Moving patterns recognition & Location for Robotics

Final thesis report

Author:

Carlos Romero Bretó

Directors:

Dr. Ángel Valera Fernández

Dr. Antonio José Sánchez Salmerón

Introduction

This work, developed as a final degree project at Universidad Politècnica de València, feeds off the fields of pattern recognition and robotics, applying the former to fine-tune control the behaviour of the latter. Robot control algoritms, even designed as they indeed are to be accurate, are subject to noise, random interference from the environment, etc. which may cause them to prove themselves not quite so. Some of these problems or deviations from what should have been an ideal controller response, in particular when it comes to know where a certain robot is related to the world, can be detected and therefore rectified easily with a camera and software analyzing the scenario. This camera should track its movements and, somehow, be able to tell one robot from the other (for instance, by means of marks present in them).

One way to achieve this is by means of patterns attached to the robots themselves. Since we know the configuration of our patterns (which can vary greatly in size, kind of elements composing them, etc.) we can analyze the captured images with pattern recognition techniques and find out this and other information.

Aims

The project aims at providing a location and identification solution for a set of moving robots with their movements recorded and simultaneously processed. Gathered information could be dealt with in several ways:

- Displayed in an application.
  The process could be either set to continuously get and process information in real time, and thus showing all or part of the data gathered as it changes over time, or in a more detailed and paused fashion, capturing just some frames and showing all of their details.
- Stored for further anylisis.
  For developing routes tracing for mobile robots. For instance, the desired and resulting routes could be displayed graphically, making them easy to compare to one another.
- Be fed to the robots controlling algorithm.
  Of course, the algorithm itself can improve its behaviour by using the data provided as its input, effectively becoming feedback on how well it has been faring so far.
- etc.

Possible scenario

A possible scenario is depicted in figure 1:

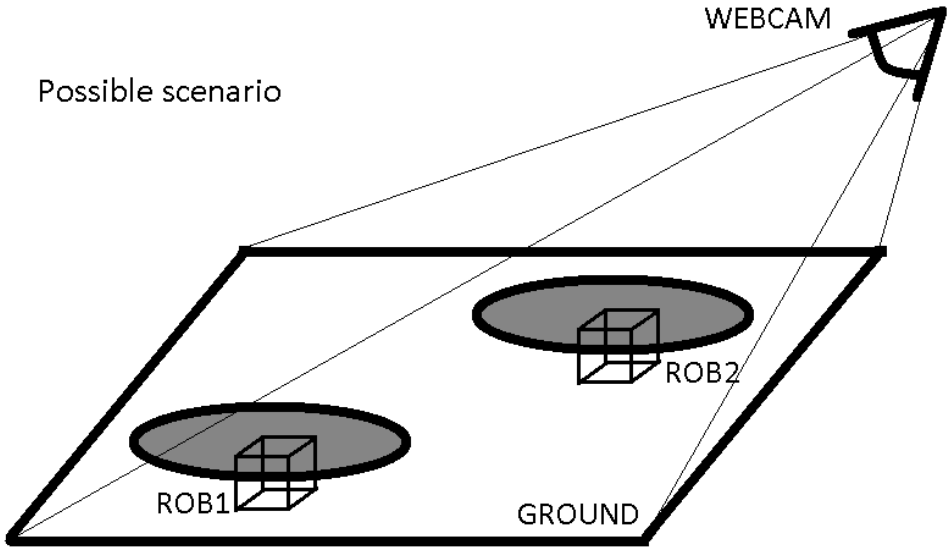


Figure 1 – *Depiction of possible scenario*

Robots are assumed to move freely on the area marked as ground, meaning they can move anywhere in

the area the camera captures. It is necessary to impose some restrictions in its topology, for instance, a gradually descending slope or higher/lower areas. In particular, patterns must remain within sight and in an affordable angle, so no critical information is missed (if that happens, at best the information may stop its flow, at worst it may turn out to be wrong). As far as this project is concerned, the ground is perfectly horizontal and fairly well illuminated. Also, we will discuss why a zenital camera location (that is, its vector parallel to the ground normal vector) is better, and how those with an angle are bestleft avoided, mainly due to perspective distortion. Here, robots are represented just by cubes. Each one of them carries a pattern on top of it, so its normal is parallel to the ground's as well.

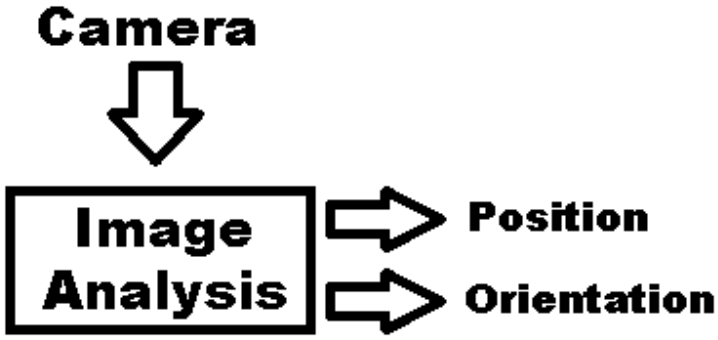
Regarding how the robots actually move, we have got several possibilities at our disposal. Among the ones we will later on offer a brief explanation of, we will choose a few to reflect how results vary according to the movements patterns typically found in each one of them.

Behaviour and expected results

Output should follow the structure shown in table 1, the 3 variables corresponding to the 3 degrees of freedom present in a 2D scene. Please note this table will be repeated for each detected artifact, so we must ensure as well a way to find out which of them the corresponding data is associated to (that is, an ID for each set of results).

Orientation	Position
α_1	x_1 , y_1
α_2	x_2 , y_2
α_3	x_3 , y_3
...	...

Table 1 : *Output*



Information is gathered and processed as fast as it is possible. The processing time in which each frame is taken care of should differ marginally from one to the following, so a frequency can be computed with *frames*/*second* units or FPS. The

Justification

Nowadays a great amount of research is done involving computer vision. It is a big field of study, where many approaches to one same problem are possible, each one with advantageous characteristics and some other not quite so ones. Often, these projects cannot rely just on themselves to fully check whether they are doing things the right way. An additional work, focused on analyzing their outputs and offering a second opinion, is desirable to get more accurate or fast results.

This project attempts to be of help in those cases where other projects are suitable to benefit from it, like robot coordination. In this sense, it can be understood like a support tool which checks whether the behaviour of other systems remains within acceptable ranges.

Report organization

The report has been organized in four main groups plus two annexes:

- A theoretical approach, covering aspects and options available to be taken in consideration

when dealing with the problem, like the necessary techniques to apply, the technology available, etc.

- A practical approach, where the algorithm is actually implemented. Means to test it and watch it work are developed here as well. Explanation of the code, both of the computations and the testing environment, is provided in detail.

- An interface overview for the accompanying application. This application helps in visualizing the problem and how a solution is reached; usage samples are provided as well in Annex B.

- A compilation of the software tools required, and how they must be set up.

- Annex A is a compilation of the patterns used, in real size.

- Annex B deals with an application sample usage to demonstrate how the intended recognition can be carried out once every necessary component has been installed and set up.

- Annex C depicts the UML schema for the developed application.

Theoretical approach

As has been briefly stated the amount of fields of study involved is quite varied. It is an important task to coordinate all of them in the proper way so the task gets carried out while its results prove accurate and close to what really is going on. Additionally, each one of them must be properly set to work in an efficient way so the highest FPS possible is achieved.

We will discuss the different options which this work found along the way. Among several possibilities for a given circumstance it will be stated which was decided for and applied, or which just could not be further contemplated due to some impossibility in their usage. Some of them, particularly those related to feature extraction, were partially implemented and subsequentially abandoned due to being of lower quality of some others. The best example of this is considering the image as either a set of blobs, which can be connected with 4- or 8-connectivity, or as a set of edges which can be approximated to a finite number of ellipses.

Some background information is provided as well on several aspects, offering an environment onto which set this work and clearly see its relationships with each discipline. Available technology (like the different types of cameras, light conditions, etc.) is described, its properties explained and finally discussed about on whether they are desirable or not. Even if not extremely crucial, the basics of robot control are shown as well – as long as the required information is present within each frame, it does not matter much the way it moved from point *A* to point *B*.

Last but not least it is necessary to establish a formal algorithm to be followed then finally obtaining the output values. Thought is given to how it manages its data structures and how it handles the available resources.

Image definition

The way captured images are prepared and analyzed is a cornerstone in the recognition plus location process, or simply location. The factors contributing to a good handling are diverse, like aproppriate settings for the camera, proper preparation for the data gathering to be carried out next, and correct interpretation when it comes to features extraction.

Image dissection

An image will be represented by a grid of pixels or *pictorial elements*, following a box-like pattern and area assumed not to be null. Each one of them is identified by a 2-D coordinate (x_i, y_i), which determines its relative position to some origin. This is called an array and within each of these cells one or several values are stored, depending on the type of image it is representing. These bidimensional arrays are bound by two values, the width and height of the image, and so a valid pixel for a given image is that the coordinates of which are within the range

([0, *width* - 1] , [0, *height* – 1])

for *X* and *Y* respectively.

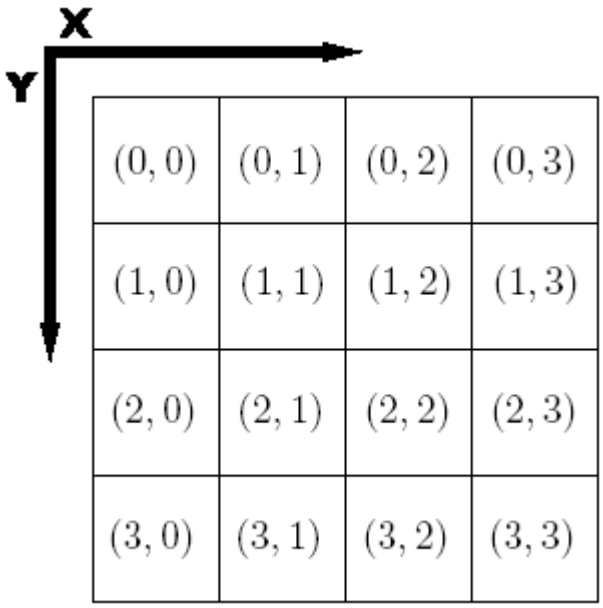


Figure 2 - *Image pixels*

Pixel connectivity
Furthermore, connections among these pixels can be established. For instance, the neighbours of a given one are defined as those in touch with it in 4 or 8 directions (commonly called 4- and 8-connectivity):

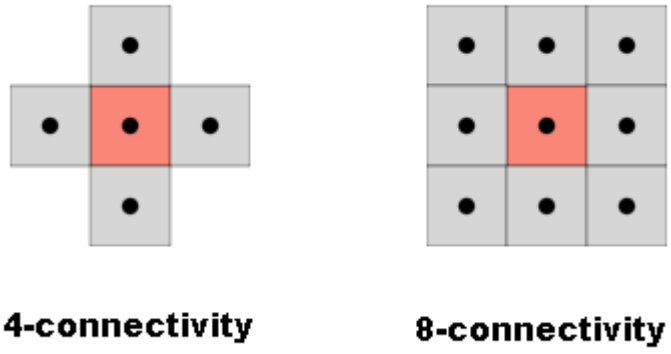


Figure 3 - *Types of connectivity*

8-connectivity includes, as figure 3 shows, those pixels only in touch via the corners of the pixel.

Colour models

A colour model defines a meaning for each of the values stores in the pixels. Different values in different colour spaces can provide the same resulting image, and conversions among them are possible via different formulas. The *de facto* standard for computers is RGB, while YUV being closer to that of humans and more compact is more commonly found in analogial or digital TV equipment; for instance, it is used in the *PAL* and *NTSC* broadcast standards.

Y'UV
Its term encompass one for *luminance* and 2 for *crominance;* the separate terms bounds are as follows:

- Y' : [`0.0` , `1.0`]
- U : [`-0.436` , `0.436`]
- V : [`-0.615` , `0.615`]

Y'UV has as a property greater capabilities of hiding flaws in the image due to errors in transmission or compression. Even if initially was considered as an option for our images it was later declined due to ease of compatibility issues with other parts of the project.

RGB
3 values are stored for each pixel, one for each component in the *RGB* colour mode, corresponding respectively to the red, green and blue components in the final colour. Normalized bounds for all three components are:

[`0.0` , `1.0`]

There may exist an additional one for the *alpha*-channel, usually used in transparencies and of no relevance for our purposes.

Conversion between Y'UV and RGB
As can be seen in figure 4 conversion between the two colour models is pretty straightforward with a conversion matrix:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0,299 & 0,587 & 0,114 \\ -0,147 & -0,289 & 0,436 \\ 0,615 & -0,515 & -0,100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Figure 4 - *Y'UV to RGB and vice versa conversion matrix*

Given that it is widely accepted and that *OpenCV* is particularly focused towards it, the *RGB* colour model will be assumed in this project. Both colour and gray-scale images use this system, even though in the gray-scaled ones case its three components are summarized as their mean into one single value, corresponding to the pixel *intensity*.

Image types

Usually cameras capture colour images, which are great for human vision. Grey-scaled are just quite like so; however, their representation is simpler as less information is required: to convert from a colour image to a gray-scaled one, one just must obtain the mean of the component values and stick with it. Also, several techniques are more easily applied on images the pixels of which are defines by just a single value.

Colour images

These images will be those initally captured and will serve as the starting point for the rest to come. They also provide an excellent background on which to represent data we consider interesting, as they show exactly what the cameras record and the information can be overlapped on it to produce user-friendly output.

Gray-scale images

Grey-scaled images can directly be obtained from a colour one by means of this formula, applied on a pixel-per-pixel basis:

$$G_{x,y} = (R_{x,y} + G_{x,y} + B_{x,y}) / 3$$

The obtained values represent from now on how close the pixel intensity is to complete black or complete white, in an ascending scale. Given we are dealing with normalized values, the intensity level will remain within [`0.0` , `1.0`] as well.

Cameras

Cameras represent means by which the world's representations can be captured and stored. The term camera comes frome the *camera obscura* latin term meaning "dark chamber". Cameras are the source of the captured images, capturing the world and projecting it onto a suitable sensor, i.e., a device that records and stores images.

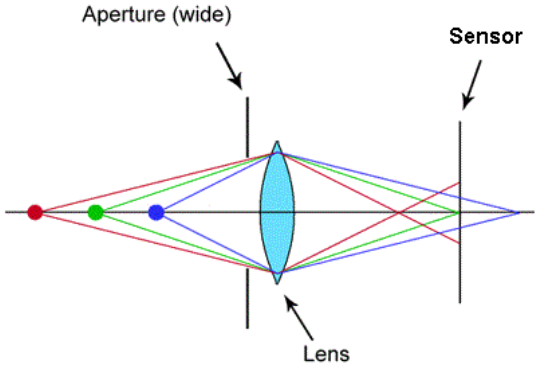


Figure 5 - *Camera schema*

The first camera is known as *pinhole camera* and merely consists of an enclosed volume with a pinhole (which would later become a lens) through which light is admitted, forming an image of external objects on some surface. These images can be still or a moving set of them, forming a video for instance. An schematic view of this can be seen in figure 5, where the surface images are projected on in this case is some modern sensor.

When it comes to transferring the data from the sensor to the computer, the type of cameras comes into play. For USB cameras it is enough to copy the data to a local buffer and use it from that point. For web cameras it is often needed to fetch the image via HTTP, accessible in some simple server the camera itself provides.

Their main components of a camera are the lens, the sensor and the diaphragm; all of them modify how the resulting images are acquired. A brief overview of them follows, along its most important properties. Also, two cameras which were used during development are shown as examples.

Lenses

Lenses typically found in cameras are positive or convergent or convex, meaning their focal distance is positive as well. Focal distance is defined as the length of the optical centre of the lense and the point in which the incoming, parallel light beams become focused. In analogy to the human eye, the lense would be the pupil.

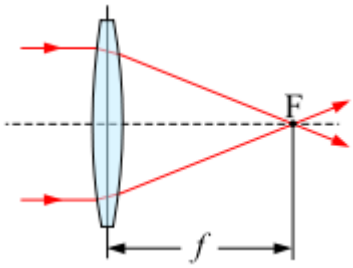


Figure 6 - *Focal distance*

Sensors

Image sensors are in charge of converting an optical image into an electronic signal by reacting to the amount and frequency of the light it receives. Most popular sensors are formed up by either a charge-coupled device (CCD, which is an analogical device) or a complementary metal-oxide-semiconductor (CMOS, which requires some additional circuitry). Analogous to a human eye, a sensor would correspond to the nerve endings present in the back of it.

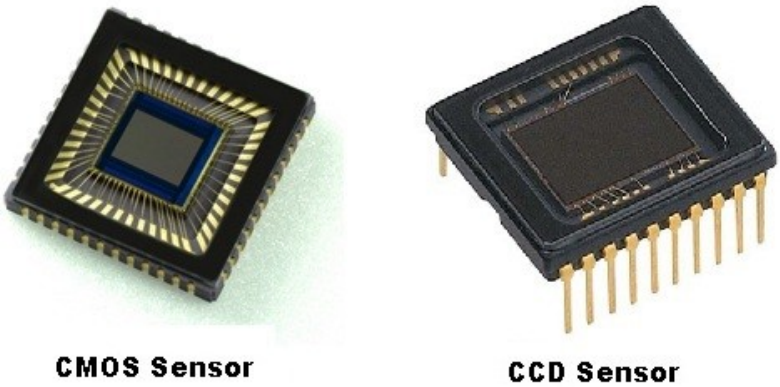


Figure 7 - *CCD and CMOS camera sensors*

Diaphragms

A diaphragm is a thin opaque structure with an opening at its centre; diaphragms stop the passage of light except for that passing through this aperture. Again, in the human eye the diaphragm tasks are performed by the iris, limitating the amount of light reaching inside through the pupil. It is worth noting that the diaphragm must be placed in the light path of the lens, and the center of the aperture must coincide with its optical axis as well. The aperture levels are standardized with fixed values in the form f *X*, where some examples for *X* are shown in figure 7.

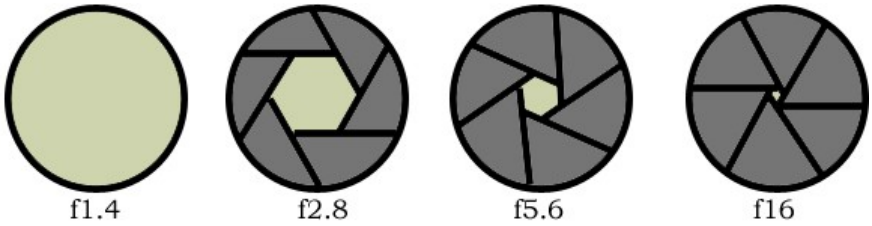


Figure 7 - *Different apertures for a diaphragm*

Some example of cameras

The two cameras described here are very briefly summarized, with their distinctive features highlighted. Much more information could be extracted from it, however, most of it is irrelevant for the project. For instance, shutter time (the shutter would be the "eyelid" of the camera and controls exposition time) can be relatively low (meaning new images are available at a higher frequency) but, since the processing overhead is large by itself, it means most of them will be wasted while we deal with the current one. Put in short, our goal regarding this will be to process each image as fast as possible and skip to the first available after that has been done.

USB cameras – Logitech Pro9000

This camera is depicted in figure 8. Some important specifications provided by the manufacturer are in table 2. During development this camera was positioned with the lens center parallel to the ground's normal (that is, facing exactly down) and therefore perspective deformations were kept to a minimum. As we will see, this offers many advantages like smaller sizes for patterns and their contents more easily segmentated.

Figure 8 - *Logitech Pro9000 camera*

Focal length	2.7 mm
Image Sensor	1/2" CMOS
Diaphragm aperture	f 1.8
Resolution	640x480 to 1600x1200

Table 2 - *Pro 9000 characteristics*

Network cameras – AXIS 212 PTZ

Again, the important facts are gathered in table 3. This camera is capable of panning, tilting and zooming; however, changes in any of those during capture would mean different or otherwise changed views of the scene, making it an unsuitable option (specially since we will need a background image according to which the processing of the rest of captured images will be carried out). During development this camera had a great inclination angle (and was positioned at a greater distance, with only lower resolutions available) making it a bad choice for image acquisition. However, for some time it was the only one available so it finally was included, serving as an example on how images are fetched from a web camera.

Focal length	3.7 mm
Image Sensor	CMOS
Diaphragm aperture	f 2.0
Resolution	160x90 to 640x480

Table 3 - *AXIS 212 PTZ characteristics*

Figure 9 - *AXIS 212 PTZ camera*

Image processing

Image processing can be applied both to a photograph or a video frame and involves images as input, such as the previously described, and another image, or perhaps a set of features extracted, etc. as output. Most techniques treat the image as a two-dimensional signal (with several or just one components) and apply standard signal-processing techniques to them.

Some common techniques

The following are some of the more widely used ways to transform an image. We offer a selection of those which were considered to be included in the final project, and is by no means a complete list.

Geometry transformation
These encompass those of Euclidean nature, like enlargement, reduction and rotation. They produce images like the original except their geometry has varied: it may have been stretched in a certain direction, or rotated α degrees. Enlargement is of particular interest, it would allow a certain difference between two consecutive pixel values to be distributed throughout a larger number of them, therefore making it simpler to trace a dividing line between those lying "closer" to one value or the other.

Images difference
The main purpose of this technique is to detect changes between two images, considering one of them as the background and the other as the one we should detect objects in. Some requirements exist for this to work, namely, aligned, equally-sized images and pixel values lying within the same range, so the comparison makes sense. This background substraction can be carried out either by directly substracting the values and clamping the results to the previously set bounds, or by dividing such values and therefore obtaining a ratio.

Radial distortion correction
Radial distortion, also known as barrel distortion, is an effect caused by the camera lens. Modern ones are relatively free of such geometric distortion; however, a small amount always remains. It is most visible when taking pictures of structures having straight lines which then appear curved up to a certain degree determined by the distance from the centre in a symmetrical fashion – hence the name. A polynomial transformation is enough to correct this. However, we will not be correcting this parameter due to the shape of the patterns we will adopt, which we shall discuss later in this document to be sort of oblivious to it.

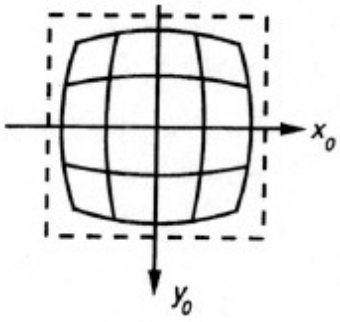


Figure 10 - *Barrel distortion*

Preprocessing
The stage of preprocessing starts as soon as the image to be analyzed is captured, and aims at transforming it in such a way other algorithms, like pattern recognition ones, can be easily applied. Preprocessing is a standard step in image processing and crucial when it comes to properly analyze the information contained in them, either by highlighting the aspects we are interested in or hiding those in which we are not, as both tend to interweave. We will assume the input of this step to be are gray-scale images, acquired either by converting the colour ones into them or simply setting the camera to capture like so for us.

Thresholding
Thresholding is the simplest method of segmentation, and can be used to create binary images. It feeds from a gray-scale image and has each of its of its pixels evaluated as lower or higher than a certain threshold and labeled, respectively, "background" or "object" pixels. This is known as *threshold above*; the opposite, considering "background" pixels those with higher value than the threshold and vice versa, is known as *threshold below*.

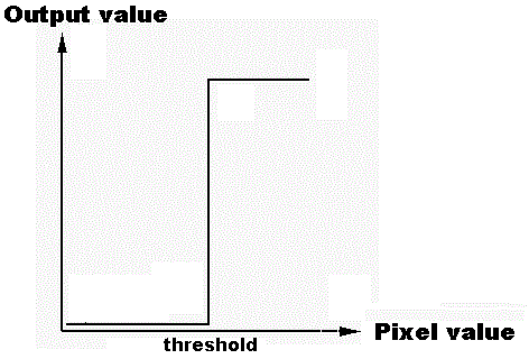


Figure 11 - *Thresholding function*

Once these labels have been set a binary or monochrome image can be created by assigning the maximum value to "object" pixels and the minimum to "background" pixels, effectively making it a black and white image.
Additionally, thresholds may be multiple (choosing for instance those pixels with their values ranging between the two of them) and have varying values according to which region of the image they affect (also known as *adaptive thresholding*). The effect of a single-value thresholding can ben seen in figure 12.

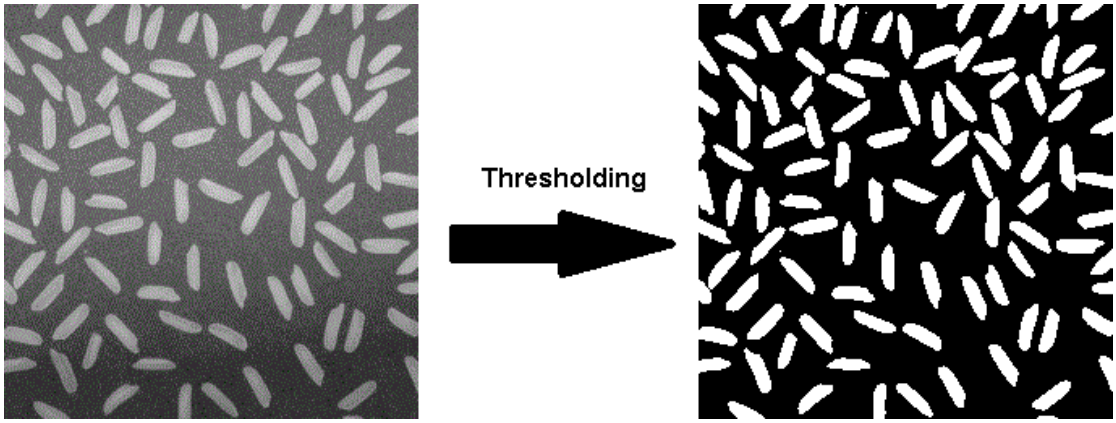


Figure 12 - *Thresholding*

Thresholding value selection
The most important aspect to take into account when using thresholding is the value which will be used to distinguish between "background" and "object" pixels. It can be manually set, or a thresholding algorithm may compute a value automatically based on the results obtained for a certain initial value (this is known as *automatic thresholding*). Some simple ways to determine this number are, for example, taking the mean of all pixels or choosing that corresponding to the biggest valley in the image histogram (though, regarding the latter, it cannot be assured the histogram will possess clearly defined valleys).

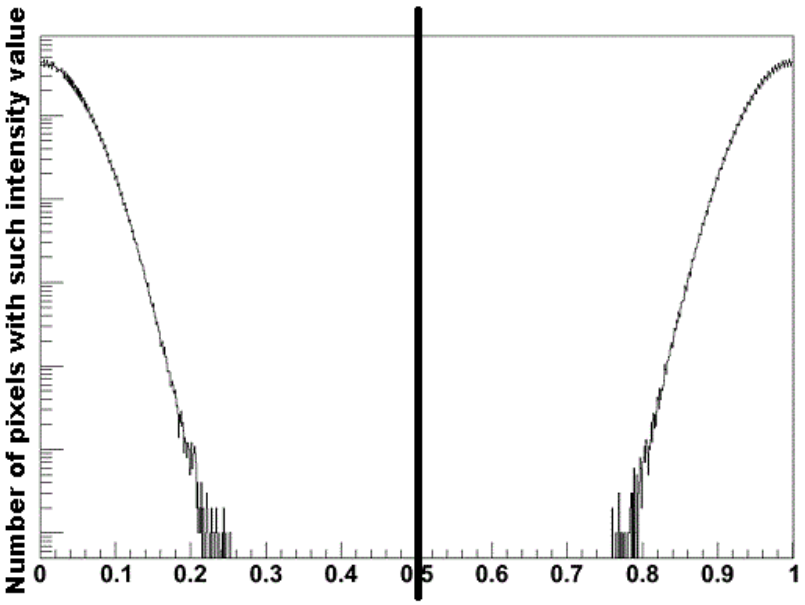


Figure 13 - *Figuring out a good threshold value*

For our purposes a single value, non-adaptive threshold will be used, as our experimenting conditions will tend to be foreseeable .

Light correction
The area the camera will capture will most probably be subject to gradual changes in illumination across its surface, be it because, for instance, the image is lighter on one of its sides because of a closer light source. Previously to any of the patterns to be present in the scene, light can be captured easily by taking a grey-scale picture (as its values merely represent intensity) to be used as "what the empty scene looks like". It can later on be used to alter the values in subsequent captures in order to minimize the effect of light.
Background image effect on the result
A per-pixel ratio is computed from the background image pixel values against its mean illumination.

$$Ratio_{x,y} = BGround_{x,y} / BGround_{mean}$$

The higher the value of this ratio the further the currently examined pixel should lie from the mean, probably due to irregular light patterns, in the subsequent captured images. This can be used to correct them, adjusting up and down the illumination in accordance to what the background suggests.

$$Img'_{x,y} = Img_{x,y} * Ratio_{x,y}$$

Robots

A robot is an electro-mechanical system programmed to perform a certain task, taking decisions on how to achieve it by reading its sensors' input. It could be said, by observing it, to have a purpose of its own, particularly if they are able to exhibit an intelligent behaviour for instance mimicking that of humans or other animals. Robots are mainly divided into mobile and static according to their capabilities or purpose of moving around as they proceed with their assigned tasks, although many other features exist: whether they possess a moving arm, intended for industrial usage or otherwise home-related jobs, etc.

Robotics is a broad term dealing with the design, construction, operation, etc. of robots.

What a robot is

There is not a rule of thumb to tell what actions are robot-like and which are not. The general definition of a robot expects from it things like moving, working a mechanical arm and sensing its surroundings while carrying out a task by means of thought-out actions. In this project robots are merely observed and are used, along with the patterns they carry, as a source of coordinates and orientation angles to be estimated. Once that is accomplished, feedback is provided for instance on whether the control algorithm is performing OK. Naturally, mobile robots are the best option available since they are able to actually move and drag their patterns around with them. Static robots could be used as well, attaching the pattern to its arm and tracking the location of it instead of the main robot body.

Special attention should be paid in the case of say an arm configuration, for the elevated amount of degrees of freedom may cause the pattern to become hidden. Degrees of freedom are the set of independent displacements and rotations that specify the current location (position plus orientation) of a body or system. For robots which execute tasks with a mechanical arm in the *open kinematic chain*, consisting of rigid links connected at joints, each joint may provide an additional degree of freedom. One of these systems is said to be redundant if the number of controllable degrees is greater than the actual amount of them – for instance, the human arm operates in a 6-degrees of freedom fashion (3 for position and 3 for orientation) while possessing the ability to operate with 7 (3 in the shoulder, 1 in the elbow and 3 more in the

wrist).

Brief history of robotics

Even though there have been attempts at designing and building robots for hundreds of years, the first fully automated robot only appeared in the second half of the 20th century. However, it took a short time to become widely-known and the concept spread fairly quickly. For instance, the term *robot* was first used in printed form by Czech writer Karel Čapek in his play *Rossum's Universal Robots*, in 1921, while the term *robotics* was coined in 1941 by Isaac Asimov in a short story called *Liar!* A brief timeline of experiences with robots can be found in table 4.

Year	Significance
1st century	Descriptions of over 100 machines and automata utilizing pneumatics, like a wind organ.
1206	Programmable automaton band.
1738	Mechanical duck able to eat, flap its wings, excrete...
1961	First installed palletizing robot.
1975	Programmable universal manipulation arm.
2009	Largest, strongest industrial robot with six axes is built.

Table 4 - *Important historical facts for robotics*

Robot components

The list of important components varies according to the configuration of the robot. However, some of them are likely to appear in every configuration.

Power source

Currently the most commonly found option is batteries, with compressed gases or liquids as a possible alternative.

Actuators

Actuators convert stored energy into movement. The classical actuator is the electric motor; research on pneumatic muscles (contracting when air is forced inside it) and electroactive polymers (slight contraction upon electricity running through it) has been carried out however.

Figure 14 - *Lego NXT motor*

Sensors

Sensors may offer an estimation of a physical variable important for the robot's behaviour for some reason. The nature of this variable can vary greatly – from temperature or contact/non-contact to present electromagnetic radiation in the form of visible or infrared light, and coded as an image in some format, or ultrasounds.

Figure 15 - *Lego NXT distance sensor*

Manipulators
Robots need to interact with objects in order to affect the real world. Objects can be picked, assembled, destroyed... via manipulators like mechanical grippers, consisting of at least two fingers for picking and letting go, or vacuum grippers (provided the surface is smooth enough to allow suction).

Classification of robots

Whether a robot moves in space or not is a distinctive enough feature to classify them as mobile or stationary robots. In this work our attention will be focused on the mobiles ones, as they will provide the real challenge in tracking the patterns along their path, without dwelling much in how they achieve this. A brief, broader discussion on this is nevertheless necessary.

The three more obvious ways to classify a mobile robot are:

According to their intended use
First characteristic to establish a separation between the different types of robots is the purpose they are expected to achieve. In that way, we can talk about industrial robots, which are expected to be productive at some certain repetitive task, or mobile robots which are of a more research-inclined nature.

Industrial robots
An industrial robot is defined as an automatically controlled, reprogrammable, multipurpose manipulator with three or more axes designed for a variety of industrial tasks like welding, painting, assembling, palletizing, etc.

The most commonly used are:

- Articulated robots.
- SCARA robots.
- Cartesian coordinate robots.

Figure 16 - *KUKA KR 210-2 industrial robot*

- **Mobile robots**

A mobile robot is an automatic machine capable of moving in a given environment, therefore not fixed to one physical location. This kind of robots are our main interest in this thesis, as they provide movement to the patterns in order to check the correct working of their tracking. Most of the experimentation was carried out using Lego NXT robots, which allow many possible configurations and sensors.

Figure 17 - *Lego NXT robot configuration*

According to the environment they move in

This in an important factor when it comes to robot design, as different environments will require different means of locomotion. The most popular classification is:

Land-based

The most common to be found, they are usually wheeled but also include legged robots (with two legs mimicking humans and resembling animals or insects with more of them). The robots classified under this category are of interest to us.

Air-based

They are known are UAVs (unmanned aerial vehicles) and include planes or helicopters. They are characteristic by the considerable difficulty in flight control. Also, given their usually fast speed, applying our project to track them would prove a rather limited option as it deals with static cameras.

Figure 18 - *Parrot Air Drone*

Water-based

Most of the work done in the area has had a commercial focus, like pool-cleaner robots. They are usually known as AUVs (Automated Underwater Vehicles). Should these kind of robots interact with our research, it would be needed of them to remain in a roofed area (or keep a fixed camera above them by some other means) and not to go underwater (as the pattern would no longer be visible).

According to the employed movement device

The choice of means of movement will dictate where the robot may actually move.

Given the nature of the work we are dealing with we will limit ourselves to a classification within the wheeled robots. Far more options exist, like robots with 5 or more wheels or a car-like steering approach, though they are not that common in robotics and were not further considered for our purposes.

1 point of contact robots

These are based on the inverted pendulum physics, where the point of contact can be for instance a wheel or even a ball. Gyroscopes are used to detect the inclination and an appropriated motor response is calculated many times per second, therefore keeping the main mass of the robot above the pivoting point. One of these was used in the tests carried out during development, consisting of a ball at the base of the robot which is made spin by two motor-controlled wheels in touch with its surface.

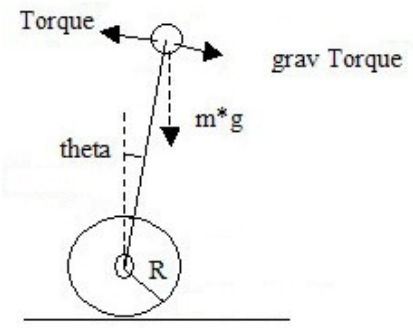


Figure 19 - *Inverted pendulum*

2-wheeled robots

Also called dycicles they are rather hard to balance as they need to keep moving not to fall. The trick consists in keeping the base of the robot under its centre of gravity. A good example is the *Segway personal transporter*, which resembles the model depicted in figure 20 with an additional vertical handle. A person can climb on its base and, by means of inclining this handle forward or backward with their weight, they can tell the robot to move in the appropriate direction while keeping balance.

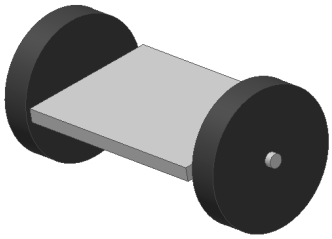


Figure 20 - *2-wheeled robot*

3-wheeled robots

This kind of robots are among the most estable provided they move on some reasonably smooth surface. There are two options regarding how the movement is controlled. In figure 21 a *differentially steered* robot schema is shown: rotation during movement is afforded by varying the relative rate of rotation of both the wheels attached to motors, while the free turning wheel is just used to keep balance. Another possibility, called *powered steering*, keeps both motors at equal rates of rotation and adds a third motor attached to the steering wheel to help in rotation along the ground's normal vector. For our purposes we will stick to the *differentially steered* approach, as it is simpler and most robots available follow it.

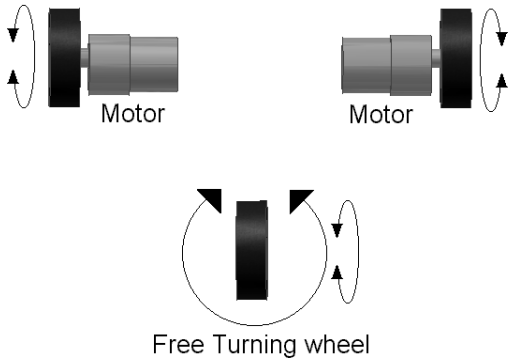


Figure 21 - *Differentially steered robot*

4-wheeled robots

One way to achieve this is take the *differentially steered* approach and add a second free turning wheel for extra balance. On the other hand, four total motors could be attached, one per wheel, and be fed with the same impulses if they are on the same side. As can be guessed the main difficulty in this is keeping each pair of wheels turning at the same pace.

Types of robot controllers

Robot controllers are a particular case of system controllers where the variables controlled reflect the actions and current state of the robot. A very straightforward way to put them to use lies in motor control, feeding apropppriate responses to it. Merely as an introduction we will offer a brief description of the main types of controls to be found – mobile robots in general will definitely need a closed-loop controller because feedback is crucial in their behaviour.

Open-loop controller

An open-loop controller obtains its output by just processing in some way its inputs, meaning the system will not pay attention the process' feedback to check on how it actually is faring. This model is adequate for well defined, simple tasks.

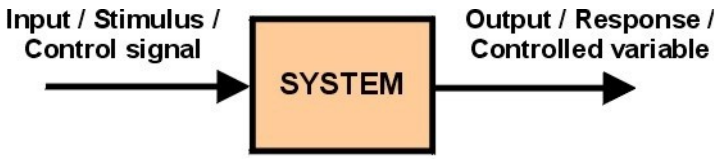


Figure 22 - *Open-loop controller*

Closed-loop controller

On the contrary, a closed-loop controller does take into account the output of the system in its calculations, measuring it by a sensor and comparing it to the reference or setpoint value.

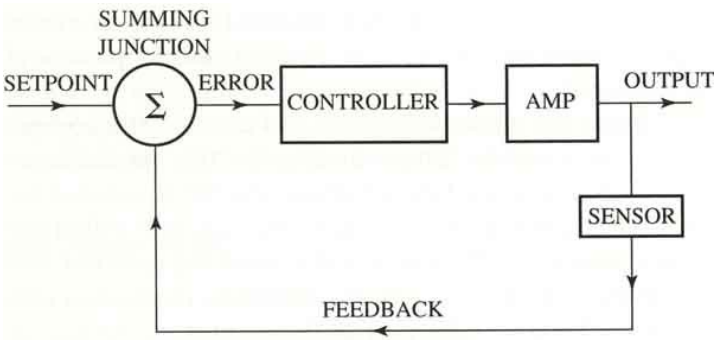


Figure 23 - *Closed-loop controller*

The way this computed deviation from the desired result, or error, affects the behaviour of the controller gives way to 3 possible actions based on current and/or previous values.

Proportional action

The computed action to be fed to the system is only dependent on the current error. On its own it may cause sluggish performance and oscillations in the output signal. In some delicate equipment this can cause breakdown.

Derivative action

Derivative actions deal with the rate of change of the error signal – not only it is important to know how far we are from the desired value but also how fast we are approaching it.

Integral action
This actions gives a greater importante to relatively long lasting, constant error values in an attempt to reduce them to zero.

Patterns

With a general idea in mind regarding what will be going on it is necessary to pay attention to how patterns will be defined along with the conditions the camera will be capturing frames in. These two considerations will remain closely tied together and, and often, changes in one of them will affect the other and vice versa.

We have got some freedom during pattern design; after all, it is up to us to determine how they shall be recognized and located. In the same fashion, the cameras can be adjusted to best fit the conditions we experiment in: all tests were carried out in a lab room where light conditions, interferences from non-project related sources and so on were kind of avoidable. This was nonetheless a recurring area to think on since some of the changes and additions adopted during development (like an inclination in the camera angle or the quality of the camera itself) greatly modified the restrictions imposed on patterns.

Defining patterns

A good pattern should fulfill the following requirements:

- First and foremost it must contain the information we will look for when examining it, and just that. This means its position and the direction it is facing should be inferable from its contents. In other words, redundant information should be kept to a minimum while keeping explicit.

- It should be possible with fairly ease to, given one of the components of the pattern design, be able to determine which other components belong to the same group. This could be understood as unity and compactness.

- Patterns should be easily distinguishable one from each other, even when packed together in a small area. In order to determine which of the detected artifacts in the image belong to a certain pattern, relative position among them should be a preferred guideline than, say, distances. A good way to express this is isolation among patterns.

Let's see how this reflects on the design decissions:

Content
As we know, most important features to extract from the patterns are position (or a centre), an angle and an identification. A good way to make the representation more compact is to pack two or more of the desired attributes into one, so with just one feature several bits of information can be obtained.

To see how this applies to our case, let's consider concentric circles (other types of figures were considered, like triangles for instance, but none proved reliable enough to pack position and ID into them like circles). By considering a centre shared by several circles we can not only mark that center as the central point of the pattern, and thus obtain the position, but also to count them up an take this number as an identifier for the global structure.

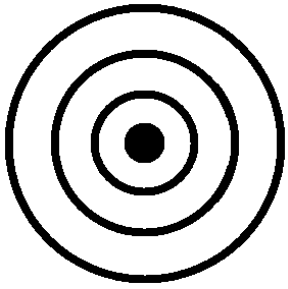


Figure 24 - *Concentric circles can be used for location and identification tasks*

The immediately arising question is, how many of these concentric circles are necessary in order to consider grouping them into a detected pattern? Two of them proved to be enough as long as some rules to avoid randomly detected circles which happened to share their centre fooling us into believing a real pattern is present.

Next and last aspect to consider is the angle the pattern is rotated. Inherent to the circle is its radius, which can be taken as an arrow pointing from the centre of the pattern to the faced direction.

All things considered, the final appearance of a pattern is depicted in figure 25. The basic pattern is conformed by a middle dot and an outer circle, and the number of additional circles between them will identify the pattern: no inner ellipses would mean pattern #0 and five ellipses would correspond to pattern #5, for instance. A radius is added as well, without actually touching the circles.

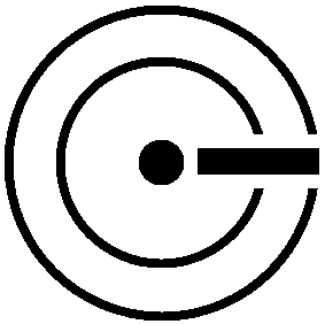


Figure 25 - *Pattern #1*

Size

A reasonable size is important for two reasons: first, if it is too small the camera may be unable to properly detect the features in it, and second, if it is too big robots may get cut short on area available for movement.

Finding out an appropriate size was a rather empirical process. The resolution and location of the cameras played prominent roles, as both affected directly the minimum required size. As we will discuss shortly, this could mark the difference between using a pattern with a 60cms long diameter and one with 25cms.

Number of paterns

The maximum number of detectable patterns is a fairly arbitrary value which, nonetheless, should be established prior to execution as we are likely to know how many robots will come into play. For our work a total of 6 was taken, this value also is a good balance between demonstration purposes and the room available within the pattern, as a higher number of patterns means more tightly-packed lines.

Shooting conditions

Under shooting conditions we encompass the variables affecting the camera while it shoots. We shall focus primarily in the distance to the objects of interest, the resolution it takes pictures at and its relative position.

Distance and camera location

The *Logitech* USB camera was placed, as we mentioned earlier, with its shooting direction parallel to the ground's normal. This caused the circles in the patterns not to appear distorted and eased the tasks of recognition. On the other hand, the *AXIS* web camera aimed at the scene at approximately 45º which for one caused perspective distortion and also meant a greater shooting distance.

No data was gathered as to which impact the actual shooting distance had on the result, as the cameras were not moved during development, but presumably the greater it is the more resolution will be needed and the greater the impact of its inclination, if any. A distance of about 3 metres was estimated for the straight-down looking one (*Logitech*) and 5-6 for the perspective one (*AXIS*).

Resolution
3 resolutions were employed during tests. For one side we are interested in keeping it as small as possible, given that lower resolutions involve less computational cost. However, they offer a lesser degree of detail as well. Out of `800x600`, `960x720` and `1024x764` the second one proved to be more balanced (and the greatest achievable in the *AXIS* camera due to its limitations)

Some conclusions
It is easy to conclude best results were obtained with the *Logitech* camera. Not only the size of patterns reduced drastically, but it also was way better located and of significantly better manufacture. Just as a reference, with this camera an `800x600` resolution usually missed about half the patterns in Annex A, while with `960x720` managed to get them all right. Moreover, data is transferred via USB which is definitely faster than accessing a web server and retrieving an image object with an HTTP request.

Fine-tuning patterns
This section will deal with some final thoughts on proper pattern design. The support for the content should ideally be pure white, while the contours should be black. Also, matt is preferable to shiny as the latter may confuse the algorithms due to bright spots being taken for white space (that is, the reflection of light on a shiny black surface may appear white in a picture of the scene).

Enclosing patterns
It has been stated that the radius present in the patterns does not get in touch with the concentric circles, so they are not considered as the same object during analysis. Therefore, circles should have an opening or gap through which this radius can be drawn. However, this must not be the case for the outer circle: if left open, the detection of this outer contour as an additional circle would depend on the ground and particularly the shadows the robots itself cast. In other words, it would mean the number of concentric ellipses would vary. Since pattern identification relies on that number to compute it, we cannot afford it and will instead stick to clompletely closed outer circles.

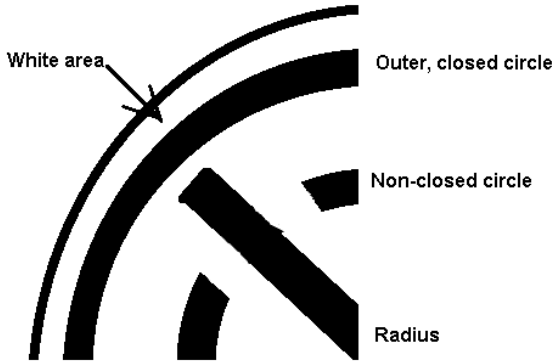


Figure 26 - *Detail of patterns' outer circle*

For additional security the outer circle should have a further enveloping area of white so the whole outer circle and its contents remain isolated from the ground.

All this can be graphically viewed in figure `26` or in annex A.

What about overlapping patterns?
A particular situation may arise when using robots smaller than the patterns attached to them. Even if the robots are not that close together (given their size) it will result in an unacceptable position as the patterns will overlap with each other, effectively causing at least one of them to not be identified properly. There is not much that can be done regarding this, the best solution being either using smaller robots or further reducing the size of patterns.

Features extraction

Features extraction is a technique consisting of drastically reducing the amount of information to a small in comparison set of features. Image processing can benefit greatly from it as the input data it usually handles is usually large and redundant, making it suitable to be transformed into a set where only the meaningful data and characteristics are stored. If chosen wisely, they should prove to be a source of relevant information for the task at hand, using a reduced representation instead.

Example

For instance, to illustrate the concept applied to image processing let's consider a picture like the one presented in section *Thresholding* and figure 27.

Figure 27 - Thresholded image showing rice shapes

The image has been thresholded and the silhouettes of the rice grains can be seen. If we were interested in, say, find out information on thegrains we could apply a blob extraction algorithm which would create uniquely labeled sets of connected components. In this case, the connection or property between the pixels conforming a grain of rice is sharing the same color (white) and being isolated from other grains' pixels by a different color (black). Each one of these sets is a blob and each one of them may be processed separately – for instance, we could count them up or compute their area, resulting in synthesized information from the image.

Classification of techniques

Feature extraction techniques can be classified by their complexity, by the kind of characteristics they look for...

Low level: Edge detection

An edge is not a physical entity, the same way shadows are not, but rather the points where the picture ends and the wall starts; more formally, it is a set of points where their brightness is estimated to vary sharply. This can also be expressed by stating the image is a 2D function and edges are variations which at some points render it discontinuous. Edges are assumed to be pointing in one of a variety of directions.

Canny edge detector

The canny edge detector uses an edge detection operator like for instance Sobel. This operator returns values both for the first derivative in the horizontal and vertical direction, from which gradient and direction can be computed with the formulas in

figure 28.

$$\mathbf{G} = \sqrt{\mathbf{G}_x{}^2 + \mathbf{G}_y{}^2}$$
$$\mathbf{\Theta} = \arctan\left(\frac{\mathbf{G}_y}{\mathbf{G}_x}\right)$$

Figure 28 - *Formulas for gradient and direction*

The resulting direction must be approximated to 1 of 8 pre-established directions like those which 4 axes afford. It is important to note how these angles represent vertical, horizontal and diagonal directions, though as a whole they may be rotated (and therefore not necessarily at 0, 45, 90 or 135 degrees).

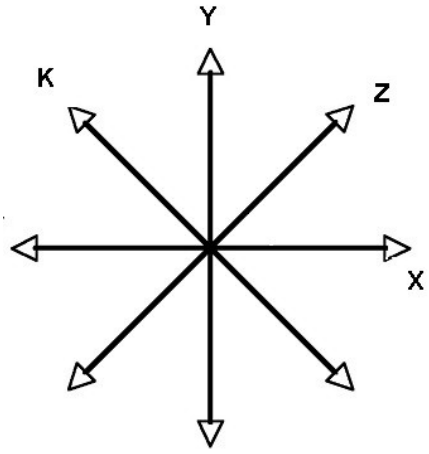


Figure 29 - *4 axes, 8 directions*

The gradient magnitude is then checked in the rounded up direction against those directions conforming a 90º degrees angle with it. For instance, if the gradient direction is 45º, the magnitude in the positive way of the Z angle will be compared to to that of both the positive and negative ways of the K angle. If it is greater than the these two other values separately, the examined area (a pixel for example) will be considered an edge.

Shape-based
Shape-based feature extraction techniques form subsets of pixels which when considered together satisfy certain geometrical conditions. The two examples we are interested and will next explain are blob extraction (grouping up neighbouring, similar pixels) and the least square fitting of ellipses (which deals with the best approximation of ellipses given an input binary image for the edges present in it).

Blob extraction
This technique uniquely labels all groups of neighbouring "object" pixels, grouping them into blobs or connected regions. The algorithm consists of two steps.

- Iterating through each of the pixels, we look for one which is "object" and has **not** been labeled yet. We will denominate it the *seed* pixel, and once found, we assign to it a new label and proceed to step 2. Once every "object" pixel in the image owns a label the algorithm concludes.

- Using a filling algorithm, we will assign this new label to all pixels neighbour to

the *seed* pixel. This part will end when we run out off for expanding (that is, the whole of the blob is surrounded by "background"), returning to part 1.

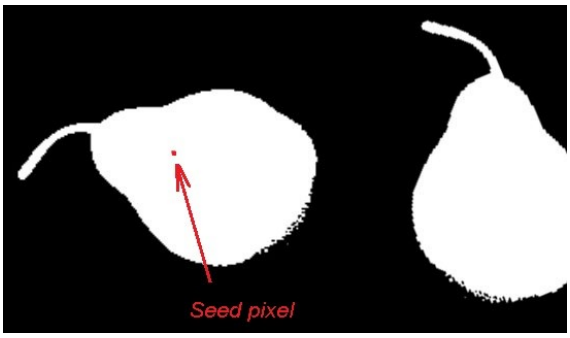


Figure 30 - *2 blobs in an image*

In figure `30`, all pixels conforming the left pear could have a label value of `1` while the one in the right would have value `0`.

When applying the blob extraction algorithm for our purposes a few features should be extracted from those detected. For example:

- Centre of gravity. In the case of a ring-shaped blob (like those acquired when segmenting a concentric circle) it would roughly correspond to the circle's centre.
- Bounding box. That is, the maximum coordinates for both axes present in the pixels set minus the minimum coordinates. This is a rough estimation of the blob's size, from which a radius can be inferred if dealing with ring-shaped blobs.

Figure 31 - *Blob bounding box*

Least-square fitting of ellipses

This technique aims at adjusting the parameters of a model function with the form

$$f(x, \beta)$$

to best fit a data set. In our case the function to adjust is that of a general ellipse, that is, a group of points with constant distances to `2` fixed points, rotated a counter-clockwise α angle and with a and b as its axes lengths.

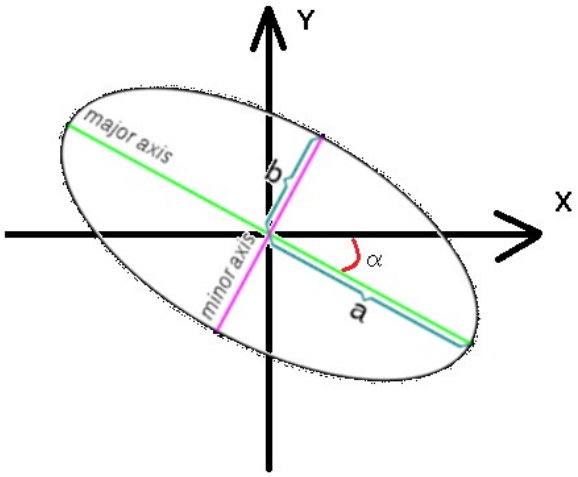


Figure 32 - *General ellipse*

$$\frac{(x\cos\alpha + y\sin\alpha)^2}{a^2} + \frac{(x\sin\alpha - y\cos\alpha)^2}{b^2} = 1$$

As we have stated, an image and in particular a gray-scaled one can be interpreted as a 2D function with the form `f(x, y)`, where the result for integer values of both `x` and `y` is the brightness in pixel `(x, y)`. If a thresholding has been applied to this image we can consider the pixels as a data set of `n` points (X_i, C_i), with `i = 1 .. n` and C_i the coordinates of the pixel.

A residual r_i will be defined by the difference between the expected distance to the ellipse's corresponding point and that offered by the pixel coordinates.

$$r_i = C_i - f(x_i, \boldsymbol{\beta})$$

The optimum least-square approximation will be that which minimizes the sum of these residuals, hence best fitting the desired function.

$$S = \sum_{i=1}^{n} {r_i}^2$$

This is the technique we will be using in this report, as it proved more efficient to directly detect ellipses from the circles and radius present in the patterns than finding their regions and then figuring out to which of them they correspond. It is implemented in the *OpenCV* function `cvFitEllipse2d()`.

Tools and libraries

Much of the work carried out by the project deals with coordinating external, previously developed work. The main tools employed during the development stage were:

Microsoft Visual Studio 2008

Visual Studio is an IDE or Integrated Development Environment for Windows Systems. Several programming languages are supported, like Visual Basic, C#, etc. though the entirety of this project will be written in C++. With many features to make it appealing as a good candidate, its

integrated debugger along the many capabilities it offers is particularly worth mentioning.

MFCs

The Microsoft Foundation Classes or MFCs conform a library with the objective of wrapping the Windows API in C++ classes. It is particularly easy with MFCs to create Windows-like interfaces by means of the classes it provides and their respective methods. Many of Windows controls also have got their respective classes. The typically found controls palette can be observed in figure #.

Figure 33 - *MFCs controls palette*

- A group box is a static control used to consider as a whole a group of other, related controls.
- Buttons are used to initiate an action. A button is useful when clicked, positioning the mouse over it.
- A radio button is a Windows control made of two sections: a round box O and a label. In practice two or more radio buttons are required to be of any use, with only one of them selected at any given time.
- An edit box is a control used to either display text, request it, or to do both. It it provided as a rectangular control with a sunken white background and 3-D borders. We shall only use them for displaying text information, keeping them non-editable.

The version used is MFC `9.0.30729.5570`, released in April `2011` and shipped with SP1 for Visual C++ `2008`.

Linked libraries

A small number of libraries are required as well during both compilation and execution. For the latter they will need to be present in either the execution directory or the OS library path. During development it was necessary to tell VS`2008` to dynamically link with them (Project properties -> Configuration properties -> Linker -> Input -> Additional dependencies).

`libcurl.lib`	Retrieves internet objects given a resource URL. With *WebcamXP* the server is simulated locally.
`cv210.lib` **`cv0aux210.lib`** **`cxcore210.lib`** **`highgui210.lib`**	Required for OpenCV capabilities usage.
`libsasl.dll` **`openldap.dll`** **`zlib1.dll`**	Necessary for HTTP fetching. *curl* library relies on these ones.

Table 5 – *Required libraries*

OpenCV library

OpenCV, or Open Source Computer Vision Library encompasses a set of programming functions dealing with real time computer vision. It was originally launched by Intel in 1999 and officially released at the IEEE Conference on Computer Vision and Pattern Recognition in 2000. Originally written in C, since version 2.0 it incorporates a new C++ interface. It has many applications in mobile robotics, providing toolkits useful for the field as we shall see in the *Practical Approach* section.

Installing OpenCV library

Latest version for windows can be found on

http://sourceforge.net/projects/opencvlibrary/files/opencv-win/

It consists of a simple executable file which will handle the process for us.

Webcam XP

Webcam XP handles connections to remote web cameras integrating a server responding to HTTP requests. The server is simulated locally, allowing to retrieve captured images accessing the 127.0.0.1 or localhost address.

Installing and connecting to a web camera

An installer can be downloaded from

http://www.webcamxp.com/download.aspx

The free version only allows connections to be established with one single camera, which suits us just fine as such is the way we will be working. Its interface is shown in figure 34.

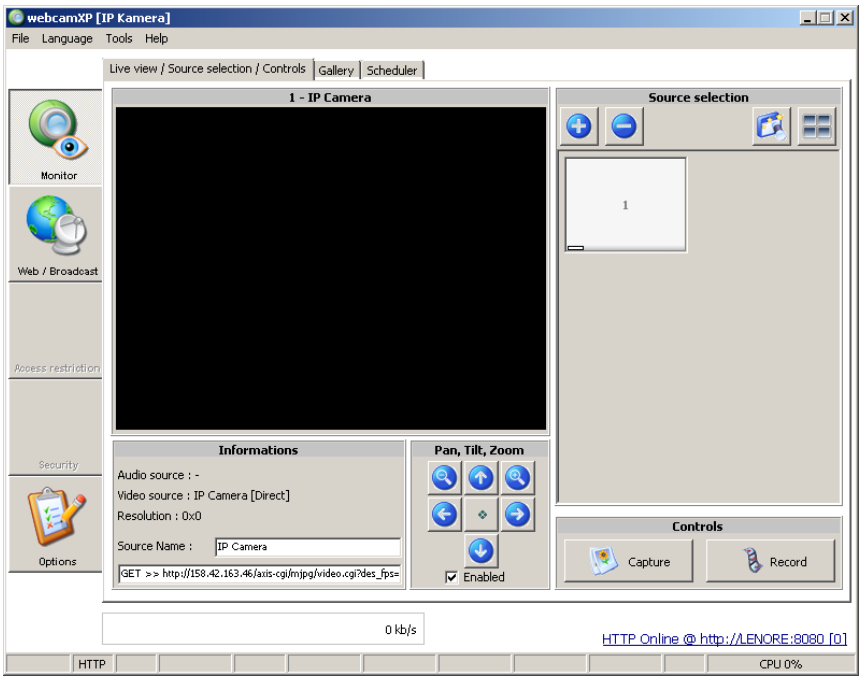


Figure 34 - *Webcam XP interface*

Most of the features it provides remain unused here, as our main concern is the obtention of images. The process to connect to a camera is relatively simple: on the right tab we select one of these available slots and fill its properties page with:

- Web camera vendor and model.
- Web camera IP address.
- User & Password, if any.
- Source or local name.

From that point on and as long as Webcam XP keeps executing the images will be accessable in the camera's default path. In the case of *AXIS* camera the path will be:

`http://localhost:8080/cam_1.jpg`

Where *cam_1.jpg* will be the source or local name we used in the previous steps.

Version `5.5.0.8` was used.

Logitech pro `9000` drivers

These drivers handle the connection and data transfer between the PC and the Logitech camera via the USB port.

The drivers are available for download here:

http://www.logitech.com/es-es/support-downloads/downloads/business-products/devices/4597?WT.z_sp=Model

Again, instalation will be handle for us.

HTTP fetching *curl* library

libcurl is a free client-side URL transfer library, supporting FTP and HTTP among others, user+password authentication, file transfer resume, etc. In our thesis it is used as the coded way to access the simulated local server in our machine for accessing images. A command-line utility is also available, though it would not be particularly useful here.

Installing *libcurl*

curl is available for download in the following internet address:

http://curl.haxx.se/download.html

For this thesis *curl* version `7.21.6` was used. The Windows executable download will properly set up the library.

Code functions tree

cam.h
cam.cpp

```
size_t    cam_writeData (void*, size_t, size_t, FILE*);
int       cam_initialize (void);
void      cam_shutdown (void);
int       cam_shoot (IplImage **gray, IplImage **color);
```

procells.h **procells.cpp**	void	**ells_segment** (void); **ells_shutdown** (void); **ells_initialize** (void); **ells_properSize** (ellipse*); **ells_fillEllipseData** (int, CvSeq*);
procii.h **procii.cpp**	IplImage*	**pii_processFrame** (void); **pii_threshold** (IplImage*); **pii_normIntensity** (IplImage*); **pii_bgroundGetMean** (void);
robs.h **robs.cpp**	void	**robs_extract** (void); **robs_concentric** (ellipse*, ellipse*); **robs_initialize** (void); **robs_shutdown** (void); **robs_setIDFromNSubelipses** (int); **robs_markRobotsLocations** (IplImage*); **robs_fillRobotData** (int, int*, int);
pfc.h **pfc.cpp**	void	**showMessage** (int, int); **initialize_all** (void); **shutdown_all** (void);
PFC_VS8Dlg.h **PFC_VS8Dlg.cpp**	void	**dlg_initialize** (void); **dlg_showVideo** (void); **dlg_showFoto** (void); **dlg_cycleThroughStuff** (void); **dlg_cleanEllipseData** (void); **dlg_showNumericalData** (void);

Table 6 – *Code functions tree*

Practical approach

This section will deal with the most code-oriented part of the project, dwelling into how it is organized and its general structure. We will start explaining some general features such as how images are captured, how they are displayed and some protocols and other details to keep in mind.

Acquiring images

PTZ-212 AXIS (webcamera) and Logitech Pro9000 camera models were used. First thing to do is creating a connection to the webcam, then retrieving images at a certain rate.

Accessing the camera

Two different kinds of cameras are taken into consideration: webcam and USB-connected.

Web cameras

In the first case, the camera should allow snapshots to be taken accessing a certain file on the server it provides, without any particular security or identification required. Hence *curl* library, by which we shall fetch the objects with an HTTP request, will need to access the following URL:

```
#define CAM_BITMAP_URL "http://158.42.163.46/bitmap/image.bmp"
```

Where the shown IP address is that which was used during development. However, *Webcam XP* software which will allow us to simulate the server on our computer, besides some other additions. The accesses will therefore take place in *localhost*, as if downloading a local JPG file:

```
#define CAM_JPG_URL "http://localhost:8080/cam_1.jpg"
```

Webcam XP just requires to be running when downloading the files, as the connection needs to be kept. The webcam IP is fed to this software, as well as the image filename (`cam_1.jpg`, here).

Camera initialization is pretty straightforward and is packed into `cam_initialize()`. A handle of type `CURL*`, named `curl`, is created for ease of use as well as a file descriptor onto which dump images:

```
FILE *camFD = NULL;
CURL *curl = NULL;
...
void cam_initialize (void) {
      curl = curl_easy_init ();
      curl_easy_setopt (curl, CURLOPT_URL, CAM_JPG_URL);
      curl_easy_setopt (curl, CURLOPT_WRITEFUNCTION, cam_writeData);
      curl_easy_setopt (curl, CURLOPT_WRITEDATA, camFD);
}
```

This results in `CAM_JPG_URL`, `camFD` and `cam_writeData()` associated to `curl` as the resource URL, file descriptor and writing function it expects, respectively.

`cam_writeData()` is simple, its header is specified by *curl*:

```
size_t cam_writeData (void *ptr, size_t size, size_t nmemb, FILE *stream)
{
```

```
        // stream remains unused
        return fwrite (ptr, size, nmemb, camFD);
}
```

Releasing all resources is done easily by calling *cam_shutdown*() which, in turn, calls *curl*'s shutdown function and frees the file descriptor, should it have not been properly closed earlier:

```
void cam_shutdown (void) {
        curl_easy_cleanup (curl);
        if (camFD) fclose (camFD);
}
```

In summary, a picture is retrieved from `CAM_JPG_URL` and written to the file pointed by `camFD` via the `cam_writeData()` function.

USB cameras
Regarding USB-connected cameras, the initialization is much shorter, as we get *OpenCV*'s help:

```
CvCapture *capture;
. . .
void cam_initialize (void) {
        if ((capture = cvCaptureFromCAM (0)) == NULL) {
                showMessage (PFC_MESS_CVCAPTUREFROMCAM, false);
                return PFC_BAD;
        }
}
```

Same happens with their shutdown function:

```
void cam_shutdown (void) {
        cvReleaseCapture (&capture);
}
```

Preparing images storage

OpenCV offers an easy interface to store images via an `IplImage*` structure, relevant fields of which are:

• `int nChannels` • `int depth`	Represent, respectively, number of color components per pixel and the size, in bits, of each one of them. Product of both yields the memory required to store a single pixel.
• `int height` • `int width` • `int imageSize`	Size, in pixels. `height * width = imageSize`
• `int widthStep`	Offtset, in bytes, from one pixel to the one immediately beneath; or, amount required to represent a full horizontal line of pixels.
• `char* imageData`	Pointer to memory location holding start of image.

Table 7 – *IplImage structure*

Accessing stored images

Each time `cam_shoot()` is invoked `curl_easy_perform()` will be too in turn for webcameras and retrieve a frame. In webcameras, as we will be fetching an image object, data will be channeled to `"color.jpg"`, using `camFD` file descriptor (which needs to be open/closed separately). It is then read both as a color and a gray-scale image into the corresponding variables and further processed from that point:

```
int cam_shoot (IplImage **gray, IplImage **colour) {
	camFD = fopen ("color.jpg", "wb");
	curl_easy_perform (curl);
	fclose (camFD);
	IplImage *gray = cvLoadImage ("color.jpg", CV_LOAD_IMAGE_GRAYSCALE);
	IplImage *color = cvLoadImage ("color.jpg", CV_LOAD_IMAGE_COLOR);
}
```

Regarding USB-connected cameras, the frame obtained is directly stored in an `IplImage` structure, so no access to disk is requierd:

```
int cam_shoot (IplImage **gray, IplImage **colour) {
	IplImage *colour;

	. . .

	colour = cvQueryFrame (capture);
	gray = cvCreateImage (cvSize (g_img_w, g_img_h), 8, 1);
	cvCvtColor (colour, gray, CV_BGR2GRAY);
}
```

Finally the gray-scaled image, however it was obtained, is equalized to improve quality.

```
cvEqualizeHist (gray, gray);
```

Displaying images

Choosing a destination window and dumping onto it the data from the desired image buffer is enough. The following line of code creates a window called `"umbral"`:

```
cvNamedWindow ("umbral", CV_WINDOW_AUTOSIZE);
```

And the following one dumps the image data stored in `IplImage *umbral` to that window:

```
cvShowImage ("umbral", umbral);
```

The result is a window titled `"umbral"` where the image stored, assuming it contains valid data, will be displayed. This is quite versatile, as replacing `umbral` for any other `IplImage*` variable, `color` for instance, would instead show the corresponding frame.

Operation modes

Two operation modes are allowed during application execution:

- *Photograph* mode
- *Video* mode

First one obtains a single frame and offers as much information as can be obtained from it after processing. Second one is pretty much the same; however, processing is carried out over a sequence of frames and only important information (namely, final pattern coordinates and orientation) is displayed, graphically. Typically, the former will be used to fine-tune parameters, depending on light conditions, etc. and the latter will employ the so obtained values.

Both modes allow interaction – on *Photograph* ellipses and patterns are treated as lists for the user to browse the details of each one separately, while *Video* just offers the resulting location data for each robot in a graphical way. Please refer to sections *Button "Foto!"* and *Button "Video!"* for further details and particularly for a list of key bindings.

Processing loop

The pattern-detection loop will be applied once to every captured frame, and its basic outlay is this:

```
IplImage *pii_processFrame (void) {
        Capture a frame. -- cam_shoot ()
        Preprocessing: intensity normalization, thresholding. -- pii_intensity(),
                                                                 pii_threshold()
        Detecting ellipses. -- ells_extract()
        Pattern detection based on the obtained set of ellipses. -- robs_segment()
        Displaying information graphically. -- robs_markRobotsLocations()
}
```

Before any of this is carried out, however, several general initializations are required to take place with a call to `initialize_all()`, which shall be invoked upon application start.

```
void initialize_all (void) {
      ells_initialize (); // ellipses
      robs_initialize (); // patterns/robots
      cam_initialize  (); // camera
}
```

It handles the rest of the project's modules, the structures holding the retrieved information in particular. They follow here, along with these structures' definition; ellipses...:

```
#define MINIMUM_AREA_I_INITIAL 6
. . .
struct ellipse {
      IplImage *img;
      CvPoint center;
      CvSize size;
      float angle;
      int meanSize; // Media de los dos radios.
};
. . .
struct ellipse *g_ells;
int g_ells_num;
int g_ells_actual;
. . .
void ells_initialize (void) {
```

```
	g_ells = NULL;
	g_ells_actual = g_ells_num = 0;
}
```

... and patterns:

```
#define MAX_ROBOTS 6
. . .
struct robot {
	int *ells;
	int n_ells;
	float angle;
	int ID;
	CvPoint center;
	double radius;
	IplImage *img;
};
. . .
int g_robs_num;
int g_robs_actual;
struct robot g_robs[MAX_ROBOTS];
. . .
void robs_initialize (void) {
	g_robs_actual = g_robs_num = 0;
	for (int i = 0; i < MAX_ROBOTS; i++) {
		g_robs[i].ells = NULL;
		g_robs[i].n_ells = 0;
		g_robs[i].angle = 0;
		g_robs[i].center = cvPoint (0, 0);
		g_robs[i].radius = 0;
		g_robs[i].ID = -1;
	}
}
```

Also, their shutdown functions are like this:

```
void ells_shutdown (void) {
	if (g_ells != NULL) {
		free (g_ells);
	}
	g_ells_num = g_ells_actual = 0;
}
```

```
void robs_shutdown (void) {
	for (int i = 0; i < MAX_ROBOTS; i++) {
		g_robs[i].n_ells = 0;
		g_robs[i].angle = 0;
		g_robs[i].ID = -1;
		g_robs[i].center = cvPoint (0, 0);
		g_robs[i].radius = 0.0;
		if (g_robs[i].ells != NULL) {
			free (g_robs[i].ells);
			g_robs[i].ells = NULL;
		}
	}

	g_robs_num = g_robs_actual = 0;
}
```

Finally, `shutdown_all()` will take care to release all resources by calling the respective module

shutdown functions. Here, `fondo` represents the background image taken with no robots present.

```
void shutdown_all (void) {
	cam_shutdown ();
	robs_shutdown ();
	ells_shutdown ();
	if (fondo) cvReleaseImage (&fondo);
}
```

Image acquisition

We need to store two versions of the captured image: the colour one and the gray-scaled one. `cam_shoot()` makes this task easy:

```
IplImage *gray, *colour;

if (cam_shoot (&gray, &colour) == PFC_BAD) { // guarda en fichero.
	return NULL;
}
```

Preprocessment

This stage encompasses intensity normalization and thresholding the image to a binary black/white format on which to detect objects.

A more in-depth explanation of both follows:

pii_normIntensity – IplImage* **pii_normIntensity** (IplImage ***gray**)
`fondo` represents a background image loaded either at application startup, using the last acquired background image, or when button "Fondo" is pressed (see the *Button "Fondo"* section). Upon load its mean intensity is computed and stored into `int_media`. The following algorithm computes a set of per-pixel ratios for the background image which, when multiplied, would make the values for each one equal to this mean.

```
IplImage *fondo;
. . .
IplImage *pii_normIntensity (IplImage *gray) {
	if (fondo) {
		float f, u;
		IplImage *a = gray;
		for (int i = 0; i < g_img_h; i += 1) {
			for (int j = 0; j < g_img_w; j += 1) {
				f = (fondo->imageData + i*fondo->widthStep)[j];
				u = (a->imageData + i*a->widthStep)[j];
				u *= (int_media / f);		// ratio
				if (u > 255.0) u = 255.0;
				(a->imageData + i*a->widthStep)[j] = u;
			}
		}
	}
}
```

pii_threshold – IplImage* **pii_threshold** (IplImage ***in**)
This function returns the binarized input image.:

```
#define GEN_THRESHOLD_I_INITIAL 190
#define GRAY_MAX_VALUE          255
float g_gen_threshold = GEN_THRESHOLD_I_INITIAL;
```

Figure 35: *pii_threshold()*

```
. . .
IplImage *pii_threshold (IplImage *in) {
      IplImage* out;
      out = cvCreateImage (cvGetSize (in), 8, 1);
      cvThreshold (  in, out, g_gen_threshold,
                     GRAY_MAX_VALUE, CV_THRESH_BINARY);
      return out;
}
```

The resulting image will still have an 8-bit depth, even if only two values are required from now on in terms of representation (255 for white or «background», 0 for black or «object»). In short, the other 7 bits per pixel are wasted.

Information extraction

Two more blocks of code –ellipse and robot/pattern detection– are called last within pii_processFrame().

Ellipse recognition

A set of ellipses in the image will be identified via *OpenCV*. The resulting is stored for artifact recognition, where concentric sets mark pattern central coordinates..

As opposed to the mathematical model, *OpenCV* ellipses only possess one center, height and width (*h* and *w* in Figure 36, respectively) and an orientation angle *α* against the Y vertical axis. In the figure's generic example α equals roughly to 90 + 45 = 135 degrees.

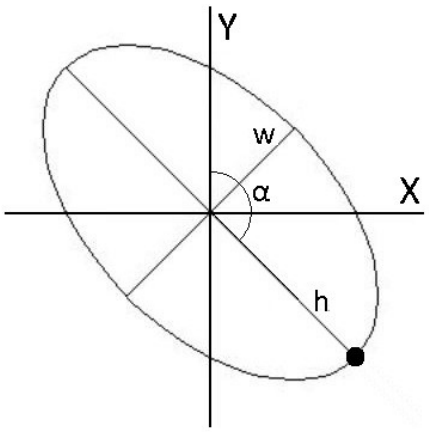


Figure 36 - *An ellipse*

ells_segment() coordinates de identification tasks and dumps the extracted information in the designated areas, i.e. g_robs[] and g_ells[]:

```
void ells_segment (IplImage *umbral) {
      CvMemStorage *storage = cvCreateMemStorage (0);
      CvSeq *contour = cvCreateSeq (   CV_SEQ_ELTYPE_POINT,
                                       sizeof (CvSeq),
                                       sizeof (CvPoint), storage   );
      cvFindContours (   umbral, storage, &contour, sizeof (CvContour),
                         CV_RETR_LIST, CV_CHAIN_APPROX_NONE,
                         cvPoint (0, 0)    );
      CvSeq **auxContour = (CvSeq **) malloc (10 * sizeof (CvSeq *));
      for (CvSeq *iter = contour; iter ; iter = cont_iter->h_next) {
            if (iter->total >= g_ells_area_minima) {
                auxContour[g_ells_num] = iter;
                g_ells_num++;
                if (g_ells_num % 10 == 0) {
                   auxContour = (CvSeq **) realloc (  auxContour,
```

```
                                                        (g_ells_num + 10) *
                                                        sizeof (CvSeq *));
                }
            }
        }
        auxContour = (CvSeq **) realloc (  auxContour,
                                           g_ells_num * sizeof (CvSeq *));
        g_ells = (struct ellipse *) malloc (g_ells_num * sizeof (struct ellipse));
        for (int i = 0; i < g_ells_num; i++) {
            ells_fillEllipseData (i, &auxContour[i]);
        }

        cvReleaseMemStorage (&storage);
        free (auxContour);
}
```

As can be seen the procedure consists in extracting the contours from the image, count up those which actually have a chance of turning into an ellipse, then gather their information in `g_ells[]` by means of successive calls to `ells_fillEllipseData()`, which will actually dump it all onto the structure. The number of valid ellipses (that is, those where the originating contour number of pixels are greater than the specified value in `g_area_minima`) is counted up in `g_ells_num`; the rest are discarded. This data is of mostly numerical nature:

```
void ells_fillEllipseData (int i_ells, CvSeq **contour) {
        CvMat *p_f = cvCreateMat (1, (*contour)->total, CV_32FC2);
        CvMat p_i = cvMat (1, (*contour)->total, CV_32SC2, po_f->data.ptr);
        cvCvtSeqToArray (*contour, p_f->data.ptr, CV_WHOLE_SEQ);
        cvConvert (&p_i, p_f);
        CvBox2D box = cvFitEllipse2 (p_f);
        g_ells[i_ells].center = cvPoint (   cvRound (box.center.x),
                                            cvRound (box.center.y)   );
        g_ells[i_ells].size = cvSize (      cvRound (box.size.width *0.5),
                                            cvRound (box.size.height *0.5));
        if (box.size.width > box.size.height) {
            g_ells[i_ells].sizeRatio = box.size.width / box.size.height;
        } else {
            g_ells[i_ells].sizeRatio = box.size.height / box.size.width;
        }
            g_ells[i_ells].angle = box.angle;
        g_ells[i_ells].meanSize = cvRound (  (g_ells[i_ells].size.height +
                                              g_ells[i_ells].size.width)
                                              * 0.5   );
        cvReleaseMat (&p_f);
}
```

Data returned by `cvFitEllipse2()` assumes the returned size of the ellipses' axes in `box` at their full length; however, for our computations employing "radius" lengths is more convenient, thus the values are halved. `.meanSize` is just the mean of both radiuses and is used to display an approximate version of the pattern by means of simply a circle, later on.

Pattern recognition

With the set of ellipses obtained what is left to do is grouping those which share very nearby centers, with two or more of them present meaning the presence of a pattern, and finding out its orientation. This is done in function `robs_extract()`:

```
void robs_extract (void) {
	bool *taken = (bool *) malloc (g_ells_num * sizeof (bool));
	for (int i = 0; i < g_ells_num; i++) taken[i] = false;
	int *aux_list = (int *) malloc (g_ells_num * sizeof (int));
	for (int n = 0; n < g_ells_num; n++) {
		int aux_n = 0;
		if (taken[n] == true) continue;
		taken[n] = true;
		if (ells_properSize (&g_ells[n]) == false) continue;
		for (int nn = 0; nn < g_ells_num; nn++) {
			if ( (n != nn) && (taken[nn] == false) &&
				(robs_concentric (&g_ells[n], &g_ells[nn]) &&
			(ells_ells_properSize (&g_ells[nn]) == true)) {
				taken[nn] = true;
				aux_list[aux_n] = nn;
				aux_n++;
			}
		}
		if (aux_n > 1) {
			int aux_n_biggest = -1;
			for (int i = 0; i < aux_n; i++) {
				if (  g_ells[aux_list[i]].meanSize >
					g_robs[g_robs_num].radius) {
					g_robs[g_robs_num].radius =
					g_ells[aux_list[i]].meanSize;
					aux_n_biggest = i;
				}
			}
			if (g_robs[g_robs_num].radius < 21) continue;
			robs_fillRobotData (n, aux_ell_list, aux_n);
			g_robs_num++;
			if (g_robs_num == MAX_ROBOTS) {
				return;
			}
		}
	}
	free (taken);
	free (aux_ell_list);
}
```

As data is gathered it is stored in the temporal structure `aux_ell_list` on which `aux_n` is a counter, representing all the ellipses associated to a certain one under current study. Thus, through each iteration a value for `aux_n` greater than `0` means a robot has been found (i.e. at least another ellipse is concentric with it) and `g_robs_num` is updated accordingly – this variable is also used to track which pattern position to write on.

`taken[]` keeps track of which ellipses have already been checked and/or are part of a robot along one or more concentric additional ellipses, so they can be skipped in later loop iterations. There also exist size restrictions, checked separately; therefor, to be valid, an ellipse must successfully pass these two function guards, `robs_concentric()` and `ells_properSize()`:

```
#define CONCENTRIC_SLACK_X 2
#define CONCENTRIC_SLACK_Y 2
. . .
bool robs_concentric (ellipse *e, ellipse *ee) {
      int dif_x = abs (e->center.x - ee->center.x);
      int dif_y = abs (e->center.y - ee->center.y);

      return (  (dif_x <= CONCENTRIC_SLACK_X) &&
                (dif_y <= CONCENTRIC_SLACK_Y)    );
```

The maximum and minimum values for the following checks have been obtained empirically, taking into account the size of the patterns and the distance from the floor to the camera:

```
bool ells_properSize (struct ellipse *e) {
      bool ok = true;

      if ((e->size.height <= 1) || (e->size.width <= 1)) {
            ok = false;
      }
      if (e->meanSize > 31) {
            ok = false;
      }
      if (e->sizeRatio > 1.3) { // deben ser redondas, aprox.
            ok = false;
      }
      return ok;
}
```

Finally, `robs_fillRobotData()` is in charge, in the first place, of dumping the so obtained information to its corresponding place in the designated structures, and second of finding out its orientation angle by guessing which is the correct ellipse to examine in order to do so (i.e. the ellipse corresponding to the 'radius' on the patterns):

```
void robs_fillRobotData I (int n, int *aux_ell_list, int aux_n) {
      aux_n++;
      g_robs[g_robs_num].ells = (int *) malloc (aux_n * sizeof (int));
      g_robs[g_robs_num].n_ells = aux_n;
      robs_setIDFromNSubellipses (g_robs_num);
      g_robs[g_robs_num].center.x = g_ells[n].center.x;
      g_robs[g_robs_num].center.y = g_ells[n].center.y;

      g_robs[g_robs_num].ells[0] = n;
      for (int i = 1; i < aux_n; i++) {
            g_robs[g_robs_num].ells[i] = aux_ell_list[i - 1];
      }
      . . .
}
```

Finding the pattern orientation remains yet to do, and is achieved in the last block of code by choosing that ellipse which center is within some boundaries (that is, by checking the related ellipse is centered in the area surrounding the centre of the pattern) while simultaneously **not** being concentric. Again, the size characteristics of such ellipse have

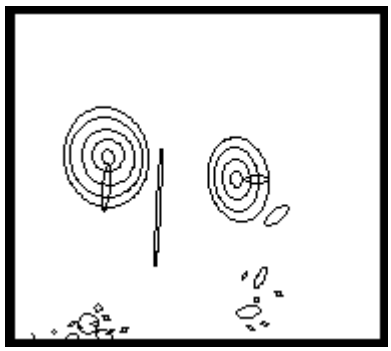


Figure 37: *Detected ellipses*

been found empirically, and the results can be observed in Figure 37. Also, `robs_setIDFromNSubellipses()` just takes care of substracting 2 from the number of subellipses conforming the robot to find its ID, clamping the values to [0, 5].

In the following second part of the function the ellipse determining the angle is found and the value adjusted.

```
void robs_fillRobotData II (int n, int *aux_ell_list, int aux_n) {
      . . .
      for (int a = 0; a < g_ells_num; a++) {
            if (a == n) continue;
            bool found = false;
            for (int aa = 0; aa < aux_n; aa++) {
                  if (aux_ell_list[aa] == a) {
                        found = true;
                  }
            }
            if (found) continue;

            // Además debe ser alargada...
            if (g_ells[a].sizeRatio < 2.0) {
                  continue;
            }
            // ...y no exceder ciertos tamaños.
            if ((g_ells[a].meanSize > 7)       ||
                  (g_ells[a].size.height <= 1) ||
                  (g_ells[a].size.width <= 1)) {
                  continue;
            }
            int desp_x = abs (   g_robs[g_robs_num].center.x -
                                 g_ells[a].center.x   );
            int desp_y = abs (   g_robs[g_robs_num].center.y -
                                 g_ells[a].center.y   );
            int d = (int) sqrt ((double)(desp_x*desp_x + desp_y*desp_y));
            if (   (d > (g_robs[g_robs_num].radius / 3)) &&
                  (d < ((2* g_robs[g_robs_num].radius) / 3))) {
                  g_robs[g_robs_num].angle = g_ells[a].angle;
                  // corregir angulo según el cuadrante de la elipse:
                  // Arriba izquierda
                  if (isUpLeft (g_ells[a].center.x, g_robs[g_robs_num]) {
                        g_robs[g_robs_num].angle += 180;
                  }
                  // Abajo izquierda
                  if (isDownLeft (g_ells[a].center.x, g_robs[g_robs_num]) {
                        if (g_robs[g_robs_num].angle < 90)
                              g_robs[g_robs_num].angle += 180;
                  }
                  // Arriba derecha
                  if (isUpRight (g_ells[a].center.x, g_robs[g_robs_num])) {
                        if (g_robs[g_robs_num].angle > 90)
                              g_robs[g_robs_num].angle -= 180;
                  }
                  // Abajo derecha
                  if (isDownRight (g_ells[a].center.x, g_robs[g_robs_num])) {

                  }
                  // Arriba centro
                  if (isUpCentre (g_ells[a].center.x, g_robs[g_robs_num])) {
                        g_robs[g_robs_num].angle -= 180;
```

```
                        }
                        // Abajo centro
                        if (isDownCentre (g_ells[a].center.x, g_robs[g_robs_num])) {
                              g_robs[g_robs_num].angle += 180;
                        }

                  }
            }
      }
```

The angle correction is summarized in figure 38:

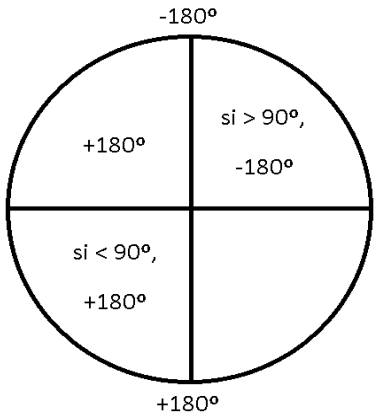
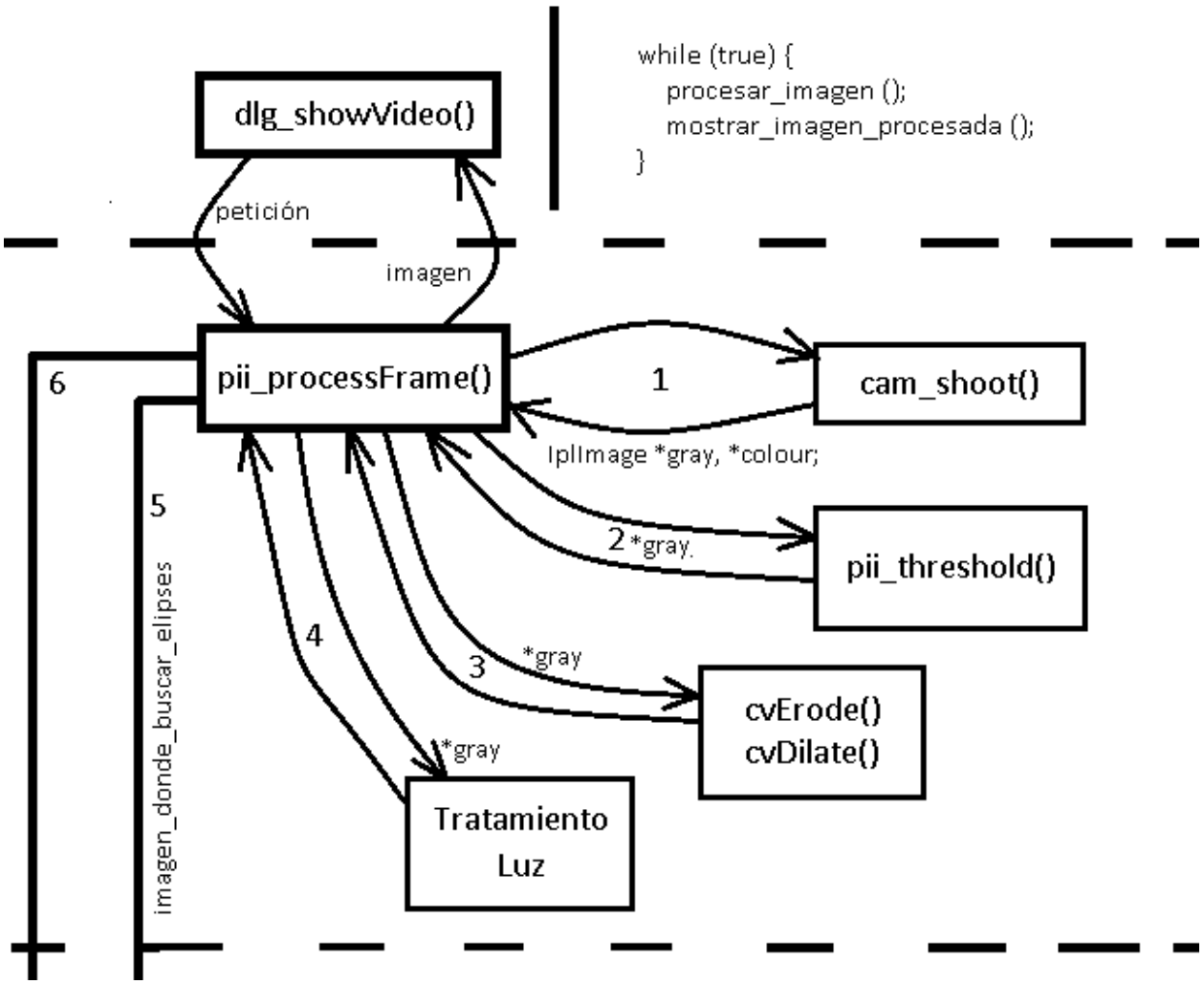
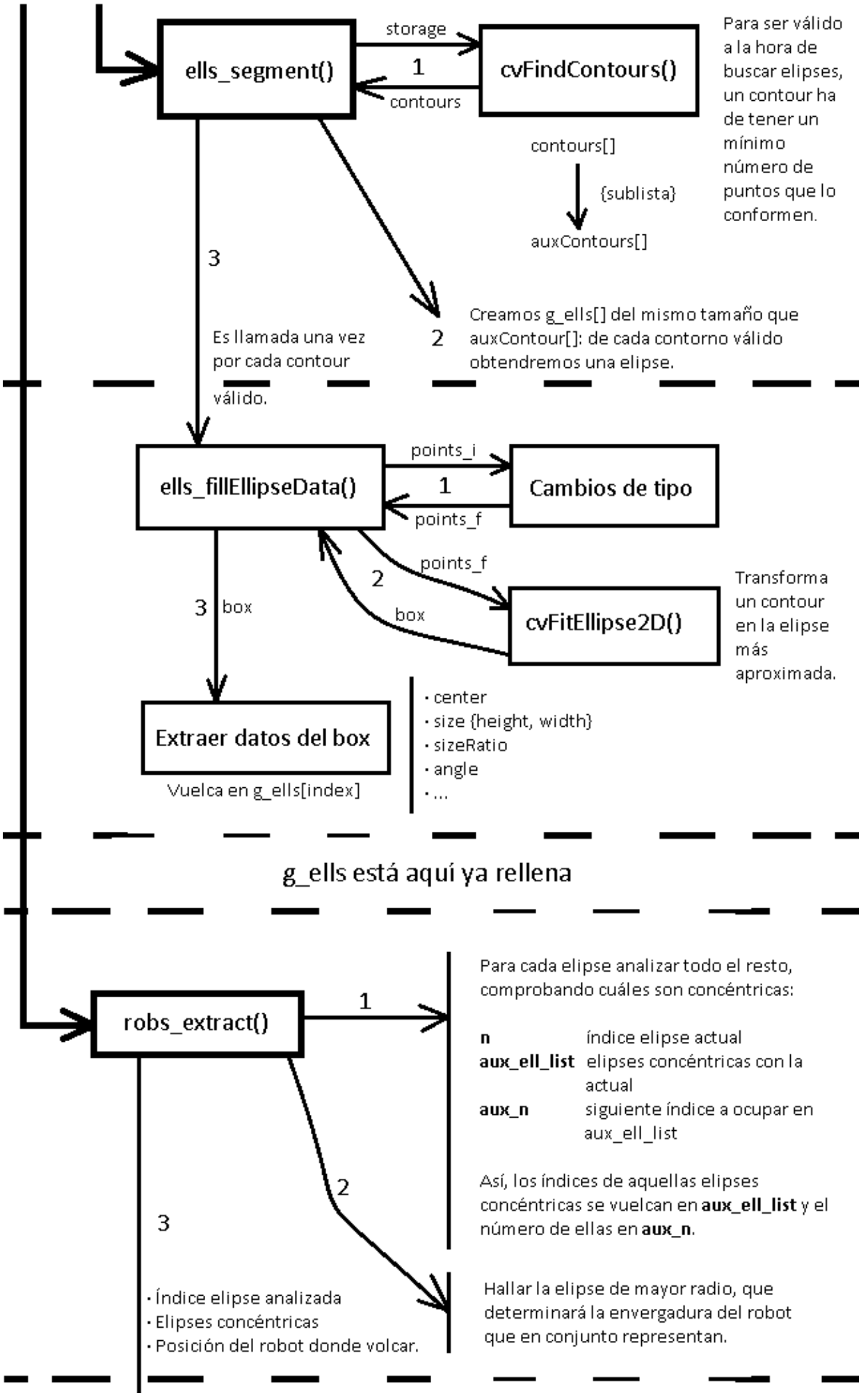
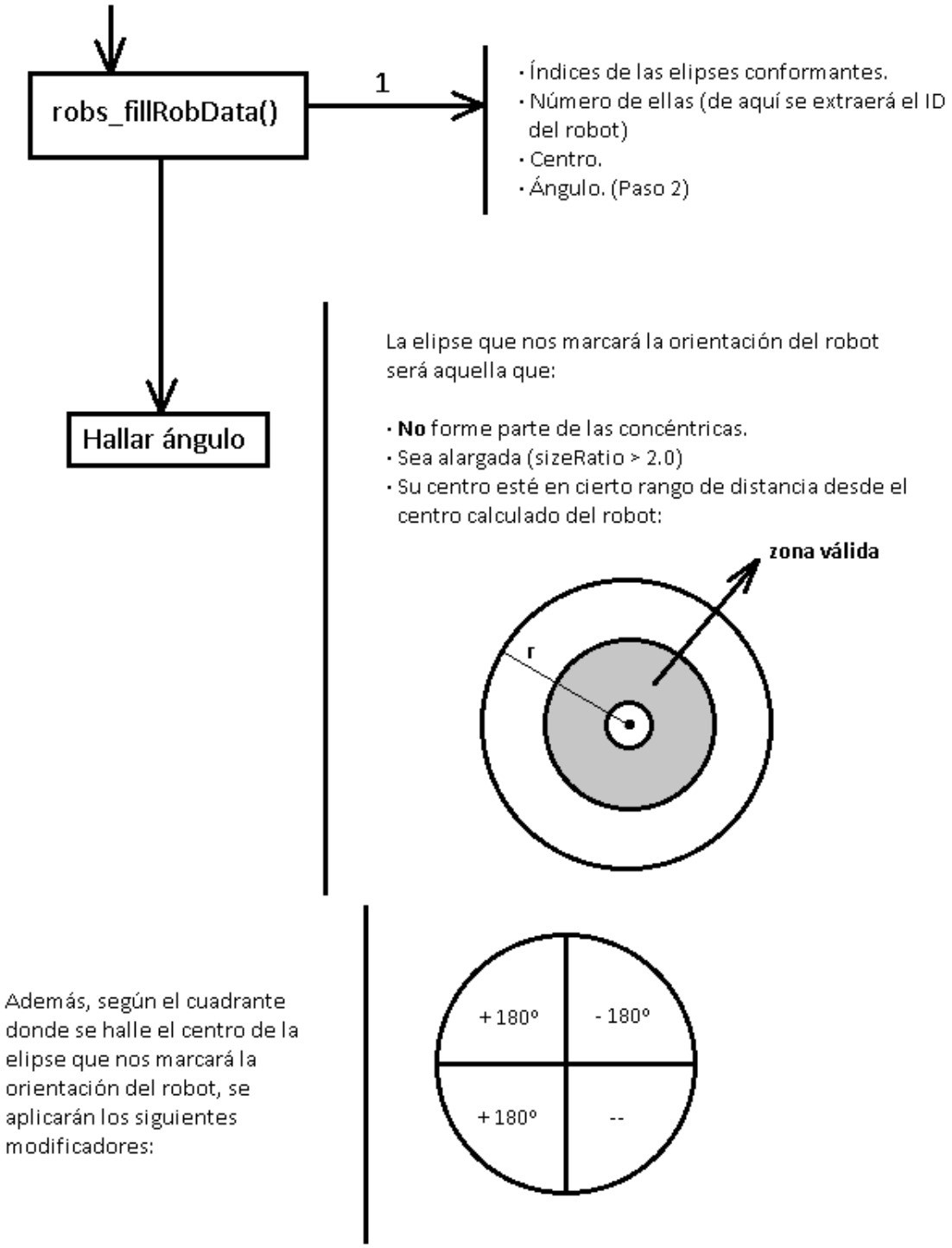


Figure 38 : *Angle correction*

Processing diagram




ells_segment()
storage
1
contours
cvFindContours()
Para ser válido a la hora de buscar elipses, un contour ha de tener un mínimo número de puntos que lo conformen.
contours[]
{sublista}
auxContours[]
3
Es llamada una vez por cada contour válido.
2
Creamos g_ells[] del mismo tamaño que auxContour[]: de cada contorno válido obtendremos una elipse.
ells_fillEllipseData()
points_i
1
points_f
Cambios de tipo
points_f
2
box
3
box
cvFitEllipse2D()
Transforma un contour en la elipse más aproximada.
Extraer datos del box
Vuelca en g_ells[index]
· center
· size {height, width}
· sizeRatio
· angle
· ...
g_ells está aquí ya rellena
robs_extract()
1
Para cada elipse analizar todo el resto, comprobando cuáles son concéntricas:
n índice elipse actual
aux_ell_list elipses concéntricas con la actual
aux_n siguiente índice a ocupar en aux_ell_list
Así, los índices de aquellas elipses concéntricas se vuelcan en aux_ell_list y el número de ellas en aux_n.
2
3
· Índice elipse analizada
· Elipses concéntricas
· Posición del robot donde volcar.
Hallar la elipse de mayor radio, que determinará la envergadura del robot que en conjunto representan.



Displaying obtained information

`robs_markRobotsLocations()` takes an input image and adds to it markers for both location and identification for each detected pattern. Regarding its usage a few things need to be set beforehand. *OpenCV* provides means to write text into an image – we are interested in writing the robot IDs, and hence define the text properties in order to actually do so on the image along their outlining circle:

```
void robs_markRobotsLocations (IplImage *out) {
      CString text;
      CvFont font;
      double hScale = 0.75;
      double vScale = 0.75;
```

```
        int lineWidth = 2;

        cvInit  &font, CV_FONT_HERSHEY_SIMPLEX | CV_FONT_ITALIC,
        Font (  hScale, vScale, 0, lineWidth      );

        for (int i = 0; i < MAX_ROBOTS; i++) {
              if (g_robs[i].ID != -1) {

                    CvPoint    aux_c    = g_robs[i].center.x + g_sciss_x,
                    cvPoint (             g_robs[i].center.y + g_sciss_y);

                    cvCi  out, aux_c, (int) g_robs[i].radius,
                    rcle  CV_RGB (255, 0, 0), 2   );
                    (

                    text.Format (CString ("%ld"), g_robs[i].ID);

                    cvPut out, (const char*) text.GetBuffer(),
                    Text  aux_c, &font, CV_RGB (0, 0, 255)   );
                    (

              }
. . .
```

The last step involves drawing the orientation-indicating line, which should lie on top of the physical pattern starting from its center and with a length equal to the mean value of the most external ellipses radiuses.

```
. . .
              double seno   = (double) sin (g_robs[i].angle*0.01745329251994);
              double coseno = (double) cos (g_robs[i].angle*0.01745329251994);

              CvPoint aux_c2;
              aux_c2 = cvPoint (g_robs[i].radius*seno, -g_robs[i].radius*coseno);
              aux_c2.x += g_robs[i].center.x;
              aux_c2.y += g_robs[i].center.y;

              cvLine (out, aux_c, aux_c2, CV_RGB (255, 0, 0), 2);
        }
}
```

An example of resulting image obtained when all 6 patterns are present is shown in figure 39:

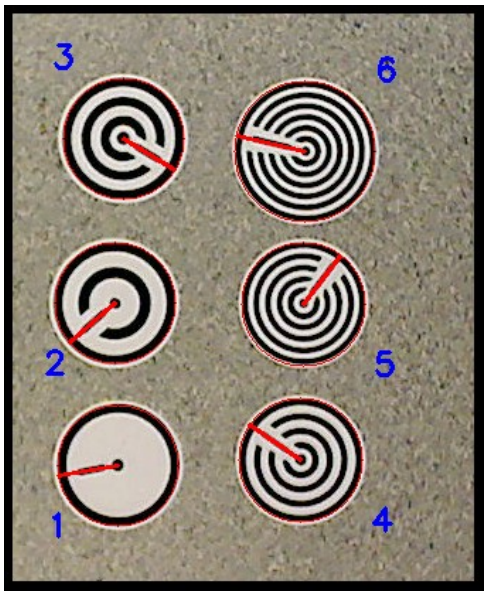


Figure 39 - *Identification results*

Application overview

Figure 40 displays the initial application's main window. It is composed of 3 different tabs, of which one gathers user-dependant information and the two others show data relative to found ellipses and detected patterns. `Foto!` and `Video!` buttons enter `Photograph` and `Video` modes, respectively, while button `Fondo` captures a frame to be used as a background image. There is also an exit button on the bottomwhich upon being clicked on will free all allocated resources and shut the application down.

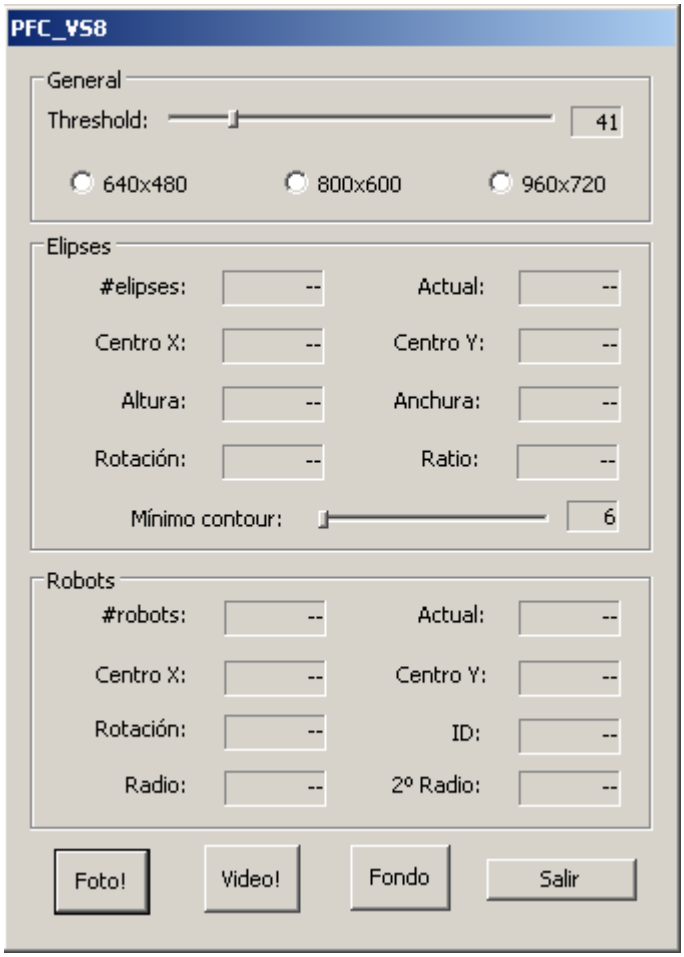


Figure 40: *Application interface*

Upon dialog startup all slider controls are initialized the way described in the next code snippet. The variables are Control Variables, meaning modifying them or invoking their methods directly modifies the interface after the call to `UpdateData()`. Also, the background image can be loaded from a previous session if it exists, that is, if file `fondo.jpg` can be opened (as can be observed, every freshly acquired one is saved in disk with that name).

```
#define GEN_THRESHOLD_I_INITIAL      41
#define INITIAL_STR_BIN_THRESHOLD   "41"
#define GRAY_MAX_VALUE 255
#define MINIMUM_AREA_MIN             6
#define MINIMUM_AREA_MAX            50
#define MINIMUM_AREA_I_INITIAL       6
#define MINIMUM_AREA_STR_INITIAL    "6"
. . .
void CPFC_VS8Dlg::dlg_initialize (void) {
      m_gen_threshold.SetRangeMin (0, false);
      m_gen_threshold.SetRangeMax (GRAY_MAX_VALUE, false);
      m_gen_threshold.SetPos (GEN_THRESHOLD_I_INITIAL);
```

```
	m_gen_threshold_value = INITIAL_STR_BIN_THRESHOLD;
	m_ell_contour.SetRangeMin (MINIMUM_AREA_MIN, false);
	m_ell_contour.SetRangeMax (MINIMUM_AREA_MAX, false);
	m_ell_contour.SetPos (MINIMUM_AREA_I_INITIAL);
	m_ell_contour_value = MINIMUM_AREA_STR_INITIAL;

	fondo = cvLoadImage ("fondo.jpg", CV_LOAD_IMAGE_GRAYSCALE);
	if (fondo) {

		cvNamedWindow ("fondo utilizado", CV_WINDOW_AUTOSIZE);
		cvShowImage ("fondo utilizado", fondo);
		showMessage (PFC_MESS_USING_THIS_BACKGROUND, false);
		cvDestroyWindow ("fondo utilizado");
		cvEqualizeHist (fondo, fondo);
		pii_bgroundGetMean();
	} else {
		showMessage (PFC_MESS_NO_BACKGROUND_FOUND, false);
	}

	dlg_cleanEllipseData ();
	UpdateData (false);
}
```

`dlg_cleanAcquiredData()` is another dialog function which sets all information output labels to `"--"` once the information they may be displaying is no longer relevant (and during dialog initialization as they will not have been used yet):

```
void CPFC_VS8Dlg::dlg_cleanAcquiredData (void) {
	m_ells_num =      "--";
	m_ells_actual =   "--";
	m_ells_center_x = "--";
	m_ells_center_y = "--";
	m_ells_altura =   "--";
	// etc.
	UpdateData (false);
}
```

Also, whenever the application is running and one of the sliders is modified an `OnHScroll` event is generated. This event is captured and taken care of by means of a call to the following function, with a pointer to the slider received as a paramater:

```
void CPFC_VS8Dlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar) {
	if (nSBCode == SB_THUMBTRACK) {
		if (pScrollBar == (CScrollBar *) &m_gen_threshold) {
			m_gen_threshold_value.Format (CString ("%ld"), nPos);
			g_gen_threshold = (float) nPos;

			SetDlgItemText (  IDC_STATIC_GEN_THRESHOLD_VALUE,
					  m_gen_threshold_value    );

			m_gen_threshold.SetScrollPos (nPos, true);
			UpdateData (false);
		} else if (pScrollBar == (CScrollBar *) &m_ell_contour) {
			m_ell_contour_value.Format (CString ("%ld"), nPos);
			g_ells_area_minima = nPos;

			SetDlgItemText (  IDC_STATIC_ELLS_AREA_MINIMA_DATA,
					  m_ell_contour_value);

			m_ell_contour.SetScrollPos (nPos, true);
			UpdateData (false);
		}
```

```
        }
}
```

Based on the received slider identifier the corresponding one is updated, setting the global variable's value to the correct value and the scroller in position.

The user input via these controls affects the processing loop directly, modiying the values it will use. Descriptions of the different parts of the interface and their functions follow:

General tab

The slider is attached to g_gen_threshold, directly influencing the main loop results (particularly in Video mode, as the results are immediately seen). Binary threshold values must lie within `[0, 255]` range since these are the possible values for a pixel in a gray-scaled image. The three radio buttons, for their part, determine the resolution at which we will attempt to captures frames; again, changes here will start affecting the obtained results obtained with `pii_processFrame()`.

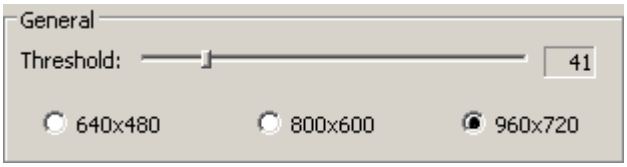


Figure 41: *General tab*

```
void CPFC_VS8Dlg::OnBnClickedRadio640x480() {
	g_img_w = 640;
	g_img_h = 480;
}
```

Button "Fondo"

The button titled `"Fondo"` is pretty straightforward: upon clicking on it a picture is taken and stored in a dedicated image buffer to be used when normalizing the subsequent images' intensity.

```
void CPFC_VS8Dlg::OnBnClickedButtonCapturaFondo() {
	IplImage *colour;

	if (cam_shoot (&fondo, &colour) == PFC_BAD) {
		showMessage (PFC_MESS_CURL_EASYPERFORM_JPG, true);
		return;
	}
	cvReleaseImage (&colour);
	cvSaveImage ("fondo.jpg", fondo);
	cvEqualizeHist (fondo, fondo);

	cvNamedWindow ("fondo utilizado", CV_WINDOW_AUTOSIZE);
	cvShowImage ("fondo utilizado", fondo);
	showMessage (PFC_MESS_USING_THIS_BACKGROUND, false);
	cvDestroyWindow ("fondo utilizado");

	pii_bgroundGetMean ();
}
```

It is pretty similar to the previously discussed background image loading; here, the camera is the source instead of the hard drive.

Button "Foto!"

Button "Foto!" will invoke onBnClickedButtonFoto() ...

```
void CPFC_VS8Dlg::OnBnClickedButtonFoto() {
	dlg_showFoto ();
}
```

... which will in turn force the application to capture data and enter the loop present within dlg_showFoto(), that is, *Photo* mode:

```
void CPFC_VS8Dlg::dlg_showFoto (void) {
	IplImage *foto = pii_processFrame ();

	// Foto original en color, aquí marcaremos los centros de robot.
	cvNamedWindow ("foto", CV_WINDOW_AUTOSIZE);

	if (g_ells_num == 0) {
		showMessage (PFC_MESS_NO_ELLIPSES_FOUND, false);
		cvDestroyWindow ("foto");
		return;
	}
	cvShowImage ("foto", foto);

	dlg_cycleThroughStuff ();

	cvReleaseImage (&foto);
	cvDestroyWindow ("foto");
}
```

When in *Photo* mode and after processing is done detailed information will be available regarding both ellipses and patterns. This is shown via dlg_cycleThroughStuff() function, which arbitrates actions to take to properly display the information. Most important variables here are g_ells_actual and g_robs_actual, which the function modifies according to the user input and which represent, respectively, the currently displayed ellipse and pattern.

```
void CPFC_VS8Dlg::dlg_cycleThroughStuff (void) {
	int key = 0;
	IplImage *ellipses = cvCreateImage (cvSize (g_img_w, g_img_h), 8, 1);
	IplImage *robots = cvCreateImage (cvSize (g_img_w, g_img_h), 8, 1);
	// Ventana para ellipses
	cvNamedWindow ("ellipses", CV_WINDOW_AUTOSIZE);
	cvZero (ellipses);
	cvEllipse (ellipses, g_ells[g_ells_actual].center, . . . );
	cvShowImage ("ellipses", ellipses);
	// Ventana para artefactos/robots
	cvNamedWindow ("robots", CV_WINDOW_AUTOSIZE);
	cvZero (robots);
	if (g_robs_num > 0) {
		cvCircle (robots, g_robs[g_robs_actual].center, . . .);
	}
	cvShowImage ("robots", robots);

	dlg_showNumericalData ();

	bool cambio;
	while (key != 27) { // ESC
		cambio = false;
		// izquierda elipse?
```

```
            if ((key == 'a') || (key == 'A')) {
                  cambio = true;
                  g_ells_actual--;
                  if (g_ells_actual < 0) g_ells_actual = g_ells_num - 1;
                  cvShowImage (g_imgLabels[I_ELLIPSES], g_ells[g_ells_actual].img);
            }
            // derecha elipse?
            if ((key == 'd') || (key == 'D')) { . . . }
            // izquierda robot?
            if ((key == 'q') || (key == 'Q')) {
                   = true;
                  g_robs_actual--;
                  if (g_robs_actual < 0) { g_robs_actual = g_robs_num – 1;
                  cvShowImage (g_imgLabels[I_LOCALIZACION], g_robs[g_robs_actual].img);
            }
            // derecha robot?
            if ((key == 'e') || (key == 'E')) { . . . }
            if (cambio == true) {
                  cvZero (ellipses);
                  cvEllipse (ellipses, g_ells[g_ells_actual].center, . . .);
                  cvShowImage ("ellipses", ellipses);
                  if (g_robs_num > 0) {
                        cvZero ("robots");
                        for (int i= 0; i <g_robs[g_robs_actual].n_ells; i++) {
                              cvCircle (robots, g_robs[g_robs_actual]. ...);
                        }
                        cvShowImage ("robots", robots);
                  }

                  dlg_showNumericalData ();    // show changes in displays.
            }
            key = cvWaitKey (10);
      }
      cvDestroyWindow ("ellipses");
      cvDestroyWindow ("robots");

      dlg_cleanAcquiredData ();       // clean displays.
}
```

Two functions interact with the interface in order to actually modify it. First one is `dlg_cleanAcquiredData()`, which has already been discussed, and second is `dlg_showAcquiredData()` which rather uses uses several instructions similar to the following ones to display it:

```
void CPFC_VS8Dlg::dlg_showAcquiredData (void) {
      m_ells_center_x.Format (CString ("%ld"), g_ells[g_ells_actual].center.x);
      m_robs_angle.Format (CString ("%.2f"), g_robs[g_robs_actual].angle);
      . . .
      UpdateData (false);
}
```

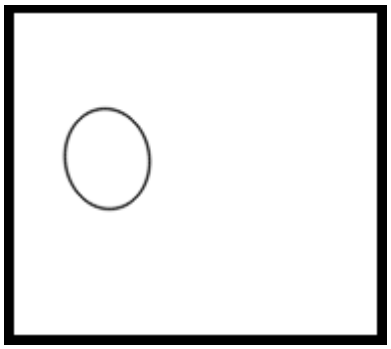


Figure 42: *Ellipses window*

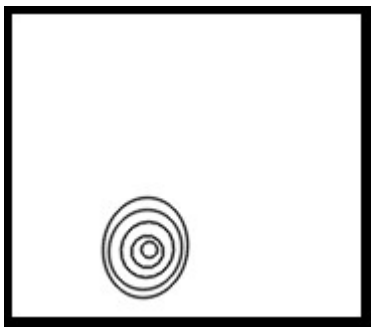


Figure 43: *Artifacts window*

Variables starting with `m_` represent interface controls of which method `.Format` is called for each one to properly display the required value. A final call to `UpdateData()` is again necessary as a means to make the changes visible. In this particular example, the current ellipse's center `X` coordinate and current pattern's rotation angle are being updated, respectively.

What follows is a summary of the key bindings used during the cycling:

Key	Effect
'**a**' or '**A**'	Cycle to previous ellipse.
'**d**' or '**D**'	Cycle to following ellipse.

Table 8 - *Key bindings for cycling ellipses*

Key	Effect
'**q**' or '**Q**'	Cycle to previous artifact.
'**e**' or '**E**'	Cycle to following artifact.

Table 9 - *Key bindings for cycling robots*

Button "Video!"

A single window will open upon pressing `"Video!"` button, showing the captured frames along information on any located and identified patterns.

```
void CPFC_VS8Dlg::dlg_showVideo (void) {
	int key = 0;
	IplImage *video = NULL;
	IplImage *auxVideo;

	cvNamedWindow ("video", CV_WINDOW_AUTOSIZE);
```

```
        while (key != 27) {
                auxVideo = video;
                video = pii_processFrame (); // video will already carry location
                                             // marks for identified patterns.

                cvShowImage ("video", video);

                if (auxVideo) cvReleaseImage (&auxVideo);

                key = cvWaitKey (20); // ms
        }

        cvReleaseImage (&video);
        cvDestroyWindow ("video");
}
```

Upon return from function `pii_processFrame()` the main difference with *Photo* mode arises: rather than displaying nearly all of the gathered information for the user to browse, it limits itself to location data regarding every robot in a graphical way. What is more important, the process is repeated within a loop and thus updates continuously take place (analyzing robots as they move about, for instance). With the currently involved processing a framerate of around `4-5` fps was achieved.

Tabs related to information output

These two tabs, regarding detected ellipses and robots, follow the same working pattern. They remain initially deactivated, and once the processing loop has started they will only be functional when in *Photograph* mode (their status will not be modified during *Video* mode).

As was mentioned in the previous section, once a single frame in *Photograph* mode has been processed and all possible information extracted, two different sets of items will be created: first comes a list of all ellipses, bounded by [`0`, `g_ells_num`], and then a list of detected patterns bound by [`0`, `g_robs_num`]. Two other variables, `g_ells_actual` and `g_robs_actual` keep track of the index of the currently displayed ellipse and pattern, respectively, and therefore of all its associated information. Also, both ellipses and patterns possess a center in (`X`, `Y`) which for both of them is displayed as well.

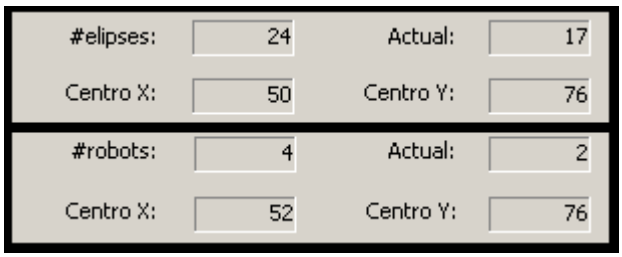


Figure 44: *Common information regarding ellipses and patterns*

With each of the previously described key bindings one of the adjacent ellipses/patterns in the list will gain the application focus, updating the interface controls. In Figure `44`, for instance, there are `24` ellipses and the currently selected one is `#17`, while we are seeing the second pattern's details ouf of a total of `4` of them. In this case they are almost concentric as well, which leads to think said ellipse must be part of the currently selected pattern.

We will now explain the other information displayed for each type of item:

Ellipses tab

Feature	Explanation

Altura – Height	Distance measured from the ellipse's along the vertically-running axe Y when rotation equals to `0` i.e. ellipse axes are parallel to global axes.
Anchura – Width	Same applies to its width, this time measured from ellipse's center to the sideways-running axe, X.
Rotación – Rotation angle	α-degrees rotation applied to the ellipse.

Table 9 - *Ellipses most prominent characteristics*

A slider control is also present in this tab, titled `"#mín. contour"`. It controls the minimum number of contour points required for an ellipse to be such, basically acting as a size filter. The Lowest possible value in the variable `g_ells_area_minima` cannot be lesser than `6` since below this value `cvFitEllipse2()` will not be able to determine the layout of the would-be ellipse.

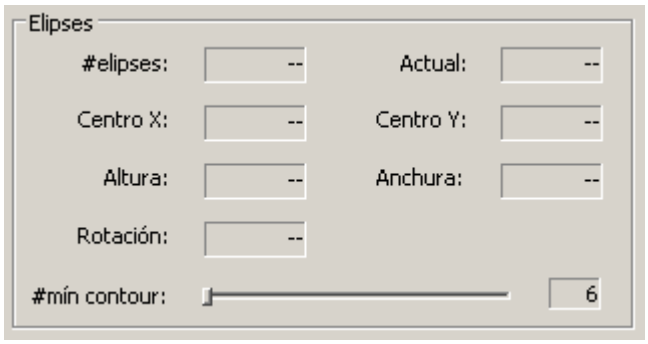


Figure 45 - *Ellipses information*

Patterns tab

Feature	Explanation
Rotación – Rotation angle	α-degrees rotation applied to the pattern, assumed to be equal to that of the first ellipse known to be part of the pattern.
ID	Identification detected for the pattern, or, number of concentric ellipses between the outer bounding ellipse and the central point.

Table 10 - *Robots most prominent characteristics*

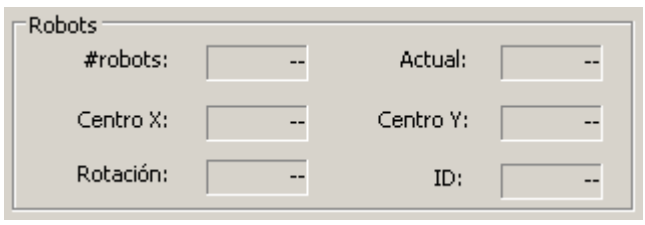


Figure 46 - *Robots information*

Conclussions

An affordable, relatively easy solution was offered for the problem of identifying and locating patterns in a 2D space. Even if its results cannot be immediately applied nor have a visible effect on the environment, it is indeed of great help when used in conjunction with those other works related to controlling the analyzed moving robots. The output can be fed to these systems to obtain, for example, an estimation of the error to which react accordingly, making the system as a whole more efficient and accurate.

A set of videos can be played, showing the results of the identification and location of patterns as the group of mobile robots perform tasks or simply move in the designated area.

- Videos showing the camera output along the summary of results. This video is a record of the robots' actual movements, feeding directly from the camera, and some additional information stating their coordinates and rotation angle in a graphical way. This information is updated with each complete loop of the algorithm, which happens several times per second.

- Videos showing internal steps in the identification and location tasks. Typically these correspond to umbralization images or the resulting set of detected ellipses.

Future work

A very closely related project to work with is robot coordination. A group of them may perform tasks collectively – for instance, we can assume each one of them individually is unable to move a certain object, while collectively it is feasible. Even if their controlling algorithms already contemplate how to detect other robots and keep track of their locations (with wireless communications, for example), it would be a great addition to have a supervision system watching for them, which can either corroborate their results or correct them.

In order to measure how effective it actually turns out to be, a robots coordination project could toy with the option to run with or without this additional feedback. At the very least some improvement should be observed, like a greater chance of reacting to situation changes in a faster and more reliable way. It would be interesting to know, for example, how previously hard situations to solve are now reduced. For instance, robots may block each other during identification if they carry it out with mounted cameras (meaning any robot could not see what lies behind those around in the path of its line of sight). With the help of a zenital camera this is now trivial, as the different patterns will remain within visibility range constantly.

Finally, some more work could be put into reducing the size of patterns, for instance, by experimenting with different camera shooting distances (trading off visibility area) or higher resolutions (which may cause a decrease in FPS due to heavier workload).

Bibliography

[1] The C++ Resources Network. http://www.cplusplus.com/

[2] OpenCV Processing Library. http://ubaa.net/shared/processing/opencv/

[3] MFCs Reference. http://msdn.microsoft.com/en-us/library/d06h2x6e%28v=vs.71%29.aspx

[4] Wikipedia. http://en.wikipedia.org/wiki/Main_Page

[5] The Tech Museum – Degrees of Freedom. http://www.thetech.org/exhibits/online/robots/arms/7deg.html

[6] Hanno's viewport. Sander, I. http://ingrit.com/ingolf/viewport/index.htm

[7] Digital Image processing. González, R.C. Woods, R.E.

[8] Técnicas de segmentación de imagen. Salmerón, A. Pérez Jiménez, A. http://web-sisop.disca.upv.es/~sdv/

[9] Classification of robots. Rapp, Steve.
http://www.hgs.k12.va.us/Engineering_and_Robotics/Robotics/FlashPaperVersion/ROB_Chapter11.html

[10] Rasterización de imagen sintética: rellenado de áreas. Monserrat, C.
http://users.dsic.upv.es/~cmonserr/TID/Bloque1_apuntes.pdf

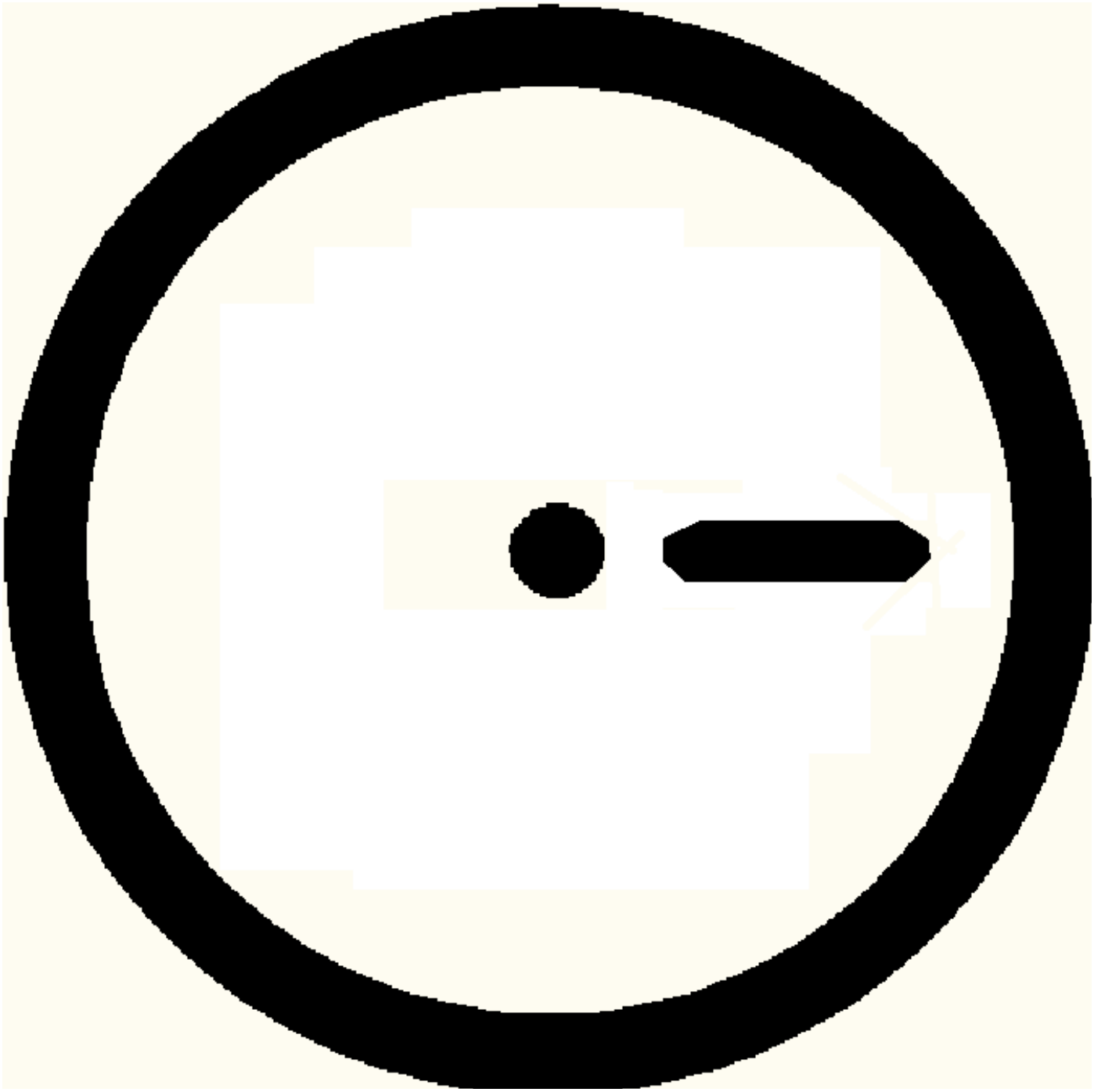
[11] The method of least squares. J.Miller, Steven.
http://web.williams.edu/go/math/sjmiller/public_html/BrownClasses/54/handouts/MethodLeastSquares.pdf

[12] AXIS PTZ-212 cameras. http://www.axis.com/files/datasheet/ds_212ptz-v_34051_en_0812_lo.pdf

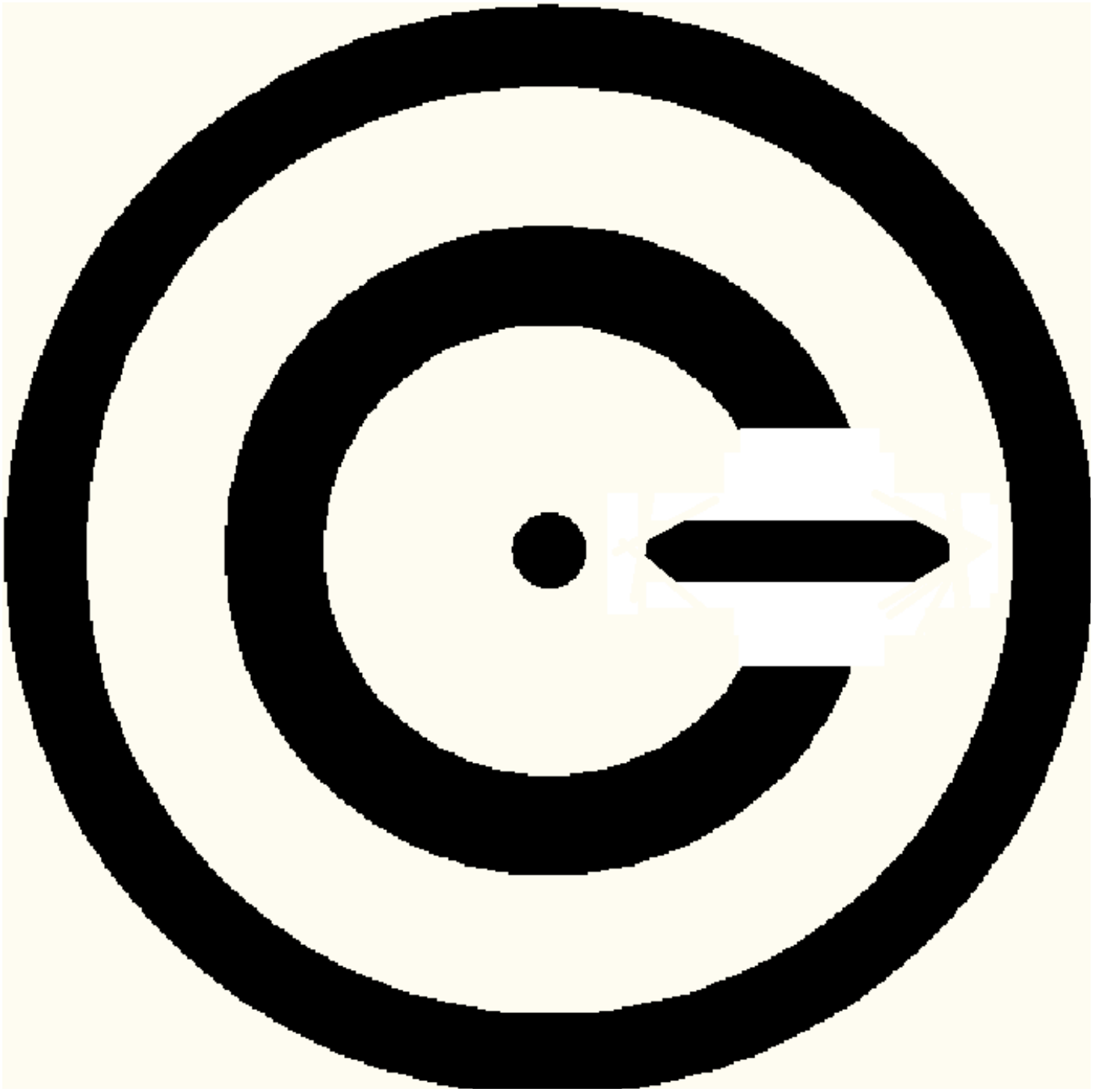
[13] Logiech Pro 9000 cameras. http://www.logitech.com/repository/1403/pdf/25618.1.0.pdf

Annex A – Set of patterns

Pattern #1



Pattern #2



Pattern #3

Pattern #4

Pattern #5

Pattern #6

Annex B – Usage example

Depending on whether we will be using a USB camera or a web camera we will need to choose, respectively, among the following two sections. Af ter that the final thesis project can be launched.

Setting up Logitech Pro 9000 controller

If we do not launch the software by ourselves it will run upon attempting to access the camera. In section *Web Camera Controller* we can adjust some parameters. No tracking, panning and tilting, etc. features should be used. Zoom is not recommended either. Auto focus is a handy one though.

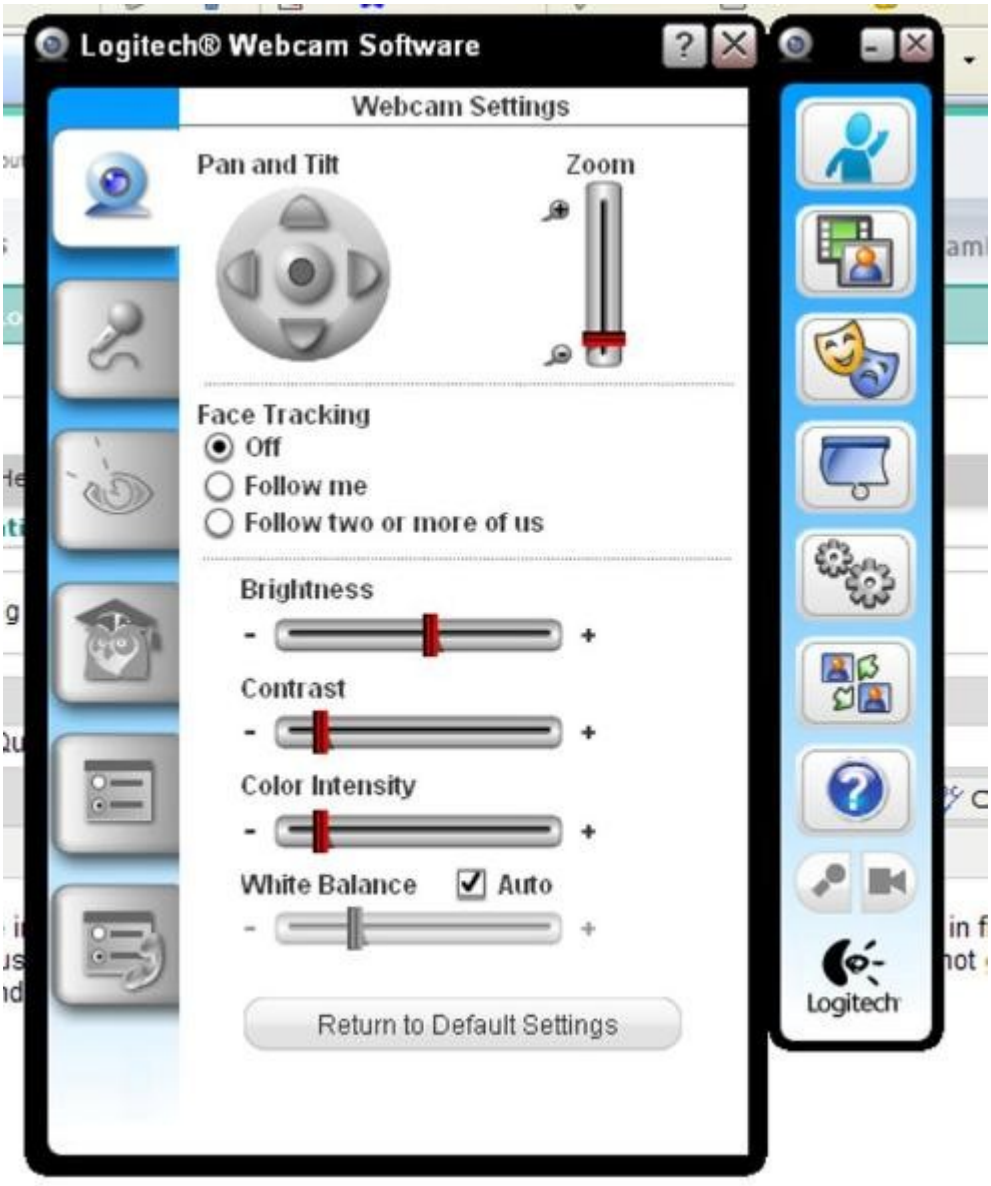


Figure 47 - *Logitech web camera controller interface*

Setting up Webcam XP

After installing the software we just need to execute it and follow these steps:

Figure 48 - *Connecting to a web camera*

Figure 49 - *Choosing camera manufacturer and model*

Figure 50 - *Web camera parameters*

Running final thesis project

If this is not the first time it is launched and a background image was previously loaded, it will be shown in a separate window and a message will pop up informing us.

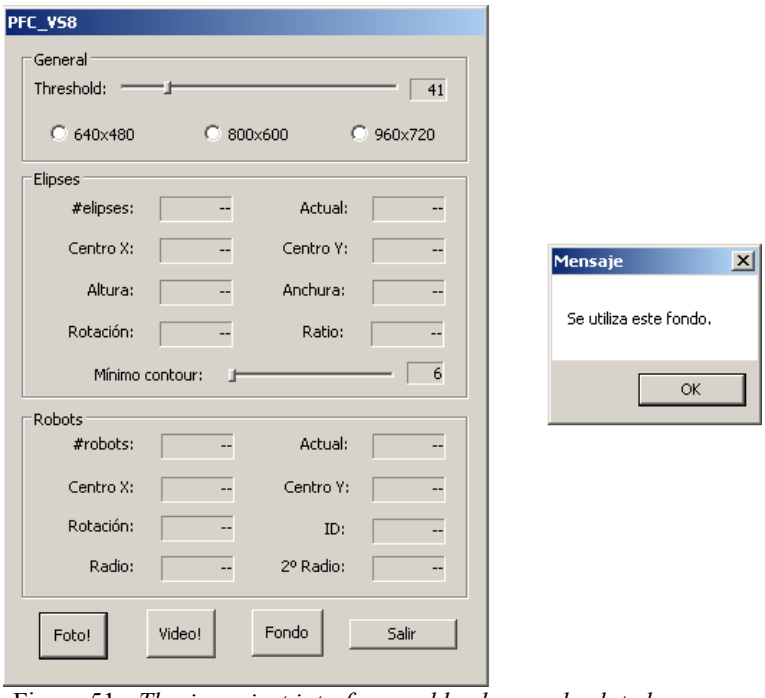


Figure 51 - *Thesis project interface and background-related message*

If the last background image was taken some time ago or under different conditions it will be a good idea, before setting any of the robots or patterns in the scene, to press the button "Fondo" in order to capture it. If successful, we will be told so and then returned to the main window. Once we have got a suitable background image it is the right moment to distribute the robots and their patterns.

Let us focus on the "Video!" button now. When clicked the application will enter its main loop and show graphical output. It is most probably necessary to switch to the binary image output (see section *pii_threshold*) to adjust the thresholding value correctly.

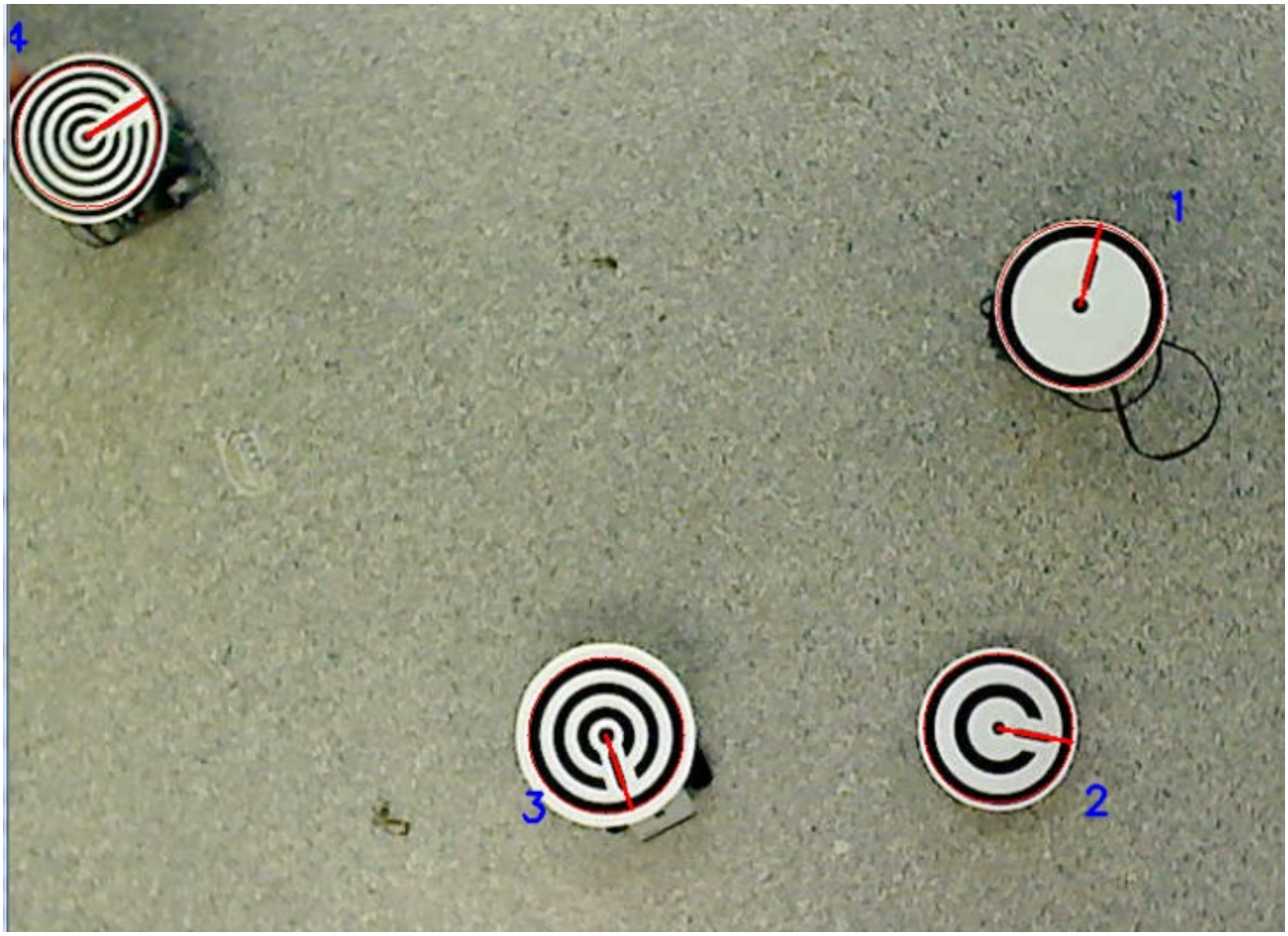
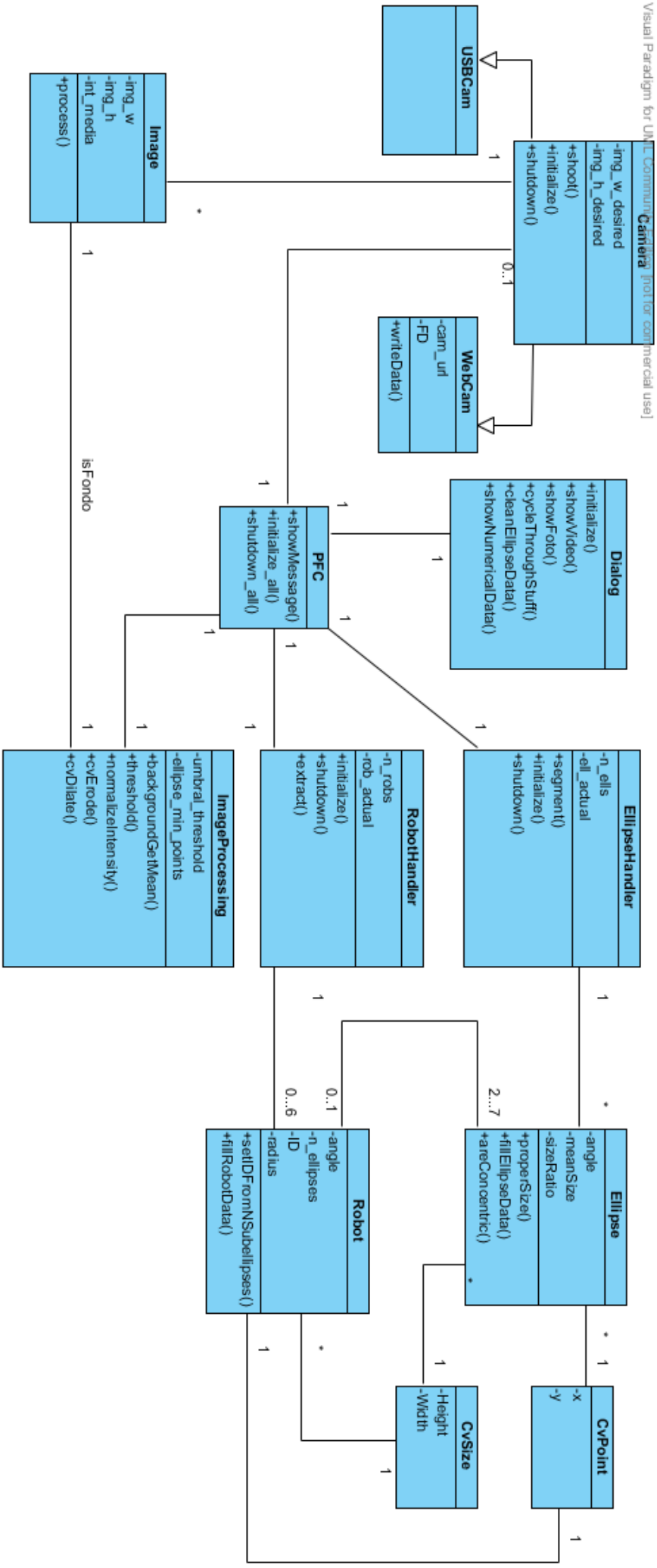


Figure 52 - *Moving patterns identification & location*

At any moment we can change the resolution by using the radio buttons provided for that purpose. The *esc* key will take us back to the main window. On the other hand button "Foto!" will perform the same task, though only to a single frame. Diverse information regarding ellipses and any detected pattern will be displayed in the main window, using the keybindings in sections *Key bindings* to navigate through it.

Finally, it is important to keep in mind that the windows opened by these two buttons cannot operate simultaneously – any of them active will need to be shut down with *esc* before the other can be activated.

Annex C – UML


Visual Paradigm for UML Community Edition [not for commercial use]
USBCam
Camera
-img_w_desired
-img_h_desired
+shoot()
+initialize()
+shutdown()
Image
-img_w
-img_h
-int_media
+process()
WebCam
-cam_url
-FD
+writeData()
Dialog
+initialize()
+showVideo()
+showFoto()
+cycleThroughStuff()
+cleanEllipseData()
+showNumericalData()
PFC
+showMessage()
+initialize_all()
+shutdown_all()
isFondo
EllipseHandler
-n_ells
-ell_actual
+segment()
+initialize()
+shutdown()
RobotHandler
-n_robs
-rob_actual
+initialize()
+shutdown()
+extract()
ImageProcessing
-umbral_threshold
-ellipse_min_points
+backgroundGetMean()
+threshold()
+normalizeIntensity()
+cvErode()
+cvDilate()
Ellipse
-angle
-meanSize
-sizeRatio
+properSize()
+fillEllipseData()
+areConcentric()
Robot
-angle
-n_ellipses
-ID
-radius
+setIDFromNSubellipses()
+fillRobotData()
CvPoint
-x
-y
CvSize
-Height
-Width
1
0..1
*
2...7
0...6