Master Thesis:
Coordinates obtaining for the disassembly of Duplo bricks structures

ACIN
Automation and Control Institute

Alejandro Moser Passani

July 2, 2015
Abstract

This project is part of the Distributed and Intelligent Disassembly of Products with Mobile Robots project launched by the research program of the Sparkling Science of the "Federal Ministry of Science and Research (BMWF)" with the collaboration of the Institute of Automation and Control, University of Vienna. It contributes to develop software and research on how to provide knowledge to a robotic to disassembly of objects.

The fact of providing a robot with a certain ability to develop this task, requires a prior study of the current image processing techniques, with the aim of obtaining the right solution to solve the problem. This thesis shows a study of different image processing algorithms available nowadays. It also studies the possibility of adapting the software to be developed in an embedded platform, in order to avoid the robot to depend from a high performance PC.

As a starting point for the development of the idea, it poses a simple problem: the disassembly of Duplo bricks structures. To address the difficult task of image processing, the well known OpenCV libraries created by Intel, currently in Open source, will be used. These libraries contain more than 500 functions.

The problem of disassembling the Duplo bricks has been divided into 3 steps: object recognition, verification and feature extraction and calculation of its coordinates in the real world. For the object recognition, the system has been trained with pictures from different angles, backgrounds and lighting of the piece to be recognized. The complexity of this first stage lies on the fact that the Duplo bricks have a uniform color. Due to the difficulty involved in generating models that recognize the full bricks, it has been chosen to recognize the small hollow cylinders that serve as anchorage between bricks, getting an extra edge: Making possible in a future to recognize bricks with different number of cylinders. The verification and feature extraction has been done manually, as the functions provided by
OpenCV were not quite optimal. This verification consists in checking if the founded cylinders are lined up in groups of 8 (in our particular case) forming a rectangle. The last step has been solved using a specific function that compares the obtained points in the picture with the points that define the object. After this last stage, and with the previous calibration of the camera, the translation and rotation matrices of the object in the real world are obtained. With these data, the robot is able to find with good precision the brick to take.

The result of this Project is a completely functional code able to give the coordinates of the Duplo bricks got through a camera.
Contents

1 Introduction ............................ 1
  1.1 Context .................................. 1
  1.2 Purpose of this thesis .............. 2
  1.3 Approach ............................... 2

2 State of the art ....................... 5
  2.1 History .................................. 5
  2.2 Image processing ........................ 8
  2.3 Segmentation ............................ 13
  2.4 Representation and Description .... 16
  2.5 Object recognition ..................... 18
  2.6 Interpretation ........................... 22
  2.7 Festo Camera ............................ 23

3 Analysis ................................. 26
  3.1 Problem definition ..................... 26
    3.1.1 Object Recognition .................. 26
    3.1.2 Object Verification .................. 27
    3.1.3 Obtaining Coordinates ............... 29
    3.1.4 Festo camera ........................ 29
  3.2 Proposals to solve the problem ...... 30
    3.2.1 Object Recognition .................. 30
    3.2.2 Object Verification .................. 31
    3.2.3 Obtaining Coordinates ............... 32
    3.2.4 Festo Camera ........................ 33
Chapter 1

Introduction

1.1 Context

In recent years society has begun to realize the damage we are causing to our planet: The effects can be seen today, for example with climate change and the shortage of some materials. We live in a consumer society where it is cheaper to buy than repair, we throw things because they aren’t good or simply because we don’t like or are no longer fashionable. The need arises to raise awareness about the issue and begin to curb these acts that could bring the planet into chaos.

One of the points is the waste of materials, recycling is promoted greatly. Today it is common to have special containers for each type of material. But it is not enough, land lls are lling up as new products and objects, where they oxidize and become useless junk.

The idea resides at this point, recycling. Being able to reuse products that are thrown away just because, reuse their parts, their material, etc... The problem of this, is that it would require many people working in order to disassemble each object, long time and knowledge of how each object is disassembled and which are the parts of interest. The idea of automating this process has been born here, proposing robots capable of doing this task. The proposed project: Distributed and Intelligent Disassembly of Products with Mobile Robots. A project which seeks to explore ways to...
provide to robots smart enough to be able to recognize objects and disassemble their interesting parts. In this way able to have for example a robot capable of recognizing several models of washing machines, disassemble them and remove the engine to be reused.

1.2 Purpose of this thesis

The aim of this project is to find a solution to endow a robot some kind of artificial intelligence, in order to make it able to disassemble objects. Consider first what is already done, investigate the possible paths to take, implement a solution, test their effectiveness and think about the encountered problems and possible future improvements. All this should be done, trying to use nowadays existing tools.

Another important objective is the need to perform a program capable to run on board, with an embedded operating system cheaper than a PC. From this came another of the objectives raised in this thesis. It is necessary to study the possibility of running the software developed in a commercial board. For this specific case, it is claimed to work on a Festo camera, with an embedded Linux, facilitating the task of acquisition of images.

This gives way to other of the purposes of this thesis, trying to implement the solution using the OpenCV library (Open Source Computer Vision), and use them on the embedded platform.

1.3 Approach

Due to the complexity that it supposes endowing a robot of the necessary knowledge to disassemble, for example, a washing machine, it is decided to begin with more simple objects in an ideal environment (smooth and white surfaces). The idea is to find an initial solution to obtain a knowledge on where are the most complicated points and to be able to do a study of pros and cons. Thereby, we decided to start working with structures done
with Duplo blocks like the one that appears in the figure 1.1. To facilitate also the development of software, it is decided to develop the software (as previously commented) using OpenCV libraries with the aim also to study its functionality, its potential, etc.

Figure 1.1: Duplo brick

Below we are listing the objectives as an approach of the steps to make to achieve the purpose of this project:

1. Familiarize with OpenCV
   - Information search
   - Install it and test it on PC

2. Software development (PC)
   - Definition of the architecture
   - Divide the problem into sub-tasks
   - Study and test the functions
   - Write the program
   - Test results
   - Optimise

3. Demonstrate feasibility of using OpenCV in a Festo camera
1.3. APPROACH

• Familiarize with the camera
• Install OpenCV
• Try some examples with cross compiler

4. Run the software in the Festo camera

• Adaptation of the software
• Study the performance
• Improvements
Chapter 2

State of the art

As in any project, one of the most important things to be performed before beginning, is to look for information on what is already done. A previous research work necessary both, to become familiar with the subject, and to be able to foresee which paths to take to develop the project. In this way progress can be made, new things can be done.

This chapter presents a study of the state of the art on the different subjects that we touch on this project. Besides a short introduction to the history of processing images, the most relevant points to the project will be presented in greater depth.

2.1 History

From the beginnings of the computer science, the replication of the human behavior on the machines has been chased. The human body has been and is considered the most efficient model to follow for certain disciplines. For example: robotics, computer vision, learning based on experience, and a large list of disciplines included in computing. The problem of the choice of such a model as template, results in frustrating consequences for any line of investigation. Today, trivial behaviors for humans are still not scientifically typified, and among them, the process of human vision.

For this reason the construction of a system that emulates the hu-
man visual system would be practically impossible. Although there is a huge number of publications in neurophysiology, psychology and psychophysics, the knowledge of the human visual system beyond the eye itself is mostly disjoint, speculative and scarce. All this caused the gradual abandonment of this line of research, until, in the decade of the eighties, there was a rethinking on the specification of objectives for this discipline. The ambitious name of "Artificial Vision" was changed by one, humbler and consistent with the objective, such as "Computer Vision". The appearance of more powerful computers, specific hardware items that relieved the use of complex algorithms, and greater historical experience in other fields of computing, joined the change on this new perspective of the subject. Methodologies that divided the problem of computer vision in different phases, making solutions more affordable, and related to other disciplines which were independent, such as image processing or pattern recognition, began to be described.

Computer vision today includes both, obtaining and characterization and interpretation of images. This means algorithms of very different types and complexities. Six stages or phases can be distinguished on an actual computer vision system:

- Capture: This is the process across which, a visual image is obtained.

- Preprocessing: Includes techniques such as noise reduction and detail enhancement.

- Segmentation: The process that divides an image into objects which are of our interest.

- Description: The process by which desirable characteristics are obtained to differentiate one type of object from another, such as size and shape.

- Recognition: The process that identifies objects in a scene. For example the different types of workpieces in a chessboard.
CHAPTER 2. STATE OF THE ART  2.1. HISTORY

• Interpretation: The process that associates a meaning to a set of recognized objects.

This involves different types of processing depending on the level in which we move:

• Low-level Vision: includes the acquisition and preprocessing. Runs typically filtering algorithms, image restoration, enhancement, contour extraction, etc..

• Intermediate level Vision: includes segmentation, description and recognition with algorithms of typically feature extraction, shape recognition and labeling of these.

• High-level Vision: includes the interpretation phase. Usually, these algorithms refer to the data interpretation generally by methods that are typical of artificial intelligence for database access, searches, approximate reasoning, etc..

Finally, it is important to highlight the range of applications where computer vision has more and more an significant role:

• Military: much of the computing achievements made have been promoted or subsequently acquired and improved by this sector. Among the most common applications we highlight the detection and target tracking.

• Robotics: industrial applications for robot guidance. Analysis of satellite imagery.

• Automatic identification of fingerprints: widely used in security systems and access control.

• Quality control: widely used in assembly lines.

• Medicine: with special emphasis on the progress made with MRI imaging (magnetic resonance imaging), and known achievements in X-ray imaging.
• Character Recognition: within this area we would find applications such as label reading, bank check processing, text reading, etc..

• Mapping: mapping from photographs, synthesis of weather maps, etc..

In short, each day appears a new computer vision application applied to an area that gets much benefit from that contribution.

2.2 Image processing

This point deals with the operations and transformations that are applied to digital images in a processing step prior to segmentation and recognition. Its purpose is to enhance or highlight some element of the images, so that it is possible to make the later stages of segmentation and classification. Due to the large number of existing operations and, because, we already will go into details in the following stages, we will do only a brief introduction and an overview of the most important operations.

All operations that we will describe in this chapter can be explained from the perspective offered by the theory of filters. A filter can be seen as a change mechanism or transformation of an input signal which is applied to a function, known as transfer function, to obtain an output signal. In this context, it means a signal depending on one or more independent variables. The sounds and images are typical examples of signals.

Bibliography used: [1, 2, 3, 4]

Basic operations between pixels

Direct operations on pixels can be classified into arithmetic-logic operations and geometric operations.

The first ones are by far the most used at any level in an image processing system, as they are those that are used to read and give values to the pixels of the images. The basic operations are:
• Conjunction - logical AND operation between the bits of two images. Used to erase pixels in an image.

• Disjunction - OR logical operation between bits of two images. Is used to add pixels to an image.

• Negation - Inversion of the bits that form an image. Is used to obtain the negative of an image.

• Sum - addition of pixel values of two images.

• Subtraction - Subtracts the pixel values of two images.

• Multiplication - Multiplication of the pixel values of an image by the ones of another. Use it to add texture to an image.

• Division - Division of the pixel values of an image by the ones of another.

It has been seen that in gray levels images the value 255 is typically used to represent white and the 0 for black. So, the conjunction operation between black and white gives as result black. When performing arithmetic operations it should be taken the precaution of verifying that the result $R$ of an operation falls within the domain of allowed values.

As geometric operations we have that if the points are expressed in homogeneous coordinates, all transformations can be treated by matrix multiplication. The most common geometric operations are:

• Translation - Movement of the pixels in an image according to a motion vector.

• Scaling - Resizes an image.

• Rotation - Rotates the image pixels around the origin of coordinates.

The geometric matrix operations can be grouped by multiplying the matrices. In this way for example, it is possible to have one single matrix to perform a movement, a rotation, another displacement, and a rescaled
within a single step. By making this operations composition, it must be remembered that the matrix product does not comply the commutative property.

**Operations on the histogram**

It is known as histogram of the image quantization levels, or simply image histogram, to a bar diagram in which each bar has a height, proportional to the number of pixels of a given quantization level. Usually, the abscissa axis shows the different value quantification levels that the pixels of that image could take, while the vertical axis shows the number of pixels for each level of quantification.

The histogram of an image in grey levels provides information about the number of pixels for each level of intensity. In RGB color images 3 histograms are used, one for each color component.

There are several operations that can be done with histograms:

- **Increase and reduction of contrast**: The contrast is defined as the relative difference in intensity between a point on an image and its surroundings.

- **Equalizer**: The transformation that aims to obtain for an image a histogram with a uniform distribution. In other words, that the number of pixels is the same for each histogram’s grey level of a monochrome image.

**Spatial filtering operations**

Spatial filters are filters that are made directly on the image and therefore in the domain of space. The most relevant are:

- **Smoothing filters**: spatial aliasing filtering is based on the averaging of adjacent pixels to the pixel being evaluated. We can find two types of smoothing filters:
- Median filter: It is based on substituting the value of a pixel by the median of the set formed by it and its eight neighbors.

- Frare fellow filter: Consisit on the comparison of the pixel intensity with the intensity of its 8 neighbors. If the difference is greater than certain value U (previously defined), the pixel is then substituted by the average value of its neighbor pixels. In other case its intensity value is maintained.

- Filters for obtaining contours: Allow to highlight those areas where there is a image contour. This is achieved thanks to that the calculation of the directional derivative of a function allows to know how changes occur in a particular direction. Such changes usually correspond to the contours of the objects in the images.

- Laplacian filter: This operator, which is based on the second derivative, is zero when the first derivative becomes maximum, in other words, when a sign change appears on the first derivative.

**Operations in the frequency domain**

We have seen that a digital image is a representation that refers directly to the light intensity of points in a space, therefore it is said that a digital image is a domain representation of space. There are other representations that contain the same information, but not in the domain of space. This is the case of representations in the frequency domain.

The representations in the frequency domain, detail how often certain patterns are repeated in an image and, with it, achieve to represent the information of such an image. This representation can be particularly useful, as having the repetition frequency of such patterns, elements in the images such as noise, contours or textures, can be detected and directly altered.

To perform a transformation to the frequency domain, the use of transforms as the DFT (Discrete Fourier Transform) is required.
Within each operation in the frequency domain there is another type of filtering, the frequency filtering. Once the formulation of the Fourier transform for grey level images is known, it becomes possible to transform an image from the space domain to frequency domain. As already seen, once in the frequency domain, it is easy to perform filtering to remove elements appearing within certain period.

There are several types of frequential filters:

- **Low Pass Filter**: Low pass filters are filters that remove high frequencies, leaving "pass" the low frequencies. To make a filter of this type is just enough to set to zero the modules of the Fourier coefficients on the high frequencies, leaving unchanged those on low frequencies. It is important to choose an appropriate cutoff value at which it is considered that a frequency is high or low.

- **High Pass Filter**: Idem as above but with the high frequencies.

- **Bandpass filter**: The bandpass filters are filters on which a range (or band) of frequencies remains unchanged and where coefficients corresponding to other frequencies are removed. So, the High Pass and Low Pass Filters constitute the the limit cases of the Band Pass Filter.

### Morphological operations

Traditionally, the morphology has been a part of biology that studies the shape of animals and plants. Likewise, mathematical morphology is a set of mathematical techniques that allow to treat problems involving shapes in an image.

The morphological operators analyze the geometric structure of an image using as probe an adjustment pattern called structuring element (S.E). Moving the SE over the image, the operator typically analyze its position in relation to the foreground and background of it. The SE can have any size and shape (circle, square, etc..). Its center is located at each pixel of
the original image by applying the morphological operation on the points located in the SE.

S.E operations:

- **Dilation**: The output of the expansion is the set of points swept by the center of the SE while a point of SE matches any of the image. Alternatively, the dilation can be interpreted as the result of replacing each white pixel of the original image by a replica of structuring element.

- **Erosion**: The output of erosion is the set of points swept by the center of the SE as long as it complies that all the points of the SE were contained in the image. One of the most typical uses of the erosion is the removal of irrelevant details.

By combining these two operations morphological filters can be performed:

- **Opening**: The composition of an operator of erosion and a dilation with the same structuring element.

- **Closure**: Is the composition of a dilatation operator followed by another of erosion with the same structuring element.

### 2.3 Segmentation

Bibliography used: [1, 5, 6, 8, 7, 9]

Segmentation is the process of dividing a digital image into homogeneous regions with respect to one or more features (such as brightness or color) in order to facilitate further analysis or automatic recognition. Locate a person’s face within the image of a photograph or finding the limits of a word within a text image are examples of segmentation problems.

Although there are different approaches to the segmentation, practice shows that the segmentation does not have strict rules to follow, depending on the problem at hand, it may be necessary to devise specific techniques.
Segmentation should be seen as a process in which, from an image another is created in which each pixel is associated with a distinctive label of the object to which it belongs. Thus, once an image is segmented, may form a list of objects consisting of groups of pixels having the same label.

Segmentation ends when the extracted objects from an image correspond unequivocally to the different disjoint regions located in the same one. In this case we speak about complete segmentation of the scene or image and in the opposite case, about partial segmentation. In a complex scene, the segmentation result could be a set of overlaped homogeneous regions, and in this case, a partially segmented image would have then to be subjected to further treatment in order to achieve a complete segmentation.

In general, the segmentation process is often complex because, on one hand, there isn’t adequate information of the objects to be extracted and, secondly, noise appears in the scene to be segmented typically. This is the reason that the use of knowledge about the image type to segment or some other high-level information can be useful to achieve image segmentation.

Some typical examples of segmentation processes are: to try to separate the characters forming a word within a text image, detect certain types of cells in medical images, extract vehicles that appear in an image of a road.

The different objects in an image can be located considering aspects such as contours or their texture. We can differentiate between 3 groups of techniques: thresholding based techniques, object contours based ad techniques based on local properties of the regions.

Techniques based on:

• Thresholding: The thresholding is a process that allows converting a grey level or color image in a binary image, so that the objects of interest are labeled with a different value from the background pixels.

• Edge Detection: The segmentation based on edge detection includes a large number of techniques that use information provided by the
boundaries of objects in an image. Since is desired to find individual objects present in an image, it seems logical that if we found the boundaries of such objects with the background, the objects could be segmented from the generally scene.

• Growing regions: The techniques grouped under the name of growing regions determine areas within an image based on criteria of similarity and proximity between the pixels. In these techniques the homogeneity (or default of homogeneity) between adjacent regions is the criteria used to join (or divide) regions of the image. This homogeneity can be defined based on criteria such as: the medium grey level, color, shape, etc.. The result of the segmentation is a partition of the image in homogeneous regions.

• Color: it consists in segmenting regions where its RGB components matches with the desired RGB pattern. Within this technique, HSV format images are commonly used. The color is here defined by only one parameter in front of the 3 parameters used by the RGB format.

• Texture: In this approach, models are defined textures to think to an image not as a collection of pixels, but rather to treat it as a function \( I(x, y) \). The purpose of the model is to transform an image window in a collection of values that constitute a feature vector. This vector will be a point of a n-dimensional space. The representation will correspond to the texture if the windows taken of the same sample of texture are "nearby" in the considered feature space. Also, if windows from the image texture with different patterns are "far away" in the considered feature space. Texture models are divided roughly into three categories: based on pyramidal structures, which try to capture the spatial frequencies at different levels of resolution; based on random fields, assuming that the values of a pixel are selected through a two-dimensional stochastic process; and those based on statistical methods using cooccurrence matrices constructed from the images. From these matrices a series of measures such as mean value, variance,
2.4. REPRESENTATION AND DESCRIPTION

CHAPTER 2. STATE OF THE ART

entropy, energy and the correlation between pixels are extracted.

- Movement: The movement can be a powerful tool for the segmentation of animated objects over static backgrounds. The basic techniques consist on the study of the image resulting from the subtraction of two consecutive images of an animated sequence. This technique is known with the name of background subtraction. The objects that move between these two images produce a set of pixels with non-zero values on the image subtraction. Meanwhile, the static elements of the image, by not varying, produce zero after subtraction.

2.4 Representation and Description

Bibliography used: [8, 9, 12]

After segmentation, the regions should be represented in a suitable form for further processing. The representation of regions (objects) is conceptually similar to a compression process. The aim is to represent the image (possibly with losses) in a simple form, with few parameters.

There are two ways of description of areas:

- By its Border: shape parameters.

- By its Inside: color parameters, texture parameters, etc..

Descriptors allow to capture essential information about an important object for any application (for recognition, for example).

Among the most important methods of representation following can be found:

- Codes of chains: Used to represent a boundary by a connected sequence of rectilinear segments of specified length and direction.

- Polygonal approximations: They seek describing a curve by a polygon with a small number of straight segments. For a closed curve, the approximation is accurate when the number of segments on the
polygon is equal to the number of points on the curve. In practice, what is intended is to obtain the representation of the object boundary with the lowest number of segments of the polygon.

- Signatures: 1-D representation of the border

- Border segments: They follow the principle of dividing the boundary into segments whose description is simpler.

- Skeletons: The skeleton allows to describe a region by a graph. The skeleton can be obtained using mathematical morphology, although these techniques do not guarantee always a connected skeleton.

As the most important types of descriptors we can find the following:

- Border:
  - Simple descriptors: such as length, diameter and curvature.
  - Shape descriptors.
  - Fourier descriptors: depending on the number of used coefficients it may represent borders in a more or less accurate form.
  - Statistical moments: can be obtained for any 1-D representation of an image.

- Region:
  - Simple Descriptors: perimeter (length of its border), area (number of pixels in the region), compactness (perimeter coefficient^2/area), median and mean grey level, etc.. The compactness is unvariant to changes of scale, to a turn, and so on.
  - Topological descriptors: the topology is the study of unvariant properties under deformations that do not alter the relations of neighborhood (ie, without breaks or mergers).
2.5 Object recognition

Bibliography used: [25, 24, 26, 27, 28, 29]

This chapter will give an overview of the different algorithms to classify the items that appear in a scene to fully understand it. Classification algorithms have the task of distinguishing between different objects of a predefined set called the universe of work. Normally, the universe of work is considered as divided into a collection of K classes ($\alpha_1, \alpha_2, ..., \alpha_K$), belonging the different objects to some of these classes.

This chapter will show different methods for determining, automatically, to which class an object of a working universe belongs. These methods are known as classifiers.

To perform the automatic recognition of objects it is being performed a transformation that converts an object of the universe of work in a X vector whose N components are called discriminating characteristics or traits.

These features should allow to discriminate to which classes any object in the world of work can belong to.

The value of the feature vector for a particular object is known as a pattern. In other words, a pattern is a particular instance of a given feature vector.

The determination of the N discriminating characteristics is a difficult process that often require the use of imagination. In general, they are commonly used features such as moments of the objects to recognize,
some transformation of them (Fourier, cosine ...), the images themselves, or any characteristic that can be derived from the objects using some algorithmic procedure.

Once the discriminating characteristics for a particular problem have been determined, the classification of an object begins by obtaining its pattern. The next step is to determine the proximity or degree of belonging of this pattern to each of the existing classes. To this effect, we define the discriminant functions or decision functions as those functions that assign to a pattern a degree of similarity in front of each of the different classes.

**Learning sample**

To perform the calculation of the discriminant functions it is usually required the existence of a set of patterns similar to those desired to recognize, called set of learning or training set. The patterns of this set are used as models to create the discriminant function, that will classify correctly the patterns of the universe of work. Therefore, the training set must be comprised of a representative subset of the universe of work.

When the sample is plentiful, it is interesting to create another set with it. This second set is used to test the results of the calculated discriminant functions, and is known as test set. It is important that the overall learning and test are independent. As a general rule, in the case of a large universe of work, independence is ensured if the training set and the test does not have common elements. This independence allows some empirical confidence that the developed classifier has the property of generalization. This property ensures that a system classifies patterns correctly that has not seen during the calculation of discriminant functions.

If once constructed the classifiers, they are tested using the test set, and test results are poor, it will be necessary to discard the test set and start again with a new classification and new sets.

Normally it is recommended to take about a 65% of the sample to construct the training set and a 35% for the test set.
Selection of features

In general it is looked for the minimum set of features that allow univocally to determine to which class every object of the work universe belongs. A bad choice of discriminating characteristics can make the system to be unnecessarily expensive and slow, or to be impossible to build a classifier to solve a problem using these features.

Every feature may require five properties:

- Economy: The precise mechanism for calculating or obtaining discriminating features (sensors, etc..) must have a reasonable cost.

- Speed: The computation time should not exceed the threshold to do it unfeasible.

- Independence: The features should not be correlated between them. One feature that relies heavily on the rest does not add discriminating information and therefore can be eliminated without implying any loss of discriminatory power.

- Reliability: Reliability means that objects of the same class have vectors that should have similar numerical values. This is accomplished if the feature vectors of a class have a small scattering. The dispersion can be measured on the diagonal of the covariance matrix. The larger the values of the diagonal are, the greater the dispersion is.

- Discriminant capacity: the discriminatory power of a feature can be described as the property that ensures that patterns of different classes have clearly different numerical values.

Classification algorithms

Here are some of the most widely used classification algorithms in the field of computer vision:

- KNN (K-nearest neighbors): The k-nn method is a supervised classification method (Learning, estimation based on a training set and
prototypes) used to estimate the density function $F(x \mid C_j)$ of the predictor $x$ for each $C_j$ class. This is a nonparametric classification method, which estimates the value of the probability density function or directly the posterior probability that a $x$ element belongs to the $C_j$ class, from the information provided by the set of prototypes. In the learning process any assumption about the distribution of the predictors isn’t done. In pattern recognition, the k-nn algorithm is used as a method of classification of objects (elements) based on training examples that are close by in the space of the elements. K-nn is a type of "Lazy Learning", where the function approximates only locally and all computation is deferred to the classification.

- **DTREE (decision trees):** A decision tree is a predictive model used in the field of artificial intelligence. Given a database, logical constructions diagrams, similar to those prediction systems based on rules, are built. These serve to represent and categorize a series of conditions that occur in succession, to solve a problem. A decision tree carries a test out, as this is traversed to the leaves, so as to reach a decision. The decision tree typically contains internal nodes, nodes likely, leaf nodes and arcs. An internal node contains a test on some value of a property. A probability node indicates that a random event must occur according to the nature of the problem. This type of nodes is round, the others are square. A leaf node represents the value that the decision tree returns and finally the branches provide the possible paths that are in accordance with the decision.

- **RTREE (random trees):** The random tree algorithm tries to solve problems of classification and regression. The random trees are a collection of tree predictors called "forest". The classification is performed as follows: the algorithm takes the input vector and classifies it by all the trees of the "forest" and as exit it returns the class to which it belongs (which has won "most votes"). In the case of regression, the response of the classifier is the average of the responses of all the trees of the "forest".
• NN (neuronal networks): Networks of artificial neurons (commonly known as as ANN) are a paradigm of learning and automatic processing inspired by the way the nervous system of animals works. It is an interconnection system of neurons in a network which works to produce an output stimulus. On artificial intelligence it is often referred to them as networks of neurons or neuronal networks.

• (K-Means) algorithm of the K-means: The k-means algorithm allows to determine the position of k centroids that distribute equitable a set of patterns. Note that this algorithm needs to know the k number of existing classes. Initially it does not know which patterns belong to each of the classes, but it is known that the sample is divided into k classes. Thus, in the first instant, k centroids are taken as random patterns.

2.6 Interpretation

This is the lattest process to be performed. It consists in the interpretation of results. There is no specific methodology as it depends entirely on the upstream processes and the desired objective. This means that depending on the objective, this process can be different to another process of interpretation with another goal.

The best way to understand this point is through some examples:

Imagine the case where it is desired to implement a program with the purpose of recognizing license plates. Say the case where the above steps provide the values and letters recognized in a particular area of the image, likely to be a license plate. The purpose of this last stage would mean verify that the values obtained correspond to a plate. For this, it could look if the elements keep the format of a plate (eg 3 characters, dash, and 4 numbers). It could also calculated the distance between the different elements. Interpretation is also understood as to what you want to do after with the results: creating a database, look for possible crimes, etc..

Another possible example would be the object recognition based on
the recognition of different parts that compose it. It may be the case we are not able to train a system capable of recognizing a particular object, but to train the system to recognize different parts of it. Therefore, the interpretation step would consist in checking that all parts that define our object have been found, and calculate that they are equal geometrically situated compared to the original object.

### 2.7 Festo Camera

One of the objectives of this project is to run the program to be developed on a Festo camera model SBOC-C-R3C. Therefore it is necessary to make a preliminary study to become familiar with the camera.

This camera carries integrated an embedded Linux and a FPGA, which allows to take images and to execute programs to work with them. This lets, in our case, to separate the processes of image processing and data processing. So, the robot can work independently and with a lower workload, since the image processing requires large CPU consumption.

The following are the general characteristics of the Festo camera:

- 752 x 480 pixels
- 60 fps
- Colour sensor
- Xilinx Spartan 3 FPGA
- IP65, IP67
- Intel XScale core running at 400MHz
- 64MByte of SDRAM and flash memory attached
- 16kBit of FRAM
- 10/100MBit Ethernet interface
2.7. FESTO CAMERA  

CHAPTER 2. STATE OF THE ART

Figure 2.1 shows the block diagram of the overall scheme for the different existing camera models.

![Camera Hardware Block Diagram](image)

Figure 2.1: camera hardware block diagram

One of the most interesting points offered by this camera is the connection via the Ethernet port. Through this connection it is possible to establish a communication with the camera and control it. The camera has special functions to be remotely executed. In this way it is feasible to have an application running on a PC and obtain images from the camera. This can be really useful if we need a more powerful processor than the one in the camera. It is also thanks to this connection, and the web application available in the camera, that we can get images by changing the camera settings from any PC. This tool is really useful to adjust the camera settings to get well parameterized images (exposure, focus, ...). Another advantage of this communication feature is the possibility of mounting a folder, on the PC, with which the camera can work likely of having it in its file system. In this way it is not necessary to transfer the programs to run in the camera each time, saving time in development and testing of software.

In addition, as the camera has not so much space memory and in the case, for example, of having to generate and store huge amounts of images for testing applications, we wouldn’t have enough memory available. A folder created on the PC would solve this issue.

Finally, it is possible to establish telnet communications with which to
execute commands and control the programs to be run. The document-
tation explaines very well how to do what was just discussed and it even
provides a program to find the IP of the camera in case of not knowing it.
Making this connection is as simple as changing the IP of any computer
with one within the range of the IP camera.

Due to that the operating system that contains the camera is an embed-
ded Linux, it is provided a specific compiler to perform cross compilations
so that programs can be executed in the camera. The installation and use
is quite simple, and there is enough information in the documentation of
the camera. It also contains information of how to compile libraries to be
added and used in the camera.

Finally, something that is very helpful: the examples. The camera in-
cludes several using examples of the most important functions such as
configuration and image acquisition. On these examples we can see how
to obtain images from a few lines of code. This helps significantly in taking
contact and control using the camera. The most relevant functions when
configuring and acquiring images are shown below.

*pCamera->Open(640*480*4);* specify frame buffer size = four images

*pCamera->SetCameraWindow(0,0,640,480);* use full VGA window

*pCamera->SetShutterTime(2000);* Setting 2000us exposure time

*pCamera->SetTriggerSource(OCamera::TRIG_SRC_SOFTWARE);* trig-
ger by software

*pCamera->SetGain(5);* set camera ADC gain

*pCamera->SetReadTimeout(2000000);* do not wait longer than 2s for an
image

*pCamera->SetAcquisitionMode(OCamera::ACQ_MODE_SINGLE_SHOT);*
start image acquisition in single shot mode

*pCamera->GetImage(&img);* get an image via the driver internal image
buffer space
Chapter 3

Analysis

As has already been mentioned, OpenCV has many functions, so it is important to know and select which can be useful and which not.

In this chapter we will show in detail the problems that arise at each stage of the project. We make a review of all the aspects to be considered and outline possible ideas to develop each point.

3.1 Problem definition

It is time to enter more in depth and explain well what every principal stage of the software to be developed consist, as well as the difficulties that will arise. After a previous study we decided to divide the software to be developed in 3 principal blocks, with the idea of being able to work independently with every block. The 3 main blocks are shown in figure 3.1.

3.1.1 Object Recognition

This is the most difficult stage of the project, and consists on the fact that our software has to be able of finding those Duplo blocks that are capable to being taken by a robotic arm. Here, the difficulty is in that these blocks can appear in infinite different structures, giving form to different objects.
We must be able to recognize those blocks that can be extracted, so that we can disassemble the brick structure. Therefore, it is not enough just to identify where the blocks are. It is necessary to keep in mind that the bricks can be of different colors, some can be overlapped by others, being these impossible to be grabbed. They can be in any position, etc. In addition, it is unknown where they can be found, the pitch of the camera, as well as the distance from the camera to the object, etc. All these variables make it to be the most important point to overcome. From nowhere we should be able to find pieces viable to be picked, in order to find their exact position in the real world through the following stages. Therefore, the purpose of this stage consisting finding some method of finding objects tolerant to all these mentioned variables.

### 3.1.2 Object Verification

Once our program is able to find these pieces, it is necessary to check if they are really the pieces that we are looking for (in our case Duplo
Bricks), and that these can be taken. It would be absurd to try to extract the coordinates of a piece that we are not able to catch. The Duplo bricks are symmetrical and of a uniform and shiny color, that can cause cases where the bricks be found in a wrong position. Let’s imagine the case to have two red bricks in parallel, to a certain distance and with sheens provoked by the lighting, where even for the human eye is difficult to recognize if the pieces are in a horizontal or vertical position. So this point is vitally important, due to the high probability of finding false pieces in the previous stage.

This stage consists also, after the verification, in the feature extraction of the piece that define it. It is not enough to know that this is a valid piece, more information is needed to be able to calculate later its position. In the case of our brick, the points that could define it, might be for example the 4 points of the corners of the cuboid. Like in the previous case, the fact of having pieces with an uniform and shiny color makes this task more difficult since the light can come from multiple directions.

The next to pictures shows the mentioned problem due to the uniform color. It can be seen how difficult even for humans is to recognize bricks position with same color (blue brick in first image, yellow brick in second image).

![Figure 3.2: Color uniformity 1](image)
CHAPTER 3. ANALYSIS  

3.1. PROBLEM DEFINITION

3.1.3 Obtaining Coordinates

The last stage consists in, once obtained the points that define a block, to calculate its exact position in the real world. This means that the program must provide with good accuracy the coordinates where to find the object. With this data the robot must be capable by means of the robotic arm to grab the piece. So, it needs to know exactly the inclination of the piece in the 3 axes, the distance between the camera and the object, etc. It is necessary to keep in mind that the idea is to work with one camera, which works with 2D, which makes more difficult to obtain information from the 3D world.

3.1.4 Festo camera

As previously mentioned, the idea is to develop software to be executed on the Festo camera. To do this, we must verify firstly that the camera supports the OpenCV libraries and compile and install them. In case of demonstrating that it is possible to work with the Festo camera, we will have to develop the software bearing in mind that it has to be totally supported by the camera. As mentioned, the idea of using this camera is to separate the image processing of the data processing, thereby freeing resources from the robot controller.

The difficulty of this stage lies in the fact that the camera has not a very powerful processor, for what we will have to monitor the workload, trying
3.2. PROPOSALS TO SOLVE THE PROBLEM CHAPTER 3. ANALYSIS

to optimize the functions as well as the memory, since the camera has just 64 MB. Another important topic is that the camera does not have graphical interface, so it will be difficult to check its performance in every stage of the software.

3.2 Proposals to solve the problem

Once we have an idea of the issues at each stage it is necessary to give now an approach on how to undertake them, how we propose to solve each stage. Note that in order to have a vision of how to solve every question it is necessary to have a general knowledge of the available functions in OpenCV.

3.2.1 Object Recognition

Knowing clearly which is the objective, and after a review of the OpenCV documentation, we have found several possible solutions for the object recognition. One of them is to use functions to obtain features that define the object. This means that these functions use different algorithms to obtain image areas with specific features. These features vary depending on the chosen algorithm, So, we can get areas where there is a high contrast between pixels, zones where there are some corners, zones where there are lines, etc. In this way we obtain patterns of the different positions of where these features are for, later, try to find them in a new image. Some of these functions are: \texttt{cvMatchTemplate}, \texttt{cvFindGoodFeaturesToTrack}, \texttt{ExtractSURF}...

Another method, found in OpenCV, is the Cascade of Boosted Classifiers Based on "Haar-like Features". This algorithm even being quite old, is very popular nowadays and it is being used in many systems for the face recognition. After a training, for example with different images of faces, the algorithm is able to identify any face, even those who have not been used in training. OpenCV incorporates different programs to use this features,
CHAPTER 3. ANALYSIS

3.2. PROPOSALS TO SOLVE THE PROBLEM

as for example the needed for perform the training. This training consists in the generation of a classifier from positive and negative images, where positives images are the normalized images (same size) of the object to be recognized and negative are images where the object to be recognized does not appear. The classifier can contain different stages, being better having more. There is a lot of information about how to work with this method in the libraries documentation. The advantage of this method is that we get patterns that describes the object automatically from the samples that we want, while other functions do not offer this possibility.

3.2.2 Object Verification

The idea is, that once we know the zone where the piece to extract is placed, analyze it in order to check if it is or not such object. One of the conditions that must be fulfilled is that the 8 holes that the bricks contain have to be identified. So, it would be confirmed that the brick does not have any other block covering it, being capable to be caught. For achieving this, we will use the *cvHoughCircles* function, which seeks circles in the picture. To do this, fi the function *cvCanny* is applied, which from a black and white image (gray scale), colors in white those areas where there is a high contrast between pixels, and in black those areas in which the contrast between pixels is similar, to detect in this way the edges. As our Duplos bricks contain holes, we assume that using these functions we can find these circles.

Once the piece is verified, we intended to find points that define it. The aim is to identify the corners of the block, as well as trying to recognize the side edges. For this, there are available several functions. Here are some of them, as well as a short description:

- **cvHoughLines2** This function as well as the *cvHoughCircles* function, uses first the *cvCanny* function, to then look for lines. This would make it possible to detect the edges of the blocks.

- **cvCornerHarris** This function is also based on the *cvCanny* function, and
3.2. PROPOSALS TO SOLVE THE PROBLEM CHAPTER 3. ANALYSIS

its function is to look for corners. The idea is to combine this function with the previous one, in order to determine better where the edges begin and end.

cvFindContours This function consists in another method to find the previously commented. It is also based on the cvCanny function and it finds closed contours. It also offers the possibility of sort them in different ways being able to filter out unwanted contours. The advantage of this function is that it opens the door to the use of other functions that work based on this. Thereby, it would be possible, for example, to use a function that calculates the area of the contours and in the case of finding 8 similar areas determine that the 8 holes have been found.

3.2.3 Obtaining Coordinates

On a first thought it seems impossible to associate an image with the real world. How, it will be possible to extract 3D data from a 2D image. The idea is to, thanks to the previous stage, to find points that define the object in order to compare them with the same points of the object in the real world. After reading the documentation [33], we found 2 functions that do just this.

These functions are able to get the rotation matrix and translation vector of a previously defined object, and describe the relationship between the object coordinates system in relation with the one of the camera. To define the object means as said, to describe, for example, the position in which the corners of our brick on the 3 axes are. The functions calculate the position by the association of each object 3D point with same points found in the 2D image. However, to do this it is necessary to calibrate previously the camera. The calibration is done by capturing images of a chess board with known dimensions for, after this, a specific function calculates the intrinsic parameters and distortion coefficients of the camera. This is to correct image distortions produced by the camera lens and get
the parameters that relate the pixels with the units of measurement of the real world.

The two available functions to obtain coordinates are: \texttt{cvFindExtrinsicCameraParams2} and \texttt{cvPosit}. Both functions are nearly equal. The difference is that the first can work with coplanar points of the object or not (points in the same plane), while the other needs not coplanar points. Another difference is that the second function calculates the data several times, using as a basis the previous calculation results, offering therefore more precision. We will choose this second function, which makes the previous stage to take more importance due to the need to find points of different planes.

\subsection*{3.2.4 Festo Camera}

To install the OpenCV libraries first of all we returned to the camera documentation. On it, we found a way to compile these libraries and how to use them. So, the first step we will try to give is follow the stages described in the documentation. The next step will be try to install a higher version than the one described in the documentation of the camera, as it describes the steps for version 1.0 and the current one of OpenCV is 2.2.

In order to solve the problems that could arise by not having a graphical interface in the camera, we decided initially to develop the software on a PC. Another way would make the camera to save the images during the execution of the program to check the obtained results, but this would waste time. In addition, in this way we would accelerate the development process, since implementing the program on a PC is faster than in the camera due to technical differences. To prevent possible conflicts between the camera and the PC we will occasionally compile the program to be executed in the camera.
3.3 Conclusion

Having defined in depth the difficulties at each stage, and after an intense documentation of OpenCV libraries, we can pass to the next stage: implementation. Even having some ideas on how to solve each stage it is quite possible that some complications arise, having to modify the procedures to address each one. We must consider and be aware that an image even be of a same object, may be completely different from another. And to endow a robot with enough intelligence to adapt itself to each possible situation is a difficult task.
Chapter 4

Implementation

In this chapter we explain how each stage has been solved and the problems we have run into. We show the steps that have been followed, and which were necessary to make in order to understand how all the available functions in OpenCV work. We will see examples of why some functions are valid and some not.

4.1 Algorithm Design

As in the preceding paragraphs, below we explain the taken solutions for each step of the project, just to have a clear structure of what each stage does. The implementation is done on a PC that allows to see the results of each stage fast and graphically.

4.1.1 Object Recognition

We decided to start testing the feature extraction functions because they are easier to use and they involve less preparation and less workload. First thing that we note using these functions is that we obtain only a few good points that define the object. In addition, the found points are not repetitive and its position vary depending on the position and lighting of the piece, also affecting the color of these.
We find here the first problem, due to that the piece is of an uniform color and symmetrical. In the figure 4.1 we can see how the points found by the function that are located in the holes are not exactly in the same place as in the rest, even being the bricks in a very similar position. In addition we can see that in the parts where the background is not white there are a lot of points, which could mislead when searching for an object. All of this complicates the task of find a pattern that defines our object. We would have to create large databases with points extracted from hundreds of pictures and then compare them one by one to try to find an object in an image. We have to discard this method because it is impossible to reproduce all the conditions in which the object can be found. In addition we have to consider that these functions are more focused on planar objects. For example, we can see in Figure 4.2 the result of using the function SURF (Speed-Up Robust Feature), to try to find the object in a scene. The figure shows only 3 coincidences, where only one of them is correct.

![Figure 4.1: Extracting features](image)

**Haar Classifier**

Discarded these methods, we proceeded to implement the function `cvHaarDetectObjects` with the system Cascade of Boosted Classifiers Based on "Haar-like Features". As mentioned, this method is based on the Haar-like
features detection, describing the existence of orientation of the contrast between regions within an image. A set of these features can be used to encode the contrast exhibited by the human face (or any other object) and its spatial interrelationship. The Haar-like features are so named because they are computed similarly to the coefficients of the Haar wavelet transforms. More information in: [32].

We decided therefore to perform the first step to train the classifier: to capture positive images. Remind that positive images are standardized images in size, where the object appears. OpenCV has a program to normalize the images from a file which describes the name of each image and the exact coordinates of the object's position. In order to speed up the task of image obtaining and definition of the coordinates, we decided to create a specific program, because the ones we found on internet did not fit to our needs. The program has a graphical interface that displays each frame of a video. In our case, several videos showing the object from different angles, distances and lighting. The program lets you to select the image region where the object is, using a square controlled by the mouse. One recommendation in creating positive images is to avoid backgrounds with different colors, in order to obtain clearer pictures adjusted to the object. Because this the program allows to adjust the size of this square, as well as different actions such as move to the next frame with-
4.1. ALGORITHM DESIGN

CHAPTER 4. IMPLEMENTATION

out saving, repeat the frame if there are more objects in the image, and so on. The program creates automatically a text file with the descriptions previously commented and the selected frames to be used, with the same name used in the text file (see figure 4.3). Once obtained a good number of images, we create the normalized image file using the OpenCV tool: createsamples.

Previously to start the training, negative images should also be taken, those that do not contain an image of the object but on the same scenario where it is likely to be found. Training this way the system would have to avoid false positives. For this purpose, we also developed a specific program. This generates image files from a video: Two images for every frame, one in color and one in black and white. Once everything was ready, we created the classifier with the tool: haartraining. The generation of the classifier takes several hours (or days in case of a good classifier). The first tests we did, showed that the classifier was weak, because it found many false positives. After several failed attempts and after an optimal configuration of the search function, we obtained a good result, as shown the figure 4.4.

Trying to apply this classifier to a less ideal image and with more bricks, we began to see that we obtain poor results. The point was that we made the training from a single block, where all its sides were visible, which
does not occur when more blocks are put together. Also, checking the documentation we found that the training was to be done from similar images to obtain a similar pattern among them. Our brick, being rectangular and trying to find it in different angles and positions, made impossible to generate a good pattern.

We tried another option, which gave way to the success of this stage: train the classifier to detect the holes of the blocks (see figure 4.5, normalized image). The holes, being circular, present a more regular pattern than the bricks themselves, even putting them in different angles. In addition it presents the advantage that if a hole is covered by a block, it is not found. Therefore this block can not be catched. The previous model wasn’t able to identify covered blocks. So, we repeated the classifier, but in this case with pictures of the holes. We used arround 2800 of positive frames and 1000 of negative frames. With just a few stages of the classifier we got a great improvement but with a high number of false positives. This gave us the confirmation that we were going on the right direction and we decided to continue.

Figure 4.5: Positive image sample
Once we reached the 20 stages, we decided to stop training. The computing time per stage ascended to 2 days and the results were already good enough to continue working. The following image 4.6 shows the result obtained by using the new classifier with different stages using the OpenCV program: performance. This tool allows to calculate the percentages of success and failure of the classifier and display the results in images. We can see how in the left image (classifier with 10 stages), apart from not finding all the holes, there are numerous false positives. While in the right image (20 stages) all holes are found without any false positive. The tool performance also generates results in text format, such as figure 4.7 where we can see the results of each used image (in this case 3, with a total of 96 holes found). We see that with the 20 stages classifier we reached almost the 95% of effectiveness. We did just the test with 3 pictures to get a general idea of the classifier performance. The software needs as in the case of creating normalized images, a file describing all the positions of the holes in order to check if they have been found in the correct position or not.

Having a good classifier, we created a program to process videos and to apply the search objects function to see its dynamic behavior. The aim was also to see the effectiveness of the function by changing their settings. Since the size of the object to find in the image is unknown, the function searches for the object through a window. This window moves
through all regions of the image, also it changes its size on every scan. This is achieved because the classifier can be scaled easily, avoiding the need to change the image size. These parameters can be configured: the initial window size, the size increase percentage, the minimum number of valid coincidences to give an object as found, etc.. After running the developed program, we saw that the function worked well, although, it didn’t find always all the holes and in some cases it gave false positives. We also observed that in images where the object was far away, the percentage of found objects decreased considerably. We decided to implement two improvements, to remove everything on the image that didn’t interest us and to increase the interesting parts.

**Background segmentation**

The idea of erasing parts of the image would serve to avoid false positives. As we were working with 4 known color objects, we decide to implement functions to clear any different color found in the images. Searching about this topic, we found that the easiest method for color segmentation was to work with HSV color models. This format encodes colors with three values:

**Hue** Contains the value of the color encoded in 360°. The image 4.8 shows the equivalence of the color depending on the angle at the outer ring.

**Saturation** Contains the value of color saturation. The higher, more intense is the color and vice versa.
Value Corresponds to brightness value. The lower, more darker is the color.

![Figure 4.8: HSV model](image)

Due to the absence of a specific function that performs the color segmentation, we implemented a loop goes over all the pixels of the HSV image (previously converted with the `cvCvtColor` function), comparing the values of each pixel with the values that define our colors. If we find a pixel outside the threshold, we paint it in white, eliminating unwanted background. On the first tests we integrated, in the program, bars that allowed us to change dynamically the 3 HSV values and see the effects on the image with the purpose of configuring the parameters correctly. We saw how it was not possible to remove the entire background. We saw also that we deleted some pixels of our object. We decided then to use the erosion and dilation functions: `cvErode` and `cvDilate`. Alternating these functions, we achieved to eliminate the non deleted pixels in areas where the remaining pixels had been removed and do not eliminate the pixels that have been deleted in areas where the remaining pixels are not deleted. One of the characteristics of these functions is that they need to work with black and white images (not grayscale images). Therefore, to perform the segmentation, we had to create firstly a temporary image and paint with black color the considered background and with white color the considered our object. Having applied the erosion and dilation functions, we used the resulting black and white image as a reference to erase the background of the original image. Figure 4.9 shows the steps performed to segment the background. As a last measure, and with the idea of facilitating the task in
the next stage, we decided to perform the segmentation in 4 stages. That is, segmentate the 4 colors of our blocks in 4 steps. In the image 4.10 the result is shown.

![Figure 4.9: Segmentation steps](image)

Once all improvements were done, we saw how the false positives were eliminated, those that we previously found in regions where the object wasn’t. As the program still had problem to find the holes in small images, we implemented another improvement: a zoom.

**Zoom**

On the OpenCV libraries we can find the `cvPyrUp` function. It allows to increase the size (x2) of an image, thus obtaining a new image with twice
of pixels. We had an initial problem. When we performed a zoom x2 or x4 the memory consumption increased a lot. Consequently, the search object function worked more slowly because now, we had a bigger image where looking for our object. Therefore, we decided to escalate only those regions where the objects were supposed to be. To perform this task we used the cvFindContours function. This function works with the intermediate B&W image of the color segmentation stage. It gives as a result the points that define the contours of white pixels. Having identified a contour, we calculated the maximum and minimum points on the axes X and Y, to define, thereafter, a rectangle containing that contour inside. Then we copied that zone of the final segmented image and we pasted it into a new temporary image, to which we applied the zoom. As a filter, we checked if the temporary images were between a specific range of size, in order to decide if they had to be amplified or not. The following figure 4.11 shows how the contour is delimited as well as the rectangle, closing so the yellow piece. We drew the contours on the original image to see if the program was running well or not.

![Figure 4.11: Zoom](image)

**Final results of Object Recognition**

Below, we show (figure 4.12) one of the images used to test the final result of applying all the steps previously exposed on the object recognition stage. As it can be observed, after processing each color separately all the bricks holes were found.
4.1.2 Object Verification

After finding a supposed object, we needed to verify that it was the object that we were looking for, and then, to extract points that could define it. Initially we tried some of the functions described in paragraph Analysis. We studied these functions with different images of the previous stage in order to see quickly the response to these functions.

Since most of the functions we wanted to study depended on the cv-Canny function as a previous step, we found convenient to start working with it to see how it worked and what results we could obtain. Remember that this function is used for the detection of edges in an image, being edges the areas where there is a high contrast. To do this the function performs first the derivative in the X and Y axis on a gray scale image, giving as a result the values of the slopes where there is a tonality change. Then it calculates the second derivative in the 4 directions to determine the maximums where there is a greater contrast. Once obtained these points and with the help of two configurable limits, a maximum and a minimum, it determines if the point has to be considered an edge or not. If the value of a point is greater than the upper limit, then it is considered as an edge. However if the value is below the lower limit it is not considered as an edge. And finally, if the value is between the two limits, it will be considered an edge only if this has a point considered edge as neighbor.
We implemented a program to test this function with the possibility of changing dynamically the limit values in order to see the effects on our object. Figure Canny shows the result of applying different values to these limits from lowest to a highest value. The ratio between the upper and the lower value is 2:1 as recommended in the documentation. We can see that depending on the values of the limits, the results are quite different.

We concluded that to obtain better contours to work we had to apply a lower limit value, having the disadvantage that in the image, particularly where the object was, it appeared unwanted noise. And, if we wanted to remove this noise, it was then necessary to increase the limit value, with the disadvantage that contours that could be useful were also removed. Reach to a midpoint would be almost impossible. It had required a dynamic value adjustment, taking the results as a reference. A possible solution could be to perform a routine changing the values and looking to find the points of our interest, but that would imply a high CPU load, so it was discarded. We saw that this function worked very well in the case of having areas with high contrast. If we have a look at the top left of the images, the contours of the chess board are distinguished perfectly in all cases. As we anticipated in section Analysis, the fact that the pieces were bright and of an uniform color, would make this stage complicated. Even so, we decided to try to test the rest of the functions, although we anticipated that good results could not be achieved, because the vast majority of these functions depend on the cvCanny function.

In the pictures below we can see how effectively, we didn’t get good results by applying the functions that were intended to use at this stage.

cvHoughLines2 Figure 4.14. We observed that depending on the function parameters configuration, more or less lines were found, and also, even in areas where it should find long lines some short lines appeared. Another problem was that according to the configuration, the function could detect hundreds of lines making difficult the verification task. We got only minimally acceptable results in the case of changing the parameters dynamically observing the results. Something that in practice
can not be done because the software has to be autonomous.

\textit{cvHoughCircles} Figure 4.15. The obtained results demonstrate us how complicated it is to find appropriated circles to work. Although, finding circles where they really aren’t doesn’t difficult our task, due to with the previous step we know approximately where they are. The idea of using this function is to find the exactly position of these circles in order to be able to calculate the real position of the block with more accuracy. But, as the image shows, we didn’t find them in the exactly position, and neither we found the 8 holes. Another problematical of the function is that it has a lot of configuration parameters that difficults the calibration step. Also, we observed that this function doesn’t work good with ellipses. And this is something that appears
Figure 4.14: Hough Lines

when the bricks are inclined, the circles seems ellipses.

Figure 4.15: Hough Circles

*cvCornerHarris* Figure 4.16. We can see that some of the found corners are useful for us. But if we take a look, we can observe that the function has found a lot of useless corners. It also finds points near to the lines, which would suppose to get false intersections in the case of verify jointly lines and corners. With the addition that the function doesn’t detect always all of the interesting corners. Also, with the high number of lines and corners found, we would have to develop a very complex algorithm.
Figures 4.16 and 4.17: Corner Harris

cvFindContours Figure 4.17. Apparently, a lot of good contours are found. But we have to keep in mind that we used previously the cv-Canny function, which we configured with the optimal parameters manually found. In addition, false contours are found. We can see how some of the contour holes found are not closed.

Figure 4.17: Contours

In conclusion, we discarded the use of these functions due to the bad obtained results. In the case of using them, we would need to configure
well all of the functions in order to work always with the same conditions. It means, working always with the camera in the same position, with the same illumination conditions, etc. Something that is not possible, because the idea is to be able to find the blocks in different environments, places and conditions. Another important reason is that it is very difficult, in the case of finding good points that define our object, to obtain always the same points. This is totally important in order to be able to calculate later the position of the brick in the real world. Remember that the same points are needed to use the position calculation function.

**Solution**

Finally we decided to work not with the idea of finding new points, but with the points that we have found previously. The principal idea of this Object Recognition stage was to find those regions where an object was placed, with the help of the color segmentation and the holes finder. The localization of the holes was at the same time a definition of points that could identify our object. How to get these points from a square drawn over the holes? The solution consisted in finding the central point of this square and draw it in a new image, obtaining so a map with all the found points. The figure 4.18 shows this map.

![Figure 4.18: Holes](image)

To filter bad maps, we processed only those images where at least 8 holes have been found, due to be the number of the holes that our brick had. Now, to verify if we had a brick or not, it was needed to implement an
algorithm to check if the points were aligned forming 2 parallel lines. We needed to do this verification in order to be able to discard false points in the map, as well as to be able to check if we found more than one block in the same image. Also we needed to get the position of the verified points.

**Lines Detector**

To do the verification we worked with the idea that the points had to be aligned. Our objective was to find lines passing through the points. Firstly, we tried to use the `cvHoughLines2` function, since one of its parameters defines the allowed distance between points to define a line. In the initial tests we obtained the expected results, but we found that sometimes the points were not completely aligned. To solve this, we tried to increase the size of the drawn white points, as the function worked trying to join white pixels. Doing this, we should be able to find more lines. After changing the size of the white points we were able to find more lines even having misaligned points. In counterview, the function found more than one line joining the same 4 points (4 points for each line), doing necessary to implement a new verification step to check if the lines contained or not the same points. Also, the configuration task was extremely complicated, because almost always the function found lines joining wrong points and it was easy to find more than 100 lines. Finally we decided to implement a different algorithm with a different approach.

We chose to do this manually. We implemented a new algorithm that from taking 2 random points it calculated the line function ($Y = m \times X + b$), in order to search points that are member of this line. In the case of finding a line containing 4 points, the algorithm set it as a valid line, and continues searching more. To avoid to charge the CPU with a lot of work, and to increase the efficiency we implemented different filters. The first consisted in deleting false positive points, those points that were far away enough to suppose that they didn’t belong to any line. To do this, we calculated first the average distance between each point with its nearest point. After calculating this, and after applying a tolerance, we deleted these far points.
The next step consisted in to take 2 points to extract the line function. As a filter we discarded the, by the previous line function extraction, used points, to avoid to repeat them. We also discarded the points that had been validated by a validated line. To give some tolerance to those points that belong to a line but were quite misaligned, we added a limit of distance. Making possible to solve little position errors caused by the 4.1.1 stage. We saw that the points that were more horizontal than vertical aligned had more possibilities to be validated in a line thanks to this limit. This happened because, to see if a point belonged to a line we used the position of the point in the $X$ axis in the line function $Y = m \times X + b$. We compared then, the result of $Y$ with the position of the point in the $Y$ axis applying the tolerance. In a vertical line, the possible misalign affects only to the $X$ axis. To solve this problem, we decided to use the function $Y = m \times X + b$ with the more horizontal lines, and the function $X = Y - b/m$ with the more vertical lines. To know if a line is more vertical than horizontal we used the slope of the line function.

Once we found 2 points that belong to a line (2 points that define the line + 2 found points = 4), we checked if the points were to a distance to each other of: average point distance (previously calculated) ± tolerance. In a positive case, we set the points as validated and we put them into a variable.

After finish searching lines, it is necessary that these lines could define a rectangle as the surface of our objects are rectangular, verifying that we found a brick. To do that, our program takes a verified line as a reference, and tries to check some features with the rest of lines, like in the previously step (lines searcher). These features are: check if the distance between the first point of the line one (reference) and the first point of the second line is the same distance that between the second point of the line one and the second point of the second line. In the case of having a possible good line we filter the distances. We use the the “average points distance” calculated in the previous step applying some tolerance. In this way we find those lines that are parallel or almost parallel. With the filter, we avoid to give as valid those parallel lines that are far apart.
The last step is to sort the points that will serve us for the extraction of coordinates. We take only the 4 external points of the rectangle. The numbering of each point is important, in order to be able relate correctly each point with the points on the real object. These points have to be always in the same position.

**Final result of Object Verification**

The implementation of this stage give as results the images that are shown below. In the figure 4.19 shows the complete verification process. Starting with the location of the holes and generation of points. Search and verification of lines. And finally, verification of found rectangle. At the same time displays the external points that define our object. Figure 4.20 shows by way of debug mode, that the yellow brick has been found, at the same time indicating the point 1 of the object, drawed with a white dot. The only way to know that we are always defining the same points is through this augmented reality.

![Figure 4.19: Verification](image)

The following figure 3 shows another case of the lines detector. We have used an image where only the background has been segmented. And where only a few holes have been found, in order to see if the lines searcher work well or not. We can see how in this case we verified only one brick, even having found more lines. We take for good the algorithm and we proceed to continue to the next stage.
4.1.3 Obtaining Coordinates

Last stage of the software: to get the coordinates of the object. As we mentioned, to perform the task of obtaining the coordinates of the object in the real world we could use two functions: \texttt{cvPosit} and \texttt{cvFindExtrinsicCameraParams2}. Both are useful to obtain the extrinsic parameters: rotation matrix and translation vector. The rotation matrix define the angular variation of each axis of the real world in relate to the coordinate system of the camera, while the translation vector defines the offset between the center point of the camera and the object coordinate system in the real world.

The first gives better results due to that it performs the calculation repeatedly, taken the previous result as reference. The problem is that this
function requires points that do not belong to the same plane, something which has not been achieved in the previous stage. We decided to try it anyway in order to see the power of this function and determine whether we were going on the right direction. We tested it taken one example found in the documentation of OpenCV [30]. Figure 4.22 shows the result of using the function to get the position of a cube and a LEGO piece. We took manually the reference points to define the object in the image (4 points). They are the red colored circles. We took them manually since we wanted to study the function regardless of the previous stage. We can also see how once we get the position data of the objects, we can calculate the position of other points that define the object in the real world and project them onto the image (augmented reality). Even we were able to draw the area where the robot has to catch the objects. Having an idea of the functionality and having this good results we decided to start using the second function.

![Figure 4.22: cvPosit](image)

Before being able to begin the implementation of the code to calculate the coordinates with the `cvFindExtrinsicCameraParams2` function, it was necessary to perform a camera calibration. This function needs the camera’s intrinsic parameters, which define the physical features of it: position and inclination of the sensor, focal length and lens distortion. All of them are then used to relate the real-world units to image units (pixels) as well as

---

1 The image of the cube is provided in the downloaded example.
correcting distortions. Starting from a OpenCV example, we implemented the necessary code to get the calibration file with the commented data. We developed the code separately from the main program, since it was only necessary to run it once. It would only be necessary to execute it again if its features would change: image size, lens focus, changing to a new camera. Remember that to perform the calibration, it is necessary to have a chessboard pattern and set the squares measures in the program as well as the number of corners that are found, both horizontally and vertically. The calibration program uses the `cvFindChessboardCorners` function to find the inner corners of the chessboard pattern in order to relate them later with the position of the real points. Finally the `cvCalibrateCamera2` function is executed, obtaining the intrinsic parameters. Note that it is possible to calibrate the camera with any image where points could be found with good accuracy and then compare them with their real position. Although, it is strongly recommended to use the chessboard method, since OpenCV provided functions are well optimized. Once the code was implemented, we executed the program to see the results. The images on the left of Figures 4.23 and 4.24 show an image of the chessboard pattern. As we can see, we applied a fisheye distortion simulating (in an exaggerated manner) the distortion that the lens of a camera can cause. On the right we can see the corrected image by using the distortion parameters obtained after the calibration, using the `cvRemap` and `cvInitUndistortMap` functions. Also it can be observed that a 3D geometric figure has been drawn. This was previously defined in the program. It is possible to place it thanks to the extrinsic parameters obtained from the chessboard position, that the `cvCalibrateCamera2` function offers. With the generated files we were capable then to work with Duplo pieces.

The first step was to define, in the software, the points that define the object in the real world. For this we used Duplo brick drawings found on the net. Once the position of the holes was defined, we fixed a central point to use it as a reference point to calculate the distance between the camera and the object in the three axes. Similarly, we defined the points of the cuboid slightly larger than the original piece to have some tolerance.
We had to take into account that the positions of the points that define the holes could be somewhat imprecise. After this, we related each found point with the corresponding image points in the actual piece. We have already discussed the importance of identifying well each point. Finally, we applied the `cvFindExtrinsicCameraParams2` function. Using the rotation matrix, the translation vector, the intrinsic parameters and the `cvProjectetPoints2` function, we drew the results on the image. The drawing of the 3D object of the image was used to debug, and to see the accuracy of the obtained data. On the terminal console we show the values generated by the calculation function of the extrinsic parameters and the central point position of the brick. These data correspond to the X, Y, Z coordinates of the piece related to the central point of the camera. These values are obtained manually calculating the angular variation of each point by the
rotation matrix and adding the offset by the translation vector. Note that in the coordinate system of the camera’s the Y axis values are positive if the point is below the central point. Figure 4.25 shows the final result of all the above mentioned. It also shows the position of the central point of the piece (red dot) of the blue block in relation to the central point of the camera. As the camera has been calibrated in cm units, the results also are in cm.

Figure 4.25: Coordinates

Figure 4.26 shows more examples of the projection of the 3D objects on the image. These projections show that the calculated data are quite accurate, since the DUPLO bricks are inside the drawn cuboid. In order to verify whether the coordinates data provided were correct, we took different images measuring the distance between the camera and the objects, for later, compare it with those calculated by the camera. As the obtained results were satisfactory, we ended this stage.
4.1.4 Software structure

Figure 4.27 shows the basic structure of the developed software. It consists on a main block, where major decisions are taken and that controls the flow of data sent and received from external functions. The other three blocks correspond to each of the steps discussed in this section.

4.1.5 Festo Camera

Once the program was operative on the PC, it was time to bring it to run on the Festo camera. The first step was to verify that the camera handled the OpenCV libraries. Once verified the software had to be adapted.

Compilation

On section Analysis we mentioned that the first step would be first to try to install version 1.0 and later 2.2. But as the PC software was developed with the latest version we decided to do the same with on the camera.

OpenCV’s official website describes the steps to follow to compile the libraries for all platforms, except for embedded systems. We met with the first problem, although there were a lot of information on the internet.
While we were looking for information on how to compile the libraries for our camera we found a very important detail. It specifies [31] that GCC compiler version must be at least version 4.3 and the Festo camera uses the version 3.3.2. So we discarded to use the latest version of OpenCV, which endangered that any of the used functions could be not available in version 1.0.

The method of installation of version 1.0 is somewhat different than the current version, but we had enough information, in the camera documentation as well as in the network. The only thing we had to do was to configure
the installation, telling that the system where it had to run was a platform with embedded Linux on an ARM architecture. This, and the configuration of the cross-compiler location was done using the commands:

```bash
./configure CXX=/opt/sbo/gcc332/bin/arm-linux-g++ --host=arm-linux --target=arm-linux --prefix=/opt/sbo/gcc332 CPPFLAGS=-I/obt/sbo/gcc332/include.
```

After installation, we moved the compiled libraries to a new folder inside the camera. Then, we had to configure the `ld.so.conf` file, adding the path of the new libraries and run the command `ldconfig`. Doing Thus, when executing a program, would make the operating system to know where to find the libraries. The first tests we ran with simple programs produced an error, preventing the execution of such programs. At the end we saw that the problem was caused by the lack of some libraries. We solved it by adding to the camera the libpng\(^2\) and litiff\(^3\) libraries, previously compiled in the same way that we compiled OpenCV.

Software adapting

After verifying that libraries worked well in the Festo camera, we encountered the problem of not being able to dispose of it for the next stage of the project. This required to study well the methodology in this stage. To be certain, what we had to do, was well done.

It was time to adapt the software to run it in the camera. By working with different versions of OpenCV it could happen that some of the used functions do not exist or could have changed. After performing the first cross-compiling we saw that it appeared only an error in one of the functions. Luckily we had only to adapt it, without the behavior that we wanted to achieve would be affected.

On initial testing we saw that the camera didn’t support the use of functions dependent on the `highgui` libraries. These correspond to functions that interact with the GUI, that the camera do not has. So the first adaptation consisted in removing all these functions. To have a flexible program

\(^2\)http://www.libpng.org/pub/png/libpng.html
\(^3\)http://www.remotesensing.org/libtiff/
to be usable on both, the PC and the camera, we added additional statements such as `#ifndef FESTO_CAMERA` so that the compiler could take into account or not some code depending on the platform.

Once the program was compiled without any errors, we began to study the memory consumption, as the camera has limited resources. As a first study, we decided to monitor the usage memory with the Task Manager of the operating system. We saw immediately how the program consumed large amounts of memory, much more than the one available in the camera. In addition, the consumption was increasing in each program cycle, arriving to the limit of available memory on the system until the program closed due to lack of memory. We decided to put stops in various parts of the program, to study in which moment this increase in memory consumption occurred. After a while, we got to determine the problem. The cause was the non-release of memory consumed by the temporal images created in each phase of the program. We developed the program placing the definition of the variables within their area of its usage. In this way we were able to identify properly the variables in each stage of the program.

The mistake was do not consider that the creation of the variables of images was done by an internal function of OpenCV. This function allocated memory each time it was called. Calling the same variable in each new cycle of the program, the function reserved new memory, and as expected, it was reserved in a new memory location. To solve this problem, we added the memory release functions in the points at which the images were no longer needed. Our idea was to use the images only when they were strictly necessary, due to low available memory. We tried also to optimize the use of some variables and provided functions to reduce memory usage. After implementing these improvements, we saw how the program remained stable in memory consumption.

Another aspect that we observed, was that given the large number of temporary images needed, the consumed memory was somewhat elevated. This was especially relevant when working with large images. So we decided to work with smaller images, which offered equally good results.
Finally, we added the necessary code to take images from the camera (so far we did from files) and adapt the data to work with OpenCV. The use of these functions had been checked previously.

### 4.2 Conclusion

During the development of software we were able to study in depth some of the functions provided OpenCV and we could see its potential in each of the stages. At some points it was really difficult to find an optimal solution, especially due to the characteristics of the object to find, which has greatly hindered the search functions process. It has been necessary to make many test programs, even more than those mentioned here. However, we have been able to obtain a functional program, to conduct the proposed task. Although the final program has not been yet tested in the camera, we expect it to function properly due to the methodology that we have followed during its development. (See Code on page 75)
Chapter 5

Outlook

In this section we present a few ideas of possible steps to take after the completion of this project. Let us remember that this project is a first study on how to use the artificial vision, applied to robotics, for the automatic disassembling of objects.

5.1 What’s next

The first step would be to study more thoroughly the results obtained by our software. It would have to be verified if the robot can pick up a piece with enough accuracy or not, since that was the main objective of the project. After obtaining results, it would be advisable to consider the need to optimize or not a section of the program.

After completing these points and to move forward, it would be necessary to study the feasibility of using this software for the disassembly of other objects. One of the most important points, is to avoid to implement a specific code for each piece that we want to extract from an object, since it would be needed to process too much information. The section of object recognition is hardly reusable, but the verification stage and calculation of coordinates can be profitable. It could be done, for example, that the first stage of recognition would give as results always aligned points, to define the object with different geometrical shapes. Then in the verification
CHAPTER 5. OUTLOOK

5.1. WHAT’S NEXT

stage, the algorithm could seek these different geometrical shapes and re-
late them to a particular object defined in a database. This could be done
thanks to that the lines and rectangles search algorithm is easily adapt-
able. With the current version and with a few changes, we could work to
identify Duplo bricks with different number of holes. Then, once identified
an object, the coordinate calculation step should only compare the points
obtained with the points of the real object of the database. Note that the
robot would also need to have, in the database, information about how to
extract that piece.

The most complicated point to streamline and reuse is the first step, ob-
ject recognition. Remember that due to the difficulty encountered during
the object recognition step, we resorted to use a function that requires prior
training of the object. Such training requires thousands of pictures both,
positive and negative, for proper operation. Get such amount of positive
images of each object requires time consuming. One possible improve-
ment would be to automate this process. This would be possible if, for
example, we would put the object to be found within a recognizable frame-
work for then, segmenting and resizing the image. Or perhaps, it would be
enough to capture photos or videos of the object on a characteristic back-
ground, as chroma, to then with a program, segmentate the background
color and find the object contour. So the object could be cut and saved in
a new image. This would have the advantage that it wouldn’t be needed to
get the object coordinates. It would be enough to know the dimensions of
the image. Remember that the positive images of the training need a file
describing the position and size of the object within the image. Therefore,
if a picture would have the object well framed, the position coordinates
would be $X = 0$ and $Y = 0$, while the size would correspond to the size of
the image itself. Although it must be keep in mind that even automating
this process, the CPU load required to try to find an object would be high,
since we should look for all the objects of the database. With the addition
of that this search algorithm needs to process large amounts of data. It
would be necessary to make a good optimization of the program.

Another possible solution would be to find another object recognition
5.1. WHAT’S NEXT

method, more flexible and reusable.
Chapter 6

Conclusions

Once the project is finished, it is time to make a evaluation of the work done. The feeling we have is good. We have been able to develop a functional program that meets the proposed objectives: To get the coordinates of Duplo bricks in order to make able a robot to disassembly structures.

At some points of the development we thought that it was not possible to do some tasks. We had to think to the fullest and take ways that we discarded originally. The main problem was not having ever worked with the processing of images. We are really very grateful of the existence of OpenCV libraries. They allow a "rookie" to do authentic and real magic. One mistake we made was to start working with not real images (3D models of the object). The functions that we used initially had a completely different behavior using them with real images. This was a big bump, since it supposed to change almost entirely the focus of the project. The point where we needed more time was undoubtedly the object recognition stage. Our idea was that this stage would give us an approximation of the position of the object. But finally this stage was in charge of giving the information with which we would become able to calculate the coordinates. We had to do a lot of tests until we could reach an optimal result. Especially with the training topic, which is vital.

As mentioned at the beginning, this project was created to explore the possibility of disassembling objects automatically and through the image
CHAPTER 6. CONCLUSIONS

processing. Although there is still much research work ahead, this project is a seed, which may provide new studies, and set an example for future projects. May be to focus the project on something as concrete as the disassembly of Duplo pieces makes, as we mentioned, difficult to reuse all the developed software. But it gives an idea of which ways to take. As in any research, we need something to start from. It is important to start with "simple" and concrete things in order to be able to evaluate the immensity of that what we want to achieve.

We have discovered the power and application of the image processing both, nowadays and for the future. If we have been able to get 3D data from a 2D image, everything is possible.

"Sciences have bitter roots, but very sweet fruits" \(^1\)

\(^1\) Aristotle (384 BC-322 BC)
List of Figures

1.1 Duplo brick ................................................. 3
2.1 camera hardware block diagram ............................... 24
3.1 Software main blocks ........................................ 27
3.2 Color uniformity 1 ........................................ 28
3.3 Color uniformity 2 ........................................ 29
4.1 Extracting features ........................................ 36
4.2 SURF Matching ............................................. 37
4.3 Positive images capture .................................... 38
4.4 Brick found ............................................... 39
4.5 Positive image sample ..................................... 39
4.6 Performance ............................................... 40
4.7 Performance ............................................... 41
4.8 HSV model ............................................... 42
4.9 Segmentation steps ........................................ 43
4.10 Color segmentation ....................................... 43
4.11 Zoom .................................................. 44
4.12 Final result O.R. ......................................... 45
4.13 Canny .................................................. 47
4.14 Hough Lines ............................................. 48
4.15 Hough Circles .......................................... 48
4.16 Corner Harris ............................................ 49
4.17 Contours ............................................... 49
LIST OF FIGURES

4.18 Holes ................................................. 50
4.19 Verification ......................................... 53
4.20 Verification ......................................... 54
4.21 Verification ......................................... 54
4.22 cvPosit ................................................. 55
4.23 Calibration ........................................... 57
4.24 Calibration ........................................... 57
4.25 Coordinates .......................................... 58
4.26 Projected Points ..................................... 59
4.27 Software structure .................................. 60
Bibliography


[7] Learning OpenCV: Computer Vision with OpenCV, Gary Bradsky & Adrian Kaehler, O'Reilly, 2008


BIBLIOGRAPHY


Appendix A

Code

A.1 Main source

```c
#include <windows.h>
```

/* Includes */
#ifndef FESTO_CAMERA
#include <highgui.h>
#endif

#include <cv.h>
#include <stdio.h>
#include <stdlib.h>
#include <cstdlib>
#include <iostream>

using namespace std;

#include "LBPD.h"
#include "posDetection.h"
#include "objDetection.h"
#include "linesDetector.h"

#ifndef LINUX
#define WIN
#endif

/*-----------------------------------------------------------
** Function : main
** Description :
** Precondition :
** Postcondition: None
** Parameters : input source: NULL if camera; path & name for a file;
** Return value : None
**---------------------------------------------------------------------*/
int main( int argc, char ** argv )
{
    printf("Duplo bricks structure disassembly\n");

    /* HSV image parameters */
    int hlower = 0;
    int hupper = 255;
```c
int hlower2;
int hupper2;
int Saturation = 100;
int Brightness = 0;
int Dilate = 15;
int Erode = 2;

bool first = true;

char imageName[30];
int contImgSaved = 0;

bool lastImage = false;
bool Pause = false;

if (argc < 2)
{
   printf("no parameters, using camera\n");
   //exit(-1);
}

/* classifier */
CvHaarClassifierCascade* classifier = (CvHaarClassifierCascade*)cvLoad(
   "(/home/alex/Escritorio/debian/modelo-duplo-real/muestra/cascade.xml",
   0, 0, 0);
#else
   "C:/Users/Alex/Desktop/debian/modelo-duplo-real/muestra/cascade.xml", 0, 0, 0);
#endif

if (classifier == NULL)
{
   printf("Haar classifier not found\n");
   exit(-1);
}

/* load of intrinsic and distortion parameters */
CvMat *intrinsicParameters = (CvMat*)cvLoad("Intrinsics.xml");
CvMat *distortionParameters = (CvMat*)cvLoad("Distortion.xml");
if (intrinsicParameters == NULL)
{
   printf("Error loading Intrinsics.xml");
   exit(-1);
}
if (distortionParameters == NULL)
{

```
printf("Error loading Distortion.xml");
exit(-1);

/* memory allocation */
CvMemStorage* storage_contours = cvCreateMemStorage(0);
CvMemStorage* objMemStorage = cvCreateMemStorage(0);

/* images */

/* input */
//CvCapture* inputSource = cvCaptureFromAVI(argv[1]);
//CvCapture* inputSource = cvCreateCameraCapture(0);
CvCapture* inputSource = 0;

//assert( inputSource );

// Find whether to detect the object from file or from camera.
if( argc < 2 )
{
    inputSource = cvCaptureFromCAM( 0 );
    cvSetCaptureProperty( inputSource, CV_CAP_PROP_FRAME_WIDTH,320 );
}
else
    inputSource = cvCaptureFromAVI( argv[1] );

/*++++++++++++++++++++++++++++++++++++++++*/
/* windows *//* uncomment to view process */
#endif

#ifndef FESTO_CAMERA
/* Dilate & Erode *//* uncomment to change values dynamically */
//cvCreateTrackbar("Erode","Parameters",&Erode,10, NULL);
//cvCreateTrackbar("Dilate","Parameters",&Dilate,20, NULL);
#endif

/* Main loop */
bool first2 = true;
while(1) {
    // stop debug point
    //printf("Start of main loop\n");
    //cvWaitKey(0);
    IplImage* frame;
    IplImage* frame2;

    /* control of multiple inputs: camera, video, or picture */
    if (!lastImage) {
        int frameCounter = 3; /* frames to jump */
        if (first2) {
            frameCounter = 0;
            first2 = false;
            frame2 = cvQueryFrame(inputSource);
        }
        while (frameCounter > 0 && frame2) {
            frame2 = cvQueryFrame(inputSource);
            --frameCounter;
        }
    }
    if (!frame2 && !lastImage) {
        lastImage = true;
    } else if (!lastImage) {
        if (!first) {
            cvReleaseImage(&frame);
        }
        frame = cvCloneImage(frame2);
        /* get undistorted image */
        UndistortImage(frame, intrinsicParameters, distortionParameters);
    }
    if (first) {
        /* creation of images with the same features of the original image */
        img_hsv = cvCreateImage(cvGetSize(frame), IPL_DEPTH_8U, 3);
        img_mono = cvCreateImage(cvGetSize(frame), IPL_DEPTH_8U, 1);
        img_rgb_seg = cvCreateImage(cvGetSize(frame), IPL_DEPTH_8U, 3);
img_cont = cvCreateImage( cvGetSize(frame), IPL_DEPTH_8U, 3 );

first = false;
}
cvCopy(frame, img_rgb_seg, 0);
// Convert image: RGB to HSV
cvCvtColor(frame, img_hsv, CV_RGB2HSV);

for (int colorNumber = 0; colorNumber < NUM_SEGMENTED_COLORS; colorNumber++)
{
    cvCopy(frame, img_rgb_seg, 0);

    switch(colorNumber)
    //switch(5)
    {
    case 0:
        hupper = 30;
        hlower = 0;
        //printf("\n\n// Blue //\n\n");
        break;

    case 1:
        hupper = 80;
        hlower = 35;
        //printf("\n\n// Green //\n\n");
        break;

    case 2:
        hupper = 105;
        hlower = 80;
        //printf("\n\n// Yellow //\n\n");
        break;

    case 3:
        hupper = 135;
        hlower = 105;
        //printf("\n\n// Red //\n\n");
        break;

    case 5:
        hupper = 255;
        hlower = 0;
        //printf("\n\n// All colors //\n\n");
        break;
CvScalar pixel;
/* image scanning */
for(int i = 0; i < img_hsv->height; i++)
{
    for(int j = 0; j < img_hsv->width; j++)
    {
        /* Segmentation of the image, using Hue angles */
        if (hupper < hlower) /* TODO: */
        {
            hupper2 = 127 - hupper;
            hlower2 = 127 - hlower;
        }
        else
        {
            hlower2 = hlower;
            hupper2 = hupper;
        }

        pixel = cvGet2D(img_hsv, i, j);

        if (((pixel.val[0]) >= hlower2) && ((pixel.val[0]) <= hupper2))
        {
            if (pixel.val[1] >= Saturation)
            {
                /* pixel = black */
                pixel.val[0] = 255;
                cvSet2D(img_mono, i, j, pixel);
            }
            else
            {
                /* pixel = white */
                pixel.val[0] = 0;
                cvSet2D(img_mono, i, j, pixel);
            }
        }
        else
        {
            /* pixel = white */
            pixel.val[0] = 0;
            cvSet2D(img_mono, i, j, pixel);
        }
    }
}
{ /* pixel = white */
    pixel.val[0] = 0;
    cvSet2D( img_mono, i,j, pixel );
} /* lastImage del for j */
} /* lastImage del for i */

/* Erode & Dilate, to delete lost pixels */
cvErode( img_mono, img_mono, 0, Erode);
cvDilate( img_mono, img_mono, 0, Dilate);

/* color segmentation */
for(int i = 0; i < img_hsv->height; i++)
    for(int j = 0; j < img_hsv->width; j++)
    {
        pixel = cvGet2D( img_mono, i, j);
        if (pixel.val[0] == 0)
        {
            pixel.val[0] = 255;
            pixel.val[1] = 255;
            pixel.val[2] = 255;
            cvSet2D( img_rgb_seg, i,j, pixel );
        }
    }

///// stop debugg point /////
//printf("After color segmentation, dilate/erode and before contours\n");
//cvWaitKey(0);
/////////////////////////

//////////////
// contours //
//////////////
cvClearMemStorage(storage_contours);
//CvMemStorage* storage_contours = cvCreateMemStorage();
CvSeq* first_contour = NULL;

int numContours = cvFindContours( img_mono, storage_contours, &first_contour,
    sizeof(CvContour),
    CV_RETR_EXTERNAL // CV_RETR_EXTERNAL CV_RETR_CCOMP CV_RETR_TREE
    CV_RETR_LIST
);
CvSeq* fc = first_contour;
cvCopyImage( frame, img_cont);
CvPoint offset = cvPoint(0,0);
int maxLevelDrawLines = 20;
cvDrawContours( img_cont, fc, CVX_RED, CVX_BLUE, maxLevelDrawLines, 1, CV_AA , offset);

int min_y, min_x, max_y, max_x;
for( CvSeq* c = first_contour; c != NULL; c = c->h_next )
{
    min_y = img_hsv->height, min_x = img_hsv->width, max_y = 0, max_x = 0;
    for( int i = 0; i < c->total; ++i )
    {
        CvPoint* p = CV_GET_SEQ_ELEM( CvPoint, c, i );
        max_x = (p->x > max_x) ? p->x : max_x;
        max_y = (p->y > max_y) ? p->y : max_y;
        min_x = (p->x < min_x) ? p->x : min_x;
        min_y = (p->y < min_y) ? p->y : min_y;
    }
    //cvCopy(img_cont, roi_Image, 0);
    //cvCopyImage(img_rgb_seg, roi_Image);
    /* drawing rectangle to segmentate */
cvRectangle( img_cont, cvPoint(min_x, min_y), cvPoint(max_x, max_y),
        CVX_BLUE, 1);
    CvPoint roiIniPoint;
    roiIniPoint.x = min_x;
    roiIniPoint.y = min_y;
    int roiWidth = max_x - min_x;
    int roiHigh = max_y - min_y;
    /* contours? */
    if (numContours > 0)
    {
        /* set of ROI */
        cvSetImageROI( img_rgb_seg, cvRect( roiIniPoint.x, roiIniPoint.y,
            roiWidth, roiHigh ) );
        /* creating a new image from the ROI */
        IplImage *img_roi = cvCreateImageHeader( cvSize( roiWidth, roiHigh ),
            img_rgb_seg->depth, img_rgb_seg->nChannels );
        img_roi->origin = img_rgb_seg->origin;
        img_roi->widthStep = img_rgb_seg->widthStep;
        img_roi->imageData = img_rgb_seg->imageData + roiIniPoint.y *
            img_rgb_seg->widthStep + roiIniPoint.x * img_rgb_seg->nChannels;
        printf ("Cutting size: w-%d, h-%d\n", img_roi->width, img_roi->height) ;
    }
#ifndef FESTO_CAMERA
    cvShowImage( "Contours image", img_cont );
#endif

cvResetImageROI(img_rgb_seg);

int numFoundPoints;
CvPoint foundImagePoints[MAX_FOUND_NUM_POINTS];
int numPyr = 0; /* to know the pyr (zoom) applied */

//// stop debug point ///</
//printf("Before contours drawing, ROI and before objDetection\n");
//cvWaitKey(0);
/////////////////////////

**** calling of find object function ****/
numFoundPoints = ObjDetection(img_roi, objMemStorage, classifier, &
    foundImagePoints[0], MAX_FOUND_NUM_POINTS, &numPyr);
/***************************************/

cvReleaseImage(img_roi);

//// stop debug point ///</
//printf("Despues de objDetection\n");
//cvWaitKey(0);
/////////////////////////

if (numFoundPoints >= 8)
{

    int brickNum, brickPoints[ numFoundPoints / BRICK_POINTS_FOUND ][
        BRICK_NUM_POINTS ];

    // stop debug point ///</
    //printf("Before call LD\n");
    //cvWaitKey(0);
    //////////////////////

    **** calling of lines detector function ****/
    brickNum = LineDetector(&foundImagePoints[0], &numFoundPoints,
        brickPoints);
    /***************************************/

    // stop debug point ///</
    //printf("After call LD\n");
    //cvWaitKey(0);
    //////////////////////

    /* adapting coordinates: The segmentation and pyr changes the
    image size */
    if (brickNum > 0)
CvPoint point0, point1, point2, point3;

for(int i = 0; i < brickNum; i++)
{
    point0 = foundImagePoints[brickPoints[i][0]];
    point1 = foundImagePoints[brickPoints[i][1]];
    point2 = foundImagePoints[brickPoints[i][2]];
    point3 = foundImagePoints[brickPoints[i][3]];

    if (numPyr)
    {
        point0.x /= numPyr;
        point0.y /= numPyr;
        point1.x /= numPyr;
        point1.y /= numPyr;
        point2.x /= numPyr;
        point2.y /= numPyr;
        point3.x /= numPyr;
        point3.y /= numPyr;
    }

    point0.x += roiIniPoint.x;
    point0.y += roiIniPoint.y;
    point1.x += roiIniPoint.x;
    point1.y += roiIniPoint.y;
    point2.x += roiIniPoint.x;
    point2.y += roiIniPoint.y;
    point3.x += roiIniPoint.x;
    point3.y += roiIniPoint.y;

    CvPoint points[] = {point0, point1, point2, point3};
    CvPoint* curveArr[1]={points};
    int nCurvePts[1]={4};
    int nCurves=1;
    int isCurveClosed=1;
    int lineWidth=2;

    /* drawing found brick to debug */
    cvPolyLine(img_cont, curveArr, nCurvePts, nCurves,
                isCurveClosed, cvScalar(255,0,255), lineWidth);
    cvLine( img_cont, point0, point1, cvScalar(255,0,0), 1,
            CV_AA);
cvCircle(img_cont, point0, 3, cvScalar(255,255,255), -1);

#ifndef FESTO_CAMA
    cvShowImage( "Contours image", img_cont );
#endif

// stop debug point ///
//printf("Before call posDetection\n");
//cvWaitKey(0);

posDetection ( frame, &points[0], intrinsicParameters, distortionParameters );

// stop debug point ///
//printf("After call posDetector\n");
//cvWaitKey(0);

/* Pause Control */
if (Pause)
{
    char p = cvWaitKey(0);
    if( p == 112 )
    {
        Pause = false;
        printf("running\n");
    }
}

char c = cvWaitKey(40);
if( c == 27 ) exit(0); // press ESC to exit
if( c == 112 && !Pause)
{
    Pause = true;
    printf("pause\n");
}
else if( c == 112 && Pause )
{
    Pause = false;
    printf("running\n");
APPENDIX A. CODE

A.2. OBJECT DETECTION

if (c == 115) /* S pressed, save image */
{
    sprintf (imageName, "img_%d.bmp", contImgSaved);
    #ifndef FESTO_CAMERA
        cvSaveImage(imageName, frame);
    #endif
    ++contImgSaved;
}

/* release memory */
cvReleaseMemStorage( &storage_contours );
cvReleaseMemStorage( &objMemStorage );
#endif

cvReleaseImage(&frame);
cvReleaseImage(&img_hsv);
#endif

cvReleaseImage(&img_mono);
cvReleaseImage(&img_rgb_seg);
cvReleaseImage(&img_cont);
#endif

cvReleaseCapture(&inputSource );
#endif

cvDestroyWindow( "Source image" );
#endif

cvDestroyWindow( "HSV image" );
#endif

cvDestroyWindow( "Monochromatic image" );
#endif

cvDestroyWindow( "Segmented image" );
#endif

cvDestroyWindow( "Contours image" );
#endif

cvDestroyWindow( "ROI image" );
#endif

cvDestroyWindow( "Parameters" );
#endif

cvDestroyWindow( "Object position" );
}
#endif

cvDestroyWindow( "Parameters" );
exit(0);
}

A.2 Object detection
A.2. OBJECT DETECTION

APPENDIX A. CODE

/*-------------------------------------------------------------
** Project Name:
** $Archive: / $
** $Author: Alex Moser $
** $Date: 14/06/11 $
** $Revision: 1 $
**-------------------------------------------------------------
** Target system:
** Compiler:
**-------------------------------------------------------------
** A U T H O R  I D E N T I T Y
**-------------------------------------------------------------
** Initials Name Company
** -------------- --------------------- ----------------------
** AMOSER Alex Moser TUW
**-------------------------------------------------------------
** R E V I S I O N  H I S T O R Y
**-------------------------------------------------------------
** $Log: / $
** User: Amoser Date: 14/06/11 Time: 16:07
** Comment:
** File Creation
**-------------------------------------------------------------
/*-------------------------------------------------------------
** Includes */
#include <highgui.h>
#include <cv.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include "objDetection.h"
/* Defines */
#define MIN_VALID_OBJECTS (int)8
#define MIN_WIDTH_PYR (int)100
#define MIN_HEIGHT_PYR (int)100
#define SCALE_FACTOR (double)1.1
#define MIN_NEIGHBORS (int)2
/*-------------------------------------------------------------
** Function : objDetection
** Description :
** Precondition :
** Postcondition: None
** Parameters : None
** Return value : None

```c
int ObjDetection( IplImage* img_obj, CvMemStorage* storage,
        CvHaarClassifierCascade* classifier, CvPoint foundImagePoints[], int
          maxNumFindPoints, int* numPyr )
{
    #ifndef FESTO_CAMERA
    int radiusPoint;
    #endif

    CvPoint objCentralPoint;
    /* "Resets" the memory but does not deallocate it */
    cvClearMemStorage(storage);
    #ifndef FESTO_CAMERA
    //cvNamedWindow( "ObjFinder image", 0 );
    IplImage* img_ref;
    #endif

    IplImage* img_cascade;
    IplImage* img_gray = cvCreateImage( cvSize(img_obj->width, img_obj->height),
          IPL_DEPTH_8U, 1 );
    cvCvtColor( img_obj, img_gray, CV_BGR2GRAY );
    //if (1)
    if (img_obj->width < MIN_WIDTH_PYR || img_obj->width < MIN_HIGH_PYR)
    {   
        IplImage* img_cascade2 = doPyrUp( img_gray );
        img_cascade = doPyrUp( img_cascade2 );
    #ifndef FESTO_CAMERA
    IplImage* img_ref2 = doPyrUp( img_obj );
    img_ref = doPyrUp( img_ref2 );
    cvReleaseImage( &img_ref2 );
    #endif

    img_cascade2 = doPyrUp( img_gray );
    img_cascade = doPyrUp( img_cascade2 );
    #ifndef FESTO_CAMERA
    IplImage* img_ref2 = doPyrUp( img_obj );
    img_ref = doPyrUp( img_ref2 );
    cvReleaseImage( &img_ref2 );
    #endif
    
    cvReleaseImage( &img_cascade2 );
    *numPyr = 4;
    }
    else
    {
        img_cascade = cvCreateImage( cvGetSize( img_gray ),img_gray->depth,
          img_gray->nChannels );
    #ifndef FESTO_CAMERA
```

---

89
A.2. OBJECT DETECTION

```c
img_ref = cvCreateImage( cvGetSize( img_obj ), img_obj->depth, img_obj->nChannels );
cvCopy( img_obj, img_ref, NULL );
#endif

cvCopy( img_gray, img_cascade, NULL );
}

/* object detection */
CvSeq* objects = cvHaarDetectObjects( img_cascade, classifier, storage, SCALE_FACTOR,
MIN_NEIGHBORS, CV_HAAR_DO_CANNY_PRUNING, cvSize(24, 24) );
printf("%d objects found\n", objects->total);
/* If any faces were detected, draw rectangles around them */
if (objects)
{
    for(int i = 0; i < objects->total; ++i)
    {
        /* Setup two points that define the extremes of the rectangle, */
        /* then draw it to the image */
        CvPoint point1, point2;
        CvRect* rectangle = (CvRect*)cvGetSeqElem(objects, i);
        point1.x = rectangle->x;  
        point2.x = (rectangle->x + rectangle->width);
        point1.y = rectangle->y;
        point2.y = (rectangle->y + rectangle->height);
        #ifndef FESTO_CAMERA
        cvRectangle(img_ref, point1, point2, CV_RGB(255,0,0), 3,
                    CV_AA, 0);
        #endif
        objCentralPoint.x = (point1.x > point2.x) ? ((point1.x - point2.x)/2) + point1.x;
        objCentralPoint.y = (point1.y > point2.y) ? ((point1.y - point2.y)/2) + point1.y;
        #ifndef FESTO_CAMERA
        radiusPoint = rectangle->height / RADIUS_POINT_DIV_FACTOR;
        cvCircle( img_ref, objCentralPoint, radiusPoint, cvScalar
                    (255,255,255), -1 );
        #endif

        if (i < maxNumFindPoints)
        {
            foundImagePoints[i] = objCentralPoint;
        }
    }
}
```
if (objects->total >= MIN_VALID_OBJECTS)
{
    for(int i = 0; i < objects->total; ++i)
    {
        printf("x: %d - y: %d\n", foundImagePoints[i].x, foundImagePoints[i].y);
    }
#endif
    //cvShowImage( "ObjFinder image", img_ref);
    //cvWaitKey(500);
#endif
    cvReleaseImage( &img_cascade);
    cvReleaseImage( &img_gray);
#endif FESTO_CAMERA
    //cvDestroyWindow( "ObjFinder image" );
    cvReleaseImage( &img_ref);
#endif

    // stop debug point ////
    //printf("Befor end of objDetection\n");
    //cvWaitKey(0);
    ///////////////////////////////////////////////////
    return(objects->total);
    //cvWaitKey(0);
}
A.3 Lines detector

```cpp
    return( out );
```

A.3. LINES DETECTOR

---

/* Project Name:
** $Archive: / $
** $Author: Alex Moser $
** $Date: 14/06/11 $
** $Revision: 1 $
** Target system:
** Compiler:
**
** AUTHOR IDENTITY
**
** Initials    Name       Company
**  -------    ---------   ----------
** AMOSER      Alex Moser   TUW
**
** REVISION HISTORY
**
** $Log: / $
** User: Amoser Date: 14/06/11 Time: 16:07
** Comment:
** File Creation
**
**
```
/* Includes */

#ifndef FESTO_CAMERA
#include "highgui.h"

#endif

#include <cv.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
```
```c
#include "linesDetector.h"

/* Defines */
#define MAX_DIST_FACTOR (float)1.5
#define MAX_DIST_POINT_LINE_FACTOR (float)3.5
#define POINTS_PER_LINE (int)4
#define STANDART_SLOPE (int)0
#define POINT_HYSTERESIS (int)10

/*-------------------------------------------------------------
** Function : LineDetector
** Description :
** Precondition :
** Postcondition: None
** Parameters : None
** Return value : None
**----------------------------------------------------------*/

int LineDetector ( CvPoint image_points[], int* total_points, int brickPoints[][], int points_avrDist)
{
    int total_points2 = *total_points;
    
    bool meanOK = true;
    int points_avrDist;
    int distBetwPoints[total_points2][total_points2];
    do
    {
        int distAcum = 0;
        int dists[total_points2];
        meanOK = true;
        for (int j = 0; j < total_points2; j++)
            dists[j] = 0xFFFF;
        for (int j = 0; j < total_points2; j++)
            {
                CvPoint punt_ref = image_points[j];
                for (int i = 0; i < total_points2; i++)
                    {  
```
long dx, dy;
int dist;
if (j != i)
{
    dx = abs(punt_ref.x - image_points[i].x);
    dy = abs(punt_ref.y - image_points[i].y);
    if (dx == 0)
    {
        dist = dy;
    }
    else if (dy == 0)
    {
        dist = dx;
    }
    else
    {
        dist = (int)(sqrt((dx * dx) + (dy * dy)));
    }
    printf("j:%d i:%d dist:%d\n", j, i, dist);
    distBetwPoints[j][i] = dist;
    //distAcum += dist;
}
for (int j = 0; j < total_points2; j++)
{
    distAcum += dists[j];
}
points_avrDist = distAcum / total_points2;

/* check for points too far */
for (int j = 0; j < total_points2; j++)
{
    if (dists[j] > (points_avrDist * MAX_DIST_FACTOR) /*|| dists[j] < (points_avrDist / 1.5)*/)
    {
        meanOK = false;
        --total_points2;
        *total_points = total_points2;
        for (int jj = j; jj < total_points2; jj++)
        {

    ...

94
image_points[jj] = image_points[jj + 1];
}
}

printf("\nAverage distance between points: %d \n", points_avrDist);

for (int j = 0; j < total_points2; j++)
{
    printf("Dist %d: %d\n", j, dists[j]);
}

while (!meanOK);


////////////////////////////////////////////////////
// manual lines detector //
////////////////////////////////////////////////////

IplImage* img_color2 = cvCreateImage( cvGetSize(img_puntos), 8, 3 );
IplImage* img_color2 = cvCreateImage( cvSize(1000,1000), IPL_DEPTH_8U, 3 );

//cvCvtColor( img_puntos, img_color2, CV_GRAY2BGR );
//cvNamedWindow( "Detector lineas manual", 0 );

int linePoints[ total_points2 / POINTS_PER_LINE ][ POINTS_PER_LINE ];
int pointsOK[total_points2];
int pointsOKcount = 0;
int numLines = 0;
int numBricks = 0;
bool usedPoint = false;

/* point 1 of the line */
for (int LP1 = 0; LP1 < total_points2; LP1++)
{
    int countTemp;
    usedPoint = false;
    CvPoint pointRef = image_points[LP1];
    printf("Line point 1: %d\n", LP1);

    /* avoid repeating validated points */
    for (int ii = 0; ii < pointsOKcount; ii++)
    {
        if (pointsOK[ii] == LP1)
A.3. LINES DETECTOR

APPENDIX A. CODE

{)
    usedPoint = true;
    printf("Point 1 already used: %d\n", pointsOK[ii]);
}

if (!usedPoint)
{
    /* Point 2 of the line */
    for (int LP2 = LP1 + 1; LP2 < total_points2; LP2++)
    {
        usedPoint = false;
        int contPuntosOld = pointsOKcount;
        /* avoid repeating validated points */
        for (int iii = 0; iii < pointsOKcount; iii++)
        {
            if (pointsOK[iii] == LP2 )
            {
                usedPoint = true;
                printf("Point 2 already used: %d\n", pointsOK[iii]);
            }
        }
        if (usedPoint)
            continue;

        if (distBetwPoints[LP2][LP1] > (points_avrDist * MAX_DIST_POINT_LINE_FACTOR))
        {
            printf("Point %d too far, discarded\n", LP2);
            continue;
        }

        CvPoint pointRef2 = image_points[LP2];
        printf("Line point 2: %d\n", LP2);

        float dx, dy;
        float slope, lineOffset;
        int slopeType = 0;

        /* calculation of the line equation */
        dx = pointRef2.x - pointRef.x;
        dy = pointRef2.y - pointRef.y;

        /* avoiding infinite and 0 slope */
        if (dy != 0 && dx != 0)
        {
            slope = dy / dx;
            lineOffset = pointRef2.y - (pointRef2.x + slope);
            slopeType = 0;
            printf("Slope of the line: %f\n", slope );
        }
APPENDIX A. CODE

A.3. LINES DETECTOR

```c
}  
else if (dy == 0)  
{  
    printf("Slope not valid\n");  
slopeType = 1;  
}  
else if (dx == 0)  
{  
    printf("Slope not valid\n");  
slopeType = 2;  
}  

int coincidenceCounter = 0;  
int pointsPosition[10];  

if (slopeType == STANDART_SLOPE)  
{  
    /* search for points belonging to the line */  
    for (int jj = 0; jj < total_points2; jj++)  
    {  
        usedPoint = false;  
        CvPoint testPoint = image_points[jj];  
        
        if (LP2 != jj && LP1 != jj)  
        {  
            for (int iiii = 0; iiii < pointsOKcount; iiii++)  
            {  
                if (pointsOK[iiii] == jj)  
                {  
                    usedPoint = true;  
                    printf("Point 3 already used: %d\n", jj);  
                }  
            }  
        }  

        if (!usedPoint)  
        {  
            float result;  
            printf("Belogs point %d to the line?\n", jj);  
            
            if (slope >= -1 && slope <= 1) /* Y axis closest */  
            {  
                result = slope * testPoint.x + lineOffset;  
                
                if ((result >= testPoint.y - POINT_HYSTERESIS) && (result <= testPoint.y + POINT_HYSTERESIS))  
                {  
                    pointsPosition[coincidenceCounter] = jj;  
                    coincidenceCounter++;  
                }  
```
A.3. LINES DETECTOR

APPENDIX A. CODE

```c
284 }
285 }
286 /* X axis closest */
287 {
288     result = (testPoint.y - lineOffset) / slope;
289 
290     if ((result >= testPoint.x - POINT_HYSTERESIS) &&
291         (result <= testPoint.x + POINT_HYSTERESIS))
292         pointsPosition[coincidenceCounter] = jj;
293         coincidenceCounter++;
294 }
295 
296 }
297 }
298 /* If we have 2 coincident points, it means we have a line */
299 if (coincidenceCounter == 2)
300 {
301     printf("2 points found in the line :) \n");
302     countTemp++;
303 
304     pointsOK[pointsOKcount] = LP1;
305     pointsOK[pointsOKcount + 1] = LP2;
306     pointsOK[pointsOKcount + 2] = pointsPosition[0];
307     pointsOK[pointsOKcount + 3] = pointsPosition[1];
308 
309     /* arranging points */
310     int aux;
311     bool flagDistOk = false;
312 
313     for ( int k = pointsOKcount; k < (pointsOKcount + POINTS_PER_LINE); k++ )
314     {
315         for ( int kk = pointsOKcount; kk < (pointsOKcount + POINTS_PER_LINE); kk++ )
316         {
317             if ( abs( image_points[ LP1 ].x - image_points[ LP2 ].x ) > abs( image_points[ LP1 ].y -
318                     image_points[ LP2 ].y ) )
319             {
320                 if ( image_points[ pointsOK[ kk ] ].x >
321                     image_points[ pointsOK[ k ] ].x )
322                 {
323                     aux = pointsOK[ k ];
324                     pointsOK[ k ] = pointsOK[ kk ];
325                     pointsOK[ kk ] = aux;
326                 }
327             }
328         }
329     }
330 }
98
```
A.3. LINES DETECTOR

APPENDIX A. CODE

```c
326 }
327 }
else
329 {
    if (image_points[ pointsOK[ kk ] ].y >
        image_points[ pointsOK[ k ] ].y)
    { aux = pointsOK[ k ];
      pointsOK[ k ] = pointsOK[ kk ];
      pointsOK[ kk ] = aux;
    }
}

/* looking distance between points */

int aux1, aux2, aux3, aux4, dist1, dist2, dist3;

aux1 = pointsOK[pointsOKcount];
aux2 = pointsOK[pointsOKcount + 1];
aux3 = pointsOK[pointsOKcount + 2];
aux4 = pointsOK[pointsOKcount + 3];

dist1 = distBetwPoints[aux1][aux2];
dist2 = distBetwPoints[aux2][aux3];
dist3 = distBetwPoints[aux3][aux4];

if (dist1 < (dist2 + (dist2 * 0.2)) && dist1 > (dist2 - (dist2 * 0.2)))
    if (dist2 < (dist3 + (dist3 * 0.2)) && dist2 > (dist3 - (dist3 * 0.2)))
        flagDistOk = true; /* line validated */

if (flagDistOk)
{
    /* drawing the points of the line */
    int colorPointsCounter = 0;

    for (int k = pointsOKcount; k < pointsOKcount + POINTS_PER_LINE; k++)
    {
        linePoints[numLines][k - pointsOKcount] = pointsOK[ k ];

        CvPoint circle = image_points[pointsOK[k]];
        /* uncomment to draw */
        // if (colorPointsCounter == 0)
        // cvCircle(img_color2, circle, 15, cvScalar(0,255,0), 2);
```
// else if (colorPointsCounter == 3)
// cvCircle( img_color2, circle, 15, cvScalar
// (0,0,255), 2 );

colorPointsCounter++;

printf("Puntos validados: %d\n", pointsOK[k]);
//cvShowImage( "Detector lineas manual", img_color2
// );
//cvWaitKey(0);

numLines++;
pointsOKcount += 4;
}

if (contPuntosOld != pointsOKcount) /* it means a line has been found
, is not necessary continue searching with these two points */
break;

/* draw lines
for (int k = 0; k < numLines ; k++)
{
CvPoint pt1, pt2;
pt1 = image_points[linePoints[k][0]];
pt2 = image_points[linePoints[k][3]];
cvLine( img_color2, pt1, pt2, cvScalar(255,0,0), 2, CV_AA);
tempColorLineas++;
}
*/
printf("Number of lines found: %d\n", numLines);

/* looking for find a rectangle */

if (numLines >= 2)
{
    for (int LineRef1 = 0; LineRef1 < numLines; LineRef1++)
    {
        int point1Line1 = linePoints[ LineRef1 ][ 0 ];
        int point2Line1 = linePoints[ LineRef1 ][ 3 ];
        
    }
for (int LineRef2 = LineRef1 + 1; LineRef2 < numLines; LineRef2++)
{
    int point1Line2 = linePoints[ LineRef2 ][ 0 ];
    int point2Line2 = linePoints[ LineRef2 ][ 3 ];

    if ( distBetwPoints[ point1Line1 ][ point1Line2 ] < ( points_avrDist * 1.8 ) &&
         distBetwPoints[ point1Line1 ][ point1Line2 ] > ( points_avrDist * 0.7 ) )
        if ( distBetwPoints[ point2Line1 ][ point2Line2 ] < ( points_avrDist * 1.8 ) &&
             distBetwPoints[ point2Line1 ][ point2Line2 ] > ( points_avrDist * 0.7 ) )
            /* default */
            brickPoints[ numBricks ][ 0 ] = point1Line1;
            brickPoints[ numBricks ][ 1 ] = point1Line2;
            brickPoints[ numBricks ][ 2 ] = point2Line2;
            brickPoints[ numBricks ][ 3 ] = point2Line1;

            /* arranging points */
            if (image_points[ point1Line1 ].x < image_points[ point2Line1 ].x &&
                 image_points[ point1Line2 ].x < image_points[ point2Line2 ].x )
            {
                if (image_points[point1Line1].y <
                    image_points[point1Line2].y)
                {
                    brickPoints[ numBricks ][ 0 ] =
                        point1Line2;
                    brickPoints[ numBricks ][ 1 ] =
                        point1Line1;
                    brickPoints[ numBricks ][ 2 ] =
                        point2Line1;
                    brickPoints[ numBricks ][ 3 ] =
                        point2Line2;
                }
            }
            else if (image_points[ point1Line1 ].x >
                     image_points[ point2Line1 ].x &&
                     image_points[ point1Line2 ].x > image_points[ point2Line2 ].x )
            {
                if (image_points[ point1Line1 ].y >
                    image_points[ point1Line2 ].y)
                {
                    brickPoints[ numBricks ][ 0 ] =
                        point1Line2;
                    brickPoints[ numBricks ][ 1 ] =
                        point1Line1;
                }
A.3. LINES DETECTOR

APPENDIX A. CODE

```c
brickPoints[numBricks][2] =
  point2Line1;
brickPoints[numBricks][3] =
  point2Line2;

else if (image_points[point1Line1].y <
  image_points[point2Line1].y && image_points[
  point1Line2].y < image_points[
  point2Line2].y)
{
  if (image_points[point1Line1].x >
      image_points[point1Line2].x)
  {
    brickPoints[numBricks][0] =
      point1Line2;
    brickPoints[numBricks][1] =
      point1Line1;
    brickPoints[numBricks][2] =
      point2Line1;
    brickPoints[numBricks][3] =
      point2Line2;
  }
  else if (image_points[point1Line1].y >
    image_points[point2Line1].y && image_points[
    point1Line2].y > image_points[
    point2Line2].y)
  {
    if (image_points[point1Line1].x <
        image_points[point1Line2].x)
    {
      brickPoints[numBricks][0] =
        point1Line2;
      brickPoints[numBricks][1] =
        point1Line1;
      brickPoints[numBricks][2] =
        point2Line1;
      brickPoints[numBricks][3] =
        point2Line2;
    }
  }
}

numBricks++;

/* puntos_imagen */
/*
  CvPoint curve1[] = {image_points[point1Line1],
    image_points[point1Line2], image_points[
```
APPENDIX A. CODE

A.3. LINES DETECTOR

```c
point2Line1], image_points[point2Line2]);
CvPoint* curveArr[1] = {curve1};
int nCurvePts[1] = {4};
int nCurves = 1;
int isCurveClosed = 1;
int lineWidth = 2;

cvPolyLine(img_color2, curveArr, nCurvePts, nCurves,
isCurveClosed, cvScalar(255, 0, 255), lineWidth);
*/
```

```c
for (int jj = 0; jj < total_points2; jj++)
    CvPoint testPoint = image_points[jj];
    printf("point %d in x = %d y = %d \n", jj, testPoint.x, testPoint.y);
}

printf("%d bricks found\n", numBricks);
```

```c
return (numBricks);
```
A.4 Position calculation

```cpp
#include <highgui.h>
#include <cv.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <iostream>
#include "posDetection.h"

using namespace std;

/*----------------------------------------------------------------------------
** Function : posDetection
** Description : 
** Precondition : 
** Postcondition: None
** Parameters  : None
*/
```
** Return value: None

```c
int posDetection ( IplImage* image_pos, CvPoint imageBrickPoints[], CvMat* intrinsicParameters, CvMat* distortionParameters )
{
    IplImage *image_pos2 = cvCloneImage(image_pos);

    #ifndef FESTO_CAMERA
    // cvNamedWindow( "Object position", 0 );
    #endif

    //cvNamedWindow( "Calibration", 0 );
    //cvShowImage( "Calibration", image_pos2 ); // Show raw image
    //cvRemap( t, image, mapx, mapy ); // Undistort image
    //cvReleaseImage(&t);

    //extrinsics
    //cvNamedWindow( "Calibration" );
    // cvShowImage( "Calibration", image );
    // int close = cvWaitKey(0);

    CvMat* object_points3 = cvCreateMat(4,3,CV_32FC1);
    CvMat* image_points3 = cvCreateMat(4,2,CV_32FC1);
    CvMat* point_counts3 = cvCreateMat(1,1,CV_32SC1);

    // definicion agujeros superiores ladrillo //
    CvMat* object_points4 = cvCreateMat(5,3,CV_32FC1);
    CV_MAT_ELEM(*object_points4, float, 0, 0) = 1.0f;
    CV_MAT_ELEM(*object_points4, float, 0, 1) = 1.0f;
    CV_MAT_ELEM(*object_points4, float, 0, 2) = 0.0f;
    CV_MAT_ELEM(*object_points4, float, 1, 0) = 1.0f;
    CV_MAT_ELEM(*object_points4, float, 1, 1) = 1 + BRICK_DIST_B;
    CV_MAT_ELEM(*object_points4, float, 1, 2) = 0.0f;
    CV_MAT_ELEM(*object_points4, float, 2, 0) = 1 + BRICK_DIST_A;
    CV_MAT_ELEM(*object_points4, float, 2, 1) = 1 + BRICK_DIST_B;
    CV_MAT_ELEM(*object_points4, float, 2, 2) = 0.0f;
    CV_MAT_ELEM(*object_points4, float, 3, 0) = 1 + BRICK_DIST_A;
    CV_MAT_ELEM(*object_points4, float, 3, 1) = 1.0f;
    CV_MAT_ELEM(*object_points4, float, 3, 2) = 0.0f;
```
A.4. POSITION CALCULATION

APPENDIX A. CODE

CV_MAT_ELEM(*object_points4, float, 4, 0) = 1 + (BRICK_DIST_A / 2);
CV_MAT_ELEM(*object_points4, float, 4, 1) = 1 + (BRICK_DIST_B / 2);
CV_MAT_ELEM(*object_points4, float, 4, 2) = -0.4f;

// definicion extremos bloque inf//
CvMat* object_points5 = cvCreateMat(8, 3, CV_32FC1);

CV_MAT_ELEM(*object_points5, float, 0, 0) = 0.0f;
CV_MAT_ELEM(*object_points5, float, 0, 1) = 0.0f;
CV_MAT_ELEM(*object_points5, float, 0, 2) = 0.0f;

CV_MAT_ELEM(*object_points5, float, 1, 0) = 0.0f;
CV_MAT_ELEM(*object_points5, float, 1, 1) = BRICK_DIST_D;
CV_MAT_ELEM(*object_points5, float, 1, 2) = 0.0f;

CV_MAT_ELEM(*object_points5, float, 2, 0) = BRICK_DIST_C;
CV_MAT_ELEM(*object_points5, float, 2, 1) = BRICK_DIST_D;
CV_MAT_ELEM(*object_points5, float, 2, 2) = 0.0f;

CV_MAT_ELEM(*object_points5, float, 3, 0) = BRICK_DIST_C;
CV_MAT_ELEM(*object_points5, float, 3, 1) = 0.0f;
CV_MAT_ELEM(*object_points5, float, 3, 2) = 0.0f;

// definicion extremos bloque sup//
CV_MAT_ELEM(*object_points5, float, 4, 0) = 0.0f;
CV_MAT_ELEM(*object_points5, float, 4, 1) = 0.0f;
CV_MAT_ELEM(*object_points5, float, 4, 2) = -BRICK_DIST_E;

CV_MAT_ELEM(*object_points5, float, 5, 0) = 0.0f;
CV_MAT_ELEM(*object_points5, float, 5, 1) = BRICK_DIST_D;
CV_MAT_ELEM(*object_points5, float, 5, 2) = -BRICK_DIST_E;

CV_MAT_ELEM(*object_points5, float, 6, 0) = BRICK_DIST_C;
CV_MAT_ELEM(*object_points5, float, 6, 1) = BRICK_DIST_D;
CV_MAT_ELEM(*object_points5, float, 6, 2) = -BRICK_DIST_E;

CV_MAT_ELEM(*object_points5, float, 7, 0) = BRICK_DIST_C;
CV_MAT_ELEM(*object_points5, float, 7, 1) = 0.0f;
CV_MAT_ELEM(*object_points5, float, 7, 2) = -BRICK_DIST_E;

// definicion agujerocentral sup //
CvMat* object_points6 = cvCreateMat(1, 3, CV_32FC1);

CV_MAT_ELEM(*object_points6, float, 0, 0) = (1 + BRICK_DIST_A) / 2;
CV_MAT_ELEM(*object_points6, float, 0, 1) = (1 + BRICK_DIST_B) / 2;
CV_MAT_ELEM(*object_points6, float, 0, 2) = 0.0f;

// test ejes //
CvMat* object_points7 = cvCreateMat(4,3,CV_32FC1);

CV_MAT_ELEM(*object_points7, float,0,0) = 0;
CV_MAT_ELEM(*object_points7, float,0,1) = 0;
CV_MAT_ELEM(*object_points7, float,0,2) = 0;
CV_MAT_ELEM(*object_points7, float,1,0) = 0;
CV_MAT_ELEM(*object_points7, float,1,1) = 10;
CV_MAT_ELEM(*object_points7, float,1,2) = 0;
CV_MAT_ELEM(*object_points7, float,2,0) = 10;
CV_MAT_ELEM(*object_points7, float,2,1) = 0;
CV_MAT_ELEM(*object_points7, float,2,2) = 0;
CV_MAT_ELEM(*object_points7, float,3,0) = 0;
CV_MAT_ELEM(*object_points7, float,3,1) = 0;
CV_MAT_ELEM(*object_points7, float,3,2) = 10;

//assignacion puntos imagen
for( int i = 0; i < 4; ++i )
{
    CV_MAT_ELEM(*image_points3, float,i,0) = imageBrickPoints[i].x;
    CV_MAT_ELEM(*image_points3, float,i,1) = imageBrickPoints[i].y;
    CV_MAT_ELEM(*object_points3, float,i,0) = CV_MAT_ELEM(*object_points4, float,i,0);
    CV_MAT_ELEM(*object_points3, float,i,1) = CV_MAT_ELEM(*object_points4, float,i,1);
    CV_MAT_ELEM(*object_points3, float,i,2) = 0.0f;
}
CV_MAT_ELEM(*point_counts3, int,0,0) = 4;

CvMat* rotation_matrix = cvCreateMat(1,3,CV_32FC1);
CvMat* translation_vector = cvCreateMat(1,3,CV_32FC1);

// obtencion parametos extrinsecos
cvFindExtrinsicCameraParams2(object_points3, image_points3,
intrinsicParameters, distortionParameters, rotation_matrix,
translation_vector);
// cvFindExtrinsicCameraParams2(object_points3, image_points3, intrinsic,
distortion, rotation_matrix, translation_vector);

//cvSave("RotationExtrinsic.xml",rotation_matrix);
//cvSave("TranslationExtrinsic.xml",translation_vector);

//CvMat *rotation_matrix2 = (CvMat*)cvLoad("RotationExtrinsic.xml");
//CvMat *translation_vector2 = (CvMat*)cvLoad("TranslationExtrinsic.xml");

107
cout << "ESTIMATED ROTATION\n";
for ( int p = 0; p < 3; p++ )
{
    float temp = CV_MAT_ELEM(*rotation_matrix, float, 0, p);
    printf ("- %f -", temp);
}
cout << "\n";

cout << "ESTIMATED TRANSLATION\n";
for ( int p = 0; p < 3; p++ )
{
    float temp = CV_MAT_ELEM(*translation_vector, float, 0, p);
    printf ("- %f -", temp);
}
cout << "\n";

// calculo de punto proyectado agujero
CvMat* image_projectPoints = cvCreateMat(5,2,CV_32FC1);
cvProjectPoints2( object_points4, rotation_matrix, translation_vector,
intrinsicParameters, distortionParameters,
image_projectPoints,
NULL,
NULL,
NULL,
NULL,
NULL,
0
);
for (int i = 0; i < 4; i++)
{
    CvPoint littleCircle;
    littleCircle.x = (int)CV_MAT_ELEM(*image_projectPoints, float,i,0);
    littleCircle.y = (int)CV_MAT_ELEM(*image_projectPoints, float,i,1);
    if (i == 0)
    {
        cvCircle( image_pos2, littleCircle, 4, CV_RGB(255,255,255 ), -1 );
        /*
         * printf(\n"pixels position of the first defined point of the object\n")
         */
    } else if (i == 1)
        cvCircle( image_pos2, littleCircle, 4, CV_RGB(50,100,250 ), -1 );
else
    cvCircle(image_pos2, littleCircle, 4, CV_RGB(0,255,255), -1);
}

CvPoint centralLittleCircle;

centralLittleCircle.x = (int)CV_MAT_ELEM(*image_projectPoints, float,4,0);
centralLittleCircle.y = (int)CV_MAT_ELEM(*image_projectPoints, float,4,1);
cvCircle(image_pos2, centralLittleCircle, 4, CV_RGB(255, 0, 0), -1);
*/
printf("\n pixel position of the central point\n");
printf("="========\n");
printf("| X: %d\n", centralLittleCircle.x);
printf("| Y: %d\n", centralLittleCircle.y);
*/ // proyeccion puntos ladrillo 3D
CvMat* image_projectPoints2 = cvCreateMat(8,2,CV_32FC1);

cvProjectPoints2(object_points5, rotation_matrix, translation_vector, intrinsicParameters, distortionParameters,
image_projectPoints2, NULL, NULL, NULL, NULL, NULL, 0, 0);

CvPoint puntosPolyLine[8];
for (int i = 0; i < 8; i++)
{
    CvPoint littleCircle;

    littleCircle.x = (int)CV_MAT_ELEM(*image_projectPoints2, float,i,0);
    littleCircle.y = (int)CV_MAT_ELEM(*image_projectPoints2, float,i,1);
    puntosPolyLine[i] = littleCircle;

    if (i == 0)
        cvCircle(image_pos2, littleCircle, 4, CV_RGB(255, 255, 255), -1);
    else if (i < 4)
        cvCircle(image_pos2, littleCircle, 4, CV_RGB(255, 255, 0), -1);
    else
        cvCircle(image_pos2, littleCircle, 4, CV_RGB(50, 50, 0), -1);
}

CvPoint curve1[] = {puntosPolyLine[0], puntosPolyLine[1], puntosPolyLine[2],
puntosPolyLine[3]};
A.4. POSITION CALCULATION

APPENDIX A. CODE

```c

CvPoint curve2[] = {puntosPolyLine[4], puntosPolyLine[5], puntosPolyLine[6],
                    puntosPolyLine[7]};
CvPoint curve3[] = {puntosPolyLine[4], puntosPolyLine[5], puntosPolyLine[0],
                    puntosPolyLine[1]};
CvPoint curve4[] = {puntosPolyLine[2], puntosPolyLine[3], puntosPolyLine[7],
                    puntosPolyLine[6]};

CvPoint* curveArr[4] = {curve1, curve2, curve3, curve4};

int nCurves = 4;
int isCurveClosed = 1;
int lineWidth = 1;

cvPolyLine(image_pos2, curveArr, nCurvePts, nCurves, isCurveClosed, cvScalar(50, 50, 50), lineWidth);

/*------------------------------------------------------------* /
// proyeccion punto central objeto
CvMat* image_projectPoints3 = cvCreateMat(1, 2, CV_32FC1);

cvProjectPoints2( object_points6, rotation_matrix, translation_vector,
                 intrinsic, distortion,
                 image_projectPoints3,
                 NULL,
                 NULL,
                 NULL,
                 NULL,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
                 0,
```
APPENDIX A. CODE

A.4. POSITION CALCULATION

```
CV_MAT_ELEM(*rotMatRod, float, 0, 1) * CV_MAT_ELEM(*object_points6, float, 0, 1) +
CV_MAT_ELEM(*rotMatRod, float, 0, 2) * CV_MAT_ELEM(*object_points6, float, 0, 2) +
CV_MAT_ELEM(*translation_vector, float, 0, 0);

ObjectDistanceY =
CV_MAT_ELEM(*rotMatRod, float, 1, 0) * CV_MAT_ELEM(*object_points6, float, 0, 0) +
CV_MAT_ELEM(*rotMatRod, float, 1, 1) * CV_MAT_ELEM(*object_points6, float, 0, 1) +
CV_MAT_ELEM(*rotMatRod, float, 1, 2) * CV_MAT_ELEM(*object_points6, float, 0, 2) +
CV_MAT_ELEM(*translation_vector, float, 0, 1);

ObjectDistanceZ =
CV_MAT_ELEM(*rotMatRod, float, 2, 0) * CV_MAT_ELEM(*object_points6, float, 0, 0) +
CV_MAT_ELEM(*rotMatRod, float, 2, 1) * CV_MAT_ELEM(*object_points6, float, 0, 1) +
CV_MAT_ELEM(*rotMatRod, float, 2, 2) * CV_MAT_ELEM(*object_points6, float, 0, 2) +
CV_MAT_ELEM(*translation_vector, float, 0, 2);

printf("\nDistances between the central point of the camera and the central point of the object:\n");
printf("-------------------------------\n");
printf("| X: %f\n", ObjectDistanceX);
printf("| Y: %f\n", ObjectDistanceY);
printf("| Z: %f\n", ObjectDistanceZ);

//cvCircle( image_pos2, cvPoint((int)(image_pos2->width * 0.5), (int)(image_pos2->height * 0.5)), 5, CV_RGB(50, 50, 0 ), 2 ); //image central point

// debug: shows the coordinates of the first point of the object
/*
float ObjectDistanceX2, ObjectDistanceY2, ObjectDistanceZ2;
ObjectDistanceX2 =
CV_MAT_ELEM(*rotMatRod, float, 0, 0) * CV_MAT_ELEM(*object_points4, float, 0, 0) +
CV_MAT_ELEM(*rotMatRod, float, 0, 1) * CV_MAT_ELEM(*object_points4, float, 0, 1) +
CV_MAT_ELEM(*rotMatRod, float, 0, 2) * CV_MAT_ELEM(*object_points4, float, 0, 2) +
CV_MAT_ELEM(*translation_vector, float, 0, 0);
ObjectDistanceY2 =
```
A.4. POSITION CALCULATION

APPENDIX A. CODE

```c
365 CV_MAT_ELEM(*rotMatRod, float, 1, 0) * CV_MAT_ELEM(*object_points4, float, 0, 0) +
366 CV_MAT_ELEM(*rotMatRod, float, 1, 1) * CV_MAT_ELEM(*object_points4, float, 0, 1) +
367 CV_MAT_ELEM(*rotMatRod, float, 1, 2) * CV_MAT_ELEM(*object_points4, float, 0, 2) +
368 CV_MAT_ELEM(*translation_vector, float, 0, 1);
369
370 ObjectDistanceZ2 =
371 CV_MAT_ELEM(*rotMatRod, float, 2, 0) * CV_MAT_ELEM(*object_points4, float, 0, 0) +
372 CV_MAT_ELEM(*rotMatRod, float, 2, 1) * CV_MAT_ELEM(*object_points4, float, 0, 1) +
373 CV_MAT_ELEM(*rotMatRod, float, 2, 2) * CV_MAT_ELEM(*object_points4, float, 0, 2) +
374 CV_MAT_ELEM(*translation_vector, float, 0, 2);
375
376 printf("\nDistances between the central point of the camera and the central point of the object\n");
377 printf("----------------------------------------\n");
378 printf("| X: %f\n", ObjectDistanceX2);
379 printf("| Y: %f\n", ObjectDistanceY2);
380 printf("| Z: %f\n", ObjectDistanceZ2);
381 */
382
383 /* Manual calculation of the position of the axis without image deformation */
384 */
385 CvPoint2D32f point2D[4];
386 for (int i=0; i<4;i++)
387 {
388    CvPoint3D32f point3D;
389
390    point3D.x =
391    CV_MAT_ELEM(*rotMatRod, float, 0, 0) * CV_MAT_ELEM(*object_points7, float, i, 0) +
392    CV_MAT_ELEM(*rotMatRod, float, 0, 1) * CV_MAT_ELEM(*object_points7, float, i, 1) +
393    CV_MAT_ELEM(*rotMatRod, float, 0, 2) * CV_MAT_ELEM(*object_points7, float, i, 2) +
394    CV_MAT_ELEM(*translation_vector, float, 0, 0);
395
396    point3D.y =
397    CV_MAT_ELEM(*rotMatRod, float, 1, 0) * CV_MAT_ELEM(*object_points7, float, i, 0) +
398```
APPENDIX A. CODE

A.4. POSITION CALCULATION

        CV_MAT_ELEM(*rotMatRod, float, 1, 1) * CV_MAT_ELEM(*
              object_points7, float, i, 1) +
        CV_MAT_ELEM(*rotMatRod, float, 1, 2) * CV_MAT_ELEM(*
              object_points7, float, i, 2) +
        CV_MAT_ELEM(*translation_vector, float, 0, 1);

        point3D.z =
        CV_MAT_ELEM(*rotMatRod, float, 2, 0) * CV_MAT_ELEM(*object_points7,
              float, i, 0) +
        CV_MAT_ELEM(*rotMatRod, float, 2, 1) * CV_MAT_ELEM(*
              object_points7, float, i, 1) +
        CV_MAT_ELEM(*rotMatRod, float, 2, 2) * CV_MAT_ELEM(*
              object_points7, float, i, 2) +
        CV_MAT_ELEM(*translation_vector, float, 0, 2);

        if ( point3D.z != 0 )
          {
            point2D[i].x = ((1565 * point3D.x) ) / point3D.z;
            point2D[i].y = ((1555 * point3D.y) ) / point3D.z;
          }
        else
          {
            point2D[i].x = 0.0;
            point2D[i].y = 0.0;
          }
        }
      */
      
      /* Draw the axes in the central point of the image*/

      int centreX = static_cast<int>( image_pos2->width * 0.5 );
      int centreY = static_cast<int>( image_pos2->height * 0.5 );

      cvLine( image_pos2, cvPoint( centreX , centreY ),
              cvPoint( centreX + (int)point2D[1].x - (int)
                      point2D[0].x, centreY + (int)point2D[1].y - (int)point2D[0].y ), CV_RGB( 0, 255, 0 ), 2 );

      cvLine( image_pos2, cvPoint( centreX , centreY ),
              cvPoint( centreX + (int)point2D[2].x - (int)
                      point2D[0].x, centreY + (int)point2D[2].y - (int)point2D[0].y ), CV_RGB( 255, 0, 0 ), 2 );

      cvLine( image_pos2, cvPoint( centreX , centreY ),
              cvPoint( centreX + (int)point2D[3].x - (int)
                      point2D[0].x, centreY + (int)point2D[3].y - (int)point2D[0].y ), CV_RGB( 0, 0, 255 ), 2 );
      */
/* Draw the axes in the first point of the object */

int centreX = static_cast<int>(image_pos2->width * 0.5);
int centreY = static_cast<int>(image_pos2->height * 0.5);

cvLine(image_pos2, cvPoint(centreX + (int)point2D[0].x, centreY + (int)point2D[0].y),
      cvPoint(centreX + (int)point2D[1].x, centreY + (int)point2D[1].y), CV_RGB(0, 255, 0), 2);

cvLine(image_pos2, cvPoint(centreX + (int)point2D[0].x, centreY + (int)point2D[0].y),
      cvPoint(centreX + (int)point2D[2].x, centreY + (int)point2D[2].y), CV_RGB(255, 0, 0), 2);

cvLine(image_pos2, cvPoint(centreX + (int)point2D[0].x, centreY + (int)point2D[0].y),
      cvPoint(centreX + (int)point2D[3].x, centreY + (int)point2D[3].y), CV_RGB(0, 0, 255), 2);

#ifndef FESTO_CAMERA

    cvShowImage("Object position", image_pos2); /* Show object found */
    //cvWaitKey(1000);
#endif

    //cvWaitKey(0);

    cvReleaseImage(&image_pos2);

    /// stop debugg point ///</
    //printf("Before going out of posDetector\n");
    //cvWaitKey(0);
    /////////////////////////////////////////////////////
#ifndef FESTO_CAMERA

    // cvDestroyWindow( "Object position" );
#endif

    return (0);
}
/** Function : UndistortImage
** Description :
** Precondition :
** Postcondition: None
** Parameters : None
** Return value : None
**--------------------------------------------------------------------------*/

void UndistortImage ( IplImage* original_image, CvMat* intrinsicParameters, CvMat
  * distortionParameters )
{

  IplImage* mapx = cvCreateImage( cvGetSize(original_image), IPL_DEPTH_32F, 1 )
  ;
  IplImage* mapy = cvCreateImage( cvGetSize(original_image), IPL_DEPTH_32F, 1 )
  ;

  cvInitUndistortMap
  {
    intrinsicParameters,
    distortionParameters,
    mapx,
    mapy
  };
  IplImage *t = cvCloneImage(original_image);
  cvRemap( t, original_image, mapx, mapy ); // Undistort image
  cvReleaseImage(&t);
  cvReleaseImage(&mapx);
  cvReleaseImage(&mapy);
}