI) {angle += 2.0*PI;}
  if(angle > PI) {angle -= 2.0*PI;}

  return angle;
}

/* Calculate a corrected angle. Compensate for delay due to the image
 * processing
 */
float get_corrected_angle(VWHandle vw, int x, int y, PositionType* imgPos)
{
  PositionType curPos;
  VWGetPosition(vw, &curPos);
  float angle = ((1-((float)(y)/144.0))*15+45)*((float)(x-88)/88)*(PI/180);
  angle -= (curPos.phi-(*imgPos).phi);
  return cor_angle(angle);
}

/* Checks whether the robot has stalled
 */
int drive_wait(VWHandle* vw)
{
  float p1;
```

```
  float tolerance = 0.01;
  int stalled = FALSE;
  int t = OSGetCount();
  p1 = VWDriveRemain(*vw);
  int t2 = t;
  while (!VWDriveDone(*vw))
    {
      if (OSGetCount() > t + 200)
        {
          t = OSGetCount();
          if (abs(p1 - VWDriveRemain(*vw)) < tolerance )
            {
              stalled = TRUE;
              break;
            }
          p1 = VWDriveRemain(*vw);
        }
      if((t2+600) < OSGetCount())
        {
          stalled = TRUE;
          break;
        }
    }

  if (stalled)
    {
      VWSetSpeed((*vw), -MAX_SPEED, 0);
      OSWait(100);
      VWSetSpeed((*vw), 0, PI/2);
      OSWait(200);
      VWSetSpeed((*vw), MAX_SPEED, 0);
      OSWait(100);
      VWSetSpeed((*vw), 0, 0);
    }

  return !stalled;
}

/* Turn, drive straight and turn, with random speed, distance and angle
 */
void chaos_move(VWHandle* vw)
{
  float alpha = rand_float()*2*PI-PI;
  float beta  = rand_float()*2*PI-PI;
  float dist  = rand_float()*MAX_CHAOS_DIST;

  VWDriveTurn((*vw), alpha, TURN_SPEED);
  if(!drive_wait(vw)) return;
  VWDriveStraight((*vw), dist, MAX_SPEED);
  if(!drive_wait(vw)) return;
  VWDriveTurn((*vw), beta, TURN_SPEED);
  drive_wait(vw);
}

/* Setup VW-drive
 * Input for VWSstartControl given in lecture
 */
VWHandle setup_vwdrive()
{
  VWHandle vw;
  vw = VWInit(VW_DRIVE,1);
  if(vw == 0) { LCDPutString("VWInit Error!\n"); return 0; }
  VWStartControl(vw,7,0.3,7,0.1);
  return vw;
}

/* Setup servo
 * For DeviceSemantic see hdt.c
 */
ServoHandle setup_servo()
{
```

```
  ServoHandle servo;
  servo  = SERVOInit(SERVO9);
  if(servo == 0) { LCDPutString("servo_init Error!\n"); return 0; }
  return servo;
}

/* This is where it all happens.
 * See flow chart in project report.
 */
int run() {
  CAMInit(NORMAL);
  CAMSet(FPS1_875, 0, 0);

  VWHandle vw;
  ServoHandle servo;

  servo = setup_servo();
  SERVOSet(servo, 152);

  vw = setup_vwdrive();

  VWSetSpeed(vw, 0, 0);

  int k = 0;

  PositionType imgPos;

  float old_goal_fi = 0;
  float old_ball_fi = 0;
  float goal_rot_dir = 0;
  float ball_rot_dir = 0;
  int confidence = 2;
  int see_no_ball_counter = 0;
  int see_no_goal_counter = 0;
  int servo_count = 0;

  // Calibrate the camera by taking 20 pictures while rotating
  int cc;
  for (cc=0; cc < 4; cc++)
    {
      take_pic();
      OSWait(20);
      take_pic();
      OSWait(20);
      take_pic();
      OSWait(20);
      take_pic();
      VWDriveTurn(vw, PI/2, 3*PI);
      drive_wait(&vw);
      LCDPrintf("%d..\n", 4-cc);
    }

  LCDClear();
  LCDPrintf("To start press S");
  LCDPrintf("n_samples: %d", n_samples);
  LCDMenu("S","","","");

  while(KEYRead() != KEY1)
    {
      //Press button to start
    }

  LCDClear();
  LCDPrintf("To quit press Q");
  LCDMenu("","","","Q");

  while(k != KEY4) {

    AUTone(1000,50);
    servo_count ++;
    if (servo_count >= 10)
```

```
  {
    SERVOSet(servo, 152);
   servo_count = 0;
  }

take_pic();
VWGetPosition(vw, &imgPos);

AUTone(4000,50);
img_proc();
AUTone(8000,50);

set_lights(HAS_BALL,SEE_TARGET_GOAL,SEE_BALL);

if(SEE_TARGET_GOAL)
  {
   old_goal_fi = get_cur_fi(vw);
   see_no_goal_counter = 0;
  }

if(SEE_BALL)
  {
   old_ball_fi = get_cur_fi(vw);
   see_no_ball_counter = 0;
  }

if (GOAL)
  {
   confidence = 2;
   ball_rot_dir = 0;
   goal_rot_dir = 0;
   see_no_goal_counter = 0;

   play_music(4);

   VWSetSpeed(vw, -MAX_SPEED, 0);
   OSWait(400);
   VWSetSpeed(vw, 0, 0);
  }
else if (SEE_BALL && !HAS_BALL)
  {
   confidence = 2;
   ball_rot_dir = 0;
   goal_rot_dir = 0;
   see_no_goal_counter = 0;

   float k = 1;
   float angle = get_corrected_angle(vw, ball_x,ball_y, &imgPos);

   LCDPrintf("See ball\n");

   VWDriveTurn(vw, cor_angle(max2(MIN_TURN_ANGLE, k*angle)), TURN_SPEED);

   if(drive_wait(&vw))
     {
       VWSetSpeed(vw,speed_lut[ball_y],0) ;
     }
  }
else if(HAS_BALL && !SEE_TARGET_GOAL)
  {
   confidence = 2;
   ball_rot_dir = 0;
   see_no_goal_counter++;

   if (see_no_goal_counter > 20)
     {
       see_no_goal_counter = 0;
       chaos_move(&vw);
       continue;
     }
```

```
        LCDPrintf("Have ball\n");

        if(goal_rot_dir == 0)
          goal_rot_dir = (old_goal_fi-get_cur_fi(vw) < 0) ? -1 : 1;

        VWDriveTurn(vw, goal_rot_dir*PI/8, TURN_SPEED*0.5);
        drive_wait(&vw);
      }
    else if(HAS_BALL && SEE_TARGET_GOAL)
      {
        confidence = 2;
        ball_rot_dir = 0;
        goal_rot_dir = 0;
        see_no_goal_counter = 0;

        float k = 0.8;
        LCDPrintf("Have ball&goal!\n");
        float angle = get_corrected_angle(vw, blue_x, blue_y, &imgPos);

        VWDriveTurn(vw, cor_angle(max2(MIN_TURN_ANGLE, k*angle)), TURN_SPEED*0.5);
        if(drive_wait(&vw))
          VWSetSpeed(vw, MAX_SPEED*0.5, 0);
      }
    else
      {
        goal_rot_dir = 0;
        see_no_goal_counter = 0;
        confidence = confidence - 1;
        if (confidence <= 0)
          {
            float rot_dist = PI/4;
            if(see_no_ball_counter*rot_dist >= (2*PI)) // turned whole varv without see
ball
              {
                LCDPrintf("no ball 2PI-chaos\n");
                chaos_move(&vw);
                continue;
              }

            see_no_ball_counter++;

            if(ball_rot_dir == 0)
              ball_rot_dir = (old_ball_fi-get_cur_fi(vw) < 0) ? -1 : 1;

            LCDPrintf("can't see ball\n");

            if(old_ball_fi == 0)
              {
                VWDriveTurn(vw, max2(MIN_TURN_ANGLE, cor_angle(ball_rot_dir*rot_dist)),
TURN_SPEED);
              }
            else
              {
                VWDriveTurn(vw, max2(MIN_TURN_ANGLE, cor_angle(old_ball_fi-
get_cur_fi(vw))), TURN_SPEED + TURN_SPEED*(see_no_ball_counter>3));
              }

            drive_wait(&vw);
            old_ball_fi = 0;

          } else {
            VWDriveStraight(vw, 0.05, 0.1);
            drive_wait(&vw);
          }
      }

    k = KEYRead();
  }

  /* exit driver, servo and turn off the lights */
  VWRelease(vw);
```

```
  SERVORelease(servo);
  set_lights(0,0,0);
  return 1;
}

/* Setup all the LUTs
 * Choose color of target goal by pressing key
 * and run
 */
int main(void)
{
  int k=0;

  setup_samples_lut();
  setup_hue_lut();
  setup_speed_lut();
  setup_dist_lut();
  setup_tolerance_lut();
  setup_ball_width_lut();

  LCDMenu("B","Y","","S");

  while (k != KEY4)
    {
      k = KEYRead();

      if (k == KEY1)
        {
          TARGET_IS_BLUE = TRUE;
          is_target_goal_col = &is_blue_goal_col;
          is_target_goal_hue = &is_blue_goal_hue;
          LCDPrintf("BLUE!\n");
        }
      else if (k == KEY2)
        {
          TARGET_IS_BLUE = FALSE;
          is_target_goal_col = &is_yellow_goal_col;
          is_target_goal_hue = &is_yellow_goal_hue;
          LCDPrintf("YELLOW!\n");
        }
    }
  srand(OSGetCount());
  run();
  return 0;
}
```

extras.c

```
#include "eyebot.h"
#include "extras.h"
#include <stdio.h>
#include <stdlib.h>
#define abs(a)   (a<0?-a:a)

void error(char *str)
{
  LCDPrintf("ERROR: %s\n", str);
  OSWait(500);
}

void println(char *str)
{
  LCDPrintf("%s\n", str);
}

void println_int(char *str, int val)
{
  LCDPrintf("%s:%d\n", str,val);
}

float rand_float()
{
  int r = rand();

  return ((float)r) / ((float)(RAND_MAX)); // [0 1]
}

/*
 * returns b if b is larger in distance, otherwise a (with the same sign as b)
 *   a must be positive
 */
float max2(float a, float b)
{
  if(abs(b) > a)
    return b;
  else
    return b < 0 ? a*-1 : a;
}


void set_lights(int l1, int l2, int l3)
{

  BYTE L = 0;
  BYTE M = 0xFF;

  if (l1)
    {
      L += 0x02;
      M = M & 0xFD;
    }
  if (l2)
    {
      L += 0x08;
      M = M & 0xF7;
    }
  if (l3)
    {
      L += 0x20;
      M = M & 0xDF;
    }
  OSWriteOutLatch(0,0x00,0x00);
  OSWriteOutLatch(0,M,L);
}

/* Only for testing, not used in final version
```

```
 *
 */
void hue2rgb(int h, int* r, int* g, int* b, int* Hi)
{
  *Hi = (int)((int)(360.0*(float)(h)/(255.0*60.0)) % 6);
  float f = (float)(h)*360.0/255.0 - (*Hi) * 60.0;
  f = (int)((f/60.0)*255.0);

  if (*Hi == 0)
    {
      *r = 255;
      *g = f;
      *b = 0;
    }
  else if (*Hi == 1)
    {
      *r = 255-f;
      *g = 255;
      *b = 0;
    } else if (*Hi == 2) {
       *r = 0;
      *g = 255;
      *b = f;
    } else if (*Hi == 3) {
      *r = 0;
      *g = 255-f;
      *b = 255;
    } else if (*Hi == 4) {
      *r = f;
      *g = 0;
      *b = 255;
    } else if (*Hi == 5)
      {
       *r = 255;
       *g = 0;
       *b = 255-f;
      }
  else
    {
       *r = 0;
       *g = 0;
       *b = 0;
    }
      if (h<0 || h>255) {
       *r = 0;
       *g = 0;
       *b = 0;
      }

      return;

}
```

takepic.c

```
/*************************************************************/
/* takepic.c                                                 */
/*                                                           */
/* This program is for taking pictures with the camera on    */
/* the Eyebot, and then uploading them to the PC             */
/*                                                           */
/* Andrew Berry                                              */
/* 18/10/99                                                  */
/*                                                           */
/* modifed by ras0, 2007                                     */
/*                                                        */
/*************************************************************/

#include "eyebot.h"
#include "robomusic.h"
#include <stdlib.h>
#include <stdio.h>
#define MAXPICS 15

typedef BYTE bigcolimage[144][176][3];
bigcolimage bilden;
bigcolimage img;

/*************************************************************/
/* TakePic                                                   */
/*                                                           */
/* Takes a picture                                           */
/*************************************************************/

bigcolimage* TakePic(bigcolimage* pic)
{
  //  AUBeep();
  CAMGetFrameRGB((BYTE *)pic);
  return pic;
}

/*************************************************************/
/* sendData                                                  */
/*                                                           */
/* Program to send the data to the PC                        */
/* Jesse Pepper 1999                                         */
/*************************************************************/

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

void sendRGBData(int port, bigcolimage* img )
{
  int i,j;
  int k;
  char temp[100];

  sprintf(temp,
          "P3\n"
          "176 144\n"
          "255\n");
  sendString(port, temp);


  for (j=0; j < 144; j++)
    {
```

```
      for (i=0; i < 176; i++)
        {
          sprintf(temp,"%d %d %d \n",(*img)[j][i][0],(*img)[j][i][1],(*img)[j][i][2]);
          sendString(port, temp);
        }
      k = KEYRead();
      if (k == KEY3)
       return;
    }
}
/*************************************************************/
/* SendPic                                                   */
/*                                                           */
/* Takes a picture                                           */
/*************************************************************/

int SendPic(bigcolimage* colimg)
{
  int port = SERIAL1;

  AUBeep();
  AUBeep();

  OSInitRS232( SER115200, NONE, port );

  sendRGBData(port, colimg);

  /* send RS232 termination character */
  sendString(port, "\x04\n");

  return TRUE;
}

//int main()
//{
//  CAMInit(NORMAL);
//  OSWait(10);
//  CAMInit(NORMAL);
//  CAMSet(FPS1_875, 0, 0);
//
//  bigcolimage pic[10];
//
//  LCDMenu("GO", " ", " ", " ");
//  LCDPrintf("Init...");
//  while(KEYRead() != KEY1)
//    {
//      TakePic(&pic[0]);
//    }
//
//  // NUMBER OF PICTURES YOU WANT TO TAKE!
//  int n_pic = 3;
//  int iu;
//
//  for (iu = 0; iu < n_pic ; iu++)
//    {
//      LCDMenu("TAKE", " ", " ", " ");
//      KEYWait(KEY1);
//      TakePic(&pic[iu]);
//      AUBeep();
//    }
//  LCDClear();
//  LCDMenu("", "SEND", " ", " ");
//  KEYWait(KEY2);
//  for (iu = 0; iu < n_pic; iu++)
//    {
//      SendPic(&pic[iu]);
//    }
//  play_music(4);
//  return 0;
//}
```

speed_test.c

```
#include "eyebot.h"
#include <stdio.h>
#include <math.h>
#define PI 3.14159
#define delta_turn PI/8
#define delta_straight 0.05

/*
   Program for testing the robot's ability to keep the
   ball (with rollers) in various speeds when turning and driving
   straight

   usage: gcc68 -o speed_test.hex speed_test.c
          dl speed_test.hex

   Press S for driving straight
   Press T for turning
   Increment and decrement speed with +/-.
   Change in speed when turning is PI/8
   Change in speed when driving straight is 0.05
   Stop with KEY4 anytime.
*/

typedef BYTE bigcolimage[144][176][3];
bigcolimage img;

void take_pic()
{
  int ret_value = CAMGetFrameRGB((BYTE*)img);

  if (ret_value != 0) {
    printf("camera not initialised\n");
    return;
  }
}

void turn_capture(vw)
{
  int cc;
  for (cc=0; cc < 5; cc++)
    {
      take_pic();
      // 2*PI/4 rad/s == PI/2 rad/s, så svängen ska väl
      // tamejtusan ta en halv sekund?! Yr som fan nu...
      VWDriveTurn(vw, PI/4, 2*PI/4);
      VWDriveWait(vw);
      LCDPrintf("%d..", 4-cc);
    }
}

void drive_forward(VWHandle vw)
{
  LCDClear();
  float speed = 0;
  int k;
  k = KEYRead();
  while (k != KEY4)
    {
      LCDMenu("+", "-", "", "Q");

      switch (k){
      case KEY1: speed = speed + delta_straight;
        LCDPrintf("speed: %2.2f\n", speed);
        break;
      case KEY2: speed = speed - delta_straight;
        LCDPrintf("speed: %2.2f\n", speed);
        break;
```

```
      case KEY3: break;
      default: break;
      }
      VWSetSpeed(vw, speed, 0);
      k = KEYRead();
    }
  VWSetSpeed(vw, 0, 0);
}

void turn(VWHandle vw)
{
  LCDClear();
  float speed = 0;
  int a = 0;
  int k;
  k = KEYRead();
  while (k != KEY4)
    {
      LCDMenu("+", "-", "", "Q");

      switch (k){
      case KEY1:
        speed = speed + delta_turn;
        a++;
        LCDPrintf("turn v:(PI/8)*%d\n", a);
        break;
      case KEY2:
        speed = speed - delta_turn;
        a--;
        LCDPrintf("turn v:(PI/8)*%d\n", a);
        break;
      case KEY3:
        break;
      default: break;
      }
        VWSetSpeed(vw, 0, speed);
      k = KEYRead();
    }
  VWSetSpeed(vw, 0, 0);
}

int main(void)
{
  //cam init
  CAMInit(NORMAL);
  CAMSet(FPS1_875, 0, 0);

  //VW init
  VWHandle vw;
  vw = VWInit(VW_DRIVE,1);
  if(vw == 0) { LCDPutString("VWInit Error!\n"); return 0; }
  VWStartControl(vw,7,0.3,7,0.1);
  int k;
  k = KEYRead();
  while (k != KEY4)
    {
      LCDMenu("D", "T", "", "Q");
      switch (k){
      case KEY1: drive_forward(vw);
        break;
      case KEY2: turn(vw);
        break;
      case KEY3: turn_capture(vw);
        break;
      default: break;
      }
      k = KEYRead();
    }
  //exit driver
  VWRelease(vw);
  return 1;
```

```
}
```

robomusic.c

```
#include "eyebot.h"

void play_music(int song)
{

  float multip = 10;
  float toneTime = 200;
  float waitTime = 30;

  if (song == 1)
    {
      AUTone(multip*16.35,toneTime);
      OSWait(waitTime);
      AUTone(multip*16.35,toneTime);
      OSWait(waitTime);

      AUTone(multip*24.50,toneTime);
      OSWait(waitTime);
      AUTone(multip*24.50,toneTime);
      OSWait(waitTime);

      AUTone(multip*27.50,toneTime);
      OSWait(waitTime);
      AUTone(multip*27.50,toneTime);
      OSWait(waitTime);

      AUTone(multip*24.50,toneTime);
      OSWait(waitTime*2.0);

      AUTone(multip*21.83,toneTime);
      OSWait(waitTime);
      AUTone(multip*21.83,toneTime);
      OSWait(waitTime);

      AUTone(multip*20.60,toneTime);
      OSWait(waitTime);
      AUTone(multip*20.60,toneTime);
      OSWait(waitTime);

      AUTone(multip*18.35,toneTime);
      OSWait(waitTime);
      AUTone(multip*18.35,toneTime);
      OSWait(waitTime);

      AUTone(multip*16.35,2*toneTime);
      OSWait(waitTime);

    }
  else if(song == 2)
    {
      AUTone(multip*24.50,2*toneTime);
      OSWait(2*waitTime);
      AUTone(multip*24.50,2*toneTime);
      OSWait(2*waitTime);
      AUTone(multip*24.50,2*toneTime);
      OSWait(2*waitTime);

      AUTone(multip*19.45,2*toneTime);
      OSWait(2*waitTime);
      AUTone(multip*29.14,0.7*toneTime);
      OSWait(0.7*waitTime);
      AUTone(multip*24.50,2*toneTime);
      OSWait(3*waitTime);

      AUTone(multip*19.45,2*toneTime);
      OSWait(2*waitTime);
      AUTone(multip*29.14,0.7*toneTime);
```

```
      OSWait(0.7*waitTime);
      AUTone(multip*24.50,2*toneTime);
      OSWait(3*waitTime);

    }
  else if(song==3)
    {
      AUTone(multip*36.71,2*toneTime);
      OSWait(2*waitTime);
      AUTone(multip*36.71,2*toneTime);
      OSWait(2*waitTime);
      AUTone(multip*36.71,2*toneTime);
      OSWait(2*waitTime);
      AUTone(multip*38.89,2*toneTime);
      OSWait(2*waitTime);
      AUTone(multip*29.14,toneTime);
      OSWait(waitTime);
      AUTone(multip*24.50,3*toneTime);
      OSWait(3*waitTime);
      AUTone(multip*19.45,2*toneTime);
      OSWait(2*waitTime);
      AUTone(multip*29.14,toneTime);
      OSWait(waitTime);
      AUTone(multip*24.50,4*toneTime);
      OSWait(4*waitTime);

    }
  else if(song == 4)
    {
      AUTone(3*multip*16.35,0.5*toneTime);
      OSWait(0.8*waitTime);
      AUTone(3*multip*16.35,0.3*toneTime);
      OSWait(0.3*waitTime);
      AUTone(3*multip*16.35,0.8*toneTime);
      OSWait(0.8*waitTime);
      AUTone(3*multip*24.50,3*toneTime);
      OSWait(3*waitTime);
    }
}
```