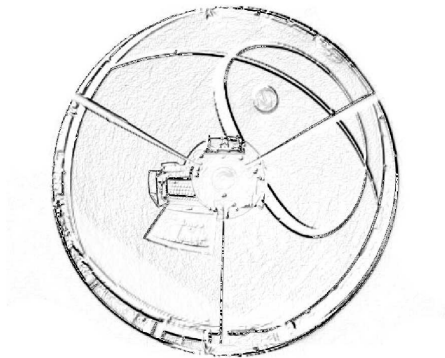




**Daniel Filipe de
Almeida Martins**

**Sistema de Processamento de Imagem para
Aplicações Robóticas**

Image Processing System for Robotic Applications

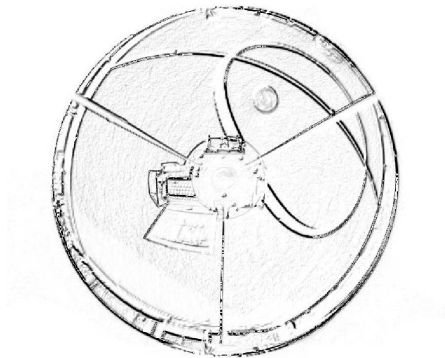




**Daniel Filipe de
Almeida Martins**

**Sistema de Processamento de Imagem para
Aplicações Robóticas**

Image Processing System for Robotic Applications





**Daniel Filipe de
Almeida Martins**

**Sistema de Processamento de Imagem para
Aplicações Robóticas**

Image Processing System for Robotic Applications

dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Electrónica e Telecomunicações, realizada sob a orientação científica do Doutor António José Ribeiro Neves, Professor Auxiliar Convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e do Doutor Armando José Formoso de Pinho, Professor Associado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

À Saruca...

o júri

presidente

Doutor Paulo Jorge dos Santos Gonçalves Ferreira

professor catedrático da Universidade de Aveiro.

Doutor Armando José Formoso de Pinho

professor associado da Universidade de Aveiro

Doutor Alexandre José Malheiro Bernardino

professor auxiliar do Instituto Superior Técnico da Universidade Técnica de Lisboa

Doutor António José Ribeiro Neves

professor auxiliar convidado da Universidade de Aveiro

agradecimentos

À minha família, pai e mãe, que sempre me apoiou. Ao meu maninho engenheiro que me ajudou a matar o tempo jogando PES. Aos meus amigos, David, Nelson, Tozé e Sr. Nuno pelo incentivo e por acreditarem no meu trabalho. Ao meu orientador e co-orientador pela enorme e incansável ajuda na escrita desta bela obra. Agradeço também aos elementos da equipa CMBADA, em especial ao grupo da visão.

Por fim, a quem a minha memória possa ter falhado. . .

palavras-chave

Visão robótica, câmaras omnidireccionais, transformada de Hough, detecção de contornos, detecção de objectos, análise de imagem, processamento em tempo-real.

resumo

A procura e reconhecimento de padrões foi sempre um desafio para a mente humana e sem dúvida a sua maior capacidade. Esta tese encontra-se inserida no domínio do RoboCup e apresenta uma solução em tempo-real para a detecção de objectos através do processamento de imagem. Ao longo do trabalho, desenvolvemos vários algoritmos para análise de imagem com vista a encontrar objectos através da sua cor e das suas propriedades morfológicas. Nos algoritmos baseados na procura por cor, foram usados métodos de segmentação de cor e procura radial na imagem, permitindo bom desempenho em tempo-real. A pesquisa por propriedades morfológicas baseia-se em algoritmos de detecção de contornos em conjunto com a transformada circular de Hough. Ambos algoritmos, procura por cor e por características morfológicas, provaram a sua fiabilidade, sendo capazes de boas taxas de detecção em condições de tempo-real. Para além do anteriormente referido, foi desenvolvida uma biblioteca para manipular imagens e assegurar uma abstracção sobre os possíveis modos da imagem e uma ferramenta para ajudar na calibração da visão perspectiva.

keywords

Robotic vision, omnidirectional cameras, Hough transform, edge detection, object detection, image analysis, real-time processing.

abstract

The search and recognition of patterns has always been a challenge for the human mind, and without any doubts its biggest capacity. This thesis is inserted in the RoboCup domain and presents a real-time solution to object detection using image analysis. In this work, we developed several image analysis algorithms to find objects based in their color and morphological properties. The color based search algorithms use color segmentation methods along with radial image scanning, allowing real-time performances. The morphological analysis is based in edge detection algorithms and the circular Hough transform. Both algorithms, search for color and morphological properties, proved their reliability, being capable of good detection ratios in real-time situations. Moreover, this thesis presents several tools, namely, an image library created to better manipulate the images and assure abstraction over the possible image modes acquired by digital cameras, and a tool to help in the perspective vision calibration.

Contents

1	Introduction	1
1.1	The CAMBADA team	1
1.2	Other teams in RoboCup middle size league	3
1.3	Objectives achieved	3
2	Hybrid vision system	5
2.1	Hardware architecture	5
2.2	Software architecture	6
2.3	Calibration of the vision system	7
2.3.1	Calibration of the camera parameters	8
2.3.2	Color calibration	8
2.3.3	Distance mapping calibration	8
2.4	Color processing sub-system	9
2.4.1	Color classification	9
2.4.2	Color extraction	10
2.4.3	Object detection	13
3	Color ball detection improvements	15
3.1	Shadowed ball recover algorithm	15
3.2	DistanceVsPixel validation algorithm	16
3.3	Results	18

4	Morphological ball detection	23
4.1	Overview	23
4.2	Edge detection	24
4.2.1	Sobel operator	25
4.2.2	Laplace operator	25
4.2.3	Canny operator	26
4.2.4	Choosing an edge detector	27
4.3	Hough transform	30
4.3.1	Implementation	30
4.4	Results	31
5	Developed Tools	35
5.1	The PerspectiveMapCalib	35
5.1.1	Results	40
5.2	ImageHolder class	40
6	Conclusions and future work	45
6.1	Future work	45
A	ImageHolder	47
A.1	Detailed Description	51
A.2	Constructor & Destructor Documentation	51
A.3	Member Function Documentation	52

Chapter 1

Introduction

The search and recognition of patterns has always been a challenge for the human mind, and without any doubts its biggest capacity. Given this potential, we look for synthesize this capacity in a real-time processing unit, in order to accelerate and improve the control of various processes.

In the last years, robotic vision has been under heavy studies in order to face new obstacles introduced by the RoboCup championship. In RoboCup, autonomous robotic agents have to play football according to FIFA rules with some modifications (see <http://www.robocup.org>). In order to control an autonomous robotic agent, surrounding environment information has to be perceived so the agent can react accordingly, which leads to the use of robotic vision systems.

Being a strongly color based environment, the vision system is the main sensorial source in RoboCup. Trying to mimic the human vision, solutions for image analysis in real-time have been created and explored by many teams around the world. More than “*By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team.*”, the RoboCup is a development platform, orientating and boosting the research in the robotic field.

1.1 The CAMBADA team

CAMBADA (acronym of Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture) is the RoboCup middle size league soccer team of the University of Aveiro, Portugal. This project started officially in October 2003 and, since then, the team has participated in several RoboCup competitions and Portuguese Robotics Festivals, achieving

the following results:

- Portuguese Robotics Open 2004: 5th place;
- Portuguese Robotics Open 2005: 4th place;
- Portuguese Robotics Open 2006: 3rd place;
- Portuguese Robotics Open 2007: 1st place;
- RoboCup'2007: 5th place;
- Portuguese Robotics Open 2008: 1st place.

The team also participated in the following events:

- RoboCup'2004;
- RoboCup'2006;
- DutchOpen'2006;

The CAMBADA robots were designed and completely built in-house. Each robot is built upon a circular aluminum chassis (with roughly 485 mm diameter), which supports three independent motors (allowing for omnidirectional motion), an electromagnetic kicking device and three NiMH batteries. The remaining parts of the robot are placed in three higher layers, namely:

- The first layer upon the chassis is used to place all the electronic modules such as motor controllers and battery status sensors;
- The second layer contains the main processing unit, currently a 12" notebook based on an Intel Core2Duo 2.0 GHz processor with 1024 MB of memory RAM;
- Finally, on the top of the robots stands a hybrid vision system consisting of an omnidirectional sub-system plus a perspective sub-system. Both sub-systems have a standard firewire camera, while the omnidirectional sub-system has also an hyperbolic mirror.

1.2 Other teams in RoboCup middle size league

Being the RoboCup World Championship a world wide competition, many teams compete in the Middle Size Soccer League. This allows the emergence of diverse solutions for the various problems inserted in this context.

Many teams are currently taking their first steps in 3D ball information retrieving [1, 2] while others are developing vision systems capable of detecting balls without a specific color [3, 4]. There are also some teams moving their vision systems algorithms to VHDL based algorithms taking advantage of the FPGA's versatility [1, 5]. Even so, for now, the great majority of the teams base their image analysis in color search using radial sensors [6, 7, 3, 8, 9].

1.3 Objectives achieved

Based in the CAMBADA vision system, this work covers two main areas: color image analysis and morphological image analysis. In the color analysis field, various improvements were performed in the ball detection system, with the introduction of color finding and validation algorithms. In morphological analysis, being an unexplored field in the CAMBADA vision system, we developed an early version of what will be the morphological image processing system.

This work shows a great improvement in the CAMBADA vision system, both in accuracy and in reliability. Furthermore, the improvements brought more color independence to the system, allowing the robot agents to play with balls of any color.

With a robust spatial ball localization system as objective, two stages were defined. In the first stage, the actual system, that is strongly based in color analysis, would be improved by including a ball validation system. In the second stage, a morphological based search algorithm would be developed to provide ball localization independently of the ball color. In parallel with this development, an image library denoted ImageHolder would be created to better manipulate the images and assure abstraction over the possible image modes. Furthermore, it was developed a tool to help in the perspective system calibration.

Chapter 2

Hybrid vision system

In the RoboCup domain, vision systems are without any doubts, the most important sensing system. Almost every team of the Middle Size League, if not every one, uses a camera as its main sensor, deploying the information extraction tasks into the image processing field.

In this chapter, it is presented a detailed description of the CAMBADA vision system, along with some techniques used to make it a real-time, robust and efficient system.

2.1 Hardware architecture

In the last few years, the vision system of the CAMBADA team has evolved into an hybrid vision system, formed by an omnidirectional vision sub-system and a perspective vision sub-system, that together can analyze the environment around the robots, both at close and long distances (see Fig. 2.1).

The omnidirectional vision system [10] is based on a catadioptric configuration implemented with a firewire camera (PointGrey Flea2 camera with a 1/3" CCD sensor and a 4.0mm focal distance lens) and an hyperbolic mirror. This camera can work at 30 fps using the YUV 4:2:2 or RGB modes with a resolution of 640×480 pixels. The perspective vision system uses a low cost firewire camera (BCL 1.2 Unibrain camera with a 1/4" CCD sensor and a 3.6mm focal distance lens). This camera can deliver 30 frames per second (fps) using the YUV 4:1:1 mode with a resolution of 640×480 pixels.

The information regarding close objects, like white lines of the field, other robots and the ball, are acquired through the omnidirectional system, whereas the perspective system is used to locate other robots and the ball at long distances, which are difficult to detect using the omnidirectional vision system.



Figure 2.1: The hardware architecture of the vision system developed for the CAMBADA robotic soccer team. On the top, the omnidirectional sub-system with a camera pointing to an hyperbolic mirror. On the bottom of the image, the perspective sub-system with a camera pointing towards the field.

Our system is prepared to acquire images in RGB 24-bit, YUV 4:2:2 or YUV 4:1:1 format. However, we use the HSV color space for color calibration, due to its special characteristics [11].

2.2 Software architecture

The software architecture is based on a distributed paradigm grouping main tasks in different modules. This permits a better understanding of the software work-flow and easier implementation of future improvements. The software can be split in three main modules, namely the *Utility Sub-System*, the *Color Processing Sub-System* and the *Morphological Processing Sub-System*, as can be seen in Fig. 2.2. Each one of these sub-systems labels a domain area where their processes fit, as the case of *Acquire Image* and *Display Image* in the *Utility Sub-System*. As can be seen in *Color Processing Sub-System*, proper color classification and extraction processes were developed, along with an object detection process to extract information, through color analysis, from the acquired image. The *Morphological Processing Sub-System*, explained in detail in Chapter 4, presents an early version of a color independent ball detection algorithm, that is still under heavy study and development.

Despite the obvious differences between the omnidirectional and the perspective sub-systems, the software architecture used in both is the same, changing only the *Image Mask & Radial Sensors* and the *Distance Mapping Image*.

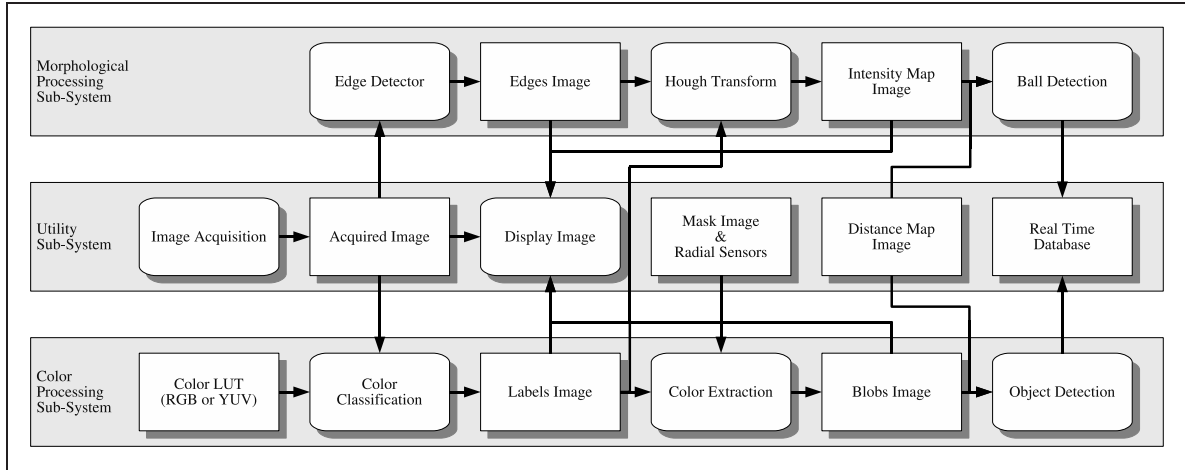


Figure 2.2: The software architecture of the vision system developed for the CAMBADA robotic soccer team.

The use of radial search lines, also called radial sensors, for object detection accelerates the object detection algorithm without compromising its accuracy. The acceleration is achieved through the reduction of the search area in the image, since the radial sensors cover only a small percentage of the image [12]. This method can be compared to other techniques like subsampling. Taking the study in [13, 14] where a “Fast and High Performance Image Sub-sampling Using Feedforward Neural Networks” is explored, it is easy to see that subsampling requires an extra step in processing while with radial search lines do not. Another disadvantage is the possible distortion of the subsampled image, and in this special case, the time consumed during the neural network learning phase. Moreover, the polar coordinates used in radial search line analysis simplifies the description of the objects, by using distance and angle measures instead of their bounding box.

2.3 Calibration of the vision system

Calibrating a robotic vision system tends to be a slow process. Many factors have to be taken into account to provide a good calibration, e.g. the light conditions and the physical structure of the vision system. If the robotic vision system has to be used under natural light, for example near a window or outside, the light conditions can change very quickly making it even harder to calibrate. Additionally, when a robotic system uses vision as its main sensorial element (as in the CAMBADA team robots), it becomes crucial to have a good calibration of the vision system, under the severe risk of having a nonfunctional robotic system.

In the CAMBADA vision system, the calibration consists in three distinct points, namely

the calibration of the camera parameters, the calibration of colors and distance mapping calibration. These three steps must be replicated for both cameras, omnidirectional and perspective, and for all robots.

2.3.1 Calibration of the camera parameters

To obtain a good color image, some parameters in the camera must be calibrated, namely exposure, white-balance, gain and brightness. These parameters are calibrated with an automatic tool called *AutoCalib*, described in [15, 16]. Once this calibration is done, the resulting image color is balanced. This allows to use the same color calibration configuration (referred next) in every robot as described in [16].

2.3.2 Color calibration

In order to use color analysis over the received images, a color calibration must be performed for defining the color range, usually as a volume in a certain color space, associated to each color class. This calibration is performed in HSV (Hue, Saturation and Value) color space, suitable for color segmentation [11, 17, 18]. Again, this calibration can be done with the tool *AutoCalib*, previously mentioned.

2.3.3 Distance mapping calibration

This important process is responsible to create the distance map image that will be used to convert image coordinates (pixels) into real coordinates (meters) relative to the robot. Mapping the objects found in the image into the real world coordinates is a crucial task for modeling the environment around the robot, allowing the robotic agent to know its localization (using a localization algorithm based in the white lines) and the surrounding objects position. To implement this operation, it is created an array, called distance map image (see Fig. 2.3), with dimension “image width” \times “image height” where each entry is a real world coordinate represented by x and y axis in meters. In this case, it is used an array of 307 200 (640×480) entries. Using the image coordinates (row and column) as an index to the array, the real world coordinate (x, y) is returned. For the omnidirectional sub-system, the distance map is obtained with the algorithm and tools described in [19]. To the perspective sub-system, the distance map is created using a tool named *PerspectiveMapCalib* developed by the author and explained in Section 5.1.

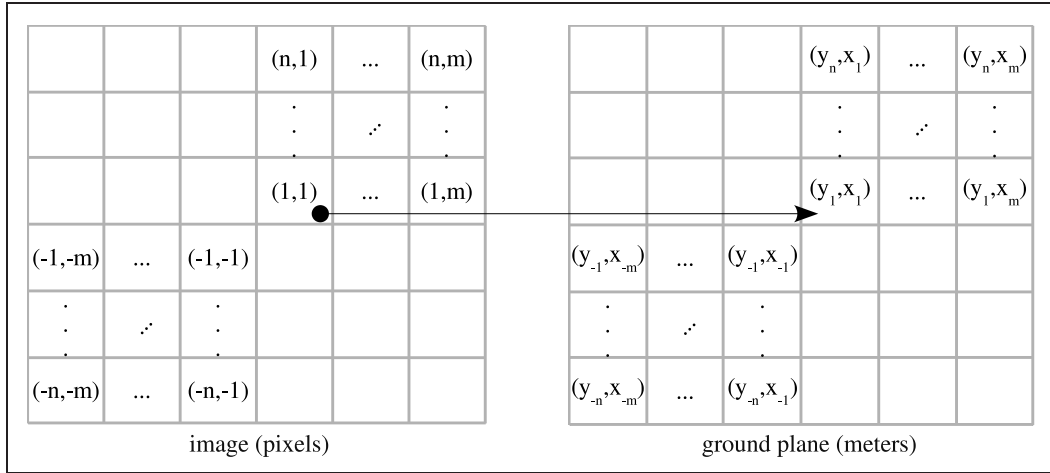


Figure 2.3: Example of a distance map image. On the left an image with referential (n, m) in pixels. On the right, an image with referential (y, x) in meters.

2.4 Color processing sub-system

In RoboCup environments, image analysis can be simplified due to the color codes of the objects. Black robots play with an orange ball on a green field that has white lines, making the analysis of the color of each pixel in the image an important feature for object detection.

2.4.1 Color classification

To take advantage of the color code used, the acquired image is processed using a look-up table (LUT) for fast color classification [12, 16]. The LUT accelerates the color classification by expanding the image color components into a big table. The table has 16 777 216 entries (2^{24} , 8 bits for red, 8 bits for green and 8 bits for blue in the case of RGB images), each 8 bits wide, occupying 16 MBytes in total. If another color space is used, the table size remains the same, changing only the meaning of each component. Each entry has 8 bits, each one expressing the presence or absence of a predefined class (associated to a color range). This allows the same color to be classified into several classes at the same time, being possible to classify up to 8 different classes. To classify a pixel, its color is used as an index to the table, and the value (8 bits) read from the table is the pixel classification, also named as “pixel color mask”.

2.4.2 Color extraction

Before the system proceeds to process the received image, some operations are performed to simplify the image analysis. Some regions in the image can be discarded without losing any information. Taking the omnidirectional sub-system as example, the robot itself appears reflected in the mirror, as well as the sticks that hold the mirror. These regions, along with image areas outside the mirror, don't include any useful information and therefore must be discarded to avoid erroneous information. The same problem appears in the perspective sub-system, where objects above the horizon line must be discarded, or else they would appear projected at infinite distance in the ground plane. In the perspective sub-system, very close objects are discarded as they are analyzed by the omnidirectional sub-system.

To discard these regions, it is used an image with the configuration of the pixels to be discarded as presented in Fig. 2.4. In the figure, white pixels represent the valid area in the image and black pixels the area to be discarded. Besides simplifying, this step potentially accelerates the object detection processing and the detection of false objects is avoided.

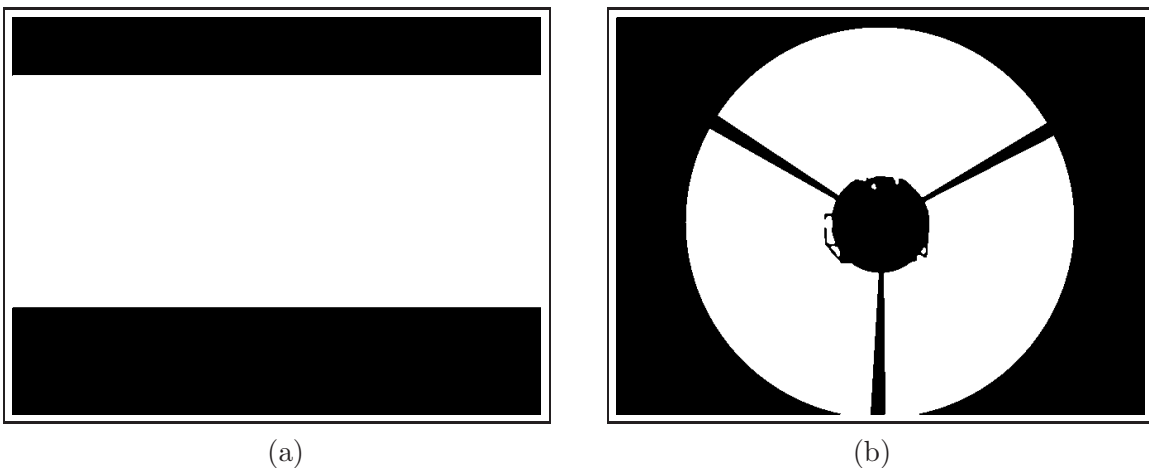


Figure 2.4: Example of images containing the valid pixels to be processed. In (a), from the perspective sub-system and in (b), from the omnidirectional sub-system. Black pixels mark the pixels that are discarded.

Proceeding to the extraction of the color information from the image, it is performed through radial search lines instead of a full image analysis. The use of radial search lines (also called radial sensors) has proved to be very effective and has a processing time almost constant, a desired property in real-time systems [12]. This technique reduces the number of pixels to be processed and, instead of the full image, only pixels covered by radial sensors are processed. Using radial sensors, the valid area of the image is reduced to near 21.9% for

the omnidirectional and 17.0% for the perspective system, when compared with the input image of 640×480 pixels. This results in a huge performance improvement, allowing this system to work in real-time applications without compromising its efficiency. To create these search lines it is used the Bresenham algorithm [20]. The search lines are created starting in the position of the robot center in the image, towards the image limits, disposed in radial orientation as presented in Fig. 2.5.

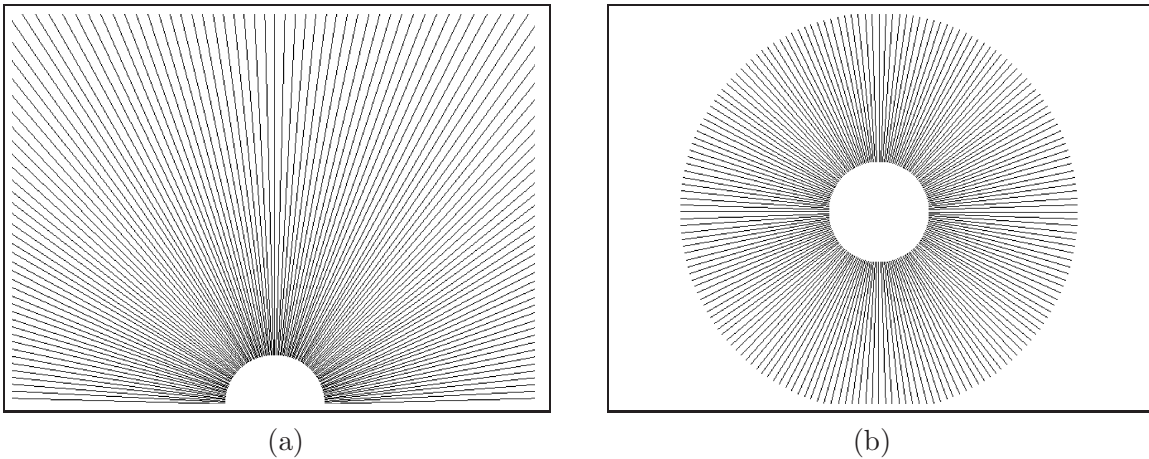


Figure 2.5: The representation of the radial sensors used for color extraction. In (a) from the perspective sub-system and in (b) from the omnidirectional sub-system. The radial lines mark the pixels that will be processed.

The search is then conducted, one sensor at a time, from the center of the robot to the peripheral area through the pixels covered by these radial lines. This search looks for areas with specific colors and transitions between two colors along the sensor. As this step can become complex, three distinct algorithms were developed to use in different occasions. Follows a description of each algorithm and when they are applied.

To detect areas with a specific color, we developed an algorithm with a good image noise removal based on a median filter. Each time a pixel with a color of interest is found, a predefined number of the following pixels is analyzed. If a number of pixels with the same color of interest is found then the current sensor is marked as having this color, otherwise the pixel is discarded and the search continues.

An extension to the previous algorithm is applied when searching for the orange color, improving the ball detection. This algorithm was developed by the author, and is further explained in Section 3.1.

In order to detect color transitions, was developed an algorithm with median filter, illustrated in Fig. 2.6. This algorithm is used to find the field white lines, avoiding the image noise.

The white lines are detected as a transition between green and non-green with a minimum number of white pixels. When the search reaches a non-green pixel, a predefined number of the following pixels (W_c) is searched and the number of non-green pixels and white pixels is calculated. Then, another search is performed in a small window after the pixel currently being tested (W_a), and before (W_b), calculating the number of green pixels. Finally, these calculated values are compared with predefined thresholds and accepted, or not, as color transitions.

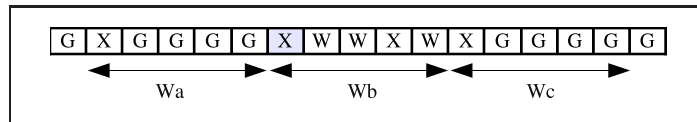


Figure 2.6: An example of a transition between green and white. “G” means green pixel, “W” means white pixel and “X” means pixel with a color different from green or white.

This first analysis extracts basic information such as the first pixel where the color was found and the number of pixels with that color that have been found in the same search line. To speed up the following process, this information is grouped into a list of colors.

The previous information is processed and grouped according to spatial proximity and size, creating what it is called of blob descriptors. The blob descriptors are themselves grouped by color and they contain the following statistical and spatial information essential to the object detection process:

- Average distance to the robot;
- Mass center;
- Angular width;
- Number of pixels (of the color to which this blob belongs);
- Number of green pixels between blob and the robot;
- Number of green pixels after blob.

The fields of blob descriptors were chosen based in the image analysis using polar coordinates. The use of polar coordinates in analysis is an advantage due the fact that it is being used over an omnidirectional image, simplifying the objects descriptor into distance and angle measures instead of their bounding box. The perspective image also has this advantage due to the position of the radial sensors.

2.4.3 Object detection

This processing step receives blob descriptors and processes their information in order to detect objects of interest, which can be white lines, black robots or an orange ball. When the objects are found, their position in the image (pixel coordinates) is converted into the real position (meters) relative to the robot using the *Distance Mapping Image*, and is sent to the upper layers of processing through the real-time database (RTDB) [21]. The three kinds of objects have different properties, and there are different algorithms to detect them. Follows a description of each algorithm.

White lines detection

Because detection of white lines is somehow trivial, at this point the white lines were already found as transitions between green and white in the *Color extraction* process and require only a small processing. In this process, the white lines are only converted to real coordinates and sent to the real-time database.

Obstacles detection

The obstacles position is calculated by applying the following algorithm to the black blob descriptors:

1. If the angular width of one blob is greater than 10 degrees, split the blob into smaller blobs, in order to obtain an accurate information about obstacles.
2. Calculate the information for each blob.
3. The position of the obstacle is given by the distance of the blob relatively to the robot. The limits of the obstacle are obtained using the angular width of the blob.

Ball detection

The ball position is calculated by applying the following algorithm to the orange blob descriptors:

1. Perform a first validation of the orange blobs using the information about the green pixels after and before the blob.

2. Validate the remaining orange blobs with the ball validation system described in Section 3.2.
3. Using the distance from the ball candidates to the robot, choose the best candidate blob. The position of the ball is the mass center of the blob.

This brief description is extensively explained in Chapter 3, entirely dedicated to color based ball detection. In Chapter 4, a morphological based ball detection system is described and explained.

Chapter 3

Color ball detection improvements

Being the ball detection a key issue, its correct detection is of most importance, and must be as accurate and efficient as possible. Taking into account the *color processing sub-system* (see Fig. 2.2), a deep analysis of the color ball detection algorithm showed that it could be described by the following logical operations:

1. Search for orange areas along the sensors;
2. Create the orange blob descriptors;
3. Validate the orange blob descriptors.

An improvement in any of this operations would also improve the final result of the color ball detection algorithm. To push the ball detection robustness a little further, we developed and implemented two new algorithms. The first, *Shadowed ball recover*, is explained next, and fits inside the first logical operation described before. The second, *DistanceVsPixel validation*, is held inside the third logical operation and is explained ahead in this chapter.

3.1 Shadowed ball recover algorithm

During early developments in the *DistanceVsPixel validation* algorithm described in Section 3.2 of this chapter, it was realized that the ball size in pixels can be strongly affected by the shadow of the ball casted over itself. The shadow effect makes the bottom side of the ball darker and therefore harder to be included in color segmentation without including other dark orange objects that could be present in the surrounding environment. Also, because the

color segmentation is based in color space volumes, minor changes in illumination could lead to a deficient color segmentation of the ball and eventually failing to detect the ball.

To reduce these effects in ball detection, it was developed an algorithm to recover darker orange pixels, previously discarded during the segmentation process due to ball shadow cast over itself. Next, the algorithm is presented.

After the first search for orange areas along the radial sensors, a second search is conducted along the sensors where orange was found. This second search starts in the first orange pixel found in the sensor in direction to the center of the robot. During the search, each pixel of the sensor is again compared with modified values of the orange color volume, previously defined during *color calibration* (described in Section 2.3.2). This modified orange color volume allows to retrieve darker orange pixels previously discarded, being its principal modification the much lower bottom limit in Value component (of HSV color space). To prevent finding false dark orange pixels (not belonging to the ball), the search in the sensors is limited in three ways. The search stops in one sensor and jumps to the next sensor when:

- Is found a predefined number of darker orange pixels;
- The search is done along a predefined number of pixels;
- Is found a pixel with a different valid color (green for example).

In Fig. 3.1 it is possible to see the effect of the algorithm applied to the omnidirectional sub-system. Results in the perspective sub-system are similar, using exactly the same algorithm. As we can see, the detected ball (orange blob with cyan cross over it) in the objects image (right side of Fig. 3.1), has an improved shape due to the shadowed ball recovery algorithm.

3.2 DistanceVsPixel validation algorithm

In a validation system, two types of errors must be avoided: false negatives and false positives. This study focus its attention in the reduction of the second type of errors (false positives) relative to ball detection, thus making valid all the existing information. This is a primary objective in order to obtain reliable ball information, essential to the good performance of the robotic agents.

By observation, it was noticed that a large number of false positives was due to the light distortion created by the color transitions, for example near the white lines in the field. Color transitions between white and green creates light distortion sometimes visible on the captured

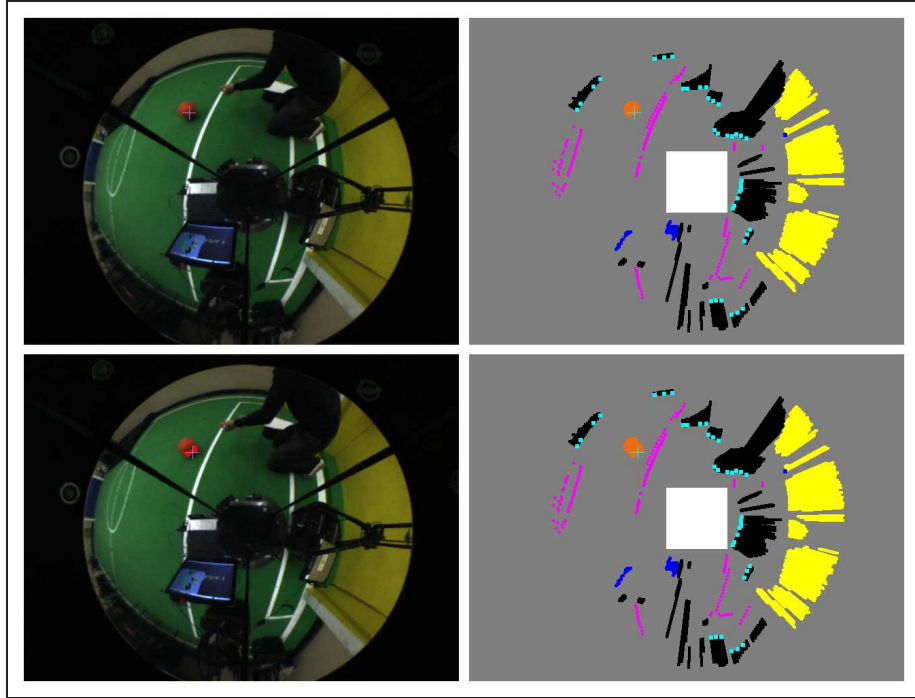


Figure 3.1: On the upper part of the figure: on the left a real image without the shadowed ball recovery algorithm, and on the right the detected objects. On the bottom part of the figure: on the left a real image with the shadowed ball recovery algorithm (recovered pixels represented with red color in real image), and on the right the detected objects.

image. This fact introduces lots of false positives all over the field, creating potential confusion in the higher layers of software.

To eliminate this false positives, it was suggested the use of a thresholding function, later called *DistanceVsPixel validation algorithm*. This thresholding function would be an unidimensional function, with the distance between the robot and the object as the independent variable. The use of an unidimensional function, allows an easy implementation, and less computational effort during the validation process, once it is implemented. To select the best discriminant variable, the vision system was presented with three different scenarios, all of them under the same light conditions. As the search for the discriminant variable was resumed to the information presented in the blob descriptors, during this three tests only two variables were logged, being them:

- Area of the object (in pixels);
- Angular width of the object (in degrees).

Note that only after succeeding in the validation test, the found orange blob can be called a ball, instead of object.

These three tests were based in one orange object standing still in the field, and the robot moving along a path in the field. In the first test the orange object was a square paper of 0.28×0.28 meters, in the second test a size 5 ball, and in the third test no objects at all were left on the field. In the three tests, the robot moved through the same path in the field.

Attending to the results of the tests, shown in Fig. 3.2, the discriminant variable was easily chosen as being the *area of the object*. Also, note that this results show the behavior of the omnidirectional sub-system. Similar results were obtained with the perspective sub-system, where datum follow the same patterns.

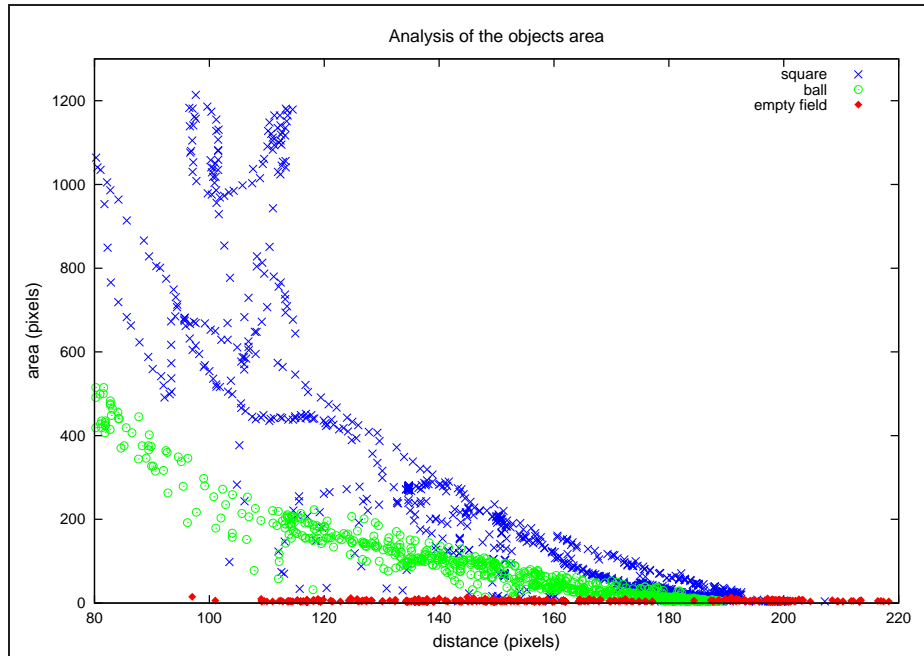
Being approximately piecewise linear, the function relating the *distance between robot and object* and the *area of object*, can be implemented through two simple linear functions. The use of linear functions makes either the implementation, as well as a possible posterior adjustment of the system, much easier. Using the selected variables, a validation system based on two thresholding linear functions ($y = mx + b$) were implemented to eliminate false ball positives. In Fig. 3.3, the thresholding functions for the two vision sub-systems are represented along with some ball samples. Samples below the thresholding functions are discarded, cataloging the samples above the threshold as valid balls.

This method, besides simple, is robust, fast and easy to implement. Furthermore, its parameters, m and b , are simple to understand, allowing fast and easy adjustments.

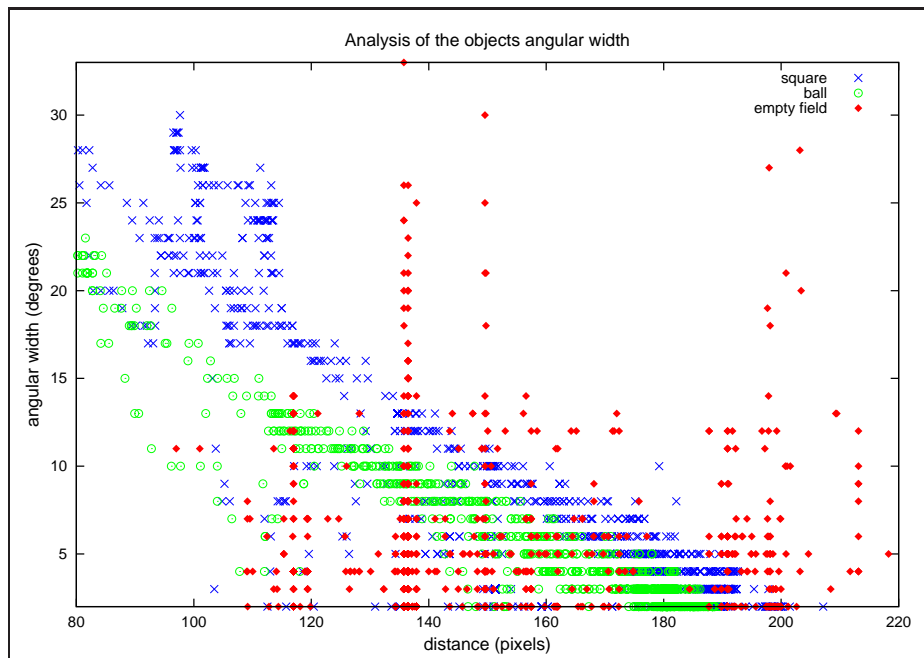
3.3 Results

To assure good results in RoboCup competitions, the system was tested with the optimizations described above. Testing systems in working conditions generates much more realistic results. For that purpose, the robot was moved along a predefined path through the robotic football field, leaving the ball in a known location. The ball position given by the robot is then compared with the real position of the ball. The results in this test may be affected by the localization algorithm errors and the robot bumps while moving, being these external errors outside the scope of this study.

The robot path in the field may be seen in Fig. 3.4, along with the measured ball position. In Table 3.1 it is presented an analysis of the acquired data during the test. In Fig. 3.4, it is possible to notice that the average of the measured positions of the ball is almost centered in the real ball position. In Table 3.1 the results for the *MidField* case show the effectiveness



(a)



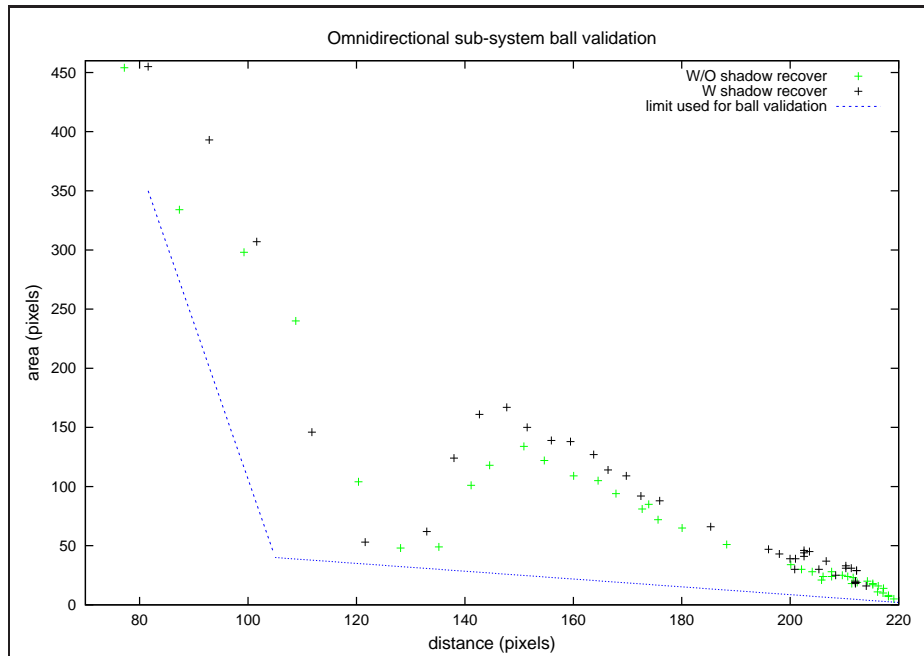
(b)

Figure 3.2: In (a), the comparison between the three tests using the area in pixels. In (b), another comparison using the objects angular width in degrees. Note that in both figures the distance is in meters.

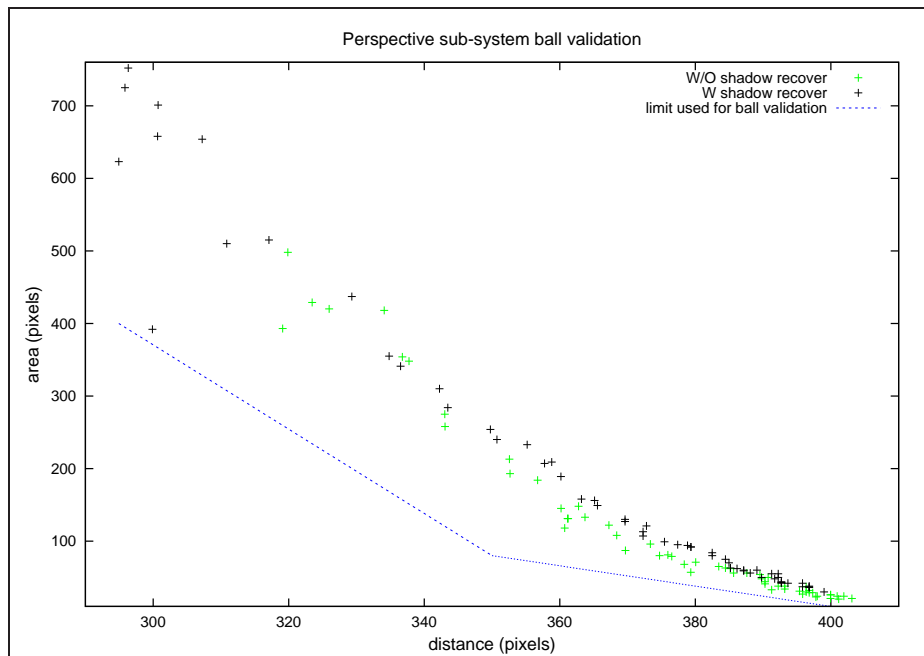
of this system, with a very high detection ratio (near 100%), and a great accuracy, with the average measures very near the real ball position. As this test samples are only from the omnidirectional sub-system, the results shown in the *Penalty* case are not so good, due to the distortion of the acquired image at long distances introduced by the physical structure itself. Even so, in about 76.8% of the samples, the ball was detected with a standard deviation below 0.30 meters. Notice the average processing time of 12 milliseconds (ms), below the upper limit of 33 ms necessary for a real-time system running at 30 Hz.

Experiment	Real Position	Measures			
		Average	Std	Detection ratio	Processing time
MidField	(0.00, 0.00)	(-0.10, -0.06)	(0.19, 0.11)	99.3%	12 ms
Penalty	(-2.39, 0.0)	(-2.56, 0.11)	(0.26, 0.22)	76.8%	12 ms

Table 3.1: Some measures obtained for the experiments presented in Fig. 3.4. All the measures are in meters, except for *Detection ratio* that shows the percentage of samples collected where the ball was detected and *Processing time* presented in milliseconds.

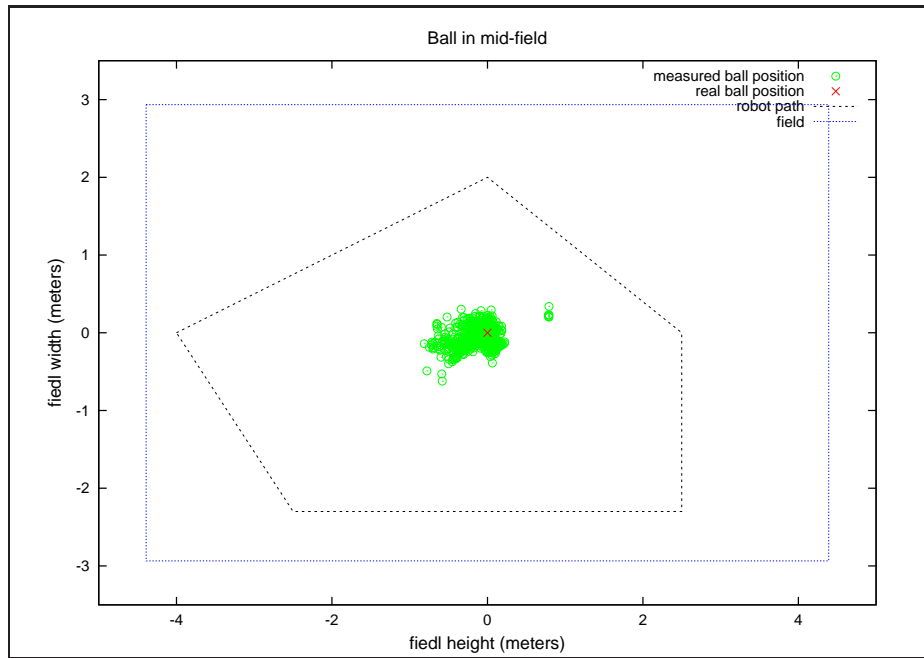


(a)

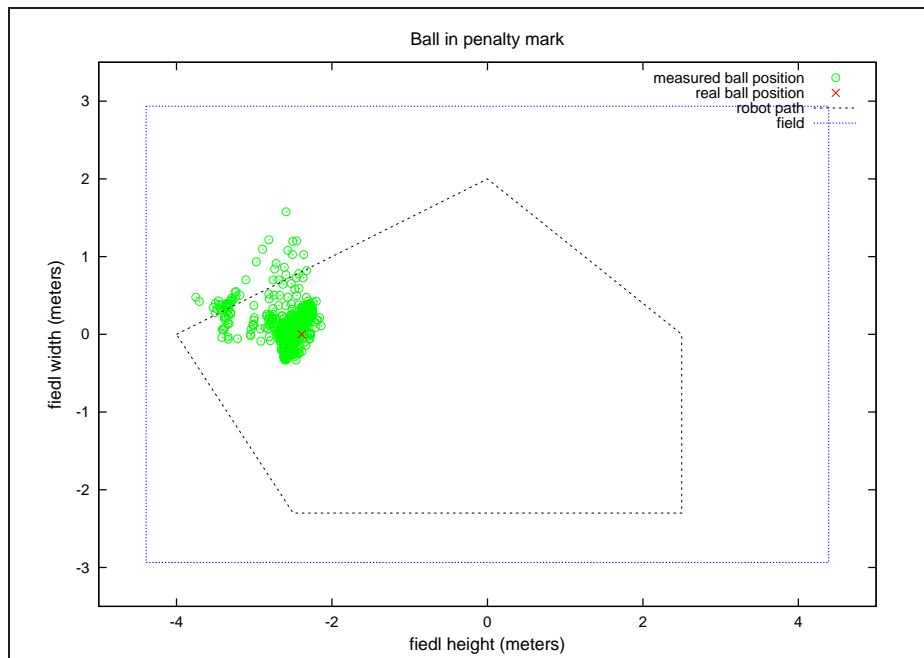


(b)

Figure 3.3: The ball validation thresholding function for the omnidirectional in (a), and the perspective sub-system in (b). The behavior of the acquired data, for the omnidirectional sub-system, around distance 120, is due to some occlusion of the ball by the mirror holding structure. This behavior is taken into account in order to improve the algorithm.



(a)



(b)

Figure 3.4: Experimental results obtained by the proposed omnidirectional vision system. In (a) the ball was positioned in the center of the field. In (b) the ball was positioned in the penalty mark. The robot has performed a defined trajectory and the position of the ball was registered. Both axis in the graphics are in meters.

Chapter 4

Morphological ball detection

Extracting information from images can be done in various ways. In the previous chapters, it was shown many kinds of information extraction algorithms based on color analysis. These algorithms work well in environments where the color codes are used to distinguish objects, but when objects don't have a predefined color, the algorithms fail.

On the RoboCup Middle Size League, the color codes tend to disappear as the competition evolves. Being the color of the ball the next color scheduled to disappear, a solution was developed by the author to detect balls independently of their color. This solution is based in a morphological analysis of the image, being strictly directed to detect round objects in the field, in this case the ball. This chapter describes the study and implementation of the algorithm made by the author.

4.1 Overview

Morphological object recognition through image analysis has become more robust and accurate in the past years, whereas still very time consuming even to modern personal computers [22, 23, 24]. Being the RoboCup a real-time environment, available processing time can become a big constrain when analyzing large amounts of data or executing complex algorithms. Many algorithms found during a previous research showed their effectiveness but, unfortunately, their processing time is in some cases over 1 second [22].

The *Morphological Processing Sub-System* (see Fig. 2.2) was developed to overcome this obstacle using a two pass analysis to detect the ball. First, the image is searched for points of interest (potential locations) where balls can be found. Then a validation system is applied to the spots previously found to discard false ball locations. A similar approach was found in

[22].

The search for potential spots is conducted taking advantage of morphological characteristics of the ball (round shape) using a feature extraction technique known as *Hough* transform. First used to identify lines in images, the *Hough* transform has been generalized, through the years, to identify positions of arbitrary shapes, most commonly circles or ellipses, in images. The author's interest and implementation of this technique in *Morphological Processing Sub-System* is explained in Section 4.3.

To feed the *Hough* transform process, it is necessary a binary image (image where the pixels can only have two values) with the edge information of the objects. This image, *Edges Image*, is obtained using an image operator commonly called *edge detector*. In Section 4.2 it is presented an explanation about this process and its implementation.

The validation system for the morphological ball detector is still not implemented. This issue is referred in Section 6.1, where some proposals are made regarding this problem.

4.2 Edge detection

Being this the first image processing step in the morphological detection, it must be as efficient and accurate as possible in order to not compromise the following processes. Besides being fast to calculate, the pretended resulting image must be absent of noise as much as possible, with well defined boundaries and be motion blur tolerant. Be tolerant to motion blur means that even when the objects present blur deformation in the image, the edge detector can retrieve its contours. In Fig. 4.1 its shown an example of motion blur deformation.

Before making the choice, some image edge detectors were compared. The comparison was made between the three main image operators used to find edges, Sobel, Laplace and Canny as these are, by far, the most well known and efficient image operators used in edge detection. The tests occurred under two distinct situations: with the ball standing still and ball moving fast through the field. The test with the ball moving fast was realized to study the motion blur effect in the edge detectors on high speed objects captured with relatively low frame rates (30 fps). In each test, two balls of different colors were used. Figure 4.2 shows an image of each studied scenario.

The three operators are based on convolving the image with a small, separable, and integer valued filter in horizontal and vertical direction and are therefore relatively inexpensive in terms of computations. In both tests, all the operators used a convolving window of size 3.

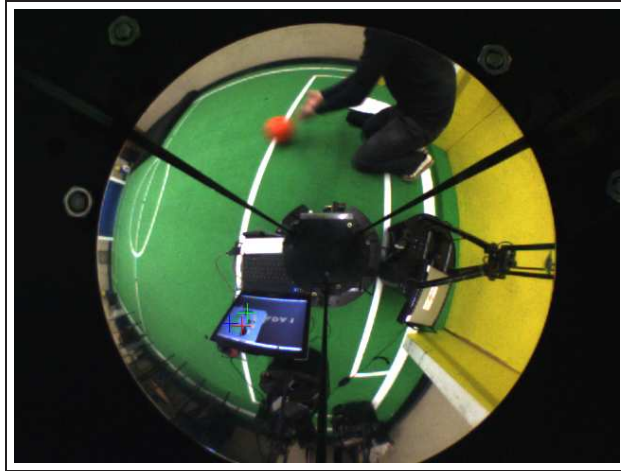


Figure 4.1: Example of motion blur effect. The orange object in the image is in fact an orange ball being pushed at high speed, creating the fade effect called motion blur.

4.2.1 Sobel operator

The Sobel operator is widely used in edge detection algorithms [25, 26, 27, 28]. It is a discrete differentiation operator, that computes an approximation of the gradient of the image intensity function. This approximation is relatively crude, in particular for high frequency variations in the image.

By calculating the gradient of the image intensity at each point, the operator gives the direction of the largest possible increase from light to dark and the rate of change in that direction. This shows how “abruptly” or “smoothly” the image changes at that point, and therefore how likely that part of the image represents an edge, as well as how that edge is likely to be oriented.

In Fig. 4.3, the resulting images of the Sobel edge detector applied to the images in Fig. 4.2.

4.2.2 Laplace operator

The Laplace operator, as the Sobel, is commonly used in image processing as an edge detection algorithm [28, 29]. Also called *Laplacian*, it is denoted by Δ or ∇^2 and is a differential operator.

In Fig. 4.4, the resulting images of the Laplace edge detector applied to the images in Fig. 4.2.

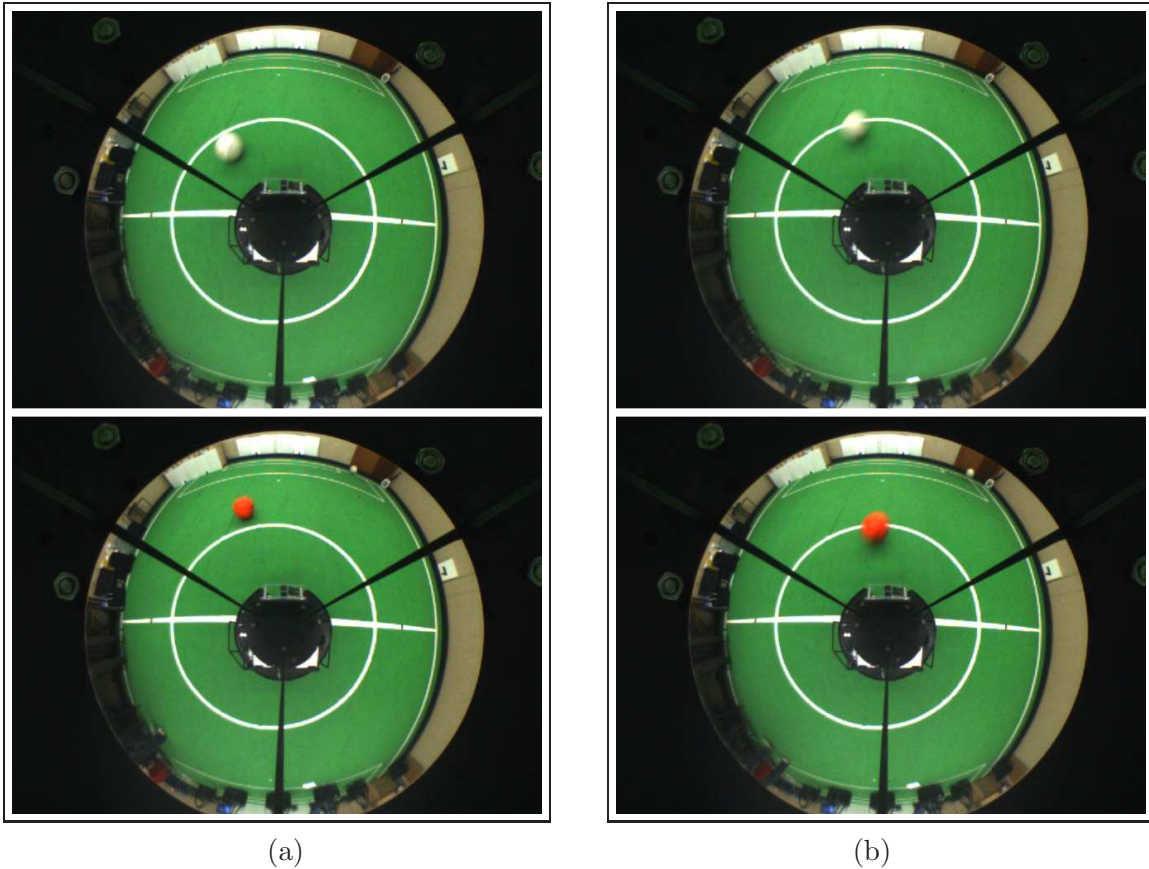


Figure 4.2: Two examples of typical images captured in the RoboCup environment. In (a) a test image with the ball standing still and in (b) with the ball moving at high speed. The upper images refer to the white ball, and the bottom images to the orange ball.

4.2.3 Canny operator

The Canny edge detection operator uses a multi-stage algorithm to detect a wide range of edges in images. The Canny operator was developed to be an optimal edge detection algorithm [30, 31, 32, 33], presenting the following features as the optimal ones:

- Good detection - the algorithm should mark as many real edges in the image as possible;
- Good localization - edges marked should be as close as possible to the edge in the real image;
- Minimal response - a given edge in the image should only be marked once, and where possible, image noise should not create false edges.

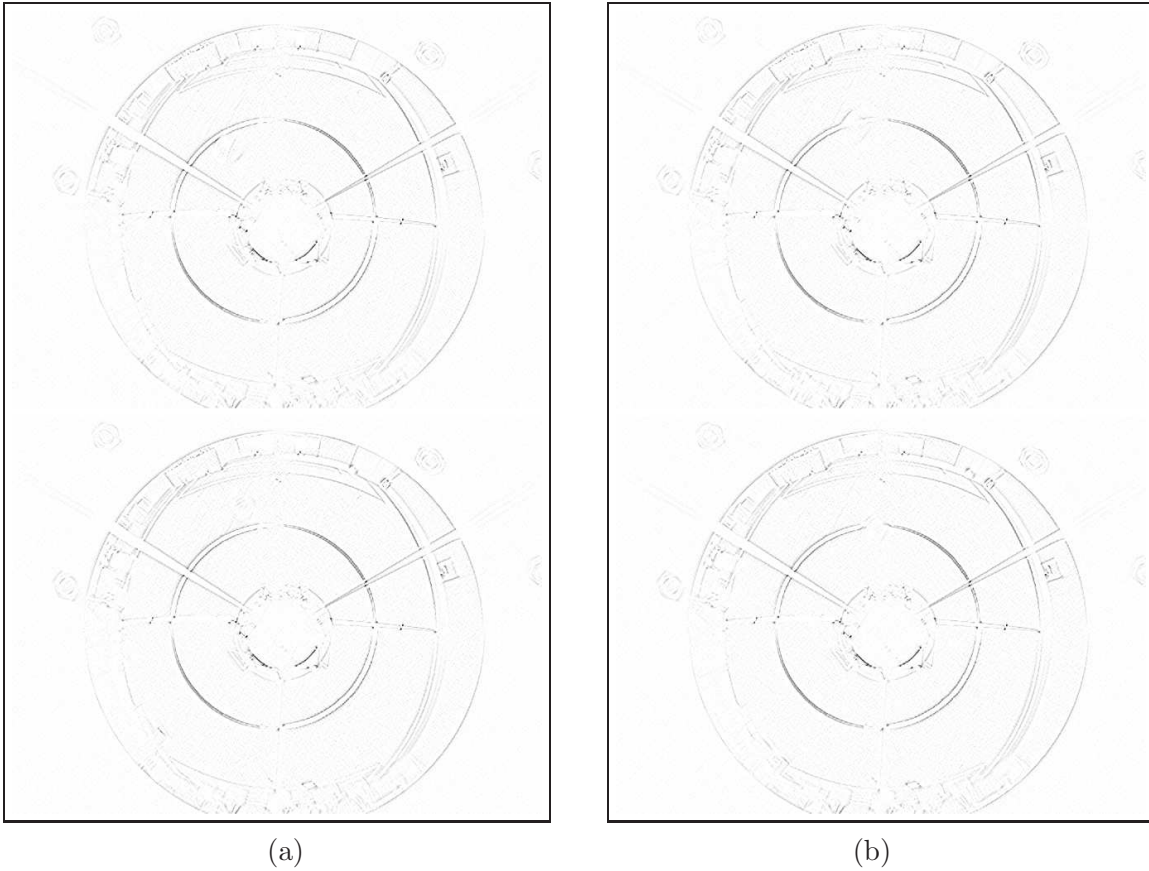


Figure 4.3: Two examples previously presented in Fig. 4.2, now with the Sobel operator applied. In (a) a test image with the ball standing still and in (b) with the ball moving at high speed. The upper images refer to the white ball, and the bottom images to the orange ball.

In the Canny operator, these requirements are obtained using calculus variations, a technique used to find a function which optimizes a functional. The optimal function in Canny detector is described by the sum of four exponential terms, but can be approximated by the first derivative of a Gaussian. Using a Canny filter it is possible to obtain an image of edges with edges with size of 1 pixel, due to its non-maximal suppression properties.

In Fig. 4.5 the resulting images of the Canny edge detector applied to the images in Fig. 4.2 are presented.

4.2.4 Choosing an edge detector

To choose the best edge detector, the results from the tests will be compared taking in account the image of edges and processing time needed by each edge detector. In one hand,

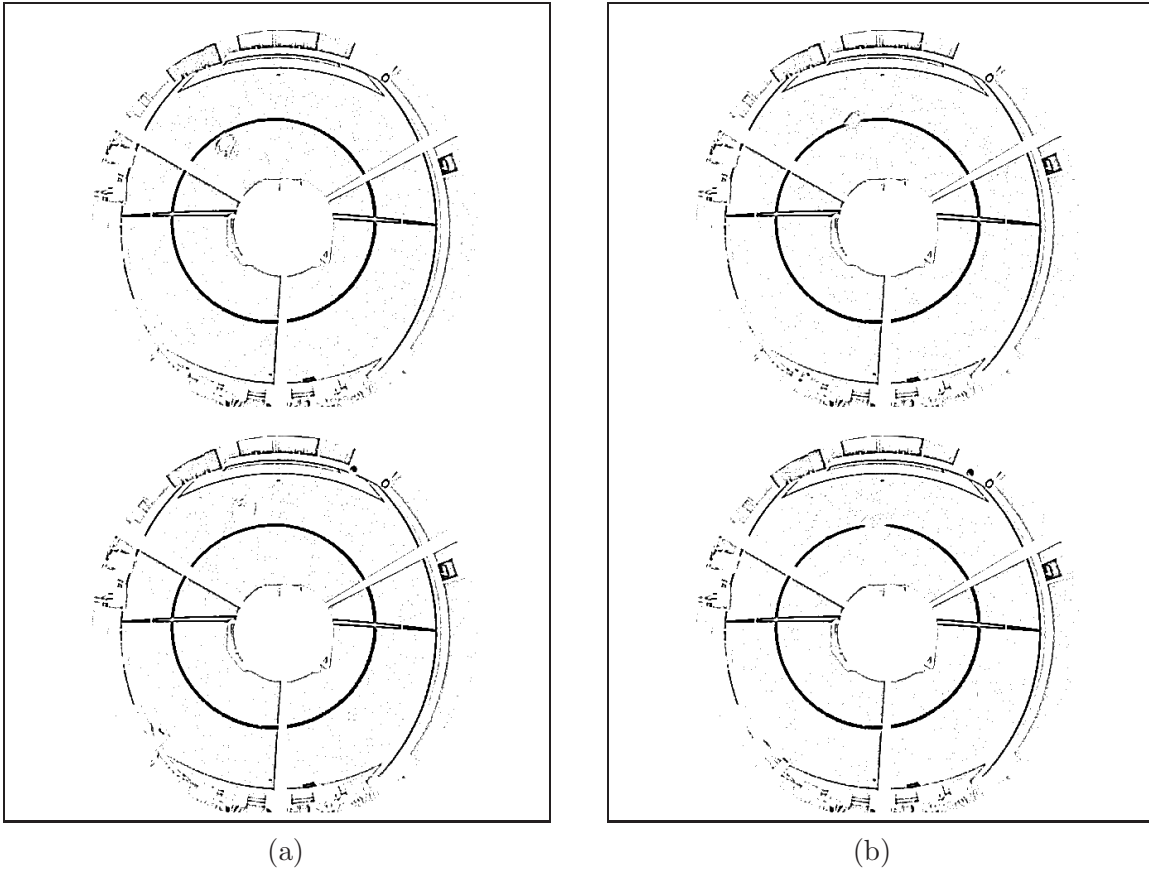


Figure 4.4: Two examples previously presented in Fig. 4.2, now with the Laplace operator applied. In (a) a test image with the ball standing still and in (b) with the ball moving at high speed. The upper images refer to the white ball, and the bottom images to the orange ball.

the real-time capability must be assured with low processing times. In other hand, the algorithm must be able to detect the edges of the ball independently of its luminance contrast with the ground, or its motion blur effect due to the speed. In this test, the use of two balls, white and orange, allowed to get a first approach to the edge detector sensitivity to luminance variations.

Note that in Figs. 4.3, 4.4 and 4.5, the edges are represented by the black pixels. In Fig. 4.3 it is possible to see the bad results provided by the Sobel edge detector. The edges are hard to see and the ball is almost invisible. In Fig. 4.4, the results improved a little. With this edge detector, the white ball can be distinguished when standing still, or at low velocities, but when using an orange ball or the ball is moving a little faster, this edge detector delivers bad results. In Fig. 4.5, the ball can be seen perfectly when standing still, independently of its color. When moving, the ball can also be pointed out, but not as clearly as when it

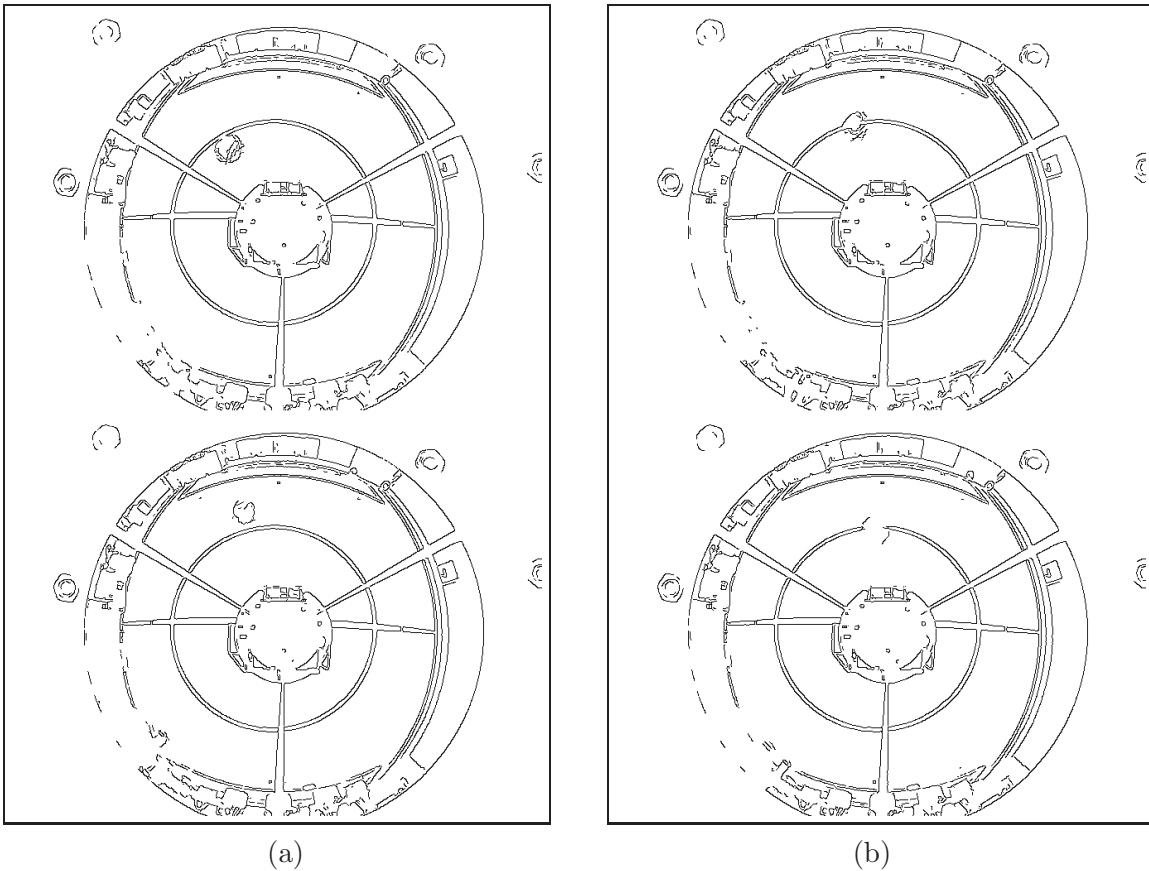


Figure 4.5: Two examples previously presented in Fig. 4.2, now with the Canny operator applied. In (a) a test image with the ball standing still and in (b) with the ball moving at high speed. The upper images refer to the white ball, and the bottom images to the orange ball.

is standing still. Comparing Fig. 4.3 with Fig. 4.4 and Fig. 4.5, it is notorious the superior results provided by the Canny edge detector.

During these tests, the processing time of each algorithm was measured, being their mean time presented in Table 4.1. A first look at the mean time spent for each edge detector (see Table 4.1) may point out the Canny as the worst choice. In fact, the Canny processing time is the highest between the ones tested but, even so, it is fast enough to be used in real-time applications, and the resulting edge images show the effectiveness of this edge detector, way above the others.

Since the Canny filter was developed to be an “optimal edge detector”, it was expected from the beginning its supremacy in the results, being finally confirmed.

Edge detector	Sobel	Laplace	Canny
Average processing time	5.17	4.11	10.59

Table 4.1: Processing time obtained from the three tested edge detection algorithms. All times are in milliseconds.

4.3 Hough transform

The *Hough* transform is a technique widely used to find instances of objects, of a certain class of shapes (lines, circles or even ellipses) by a voting procedure [34, 35, 36, 29]. Note that finding the object is different from validating the object and with this method the image is only searched for points of interest. This voting procedure is carried out in a parameter space, from which object candidates are obtained as local maxima in a so-called *Intensity Map Image* (see Fig. 2.2) that is explicitly constructed by the algorithm for computing the *Hough* transform. Follows a detailed description of the algorithm implementation and optimization for real-time purposes, and the results.

4.3.1 Implementation

In Fig. 4.6, it is shown an example of the algorithm used, where the dark continuous lines represent the edges found in the *Edges Image* and the dashed lines are the result from the *Hough* transform over discrete spots of the continuous lines. Due to the *Hough* circular transform especial features, a big round object in the *Edges Image* would produce in the *Intensity Map Image* a very intense peak in the center of the object, as shown on the left side of Fig. 4.6. For contrast, a non-round object would produce areas of low intensity in the *Intensity Map Image*, as represented in the right side of Fig. 4.6.

This approach rises the following problem: as the ball moves away, its edge circle size scales down in the image. To solve this problem, information about the distance between the robot center and the ball is used to adjust the *Hough* transform.

As this algorithm requires drawing circles (in the *Intensity Map Image*) centered in the pixel actually being processed, it was developed a tool to accelerate this operation. Based in predefined circle sizes and in the image dimensions, an array of offsets is created in memory for each circle size. When added with the index of the pixel where the circle has to be centered, this array of offsets will directly map the circle in the pixels of the image, reducing drastically the processing time commonly needed for circle creation.

To improve the robustness of this algorithm a little further, some extra details are taken

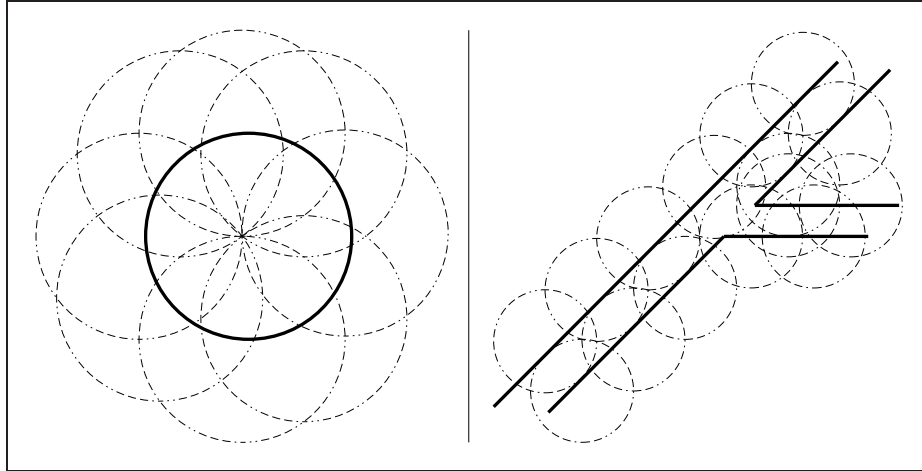


Figure 4.6: *Hough* transform example. On the left, edges from the ball. On the right, edges from the field white lines.

into account. To attain a clean image, the edges created by the robot reflection in the mirror are removed using information from the *Mask Image*. Furthermore, although the FIFA rules do not have restrictions for the ball color (see <http://www.fifa.com>), it is considered that the ball is never green. This last assumption reduces the potential risk of false positives in the middle of the field due to other robots over white lines, and crossing white lines. To do so, information from the *Labels Image* is compared against the current pixel being processed. And finally, because distant balls are very irregular in the *Edges Image*, due to its reduced size, distant edges are discarded in the *Hough* transform to avoid erroneous false points of interest.

In Figs. 4.7, 4.8 and 4.9, we can see an example of the *Morphological Processing Sub-System* pipeline, presented in Fig. 2.2. As can be observed, the ball in the *Edges Image* (Fig. 4.8), is not perfectly circular. Even so, as the *Hough* transform is very tolerant to gaps in feature boundary descriptions and is relatively unaffected by image noise [30, 37, 38, 39], it still performs well, as can be seen in Fig. 4.7. Notice as the center of the ball presents a very high peak when compared to the rest of the image in Fig. 4.9

4.4 Results

Since this is a system in an early stage of development, the results shown in Fig. 4.10 and in the Table 4.2 are very encouraging. In Table 4.2, it is not possible to include information about the detection ratio of the ball, as this system still lacks validation, resulting in always detecting a ball (even without any ball in the image). The average field in Table 4.2 shows the



Figure 4.7: Example of a captured image using the morphological ball detection system. The cross over the ball points out the detected position.

good localization results and the standard deviation that, even presenting high values, must be considered keeping in mind that this system does not include any validation, i.e., it always points out the highest value in the *Intensity Map Image*. Notice the average processing time of 25 ms, below the upper limit of 33 ms necessary for a real-time system running at 30 Hz.

Experiment	Real Position	Measures		
		Average	Std	Processing time
MidField	(0.00, 0.00)	(-0.27, 0.25)	(1.33, 1.14)	25 ms

Table 4.2: Some measures obtained for the experiments presented in Fig. 4.10. All the measures are in meters, except for *Processing time* presented in milliseconds.

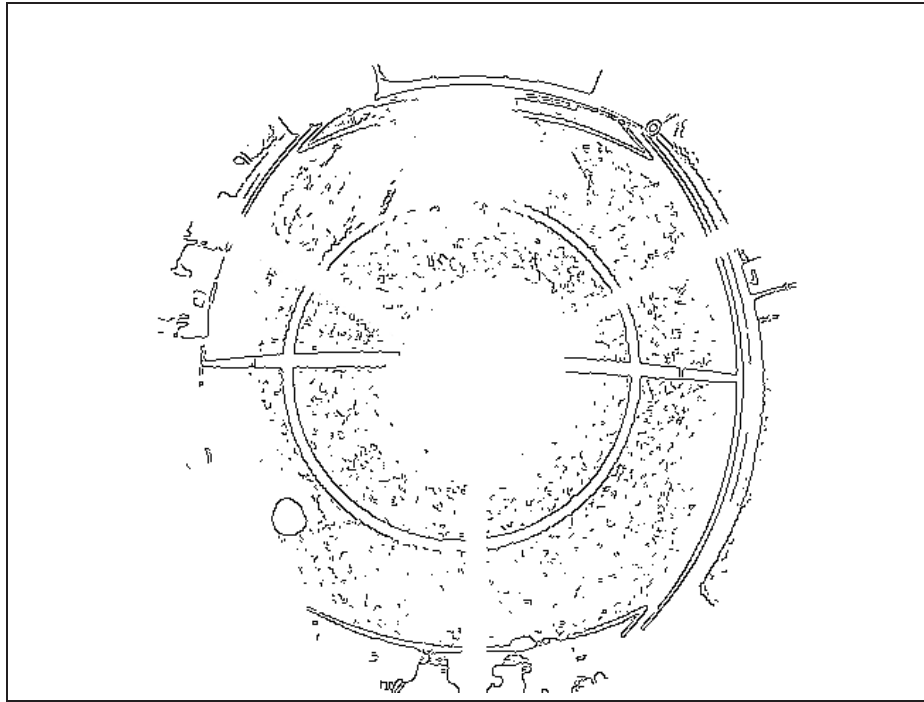


Figure 4.8: Example previously presented in Fig. 4.7, now with the Canny edge detector applied. Notice as the robot reflection was avoided from the *Edges Image*.

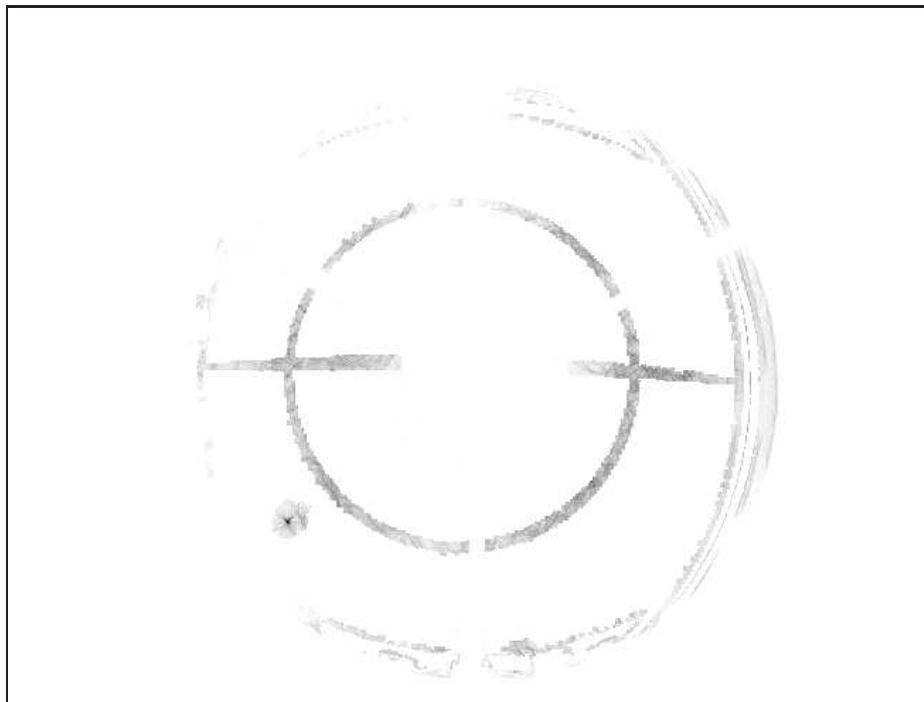


Figure 4.9: Example previously presented in Fig. 4.8, now with the *Hough* transform applied. Notice that far edges are not processed and results over green pixels are not considered.

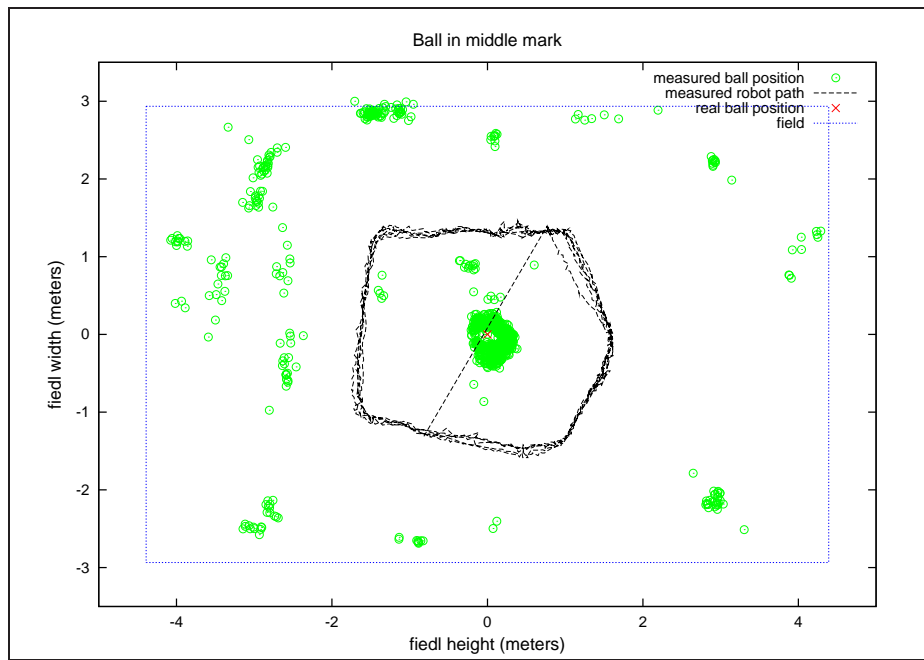


Figure 4.10: Experimental results obtained by the omnidirectional sub-system using the morphological ball detection. In this experience the ball was positioned in the center of the field. The robot has performed a predefined trajectory while the position of the ball was registered. Both axis in the graphics are in meters.

Chapter 5

Developed Tools

5.1 The PerspectiveMapCalib

The use of a distance map image is a method commonly used in the RoboCup domain. This method assumes that all the objects found in the image are in one plane (the ground plane), allowing to map the distances in the image with only one camera. To simplify the problem, it was assumed that the lens and the CCD from the camera were centered and in parallel planes, so the center of the image would be the center of the CCD. Two more approximations were made during this approach to the problem. First, the θ rotation was ignored, so the camera is considered to be pointing towards the front of the robot and second, the φ rotation of the camera was also ignored, considering it horizontally aligned (see Fig. 5.1).

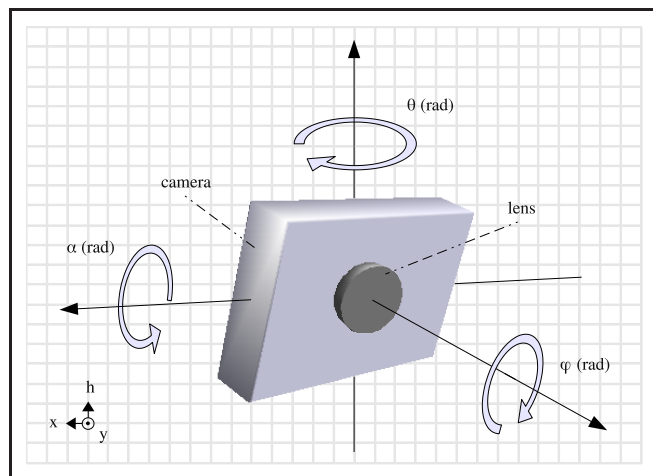


Figure 5.1: Schematic representation of the perspective camera with its rotation axis.

To summarize, the mathematical model of this sub-system needs information about the

camera, the robot and the field. Some of this information, such as the h_{offset} and r_{offset} represented in Fig. 5.2, can be measured directly with good precision. Other kinds of information, like the *pixel height*, *pixel width* and the *focal length* represented in Fig. 5.4, may be obtained from the camera data-sheet, given by the manufacturers. However, it is still missing necessary information, namely the α_{offset} (see Fig. 5.2). This information can be measured, but, in one hand, measures like the h_{offset} and r_{offset} may be used from one robot to another without compromising the resulting map image and, on the other hand, distance map results are very sensible to α_{offset} variations and it is very time consuming to obtain good measures of this variable. One way to overcome this time consuming process (of measuring the α_{offset} in each robot) is to include an automatic measuring process of this parameter into the *PerspectiveMapCalib* tool.

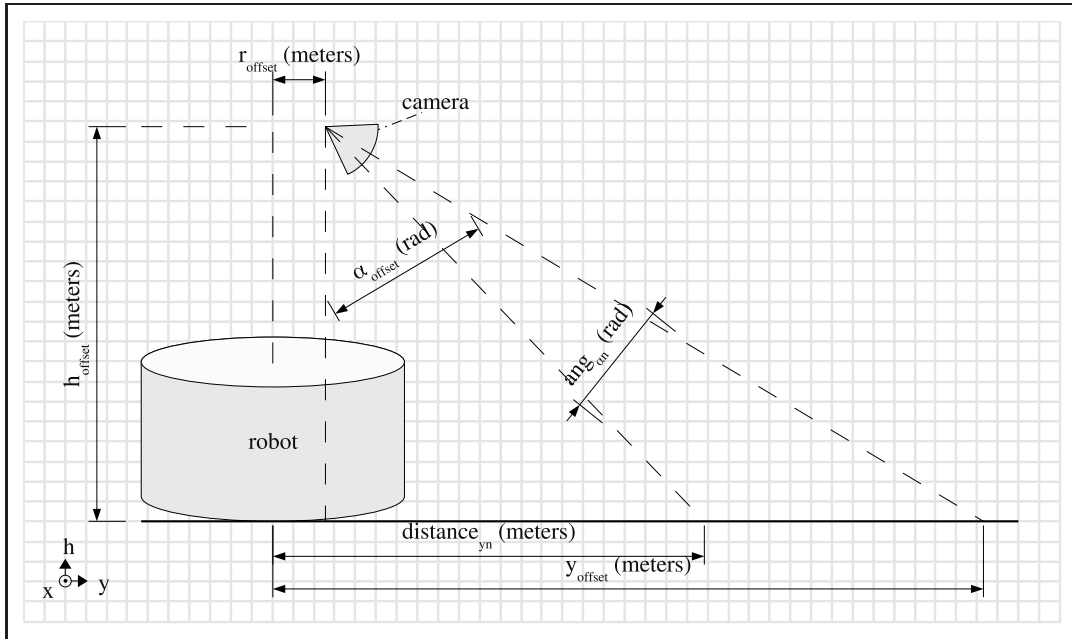


Figure 5.2: Robot and its perspective sub-system schematic side view.

In Fig. 5.2 and Fig. 5.3 it is presented a side and top schematic view of the perspective vision sub-system. A schematic view of a detail over the perspective sub-system is presented in Fig. 5.4. Follows a short explanation of the measures shown in these figures:

- h_{offset} - distance from the camera to the ground;
- r_{offset} - radial distance from the camera to the robot center;
- α_{offset} - angular offset of the camera along α axis;

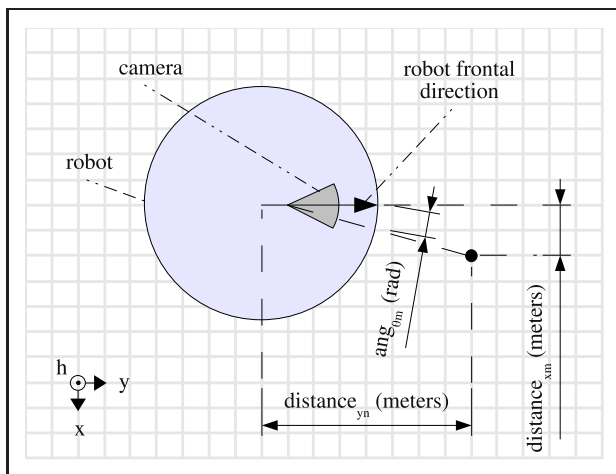


Figure 5.3: Schematic top-down view of the robot and perspective sub-system.

- y_{offset} - distance from the center of the robot to the point in the center of the image projected on the ground;
- ang_{α_n} - angle measured from the α_{offset} along α axis, relative to $pixel_n$;
- ang_{θ_m} - angle measured from the robot front along θ axis, relative to $pixel_m$;
- $distance_{y_n}$ - distance from the center of the robot to the $pixel_n$, projected on the ground;
- $distance_{x_m}$ - distance from the center of the robot to the $pixel_n$, projected on the ground;
- *focal length* - distance between lens and CCD;
- $pixel_n$ - number of pixels (n) along a CCD column;
- $pixel_m$ - number of pixels (m) along a CCD row.

Using an image taken from a known position (over the goal line, aligned with the kick-off mark, as shown in Fig. 5.5), the tool acquires some samples based on image analysis and some user input data. The tool highlights the white lines found, asking the user for the $distance_{y_n}$ between the robot center and the white line being processed. The search for white lines is conducted over the central row of the image, because the θ_{offset} is being ignored.

The following equation shows the relation between the y_{offset} and the angle α_{offset} , both referred in Fig. 5.2,

$$\alpha_{\text{offset}} = \arctan \left(\frac{y_{\text{offset}} - r_{\text{offset}}}{h_{\text{offset}}} \right). \quad (5.1)$$

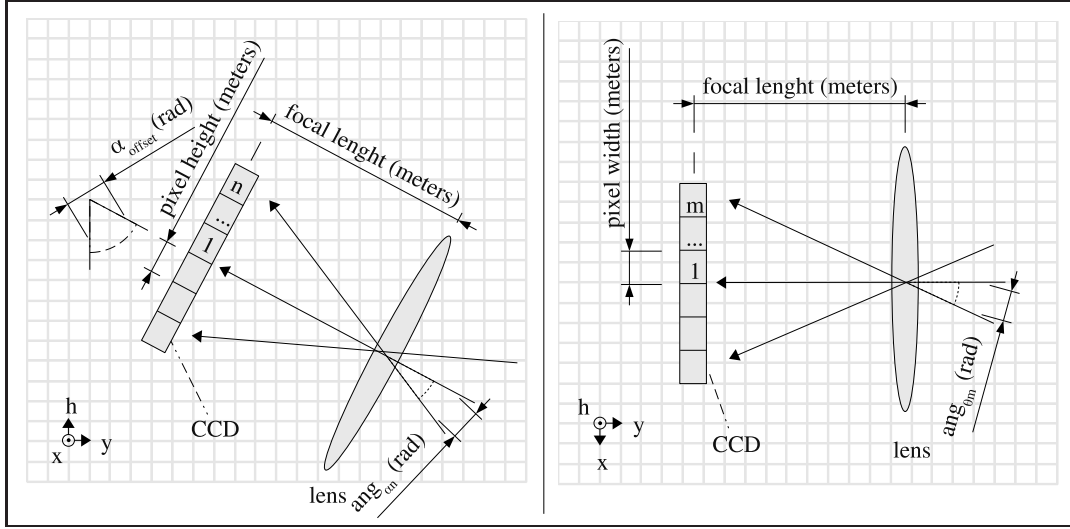


Figure 5.4: Schematic view of a detail over the lens, CCD and focus point from the perspective sub-system. On the left, a side view. On the right, a top-down view.



Figure 5.5: Example of an image processed by the tool *PerspectiveMapCalib*. Magenta spots in the image show the white lines found. Orange circles highlight the spot being processed.

From (5.1), the relation is generalized to an angle ang_{on} centered in α_{offset} ,

$$ang_{on} = \alpha_{offset} - \arctan\left(\frac{distance_{yn} - r_{offset}}{h_{offset}}\right). \quad (5.2)$$

Attending to Fig. 5.4 (on the left), using simple trigonometric rules the following equation can be obtained, relating a generic angle ang_n centered in α_{offset} and a pixel along a vertical column in the CCD,

$$pixel_n = \frac{\tan(ang_{\alpha n}) \times focal\ length}{pixel\ height}. \quad (5.3)$$

Substituting (5.2) in (5.3) results in,

$$pixel_n = \tan \left[\alpha_{\text{offset}} - \arctan \left(\frac{distance_{yn} - r_{\text{offset}}}{h_{\text{offset}}} \right) \right] \times \frac{focal\ length}{pixel\ height}. \quad (5.4)$$

Manipulating (5.4), it is possible to isolate the α_{offset} in the first member, i.e.,

$$\alpha_{\text{offset}} = \arctan \left(\frac{pixel_n \times pixel\ height}{focal\ length} \right) + \arctan \left(\frac{distance_{yn} - r_{\text{offset}}}{h_{\text{offset}}} \right). \quad (5.5)$$

With (5.5) and the samples (pairs of values $pixel_n$ and $distance_{yn}$ introduced by the user) previously acquired, the value α_{offset} can be found. To achieve a better result, the tool *PerspectiveMapCalib* uses a mean filter with, at least, three calculated α_{offset} angles to obtain the final result.

Using the previously discovered α_{offset} angle and with some manipulation of (5.4), the distance corresponding to each pixel can be found using

$$distance_{yn} = r_{\text{offset}} + h_{\text{offset}} \times \tan \left[\alpha_{\text{offset}} - \arctan \left(\frac{pixel_n \times pixel\ height}{focal\ length} \right) \right]. \quad (5.6)$$

Through (5.6) it is possible to obtain the real $distance_{yn}$ of a pixel projected in the ground along the y axis.

Based on Fig. 5.3, the following equation can be deduced, creating a relation between a generic angle $ang_{\theta m}$ and a $distance_{xm}$,

$$ang_{\theta m} = \arctan \left(\frac{distance_{xm}}{distance_{yn}} \right). \quad (5.7)$$

Now, relating $ang_{\theta m}$ with $pixel_m$ along the x axis we obtain

$$pixel_m = \frac{\tan(ang_{\theta m}) \times focal\ length}{pixel\ width}. \quad (5.8)$$

Using (5.7) to substitute in (5.8) and manipulating it, results in a relation between a $pixel_m$ and a $distance_{xm}$, both along the x axis, i.e.,

$$distance_{xm} = \arctan \left(\frac{pixel_m \times pixel\ width}{focal\ length} \right) \times distance_{yn}. \quad (5.9)$$

Summarizing, making use of (5.5), and some user input, its possible to obtain the α_{offset} . This permits the creation of the *distance map image* using (5.9) and (5.6) providing the x and y coordinates associated to each $pixel_{(n,m)}$.

5.1.1 Results

To measure the reliability of this tool, tests were made analyzing the resulting *distance map image*. To do so, the robot was moved along a predefined path through the game field leaving the ball in a known location. The ball position given by the robot is then compared with the real position of the ball. The results in this test may be affected by the errors of the localization algorithm and the robot bumps while moving, but they provide a realistic view of the problem, recreating the RoboCup environment.

In Fig. 5.6 it is possible to see the robot path along the field and the measured ball position. In Table 5.1, it is presented an analysis of the data measured during the test. Already visible in Fig 5.6, the average position of the ball is near the real position, being this result confirmed in Table 5.1. Furthermore, the standard deviation of the measured ball position is low compared with the distance between the ball and the robot. Being these measures already affected as referred above, this result is more than acceptable for its purpose, that is, to detect the ball at long distances. Also, note the detection ratio surrounding the 95.8% and 91.3%, both very high. Notice the average processing time of 8 ms, below the upper limit of 33 ms necessary for a real-time system running at 30 Hz.

Experiment	Real Position	Measures			
		Average	Std	Detection ratio	Processing time
Penalty	(2.39, 0.00)	(2.47, 0.07)	(0.19, 0.09)	95.8%	8 ms
Goal Line	(4.39, 0.0)	(4.27, -0.03)	(0.32, 0.11)	91.3%	8 ms

Table 5.1: Some measures obtained for the experiments presented in Fig. 5.6. All the measures are in meters, except for *Detection ratio* that shows the percentage of samples collected where the ball was detected and *Processing time* presented in milliseconds.

5.2 ImageHolder class

As described in [16], it is possible to acquire images from a digital camera in various formats. The direct manipulation of various image formats can lead to programing mistakes, hard coded solutions and unreadable code. Creating an abstraction layer above the image

raw data provides directives to operate over the image without worrying with its mode or dimensions.

The ImageHolder class was created by the author to solve the described problems, keeping in mind its real-time purpose. All this was achieved and all the hybrid vision system software was reviewed to use this library.

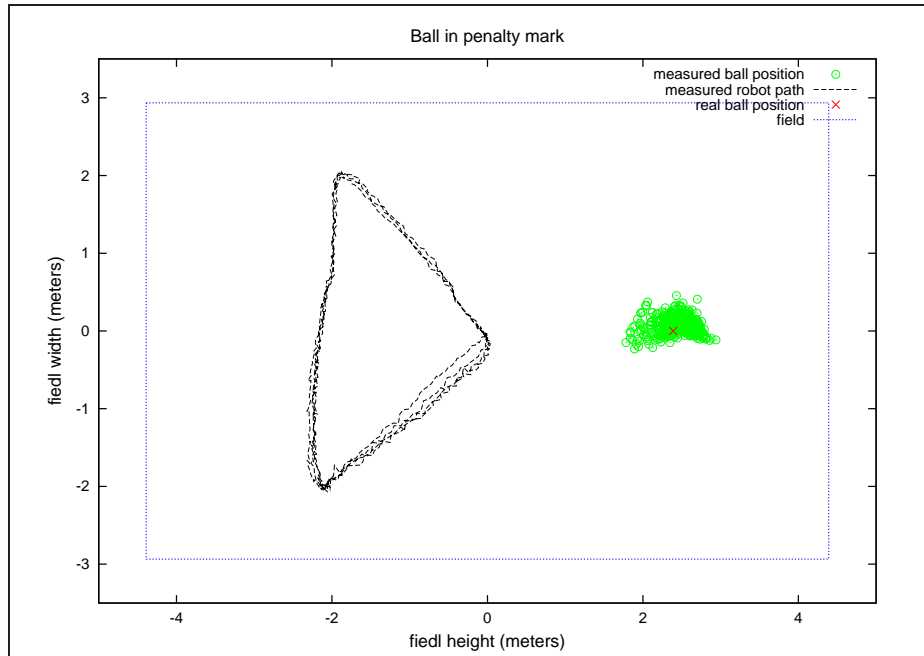
The efficiency of this library is assured using *inline functions*, a feature of *C++* programming language that allows processes speed up. The use of inline functions, eliminates the function call overhead, reducing the time spent in intense repetitive tasks. Further explanations can be found in [40].

Due to the diverse image modes used by the hybrid vision system, this library was implemented with support for six image modes. The structure of the library was designed in order to easily add image modes in the future. In the current implementation, the image modes supported are described next.

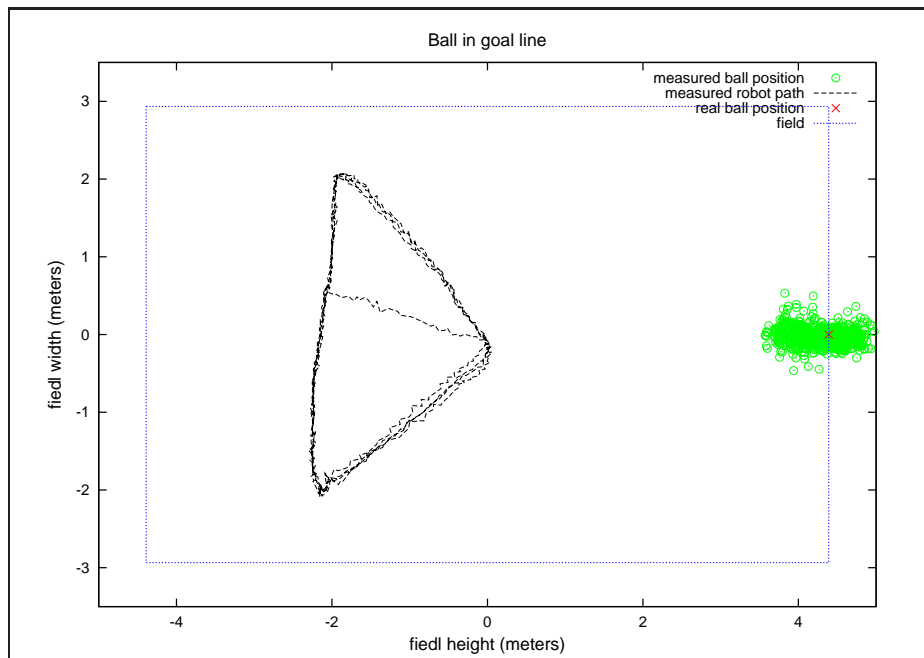
- UNKNOWN - Not a real image mode, it is only used as a signal that the image mode is not known;
- SEG - Segmented mode. The image is represented by one plane, 8 bits per pixel, each bit representing a color class;
- GRAY - Gray-scale mode. The image is represented by one plane, 8 bits per pixel;
- RGB - Red, Green and Blue mode. The image is represented in three planes, 8 bits per pixel per plane, totaling 24 bits per pixel;
- HSV - Hue saturation value mode. The image is represented in three planes, 8 bits per pixel per plane, totaling 24 bits per pixel;
- YUV422 - Luma and chrominance mode with 422 subsampling. The image is represented in three planes, where U and V components (planes) are subsampled shared with a ratio of 2 samples for 4 pixels.
- YUV411 - Luma and chrominance mode with 411 subsampling. The image is represented in three planes, where U and V components (planes) are subsampled with a ratio of 1 sample for 4 pixels.

Besides holding image data, this library has the ability to read and write images into files with the possibility of cropping the image. It is also capable of displaying the held image into the screen over SDL library (see <http://www.libsdl.org> for further info). As drawing

capabilities, the *ImageHolder* class permits to draw three geometric forms over the image, squares, crosses and circles variable in size, position and color. The information about the color of a pixel can be retrieved independently of the image mode, directly in the user preferred mode, RGB, HSV or YUV. The direct access to the image raw data is also possible, being this a necessary feature to the implementation of many complex image analysis algorithms. A brief description of the class can be found in appendix A.



(a)



(b)

Figure 5.6: Experimental results obtained by the proposed perspective vision system using the *PerspectiveMapCalib* tool. In (a) the ball was positioned in the center of the penalty mark. In (b) the ball was positioned in the center of the goal line. The robot has performed a defined trajectory and the position of the ball was registered. Both axis in the graphics are in meters.

Chapter 6

Conclusions and future work

In this work, we addressed the field of robotic vision, including real-time applications, using the example of the CAMBADA hybrid vision system.

The CAMBADA hybrid vision system was extensively described along with some new improvements proposed and implemented by the author. The already functional *Color Processing Sub-System* became much more robust and accurate with the *Shadowed ball recover* algorithm and the *DistanceVsPixel* validation algorithm. Pushing the CAMBADA vision system a little further, a morphological detection system was also developed by the author, allowing to detect the ball without previously knowing its color. The *Morphological Processing Sub-System*, the most recent development in the CAMBADA hybrid vision system and still under development, is already functional and proved its effectiveness.

Besides the referred improvements, some tools were also developed by the author. To accelerate the process of calibrating the robots was created the *PerspectiveMapCalib*, a tool to help creating the distance map images. This tool allows a non-specialized user to calibrate the distance map image of the perspective sub-system with trivial commands. The author also developed the *ImageHolder* library, created to be a real-time library to manage images in various modes.

The results shown in the Portuguese Robotics Open 2008, where the team placed 1st, are an additional prove of the effectiveness of the proposed and accomplished work.

6.1 Future work

As future work, it is proposed to continue the studies in the *Morphological Processing Sub-System*, in order to become a fully functional processing system. We propose the use of

a validation system over the points of interest represented in the *Intensity Map Image* (see Fig. 2.2). This validation could be done using the perceptron algorithm or an Haar-classifier [41, 42] (a cascade of boosted classifiers working with Haar-like features).

Reorganizing the entire hybrid vision system to become fully independent of the image size is also an important step which would improve the system robustness and processing speed. This last proposal would also allow to use Format7 image acquisition, permitting the acquisition of images controlling the desired frame size and position directly on the camera's CCD.

The *PerspectiveMapCalib* tool can also be improved to correct misaligned angles currently being discarded (considered as zero). This would create even better perspective distance map images.

The ImageHolder library could be expanded to allow the use of more image formats and include members to convert images from one format to another.

Future work could also take a step in the spatial-3D ball detection direction, crossing both cameras information. This would extract information about the 3D position of the ball, instead of assuming that the ball never leaves the ground plane.

Appendix A

ImageHolder

Class to hold and manipulate images.

```
#include <ImageHolder.h>
```

Public Member Functions

- **ImageHolder** ()
Default constructor.
- **ImageHolder** (**ImageHolder** &ImH)
Copy constructor.
- **ImageHolder** (enum **iMode** Mode, unsigned Rows, unsigned Cols, unsigned char *ImgBuf=NULL)
Initialize constructor.
- **~ImageHolder** ()
Destructor.
- unsigned char & **operator**[] (const unsigned index)
Overload to operator '['].
- unsigned char & **operator**[] (const unsigned index) const
Overload to operator '['].

- void **clearImage** (void)
*Clears the **ImageHolder** object data.*
- void **copyImage** (const unsigned char *ImgBuf)
*Copy image to **ImageHolder** object.*
- int **convertImage** (const **ImageHolder** &IH)
*Convert image to **ImageHolder** object's mode.*
- int **setImage** (const unsigned char *ImgBuf)
*Assign image buffer to **ImageHolder** object.*
- int **displayImage** (SDL_Surface *Surface, SDL_Overlay *Overlay, SDL_Rect *Rect)
Displays contained image through SDL functionalities.
- int **drawSquare** (unsigned Row, unsigned Col, **ColorID** Color, unsigned Size=5)
Draws a square in the image.
- int **drawSquare** (unsigned Row, unsigned Col, struct **Rgb** Value=**Rgb**(255, 100, 100), unsigned Size=5)
Draws a square in the image.
- int **drawCross** (unsigned Row, unsigned Col, **ColorID** Color, unsigned Size=5)
Draws a cross in the image.
- int **drawCross** (unsigned Row, unsigned Col, struct **Rgb** Value=**Rgb**(255, 100, 100), unsigned Size=5)
Draws a cross in the image.
- int **drawCircle** (unsigned Row, unsigned Col, **ColorID** Color, unsigned Size=5)
Draws a circle in the image.
- int **drawCircle** (unsigned Row, unsigned Col, struct **Rgb** Value=**Rgb**(255, 100, 100), unsigned Size=5)
Draws a circle in the image.
- unsigned char * **image** () const

Get pointer to image data.

- **iMode mode** () const

Get image mode.

- unsigned **size** () const

Get image size.

- unsigned **rows** () const

Get number of rows.

- unsigned **cols** () const

Get number of columns.

- int **pixel** (unsigned Row, unsigned Col, struct **Hsv** &Pixel) const

Get complete pixel info.

- int **pixel** (unsigned Row, unsigned Col, struct **Yuv** &Pixel) const

Get complete pixel info.

- int **pixel** (unsigned Row, unsigned Col, struct **Rgb** &Pixel) const

Get complete pixel info.

- unsigned **pixelY** (unsigned Row, unsigned Col) const

Get Y component of pixel.

- int **saveRectangle** (unsigned UpRow, unsigned LeftCol, unsigned DownRow, unsigned RightCol, const char *FileName) const

Saves a rectangle from the image to a file.

- int **saveSquare** (unsigned Row, unsigned Col, unsigned Size, const char *FileName) const

Saves a square from the image to a file.

- int **saveFrame** (const char *FileName) const

Saves all the image to a file.

Private Member Functions

- int **getPixel** (unsigned Row, unsigned Col, unsigned &c1, unsigned &c2, unsigned &c3)
const

Used to help pixel(...) function. This is an image mode independent function.

- int **setPixel** (unsigned Row, unsigned Col, unsigned c1=0, unsigned c2=0, unsigned c3=0)

Used to help many functions.

Static Private Member Functions

- static void **Rgb2Hsv** (struct **Rgb**, struct **Hsv** &)

Used to help pixel(...) function.

- static void **Rgb2Yuv** (struct **Rgb**, struct **Yuv** &)

Used to help pixel(...) function.

- static void **Yuv2Rgb** (struct **Yuv**, struct **Rgb** &)

Used to help pixel(...) function.

Private Attributes

- unsigned char * **idata**

Pointer to image data.

- unsigned **irows**

Number of rows in image.

- unsigned **icols**

Number of columns in image.

- unsigned **isize**

Size of image, in bytes.

- **iMode imode**

Image mode.

- bool **idata_priv**

Describes if image data was allocated by this object itself.

A.1 Detailed Description

Class to hold and manipulate images.

This class can work images based on their **type**, number of **rows** and number of **columns**.

Warning: Restrictions:

- Maximum resolution (UINT_MAX, UINT_MAX).
- Image formats {SEG, GRAY, RGB, HSV and YUV}.
- Pixel component magnitude [0, UCHAR_MAX].

A.2 Constructor & Destructor Documentation

ImageHolder::ImageHolder ()

Default constructor.

Initializes an **ImageHolder** object with **size** 0, and **mode** UNKNOWN.

References icols, idata, idata_priv, imode, irows, and isize.

ImageHolder::ImageHolder (ImageHolder & ImH)

Copy constructor.

Initializes an **ImageHolder** object based on another previously declared.

References icols, idata, idata_priv, imode, irows, and isize.

ImageHolder::ImageHolder (enum iMode *Mode*, unsigned *Rows*, unsigned *Cols*, unsigned char * *ImgBuf* = NULL)

Initialize constructor.

Initializes an **ImageHolder** object with info passed by arguments.

Parameters: *Mode* Desired image mode.

Rows Desired image rows.

Cols Desired image columns.

ImgBuf Pointer to image buffer to use. If NULL an empty buffer will be created.

References *icols*, *idata*, *idata_priv*, *imode*, *irows*, and *isize*.

ImageHolder::~ImageHolder ()

Destructor.

Free all buffers previously created by the own object.

References *idata*, and *idata_priv*.

A.3 Member Function Documentation

int ImageHolder::getPixel (unsigned *Row*, unsigned *Col*, unsigned & *c1*, unsigned & *c2*, unsigned & *c3*) const [private]

Used to help pixel(...) function. This is an image mode independent function.

Parameters: *Row* Indicates the row to access.

Col Indicates the column to access.

c1 The first image component.

c2 The second image component.

c3 The third image component.

References *icols*, *idata*, *imode*, and *irows*.

Referenced by *pixel()*, and *pixelY()*.

int ImageHolder::setPixel (unsigned *Row*, unsigned *Col*, unsigned *c1* = 0, unsigned *c2* = 0, unsigned *c3* = 0) [private]

Used to help many functions.

Parameters: *Row* Indicates the row to access.

Col Indicates the column to access.

c1 The first color component.

c2 The second color component.

c3 The third color component.

References *icols*, *idata*, *imode*, and *irows*.

Referenced by `drawCircle()`, `drawCross()`, and `drawSquare()`.

unsigned char & ImageHolder::operator[] (const unsigned *index*) [inline]

Overload to operator `'[]'`.

Overloads the operator `'[]'` so it can be used with **ImageHolder** object. This makes possible a direct access to the image data array.

Parameters: *index* Image data array index.

Returns: Reference to the image data array pointed by the index passed by argument.

References *idata*.

unsigned char & ImageHolder::operator[] (const unsigned *index*) const [inline]

Overload to operator `'[]'`.

Overloads the operator `'[]'` so it can be used with **ImageHolder** object. This makes possible a direct access to the image data array.

Parameters: *index* Image data array index.

Returns: Reference to the image data array pointed by the index passed by argument.

References *idata*.

void ImageHolder::copyImage (const unsigned char * *ImgBuf*)

Copy image to **ImageHolder** object.

Copies the image from the argument to the **ImageHolder** object.

Parameters: *ImgBuf* Pointer to the image to be copied.

Warning: This function doesn't check for buffer length. Use with precaution.

References `idata`, and `isize`.

Referenced by `convertImage()`.

int ImageHolder::convertImage (const ImageHolder & *IH*)

Convert image to **ImageHolder** object's mode.

Converts the data from the argument to the **ImageHolder** object.

Parameters: ***IH*** **ImageHolder** to be converted.

References `copyImage()`, `icols`, `idata`, `image()`, `imode`, `irows`, `pixel()`, and `Yuv::y`.

int ImageHolder::setImage (const unsigned char * *ImgBuf*)

Assign image buffer to **ImageHolder** object.

Assigns the image buffer passed on by argument to the **ImageHolder** object.

Parameters: ***ImgBuf*** Pointer to the image buffer to be assigned.

References `idata`, and `idata_priv`.

int ImageHolder::displayImage (SDL_Surface * *Surface*, SDL_Overlay * *Overlay*, SDL_Rect * *Rect*)

Displays contained image through SDL functionalities.

Parameters: ***Surface*** Pointer to `SDL_Surface` to be used to display the current image.

Overlay Pointer to `SDL_Overlay` to be used to display the current image.

Rect Pointer to `SDL_Rect` to be used to display the current image.

References `icols`, `idata`, `imode`, `irows`, `Yuv::u`, and `Yuv::v`.

int ImageHolder::drawSquare (unsigned *Row*, unsigned *Col*, ColorID *Color* = CBLUE, unsigned *Size* = 5)

Draws a square in the image.

Given the parameters draws a square in the image holded by the **ImageHolder** object.

Warning: Use this function to draw squares in segmented images.

Parameters: **Row** Center of the square (row coordinate).

Col Center of the square (column coordinate).

Color Color of the square previously defined in ColorID.

Size Square size in pixels.

References imode.

```
int ImageHolder::drawSquare (unsigned Row, unsigned Col, struct Rgb Value  
= Rgb(255,100,100), unsigned Size = 5)
```

Draws a square in the image.

Given the parameters draws a square in the image held by the **ImageHolder** object.

Warning: Do not use this function to draw squares in segmented images. For that purpose use the "drawSquare(int, int, ColorID, int)".

Parameters: **Row** Center of the square (row coordinate).

Col Center of the square (column coordinate).

Value Color of the square in rgb. This parameter can be passed like '*Rgb(rrr, ggg, bbb)*'.

Size Square size in pixels.

References Rgb::b, Rgb::g, icols, imode, irows, Rgb::r, Rgb2Yuv(), setPixel(), Yuv::u, Yuv::v, and Yuv::y.

```
int ImageHolder::drawCross (unsigned Row, unsigned Col, ColorID Color, un-  
signed Size = 5)
```

Draws a cross in the image.

Given the parameters draws a cross in the image held by the **ImageHolder** object.

Warning: Use this function to draw crosses in segmented images.

Parameters: **Row** Center of the cross (row coordinate).

Col Center of the cross (column coordinate).

Color Color of the cross previously defined in ColorID.

Size Cross size in pixels.

References imode.

```
int ImageHolder::drawCross (unsigned Row, unsigned Col, struct Rgb Value =  
Rgb(255,100,100), unsigned Size = 5)
```

Draws a cross in the image.

Given the parameters draws a cross in the image holded by the **ImageHolder** object.

Warning: Do not use this function to draw crosses in segmented images. For that purpose use the "drawCross(int, int, ColorID, int)".

Parameters: *Row* Center of the cross (row coordinate).

Col Center of the cross (column coordinate).

Value Color of the cross in rgb. This parameter can be passed like '*Rgb(rrr, ggg, bbb)*'.

Size Cross size in pixels.

References Rgb::b, Rgb::g, icols, imode, irows, Rgb::r, Rgb2Yuv(), setPixel(), Yuv::u, Yuv::v, and Yuv::y.

```
int ImageHolder::drawCircle (unsigned Row, unsigned Col, ColorID Color, un-  
signed Size = 5)
```

Draws a circle in the image.

Given the parameters draws a circle in the image holded by the **ImageHolder** object.

Warning: Use this function to draw circles in segmented images.

Parameters: *Row* Center of the circle (row coordinate).

Col Center of the circle (column coordinate).

Color Color of the circle previously defined in ColorID.

Size Cross size in pixels.

References imode.

int ImageHolder::drawCircle (unsigned *Row*, unsigned *Col*, struct **Rgb** *Value* = **Rgb**(255,100,100), unsigned *Size* = 5)

Draws a circle in the image.

Given the parameters draws a circle in the image holded by the **ImageHolder** object.

Warning: Do not use this function to draw circles in segmented images. For that purpose use the "drawCircle(int, int, ColorID, int)".

Parameters: **Row** Center of the circle (row coordinate).

Col Center of the circle (column coordinate).

Value Color of the circle in rgb. This parameter can be passed like '*Rgb*(*rrr*, *ggg*, *bbb*)'.

Size Circle size in pixels.

References **Rgb::b**, **Rgb::g**, **icols**, **imode**, **irows**, **Rgb::r**, **Rgb2Yuv()**, **setPixel()**, **Yuv::u**, **Yuv::v**, and **Yuv::y**.

unsigned char * ImageHolder::image () const [inline]

Get pointer to image data.

Access to the image data address.

Returns: Pointer to the image data buffer contained in the **ImageHolder** object.

References **idata**.

Referenced by **convertImage()**.

iMode ImageHolder::mode () const [inline]

Get image mode.

Access to image mode info.

Returns: Image **mode** used by the **ImageHolderp.classImageHolder** object.

References **imode**.

unsigned ImageHolder::size () const [inline]

Get image size.

Gets image size in bytes.

Returns: Number of bytes occupied by the image.

References isize.

Referenced by saveSquare().

unsigned ImageHolder::rows () const [inline]

Get number of rows.

Access to the number of rows in the image.

Returns: Number of rows in the image.

References irows.

unsigned ImageHolder::cols () const [inline]

Get number of columns.

Access to the number of columns in image.

Returns: Number of columns in image.

References icols.

int ImageHolder::pixel (unsigned *Row*, unsigned *Col*, struct *Hsv & Pixel*) const

Get complete pixel info.

Access to each pixel complete info (*because some images types have more than one component in each pixel*).

Parameters: ***Row*** Pixel row coordinate.

Col Pixel column coordinate.

Pixel Reference to ***Hsv*** structure in which the info about the pixel(row,col) will be returned.

References `getPixel()`, `icols`, `imode`, `irows`, `Rgb::Rgb()`, `Rgb2Hsv()`, and `Yuv2Rgb()`.

Referenced by `convertImage()`, and `saveRectangle()`.

int ImageHolder::pixel (unsigned *Row*, unsigned *Col*, struct *Yuv* & *Pixel*) const

Get complete pixel info.

Access to each pixel complete info (*because some images types have more than one component in each pixel*).

Parameters: ***Row*** Pixel row coordinate.

Col Pixel rolumn coordinate.

Pixel Reference to ***Yuv*** structure in witch the info about the pixel(row,col) will be returned.

References `getPixel()`, `icols`, `imode`, `irows`, `Rgb::Rgb()`, and `Rgb2Yuv()`.

int ImageHolder::pixel (unsigned *Row*, unsigned *Col*, struct *Rgb* & *Pixel*) const

Get complete pixel info.

Access to each pixel complete info (*because some images types have more than one component in each pixel*).

Parameters: ***Row*** Pixel row coordinate.

Col Pixel rolumn coordinate.

Pixel Reference to ***Rgb*** structure in witch the info about the pixel(row,col) will be returned.

References `getPixel()`, `icols`, `imode`, `irows`, `Rgb::Rgb()`, and `Yuv2Rgb()`.

unsigned ImageHolder::pixelY (unsigned *Row*, unsigned *Col*) const

Get Y component of pixel.

Access to each pixel Y component

Parameters: ***Row*** Pixel row coordinate.

Col Pixel column coordinate.

References `getPixel()`, `icols`, `idata`, `imode`, and `irows`.

Referenced by `saveRectangle()`.

int ImageHolder::saveRectangle (unsigned *UpRow*, unsigned *LeftCol*, unsigned *DownRow*, unsigned *RightCol*, const char * *FileName*) const

Saves a rectangle from the image to a file.

Givin the Upper-Left and Bottom-Right coordinates, a rectangle from the image is saved to a file.

Parameters: *UpRow* Upper limit.

LeftCol Left limit.

DownRow Bottom limit.

RightCol Right limit.

FileName Name of the file to save (append) the rectangle image.

References `Rgb::b`, `Rgb::g`, `icols`, `imode`, `irows`, `pixel()`, `pixelY()`, `Rgb::r`, and `Yuv2Rgb()`.

Referenced by `saveFrame()`, and `saveSquare()`.

int ImageHolder::saveSquare (unsigned *Row*, unsigned *Col*, unsigned *Size*, const char * *FileName*) const

Saves a square from the image to a file.

Access to each pixel Y component

Parameters: *Row* Center row coordinate.

Col Center column coordinate.

Size Square side size.

FileName Name of the file to save (append) the square image.

References `saveRectangle()`, and `size()`.

int ImageHolder::saveFrame (const char * *FileName*) const

Saves all the image to a file.

Parameters: ***FileName*** Name of the file to save (append) the image.

References *icols*, *irows*, and *saveRectangle()*.

Bibliography

- [1] R. Hafner, S. Lange, M. Lauer, and M. Riedmiller. Brainstormers tribots team description. Technical report, Institute of Computer Science, Institute of Cognitive Science, 2008. University of Osnabrü, Germany.
- [2] O. Zweigle, U. P. Käppeler, T. Rühr, K. Haussermann, R. Lafrenz, F. Schreiber, A. Tamke, H. Rajaie, A. Burla, M. Schanz, and P. Levi. Cops stuttgart team description 2007. Technical report, IPVS, 2007. University of Stuttgart, Germany.
- [3] B. Kimiyaghalam, M. Y. A. Khanian, H. R. Fard, M. Montazeri, S. Moein, S. Morshedi, S. Ebrahimijam, H. Hosseini, and M. Hosseini KH. Mrl middle size team: 2008 team description paper. Technical report, Mechatronics Research Laboratory, 2008. Islamic Azad University of Qazvin, Iran.
- [4] J. J. M. Lunenburg and G. v.d. Ven. Tech united team description. Technical report, Control Systems Technology Group, 2008. Eindhoven University of Technology, Netherlands.
- [5] D. Jahshan. Mu-penguins 2008 team description. Technical report, Department of Electrical and Electronic Engineering, 2008. The University of Melbourne, Australia.
- [6] B. Bouchard, D. Lapensée, M. Lauzon, S. Pelletier-Thibault, J.-C. Roy, and G. Scott. Robofoot Épm team description paper 2008. Technical report, Mechatronics Laboratory, 2008. École Polytechnique de Montréal, Canada.
- [7] W. Chen, Q. Cao, J. Wang, and C. Leng. Jiaolong2008 team description. Technical report, Institute of Automation, Research Institute of Robotics, 2008. Shanghai Jiao Tong University, China.
- [8] Y. Sato, S. Yamaguchi, Y. Kitazumi, Y. Ogawa, Y. Yonemura, T. Ueoka, Y. Wada, Y. Takemura, A. A. F. Nassiraei, I. Godler, K. Ishii, and H. Miyamoto. Hibikino-musashi

- team description paper. Technical report, Kyushu Institute of Technology, 2008. The University of Kitakyushu, Japan.
- [9] H. Zhang, H. Lu, X. Wang, F. Sun, X. Ji, D. Hai, F. Liu, L. Cui, and Z. Zheng. Nubot team description paper 2008. Technical report, College of Mechatronics and Automation, 2008. National University of Defense Technology, China.
- [10] A. J. R. Neves, G. Corrente, and A. J. Pinho. An omnidirectional vision system for soccer robots. In *Proc. of the EPIA 2007*, volume 4874 of *Lecture Notes in Artificial Intelligence*, pages 499–507. Springer, 2007.
- [11] P. M. R. Caleiro, A. J. R. Neves, and A. J. Pinho. Color-spaces and color segmentation for real-time object recognition in robotic applications. *Revista do DETUA*, 4(8):940–945, June 2007.
- [12] A. J. R. Neves, D. A. Martins, and A. J. Pinho. A hybrid vision system for soccer robots using radial search lines. In *Proc. of the 8th Conference on Autonomous Robot Systems and Competitions, Portuguese Robotics Open - ROBOTICA '2008*, pages 51–55, Aveiro, Portugal, April 2008.
- [13] A. Dumitras and F. Kossentini. Fast and high performance image subsampling using feedforward neural networks. *IP*, 9(4):720–728, April 2000.
- [14] A. Dumitras and F. Kossentini. High-order image subsampling using feedforward artificial neural networks. *IP*, 10(3):427–435, March 2001.
- [15] A. J. R. Neves, A. J. Pinho, D. A. Martins, and I. Pinheiro. An efficient omnidirectional vision system for real-time object detection. Submitted, 2008.
- [16] I. Pinheiro. Automatic calibration of the cambada team vision system. Master’s thesis, Universidade de Aveiro, 2008.
- [17] U. Kaufmanns, R. Reichle, C. Hoppe, and P. Baer. An unsupervised approach for adaptive color segmentation. In *Proc. of the 2nd Int. Conference on Computer Vision Theory and Applications*, pages 3–12, Barcelona, Spain, March 2007.
- [18] G. Mayer, H. Utz, and G. Kraetzschmar. Playing robot soccer under natural light: A case study. In *Proc. of the RoboCup 2003*, volume 3020 of *Lecture Notes on Artificial Intelligence*. Springer, 2003.
- [19] B. Cunha, J. L. Azevedo, N. Lau, and L. Almeida. Obtaining the inverse distance map from a non-svp hyperbolic catadioptric robotic vision system. In *Proc. of the RoboCup 2007*, Atlanta, USA, 2007.

- [20] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.
- [21] L. Almeida, F. Santos, T. Facchinetti, P. Pedreira, V. Silva, and L. S. Lopes. Coordinating distributed autonomous agents with a real-time database: The cambada project. In *Proc. of the 19th International Symposium on Computer and Information Sciences, IS-CIS 2004*, volume 3280 of *Lecture Notes in Computer Science*, pages 878–886. Springer, 2004.
- [22] S. Mitri, S. Frintrop, K. Pervözl, H. Surmann, and A. Nuchter. Robust object detection at regions of interest with an application in ball recognition. In *Proc. of the 2005 IEEE International Conference on Robotics and Automation, ICRA 2005*, pages 125–130, Barcelona, Spain, April 2005.
- [23] A. Treptow and A. Zell. Real-time object tracking for soccer-robots without color information. *Robotics and Autonomous Systems*, 48(1):41–48, August 2004.
- [24] Fourth Workshop on Intelligent Solutions in Embedded Systems. *Embedded Real-Time Ball Detection Unit for the YABIRO Biped Robot*, June 2006.
- [25] J. Zou, H. Li, B. Liu, and R. Zhang. Color edge detection based on morphology. In *First International Conference on Communications and Electronics, ICCE 2006*, pages 291–293, 2006.
- [26] T. T. Zin, H. Takahashi, and H. Hama. Robust person detection using far infrared camera for image fusion. In *Second International Conference on Innovative Computing, Information and Control, ICICIC 2007*, pages 310–310, 2007.
- [27] S. E. Umbaugh. *Computer Vision and Image Processing*. Prentice Hall, 1999.
- [28] Y. Zou and W.T.M. Dunsmuir. Edge detection using generalized root signals of 2-d median filtering. In *Proc. of the International Conference on Image Processing, 1997*, volume 1, pages 417–419, 1997.
- [29] T. Blaffert, S. Dippel, M. Stahl, and R. Wiemker. The laplace integral for a watershed segmentation. In *Proc. of the International Conference on Image Processing, 2000*, volume 3, pages 444–447, 2000.
- [30] R. Boyle and R. Thomas. *Computer Vision: A First Course*. Blackwell Scientific Publications, 1988.

- [31] J. F. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6), November 1986.
- [32] E. Davies. *Machine Vision: Theory, Algorithms and Practicalities*. Academic Press, 1990.
- [33] R. Gonzalez and R. Woods. *Digital Image Processing*. Addison-Wesley Publishing Company, 1992.
- [34] P.-K. Ser and W.-C. Siu. Invariant hough transform with matching technique for the recognition of non-analytic objects. In *IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 1993.*, volume 5, pages 9–12, 1993.
- [35] Y.-J. Zhang and Z.-Q. Liu. Curve detection using a new clustering approach in the hough space. In *IEEE International Conference on Systems, Man, and Cybernetics, 2000*, volume 4, pages 2746–2751, 2000.
- [36] W. E. L. Grimson and D. P. Huttenlocher. On the sensitivity of the hough transform for object recognition. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 12:1255–1274, 1990.
- [37] D. Ballard and C. Brown. *Computer Vision*. Prentice Hall, 1982.
- [38] A. Jain. *Fundamentals of Digital Image Processing*. Prentice Hall, 1989.
- [39] D. Vernon. *Machine Vision*. Prentice Hall, 1991.
- [40] B. Eckel. *Thinking in C++*, volume 1. Prentice Hall, second edition, 2000.
- [41] P. Karras and N. Mamoulis. The haar+ tree: A refined synopsis data structure. In *IEEE 23rd International Conference on Data Engineering, ICDE 2007*, pages 436–445, 2007.
- [42] E. F. Ersi and J. S. Zelek. Local graph matching for face recognition. In *IEEE Workshop on Applications of Computer Vision, WACV 2007*, pages 3–3, 2007.