NTNU
Norwegian University of
Science and Technology

Tracking objects in 3D using Stereo Vision

Kai Hugo Hustoft Endresen

Master of Science in Computer Science
Submission date: June 2010
Supervisor: Ketil Bø, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

The project involves the design and implementation of a mobile system to track moving and stationary objects in 3D using stereo vision. The processed data should be able to give reasonably accurate coordinates and distances to objects for further processing by a planning system.

Assignment given: 25. January 2010
Supervisor: Ketil Bø, IDI

Abstract

This report describes a stereo vision system to be used on a mobile robot. The system is able to triangulate the positions of cylindrical and spherical objects in a 3D environment. Triangulation is done in real-time by matching regions in two images, and calculating the disparities between them.

Preface

This master thesis was during the spring semester of 2010 at the Department of Computer and Information Science, Norwegian University of Science and Technology, and it is a further development of work done during my pre-study project of the fall of 2009.

The purpose of this work was to gain an understanding of stereo vision, as well as create a basis for a real-time working system for tracking objects in 3D. This work was designed to be part of a mobile robot in collaboration with people of different disciplines for competing in Eurobot 2010.

The project work was supervised by associate professor Ketil Bø, and the assignment description was as follows:

> *The project involves the design and implementation of a mobile system to track moving and stationary objects in 3D using stereo vision. The processed data should be able to give reasonably accurate coordinates and distances to objects for further processing by a planning system.*

I would like to thank my supervisor, associate professor Ketil Bø, for allowing me to pursue this assignment and reminding me to focus on my writing.I would also like to thank Kongsberg Gruppen ASA for providing the funding for the hardware to make this assignment possible, as well as the Department of Cybernetics at NTNU for the use of their equiptment and staff for mechanical work.

Trondheim,
Kai Hugo Hustoft Endresen

Contents

Chapter 1

Introduction

In this project I hope to create a real-time stereo vision system to track various objects. The reason for using stereo vision is primarily because I believe that it would be the best way to pinpoint the location of an object in a 3D environment. Since the objects are known to me beforehand (shape, size and color), a single camera could have been used, but the accuracy for pinpointing location would be more accurate with stereo vision. The results should be accurate enough to be used for a planner system, the task of which is to decide the optimal way to pick up the maximum amount of balls and cylinders. The finished system is to be used as part of a robot that will compete in Eurobot 2010.

Eurobot 2010 is an annual international robotics competition, in which teams from many different countries compete. The general premise is that two robots should compete to get the most points within a 90 second interval. This competition is well suited for computer vision approaches, stereo vision in particular. The tasks vary quite a bit from year to year, and even though some reuse of earlier solutions is possible, a lot of equiptment and software has to be made anew. So even though some teams have been competing for a decade, their advantage over more recent teams isn't that pronounced.

The motivation for this project is mainly to create a working stereo vision system. Especially measuring distances to various objects, and also learning more about approaches to stereo vision, and depth imaging in general.

Previous work in the area of stereo vision has been mostly restricted to either stationary use, or very slow movement. With the introduction of increasingly faster computing devices, more and more of the things that required considerable processing time before, can now be done in real-time.

An example of a rather slow, but functional system for navigation with stereo vision, is the Stanford cart[21]. The Stanford cart came to be originally from the need of controlling a robotic unit on the moon, which is too far away for being remote controlled from earth without any degree of autonomous behavior, and was one of the first stereo vision systems. The stereo system was a camera that moved on a rail to get several pictures of the same scene and then calculate the distance to obstacles. It was very slow and needed 10 to 15 minutes of image processing time between each move.

Stereo Vision as a phenomenon was first described by Charles Wheatstone in 1838. He did various psychological and optical experiments to try to understand how humans percieve depth. He also mentioned that Leonardo da Vinci, several hundred years before, had noticed that it was not possible to draw something on a canvas and get the same realism as in the real world.

Stereo vision systems essentially try to emulate how the human vision system works, by modelling a scene using two cameras.

The most common approach is to have two cameras parallel to each other, with an horizontal offset. Other methods include using vertically aligned cameras, or cameras tilted towards one another. All stereo vision then comes down to is finding matching points in both views, and measuring the disparity between them. This can then be used for tracking objects[14], finding their position in 3D[19] as well as robot navigation[6]. It has even been used by NASA in their STEREO project for solar observations[22]. In Norway, FINN AS, which provides 3D maps of Trondheim, Oslo and other cities in Norway, uses stereo vision technology from C3 Technologies to do stereo reconstruction of landscape and houses from aerial pictures[29].

During the writing of this thesis, a surge in use of 3D imaging has been noticed for motion pictures, TVs and gaming devices such as the Nintendo 3DS. One movie of note is *Avatar*, in which the entire movie was shot by special 3D cameras. The same system is currently being designed in a more mobile format for the next Mars Rover. The system is delivered by Malin Space Science Systems, and optically it is very impressive. For instance it is possible to adjust the focal length while still maintaining stereo calibration. Thus, one is able to increase accuracy for objects far away, while still having the possibility of a large viewing area.[28]

The remainder of this report is divided into the following chapters:

(2) Theory describes most of the theoretical background used for the experiments and solution, and some theory concerning lenses and depth imaging.

(3) Environment presents the environment in which the system is meant to operate.

(4) System Design describes the system requirements, a brief overview of the steps involved in the programming of the system, as well as an overview of the software and hardware used.

(5) Experiments describes the results of testing various approaches to different parts of the stereo vision system.

(6) Solution shows how the final implementation works under various conditions.

(7) Discussion provides an evaluation of the working solution, as well as some notes about the performance of the system.

(8) Conclusion summarizes the report, and makes some statements with regards to future work.

Chapter 2

Theory

This chapter provides all the relevant background material to the stereo vision system presented later in this report. Subjects to be looked at include depth images, color conversion, edge detection, camera calibration, mathematical morphology, color segmentation, epipolar geometry and bayer filtering.

2.1 Stereo Vision

Stereo vision is denoted as *processes directed on understanding or analysing three-dimensional visible object surfaces based on image data*. The visual systems of humans and animals prove that stereo vision works in complex enviroments. Stereo vision is a very active field of research in computer vision.

An overview of steps can be seen to the right, in figure 2.1.

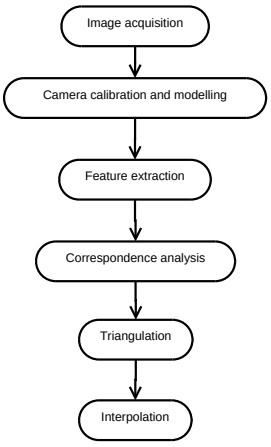


Figure 2.1: Pipeline

2.1.1 View area in stereo vision

The view area in a stereo vision system is essentially limited by the view angle of the individual cameras, and the relative orientation, as well as the distance between the cameras. See figure 2.2 for illustrations.

The further apart the cameras are, the further away the minimum distance for successful stereo vision becomes. The advantage of increasing the distance

between cameras is that the disparity increases linearly with the distance between cameras. This means that by doubling the distance between cameras, the depth resolution also doubles at a given distance. The disadvantage is that it is then impossible to measure the distance to objects very close, and the imaging system would need other methods to remedy this. This issue is also present in the human vision system, where we cannot distinguish how far an object very close to our eyes is without other visual cues.

As can be seen in figure 2.2, this issue can be reduced by mounting the cameras in such a way that the epipoles[1] are no longer parallel to each other. Instead of at infinity they should meet at a fixed point in front of the cameras, but then the geometric calculations and disparities would be much harder to calculate.

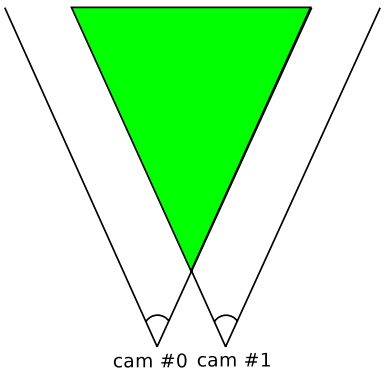


Figure 2.2: The green triangle shows the view area

2.2 Depth images

There are many ways to achieve depth images. The image formation process projects an image of the 3D world onto a 2D surface. Reversing the process is impossible as some of the information is invariably lost in the projection. Experiments with our own eyes show that the human visual system is capable of recovering a lot of this information from stereo vision; contours, texture, shadows and so on.

Using a single image, depth information can be extrapolated from shading, textures, contours, focus/defocus and various visual cues.

Shape from shading exploits the fact that if you are able to control the

[1]Epipoles are the center of the view area of the cameras.

lighting enviroment you are in, you can look at the reflections from surfaces, and therefore extrapolate the distance.

The way textures are transformed when they are projected from 3D to 2D can be used to infer depth. For instance, a wallpaper with horizontal stripes have an equal distance between each stripe when viewed at a perpendicular angle. When viewed from the side, the distance between stripes far away are greater than stripes up close.

Providing you know the physical size of an object, contour shapes can be used for calculating the distance to an object in an image. For example a perfectly spherical object can be extracted from the image by using the Hough transform. The radius of the matched circle can be measured, and you can then compare this to the physical radius of the object and find the distance.

Shape from focus/defocus exploits the fact that by adjusting the focus of a lens in consequtive steps, so that various depths of the image come into focus, and you can then calculate the distance. This might be particulary useful if the environment is static, and there are no time constraints. It is commonly used in cameras to give an approximation of the distance to an object in focus.

Other approaches tend to require more than one image, or have special hardware requirements.

Stereo vision uses two different viewpoints provided by two different cameras. The offset between them, in the horizontal or vertical direction, can be used to extract depth information. This is done by finding the pixels in both images that correspond to the same point in the 3D world, and then checking the disparity between them. This can then be used to calculate the distance.

Stereo Vision can be aided by using structured light to aid stereo vision in its task to find correspondences. This is done by for example creating a grid with lasers, and using the way in which the grid is deformed when it hits an object to model a 3D environment.

2.3 Image acquisition

For static stereo vision with moving objects, as well as dynamic stereo vision with static objects, it is essential that both cameras capture images at the same time. Otherwise, the epipolar lines might not line up, and calculations of disparities will be wrong due to movement. This means that there needs

to be provisions for getting the image acquisition done in a synchronized manner. Using regular consumer USB cameras, this can be very difficult. The cameras typically only have provisions for starting a capture, and stopping a capture. There is typically not possible to capture single frames, feeding it a clock signal or enabling any internal synchronization routines.

Various firewire cameras, following the IIDC standard allow for feeding the cameras a clock signal, which can be used to capture frames in a synchronized fashion. Some professional cameras, such as used for the system presented in this report, support auto synchronization.

The best way to synchronize common consumer USB cameras, is to make sure that the cameras are on their own USB buses. This way requests and frames can be recieved in a concurrent fashion. The synchronization will still often drift slightly with varying CPU load, and since there is typically very little relation between when a frame is captured and when it reaches the system buffer, trying to make both cameras capture frames at the exact same time might actually make things worse.

2.3.1 Color conversion

Most digital cameras used these days use a Bayer filter infront of the CCD sensor. The filter is a mosaic infront of the CCD sensor, and only lets through the relevant wavelengths for each pixel. The filter pattern is 50% green, 25% red and 25% blue. This means that for each pixel, only one of the relevant colors will be in the same pixel position, the rest is taken from neighbouring pixels. The side effects of this is that for a given pixel in the image, the colors are often interpolated from neighbouring pixels.

There are numerous different demosaicing algorithms available, with different algorithms being suitable for different tasks. The algorithms have been split into two distinct groups. Fast algorithms, and slow algorithms.

Slow algorithms are typically very good at reproducing the colors accurately, and various implementations such as VNG, AHD and PG are available. They are most used to import raw images from SLR cameras, in which visual quality has much higher priority than speed.

The fast algorithms, which are the most relevant for dynamic stereo vision, tend to either focus on visual quality, or precision. The former kind such as nearest-neighbour or bilinear demosaicing causes the image to become somewhat blurred, while linear algorithms that focus on precision causes the image to become grainy.

This means that to achieve both, it might be better to rely on the camera itself to convert the Bayer filter into YUV, or spend considerable time trying to find something that works better. Also, since the bayer filter is typically non-removable, it means that if the cameras are to be used for computer vision tasks without color information, using a dedicated Black and white camera will provide better image quality than a color camera.

A lot of professional cameras support getting the raw sensor data directly (examples are SLR Digital Cameras, as well as various firewire cameras), while others only provide MJPEG[2] or YUV.

MJPEG is unsuitable for typical computer vision tasks, as it causes the same visual artifacts as regular JPEG images, so that only leaves getting raw Bayer filter data, or YUV.

The YUV format has the following components:

- Y is the intensity in the image, and only using this channel essentially results in a grayscale image.
- U, which can also be written Cb, is the Chromatic Blue, which indicates the "blueness" of the pixel.
- V or Cr, is Chromatic Red, and is the "redness" of the pixel.

The YUV format is much more resilient to noise than RGB, and it also makes it easier to do color segmentation.

For this thesis, older cameras with YUYV color palette were used, and later replaced with Firewire cameras using the YUVY format. Both YUV formats only have 16 bits per pixel: The result is that for two pixels in RGB space, you have two Y components, but only one U and V component.

$$| Y1 | U | Y2 | V | \rightarrow | R | G | B | R | G | B | \tag{2.1}$$

What this means is that in grayscale, the edges are very sharp, while sudden changes in color in the horizontal direction may result in some visual artifacts.

There are many formulas and standards for the YUV color space. The most commonly used conversion formula uses the NTSC space. It enables fast conversion due to only relying on integer maths. Another convenience is that it can then easily be implemented on simple microcontrollers without resorting to floating point operations.

[2]MJPEG is a collection of codecs that do JPEG compression on each frame in an image

The formula for conversion into RGB using NTSC color gammut is as follows:

$$R = (298 * (y - 16) + 409 * (v - 128) + 128) >> 8 \quad (2.2)$$

$$G = (298 * (y - 16) - 100 * (u - 128) - 208 * (v - 128) + 128) >> 8 \quad (2.3)$$

$$B = (298 * (y - 16) + 516 * (u - 128) + 128) >> 8 \quad (2.4)$$

The results from these equations has to be saturated, so that all values are between 0 and 255.

2.4 Calibration

2.4.1 Undistortion

Undistortion is the act of mathematically removing radial and tangential distortions. There are many other forms of distortions, but these have minor effect for an image processing viewpoint and are not considered further. In theory, it is possible to design a lens that will introduce no distortions. However, in practise no lens can be manufactured flawlessly.

2.4.2 Radial distortions

Radial distortions occur because of the shape of the lens. There are essentially two types of radial distortion; Barrel distortion and Pincushion distortion. The effect of Barrel distortion is that the image manification is greater in the centre of the image than around the edges. The result is essentially that a plane will look like it has been projected onto a sphere.

Pincushion distortion, which is primarily only seen with older or cheap telephoto lenses has the opposite effect. Objects far away from the image centre are magnified more than objects in the middle of the image.

The distortion can be approximated by:

$$x_{corrected} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \quad (2.5)$$

$$y_{corrected} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \quad (2.6)$$

Where k is a constant, r is the radius from the center, and x,y is the image coordinates.

2.4.3 Tangential distortions

Tangential distortions are usually a result of the assembly process of the camera. This is due to the imaging sensor not being perfectly parallel to the imaging plane[8]. It can be characterized by the Taylor series:

$$x_{corrected} = x + [2p_1y + p_2(r^2 + 2x^2)] \tag{2.7}$$

$$y_{corrected} = y + [p_1(r^2 + 2y^2) + 2p_2x] \tag{2.8}$$

2.4.4 Rectification

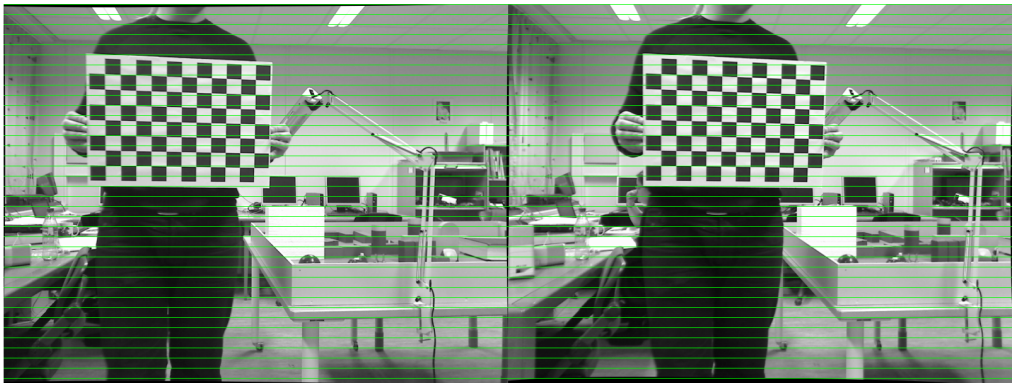


Figure 2.3: Example from rectification

There are many objects that can be used for stereo rectification, and there are examples of using three-dimentional objects that do not require you to use more than one image for calibration. The most practical choice however, unless you have the hardware necessary, is a regular pattern such as a chessboard[13].

The calibration is done by taking multiple images of the chessboard with both cameras at the same time, and looking at where each corresponding point is in the images. By using a lot of images, and making sure that you stretch the image in such a way that the epipolar lines line up in every single image set, you can calculate how the pixels in each image needs to be stretched for all the epipolar lines to line up. By these calculations you can create some matrices that can be used to remap the pixels in incoming images rather quickly.

Another factor, which is desireable when doing reprojection, is that the viewing area should be as large as possible. One algorithm to achieve this is Bou-

quet's algorithm[13, p433]. The algorithm attempts to maximize the shared viewing area, while minimizing the amount of change caused by reprojection.

2.4.5 Epipolar geometry

Once the image system has been calibrated, a horizontal line in the left image should be at the same level in the right image[26]. This means that when searching for corresponding features in both images, it is only necessary to search the same epipolar line, reducing a 2D search space to 1D. If the camera coordinate systems are only translated parallel to each other, it holds that $Z_l = Z_r = Z$ and the triangulation can be simplified to:

$$d = X_l - X_r \tag{2.9}$$

$$Z = \frac{f * b}{d} \tag{2.10}$$

Where d is the distance between the cameras, f is the focal length and b is the disparity.

The size of the viewing area in the x and y direction can be calculated by:

$$pos_x = 2 * Z * tan(\frac{\delta}{2.0}) \tag{2.11}$$

$$pos_y = 2 * Z * tan(\frac{\theta}{2.0}) \tag{2.12}$$

Where δ is the horizontal field of view, while θ is the vertical field of view.

2.5 Block Matching

A block is a collection of pixels, typically in an *n-by-n* grid. Block matching use the assumption that corresponding pixels have very similar intensities. One pixel is not enough because there are typically many potential candidates, and therefore neighbouring pixels also have to be checked.

An approach to reduce the computational strain is to match large blocks of pixels, and use the information to perform more fine-grained matches with smaller blocks.

In its simplest form, block matching merely requires that you use some similarity measure to measure a set of pixels in each image, decide if the blocks correspond or not, and then calculate the disparity between them to find the distance.

2.5.1 Similarity measures

Similarity measures in block matching are used to determine how well a block in one image matches a block in the other image. This is typically done by using some metric for the difference between two blocks of pixels.

SAD - sum of absolute differences

This metric simply takes the difference of each pixel in the block to be matched, and sums it into a single value. Two commonly used methods are described below.

$$SAD = |(A1 - B1)| + |(A2 - B2)| + ... + |(An - Bn)| \tag{2.13}$$

One thing that makes SAD especially appealing when dealing with modern x86 processors supporting SSE4.1[3], is that there is an optimized instruction just for doing SAD, *mpsadbw*, mostly designed for decoding HD video.[18]

MSE - mean squared error

The MSE is one of many approaches to quantify the difference between an estimator, and the true value of what is being measured. With block matching the estimator is typically the block in one image that you want to find in the other.

$$E = \frac{((A1 - B1) + (A2 - B2) + .. + (An - Bn))}{n} \tag{2.14}$$

$$MSE = E^2 \tag{2.15}$$

2.6 Edge detection

A major part of feature-based stereo vision is edge detection. An edge is a transition between two surfaces, genereally a sudden shift in intensity and/or color. Edge detection therefore typically entails finding these sudden shifts in intensity, and based on some threshold, mark them as edges.

[3]SSE4.1 is an instruction set supported by AMD K10 processors and Intel Core microarchitecture and newer.

2.6.1 Canny Edge detector

The Canny edge detector[9] is based on three objectives; a low error rate, that the edge points should be well localized in the image, and that there should be a one and only one edge response. Canny worked with expressing these three criteria mathematically, attempting to find optimal solutions for them.

The fact that edge points should be well localized is important when calculating disparities for edges, because otherwise, the disparities could be more inaccurate than aliasing issues allow. This makes the Canny edge detector a better choice than more simple edge detectors, as it is crucial to for the edges to be correctly localized.

A typical Canny edge implementation has the following steps:

- Calculate gradient in x and y-direction
- Calculate direction of edges based on gradients, and also the gradient magnitude.
- Non-maxima suppression (removing directions that are not maximized)
- Based on two threshold parameters, high and low, detect and follow edges.

The gradient can be calculated by using the Sobel mask for y-direction, and x-direction, while the gradient direction can be calculated by $arctan(Sx, Sy)$. The non-maxima suppression ensures that there is one and only one edge response. This is done by checking the gradients on both sides of an edge, making sure it is the most prominent one. Then a search for gradients above a specified high threshold is done, and the edge is traced, using the edge direction information, with each pixel being marked, as long as the gradient is above a certain lower threshold. This is repeated until the end of the image is reached.

2.6.2 Edges in multi-channel images

Analysing edges in multi-channel images is significantly more difficult than in greyscale images. The difficulty lies in the fact that the different channels might contain conflicting information. An edge in one channel might not appear at all in another channel, and they might even have opposite directions[16, p399-401].

2.7 Circle detection

Circles are a rather simple geometric shape, in which the edge is a fixed distance from the centre. This can be exploited in several ways to recognize and detect circles.

Provided a region segmentation, for example by using color segmentation, one could exploit these properties by doing various calculations on the region itself. One approach is to calculate the compactness of a region. This is achieved by calculating the distance around the region squared, divided by the area of the region. The smallest possible compactness value is a circle[20, p356].

$$compactness = \frac{(region\_border\_length)^2}{area} \tag{2.16}$$

It is also possible to calculate the eccentricity, which is the ratio between the minor and major axis. In a perfect circle, this should be equal to 1.0[20, p355].

Both these calculations combined with a successful segmentation of a circle, has the potential to work quite well. The problem lies in accurately segmenting out the regions without the process being too slow.

An entirely different approach, which does not require segmentation, but just edge detection, is the Hough transform.

2.7.1 Hough Transform

The Hough transform[15] is a technique that locates shapes in images. It can be used to extract lines, circles and ellipses from images. The algorithm defines a mapping from the image to an accumulator space, also known as the Hough space. This is achieved by using a function that describes the shape one wants to match[23].

In the Hough transform for circles, the basis is the equation for a circle 2.17.

$$(x - x_0)^2 + (y - y_0)^2 = r^2 \tag{2.17}$$

The Hough transform for circles, draws a circle for each edge pixel into the hough space, increasing each point by one, which is the votes for a center with radius r for each edge pixel. This means that it is necessary to extract the edges before running the Hough transform, using an appropriate edge detector.

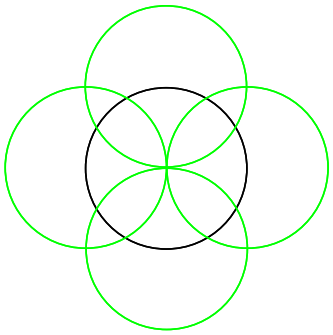


Figure 2.4: Hough transform, fixed radius.

The algorithm is as follows:

```
for each edge pixel in image:
        for angles 0 to 359:
                x0=round(x-r*cos(angle))
                y0=round(y-r*sin(angle))
                accumulator(y0,x0)++
```

Completing this, the accumulator should have peaks where circles are most likely to be present. All that is then required is to select an appropriate threshold. Care should be taken when selecting the threshold value, as false positives should be avoided, while avoiding to miss real circles.

Parameter decomposition

The Hough transform as described above is very slow. In order to obtain a reasonable run-time another approach needs to be used.

One approach is to take advantage of the fact that the line perpendicular to a tangent of a circle should point towards the center. So by drawing a line along the normal of each tangent, there should be a peak of votes close to the center of the circle. One can then figure out the size of the circle by running the inner-loop of the algorithm described in 2.7.1 for a selection of radiuses[23, p212-216].

2.7.2 Fast Finding and Fitting

The Fast Finding and Fitting algorithm is published by Marshall and Yi, 2000[31], and describes a method to find circles based on the geometric symmetry of a circle. It makes a comparison of different approaches to HT, and comes with a few insights. It exploits the fact that circles are symmetrical along the vertical and horizontal axis.

```
* find edge pixels.

for all horizontal pixels in F:
    for all vertical pixels in F:
        find all edges:
             Find the middle between them
             add the middle to accumulator space G(y,x).

use HT on accumulator space G:
    extract all possible vertical symmetry axises Lv

For all pixels in edge image:
        If pixel is symmetric to Lv:
           put pixel into F_h

for all vertical pixels in F_h:
        for all horizontal pixels in F_h:
            find all edges:
                 Find the middle between them
                 add the middle to accumulator space L(y,x).

use HT on accumulator space L:
    extract all possible horizontal symmetry axies Lh

For all pixels in F_h:
        if pixel is symmetric to Lh:
           put pixel into F_hv

For all pixels in F_hv:
        Judge if they are circles.
```

2.8 Color recognition

Human eyes are very good at detecting if a surface has an uniform color or not, and often fills in the blanks when there is a lot of specular reflection. The problem with this is that most algorithms typically look at colors from a local viewpoint. For example, if a blue surface is strongly illuminated, the result may be that a lot of pixels might actually be grey in an image instead of blue, even though we see the region as blue.

There are several ways to combat this. One method that reduces the problem is normalization. There are a lot of different approaches to normalizing an image. Among the most common ones are:

1. $r = \frac{R}{B}$
2. $r = \frac{R}{(R+G+B)}$
3. $r = \frac{R}{\sqrt{(R^2+G^2+B^2)}}$

According to Jähne[16, p156] method 2 has a higher degree of accuracy than the other approaches.

2.8.1 YUV-space

Since YUV space essentially puts all intensity information into one channel, and color in the two others, the normalization requirements are less than in RGB space. This is because RGB-space has the intensity information stored in all 3 channels.

2.8.2 Chromatic space

Chromatic space is similar to an intensity histogram, except you map color channels in a 2D space instead of intensities in 1D. With an RGB image as basis, the best results are achieved by mapping the red channel along one axis and the green axis along the other. The blue channel is discarded, as most of the color information is deemed to exist in the red and green channels. With a YUV image, you can map the chromatic red along one axis, and chromatic blue along the other for similar effect.

Before converting to chromatic space, Jähne[16] recommends to apply a smoothing filter to the image, to make the individual pixels more representative of the true color of the objects and less susceptible to noise.

2.9 Deterministic image restoration

Deterministic image restoration is applicable to images with little noise, and a known degradation function. The idea is that the Fourier transform G of the degraded image g has the following in relation to the desired image f:

$$G = H \cdot F + N \tag{2.18}$$

Where H is the Fourier transform of the degradation function, F is the Fourier transform of the undegraded image, and N is the Fourier transform of the additive noise.

In a robot vision domain, where the angular velocity is known and it is assumed that the robot only rotates while standing still, this can be used to predict the degradation due to linear motion blur. Furthermore it is assumed that the objects that require our attention are primarily stationary.

According to Rosenfeld and Kak[27] the relative moment of camera and object has the following properties:

Relative movement between the object and camera during the shutter open time t causes the image to be smoothed. if the relative moment is at a constant speed v, the fourier transform H(u,v) of the degradation caused in time t is given by:

$$L = vt \tag{2.19}$$

Where L is the length in pixels the image is blurred.

$$H(u,v) = \frac{sin(Lu\pi)}{Lsin(u\pi)} \tag{2.20}$$

Which, centered and normalized for the image size is:

$$y = \frac{u-w}{2} \tag{2.21}$$

$$H(u,v) = \frac{sin(\frac{Ly\pi}{w})}{Lsin(\frac{y\pi}{w})} \tag{2.22}$$

Where w is the width of the image, and y is the centered u.

2.9.1 Inverse filtering

Inverse filtering merely uses the properties of equation 2.18, ignoring noise. We get the equation[20, p165]:

$$F = G * H^{-1} \tag{2.23}$$

This works reasonably well if we can approximate the degradation function, and no noise is present. All the information for each pixel should still be stored in the image, just degraded. This of course works on the assumption that we are not limited to discrete pixels with only 8 bits per channel.

The problem is that even though no noise is visible in the input image, the frequencies of the invisible noise exceeds the signal in the high frequency areas of the Fourier transform. Therefore, unless a threshold is applied to the inverse function, the result is often just noise.

2.10 Mathematical Morphology

The basic idea in binary morphology is to probe an image with a simple, predefined shape, drawing conclusions on how this shape fits or misses the shapes in the image. Frequently used shapes, or *structuring elements* are the cross, circle and the rectangle. The structuring element is used when probing the image, and is used to determine which parts of the image should be checked when performing a morphological operation. Structuring elements have two properties, shape and size.

There are essentially two basic morphological operators, dilation and erosion. Most of the other morphological operators can be defined as a combination of erosion and dilation along with various set operators. Erosion removes border pixels from objects, by removing bright pixels not surrounded by other bright pixels. This will enlarge holes and small gulfs in the regions, and also separate two regions that are barely touching each other. A side effect is that all regions will decrease in size. Dilation however, will make regions larger. This is done by probing the image with the structuring element, and filling areas covered by the structuring element if a bright pixel is found.

Opening is erosion followed by dilation. It will remove sharp edges and smooth object boundaries while preserving its size. In the case of a greyscale image, it will darken bright regions that are smaller than the structuring element, while preserving larger features in the image.

Closing is dilation followed by erosion. It will remove dark features smaller than the structuring element from the image, while preserving the size of regions. In the case of greyscale images, it will fill small dark regions, while preserving larger regions[12].

2.11 Skeletons

By skeletonizing, a region is reduced to a graph by thinning. This can be used for a large variety of image processing problems, from inspection of printed circuit boards to counting asbestos fibers in air filters[25]. A simple implementation just iterates around a region, reducing it one pixel at a time, till the region has thickness of one pixel.

2.12 Contour properties

Once you have a closed contour, a lot of different properties can be calculated. The bounding box can be calculated by figuring out the rightmost, leftmost, uppermost and lowest pixels in a contour. This can then be used to give a rough estimate of where the region is in the image.

2.12.1 Rotating calipers

Rotating calipers is an algorithm[30] for solving various geometric measuring problems. The name comes from the analogy of rotating a spring-loaded caliper around a convex polygon. Every time one blade of the caliper touches the edge of a polygon, the other side of it touches the opposite blade. This forms antipodal pairs around the entire polygon, which can be used to find the narrowest and widest part of the polygon. It is then possible to find the minimum enclosing rectangle.

When the minimum area enclosing rectangle is found, it can then be used to calculate the minor axis, major axis, orientation and a good approximation of the center of the region.

The center is simply the point in the middle between two sets of points (top-left and bottom-right or top-right and bottom-left) in the rectangle. The minor axis is then shortest of the sides in the rectangle, while the major axis is the largest. The orientation is given by the orientation of the entire rectangle.

The circumference can be calculated by simply counting how many points the contour consists of.

2.13 Color segmentation

One approach for color segmentation is to go through the image with a patch of size nxn, and comparing the variance with a slightly larger patch of size $n + 1xn + 1$, and doing this for several iterations.[7] This algorithm makes sure the areas with similar color are "flattened", and the different areas are separated from each other.

Pseudo-code:

```
for each pixel P in image:
    normalize P
    until no significant change, do:
      for each pixel P in image:
          calculate v1= the variance over a 3x3 square, with P in the center
          calculate v2= the variance over a 5x5 square, with P in the center
          if v2 < v1 + tolerance:
          for each color component P_c of P:
              P_c= the average of c over a 3x3 square, with P in the center
```

2.13.1 Watershed segmentation

Grey-scale images can be seen as topographic maps, in which pixels with high intensity are hills, while low intensity values are basins. Watershed segmentation uses this information to fill basins in the image, in which each basin can be seen as a single region. The length of a gradient can be used as height information, further improving results. One approach to watershed segmentation is to find the regional minimas and build up a water source from each regional minima, till the watersheds meet.

2.13.2 Gradient of greyscale images

The gradient is a measure of change in a specific direction. The most common directions to calculate it in the field of computer vision is horizontally, vertically and diagonally. Since images are typically discrete, and calculating it very accurately is difficult, the Sobel operator is most used. To use it, the image is convolved with the sobel operator, which consists of integer values, resulting in a gradient image. To get a gradient in a specific direction, the horizontal or vertical sobel mask is used, and to get the gradient strength, the result of both calculations can be combined.

horizontal gradient:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

vertical gradient:

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

2.13.3 Color difference

One metric for color difference is the sum of absolute differences between the different color components of two pixels, calculated by:

$$difference = |r1 - r2| + |g1 - g2| + |b1 - b2| \tag{2.24}$$

The same metric can be used for I1,I2,I3 color metrics, which are calculated from R,G,B values in the following fashion, proposed by otha et al[**?**].

$$I1 = (R + G + B)/3 \tag{2.25}$$

$$I2 = (R - B)/2 \tag{2.26}$$

$$I3 = (2G - R - B)/4 \tag{2.27}$$

The difference can then be found by:

$$difference = |I1 - I1| + |I2 - I2| + |I3 - I3| \tag{2.28}$$

Equation 2.28 is less prone to intensity variances than 2.24, but calculating the I1,I2,I3 values require more computational power. This means that it might be better to separate regions, but not so good at separating two objects of essentially the same color.

Chapter 3

Environment

3.1 Enviroment overview

This project was done as part of a solution to track balls and cylinders for a mobile robot competing in Eurobot 2010[4]. The playing field is 3 m $\times$ 2 m, and the details of the configuration can be seen in figure 3.1.

Essentially there are two competing teams, one starting in the blue corner, and one in the yellow corner. The task is to pick up as many of the red and yellow balls, as well as the white cylinders as possible.By the end of a round, all balls must be moved into the goal location at the bottom of the table, on the opposite side of the starting position. There are also black cylinders present on the table, but they cannot be picked up. This means that there are four different objects that need to be tracked and picked up:

- Red balls with a diameter of 100 mm, lying on the green floor.
- Orange balls on top of cylinders, also with a diameter of 100 mm.
- Black and white cylinders, placed on the ground, in vertical position, 50 mm in diameter.

There are 14 red balls, 7 black cylinders, 11 white cylinders and 12 orange balls. The balls are in fixed positions, while the white and black cylinders can have a number of different configurations. The complete details can be seen in the Eurobot 2010 rules[4].

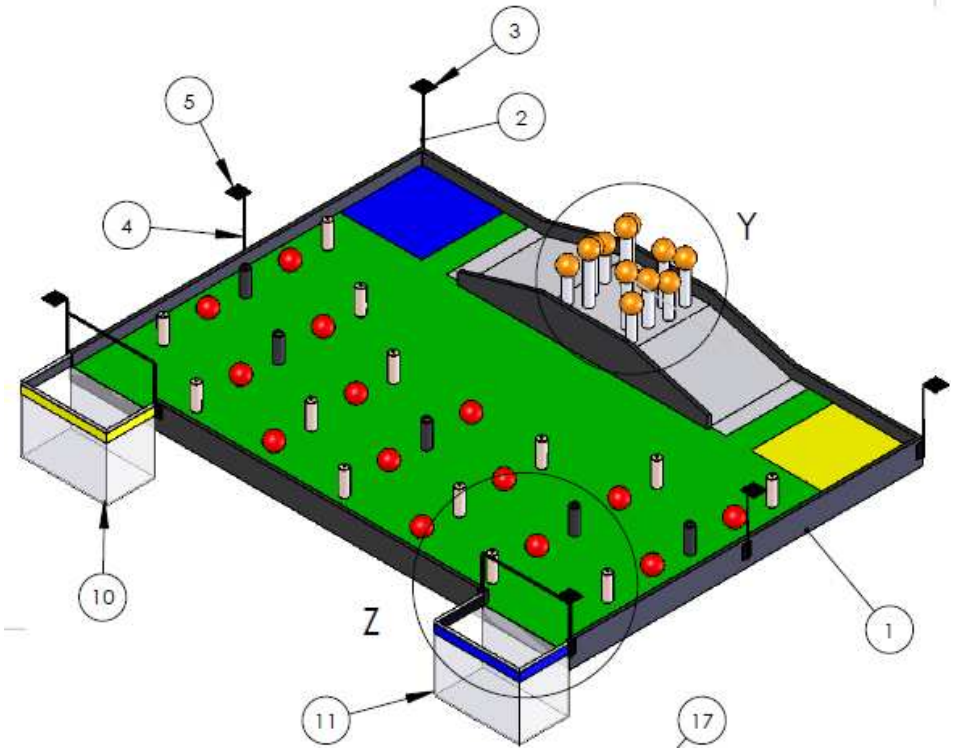


Figure 3.1: Playing field, Eurobot 2010.

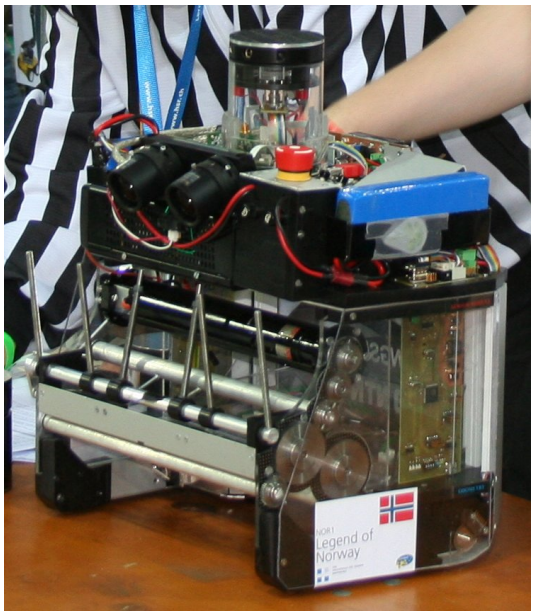


Figure 3.2: The Robot

3.2 Requirements

3.2.1 Functional requirements

The functional requirements are based on the Eurobot 2010 rules[4].

1. Must be able to find the location of as many balls as possible on the field.
2. Must be able to find cylinders, and distinguish black cylinders from white cylinders.
3. Must be able to distinguish orange balls from red balls.

3.2.2 Non-Functional requirements

Robustness

1. Should be easily adjustable to different light conditions.
2. Must be able to calculate location while in motion.

Performance constraints

Processing an image should take no longer than 500 ms. It is not necessary to update the information more often, as the artificial intelligence, which is responsible for processing the data, spends approximately 3 seconds to update the plan. The primary advantage of having more frames per second is that you might be able to cover a larger area of the playing field if the robot is rotating very fast.

Size

According to the Eurobot regulations, the robot cannot have a circumference larger than 1200 mm. So ideally, the computer and cameras should take up as little space as possible, to make room for other parts of the robot. This includes machinery for gathering, delivery and storage of player elements, batteries and driving systems.

This means that for a square object to fit into the robot, it should not be larger than $300mmx300mm$. When you add in a proper casing for the computer system, the system itself should not exceed $200mmx200mm$.

3.3 Color properties

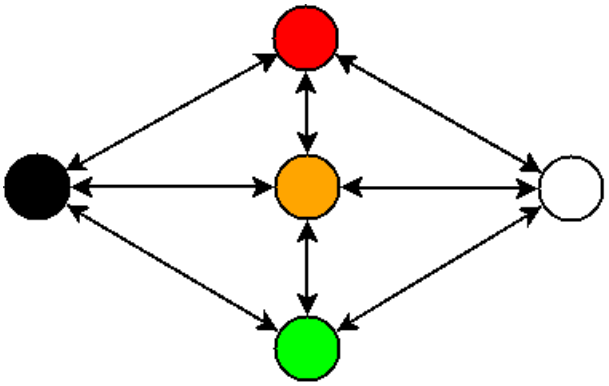


Figure 3.3: Colors on the playing field

The colors on the playing field are orange, red, green, black and white. The ground at the starting positions are yellow and blue instead of green, but these can be filtered out easily, and they are not very important to see.

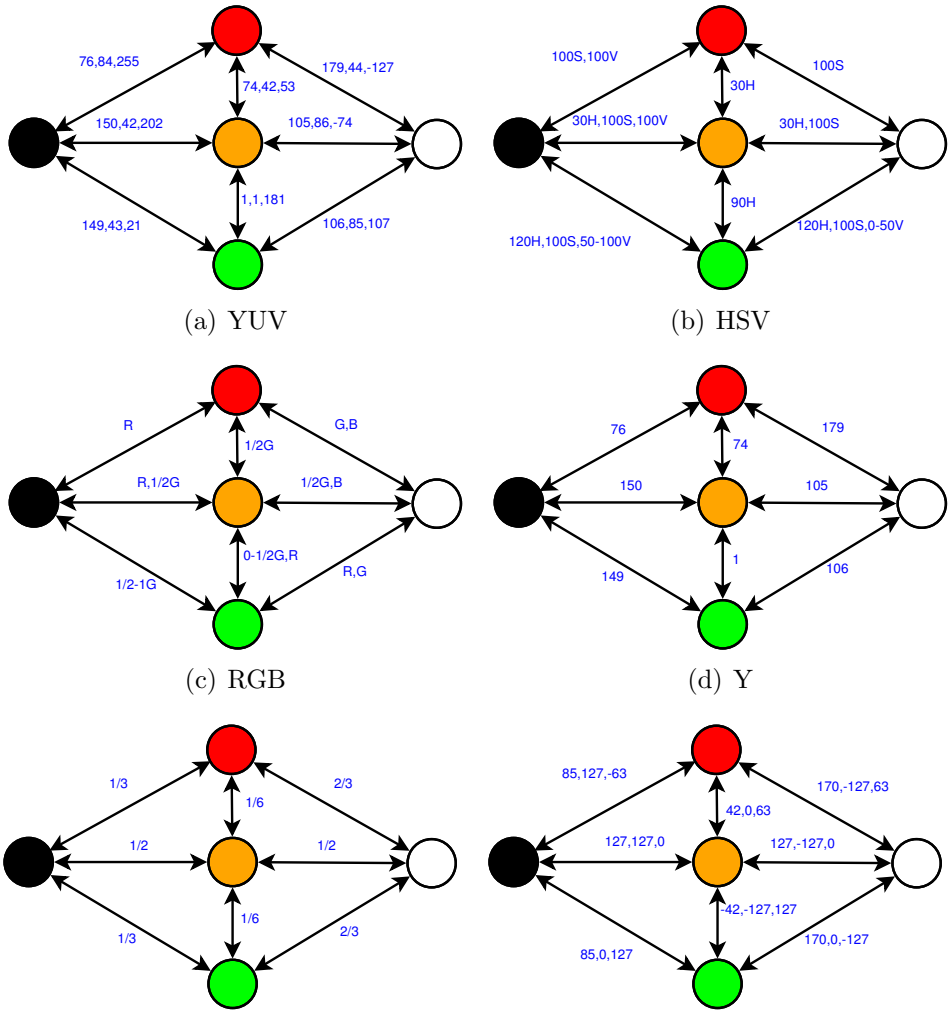
Looking at the intensity alone, there are a few issues. The orange seems to be essentially the same intensity as the green background, and therefore simply using the intensity gradient works very poorly. No single metric in RGB, YUV or HSV space is able to differentiate the colors properly.

3.4 Balls

The balls are round, with a diameter of 10 cm. They can be either orange or red. An image of a ball can be seen in figure 3.5.

3.4.1 Roundness

The balls are too soft to be perfectly round, gravity will compress them some against the ground. This causes them to be rather unsuitable to for use with Hough Transform for circles, as the balls look elliptical unless seen directly from above. The issue is not an issue with the orange balls, as these are made of a harder material.



(a) YUV (b) HSV

(c) RGB (d) Y

(e) Black & White (f) I1, I2, I3

Figure 3.4: Color difference graph for different color spaces.

3.4.2 Reflection

The balls are extremely reflective, probably due to the fact that they are juggling balls, designed to be flashy and look nice. This causes some problems for edge detection as well as segmentation. For example using edge detection, the difference between the general color of the ball, and the reflective regions, is much higher than the difference between the red color and the background. Another problem is that most of the cylinders are white, which means that just ignoring white is not possible. Using region segmentation, the same issues generally apply, as you are still reliant on change of color or intensity

Figure 3.5: Red ball against green background

in the image to separate regions.

One possible solution to this is to find all the white regions, and check if they are surrounded by a red border. Alternatively it is possible to just trace around all red regions, and ignore the inner parts. An issue with tracing around the regions is that sometimes the reflective areas are so close to the borders of the region that it is not possible to go around them.

3.4.3 Color

Orange and red is close to each other no matter what metric you use as a basis. In HSV, the hue value of orange is right next to red. Also, since red is the largest component in RGB for both colors, the hue will be considered red for both. Using RGB, the regions are very similar, except the Orange has a slightly larger green component. An issue with this is that the background is green, which means that the outer regions of the red balls will look very orange.

This means that the only sensible way to compare the color of the balls, is to find some metric for the color of the entire region, and not just its individual parts. This can either be done using a histogram and finding the tallest peaks, or just taking the average.

3.4.4 Shades

Shades in the image are in general rather easy to deal with, as normalizing the shades produces the right colors. However, the borders of the playing field, as well as about one third of the cylinders are black. In a typical scene

this would not cause many issues, because most of the light hitting the black regions would not be biased with a certain color component, meaning they could have been separated by comparing R:G:B difference. On the playing field however, because it is green, the light hitting the cylinders is also heavily biased towards green, meaning that the green shades and the black cylinders have the same color components.

3.4.5 Similarity

When using segmentation approaches that are dependent on changes in color or intensity, objects of similar (or same color) in close proximity to each other could cause issues.

For example if a black cylinder is close to the border of the playing field, the border and the cylinder will be considered the same region. This is also the case if two balls are very close to each other, unless something like the Hough circle transform is used.

One way to combat this is to use stereo vision. If you're able to determine that the border and the cylinder is the same in both images, a comparison of the two should be able to pinpoint the location of the cylinder. Another approach could be to use template matching, since all the shapes in the scene are available a priori.

3.5 Cylinders

The cylinders are white or black, and 20 cm tall, with a diameter of 5 cm. The black cylinders are fastened to the table, while the white ones are not.

3.5.1 Shape

The cylinders are solid, and made out of wood. The wood is not grinded down very smoothly, so there are significant grooves in the wood that could prove difficult for segmentation.

3.5.2 Placement

When the cylinders are lying down, the gradient along the surface of the cylinders become very strong, and this can cause issues with segmentation,

as the color difference in one part of the ball to another might be bigger than the difference between the background and the cylinder.

3.6 Occlusion

Because there are a lot of objects that are partially occluded by something else in the scene, handling occlusion is very important. This means that it might be a better idea to match red or orange regions in the image, rather than circular ones to recognize balls. It could also be difficult to separate two cylinders semi-occluded by eachother. Cylinders semi-occluded behind balls should not be difficult to detect.

3.7 Movement

Since the robot will move around at upto $1m/s$, with objects being very close, there will be a lot of visible movement. Blur is a major issue while rotating or turning. Because the robot can move forward while turning, it is not as simple as just removing horizontal blur. There are a few ways to deal with this:

- Use cameras that require low enough exposure time so that the blur becomes insignificant.
- Remove the blur using algorithms (in software or hardware).
- Use algorithms that are not as prone to blur.

Chapter 4

System Design

This chapter will present the implementation of the system, as well as the hardware used.

4.1 Hardware

Initially, the preferred hardware for this application would be a small laptop or notebook. Then you have a self-contained unit with its own power supply, screen and keyboard. But, due to performance and budget constrains this was quickly discarded as a viable option.

Ideally the system should be as compact as possible, as this leaves more room in the robot for the mechanical parts and storage. Power consumption was a concern, but because the robot only has to run for 1 minute and 30 seconds during the competition, and a few minutes in standby, this was not much of a factor when considering different options. We also needed to make sure that our batteries would be able to deliver the required power.

The following system components were selected:

- Intel Mini-ITX Socket LGA775 board (Intel DG45FC)
- Intel Core 2 Quad Q9550s (4 cores @ 2.83 GHz, 12 MB L2 Cache, 65 W TDP)
- 4 GB RAM DDR2 (2 x 2 GB)
- 16 GB Compact Flash storage (using SATA-CF adapter)
- PicoPSU M3-ATX 125W

- Firewire 400 (2 ports) PCI-E expansion card. (the top had to be trimmed to fit in the case properly)

The selected computer isn't based on the latest x86 designs (Core i7), but none of the newer processors with a power usage equal or less than 65W is able to deliver the same performance. The motherboard has no provisions for handling any CPU with a higher power requirement. The CPU itself runs quite cool when in use, but the system itself, due to being put into a very tight space, is running very hot. No stability issues were observed during operation

Under testing the entire system would use about 20 W when idle, and up to 80 W under full load.

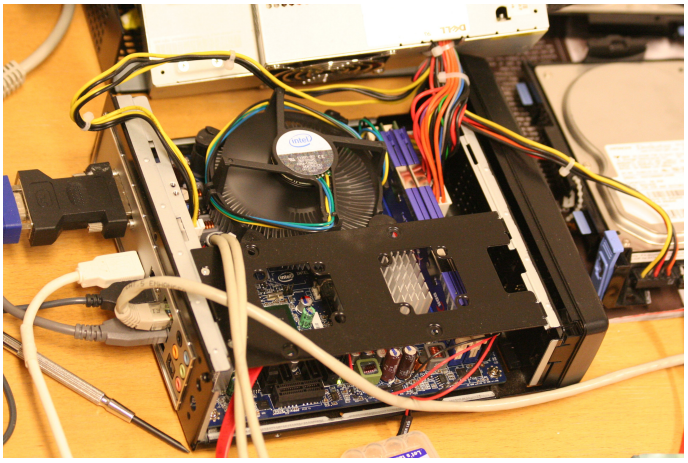


Figure 4.1: Overview of the computing device during testing

4.2 Old cameras

As a starting point it was decided to use the cameras from Eurobot 2009, because the color adjustment was very good, and the resolution was more than acceptable. The cameras were simple consumer cameras, primarily designed for laptop use. There exists a desktop variant with somewhat quicker focusing etc, but the size was prohibitive. The cameras used were also UVC[5]-compliant (USB Video Class), giving them good compatibility with Windows, Mac as well as Linux.

Model: Logitech, Inc. QuickCam Pro for Notebooks (046d:0991)
Supported resolutions: 160x120, 176x144, 320x240, 352x288, 640x480 (15

fps), 800x600 (15 fps), 960x720 (10 fps)
Interpolated resolutions: 1600x1200 (5 fps)

The cameras had a fairly long exposure time, which could be counteracted somewhat by increasing the gain and brightness, with the side effect that more noise was produced. Another problem was that the image was updated in a scanline fashion, so that with very fast movement, the top of the image was what the camera was seeing currently, while the bottom might have been something seen just recently, resulting in a "wobbly" effect. The effect was only visible during fast movements.

4.3 New Cameras

Since the old cameras had too long exposure time to be suitable for fast movement, and the synchronization using threads with USB cameras worked less than optimal when the system had very varying load. It was therefore decided to get new Firewire cameras with auto-synchronization and better optics.

Unibrain Fire-i Board Pro - 1024x768 @ 36 fps native resolution.

The cameras were used with a shutter time of 6 ms, manual focus, hue, saturation, sharpness, brightness, gain, contrast, f-stop and zoom. Auto-synchronization is supported as long as the cameras are on the same Firewire bus, and a specific non-IIDC[1], a compliant register is set.

Not much is known of how the auto-synchronization works, as it is a proprietary system, but the important part is that it does take approximately one minute before the frames are synchronized after you start a capture. During testing the time to synchronize mostly seemed to be related to the framerate, so it took approximately twice as long for the frames to synchronize using 7.5 fps vs. 15 fps.

The cameras have a status register telling when the images are synchronized, and this can be used to ensure that the cameras are properly synchronized. However, the frames are often more than sufficiently synced long before the status register is set, so manual checking can be preferable to waiting.

The final system ended up using 1024x768 @ 7.5 fps with YUV format, but Bayer pattern with 8-bit or 12-bit precision could be captured directly from the camera.

[1]IIDC (Instrumentation & Industrial Digital Camera) is the FireWire data format standard for live video, and was designed for machine vision tasks

Due to the bandwidth constraints of the Firewire bus, the acceptable resolutions were 1024x768 @ 15 fps with 8-bit Bayer filter, or 1024x768 @ 7.5 fps with 16-bit YUV (or 12-bit Bayer using 16-bit datatypes). The color reproduction was better with the built in YUV conversion than with 7 different tested debayering algorithms, and it was decided that higher resolution was preferable to higher framerate.

Other notable features is that an electronic shutter can be used with the cameras, and this could have been used in conjuction with the propulsion system on the robot. This would have made it possible to only take pictures when the robot was not moving, avoiding blur and synchronization issues.

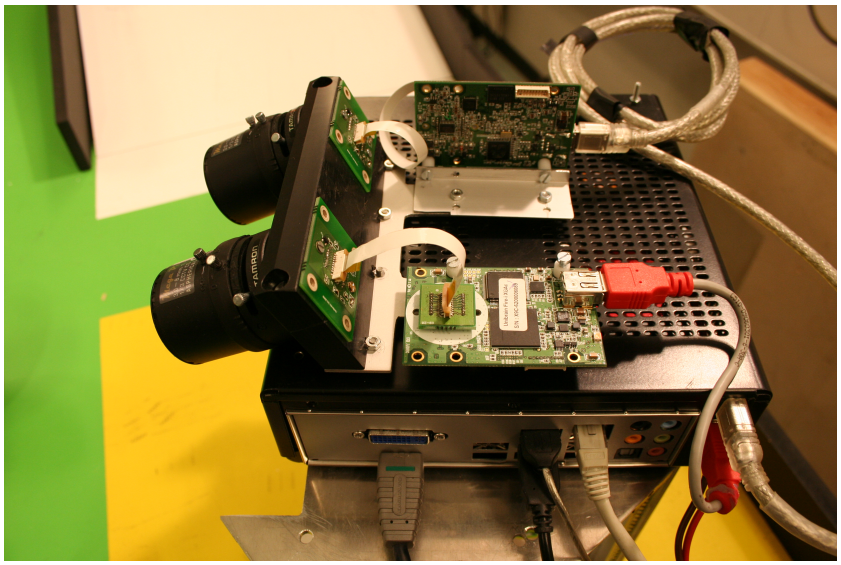


Figure 4.2: The cameras mounted on the computing device

4.3.1 Noise

The CCD sensor is on a separate PCB connected with a 16-pin flat cable, and this makes it rather prone to external noise. Using two small 20 cm Firewire cables also causes major noise issues, with diagonal lines as well as salt and pepper noise.

This is not very fortunate as the space available on the robot is severely limited. It is possible to avoid it by placing one camera PCB in a 90 degree angle facing away from the other, as well as replacing one of the short cables with a longer 1.8 m cable. It is possible that the issue could be resolved with

a more suitable 50 cm cable, but due to time constraints this hypothesis remains untested.

The cables themselves were also prone to breaking easily, so care must be taken when handling them.

4.3.2 Lenses

The Firewire camera has the possibility of using C- and Cs-mount lenses. These are quite common for security appliances and machine vision research.

The main requirements for the lenses was a focal length below 5mm, giving a suitable view angle, and a large shutter opening. The suitable view angle was primarily to be able to see atleast half the playing field at the starting position, but also to see as many elements as possible at the same time. Another factor was trying to avoid having a too small focal length, as the distortions of the lens become rather nasty at very wide angles. The distortions are rectified in software on this system, but a very large view angle would still cause less precision.

A low f-stop is very important, as more light on the CCD chip means a lower shutter time is required to capture frames. It also reduces the need to increase gain to amplify the signal, which reduces noise issues.

The solution was a lens with the following properties:

- Manufacturer: Tamron
- Model name: Mega Pixel M13VM308
- Focal Length: 3-8 mm (21.6mm-56mm in 35mm equivalents with a 1/3" sensor)
- Iris: F/1.0
- Manual Focus, Manual Iris, Manual Zoom

Since all the properties of the lens are manually adjustable, it means that the performance of the lens in different conditions is very predictable. For proper calibration of the setup, a fixed zoom and focus is necessary, as changing any of them would require recalibration. The iris could be adjusted easily, and with varying light intensities, it provides a quick way to adjust the brightness.

Having a very open iris caused the items to become slightly blurred, but there was no noticeable performance degradations of the algorithm itself.

To get the zoom and focus as similar as possible, a caliper can be used to measure the distance from the front of the lens housing to the lens on both cameras, and then adjusting the zoom untill the image is sharp for both cameras. The cameras have a very noticeable barrel distortion, most pronounced at the lower focal lengths, but this was easily rectified in software.

4.3.3 Firewire interface

Since there was no firewire card onboard, a small PCI-E[2] low profile card was connected to the PC. The only noticeable issue with the firewire interface was that if some power saving features for peripheral devices were not disabled in the BIOS, many of the frames arriving became corrupted unless the frame rate was set very low. Without the power saving enabled, no corruption issues were apparent, and the capture setup failed gracefully with an "out of bandwidth" message if any unsupported resolution and framerate was selected.

One thing worth mentioning is that for 1024x768 Stereo Vision, Firewire 400 Mbit is a bit on the low side, and cameras supporting Firewire 800 Mbit (or perhaps USB 3.0) would be necessary to get the full potential of the cameras. With 400 Mbit, it should be possible to do 36 fps at 1024x768 using 8-bit bayer filtering, or approximately 20 fps with 1024x768 with two cameras unsynchronized.

4.3.4 Camera mount

To have any chance of getting a proper stereo vision setup, a camera mount is vital. The camera mount with USB cameras used in the beginning of the project was far too large to be mounted on the robot. The advantage was that the distance between cameras could easily be adjusted between a few centimeters all the way up to half a meter.

The design requirements for the firewire camera mount was rigidity and sturdyness. The epipoles on the camera mount are fixed with a 8.5 cm separation, which is a good compromise between common view area and precision. The entire mount is made in Polyoxymethylene (POM), which is an engineering thermoplastic used in precision parts that require high stiffness. It proved very sturdy during testing, and it was one of the main materials used for

[2]The casing for the PC isn't designed for any expansion cards, so the firewire controller card had to be cut by 5 mm at the top.

the robot construction, in addition to aluminium and plexi-glass. The camera circuit boards are not screwed into the mount directly, but holes were made to fit the lens mounting bracket, and the mount is tightened around the mounting bracket, so that the CCD circuit board is behind the mount, while the lenses are connected in the front. Figure 4.2 shows the cameras mounted to the PC using the camera mount.

The camera mount fitted perfectly with the lens mounting bracket, and no noticeable degree of deviation was noticed during use. However, the lenses have a tendency to not fit perfectly with the mounting bracket, so if pressure is applied to the cameras, the epipolar lines can shift out of place, so special care has to be made to avoid ruining the calibration.

The most likely solution to this is to find a way to make the lens fit better against the mounting bracket, possibly using thread seal tape or locktight. Since the CCD is approximately 5 mm along the long edge, and there are 1024 pixels, it is obvious that extremely small shifts in movement will cause noticeable changes in the image positions relative to eachother.

4.4 Software

4.4.1 Operating System

For the operating system, we decided to use Ubuntu Linux, as this has been used for previous years for the Eurobot competition. The drivers for the CAN[3]-bus, as well as everything else made in previous years, are verified to work on Linux. There are plenty of Linux distributions that might have been better suited, but due to time constraints, nothing else was tested.

4.4.2 Compiler

The Intel C++ Compiler 11.1[10] was selected as it generally generates much better performing code than GCC 4.x[2]. On Linux, it is free for private and non-commercial use. Also, OpenCV, which is a good library for computer vision applications, can use specific Intel-optimized routines to improve performance. The main performance advantage with ICC vs. GCC is that GCC can only auto-vectorize a small subset of what ICC is capable of. This means that certain tasks, even though the amount of operations is identical, can be

[3]Controller-area Network, originally designed so that microcontrollers can communicate with eachother without using a host computer

several times faster. All of this can also be optimized by hand, but this is tedious work and generates less readable code.

4.4.3 Libraries

OpenCV

OpenCV (Open Source Computer Vision) is a library of programming functions for real time computer vision[24]. It has general image processing functions, image pyramids, segmentation, geometric descriptors, camera calibration, machine learning, matrix mathematics and a lots of useful datatypes. It is BSD licensed, and a lot of routines are optimized by Intel(R). OpenCV[24] is used for:

- Rectification and calibration routines
- Load and store images
- Convenient data formats for images, matrices, contours etc.
- Display images
- Optimize remapping of incoming frames for rectification
- morphological operations (erode, dilate)
- rotating calipers for minimum rotated rectangle calculation
- Canny edge detection (own implementation included, but much slower)
- Sobel operator, gaussian blur, Hough lines implementation
- conversion from RGB→BW
- finding contours (Moore border tracing implemented, but the OpenCV implementation has a lot more features)
- drawing lines, circles and text in the images

OpenCV can also capture frames from cameras, but it is severely limited for selecting resolution, format and setting parameters such as brightness and sharpness.

Unicap

Unicap[3] provides an uniform interface to video capture devices. It allows for the use of different cameras using Firewire as well as USB. You can also set the resolution and format you want, and if desired, deal with all the decoding of the actual pixel data yourself. It was tested using both user buffers and system buffers. The former gives you greater control when you want to read frames, while the system buffer approach essentially just creates a message handler, and it sends you a message with an attached buffer when the frame is ready. The system buffer offers much lower CPU usage, and also resultes in much better synchronization between USB cameras. It was only used for the setup with the old cameras, as libdc1394, which will be described in the next section, proved to be much more suitable for Firewire cameras.

Libdc1394 2.1.2

Libdc1394 is a library that provides a high level API for controlling and capturing from cameras following the DCAM and IIDC specifications. The library is only supported on Mac OS X and Linux at the current time, so it is not very suitable from a cross-platform perspective. It is available under the LGPL license, which means that it can be linked to even from proprietary programs.

All conceivable propeties such as resolution, frame rate, white balance, gain, brightness, shutter time and so on are adjustable via this library.

One of the major features used unrelated to capturing frames from the cameras is the demosaicing routines. It supports nearest-neighbour, bilinear, bicubic as well as more advanced debayering like AHD, PG and VNG demosaicing. For Firewire cameras it is preferred over Unicap because the API is simpler, and more Firewire specific features are exposed.

The developer of the cameras, Unibrain, also offer their own API, but it is only available for Windows. The advantage with using the API made by Unibrain is that the algorithm used on the cameras is included in the proprietary API, and seems to work better than all the routines provided by libdc1394.

FFTW3 - Fastest Fourier Transform in the West v3.0

For working in fourier space, it is desired that the discrete fourier transform and its inverse as fast as possible. Even though it is quite easy to implement

a $n \log n$ FFT algorithm, there are still many ways to make the operations as fast as possible. Several papers on optimizing the FFT has been presented over the years, FFTW3[1] is the fastest and was therefore selected. What makes it especially fast is its ability to do a performance characeristic of the system the routine is running on, to try to maximize the performance as much as possible.

POSIX threads (pthreads)

Since the processor had four cores, and the cameras should be as synchronized as possible, a way to create threads and deal with them in a sensible way was needed. For this system, *pthreads* was chosen as the thread handling library. In retrospect, the *Boost* threading library might've provided better features for threading object-oriented code, albeit with a neglible higher memory footprint.

4.4.4 Applications of note

- Gnuplot is used for all graphs.
- LaTeX for text formatting.
- GIMP is used to quickly analyse image properties such as histograms, pixel values and pixel positions.
- Firecontrol - resetting Firewire bus and checking non-IIDC registers.

Firecontrol

Firecontrol is a very simple program that can be used to reset the Firewire bus and write and read to/from all Firewire memory adresses.

To enable auto-synchronization, the following had to be inserted:

```
w 1023 0 0xFFFFF2F10018 4 0000003
w 1023 1 0xFFFFF2F10018 4 0000003
e
```

and the values could be checked by:

```
r 1023 0 0xFFFFF2F10018 4
r 1023 1 0xFFFFF2F10018 4
e
```

Where the first column is read/write, second is the Firewire card, third is the device ID, and the fourth is the auto-synchronization register address followed by how many quads you wish to read/write, followed by what to write.

4.5 Software implementation

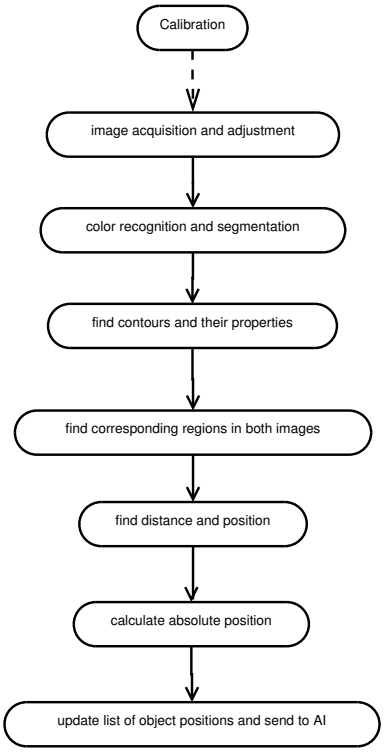


Figure 4.3: System overview

The main program uses four separate threads for different purposes. One is used to handle incoming messages from the artificial intelligence (start message, object pickup messages, request for initial cylinder configuration, stop message) as well as navigation (own position, enemy position). One thread is dedicated to tasks that require information from both images as well as displaying results. The two remaining threads are for stereo independent tasks, such as remapping, converting colors, morphological operations, color segmentation and recognition, finding contours and contour properties.

The main thread and the camera threads are synchronized only by using barriers, while the main thread and the communication threads share information via datatypes that have mutexes to ensure that only one thread reads or writes a value at the same time.

Only one thread deals with fetching data from the camera buffers directly, and then the data from one of the cameras is copied for use by the slave camera thread. To ensure faster synchronization, the cameras are polled in

reverse order every other time.

Before the initial message from AI to get the cylinder configuration, the system runs in a special synchronization mode, where it spends nearly all the execution time waiting for data from the cameras rather than processing. Then, once the initial cylinder configuration has been detected, the system switches to the default execution mode, which can be selected as described in section A.1.2.

The system is designed so that the actual processing of the images can be easily replaced. All algorithms have to have two parts, one which entails everything that needs to be done on each frame individually, and one that does everything that requires both images.

The software system created in this project is able to capture images from two cameras in a synchronized fashion, and calibrate them for stereo vision, so that the epipolar lines line up. It can also convert the raw image data into various color formats, and do color thresholding and edge detection on the converted images. It can also segment images into blobs, and the distance to these "blobs" can be determined, as well as their shape. Figure 4.3 gives an overview of the system, while figure 4.7 is a more detailed description of the implementation.

4.5.1 Calibration

Calibration is done by using Boquet's algorithm available through the OpenCV library. This is done by first capturing a lot of images of a chessboard pattern, and then running the calibration routines required to calibrate the images properly. The result is then stored, so that when the setup has been calibrated once, no further calibration is required.

4.5.2 Image acquisition and adjustment

Image acquisition is done by using the libdc1394 library to capture frames, and then converting the raw image data to the desired color format. The image is then adjusted by using the matrices created by the calibration program.

4.5.3 Thresholding

The only objects of interest in the scene are either red, orange or has a high intensity. This can be exploited by thresholding away all pixels that do not have a certain red/green ratio, as well as checking if they have a certain intensity.

All pixels that do not have the desired red-green ratio or the desired intensity are set to black (0,0,0). Two separate images are created, image α for pixels of high intensity, and image β with the desired red-green ratio.

4.5.4 Morphology

The morphology operator is only executed on the β image, and then the α image is merged into it. It is used to remove specular reflections from the balls, so that balls are segmented properly. Without it, the balls will still be segmented into regions, but they will have large holes, and the outer contour will often have large cavities, which can make matching the contours more difficult. The operator used is a "opening" with a circle as structuring element. See section 2.10 for more information. Since OpenCV's implementation for erode and dilate is used, the size of the structring element can be adjusted. If the structuring element is too large, balls that are far away will lose their proper shape, while too small structuring elements will not be able to mend the balls that are close (and therefore have larger holes). The size of the structuring element used in the implementation is $5x5$, and this was arrived at through systematic experimentation.

The size will cause the balls that are right next to the camera often exhibits very small holes, but these can easily be discarded when analysing the contours later.

4.5.5 Color segmentation

The color segmentation, which is probably the step that took the longest to arrive at, is surprisingly simple. It started off as an attempt for improving the gradients for edge detection. The balls do not have any hard edges, so using intensity works very poorly as the intensity gradually changes along the ball, instead of abruptly changing versus the background. This means that a lot of research has gone into trying to find a single metric, that could be represented as a single integer for each pixel, that would give much more pronounced edges around the edges.

An approach that compares each R,G,B pixel value individually, and combines them using sum of absolute differences was introduced.

4.5.6 Combination of color difference and Sobel operator

An approach that works surprisingly well is to combine the basics of the Sobel operator, with the metric for color difference 2.24.

The sobel operator is replaced by:

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

where the color difference δ is calculated as:

$$horizontal : \delta(a,c) + 2 * \delta(d,f) + \delta(g,i) \tag{4.1}$$

$$vertical : \delta(a,g) + 2 * \delta(b,h) + \delta(c,i) \tag{4.2}$$

This essentially means that it is a Sum of absolute differences of each individual color value in a certain pattern. The current implementation is approximately 2-4 times slower than the Sobel operator.

4.5.7 Color segmentation using scan line

The approach used here calculates the RGB gradient, and then goes through the image row by row in a scanline fashion, averaging the pixel values between each area of high gradients. It then proceeds to do the same along each vertical line in the image, but this time averaging the averaged color values in each horizontal line. The result is regions of pretty much uniform color, as long as the thresholds are set low enough.

The main issue with the scanline and color difference implementation is that if you set the threshold too high, regions will not be segmented properly, and you risk getting streaking artifacts. These artifacts are much more pronounced than using a slower variance based approach. If the threshold is set too low, a region might not be properly segmented, and small holes and irregularities in the color will appear.

It works best with very vibrant colors, and normalization could be an advantage for better segmentation, but due to illumination causing shadows,

setting a proper threshold is hard. The result is a much faster segmentation algorithm that works well when you have objects and backgrounds with very vibrant and distinct colors.

Cleaning up routines

Running the segmentation routine several times will make segmented regions even more uniform, and if combined with a cleanup routine of some sort (like a single black pixel surrounded by the same color on all sides), the segmentation can be significantly improved.

4.5.8 Finding contours & properties

Contours are found by tracing around all pixel regions of non-zero value. This is done by OpenCV's *cvFindContour()* function, as it provides ways to exclude regions that are within regions, and it is easy to calculate various contour properties on it. Moore border tracing was implemented, but due to time constraints it wasn't used in the finished system.

4.5.9 Matching contours

The contours are matched using the following properties:

- average color of region
- bounding box
- minimum and maximum disparity (distance should never be below 30 cm, and never above 3 meters).
- minimum rotated rectangle
- center of region

The average color of a region is calculated by taking the neighbouring pixel of the contour all around the region and averaging it. The region itself should be pretty much a single color, but averaging avoids irregularities. The bounding box is merely calculated as described in section 2.12, and the minimum rotated rectangle is done by OpenCV, which uses rotating calipers.

4.5.10 Calculating disparity

Once matching contours are found, the disparity between a match in the left image and the right image is calculated. This is done by checking what the disparities are for each corresponding epipolar line on the leftmost and rightmost side of the contours. This is put into an array, and only the disparities with the highest matches is used. This avoids potential irregularities in the region segmentation, which would have affected the result if an averaging method had been used.

For a ball that is behind a cylinder (the most common semi-occlusion issue in this project), the biggest disparity should always be the correct one. Therefore the largest of the left-disparity and the right-disparity is used as a basis. Considerable time was spent testing it, and it makes sense intuitively. This is because when a ball is behind a cylinder, the edge between the ball and the cylinder will be a point that is further to the right in the left image than in the right image. The disparity will then be lower than the real disparity. The other side (that is not occluded by the cylinder), will have a nice properly segmented edge which gives the correct result. An illustration can be seen in figure 4.4.

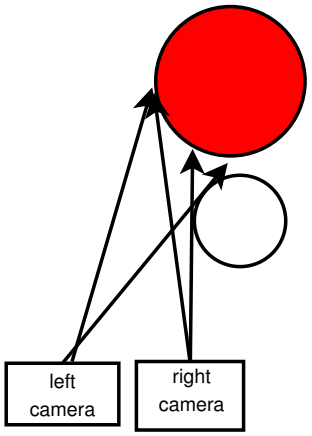


Figure 4.4: Illustration of how semi-occlusion of a ball behind a cylinder affects disparity

4.5.11 Calculating relative position and distance

The distance is a linear relation to the disparity. Given by $distance = \frac{k}{disparity}$. This is verified by experimentation in section 5.3. The constant

k has to be adjusted after recalibration due to small adjustments in view area etc. For most of the project a k value of $4700 - 5200$ was used. The accuracy is also further discussed in section 5.3.

When the distance is calculated, simple triogonometry is used to figure out how many pixels offset from the center of the view area the object is.

4.6 Updating position of game elements

After finding the initial cylinder configuration, the placement of all the game elements is known. Using the absolute position of the robot, and its orientation, it is then possible to figure out which of the game elements should be visible when a frame is captured. This information can then be used to update only the elements that are visible. It can also be combined with an error metric, so that objects that are far away but still within the margin of error are not updated, since the initial position is most likely more accurate than the measured one. An example of this is if a ball is 210 cm away, the margin of error is approximately 20-30 cm, and a measurement of 220 cm from the computer vision would then be discarded. Generally it is a good idea not to swamp the AI with too many unnecessary updates.

The steps to update elements is as follows:

- use last known position of all elements
- figure out which elements are within the field of view.
- compare found objects with objects that should be within the field of view.
- match each found object with the nearest object that should be visible.
- if found objects exceeds objects visible, use an ID of an object that is not visible anymore, or update the nearest non-visible object.
- send update to AI.

4.7 Using position and orientation

The following properties are known:

- The camera height vs. the ground

- The camera angle vs. the ground
- The camera's position relative to the rotational axis of the robot.
- The placement of the borders on the table

The robot also has systems to figure out its own position and orientation, as well as the opponent's position. All this information can be exploited to figure out which regions of the image are undesireable.

The vision system gets the robot's own position and orientation 100 times per second, and the opponents position 10 times per second. All the information is time stamped and put into a FIFO buffer that makes it possible to figure out where the opponent and our own robot was at a specified time. The data is also interpolated in a linear fashion to improve the resolution, as most of the robot's movement is constant.

4.8 Mapping out the opponent

The enemy robot can be made out of wood, steel, plexi or a number of other materials, and also have very different shapes. It is therefore easier to try to map out the robot from the images, rather than trying to separate it from the playing elements. It is done by combining the absolute position of the opponent and the robot to figure out the relative position, and then mapping it into the image. The robot's maximum circumference is 1200 mm, and all the robots are very compact and mostly circular, so a virtual box of 40 cm width was created around the enemy.

4.9 Mapping out everything not on the playing field

Making sure that the objects seen are part of the playing field is done by drawing a line a few centimeters below the borders of the playing field, and checking each contour to see if the minimum of the bounding box is below or above the line. If it is above - the object is not on the playing field, if it is below - it is.

4.10 Finding the start configuration

The robot has a fixed starting position and orientation on each side. This means that no recognition is required to figure out if a cylinder at a certain position is white or black. All that needs to be done is to check specific regions of the image and calculating the average intensity.

Since the robot is placed so that at least half the table is visible, and the cylinder configuration is mirrored along the center of the table, it is possible to figure out the placement of the black cylinders from the starting position.

Once the average of each region is checked, the four darkest regions are marked as black, while the rest is marked as white. This means that even if one cylinder is not on the table, you will still get the correct configuration.

This approach is similar to what was done by[17], and in the Eurobot competition, the starting position was found without error every single time.

Even if the routine is simple, it is the most important contribution of the computer vision system, as without it, the robot has to avoid *all* cylinders, and not just the black ones. This causes the driving route to become extremely complicated.

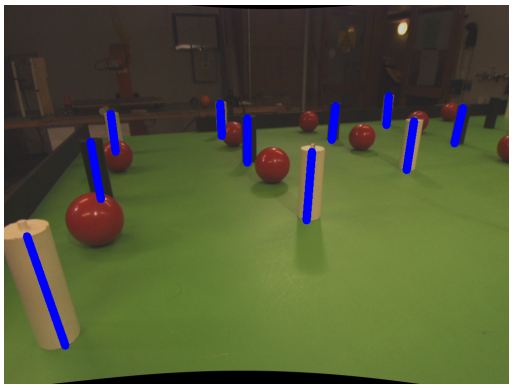


Figure 4.5: Checked image regions drawn in blue

4.11 Other features

The main program can also run various other algorithms instead of finding objects. It can create anaglyphic stereo images, run a thresholding test routine to adjust the color recognition parameters, as well as running edge detection with colors for direction, and outputting adjacent colors around

edges. It is possible to run a standard Hough transform with a fixed radius for testing purposes. There are modes to find circles in the image using the hough transform, and finding corresponding circles. The Fast Finding and Fitting algorithm is implemented, and color segmentation without thresholding is also available.

A separate program to calculate the Fourier transform of images, and doing inverse filtering was created separately for testing the reduction of horizontal blur. It merely takes in two images on the command-line, calculates the transform of the incoming image, and then runs inverse filtering with a certain blur factor. This was done separately because restoring images is much easier done in a controlled environment, rather than with live images.

4.12 Communication

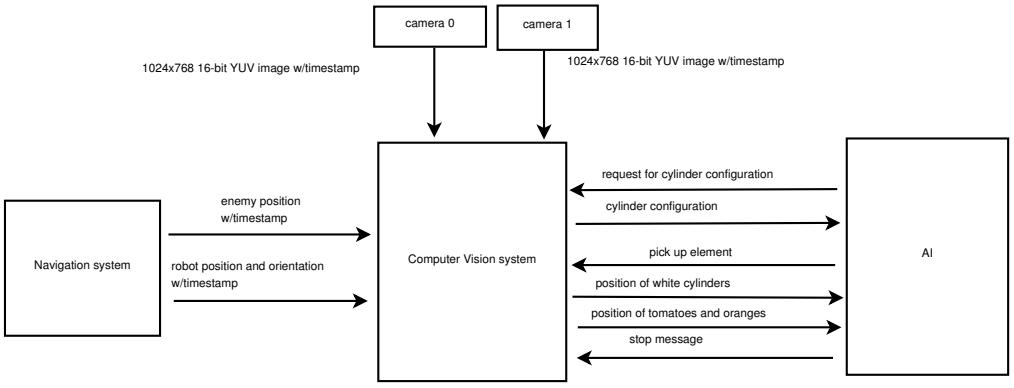


Figure 4.6: Overview of the system's external communication

The computer vision system communicates with the navigation and artificial intelligence by using POSIX messages. The navigation system receives odometry readings with timestamps from the sensor system, as well as relative positions to the enemy robot from a laser beacon system. This is converted into absolute positions, and sent to the computer vision system. [4]

The enemy position is sent approximately 10 times per second, while the robot's own position is sent 50-60 times per second. This information is then used by the computer vision system to figure out where the cameras were looking at the time of capture.

[4]The sensor system as well as the gateway for communication was written by Ole Lillevik, while the navigation system was written by Kristin Haaland.

The artificial intelligence only sends requests for the initial cylinder configuration, the stop message, as well as a message every time it thinks an object has been picked up.

The information the artificial intelligence expects from this system is only related to the game elements and their positions.

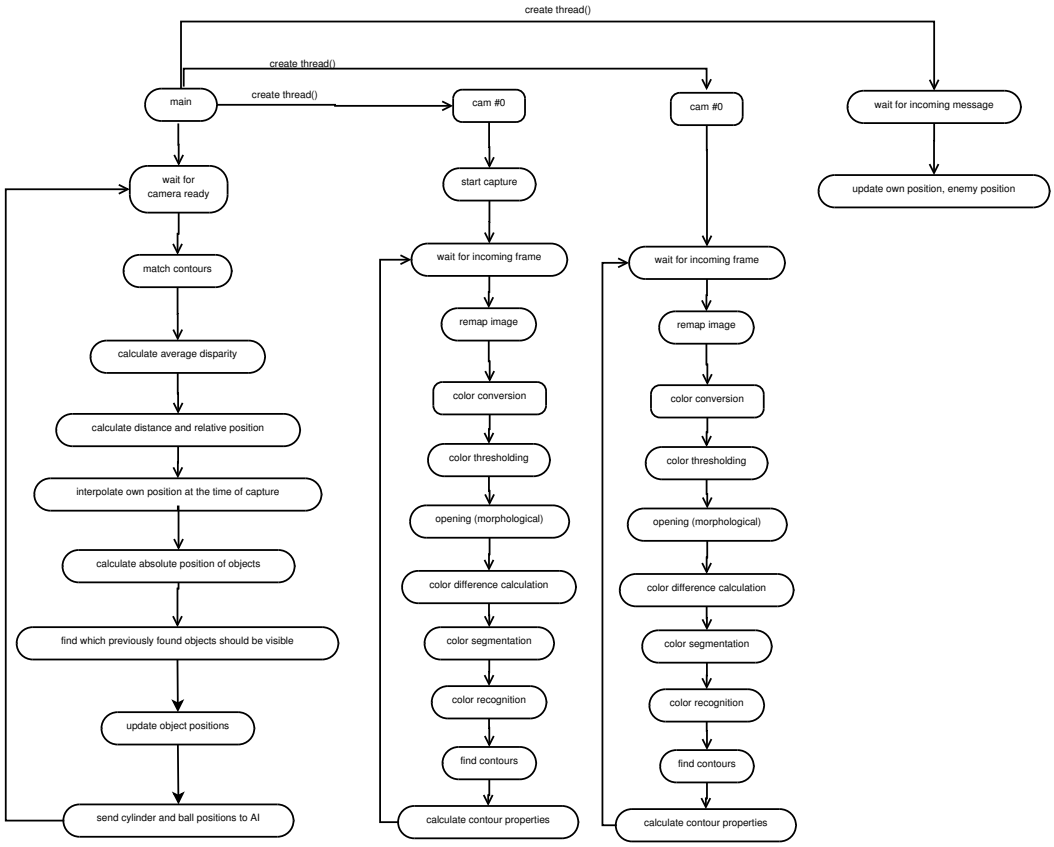


Figure 4.7: Complete overview of the system during processing

Chapter 5

Experiments

This chapter describes all the various steps of the solution that were tested, as well as some solutions that didn't work out too well, and also some preliminary results from the algorithm that locates the positions of spherical objects.

5.1 Calibration

Substantial time was spent trying to calibrate the cameras properly. Finding a good way to calibrate the cameras could be a project on its own. It was therefore decided to try to find a decent pre-made implementation of camera calibration. OpenCV has a reasonable implementation of Boquet's algorithm[13, p433]. To do calibration a stable camera mount is required, and preferably the cameras should be as aligned as possible to maximize the usable view area. The details of the camera-mount are explained in section 4.3.4. A pattern for calibration was created by printing a chessboard pattern on an A3 size paper, and gluing it to a cardboard-like surface. It is very important that the pattern is as flat as possible, or else the calibration could be more inaccurate than what is acceptable. Figure 2.4.4 shows an example from rectification input.

Testing shows that for best results one should try to take pictures of the chessboard when it occupies pretty much every part of the image. Images that are blurry because of movement during exposure needs to be excluded, or else the calibration becomes inaccurate.

The results of the calibration are stored as matrices describing how much each pixel should be shifted vertically and horizontally. These can then be

used to remap pixels on the fly by the stereo vision system.

5.2 Anaglyphic Stereo

To show of the results of simple rectification, real-time anaglyphic images were produced. An example can be seen in figure 5.1. Anaglyph images are simply two images superimposed on each other, with a certain offset to provide a 3D effect. This was achieved by simply converting the incoming camera frames to RGB, and taking the red channel from the left image, and the blue and green channel from the right image, superimposed on each other. Ordinary red and cyan 3D glasses must be worn to see the 3D effect. Because using glasses makes the images slightly darker, all the color channels were multiplied by 1.1 to increase the brightness slightly.

By using 3D glasses, you can get a rather good 3D effect, albeit with somewhat reduced colors. This is by far the cheapest way to achieve 3D images, as the only requirement is some semi-transparent plastic in the right colors.

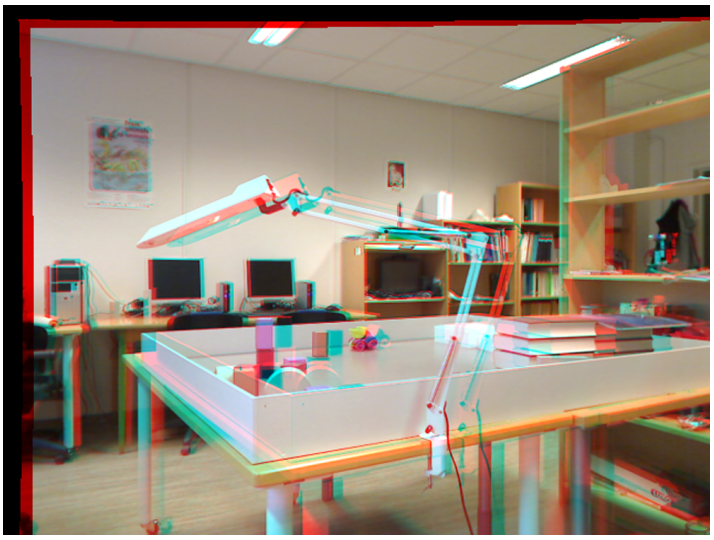


Figure 5.1: The lab in anaglyphic stereo

5.3 Triangulation accuracy

To test the accuracy of the stereo setup, the images from the cameras are thresholded with a minimum value for chromatic red, and testing is done

with a red cube. Then, using Canny edge detection with parameters (30.0, 100.0) edge images are obtained. The images are then run through a simple scan-line algorithm along the epipolar lines that checks the difference between edge n in the left image, and edge n in the right image. The result is simply the largest disparity found. Figure 5.2 shows the test in action.

Using this, the distance Z, given the right k, is:

$$Z = \frac{k}{disparity} \tag{5.1}$$

The error, due to aliasing, and the fact that one point might be two pixels in one image and one pixel in the other, is assumed to be:

$$E = |\frac{k}{disparity} - \frac{k}{disparity - 1}| \tag{5.2}$$

and

$$E = |\frac{k}{disparity} - \frac{k}{disparity + 1}| \tag{5.3}$$

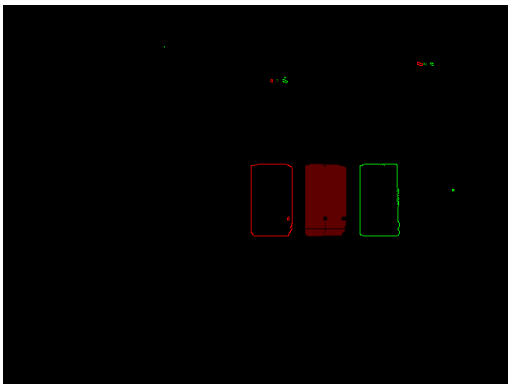


Figure 5.2: Initial testing distance to cube, disparity=138, measured distance: 41.7 cm, real distance: 47 cm

5.3.1 Distance

As figure 5.3 shows, the disparity is given by $disparity = distance/k$, where k is dependent on the focal length and the distance between cameras. Given this, the distance, Z, can then be calculated by equation 5.1. This is also confirmed by (2.10). For later use, with 800x600 resolution and a distance between cameras of 11.5 cm, the constant had to be increased to 7500. This

essentially doubles the practical depth resolution, but as can be seen from the graph, the depth resolution decreases very rapidly, and even without any aliasing issues, an increase of 1 in disparity is over 20 cm at 4 meters distance, and over 40 cm with the lower resolution and shorter distance between cameras.

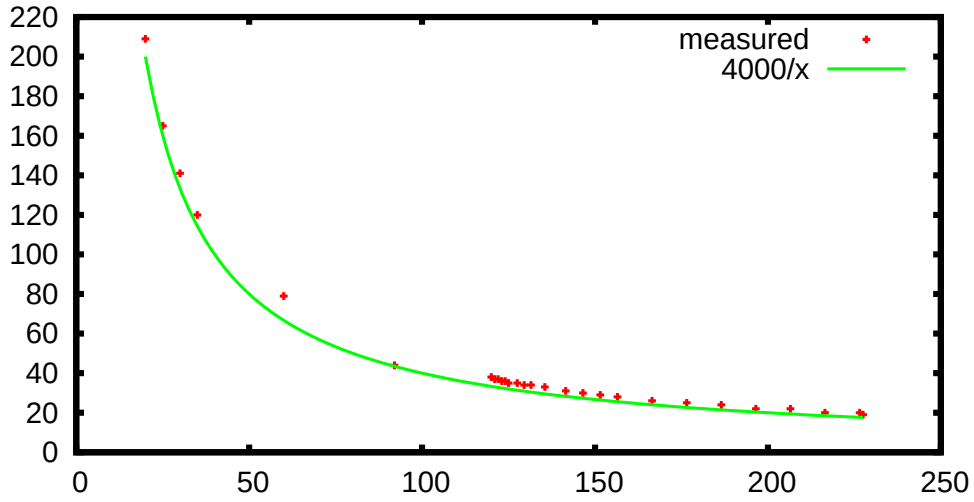


Figure 5.3: Disparities at various distances, distance between cameras d = 8 cm, resolution 640x480, see appendix B.1 for raw data

5.3.2 Position

When we have the distance, the only thing we need to calculate is how many pixels a plane at a certain distance would occupy, and from that calculate the positions in the horizontal and vertical direction. The view area is calculated by equations 2.11 and 2.12. The position in the horizontal and vertical direction can then be measured by finding the middle and diving the view area by the amount of pixels in each direction.

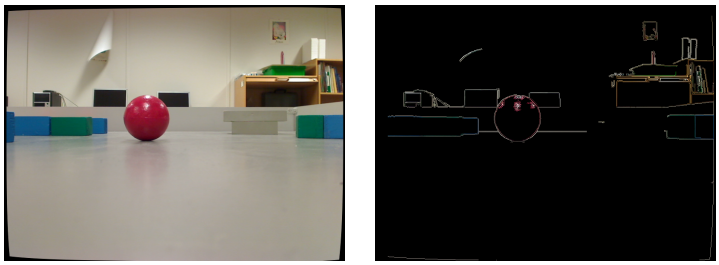
$$pos_x = \frac{coordinate_x - \frac{width}{2}}{view_x} \tag{5.4}$$

$$pos_y = \frac{coordinate_y - \frac{height}{2}}{view_y} \tag{5.5}$$

5.4 Edge detection with direction and adjacent pixels

As edge detection does not provide enough information on its own to be very good for a more general correspondence analysis, some work was put into getting edge direction, and also the pixel colors of adjacent pixels.

Finding the edge direction can be done easily by doing a Sobel mask in the horizontal and vertical direction for each matched edge, by for example the Canny edge detector. Finding the adjacent colors is done by finding the edge direction, and checking the pixels on each side of the edge (along the normal). The color on the edge itself seemed rather irrelevant. Figure 5.4 shows the two algorithms in action.



(a) Original image (b) Edge-adjacent colors

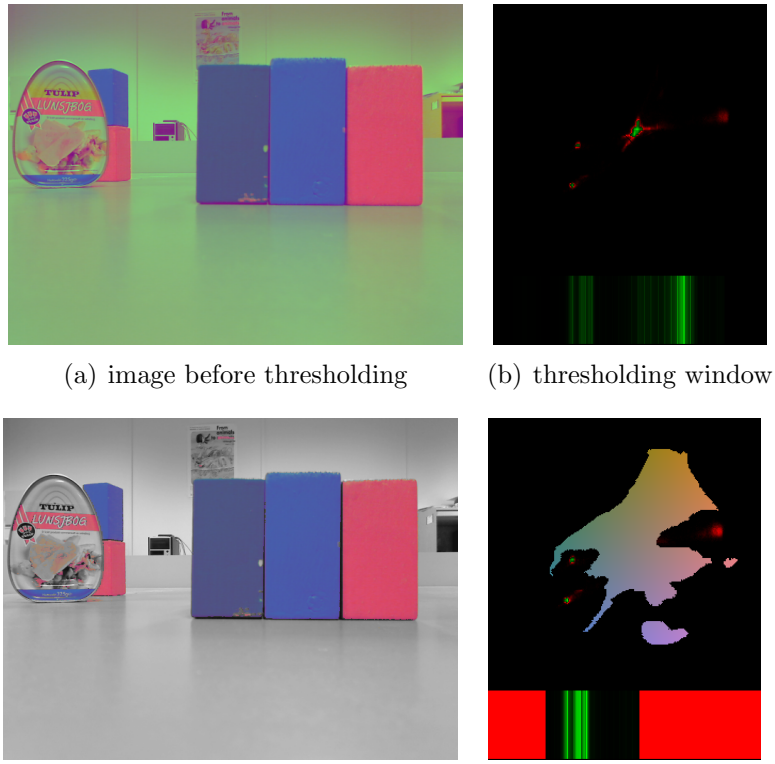
(c) Edge direction

Figure 5.4: The original image, the colors adjacent to the edges, and the direction of the edges.

5.5 Color recognition

For color recognition, the easiest approach was to take advantage of the YUV space, mapping chromatic blue and chromatic red into a 2D matrix. I also

added the brightness at the bottom of the thresholding window. This is very similar to working on a normalized RGB image when mapping them into rg-space. However, using YUV gave more accurate results. An example is shown in figure 5.5.



(a) image before thresholding (b) thresholding window

(c) image after thresholding (d) thresholding window

Figure 5.5: Color recognition in action, showing the image before and after thresholding, the image is shown with the YUV values mapped directly to RGB, G=Y, R=U, B=V, not properly converted.

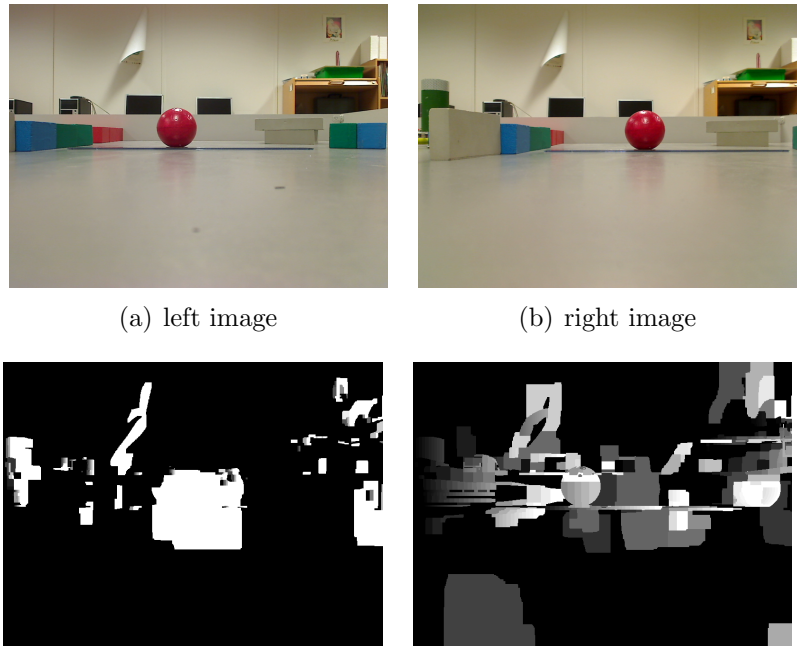
5.6 Block matching

For block matching it was attempted to match blocks of various sizes with several different similarity measures, such as SAD, but there seems to be too many false positives. There is an amazing amount of similar regions in a typical image, especially when the environment is as free of texture as the playing field previously described. The use of edge data, as well as edge direction has been attempted, and requires further attention.

Compared to using more advanced algorithms, block matching seems far better suited for obstacle avoidance than object recognition, as you seem to end up primarily with unrecognizable blobs of depth data, rather than easily recognizable shapes.

5.7 Other work

Some current implementations were tried from Daniel Scharstein's website about Stereo Vision[11], which has a lot of comparisions of different correspondence algorithms. This was mostly to see what one could expect from simply trying to create depth images. All the examples took from five minutes to one hour to execute. The algorithms are designed to be accurate, not fast. They use a Markov Random Field, which is essentially a graphical model with a set of random variables have a Markov property described by an undirected graph. Output of the standard MRF algorithm can be seen in figure 5.6.



(a) left image

(b) right image

(c) output of MRF algorithm, depth resolution: 16, runtime: 63.7 seconds

(d) output of MRF algorithm, depth resolution: 256, runtime: 765.7 seconds

Figure 5.6: Testing of one algorithm using Markov Random Fields

5.8 Horizontal motion blur

During testing, it was noticed the USB cameras required quite a long exposure time to get proper images, and using inverse filtering to counter-act the effects of motion blur was explored.

Using the camera setup with the new firewire cameras this is not a problem, but the results from the testing might still be of interest. The results are shown in figure 5.7 . As can be noted, the inverse filtered image is much sharper than the blurred image, but there are a lot of artifacts. Inverse filtering without a threshold proved futile, as the noise exceeded the signal in the high frequency areas in the Fourier transform. A threshold was devised, to avoid using the filter where the noise exceeded the signal.

(a) original image

(b) with horizontal motion blur of L=5

(c) inverse filtering

Figure 5.7: This shows inverse filtering in action. The improvement is very visible on screen, but due to the printer having a tendency to introduce artifacts of its own, and blurring, it's very hard to see in print.

5.9 Corresponding circles

Since part of the problem to be solved is to track balls, a fast way to both detect and match circles in the corresponding images seemed appropriate. This was done by performing color thresholding on the images from both cameras, and then doing a gaussian blur, followed by a Canny edge detector to extract edges, gathering evidence as to where potential circle centres might be. Finally the Hough transform was performed for the circles that has more matches than a certain threshold.

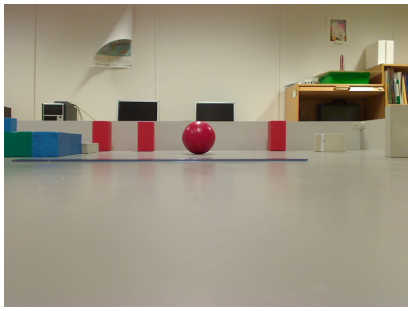
The evidence is gathered by exploiting the fact that the normal to the tangent of a circle, will go through the center of the circle. Finding the normal (or a rough approximation of the normal) can be done by calculating the direction of an edge. By drawing a line from the minimum radius to the maximum radius, and increasing each point, we will get a definite spike at the centre of the circle.

Once we have created an accumulator image in Hough space, the data is cleaned up by making sure that each circle center checked for has the largest accumulated function in its 8 connected neighborhood.

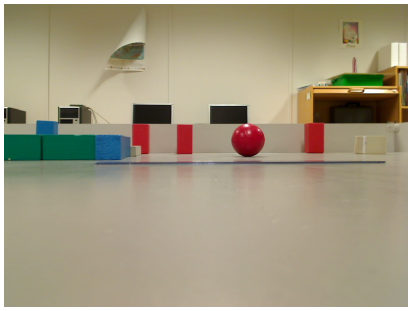
The color thresholding improves the robustness of the algorithm quite a lot as the edges stay sharp instead of being reduced to a blurry line. The thresholding is set so that there are a few false positives in the real world, but on the field used in the competition, the only objects that are even remotely red, are the balls on the ground that we are tracking. Even with a lot of unnecessary pixels from over-thresholding, the algorithm performs very well.

Substantial testing was performed with both a moving camera mount, and a moving ball, and the results look very promising.

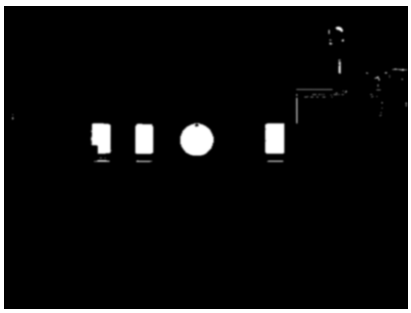
See figure 5.9 for an example.



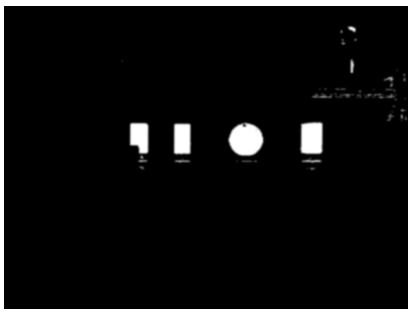
(a) left image



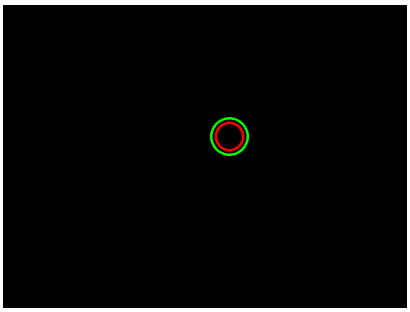
(b) right image



(c) left image after thresholding



(d) right image after thresholding



(e) matched circle in both images

Figure 5.8: Corresponding circles algorithm, showing the input images, the color recognition, and the matched circle.

5.9.1 Initial Accuracy testing

			Position					
Data			Real			Triangulated		
disparity	X_l	X_r	X	Y	Z	X	Y	Z
96	350	446	0 cm	-1.5 cm	80 cm	2.239700	-5.262307	78.125000
96	462	558	14 cm	-1.5 cm	80 cm	16.175612	-5.262307	78.125000
96	222	318	-15 cm	-1.5 cm	80 cm	-13.687056	-5.262307	78.125000
128	328	456	0 cm	-1.5 cm	60 cm	1.119850	-4.172258	58.593750
128	162	290	-15 cm	-1.5 cm	60 cm	-14.371409	-4.285022	58.593750
126	546	672	20 cm	-1.5 cm	60 cm	21.709686	-4.353038	59.523810
188	304	494	0 cm	-1.5 cm	40 cm	1.194507	-3.722390	39.473684
214	286	500	0 cm	-1.5 cm	35.5 cm	0.725635	-3.574715	35.046729

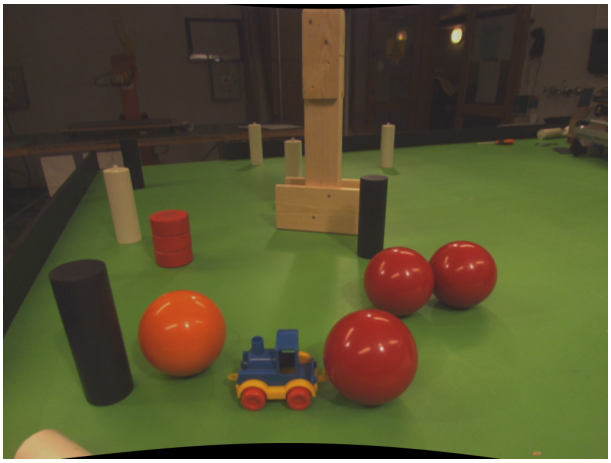
Table 5.1: Triangulation testing with corresponding circles algorithm, 800 x 600, distance between cameras=12.5 cm, X is the horizontal position, from left to right, Y is the height, going from top to bottom, and Z is the depth

From the initial results, listed in table 5.1, it seems that the center between the epipoles is slightly off from what is expected, slightly higher up and to the left. The triangulated distance is consistently below the real, measured values, which means that the constant for disparity to distance conversion needs minor adjustments. This is verified by the fact that the offset is larger at greater distances, as each pixel further away is represents a longer distance than closer to the cameras.

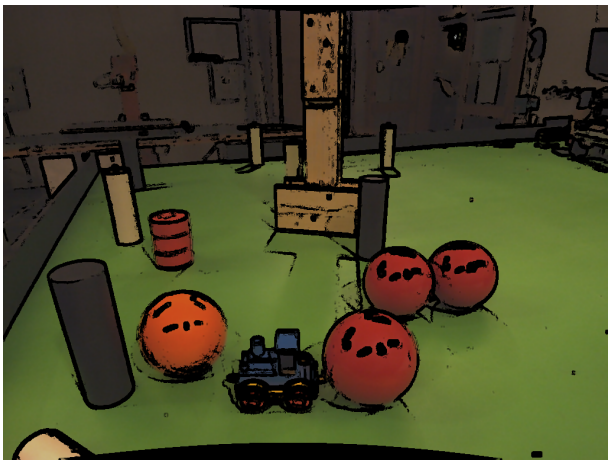
5.10 Color segmentation

As can be seen, the scanline approach works rather well, even though it does suffer from some pepper noise. This is due to the fact that the scanline approach is not very robust when it comes to noise in the input image. The playing field also had some dust and small parts of wood lying on the table, also affecting the segmentation. The other approach, described in 2.13 is however much better at dealing with noise, but the approach is also significantly slower.

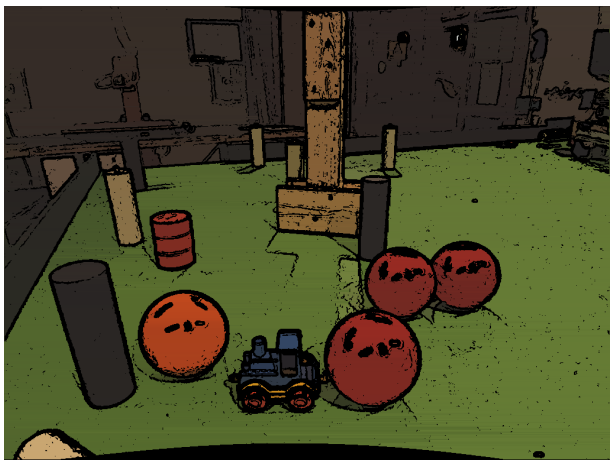
Normalization was attempted to in order to improve the segmentation, but even through systematic tuning, it was never possible to normalize an entire ball without turning the black cylinders white. The result of normalizing the rectified input image can be seen in figure 5.10.



(a) rectified input image



(b) Color segmentation using the approach from section 2.13



(c) Color segmentation using this report's scanline approach

Figure 5.9: Color segmentation using the approach described in section 2.13 took about 1200 ms to execute, while the scanline approach took 55 ms (20 ms calculating gradients, 20 ms for one scanline iteration, 15 ms for smoothing)

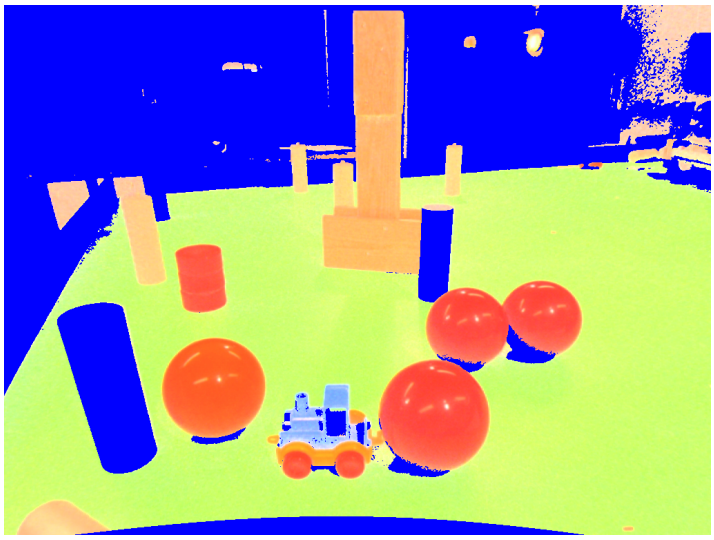


Figure 5.10: Normalization of the rectified input image. The thresholded pixels are painted blue.

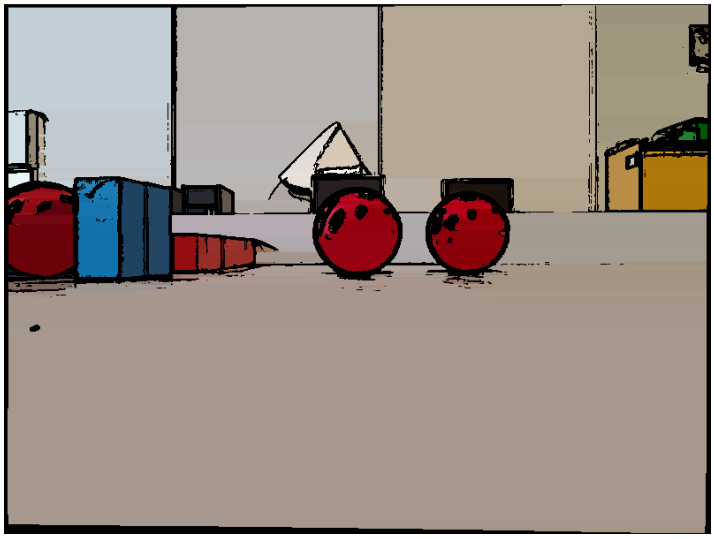


Figure 5.11: Color Segmentation at IDI's Computer Vision Lab

Chapter 6

Solution

In this chapter the results of the final solution is presented. It starts off with showing how distances to red balls, orange balls and white cylinders are calculated, and also shows some cases where the segmentation does not work as well as expected.

All the tests are performed with a threshold of 70.0 for the scanline color segmentation.

6.1 Distance precision

Getting the disparities and the relative pixel positions to match up is possible in theory, but it requires knowledge such as the exact focal length, size of the sensor and the distance between cameras. None of these parameters are that hard to come by but problems arise when rectification is applied. Atleast in the implementation used here the rectified image is scaled by a certain factor, and this causes calculations such as focal length and size to become rather useless. It is also difficult to measure distances accurately. The distances and relative position is therefore achieved by systematic tuning of the disparity/distance factor, and also adjusting the view angle.

The fluctuations in the readings are apparent in table 6.1.

The coefficient k in equation 5.1 was 4750.0 in this test. Better figures can be achieved by adjusting the coefficient further.

The calculations are also affected by the 23 degree tilting of the cameras towards the ground. By using simple trigonometric functions the solution ought to be $\sqrt{Z^2 - (35-5)^2}$, where 35 is the height of the cameras vs. the

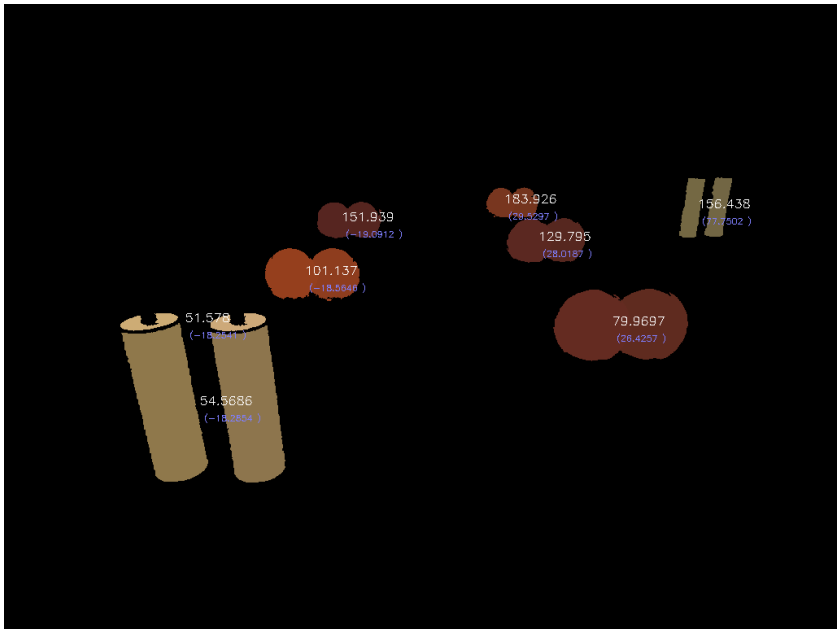


Figure 6.1: Distance to objects in centimeters, with relative position

Position						
Item	Real Position		Distance		horizontal offset	
item	Z	X	Z_{min}	Z_{max}	X_{min}	X_{max}
Red	150 cm	-20 cm	147.7 cm	151.9 cm	-18.6 cm	-19.1 cm
Orange	175 cm	30 cm	177.6 cm	183.9 cm	28.4 cm	29.5 cm
Red	125 cm	30 cm	126.75 cm	129.8 cm	27.3 cm	28.01 cm
Orange	100 cm	-20 cm	99.3 cm	101.1 cm	-18.2 cm	-18.7 cm
Red	75 cm	30 cm	78.9 cm	80.0 cm	25.9 cm	26.4 cm
White (Bottom)	50 cm	-20 cm	53.9 cm	55.9 cm	-18.0 cm	-19.1 cm
White (top)	50 cm	-20 cm	51.5 cm	53.2 cm	-18.0 cm	-19.1 cm
White	150 cm	80 cm	156.4 cm	151.9 cm	75.5 cm	77.8 cm

Table 6.1: Distance and positions readings, with fluctuations

table, and 5 cm is the height to the center of a ball. Testing data has shown that due to rectification, this is not correct.

The relative positions, which seems to measure marginally too low, is determined by the calculated distance, and the offset in pixels for each camera. The relative position is calculated for both the left and the right image, and then averaged.

The inaccuracy in the measurements appears to be greater at the outskirts of

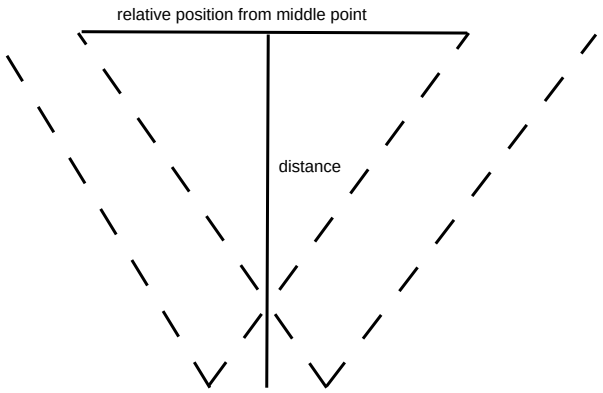
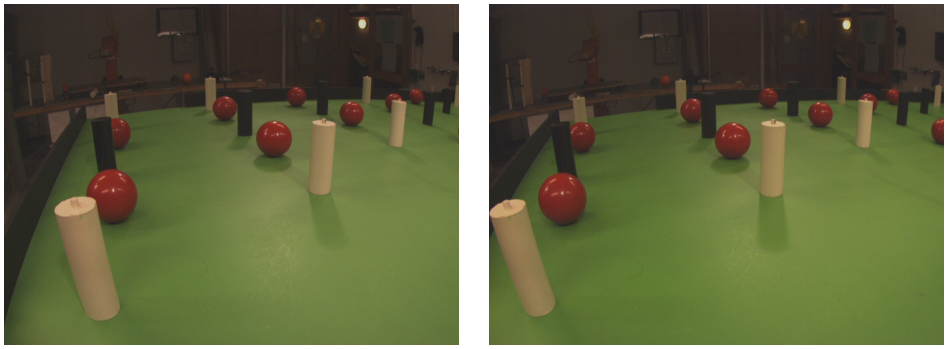


Figure 6.2: Distance and relative position from centre

the field of view than at the center. The most likely cause of this is that the chess board used for rectification is not perfect (it warps some over time), or that not enough calibration was done.

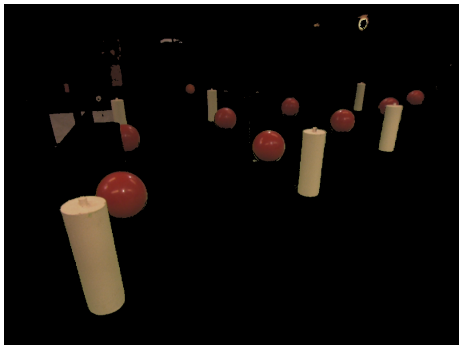
6.2 Distances to semi-occluded balls behind cylinders

In this test case, the most typical arrangement for the computer vision system is introduced. Quite a lot of the red balls are occluded behind white and black cylinders, while all the cylinders are standing upright. All balls but one is matched properly.



(a) left image

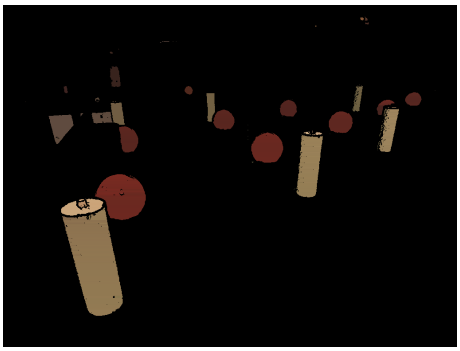
(b) right image



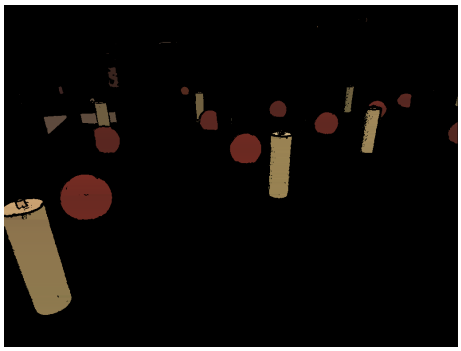
(c) left image after thresholding

(d) right image after thresholding

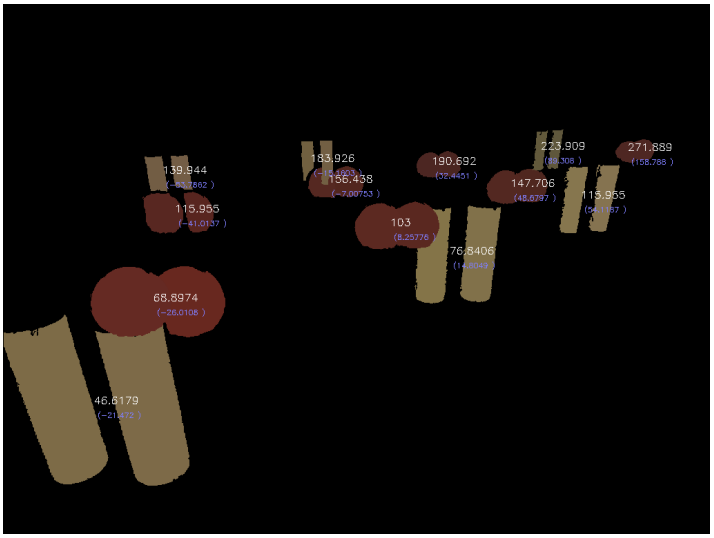
Figure 6.3: Input to corresponding objects and distance calculation



(a) left image after color segmentation



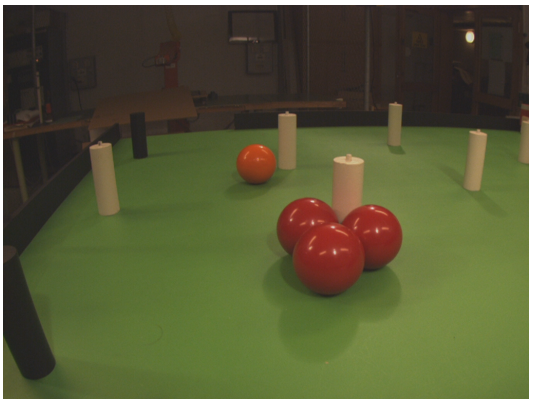
(b) right image after color segmentation



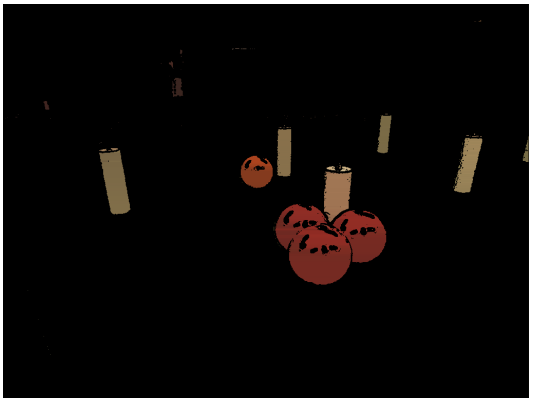
(c) Distance to objects in centimeters with relative position

Figure 6.4: Distance to objects in centimeters, with relative position

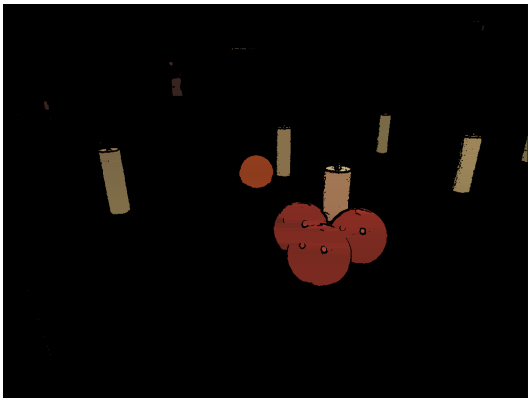
6.3 Separating objects with segmentation



(a) input



(b) without morphological operators

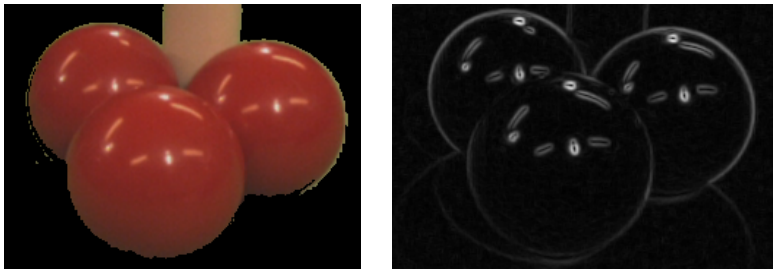


(c) with morphological operators

Figure 6.5: Segmentation issues with and without morphological erode-dilate.

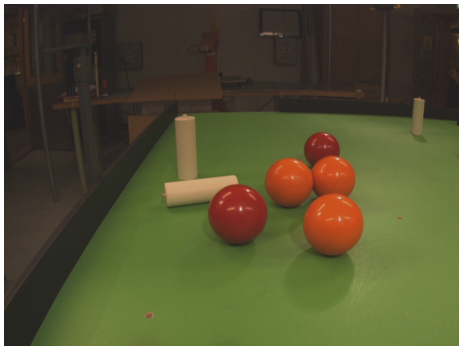
Separating objects with the color segmentation works quite well in most cases, but when the balls get very close to each other, some issues do occur.

As can be seen in figure 6.3, all the red balls are a single region. This means that the stereo vision system will only get a distance to the group of objects as a whole. Analysing the regions' contour, as well as the size of the region vs. the distance would at least register it as multiple objects. The gradients in figure 6.6 are extremely weak between the balls, and this makes it virtually impossible to separate the regions without resorting to a priori information about the regions to segment. An example where it works well is presented in figure 6.7.

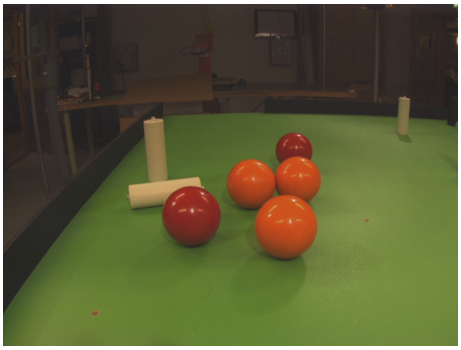


(a) Balls up close (b) RGB gradient strength

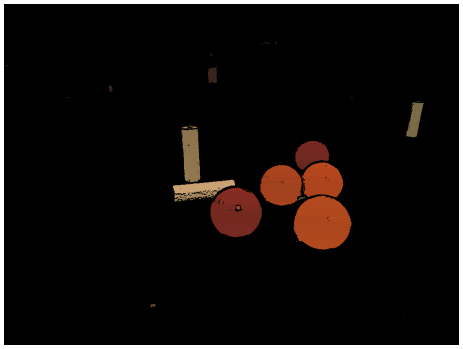
Figure 6.6: The RGB gradient in the image



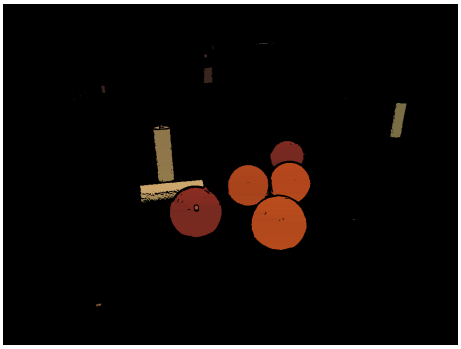
(a) left image



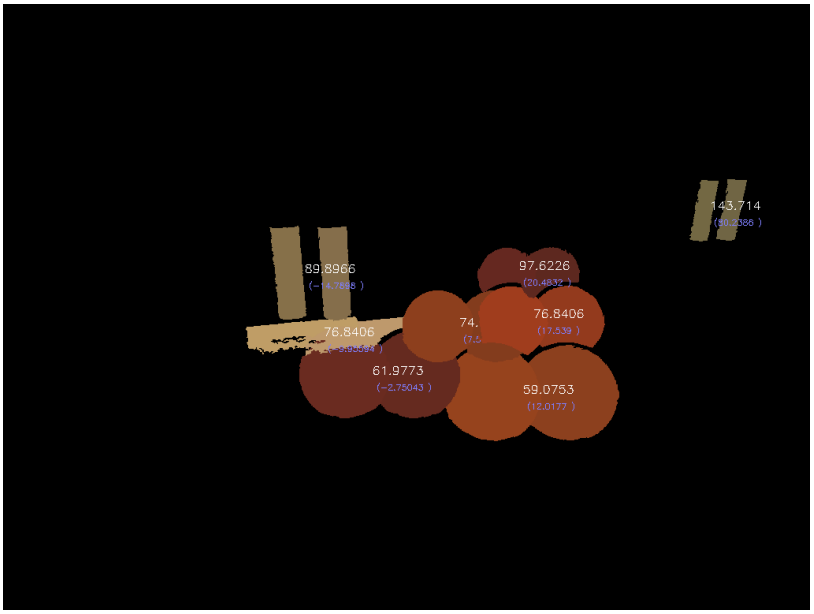
(b) right image



(c) left image after color segmentation



(d) right image after color segmentation



(e) distances and position

Figure 6.7: Color segmentation

6.4 Tracking objects while moving

Tracking objects while moving poses no noticeable issues to the computer vision system, as the shutter time is too low to introduce blurring. There are no noticeable differences between images captured while moving or standing still. See appendix D for a video of the system with moving objects.

The issue with such a low shutter time, is that if the system is used indoors under poor lighting conditions, the scenes might be too dark. Some of this could be solved with additional illumination, heavier amplification of the image signal, or by using a longer shutter time, but not so long that the blurring effects cause issues.

6.5 Using trigonometry

A system to calculate where objects would appear in the image, provided the position and orientation of the robot and the position of objects is known has been created. It was originally intended to check certain regions of the image for objects, but the points could be up to 10 cm off far away from the robot, but just a few centimeters off up close. So therefore this approach was not possible.

This system was later used to figure out if an object that was detected before should be visible in the current frame, and calculate where the opponent robot appeared in the image, to map out undesireable regions in the image. It was also used to draw corners around the table to avoid calculating distances to objects not on the table.

The system requires a nearly perfect timestamp calculation for its own position as well as the opponent. This proves rather difficult, as the Firewire bus timestamps frames upon arrival at the other end of the bus, and there are also in the timestamps for position that are recieved from the navigation system.

During the development of this project, a 2D Lidar was used to provide distances to a dummy opponent, to check if the opponent robot was mapped out of the image correctly by the stereo vision system mentioned above. Figure 6.8 shows an example of the output of the lidar.

Figure 6.8: Example of lidar output. The outer edges of the green area denotes obstacles located approximately one meter away from the lidar. The blue lines are the view angle (240 degrees).

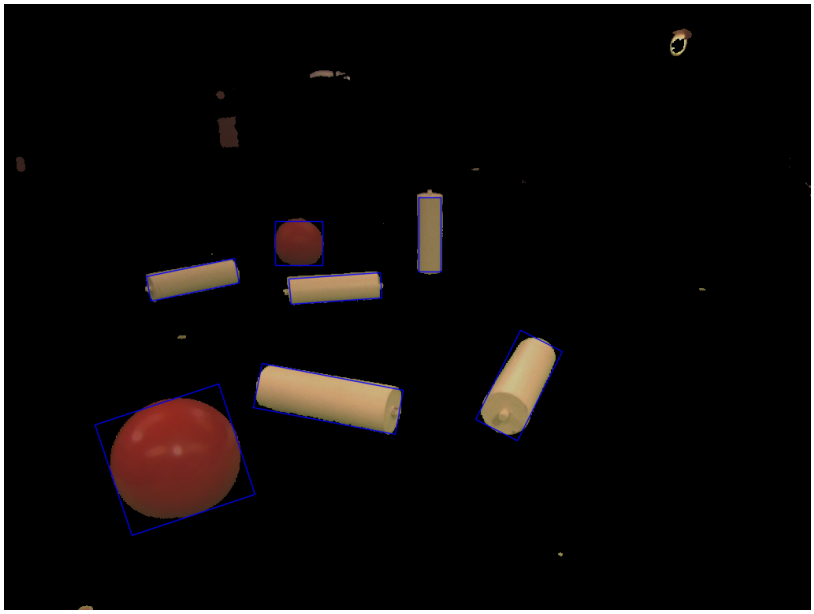
6.6 Cylinder orientation

As can be seen in figure 6.9, calculating the minimum rotated rectangle as described in section 2.12.1 can be used to determine various properties of a contour. The threshold used in this test was set higher, as the regions tend to degenerate when the cylinders are lying down. An example of this can be seen in figure 6.7.

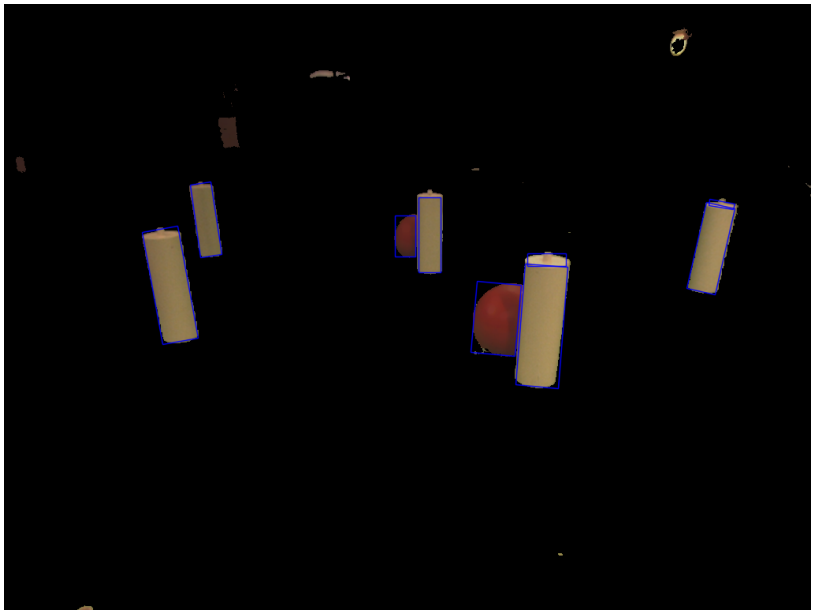
Using the minimum rectangle, the major and minor axes are easily found, as well as the center. A minor vs. major axis calculation can be used to determine if a ball is occluded behind an object, or if it consists of several objects.

The orientation is merely $\tan \frac{\delta(x)}{\delta(y)}$, and if the orientation has an angle over a certain treshold, it can be determined that the object is lying down. If an object is lying down, facing directly away from the camera, it will have the same orientation as the standing cylinders, but this can easily be discerned by checking how large the major axis is versus the minor axis.

The orientation data from both cameras can probably be used to find even more properties of the regions, but this has not been explored.



(a) Various lying cylinders and balls



(b) Standing cylinders and occluded balls

Figure 6.9: Various minimum rotated rectangle calculations

Chapter 7

Discussion

7.1 Recognition

The current solution is only designed to find distances to simple monocolored objects, but it can also find distances to more complex objects.

It should be easy to expand it to more general uses as long as the objects have strong, distinct colors.

The implementation have some issues that should be resolved. When balls get too close to each other, the segmentation tends to fail, and two balls end up as a single object. Another issue is if the segmentation maps two balls into a single region in one image, and into two separate regions in the other image. In such a case, the objects will not correspond, and no distance is calculated. This could either be solved by trying to improve the color segmentation, or by looking more into the contour properties.

The segmentation performance could have been improved by using a longer shutter time on the cameras, giving better and more distinct colors, but then the exposure time would have been too high to avoid blurring during fast movement.

Regions are not really recognized beyond the fact that they are red, white or orange. It is easy to figure out if an object is circular or elongated, but partial occlusion can cause balls to become very elongated, and cylinders can be come rather compact and squared. Only the color, center position and bounding box was therefore used to match objects. The disparities were also only checked within an expected distance of 0.4 m to 3 m.

As such, a ton of additional work could be done to find distances to a large

variety of objects. If the segmentation of an object is very accurate, you could also get the distance to an object at each epipolar line.

7.2 Performance

Processing a set of frames and finding the distance takes between 130 ms and 170 ms. This gives an effective frame rate of 5-7 Hz. This is more than fast enough to do the tasks in this project properly, and tracking of objects still look rather smooth. The AI typically spends up to three seconds to update the plan, and only receives updates from the navigation system a few times per second. With the frame rate set to 7.5 fps, a processing time of maximum 133 ms would be required to avoid dropping frames. This could have been achieved either by dropping the resolution to 640 x 480 (which is less than half the pixels of 1024 x 768), and also potentially not using the morphological operators. Dropping the resolution would have caused the objects to become very small, and it would have then been necessary to increase the zoom on the lenses to see objects at a distance properly. This would have caused the view area to drop signficantly.

Not using the morphological operators would have resulted in much worse segmented regions for the balls, but in most cases the objects would have been detected just as nicely as without. A lot of ideas for improving the actual runtime without sacrificing any recognition performance is presented later in section 8.1. An overview of the run-time for various operations is listed in table 7.1.

Operation	Time (typical)
Remapping (based on calibration)	9-11 ms
YUYV $\rightarrow$ RGB	6-9 ms
Morphological operators	40-60 ms
RGB gradient	20 ms
Color Segmentation	20 ms
Finding contours	4 ms
Calculating properties and distances	1 ms
GUI drawing	20 ms
Waiting and other	0-40 ms

Table 7.1: Performance for various operations

Of special note is the GUI drawing, which is essentially just a delay introduced to show images with OpenCV. In actual use (where no human inter-

action is possible), showing the images can be omitted. The remapping can also easily be reduced by not calculating the remapping, but just having a lookup table of where each point is shifted.

Chapter 8

Conclusion

My idea for this project started off with a rather optimistic idea of creating depth images and then locating objects based on the depth information. This plan was quickly abandoned. It takes too much time to get low-level information directly from block-matching or edges.

As time progressed, it was attempted to match circles in each image, and then matching them based on the center of the circle and the radius. This restricted the system to only work on a very limited set of problems, but on the plus side it became quite easy to verify the results as the circle size could also determine the distance to the object. Furthermore, actually getting decent edges, and avoiding false positives proved quite difficult in complex scenes.

A solution that combines the creation of depth images of all objects with the rigid approach of only matching objects of certain shapes was devised. It uses color information to segment various objects into monocolored blobs. These blobs can then be matched using numerous properties such as center position, size, orientation, color, elongatedness, and an endless supply of contour matching routines. The method of matching these blobs can be improved in numerous ways, and it shows good promise for further research.

The system forms a basis that could be useful in a number of higher level tasks, and distances to just about any object can be found as long as the region is segmented properly. For a lot of objects, several regions will be segmented, making it possible to get distances to different parts of the object. For each individual region it is also possible to calculate the distance at each epipolar line in the image, even though this is vulnerable to noise if the segmentation is less than perfect.

8.1 Future Work

8.1.1 Segmentation

Looking more into the color difference, and maybe making a second color gradient image comparing pixels at different distances, might improve the robustness, provided there is enough available processing power. Perhaps checking several lines at a time instead of one could also be a possible solution for some tasks. This would reduce the chances of not being able to segment regions properly, but the regions that were segmented would have a more crude contour. This crude contour would probably be improved by region growing.

8.1.2 Performance

Performance could be improved by parallelizing the image processing further. One approach might be to split each camera's frame into two separate buffers, and doing the first operations in four threads, then some in two threads, and also perhaps trying to do the object matching and measuring in two threads.

The color gradient calculation can most definitely be improved by looking more into optimizing sum of absolute differences calculations, and the placement of the threading barriers could be redesigned to reduce waiting time for threads.

Furthermore, since the GUI drawing requires a 20 ms delay, doing all drawing in a separate thread from the object correspondence might also reduce the computation time a small amount.

8.1.3 Recognition

Recognition wise, the system could be improved in numerous ways. When objects are not segmented properly, so that two balls end up as a single region, it should be possible to analyse the contour of the region,seeing if the distance to the entire region is consistent, or if there is a distinct difference in disparity from one part of the contour to another.

A priori information about the size of the objects could also be used, once the distance is calculated. When you know that the distance to an object is 10 cm, it is trivial to determine how many pixels it should correspond to in the image, and use that information to determine that what you have segmented

is not one single object.

Also, when two objects are segmented properly in one image, but seen as one object in the other, the metric used to match contours will fail. One solution to this is to figure out which regions do not match in an image, and trying to see if several objects in one image are in the same horizontal area as one object in the other.

For applications other than finding simple objects such as monocolored objects, and more complex objects, some shape recognition approaches could be used.

Bibliography

[1] *Fftw*, http://fftw.org/.

[2] *Gnu compiler collection*, http://gcc.gnu.org/.

[3] *unicap - the uniform api for image acquisition devices*, http://www.unicap-imaging.org/.

[4] *Eurobot 2010: Feed the world*, 2009, http://www.eurobot.org/eng/rules.php.

[5] *Wikipedia: Usb video device class*, 2009, http://en.wikipedia.org/w/index.php?title=USB_video_device_class&oldid=327723583.

[6] A Bais, *Stereo vision based self-localization of autonomous mobile robots*, Lecture notes in computer science **4931** (2008), 367.

[7] Richard E. Blake and P. Boros, *The extraction of structural features for use in computer vision*, 1995.

[8] D.C. Brown, *Decentering distortion of lenses*, **32** (1966), no. 3, 444–462.

[9] J. Canny, *A computational approach for edge detection*, IEEE Trans. Pattern Anal. Machine Intell. **8**, no. 6, 679–698.

[10] Intel Corporation, 2009, http://software.intel.com/en-us/articles/non-commercial-software-download/.

[11] Richard Szeliski Daniel Scharstein, *A taxonomy and evaluation of dense two-frame stereo correspondence algorithms*, 2002, http://vision.middlebury.edu/stereo/.

[12] R.B. Fisher, S. Perkins, A. Walker, and E. Wolfart, *Hypermedia image processing reference*, Wiley, 1996.

[13] Adrian Kaehler Gary Bradski, *Learning opencv: Computer vision with the opencv library*, O'Reilly Media, Inc., 2008.

[14] J.-S. Ginhoux, R.; Gutmann, *Model-based object tracking using stereo vision*, (2001).

[15] P. V. C. Hough, *Methods and means for recognizing complex patterns*, US Patent 3969654, 1962.

[16] Bernd Jahne, *Practical handbook on image processing for scientific and technical applications, 2nd edition*, CRC Press LLC, 2004.

[17] Kai Hugo Hustoft Endresen Kai Olav Ellefsen, *Datasyn, eurobot 2009 - eit fagrapport gruppe 1&2, landsby 17*, (2009), 22–27.

[18] Kiefer Kuah, *Motion estimation with intel(r) streaming simd extensions 4 (intel(r) sse4)*, 2007, `http://software.intel.com/en-us/articles/motion-estimation-with-intel-streaming-simd-extensions-4-intel-sse4/`.

[19] Y. Lee, S.; Kay, *An accurate estimation of 3-d position and orientation of a moving object for robot stereo vision: Kalman filter approach*, (1990).

[20] Roger Boyle Milan Sonka, Vaclav Hlavac, *Image processing, analysis, and machine vision, third edition*, Thomson Learning, 2008.

[21] Hans Moravec, *Obstacle avoidance and navigation in the real world by a seeing robot rover*, tech. report CMU-RI-TR-80-03, Robotics Institute, Carnegie Mellon University doctoral dissertation, Stanford University, no. CMU-RI-TR-80-03, September 1980.

[22] NASA, *Solar terrestrial relations observatory*, 2006, `http://stereo.gsfc.nasa.gov/`.

[23] Mark Nixon and Alberto Aguado, *Feature extraction and image processing, second edition*, Academic Press (Elsevier Ltd.), 2008.

[24] *Opencv wiki*, 2009, `http://opencv.willowgarage.com/wiki/Welcome`.

[25] Richard E. Woods Rafael C. Gonzalez, *Digital image processing, third edition*, Pearson Education Inc., 2008.

[26] Andreas Koschan Reinhard Klette, Karsten Schlüns, *Computer vision three-dimensional data from images*, Springer-Verlag Singapore Pte. Ltd., 1998.

[27] Azriel Rosenfeld and Avinash C. Kak, Academic Press, Inc., 1982.

[28] Malin Space Science Systems, *Nasa selects msss to provide three science cameras for 2009 mars rover mission*, 2004, `http://www.msss.com/news/index.php?id=5`.

[29] C3 Technologies, *C3 - the world in 3d*, `http://www.c3technologies.com/`.

[30] Godfried Toussaint, *Solving geometric problems with the rotating calipers*, In Proc. IEEE MELECON '83, 1983, pp. 10–02.

[31] WEI Yi and S. Marshall, *Circle detection using fast finding and fitting (fff) algorithm*, **3** (2000), no. 1, 74–78.

Appendix A

Usage

A.1 Main

`Usage: ./main <color> <option>`

A.1.1 color input

1. BW (Y $\rightarrow$ intensity, rest discarded)
2. YUV ($|\ Y\ |\ U\ |\ Y\ |\ V\ |\rightarrow|\ Y\ |\ U\ |\ V\ |\ Y\ |\ U\ |\ V\ |$)
3. RGB ($|\ Y\ |\ U\ |\ Y\ |\ V\ |\rightarrow|\ R\ |\ G\ |\ B\ |\ R\ |\ G\ |\ B\ |$, conversion using integer-only maths)
4. YUV (thresholding enabled)

A.1.2 option

- 0: capture
- 1: anaglyphic stereo
- 2: Canny edge detection with edge direction
- 3: set colors for thresholding (and testing)
- 4/5: hough matching correspondence
- 6: Hough Circle (specific radius)

- 7: Hough Circle (unknown radius) (SLOW!)
- 8: Hough Circle (unknown radius, thresholded) (SLOW!)
- 9: Block-matching correspondence (current implementation is slow, working poorly.)
- 10: Canny edge detection with adjacent colors
- 11: Edge Block matching
- 12: Harris Corner detection
- 13: Normalization
- 14: Color Segmentation (R.E. Blake)
- 15: Fast Finding and Fitting - Symmetrical shape tracker (Circles)
- 16: Color recognition using HSV space + edges
- 17: Color segmentation + hole filling + smoothing
- 18: Color segmentation + smoothing (used to show segmentation results)
- 19: Normalization and RGB Gradient
- 20: Find cylinders
- 21: Find game elements (SOLUTION)
- capture (produces .png images for later calibration)

As can be seen, there is a few more implemented features in this list than what is described in the report. This is due that a lot of them are just minor variations of eachother, and some are parts of full implementations.

The most important options are 18, which is the color segmentation, and 21 which finds the game elements, and the distances. All of the later routines assume that the color input is RGB (2).

All functions except capture require mx1.bin, mx2.bin, my1.bin, my2.bin Please note that only a few combinations of colors and options actually work.

A.2 Calibration

First capture images, and store them in a list called calibration_images.txt, then run ./calibrate, remove all images that it is unable to match, and hope for the best.

the calibration writes 4 binary matrices with coefficients:
remapping in the x-direction: mx1.bin, mx2.bin
remapping in the y-direction: my1.bin, my2.bin

A.3 Example run

```
Tracking of Objects in 3D using Stereo Vision
(C) 2009-2010 Kai Hugo Hustoft Endresen. All Rights Reserved.

Color conversion options:
0: YUYV->BW
1: YUYV->YUV
2: YUYV->RGB
3: YUYV->YUV (thresholded)
Color[0-3]:
3
Algorithms/Options:
0: capture
1: anaglyphic stereo
2: Canny edge detection with edge direction
3: set colors for thresholding (and testing)
4/5: hough matching correspondence
6: Hough Circle (specific radius)
7: Hough Circle (unknown radius) (SLOW!)
8: Hough Circle (unknown radius, thresholded) (SLOW!)
9: Block-matching correspondence (current implementation is slow, working poo
10: Canny edge detection with adjacent colors
11: Edge Block matching
12: Harris Corner detection
13:
14: Color Segmentation (R.E. Blake)
15: Fast Finding and Fitting - Symmetrical shape tracker (Circles)
16: Color recognition using HSV space + edges
```

```
17: Color segmentation + hole filling + smoothing
18: Color segmentation + smoothing
19: Normalization and RGB Gradient
20: Find cylinders
21: Find game elements (SOLUTION)
Option[0-21]:
```

And when the program is exitting the output is as follows, provided everything works out nicely:

```
waiting for threads to finish..
capture thread for camera 1 stopped.
capture thread for camera 0 stopped.
freeing memory
all done.
```

Also 4 windows are presented,"left image","right image","rectified left", "rectified right", "disparity" and "location".

The left and right window shows the incoming video stream, with color thresholding (if applied), the "rectified left" and "rectified" right shows the segmented images, while "disparity" shows the matched objects and their distances.

To capture the image you are looking at you you can type "c", and to quit the application press "q". This needs to be done with one of the windows active, not in the terminal, because OpenCV's window displaying routines are used to handle input. Buttons such as "y,h,u,j,i,k" can be used to adjust the view height, downwards angle of the cameras as well as the focus length, for mapping out objects in the scene.

Appendix B

Measurements

Measurements	
distance (cm)	disparity (pixels)
020	209
025	165
030	141
035	120
060	79
092	44
120	38
121	37
122	37
123	36
124	36
125	35
127.5	35
129.5	34
131.5	34
135.5	33
141.5	31
146.5	30
151.5	29
156.5	28
166.5	26
176.5	25
186.5	24
196.5	22
206.5	22
216.5	20
226.5	20
227.5	19

Table B.1: Disparity measurements with distance between cameras of d=8 cm, and a resolution of 640x480.

Appendix C

Code

C.1 Code overview

File	Purpose
image.cpp/.hpp	Data structures for easy pixel access
main.cpp	mostly code related to displaying results and starting threads.
camera.cpp	opening capture devices, setting camera properties.
individual_processing.cpp	color conversion, remapping and parts of the algorithms that do not require both images.
conversion.cpp	Color conversion, conversion between data structures
correspondence.cpp	all algorithms related block-matching, thresholding, anaglyphic stereo etc.
hough.cpp/.hpp	Various implementations of Hough transform and derivates.

segmentation.cpp/.hpp	Color segmentation implementation as well as various routines relating to color recognition.
board.cpp/.hpp	Routines to map out the opponent robot from the image, as well as things not on the table, as well as finding the initial cylinder configuration.
ground_truth.cpp/.hpp	Routines to calculate image positions from real coordinates, and real coordinates from image position.
state.cpp/.hpp	access to variables relating to state in a threadsafe fashion.
position.cpp/.hpp	Routines to interpolate own and enemy position, as well as camera position relative to the robot's rotational axis.
communication.cpp/.hpp	POSIX message handling for communication with the navigation system and AI.
recognition/game_elements.cpp/.hpp	Contains the routines for finding and matching objects on the playing field, as well as their distances.
canny.cpp/.hpp	Canny edge implementation (not currently used, OpenCVs implementation, cvCanny is much faster)
calibration.cpp	Calibration implementation, just a modification of the examples in [13, p433]
inverse_filtering.cpp	Inverse filtering test, requires FFTW3.

position_generator.cpp/.hpp	Program to generate position data from testing from 2D Lidar readings. It pretends to be the navigation system, so that various testing can be done.

Since the amount of code is rather huge, it is only available electronically through DAIM, or on CD. The CD should be in appendix D.

Appendix D

CD

The CD contains two video files, one which shows the color segmentation in real-time, and one showing the distance to objects in movement. It also contains the source code for the stereo vision system, the source code for the *dummy* position generator, and the required files from the gateway on the robot, which all the different modules on the robot requires to communicate with each other. The structure is as follows:

```
src/computer_vision/
src/gateway/
src/lidar/
src/firewire_control/
movies/
```