UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTODI INGEGNERIA DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

Master's Degree in Computer Engineering:
Artificial Intelligence and Robotics

Robotic system with TwinCAT Vision: techniques for identifying and manipulating geometric objects on a conveyor belt

Supervisor:
Prof. Giovanni Boschetti

Candidate:
Anna Furciniti
2057452

ACADEMIC YEAR 2023 – 2024
21/10/2024

Abstract

The integration of computer vision in Industry 4.0 marks an important revolution. Many industrial companies have adopted machine vision systems to identify defective products, assembly verification, robot guidance and OCR reading; arduous tasks that were previously labor-intensive, error-prone and time-consuming when performed by human. These advanced machine vision systems, equipped with cameras and artificial intelligence, now capture and analyze image of objects with remarkable precision, handling different functions and significantly enhancing operational accuracy. In fact, vision systems play a crucial role in robotics, enabling machines to operate in unstructured environments.

This thesis investigates the application of TwinCAT Vision library for the detection of geometric objects. TwinCAT Vision library facilitates industrial image processing tasks such as the detection, identification or measurement of objects directly within the PLC in real-time. It supports the use of images already saved on the PC as well as the deployment of a GigE Camera for real-time operations. Utilizing the camera MakoG192B, TwinCAT Vision identifies various shapes in motion and accurately locates their center coordinates. This capabilities enhances robot interactions with production lines, which are often controlled by PLCs.

This study highlights different vision algorithms for shape center detection and demonstrate their effectiveness through rigorous experimental tests. These algorithms enable precise localization of objects centers, which is crucial for tasks requiring high accuracy. The coordinates determined by the system are utilized by a SCARA robot for manipulation, thereby proving the strategy's efficacy in real-world applications.

By integrating image processing into the TwinCAT platform, the system achieve highly synchronized control applications and extremely short response times, showcasing significant advancements in both the efficiency and reliability of automated industrial processes.

Italian Abstract

L'integrazione della visione artificiale nell'Industria 4.0 segna una rivoluzione importante. Molte aziende industriali hanno adottato sistemi di visione artificiale per l'identificazione di prodotti difettosi, per la verifica nell'assemblaggio di pezzi, guidare robot e leggere OCR; compiti ardui che in precedenza erano laboriosi, inclini agli errori e occupavano molto tempo poichè eseguiti dall'uomo. Questi sistemi di visione artificiale avanzati, equipaggiati con telecamere e intelligenza artificiale, ora catturano e analizzano immagini di oggetti con notevole precisione, gestendo diverse funzioni e migliorando significativamente l'accuratezza. Inoltre, i sistemi di visione svolgono un ruolo cruciale nella robotica, permettendo alle macchine di operare in ambienti non strutturati.

Questa tesi approfondisce la libreria TwinCAT Vision per il rilevamento di oggetti geometrici. TwinCAT Vision, funzione presente in TwinCAT 3, facilita i compiti di elaborazione di immagini industriali come il rilevamento, l'identificazione o la misurazione di oggetti direttamente all'interno del PLC in tempo reale. Supporta l'uso di immagini già salvate sul PC così come l'impiego di una telecamera GigE per operazioni in tempo reale. Utilizzando la telecamera MakoG192B, TwinCAT Vision identifica varie forme in movimento e localizza accuratamente le coordinate dei loro centri. Queste capacità migliorano le interazioni dei robot con le linee di produzione, che sono spesso controllate dai PLC. Questo studio evidenzia diversi algoritmi di visione per il rilevamento del centro delle forme e dimostra la loro efficacia attraverso rigorosi test sperimentali. Questi algoritmi permettono una localizzazione precisa dei centri degli oggetti, cruciale per compiti che richiedono alta precisione. Le coordinate determinate dal sistema vengono utilizzate da un robot SCARA per una manipolazione precisa, dimostrando così l'efficacia della strategia in applicazioni reali. Integrando l'elaborazione delle immagini nella piattaforma TwinCAT, il sistema raggiunge applicazioni di controllo altamente sincronizzate e tempi di risposta estremamente brevi, mostrando significativi progressi sia nell'efficienza che nella affidabilità dei processi industriali automatizzati.

Index

CHAPTER 1: INTRODUCTION 9

1.1 Thesis structure 9

1.2 Objective 9

1.3 Industry 4.0 9

1.4 Artificial Intelligence 11
1.4.1 Neural Networks(NNs) and Convolutional Neural Networks(CNNs) 11

1.5 Computer vision 12

1.5.1 Classification vs. Regression 13
1.5.2 CV algorithms 13

1.6 Process of images creation 14
1.6.1 Camera 16
1.6.2 Camera 2D and 3D 17

1.7 Industrial robot 18
1.7.1 PLC 19

CHAPTER 2: TWINCAT 3 20

2.1 Development environment structure 20
2.1.1 Modules 21

2.2 PLC in TwinCAT 23
2.2.1 Creation and running of an application 24

2.3 Vision Library 25
2.3.1 Process Overview 28
2.3.2 Functions 28

CHAPTER 3: TWINCAT CODE IMPLEMENTATION 30

3.1 Camera configuration 30
3.1.1 Connection camera and computer 30
3.1.2 Camera features 33
3.1.3 Camera Assistant 35
3.1.4 Camera Calibration 36
3.1.5 PLC connection 39
3.1.6 Acquisition and trigger 39

3.2 Circles and Rectangles centers recognition 43
3.2.1 Filters application 43
3.2.2 Object Detection Implementation 47
3.2.3 Rectangles 49
3.2.4 Circle 53
3.2.5 Object Center Detection results 54

3.2.6 Template Matching 58
3.2.7 CannyEdgeDetection 59

CHAPTER 4: MACHINE LEARNING 62

4.1 Introduction 63

4.2 Dataset generation 64

4.2 Data augmentation 70

4.3 Models 72
4.1.1 First model 74
4.1.2 Second model 75
4.1.3 Third model 76

4.4 Training 78

4.5 Results in Google Colab 79

4.6 Integration in TwinCAT 82

CHAPTER 5: EXPERIMENTS WITH THE ROBOT 83

5.1 Coordinate systems 84
5.1.1 Robot's system 84
5.1.2 Photocell system 84
5.1.3 Camera system 85

5.2 Conveyor velocity 85

5.3 Coordinate transformation 87

CONCLUSION 90

IMAGES 92

SITOGRAPHY 96

Chapter 1: **Introduction**

In this chapter are highlighted the objectives of this thesis with a short explanation of general concepts useful to understand the topics present in the subsequent chapters.

1.1 Thesis structure

The first chapter is an introduction to the world of computer vision starting with the history regarding the industry 4.0, artificial intelligence, industrial robots and some detail about camera features.

The second chapter deals with a general explanation of the environment used for this project and its relative modules: the TwinCAT 3 environment integrated into Microsoft Visual Studio.

The third chapter includes the main experiments: starting with the camera settings and connection, and concluding with object recognition using different algorithms and different vision techniques.

The fourth chapter focuses on the machine learning module, where a convolutional neural network is trained in Google Colab and integrated into TwinCAT environment.

The last chapter explains the connection between the vision system and the robot routine.

1.2 Objective

The main objective of this project begin with the basic concepts of artificial intelligence and ends with the deep exploration of computer vision techniques.

Different algorithms and methods, such as blurring filters, edges and contours recognition, and machine learning, are used to define the best solution for geometric object detection.

After the recognition of the object, its coordinates are converted in World Coordinate with the respect to the camera system for the last stage, where the robot performs a pick and place, synchronized with the conveyor belt. Nowadays this process is one of the most widely used in the industrial automation and is what we define as Industrial 4.0 Revolution.

1.3 Industry 4.0

The industrial revolution began in 1780 with the introduction of steam and water power, marking a significant transformation in manufacturing and transportation, named Industry 1.0. Following this, the second Industrial Revolution, or Industry 2.0, emerged in 1870 with the introduction of electrical power, which accelerated production capabilities.

As advancements continued, Industry 3.0 was the introduced, characterized by the automation of production processes through the use of electric devices, programmable logic controllers(PLCs) and robotics. This shift to automation led to more efficient production processes compared to the previous manual method employed.

Today, we refer to the ongoing transformation as the Fourth Industrial Revolution, or Industry 4.0, which introduced the concept of the 'Smart Factory'. In these factories, autonomous systems interact seamlessly through the Internet of Things (IoT) and cloud technologies, allowing for real-time communication and collaboration between machines and humans.

Modern technologies are radically revolutionizing the manufacturing by integrating robotics, big data, artificial intelligence, computer vision and various sensors. This synergy has significantly improved human work in terms of efficiency, cost-effectiveness and time management.

A key component of Industry 4.0 is computer vision, which play a crucial role across many field. Many industrial companies now use machine vision systems to identify defective products, an arduous task previously done by humans that was prone to errors and time-consuming. Now these machine vision systems, using a camera and the artificial intelligence, can now capture and analyze image of objects, identifying defects with remarkable precision.

Moreover, machine vision system facilitates automated inspection through sophisticated image processing techniques, providing real-time data during production that can help early detection of potential issues.

In Industry 4.0 machines are interconnected within an ecosystem named IoT (Internet of Things), then the computer vision increase the power and usability of sensors within this IoT network, providing real time data capture and analysis.

This capability is crucial for:

- Automated quality inspection: checking product quality by detecting defects such as irregularities in a biscuit's shape or color inconsistencies.
- Robotic applications: enabling precise 'pick and place' operations that require visual identification of object positions and orientations.
- Traceability: using QR codes for tracking products.
- Safety enhancements: improving workplace safety by monitoring environments for unauthorized access.

As we advance further into the age of smart manufacturing, the role of computer vision continues to expand, driving innovations that transform how industries operate. The combination of AI and computer vision with Industry 4.0 is setting new benchmarks in how quality and safety are managed in production environments.

1.4 Artificial Intelligence

Artificial intelligence (AI) is the ability of a digital computer or computer-controlled robot to perform tasks commonly associated with intelligent beings[3]. Then the goal is to generate an intelligent agent that select the most advantageous action in a given scenario.

But, what does “intelligent” means? Intelligence include an agent’s ability to adapt its decision to new circumstances. Different research in AI focuses on different components of the intelligence including learning, reasoning, problem solving, perception, and using language.

The earliest work in AI is conducted by Alan Turing, involving an experiment with three participants: a computer, a human interrogator, and a human foil. The goal for the interrogator is to determine, through questioning, which participant is the computer. This experiment laid the foundational concept for what is now known as the Turing Test, a standard for evaluating machine’s ability to exhibit intelligent behavior equivalent to, or indistinguishable from, that of a human.

1.4.1 Neural Networks(NNs) and Convolutional Neural Networks(CNNs)

AI integrates numerous subfields including Machine Learning (ML), Deep Learning(DL) and Computer Vision(CV).

Deep learning, in particular, is crucial and relies on neural networks(NNs). These networks are analogous to the neural circuitry of the human brain, comprising a collection of nodes organized into three layers:

- Input layer: where data is received.
- Hidden layer: intermediate layers that process inputs through various computational functions.
- Output layer: Produces the final decision or classification, such as identifying the content of an image.

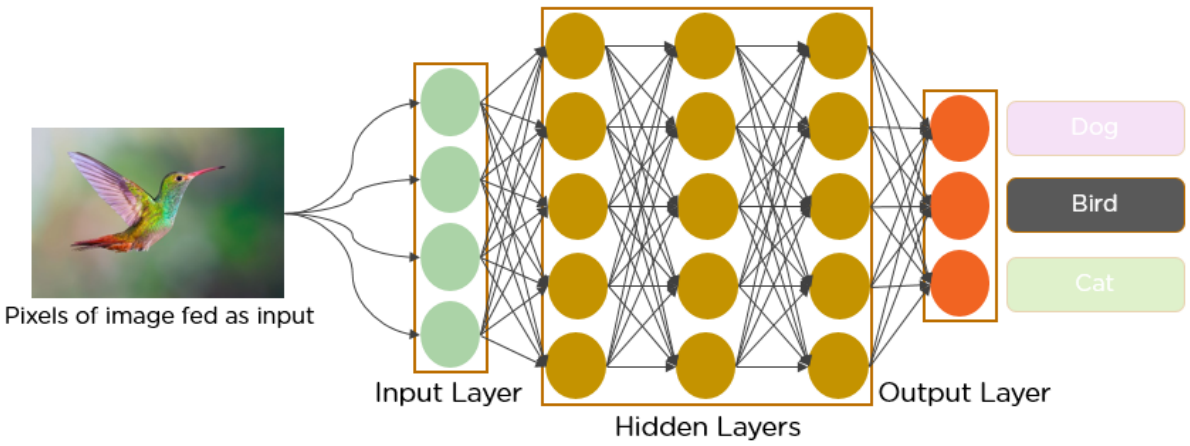

Figure 1: Convolutional Neural Network

Each layer is interconnected, allowing the information to flow from the input to the output. For instance (Figure 1), in computer vision, an image fed into the network undergoes multiple transformations through the hidden layers before the network output a label identifying the image. The NN used for the vision application is called Convolutional Neural Network(CNN).

Deep learning has revolutionized the field of computer vision, enabling machines to identify and classify objects within images with high accuracy. It has practical applications in our real life such as automated quality control in manufacturing, facial recognition for security, and navigation for autonomous vehicles.

1.5 Computer vision

Computer vision, a subset of artificial intelligence, equips computers with the capability to process and interpret the visual world. By using cameras, it enables the identification and classification of objects.

The first experiments were conduced in the 1950s focuses on rudimentary experiments such as object edge detection via neural networks. Initially, the tasks performed by machines were quite basic, such as recognizing simple shapes.

Nowadays, computer vision is omnipresent across the various devices. For instance, smartphones use their cameras to identify objects, animals or people. Similarly, image galleries on these devices often automatically categorize photos by recognizing different faces, thanks to computer vision.

A key driver for the growth of computer vision in these applications is the vast influx of data from sources such as smartphones, security systems, and traffic cameras.

Computer vision typically operate through several steps:

- Acquiring input data, which are the images;
- Pre-processing the images;
- Interpreting the images to classify them.

Training is a crucial phase in computer vision, requiring substantial datasets. During this phase, neural networks are trained with extensive image sets. After processing through various hidden layers, the neural networks become adept at predicting accurate image classifications or continuous values with regression algorithm.

1.5.1 Classification vs. Regression

Classification divide the dataset into classes like 'Male' or 'Female', 'True' or 'False', etc. Regression helps in predicting continuous values such as 'house prices', 'weather patterns', etc.

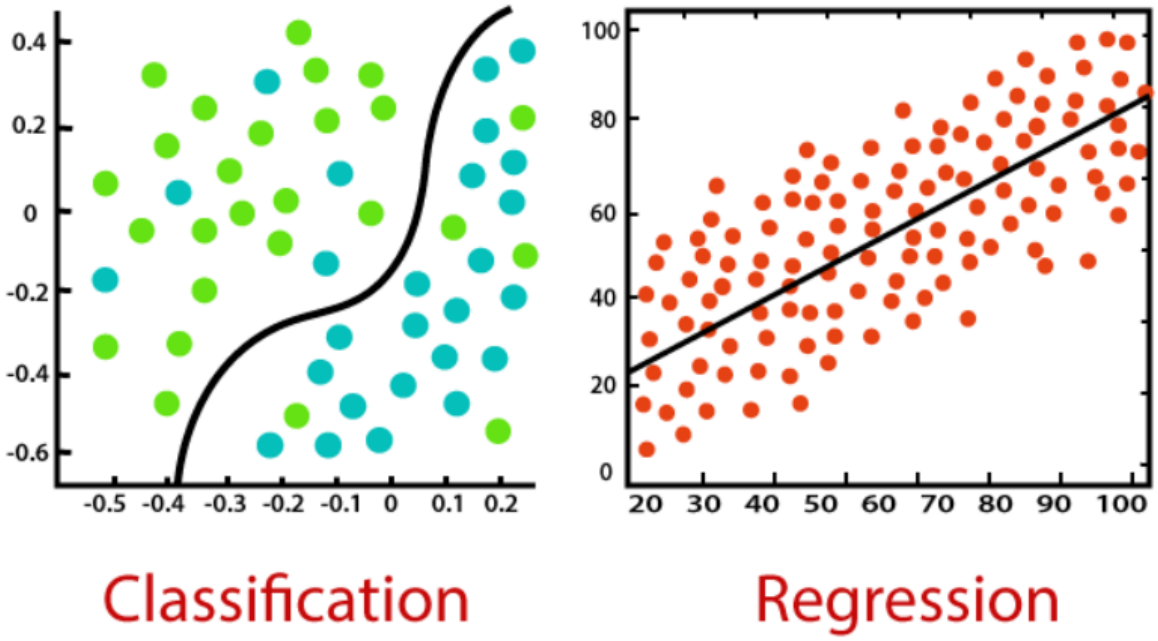

Figure 2: Classification or Regression models

1.5.2 CV algorithms

The process of image interpretation can be performed using different filters to enhance image quality:

- Noise reduction: filters such as smoothing, mean or gaussian filters reduce the rumors present in an image.

  These filters use a kernel, a matrix of pixels, whose size depends on the type of noise or on the specific task. The pixels within this kernel are used for calculations like mean, where kernel slides across the image pixel by pixel. Then the result is a central

pixel that replaces the area of previous kernel in the new output image, making the image clearer.

- Image segmentation: divides an image into regions (ROI-Region Of Interest). Consequently the focus is only on relevant segment optimizing analysis and improving resulting accuracy.
- Object detection: allows for the localization and classification of objects by using edge detection, line detection or hough transformations.
- Pattern detection: identifies recurring patterns within images.
- Feature detection: searches for relevant features useful for the classification.

1.6 Process of images creation

The process of image creation involves several critical factors, including the light source, the object placement and the sensor orientation.

Before starting with system application, it is essential to understand the basic structure of an image. An image is a collection of pixels generating the visual representation of a digital image on screen.

The color of such pixels is determined by the model used; the most used is the RGB (Red Green Blue) model.

In RGB model, the pixel value is represented by a triplet ranging from 0 to 255, which describe the brightness of given point in the image. For example, a triplet of [0,0,0] stands for black, while [255,255,255] represents white; all other colors are derived by combining numbers within the range [0,255].

Instead for the Gray scale image the pixel value is represented by a single number: [0] stands for black, while [255] represents white; other colors are intermediate value representing different shades of gray.

The camera used for this thesis employs images in GRAY model. In both cases, it is possible to convert from one model to another.

In general, a system like the one shown in the Figure 3, starting from Scene element and illumination source, creates an image by collecting the incoming energy and projecting it onto the image plane. The light source should remain consistent; variations can lead to different results and issues such as reflections and shadows, which may cause problems in the image interpretation.

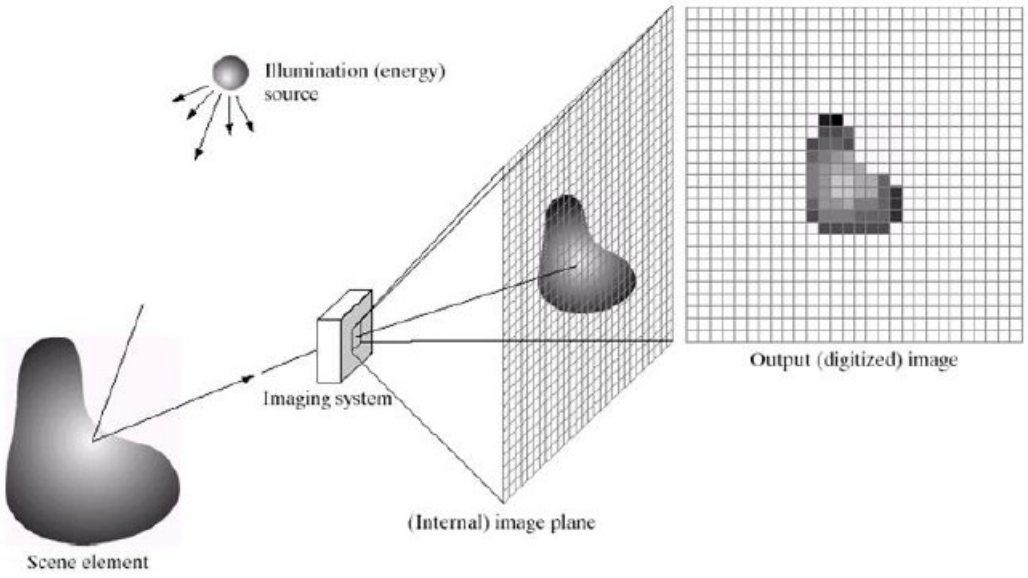

Figure 3: Creation of digitized image

An image is represented using Cartesian coordinates, where pixel value are organized in a grid.

The Pinhole camera model plays a crucial role in understanding the basics of image formation and in connecting photography with new technologies in computer vision.

It describes the mathematical relationship between the coordinates of a point in three-dimensional space and its projection onto the image plane of an ideal pinhole camera[4]. The key elements of this model (Figure 4) are:

- Optical center: the geometric point through which every ray from the object to the sensor passes.
- Image plane: the plane where the image rays converge to form the visual image.
- Optical axis: the straight line that passes through both the center of the image plane and the optical center.
- Principal point: the intersection between the image plane and the optical axis.
- Focal length: the distance between the optical center and the image plane.

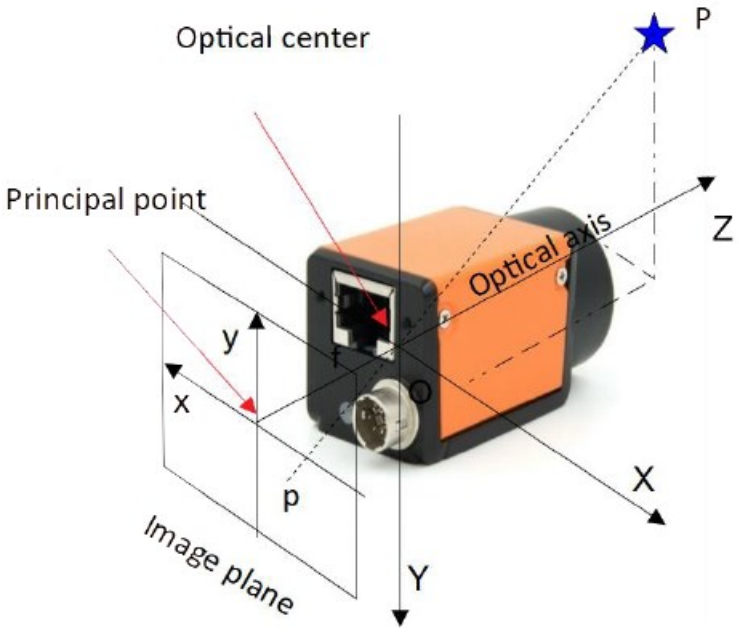

Figure 4: Pinhole model

The advantage of this model lies in how a simple aperture can form an image.

1.6.1 Camera

In practical applications with a real camera system, several factors must be considered to improve effectiveness:

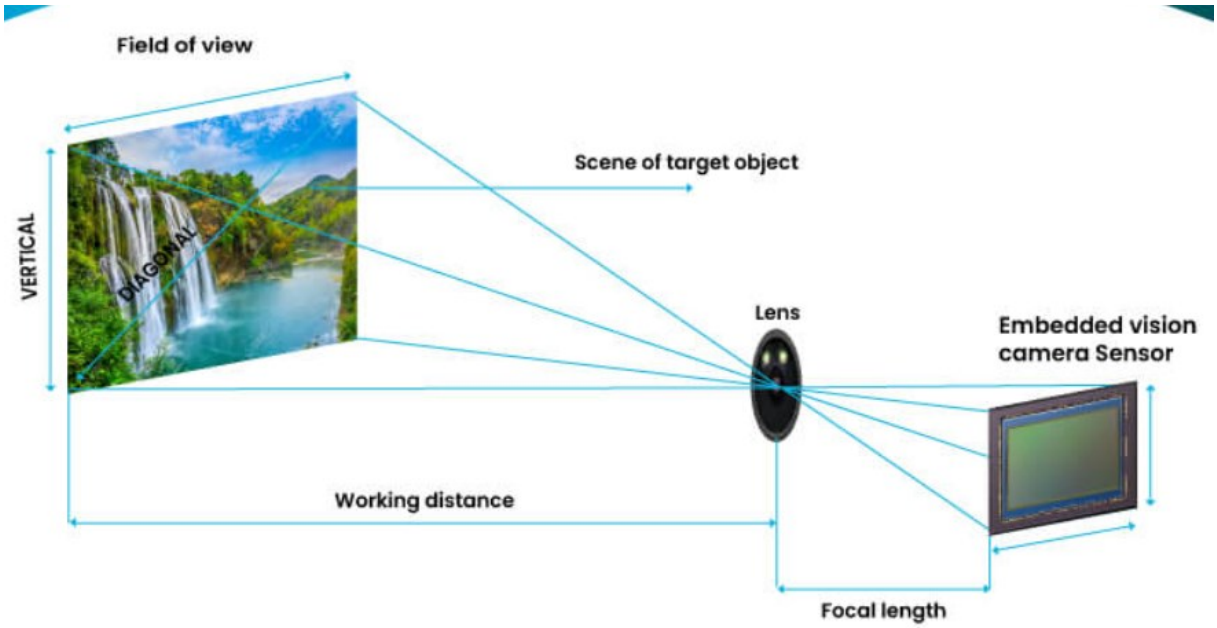

Figure 5: FoV and focal length

- Field of view(FoV): the angle perceived by the camera determines how much the camera is able to see of the object.

  A larger FoV means the camera can capture more of the scene.

When the FoV decrease, the focal length increases.

The general formula for the FoV calculation is (Sensor Dimension/ Focal Length) x 57,3. The Sensor Dimension is the size of the image sensor in millimiters, while 57,3 is the conversion factor from radians to degrees.

- Focal length: is the distance in millimeters from the optical center of the lens to the camera's sensor, which determines the composition of the captured image.
- Focus: adjusting the lens to clearly define objects.
- Aperture: controls the amount of light entering through the lens; a wider aperture allows more light in and decreasing the depth of field.
- Exposure time: the duration of the camera's sensor is exposed to light, measured in fractions of a second. A slow time is used for low light environments, while a fast exposure is ideal for freezing motion.
- Depth of field: the range within which objects appear acceptably sharp in an image.

Some lens can cause a distortion, which is a deviation from the ideal image reproduction:

- Radial distortion: depends on the distance of the distorted point from the image center. It combines Pincushion and Barrel distortion.
- Tangential distortion: caused by the misalignment between lens and sensor.

Figure 6: Radial and Tangetial distortion

The best way to remove distortion is through calibration, a process that involves estimating intrinsic and extrinsic parameters. To do this, a calibration pattern with a known shape, such as the square corners of a checkerboard, is necessary.

After the image capture, the next step involve the processing, where noise is removed and the image is improved by using some filter.

1.6.2 Camera 2D and 3D

Modern industrial vision machine manages two different cameras:

- 2D camera: the image is in two dimensions, X and Y, no height Z defined; it is pratical for the quality check, for characters reading and OCR. However, light changes can negatively impact the results, such as reducing precision in the contours of the objects.
- 3D camera: the image is tridimensional with X, Y and Z axes; it is advantageous for specific tasks like the measurements of shapes, height, angles and volumes; such as, in bin picking applications, the robot needs a precise representation in a tridimensional space.

The camera used for this project is a 2D camera.

These fundamental concepts support every visual application and are crucial for the discussion in this thesis, providing the foundational understanding necessary to explore the advanced visual technologies driving Industry 4.0.

1.7 Industrial robot

Robotic arms can execute sophisticated tasks with different configurations. Their structure and operational principles are the following:

- Links: are rigid components of an arm. They move through space respecting the mechanical limitation of the robot. Each link connects to joints.
- Joints: connect two links allowing movement. The main types of joints are: revolute and prismatic, which allow respectively rotational and translational movements.
- Degrees of Freedom(DoF): are determined by the number of axes influencing the robot's movements. More axes mean more versatility in movement. For instance, typical industrial companies use for a pick and place application robot with three to five axes. Instead robot for more complex applications may have over six degree of freedom.
- Robot Base: one end of the arm is connected to stationary base.
- End-Effector: the other end of the arm is connected to an end-effector, such as a gripper, used for manipulating objects.

Let's introduce how manipulators work:

- Kinematics treats motion without understanding the forces that cause it. Forward Kinematics in particular compute the position and orientation of the end effector relative to the base frame. Inverse Kinematics, reverse process of forward, treats the problem of calculate all possible sets of joint angles that would result in position and orientation of end-effector.

1.7.1 PLC

A Programmable Logic Controller (PLC) is a type of specialized computer that is used for controlling industrial processes and machinery.

PLCs are widely employed in the automation, such as for controlling machinery on factory assembly lines or food processing operations. They gather input from sensors or user interactions, process these inputs based on a pre-defined sequence, and generate outputs to actuators or other devices to manage machinery or processes. The programming logic used in a PLC can be implemented through various languages, including ladder logic, structured text, or block diagrams.

Robots are usually controlled though their own proprietary controllers. These controllers are then, in turn, connected to the PLC, which generally acts as the main controller that oversees different robotic components and their interactions within the overall process. For instance, in a manufacturing environment, the PLC may manage the motion of a robotic arm, the functioning of conveyor belts, the camera and the operation of sensors and switches. This arrangement ensures that operations are coordinated and efficient, where timing and sequence are essential.

Chapter 2: TwinCAT 3

The software used for this thesis is TwinCAT 3, developed by Beckhoff, which stands for The Windows Control and Automation Technology. It is a PC-based control system for industrial automation transforming any PC into a real-time system.

The first version of TwinCAT was introduced in 1996, leveraging the robust capabilities of Microsoft Visual Studio 2010, 2012, and 2013.

The integration of TwinCAT 3 with Microsoft Visual Studio offers several benefits, including increased productivity and minimized setup time through advanced tools for profiling and debugging. Additionally, it is possible to extend the environment with plugins and supports multiple programming languages.

Another benefit is its modularity, which enables functional changes and additions at any time.

2.1 Development environment structure

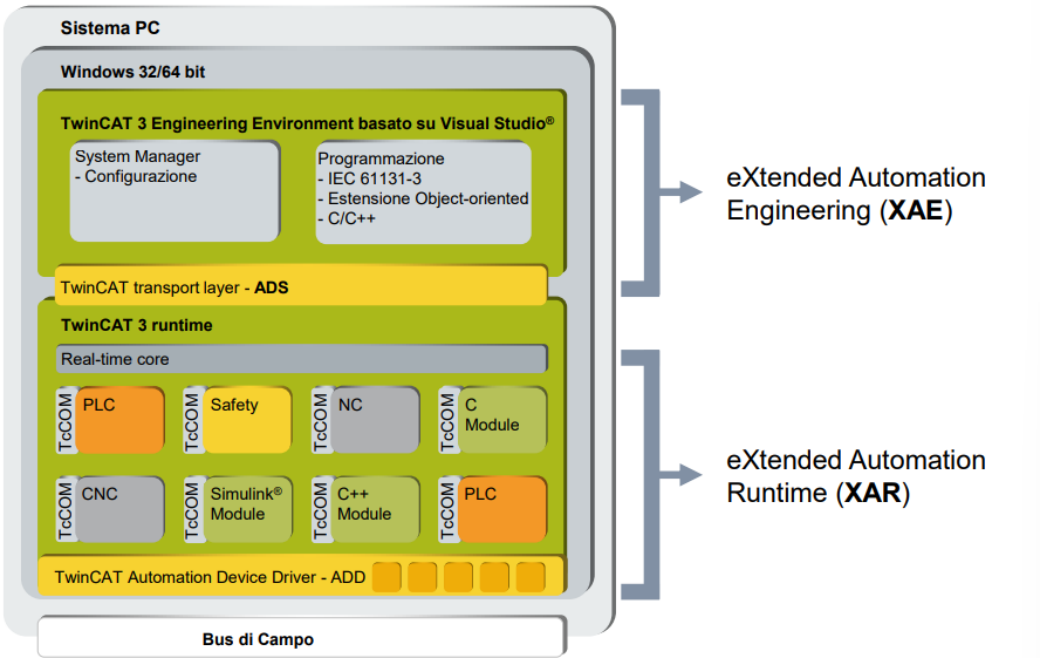

Figure 7: XAE and XAR in TwinCAT 3

The general structure of the environment is divided in two parts, as shown in the Figure 7:

- Environment in Visual Studio: based on XAE (Extended Automation Engineering) and includes the programming section used for the coding and for debugging.
  C and C++ can be used for the real-time programming.
  XAE allows hardware to be programmed and configured within a single engineering tool.

The vision resources are managed by XAE, considering camera configurations, calibration, simulation, and file source control. The choice of the camera and the loading of the file source are both considered in this section of the environment.

Once the code is completed, it is compiled and sent to the runtime part.

- Runtime based on XAR (Extended Automation Runtime): In this section, modules can be loaded, run and managed.

  They can be programmed independently by different developers.

  Its output is the Fieldbus, which by default is EtherCAT. The fieldbus enables the communication with sensors and activate actuators.

  Its main task are as follows:

  - Executing modules in real time.
  - Supporting the multi-core CPU.
  - Supporting a 64-bit operating system.

The two parts communicate using a protocol called ADS (Automation Device Specification), which can be used to upload new software or, if necessary, to read or write new variables.

2.1.1 Modules

TwinCAT is partitioned into modules, each with a distinct purpose, enabling functionality related to different aspects of industrial automation.

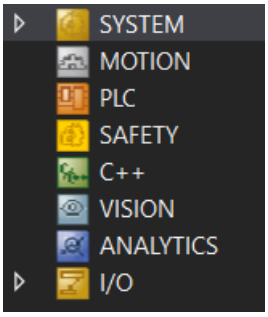

Figure 8: Modules defined in TwinCAT

Then TwinCAT project is divided into the following modules as shown in Figure 8:

- System: handles the basic setup of the entire automation system, including core configuration and router memory.

  In the core configuration, TwinCAT enables a distinction between Windows cores and isolated cores. The operating system and TwinCAT share the processor time, with the real-time portion allocated between 10% and 90%. In contrast, isolated cores are fully dedicated to TwinCAT application.

For vision applications it is recommended the use of isolated cores, such as in the Figure 9.

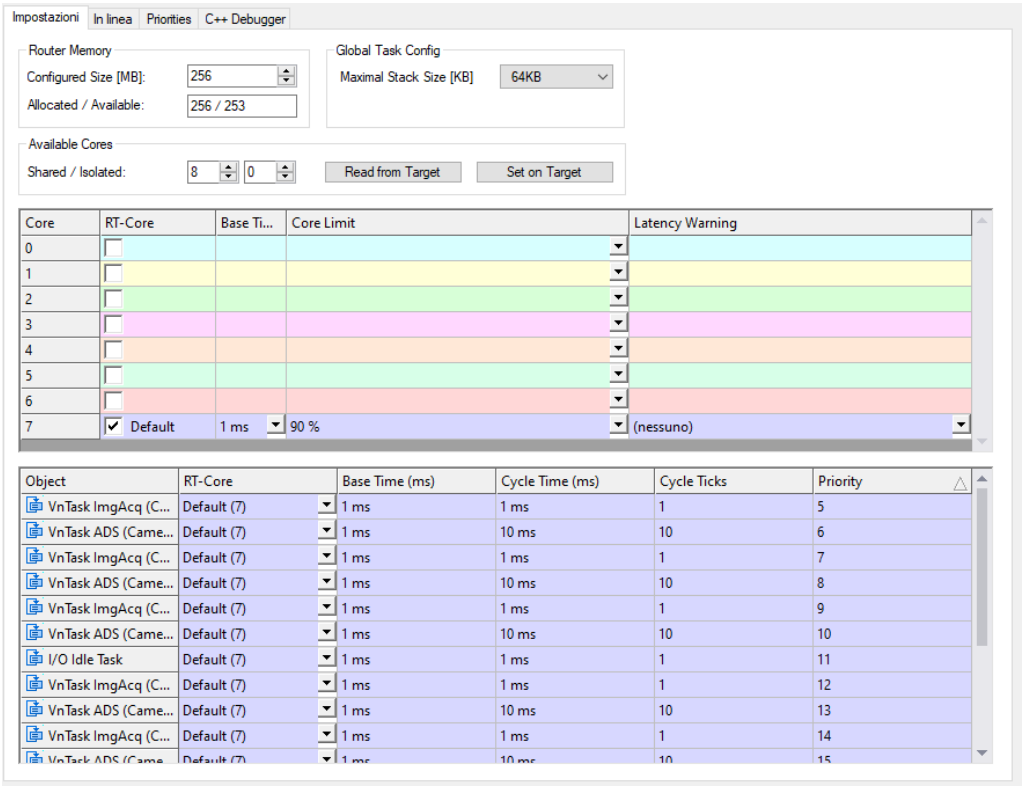

Figure 9: Router Memory, Available Cores (Shared/Isolated)

- Motion: supports motion control in automation systems.
- PLC: serves as the core of TwinCAT's automation, used for task implementation.
- Safety: implements links between safety-related inputs and outputs, such as Emergency Stop.
- C++: used for high level programming.
- Vision: integrates computer vision into TwinCAT, allowing image capture and processing.
- Analytics: used for processing and analyzing data collected from various parts of the system. Users have access to a toolbox of algorithms for implementing life time and cycle time.
- I/O: contains the configuration for tasks, with all variables known to I/O section. There is entry IO at any rate.

These modules are designed to cooperate and communicate with one another. This design allows complex application to be divided into manageable, smaller, simple parts, each one with a specific objective; this modularity not only simplifies the development of each module but also enhances the overall efficiency of the system by allowing parallel development and

testing. When modification or troubleshooting are required, this modularity provides highly advantages.

2.2 PLC in TwinCAT

A PLC, Programmable Logic Controller, is a ruggedized computer used for industrial automation. These controllers can automate a specific process, machine function, or even an entire production line[7].

The PLC capture the information from connected sensors, processes the data and triggers outputs, enhancing productivity through the automation of the starting and ending processes.

In Figure 10, the operation of a PLC is described. A PLC is generally connected to a power supply and consists of a central processing unit(CPU), a mounting rack, read-only memory(ROM), random access memory(RAM), input/output modules and a programming device. One of the advantages of a PLC is its modular organization:

- Rack: the part in which CPU, I/O modules and the power source connect. The CPU, as usual, is the brain of the PLC.
- Power supply: converts alternating current to direct current.
- Programming device: usually a PC.

PLCs are used for a wide variety of automated machine processes, including the monitoring of security cameras, traffic lights and industrial operations.

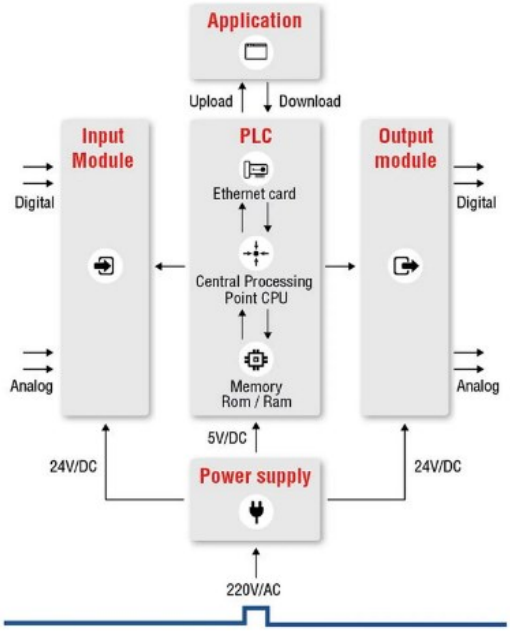

Figure 10: PLC structure and connections

Creating multiple PLCs involves dividing the program into several modules and running them on a target device. The benefits produced by this procedure can be summarized as follows:

- Uniform structure of the program.
- Understandable name of variables and instances.
- Intuitive readability of the code, even for individuals who are not involved in its development.
- Simplified maintenance.
- High code quality.

In TwinCAT 3, the project in which the PLC is programmed contains several elements crucial for operation of the controller program:

- Interfaces
- Functions: these encapsulate the image processing algorithms. A function is structured as follows:

  hr := F_VN_ProcessImage(<….>, hr);

  where hr is the status variable applicable to each function, F_VN is the prefix identifying the functions, and <….> represents the parameters useful for the function, such as input or output.
- Function Blocks: useful for complex processes, such as the communication with GigE Vision Camera.

2.2.1 Creation and running of an application

To create an application program that can run on the controller, use POUs (Program Organization Units) for declarations and implementation code, link the I/Os of the controller to program variables, and configure a task assignment. POUs are supported by programming language editors and are the main structure of the program.

The first step is the creation of a new PLC project selecting ‘standard PLC project’ as template and choosing a name, such as in the Figure 11.

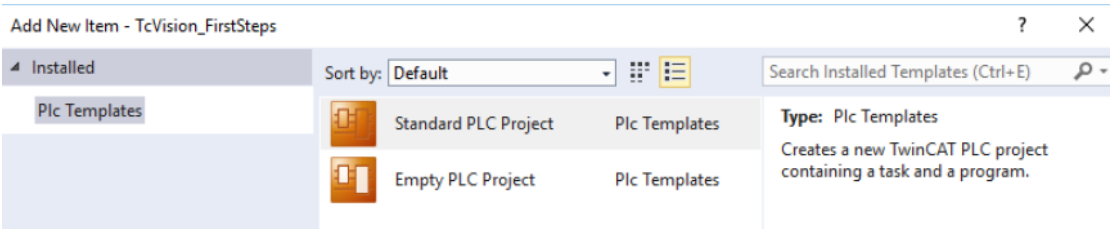

Figure 11: Create new project in Visual Studio with integrated TwinCAT 3

The PLC will then be automatically created with the MAIN program. It is also possible to define libraries are used in this project by modifying the References. For this thesis I will add the Tc3_Vision library.

As in any in code, initialize variables in the top section using VAR and END_VAR, and create the main program for control in the bottom section with PROGRAM. After defining the program to be executed, proceed to the task configuration present the system module of the TwinCAT program.

Finally, the PLC module should be compiled using Build command. After selecting the target system, follow these steps to run the program:

- Click on Activate Configuration in the TwinCAT XAE toolbar to activate the configuration.
- Accept the activation of the configuration. When the configuration is activate, and TwinCAT is setted to the Run mode with no error displayed, you are ready to load the PLC project.
- Click on Login in the PLC toolbar.
- Click on Start ; the program is now running. You can monitor the variables of the various program blocks in real time; it is possible to see the actual values of a variable directly in the block editor.
- Click on Stop on the left to temporarily halt execution, or click on Logout on the right to stop all processes.

2.3 Vision Library

In the PLC section 'References' contains all the library used in the project. Therefore, at beginning it is essential to add the TwinCAT Vision Library in this section.

By clicking, Figure 12, on References, the option 'Add library' will appear.

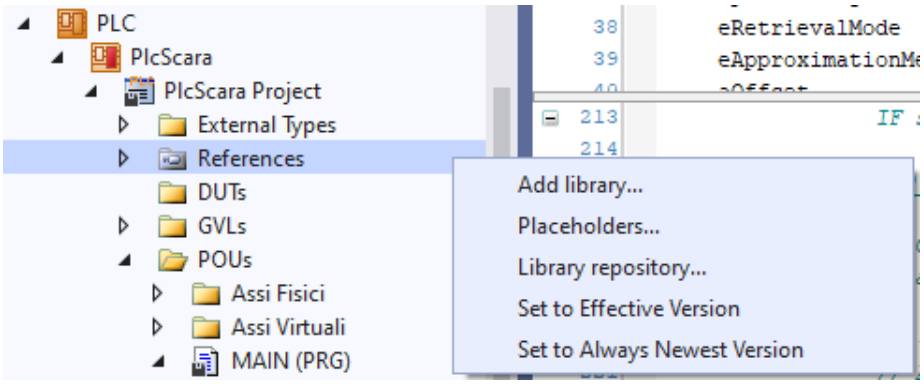

Figure 12: Add library in References

Tc3_Vision library, Figure 13, is the one necessary to run this project.

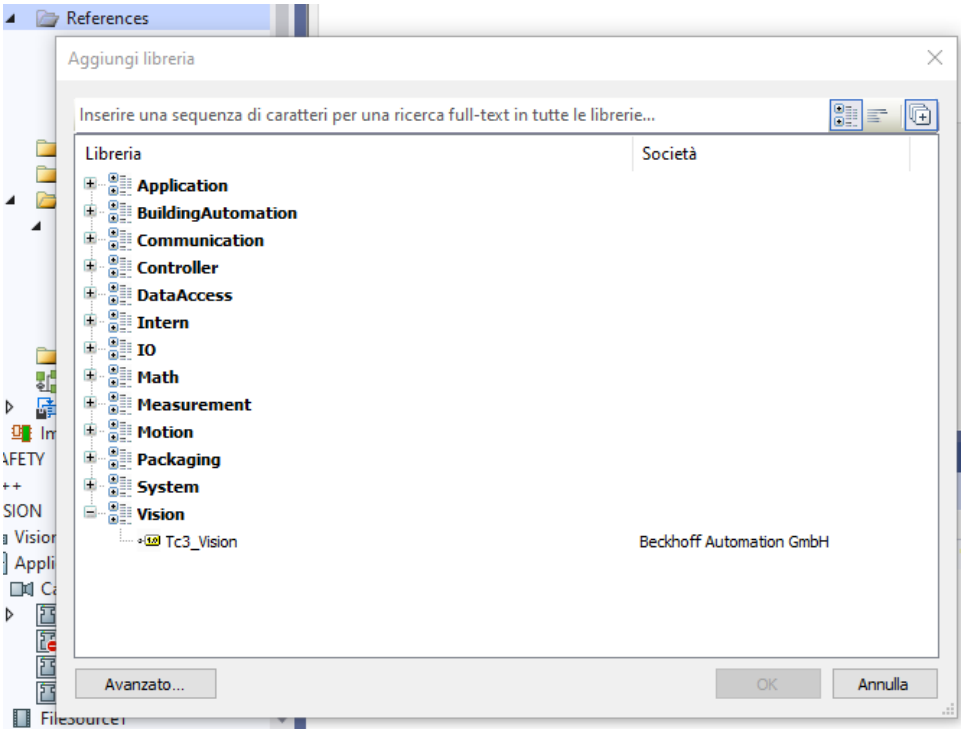

Figure 13: Tc3_Vision is added to the project

TwinCAT Vision is directly integrated into the TwinCAT Engineering environment for image processing solution. It enhances image processing capabilities with a universal control platform that incorporates PLC, motion control, robotics, IoT and HMI.

One of the advantages is the synchronization of all control functions related to image processing in real time, effectively removing latency. This addition significantly increases the importance of image processing in applications such as Industry 4.0.

TwinCAT Vision can be applied in different fields:

- Measurement: distances, diameters and roundness.
- Detection: pattern recognition, position detection and color recognition.
- Identification: Data Matrix code, Bar code and QR code.
- Monitoring: machine oversight, simplified service and simplified maintenance

In this library, there are two different ways to use the images in the TwinCAT program (Figure 14):

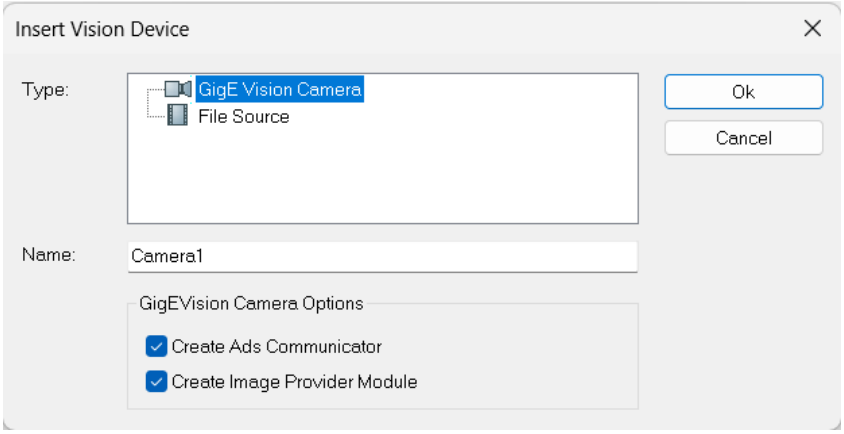

Figure 14: Vision device

- A camera can be connected and configured under a Vision node, calibrated and started for capturing and recording images. The creation of a GigE Vision Camera object begins with a new node and a connection to the camera via its IP address. After this the camera starts with the acquisition process.

  TwinCAT supports both line-scan cameras. which record single line of pixels at time, and area-scan cameras, which are based on rectangular sensors with a GigE Vision interface.

- An individual image, saved on the PC, can be loaded and modified by implementing image processing procedures. The creation of a File Source object start by adding a new node in the vision module and selecting the device type ‘File Source’ such as in Figure 14.

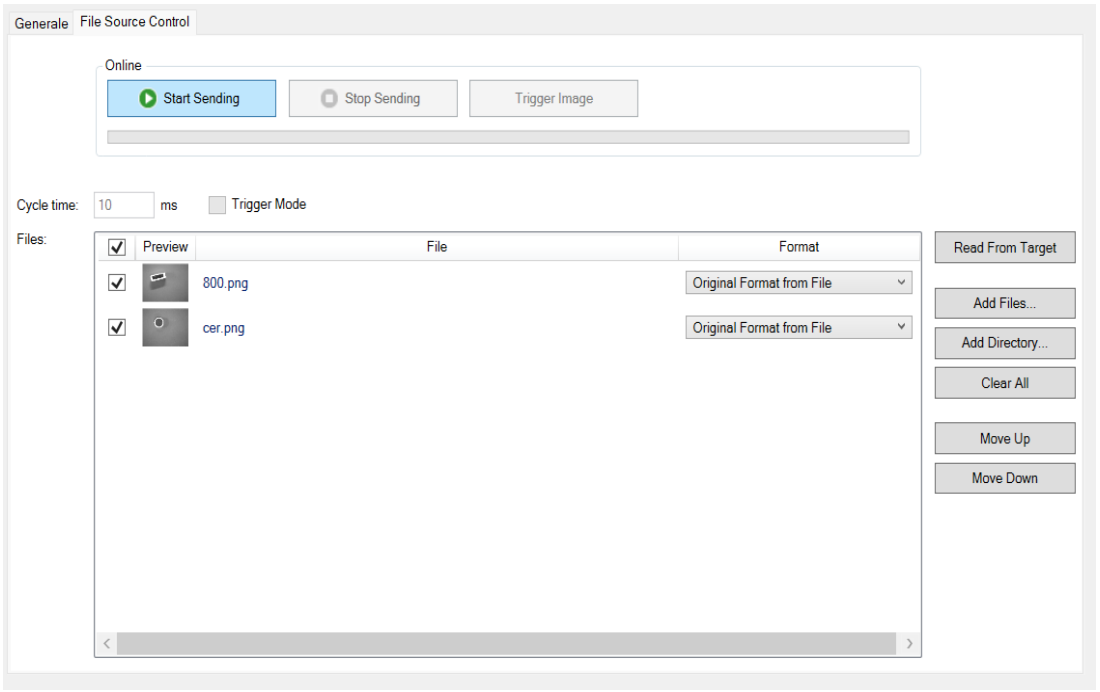

Figure 15: Add images in File Source Control

In File Source Control tab, Figure 15, you have the option to ‘Read From Target’ and load an image by clicking ‘Add Files’.

Depending on the program’s needs, you can change the image format from ‘Original Format from File’ to ‘24-bit RGB’ or ‘8-bit Monochrome’.

The respective object ID will be connected to the corresponding PLC task.

You can easily switch from the live camera view to recorded images by only clicking in the specific area of the vision module.

The sequence of image processing occurs directly within the PLC program, with the analysis chain executed in TwinCAT runtime system, enabling communication with other processes running on PLC.

2.3.1 Process Overview

The process begins with the acquisition of an image using a camera, followed by the subsequent steps:

- Object recognition: pre-process the input image by applying filters to remove noise, converting it to grayscale, or applying threshold techniques. This step involves identifying the contours of the objects in the image.
- Measurement and Further Analysis: measure the contours, calculate attributes such as area, perimeter and center of the specific figure. This analysis provides statistics for objects identified.
- Result visualization: draw the results on the image and display it for interpretation.

2.3.2 Functions

In TwinCAT Vision, functions are categorized based on their specific tasks. The following is an example of this separation:

- Images: group all operations that can be performed on images. It allows access to pixel information, width, and height, as well as the creation and display of images. Additionally, this category includes functionality for image analysis, segmentation and filtering.
- Container: useful for storing contours, enabling the addition and removal of elements. It also allows access to a single contour at specific position using indices.
- Contours: collections of 2D points crucial for object detection. This group includes checks on shape, point inclusion and geometrical features such as the center, area and perimeter.
- Code reading: facilitate reading and detection of OCR.

- Drawing: includes functions for visualizing detected features such as contours, points, lines and shapes.
- Advanced Functions: extend the basic operations.
- Miscellaneous

All TwinCAT Vision functions return an HRESULT. Then after the execution of a function, the resulting alphanumerical code can be examined to determine if the execution was successful or not.

Chapter 3: TwinCAT Code Implementation

In this chapter, we provide a detailed explanation of how to use the camera and how all the modules are interconnected to identify the centers of circles and rectangles.

3.1 Camera configuration

First, the camera is fixed above the conveyor to ensure a clear view of the objects passing on. To mitigate the impact of sun light on object recognition, the safety fence inside which the robot and the conveyor belt are placed remains closed when the application is in run mode, and artificial lighting inside the production cell is activated.

The VISION module in TwinCAT facilitate the addition of the camera, named Mako G 192B (5086) Allied Vision technology, through the following steps:

- Add new item:

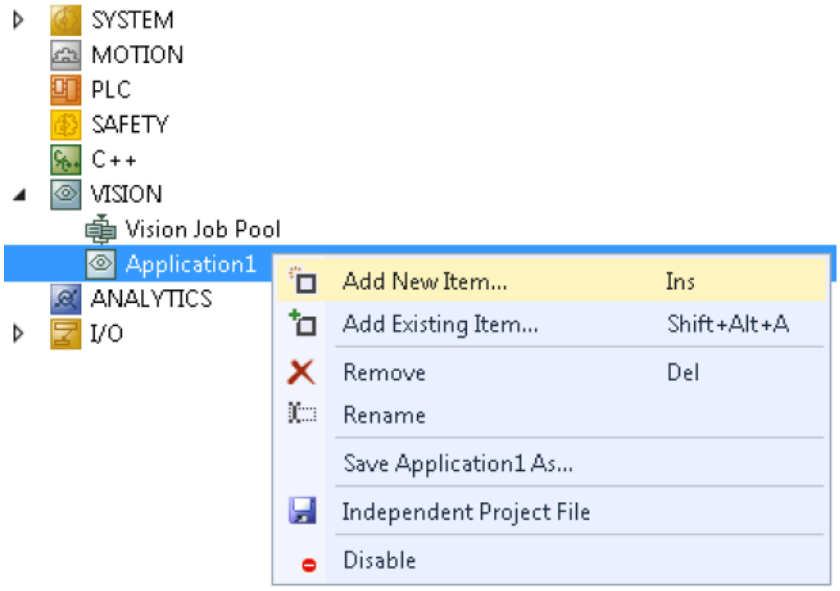

Figure 16: Add new item in VISION

- GigE Vision Camera: choose this option as indicated in the previous section (Figure 14).
- Connect the camera: enter the camera's IP address to establish the connection.

3.1.1 Connection camera and computer

The camera connects to computer through a GigE Vision interface. GigE Vision, an industrial image processing interface, utilizes a gigabit Ethernet communication protocol.

Figure 17: Computer linked to adapter, adaapter linked to camera

The computer links to an adapter via an EtherCAT connection, which in turn connects to the camera (Figure 17).

EtherCAT differs from standard Ethernet by offering in real time communication without latency, making it preferable for industrial applications.

The setup starts with the creation of a Vision node for the camera.

To access the network connection, first open the control panel of the computer. This action displays the state of Ethernet connection for the camera.

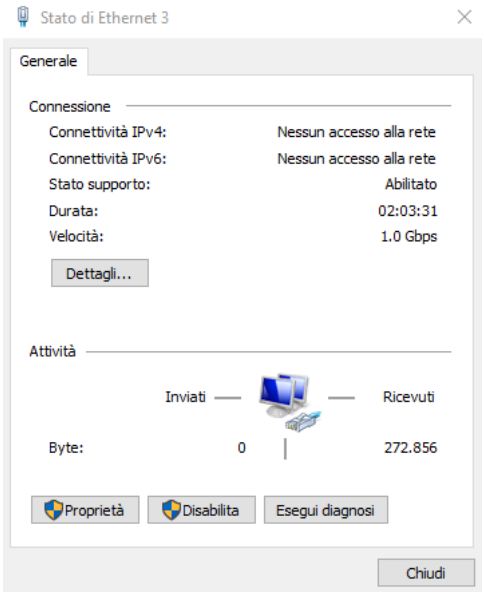

Figure 18: Ethernet

By selecting ‘Property’, in Figure 18, it is possible to set the IP address and the subnet mask to match the camera’s address such as in Figure 19.

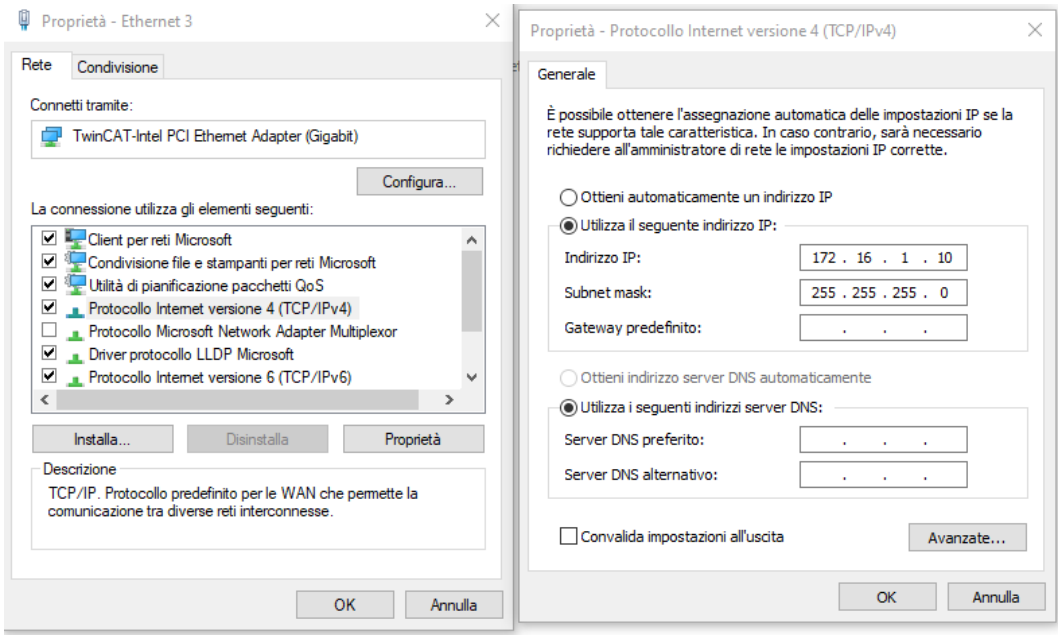

Figure 19: Property

After configuring these settings, return to the TwinCAT system and open the camera initialization assistant to verify the device’s correct connection; as displayed in the Figure 20, the IP of the Ethernet and the camer’sa IP Address are the identical. In this case, the model identified is Mako G-192B.

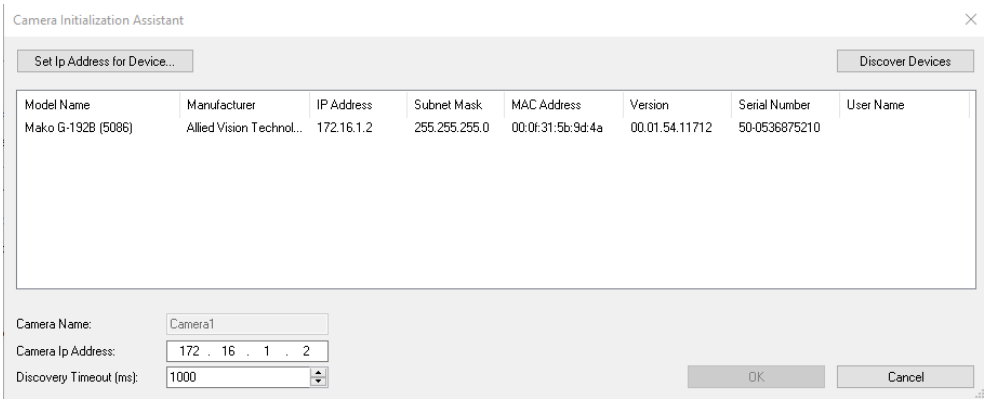

Figure 20: Discover Devices

In some instances, the connection with the camera may be lost. If this occurs, it is necessary a new ‘Discover Device’.

Automatically, a device that corresponds to the Ethernet Adapter used will be added in the I/O module (Figure 21). In this module, it is possible to check TcIoIpSettings by selecting

Parameter tab, where .IpAddress and .SubnetMask are defined. Then, we can enable the Manual Setting by changing the value in TRUE.

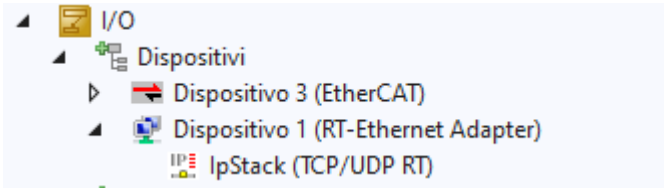

Figure 21: I/O

In I/O by opening 'Dispositivo 1' one can view the information of Ethernet 3, including IP and MAC address.

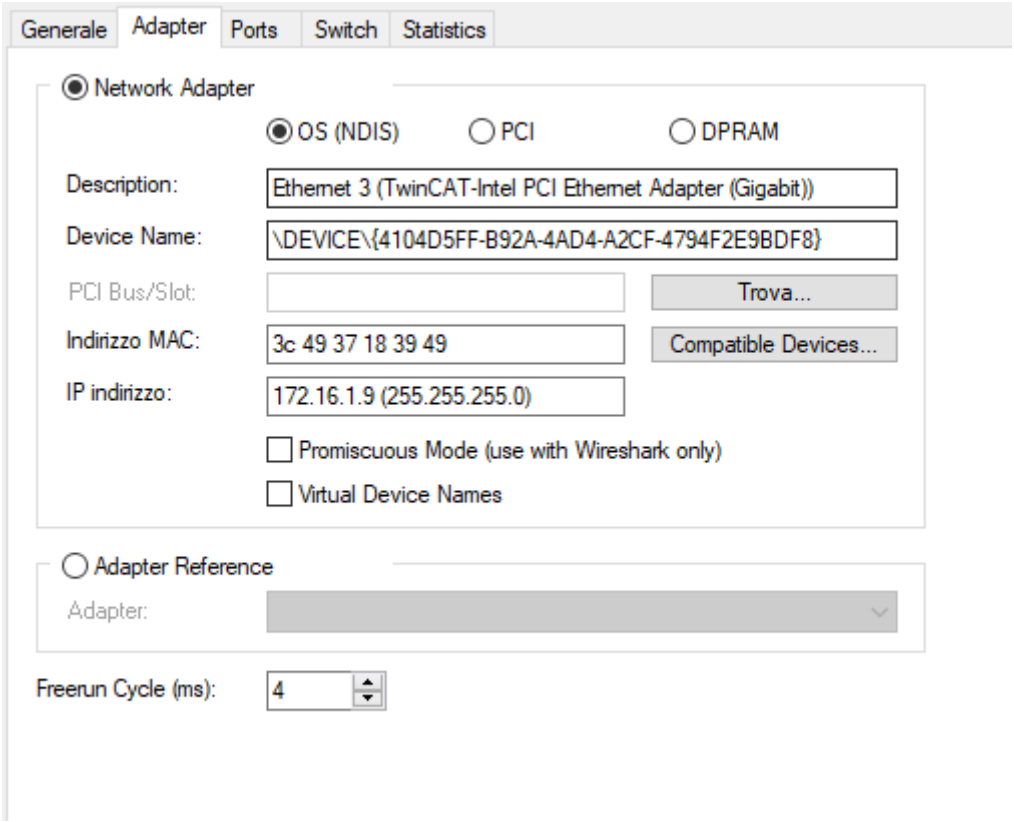

Figure 22: Ethernet informations

In this thesis, only one camera is considered for the detection, but in some cases it is useful to consider more cameras by adjusting in the parameters tab IpMaxReceivers and UdpMaxReceivers.

3.1.2 Camera features

The camera features a CMOS(Complementary Metal-Oxide Semiconductor) image sensor, which is designed to convert incoming light into a digital image using photon detectors

located on the surface of a semiconductor chip. This technology helps to achieve high image quality.

The resolution of the generated images is 800 x 600 pixels. Below is Table 1, resuming the main characteristics of the camera used.

Table 1: Camera Specifications

Specification	Details
Model	Mako G 192B (5086)
Sensor	Teledyne e2v EV76C570
Chroma	Monochrome
Sensor Technology	CMOS
Readout Method	Global Shutter
Sensor Format	1/1.8"
Sensor Diagonal	9.0 mm
Total Resolution	800 x 600
Number of Pixels	1.9 Megapixel
Pixel Size	4.5 µm x 4.5 µm
Frame Rate	60 fps
Shutter (Exposure Time)	14 µs - 0.9 s
Gain Control	0 dB - 24 dB
Interface	GigE Vision with Power over Ethernet (PoE)
Power Supply	Power over Ethernet (PoE), 12 - 24 VDC AUX
Power Consumption	2.4 W (PoE); 2.1 W (at 12 VDC)
Lens Mount	C-Mount

Dimensions	60.5 mm × 29.2 mm × 29.2 mm (including connectors)
Protection	IP30

The camera is equipped with two adjustment wheels that facilitate two types of settings:

- Aperture: regulate the amount of light entering the camera, larger aperture allows more light to enter, resulting in a brighter image.
- Focus: make the image clearer and more detailed by sharpening the focus of the image elements.

3.1.3 Camera Assistant

In this section, it is described how to make changes to the camera settings.

It is crucial to operate in Config mode, otherwise any modifications will not alter the state of the camera but will only affect the state defined in the project.

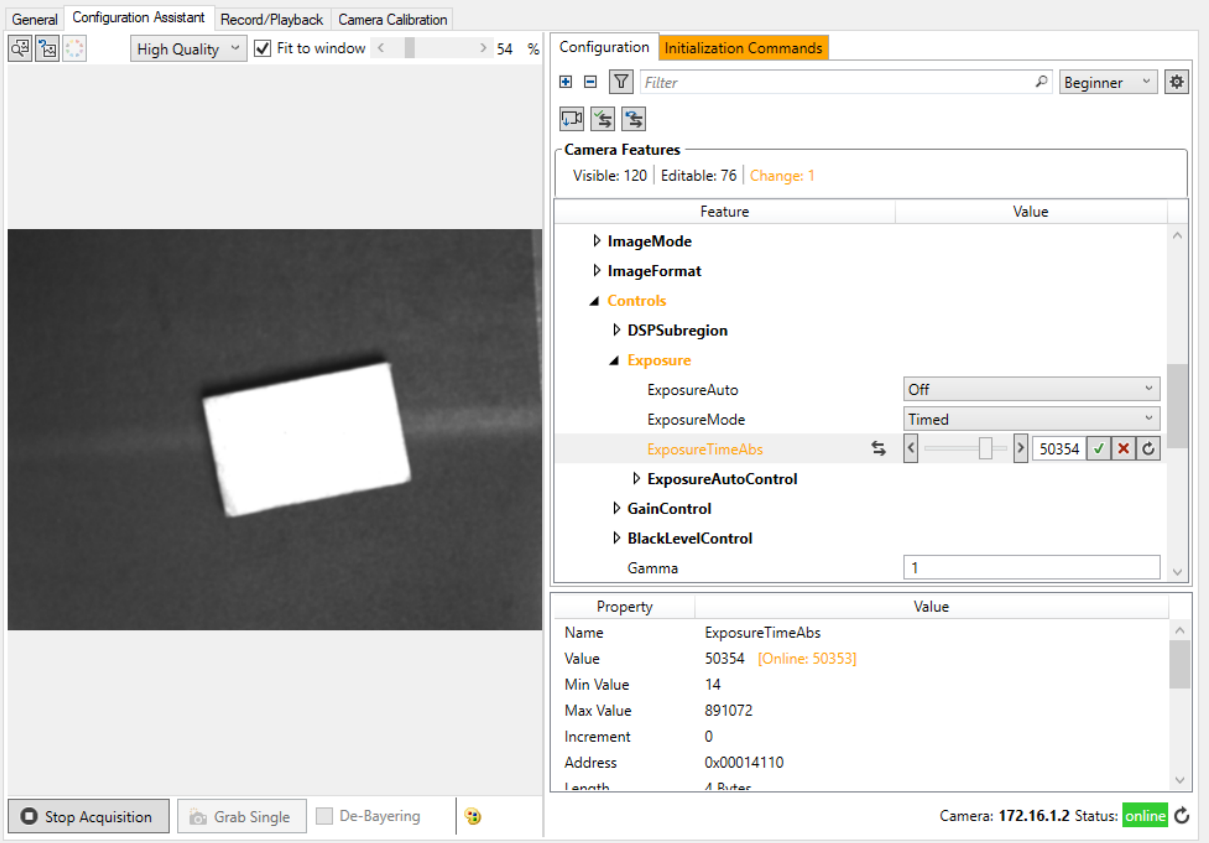

Figure 23: Camera Assistant Configuration

After making changes, click on the green checkmark to save everything, as shown in the Figure 23.

The absence of orange indicates that all the changes have been successfully saved.

Initially, the ‘Trigger Selector’ was set to ‘Continuous’, meaning the camera was in continuous acquisition mode. However, this setup is not feasible because the huge amount of data sent from the camera, slowed down the entire process.

Therefore, ‘Trigger Selector’ was changed to ‘AcquisitionStart’, see Figure 24. In practice, when an object passes through the photocell, the signal sends an activation to the camera. The camera proceeds through the states until reahes the TCVN_CS_START_ACQUISITION state and captures a single image, as indicated by the ‘AcquisitionMode’ beign set to ‘SingleFrame’. Alternatively, depending on the task, it is possible to select ‘MultiFrame’ and specify the number of images in ‘AcquisitionFrameCount’.

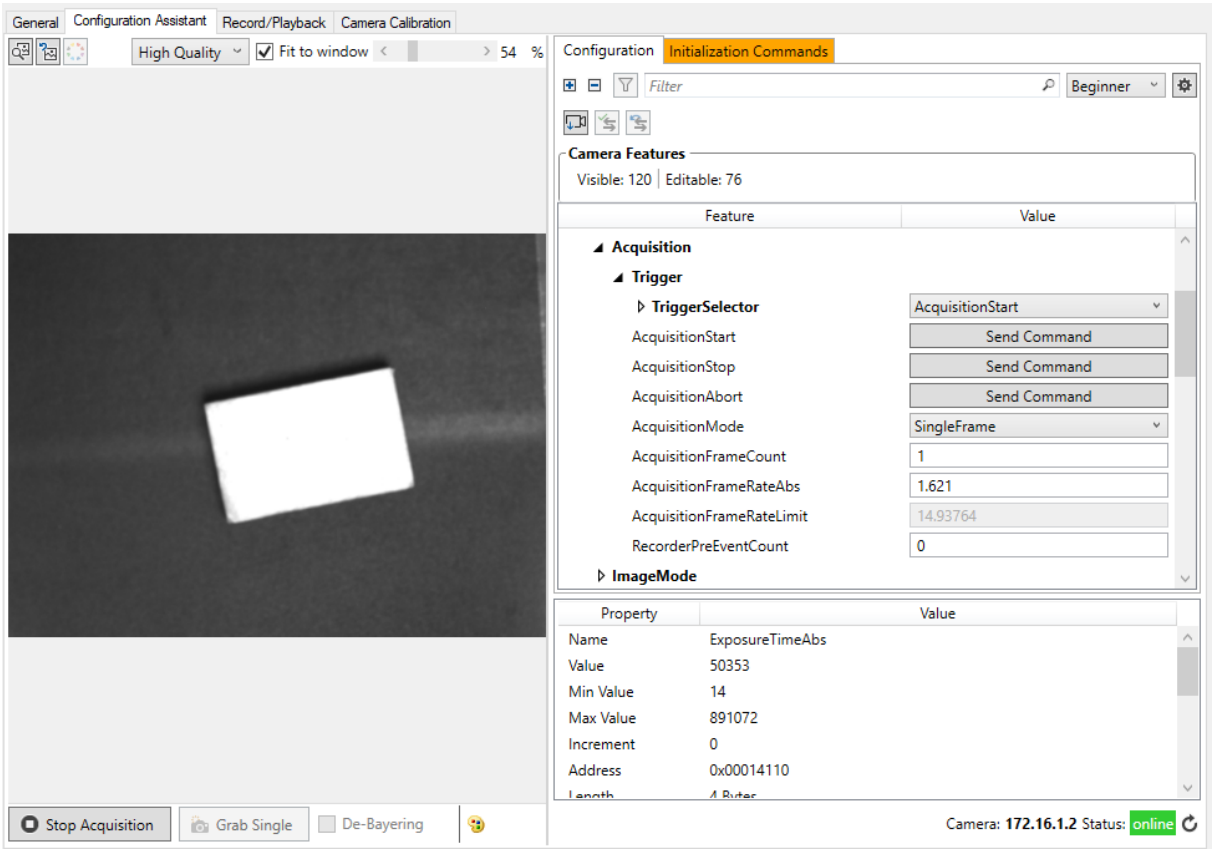

Figure 24: Camera Assistant, Trigger Selector

3.1.4 Camera Calibration

Calibration is performed using the Calibration Assistant within the TwinCAT Vision camera settings.

It is necessary to select an image with a shape easily recognizable for calibration purposes; in this instance, the chosen image features asymmetric circles.

The image used for calibration is provided directly in the TwinCAT Vision guide[22] along with the instructions for subsequent settings, as depicted in the Figure 25. It is loaded by clicking on ‘Load Images’.

Based on the image loaded, the following parameters are defined in the right hand section:

- Width: number of circles in a row.
- Height: number of circles in a column.
- X: distance between centers of two circles in horizontal direction.
- Y: distance between centers of two circles in vertical direction.
- Color inverted: by default, the system assumes dark circles on a light background. In this case, the colors are inverted.

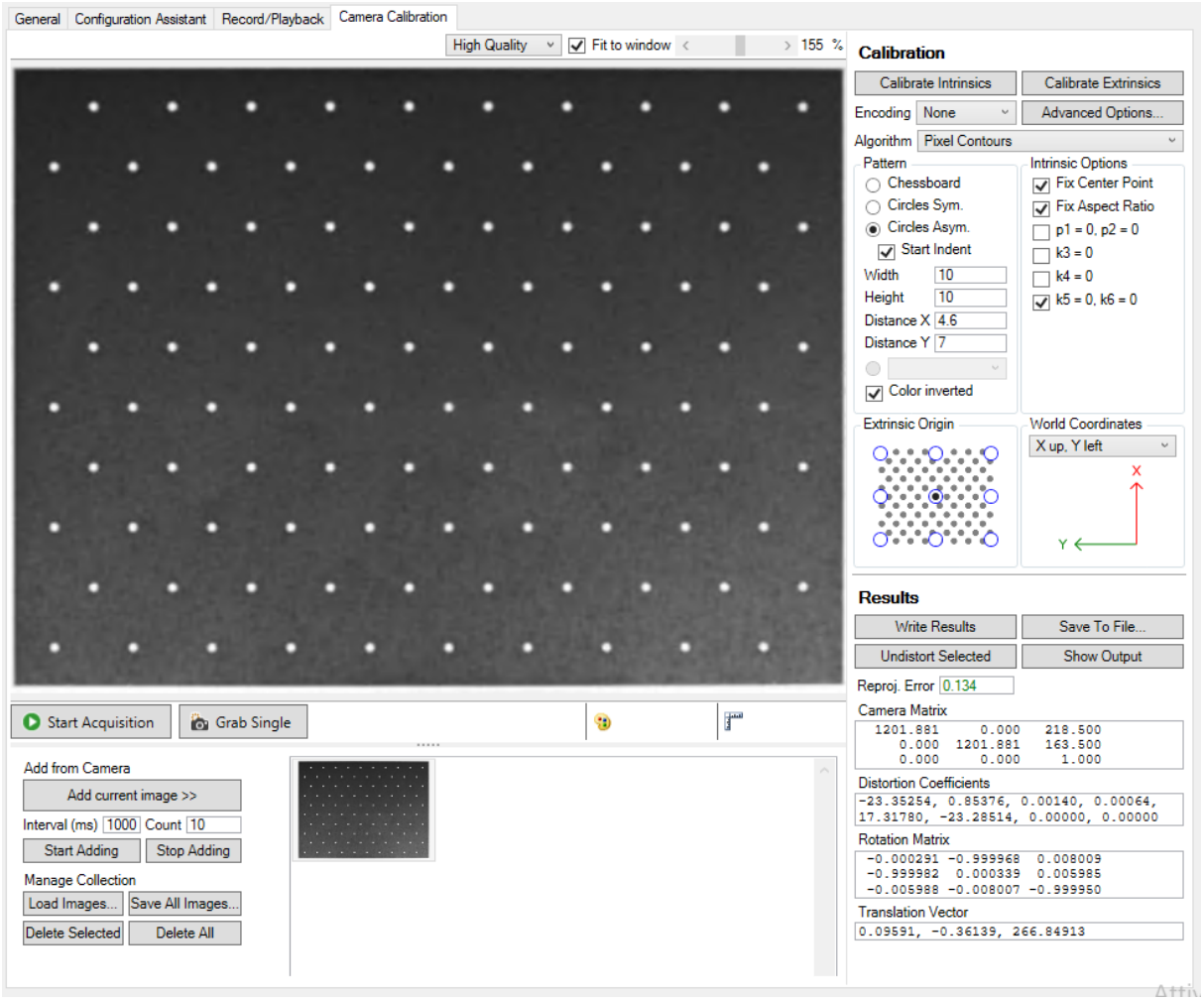

Figure 25: Camera Calibration, load image and calibrate intrinsic and extrinsic

In this specific context, the ‘World Coordinates’ coincides with the camera system coordinates (Figure 116). The camera system, as viewed from the desktop position is oriented such that the front part of the camera is rotated 90 degrees with respect to the robot system.

Given the camera's position, its X and Y axes align with those of the photocell, which facilitates future translations.

Once all the parameters are configured, the 'Calibrate Intrinsic' and 'Calibrate Extrinsic' buttons can be clicked to calculate:

- Intrinsic parameters of the camera, which depends on how it captures the images. These generally includes the Focal Length, Aperture, Field of View (FoV) and resolution. They are important because allow the system to interpret what the camera sees correctly in terms of real dimensions. This results in the camera matrix and distortion coefficients being calculated.
- Extrinsic parameters of the camera, which depends on its location and its orientation, and are calculated using image information alone. These parameters allow the system to localize and interact with objects in the three-dimensional space. This results in the rotation matrix and translation vector being calculated.

The function 'Write Results' can be used to save calibration data such as Camera Matrix, Distortion Coefficients, Rotation Matrix and Translation Vector. These values are useful for transforming coordinates from pixels to world coordinate in the subsequent sections.

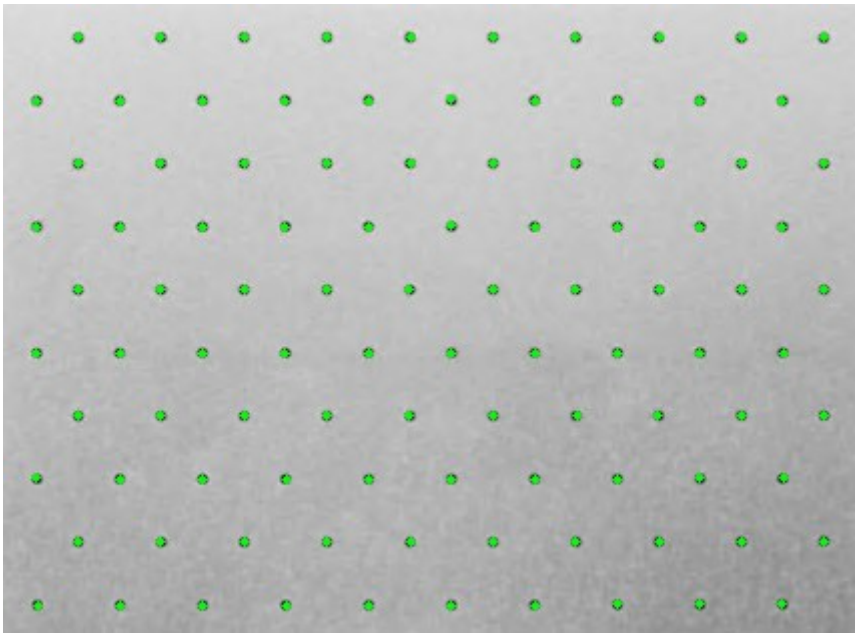

Figure 26: Camera Calibration result

When the image in Figure 26 is generated, the calibration is done.

3.1.5 PLC connection

The camera and PLC are connected through 'Symbol initialization' in the PLC_Vision Istance, see Figure 28, by selecting the specific camera used for the specific task via MAIN.fbCamera.oidITcVnImageProvider, as detailed in the Figure 27.

It works in the same way for File Source objects.

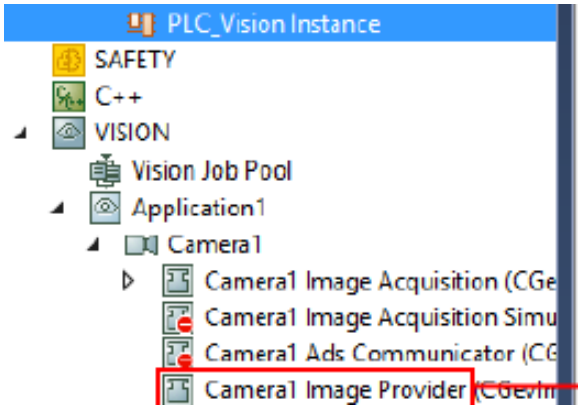

Figure 27: Image Provider

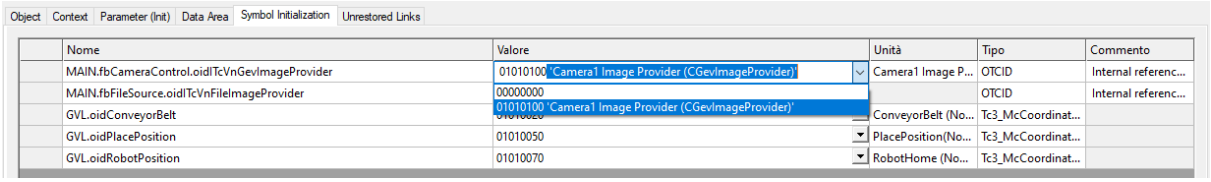

Figure 28: Symbol Initialization

3.1.6 Acquisition and trigger

The access to images is done using a state machine, then there are several functions blocks used for querying an image interface of a Vision device.

FB_VN_SimpleCameraControl, one of the previous mentioned functions blocks, provide the functionality to control the camera.

Considering the diagram in Figure 29, the main states of the camera control state machine are highlighted in blue:

- TCVN_CS_ERROR
- TCVN_CS_INITIAL
- TCVN_CS_OPENED
- TCVN_CS_ACQUIRING

The states in gray color are the transition states, representing intermediate processes:

- TCVN_CS_OPENING

- TCVN_CS_START_ACQUISITION
- TCVN_CS_STOP_ACQUISITION
- TCVN_CS_TRIGGERING

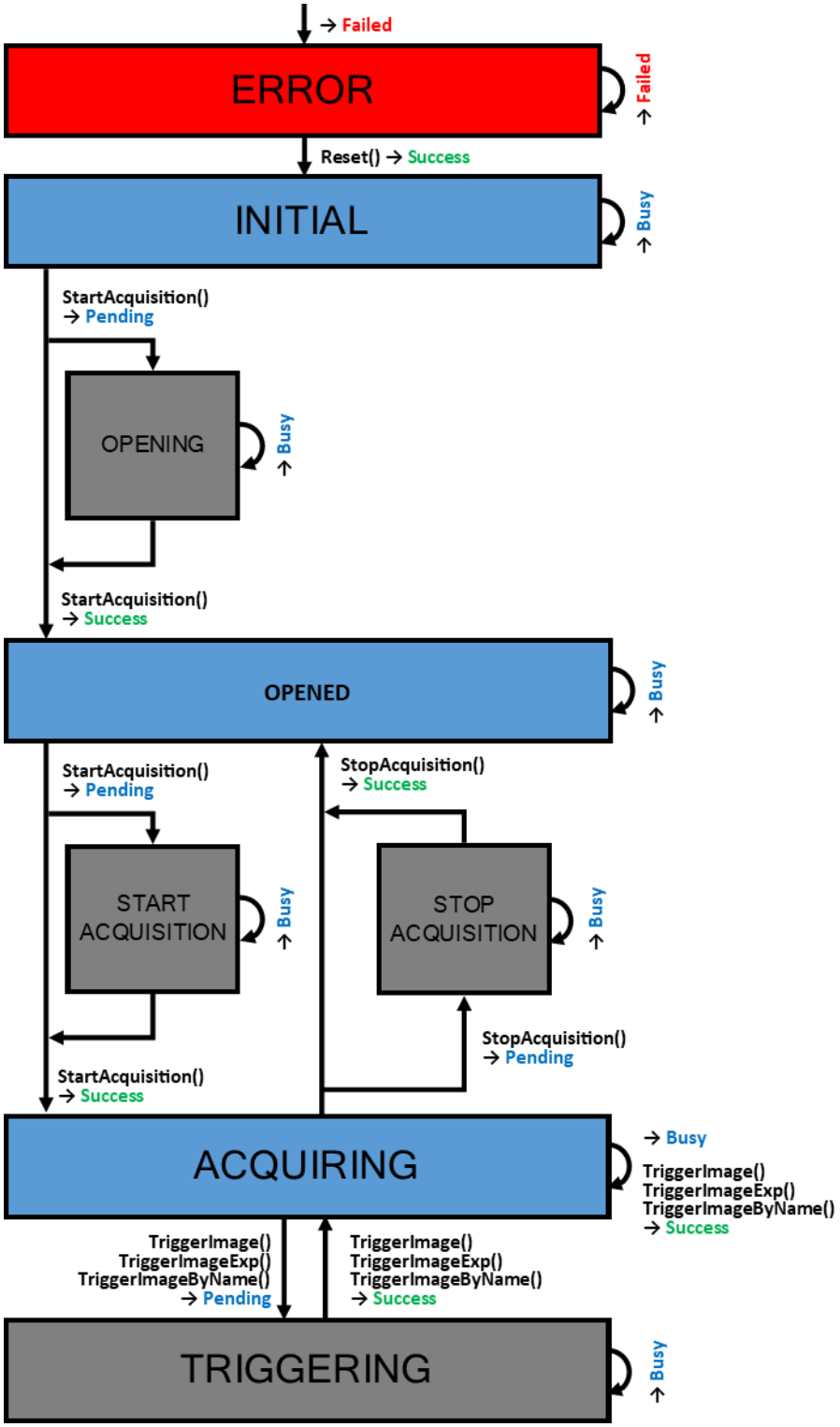

Figure 29: Camera state machine

The developed code, Figure 30/31, adheres to this specified diagram for image acquisition. Here's a structured overview of the process.

The process starts checking the camera's state using fbCamerControl.GetState(). The state returned represents the camera's current state as represented in the operational diagram when the program is in running.

Using SimpleCameraControl is done the calibration data retrieval, previously calculated, using Get method:

- GetCameraMatrix
- GetDistortionCoefficients
- GetRotationMatrix
- GetTranslationVector

This data retrieval is carried out for the future transformation (F_VN_TransformCoordinatesImageToWorld_Point) of the center coordinates from pixel points into 'World Coordinate' using intrinsic and extrinsic parameters as input. The calibration and the 'Write Results' are performed only once, unless the camera is moved to another position; however, the data are retrieved each time the code runs, because without this retrieval, it would not be possible to use them in the code.

```
140  eCameraState :=  fbCameraControl.GetState();
141
142  hr := fbCameraControl.GetCameraMatrix(aCameraMatrix);
143  hr := fbCameraControl.GetDistortionCoefficients(aDistortionCoefficients);
144  hr := fbCameraControl.GetRotationMatrix(aRotationMatrix);
145  hr := fbCameraControl.GetTranslationVector(aTranslationVector);
146
147  // CameraControl is in error state, so try to reset the camera connection
148  IF eCameraState = TCVN_CS_ERROR THEN
149      hr :=  fbCameraControl.Reset();
150
151  // Camera not yet initialized
152  ELSIF eCameraState < TCVN_CS_INITIALIZED THEN
153      hr :=  fbCameraControl.InitializeCamera();
154
155  // Camera not yet opened
156  //OPENING
157  ELSIF eCameraState < TCVN_CS_OPENED THEN
158      hr := fbCameraControl.OpenCamera();
159
160  ELSIF eCameraState = TCVN_CS_OPENED THEN
161      IF bProximity = 0 AND nNewImageCounter < 1 THEN
162              //TLParamsLocked := TRUE;
163              hr := fbCameraControl.StartAcquisition();
164      END_IF
165
166  ELSIF eCameraState = TCVN_CS_STARTACQUISITION THEN
167      hr := fbCameraControl.StartAcquisition();
168
```

Figure 30: ERROR, INITIALIZED, OPENED, START_ACQUISITION

```
// Camera streaming
// images can be captured and transferred
ELSIF eCameraState = TCVN_CS_ACQUIRING THEN
            hr := fbCameraControl.GetCurrentImage(ipImageIn);
            // hr := F_VN_SetRoi(25, 33, 700, 520, ipImageIn, hr);
            hr := F_VN_SetRoi(0, 0, 795, 410, ipImageIn, hr);
            hr := F_VN_CopyIntoDisplayableImage(ipImageIn, ipImageInDisp, hr);

            IF SUCCEEDED(hr) AND ipImageIn <> 0 THEN
                nNewImageCounter := nNewImageCounter + 1;
                hr := fbCameraControl.StopAcquisition();

                //TLParamsLocked := FALSE;
                // Place to call vision algorithms
                //hr := F_VN_CopyIntoDisplayableImage(ipImageIn, ipImageInDisp, hr);

                ObjectDetection();
                //MatchTemplate();
                //CannyEdge();
                //MachineLearning();

            END_IF

ELSIF eCameraState = TCVN_CS_STOPACQUISITION THEN
    hr := fbCameraControl.StopAcquisition();

END_IF

IF bProximity = 2 THEN
        nNewImageCounter := 0;
END_IF
```

Figure 31: ACQUIRING, STOP_ACQUISITION

The camera is then initialized with fbCameraControl.InitializeCamera().

After that, the state transitions to the OPENED state with fbCameraControl.OpenCamera(). In the OPENED state, bProximity is managed, which is a bit employed to control the photocell. When the bit bProximity is equal to 0, an object is dectected; this triggers a shift from the OPENED state to the transition state STARTACQUISITION.

Once the camera enters the previously mentioned state, it moves to ACQUIRING state where the image is actually captured.

fbCameraControl.GetCurrentImage(ipImageIn) retrieves the current available image from the camera.

To capture only the specific part of the environment through which the object will pass through, the image size is reduced from 800x600 to 795x410 with F_VN_SetRoi().

If the captured image is correctly received, the acquisition ends with the transition state TCVN_CS_STOPACQUISITION. After the stop state, the process transitions back to the OPENED state. This process is repeated for each detected object.

Meanwhile, as this process continues with other acquisitions, ipImageIn is ready for the application of different algorithms and filters; in this case the filters are applied by calling ObjectDetection() function.

3.2 Circles and Rectangles centers recognition

In computer vision, object recognition can be carried out using multiple filters. This chapter will explain all the filters applied to images to achieve the final objective: determining the coordinates of the centers of circles and rectangles, which are then passed to the robot for pick and place operations.

3.2.1 Filters application

The starting point for object recognition is the application of filters (Figure 32) to the image to remove noise.

A practical technique involves using a threshold, which can be either BINARY or not BINARY depending on the color of the plane. In this project, the background consists of a black conveyor belt, which provides the first distinction between the background and the object's color.

By selecting a pixel in the left corner of the image, for instance the pixel at coordinates 50 in x and 50 in y, if this pixel is black, then a BINARY threshold is applied; otherwise, a BINARY_INV threshold is used. This operation checks a bit named bInvert.

A BINARY threshold turns all pixels that are lighter than a certain threshold value completely white, and all pixels that are darker than this threshold completely black. Conversely, a BINARY_INV threshold does the opposite, it turns pixels that are lighter than the threshold completely black and those that are darker completely white.

```
// pick the pixel value in the left corner of the image and check if it is black
hr := F_VN_GetPixel(ipImageIn, aPixelValue, 50, 50, hr);
bInvert := SUCCEEDED(hr) AND_THEN aPixelValue[0] >= 128;

// threshold useful to distinguish objects from background
IF bInvert THEN
    stParams.eThresholdType := TCVN_TT_BINARY;
ELSE
    stParams.eThresholdType := TCVN_TT_BINARY_INV;
END_IF

// set a threshold to consider white object
hr := F_VN_Threshold(ipImageIn, ipImageWork, 170, 255, eThresholdType, hr);

//FILTERS APPLICATION

//blur the image to remove unwanted detail
hr := F_VN_GaussianFilter(ipImageWork, ipImageWork, 9, 9 ,hr);
//hr := F_VN_MedianFilter( ipImageWork, ipImageWork, 7, hr);

//contrast
hr := F_VN_HistogramEqualization(ipImageWork, ipImageWork, hr);

hr := F_VN_CreateStructuringElement(ipElement, TCVN_SES_RECTANGLE, 3, 3, hr);
//remotion small objects and irregularity
hr := F_VN_MorphologicalOperator(ipImageWork, ipImageWork, TCVN_MO_OPENING, ipElement,hr);
```

Figure 32: Threshold, Gaussian, Median, HistogramEqualization, MorphologicalOperator

Because the objects considered are white or some other light color, we are primarily interested in the brightest pixels; therefore, the maximum value for the threshold will be 255 (white). By applying the threshold using F_VN_Threshold(ipImageIn, ipImageWork, 170, 255, eThresholdType, hr), only the brightness objects will be considered, as the range between 170 and 255.

```
hr := F_VN_SobelFilter(
    ipSrcImage  :=  ipImageWork,
    ipDestImage :=  ipImageWork,
    eDestDepth  :=  TCVN_ET_USINT,
    nXOrder     :=  1,
    nYOrder     :=  1,
    hrPrev      :=  hr,
);



hr := F_VN_LaplacianFilterExp(
         ipSrcImage   := ipImageWork,
         ipDestImage  := ipImageWork,
         eDestDepth   := eLaplace_DestDepth,
         nKernelSize  := nLaplace_KernelSize,
         fScale       := fLaplace_Scale,
         fDelta       := fLaplace_Delta,
         eBorderType  := eLaplace_BorderExtra,
         hrPrev       := hr);


hr := F_VN_BilateralFilter(
        ipSrcImage      := ipImageWork,
        ipDestImage     := ipImageBilateral,
        nDiameter       := nBilateral_Diameter,
        fSigmaColor     := fBilateral_SigmaColor,
        fSigmaSpace     := fBilateral_SigmaSpace,
        hrPrev          := hr);

hr := F_VN_CopyIntoDisplayableImage(ipImageBilateral, ipImageBilateralDisp, hr);
```

Figure 33: Sobel, Laplacian, Bilateral

The resulting image will be filtered in different ways, as demostrated in the code displayed in Figure, and only the best filters will be considered in the final version of the code. Generally these functions share common parameters: ipSrcImage, the image captured from camera, and ipDestImage, the filtered image produced as output.

Below is a summary of all TwinCAT functions employed:

- F_VN_GaussianFilter: applies Gaussian filter to smooth the image. Kernel size, which can have different old sizes such as 1,3,5,7, is specified by nFilterWidth and nFilterHeight. This filter removes smaller details preserving the larger ones.

Figure 34: GaussianFilter

- F_VN_MedianFilter: applies a median filter to the image. nFilterSize specifies the size of the matrix considered. By taking into account the values present in the matrix, this filters calculates the median value. For example, consider the sequence 7 8 9 10 10 11 13 13 155, the median value would be 10.

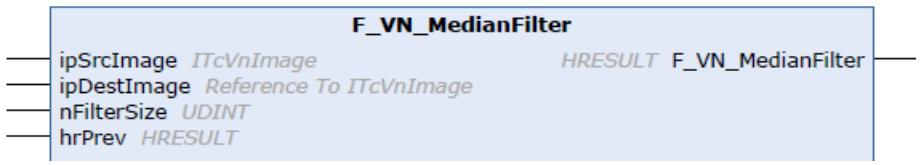

Figure 35: MedianFilter

- F_VN_MorphologicalOperator: applies a morphological operator to the the input image. The operators are enumerated in ETcVnMorphologicalOperator, with the main operators including:
    - EROSION, separates weakly connected components.
    - DILATION, merges close unconnected components.
    - OPENING, eliminate thin protrusions without reducing the size of the element.
    - CLOSING, fuse narrow breaks without increasing the size of the element.

  After various trials, the operator more compliant with this thesis is OPENING, as it effectively removes small objects and irregularity.

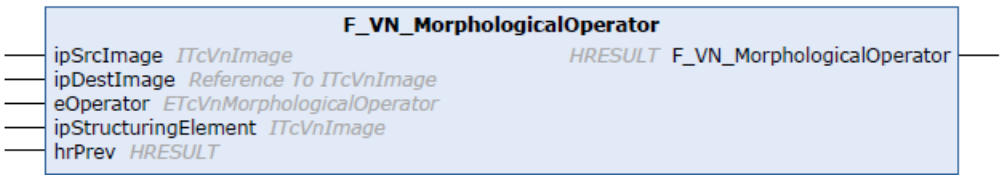

Figure 36: MorphologicalOperator

- F_VN_HistogramEqualization: is used to normalize brightness and improve the contrast of the image. An histogram is a graphical representation of data along an axis.

Histogram equalization employs a transformation that flattens the histogram of image in input. The flatter the histogram, the more enhanced the contrast in the image.

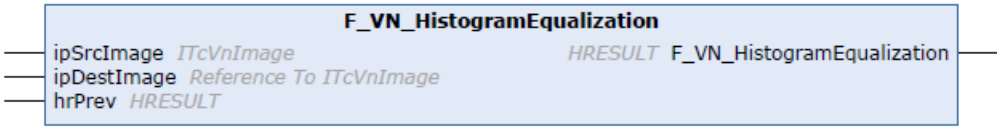

Figure 37: HistogramEqualization

- F_VN_SobelFilter: is used to detect edges by approximating the nth derivative of the transitions in adjacent pixel intensities. nXOrder an nYOrder define the derivatives in the X and the Y directions, respectively, that are used for creating filter. Depending on these settings, the filter can emphasize vertical edges (X) or horizontal edges (Y).

Figure 38: SobelFilter

- F_VN_LaplacianFIlterExp: is also used for the edge detection. It uses a kernel that can vary in size. The Exp suffix indicates an evolute version of LaplacianFilter. Unlike other edge related filter, this one includes the border extrapolation with eBorderType and a scaling factor fScale.

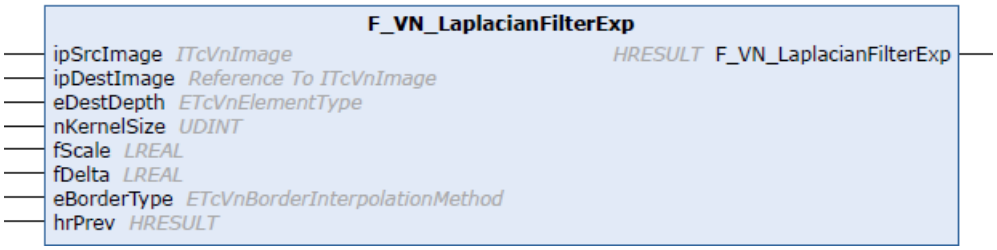

Figure 39: LaplacianFilter

- F_VN_BilateralFilter: smooths the image while preserving the edges. nDiameter specifies the diameter of the pixel neighborhood that is used. The parameters fSigmaColor and fSigmaSpace should be adjusted to obtain a good quality image.

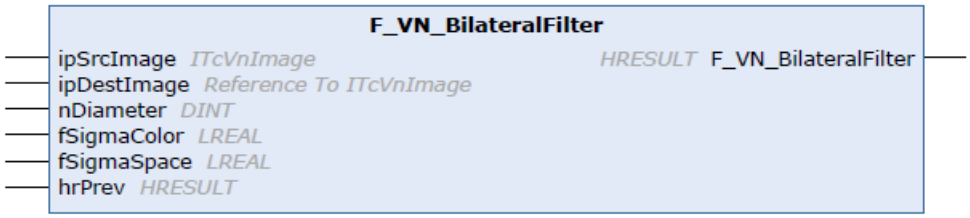

Figure 40: BilateralFilter

3.2.2 Object Detection Implementation

Object recognition was accomplished using FindContours and DetectBlobs. DetectBlobs, thanks to the stParams, is more efficient as it allows filtering based on specific characteristics, such as a range of circularity.

It's important to consider how the image can be displayed:

- With F_VN_CopyIntoDisplayableImage() if the image will be used in subsequent operations.

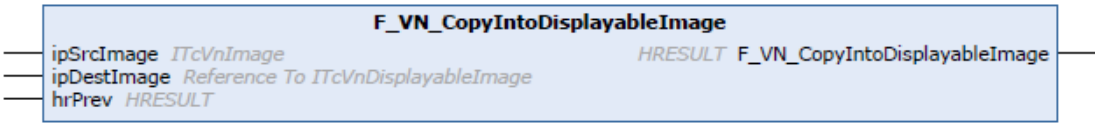

Figure 41:CopyIntoDisplayableImage

- With F_VN_TransformIntoDisplayableImage() if the image will no longer be used for other operations after this call.

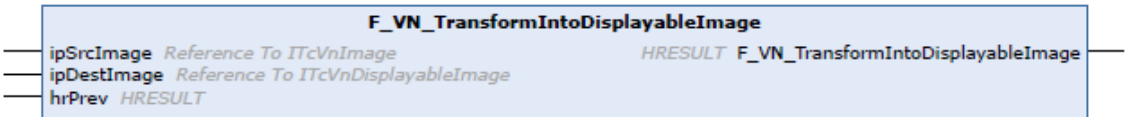

Figure 42:TransformIntoDisplayableImage

Below is the implemented code along with the respective functions. The main concept of this implementation is the 'blob', which is a group of connected pixels that share some common property.

```
63  //BLOB DETECTION
64  stBlobParams.bFilterByArea          := TRUE;
65  stBlobParams.fMinArea               := 100;
66  stBlobParams.fMaxArea               := 2000000;
67
68  //stBlobParams.bFilterByCircularity  := TRUE;
69  stBlobParams.fMinCircularity        := 0.80;
70
71
72  hr := F_VN_DetectBlobs(ipSrcImage      := ipImageWork,
73                         ipBlobContours  := ipContoursR,
74                         stParams        := stBlobParams,
75                         hrPrev          := hr);
76
77  // number of contour founded
78  hr := F_VN_GetNumberOfElements(ipContoursR, ipNumber, hr);
79
80  // In order to draw the contours in blue convert the image in RGB
81  hr := F_VN_ConvertColorSpace(ipImageWork, ipImageRes, TCVN_CST_GRAY_TO_RGB, hr);
82
83
84  //check that the contours are all detected
85  IF ipNumber > 0 THEN
86      FOR i:=0 TO (ipNumber-1) BY 1 DO
87          hr := F_VN_GetAt_ITcVnContainer(ipContoursR, ipContourR, i, hr);
88
89          //Calculate circularity of the contours and comparate with min circularity defined for the blobs
90          hr := F_VN_ContourCircularity(ipContourR, fCircularity, hr);
91
92          // if the circularity is less than 80 it is a rectangle
93          IF stBlobParams.fMinCircularity  > fCircularity THEN
94
95              // approximate contour to a simplified polygon and draw
96              hr := F_VN_ApproximatePolygon(ipContourR, ipContourApprox, 0, TRUE, hr);
97              hr := F_VN_DrawContours(
98                          ipContours      :=  ipContourApprox,
99                          nContourIndex   :=  -1,
100                         ipDestImage     :=  ipImageRes,
101                         aColor          :=  aColorGreen,
102                         nThickness      :=  5,
103                         hrPrev          :=  hr
104             );
```

Figure 43: ObjectDetection() first part, Blob detection, check contours and circularity, approximate to polygon and draw contours

The parameters for blob detection are defined in the variables as TcVnParamsBlobDetection, see in Figure 43 at line 64, 65, 66, 69:

- stBlobParams.bFilterByArea is used to filter blobs within a specific area range; in this case stBlobParams.fMinArea := 100 and stBlobParams.fMaxArea := 2000000.
- stBlobParams.fMinCircularity := 0.80 ensures that only circles are identified.

The function F_VN_DetectBlobs(), Figure 44, is used for the detection. It applies a threshold and a contour search (via F_VN_FindContours) along with options for filtering the found contours. ipBlobContours is a container where all the contours are stored for further filtering.

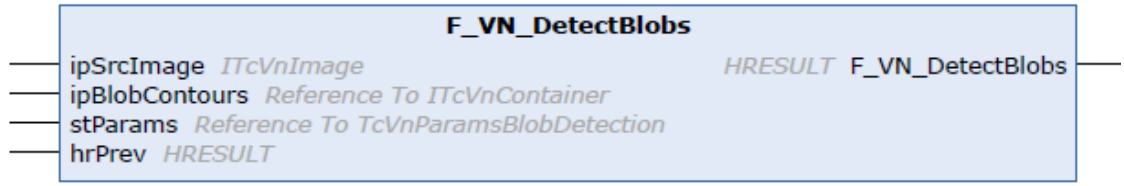

Figure 44: DetectBlobs in the image

With F_VN_GetNumberOfElements(), we obtain the number of contours in the container, nNumberOfElements. This function allows us to verify that all the contours captured by the camera in the image have been identified.

Figure 45: GetNumberOfElements in the container

F_VN_GetAt_ITcVnContainer() is useful for the extracting one contour at a time.

Figure 46: GetAt_ITcVnContainer, get one element

For each contour found using F_VN_ContourCircularity(ipContourR, fCircularity, hr), there is the possibility to determine the circularity.

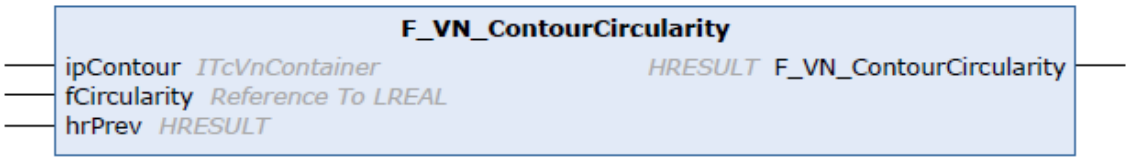

Figure 47: Circularity of each contour

Using circularity, it is possible to distinguish between circles and rectangles in order to identify the contours and their respective centers differently based on their geometric shape. Then, if the circularity of the object (stBlobParams) is less than the predefined value of circularity (fCircularity = 0.80), it indicates that the contour under consideration is a rectangle. Therefore the condition should be: IF stBlobParams.fMinCircularity < fCircularity THEN.

3.2.3 Rectangles

If the circularity is less then 0.80, the object is a rectangle.

To draw the contour in detailed manner, with F_VN_DrawContours, an approximation to a polygon is applied with F_VN_ApproximatePolygon. bClosed is set to TRUE because the contours are closed.

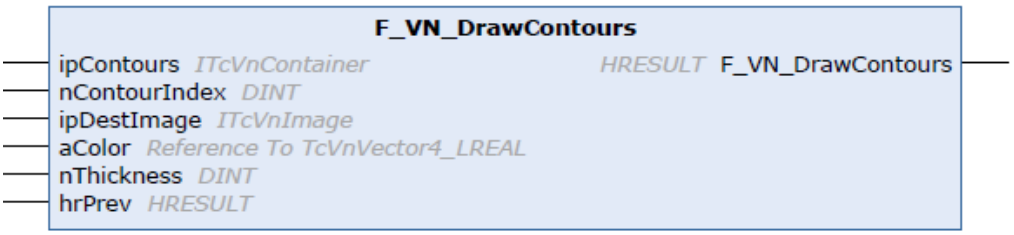

Figure 48: Draw found contour

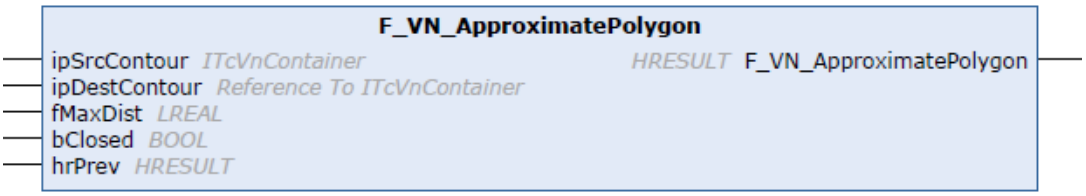

Figure 49: Approximation

```
106             //get the orientation of the contour
107             hr:= F_VN_ContourOrientation(ipContourR, stOrientation, hr);
108
109             // use the extreme point, draw diagonal lines and found the center
110             // foind 4 points of the rectangle
111             hr := F_VN_ContourExtremePoint(ipContourApprox, TCVN_EPD_TOP_LEFT, aExtremePointTL, hr);
112             hr := F_VN_ContourExtremePoint(ipContourApprox, TCVN_EPD_TOP_RIGHT, aExtremePointTR, hr);
113             hr := F_VN_ContourExtremePoint(ipContourApprox, TCVN_EPD_BOTTOM_LEFT, aExtremePointBL, hr);
114             hr := F_VN_ContourExtremePoint(ipContourApprox, TCVN_EPD_BOTTOM_RIGHT, aExtremePointBR, hr);
115
116             //in order to draw with color blue transform the image in RGb
117             // draw it based on the points of the prvious function
118             hr := F_VN_DrawPointExp(TO_UDINT(aExtremePointTL[0]),
119                                     TO_UDINT(aExtremePointTL[1]),
120                                     ipImageRes,
121                                     TCVN_DS_X,
122                                     aColor,
123                                     10,
124                                     2,
125                                     TCVN_LT_ANTIALIASED,
126                                     hr);
127
128
129             hr := F_VN_DrawPointExp(TO_UDINT(aExtremePointBR[0]),
130                                     TO_UDINT(aExtremePointBR[1]),
131                                     ipImageRes,
132                                     TCVN_DS_X,
133                                     aColor,
134                                     10,
135                                     2,
136                                     TCVN_LT_ANTIALIASED,
137                                     hr);
138
139             // draw line btw top left corner and bottom right corner
140             hr := F_VN_DrawLine(TO_UDINT(aExtremePointTL[0]),
141                                     TO_UDINT(aExtremePointTL[1]),
142                                     TO_UDINT(aExtremePointBR[0]),
143                                     TO_UDINT(aExtremePointBR[1]),
144                                     ipImageRes,
145                                     aColor,
146                                     2,
147                                     hr);
```

Figure 50: ObjectDetection, second part

Once the contour is highlighted, it is possible to find the extreme points (Figure 50) with F_VN_ContourExtremePoint: TOP_RIGHT, TOP_LEFT, BOTTOM_RIGHT, BOTTOM_LEFT of the contours.

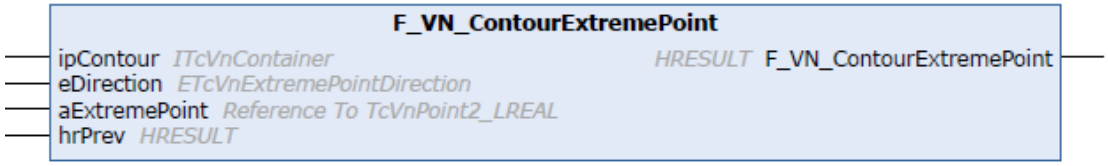

Figure 51: Extreme point of the contour

When the 4 points are retrieved, the next step is to draw lines from TOP_LEFT to BOTTOM_RIGHT and from TOP_RIGHT to BOTTOM_LEFT.

To verify the precision of the retrieved points they are visualized as follow:

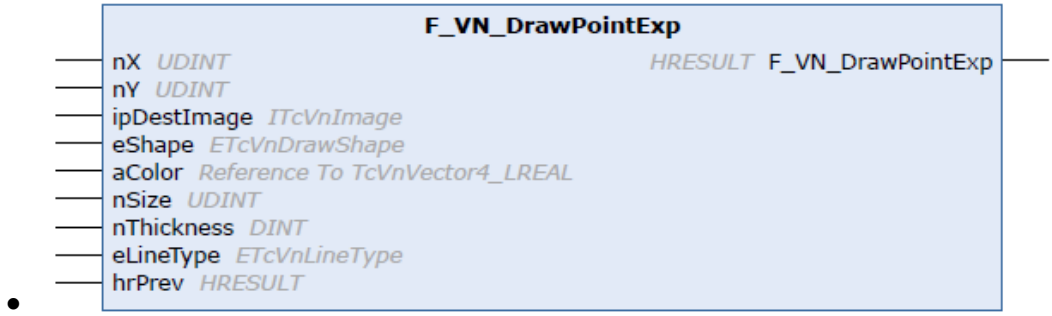

Figure 52: DrawPointExp

Figure 53: DrawLine

The contours, lines and points are ultimately visualized in green. Since the image captured form the camera is in grayscale, a conversion in RGB color space is necessary to visualize the colors. This is achieved with F_VN_ConvertColorSpace(). The transformation used in this case is specified by eTransform as TCVN_CST_GRAY_TO_RGB.

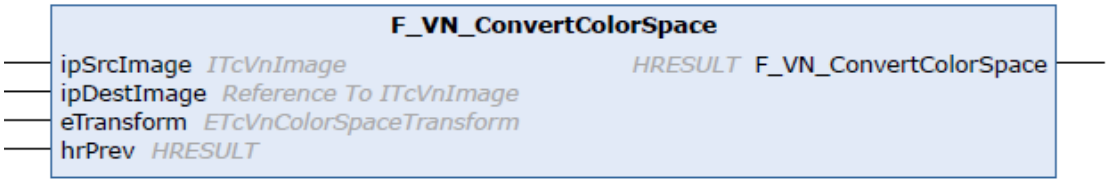

Figure 54: Conversion in another color space

Once the lines are drawn, it is possible to find their center by calculating the midpoint of each lines; this point will serve as the center: centerX and centerY in the code in Figure 55/56.

```
149             hr := F_VN_DrawPointExp(TO_UDINT(aExtremePointTR[0]),
150                                     TO_UDINT(aExtremePointTR[1]),
151                                     ipImageRes,
152                                     TCVN_DS_X,
153                                     aColor,
154                                     10,
155                                     2,
156                                     TCVN_LT_ANTIALIASED,
157                                     hr);
158
159             hr := F_VN_DrawPointExp(TO_UDINT(aExtremePointBL[0]),
160                                     TO_UDINT(aExtremePointBL[1]),
161                                     ipImageRes,
162                                     TCVN_DS_X,
163                                     aColor,
164                                     10,
165                                     2,
166                                     TCVN_LT_ANTIALIASED,
167                                     hr);
168
169
170             // draw line btw top right corner and bottom left corner
171             hr := F_VN_DrawLine(TO_UDINT(aExtremePointTR[0]),
172                                     TO_UDINT(aExtremePointTR[1]),
173                                     TO_UDINT(aExtremePointBL[0]),
174                                     TO_UDINT(aExtremePointBL[1]),
175                                     ipImageRes,
176                                     aColor,
177                                     2,
178                                     hr);
179
180             //found the center by using lines
181             centerX := (aExtremePointTL[0] + aExtremePointBR[0])/2;
182             centerY := (aExtremePointTL[1] + aExtremePointBR[1])/2;
183
184             //draw center of the rectangle
185             hr := F_VN_DrawPointExp(TO_UDINT(centerX),
186                                     TO_UDINT(centerY),
187                                     ipImageRes,
188                                     TCVN_DS_X,
189                                     aColor,
190                                     10,
191                                     2,
192                                     TCVN_LT_ANTIALIASED,
193                                     hr);
```

Figure 55: ObjectDetection, third part

```
194             center[0] := centerX;
195             center[1] := centerY;
196
197             //  transfrom in wordl frame point starting from image pixels
198             hr := F_VN_TransformCoordinatesImageToWorld_Point(center,
199                                                               centerDest,
200                                                               aCameraMatrix,
201                                                               aDistortionCoefficients,
202                                                               aRotationMatrix,
203                                                               aTranslationVector,
204                                                               fZ,
205                                                               hr);
206
207             sTextR := TO_STRING("Rectangle");
208             hr := F_VN_PutText(sTextR, IpImageRes, 50, 50, TCVN_FT_HERSHEY_COMPLEX_ITALIC, 1, aColorWhite, hr);
209
210             hr := F_VN_CopyIntoDisplayableImage(ipImageRes, ipImageResDisp, hr);
```

Figure 56: ObjectDetection, fourth part

The coordinates of the center should then be transformed from pixel format into World points, with F_VN_TransformCoordinatesImageToWorld_Point, where the reference system is based on the camera's position.

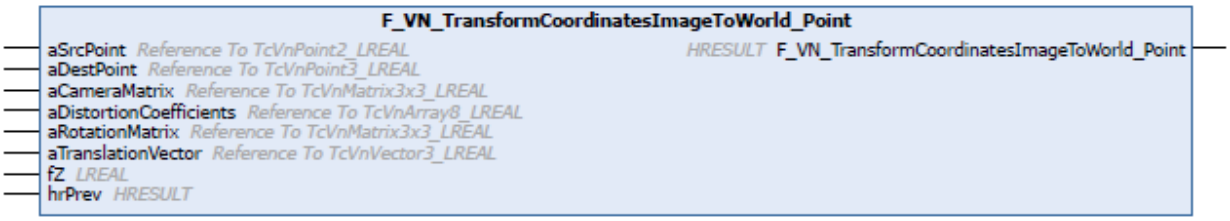

Figure 57: Trasformation reference systems

In the end, when the image is displayed, the correct label with center position will appear, based on the object recognition, with F_VN_PutText().

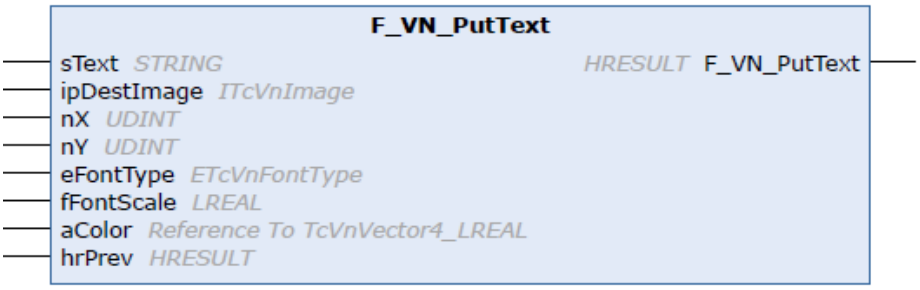

Figure 58: Put label on image

3.2.4 Circle

If the contour has circularity > 80, it is identified as a circle. After highlighting the contours, the center of mass is calculated using F_VN_ContourCenterOfMass().

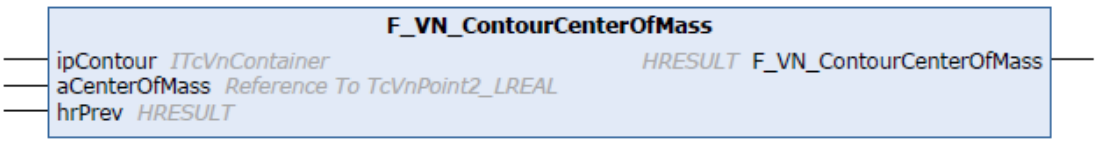

Figure 59: Calculation center of the circle

The center is stored in a data type named TcVnPoint2_LREAL, which is considered as an array of two points: x and y. Again, the transformation from pixel to world point is performed using F_VN_TransformCoordinatesImageToWorld_Point().

```
        ELSE
            hr := F_VN_DrawContours(
                        ipContours      :=  ipContourR,
                        nContourIndex   :=  -1,
                        ipDestImage     :=  ipImageRes,
                        aColor          :=  aColorGreen,
                        nThickness      :=  5,
                        hrPrev          :=  hr
            );
            //find and draw center
            hr := F_VN_ContourCenterOfMass(ipContourR, center, hr);
            hr := F_VN_DrawPointExp(TO_UDINT(center[0]),
                                    TO_UDINT(center[1]),
                                    ipImageRes,
                                    TCVN_DS_X,
                                    aColor,
                                    10,
                                    7,
                                    TCVN_LT_ANTIALIASED,
                                    hr);

            hr := F_VN_TransformCoordinatesImageToWorld_Point(center,
                                                              centerDest,
                                                              aCameraMatrix,
                                                              aDistortionCoefficients,
                                                              aRotationMatrix,
                                                              aTranslationVector,
                                                              fZ,
                                                              hr);
```

Figure 60: ObjectDetection, fifth part

After the transformation, display the image and add the correct label for the detected object.

```
            sTextC := TO_STRING("Circle");
            hr := F_VN_PutText(sTextC, IpImageRes, 50,50, TCVN_FT_HERSHEY_COMPLEX_ITALIC, 1, aColorWhite, hr);

            hr := F_VN_CopyIntoDisplayableImage(ipImageRes, ipImageResDisp, hr);


        END_IF
    END_FOR
END_IF
```

Figure 61: ObjectDetection, sixth part

3.2.5 Object Center Detection results

This section provides some examples of the Object Center Detection algorithm used.

The output can be viewed using ADS Image Watch (Figure 62). Generally, ADS(Automation Device Specification) is used for the communication between TwinCAT runtime and external components, as specified in Section 2.1. It facilitates the live visualization of camera images by transferring them from the PLC runtime to the development environment via ADS, where they are displayed using a specific Port. The image displayed are of type ItcVnDisplayableImage and are declared in MAIN, such as in Figure 'MAIN.ipImageResDisp'.

Figure 62: ADS Image Watch

To open the ADS, click ‘Visualize’ on the toolbar.

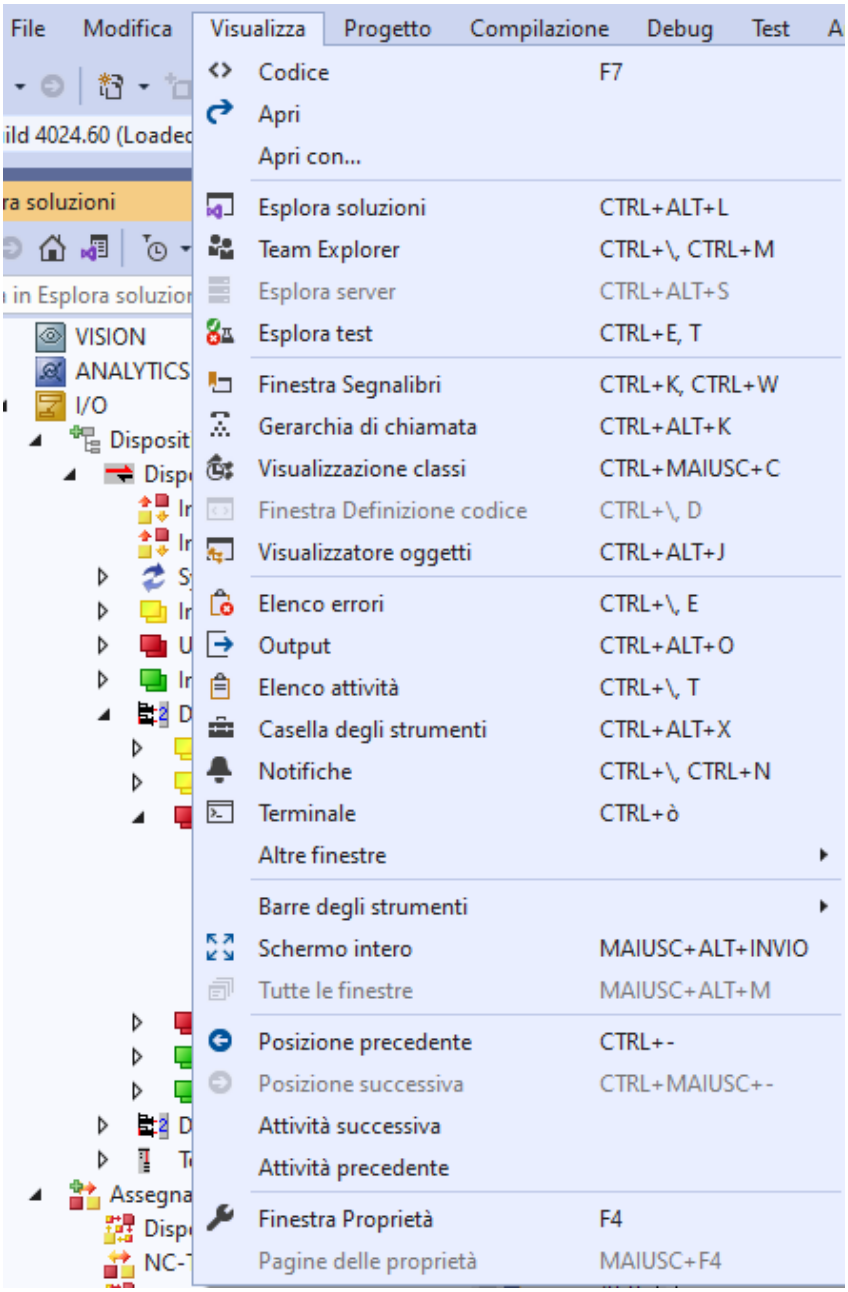

Figure 63: Toolbar

Then select ‘ADS Image Watch’.

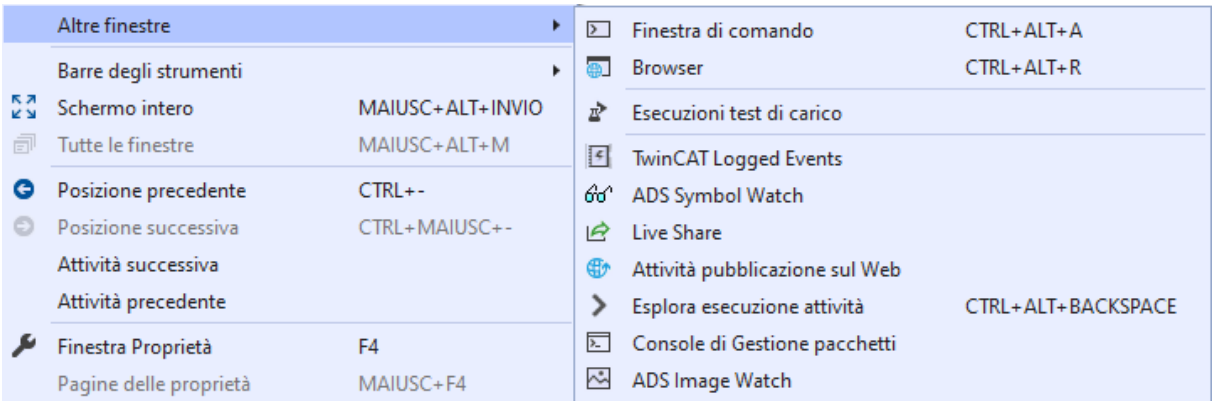

Figure 64: Other Windows

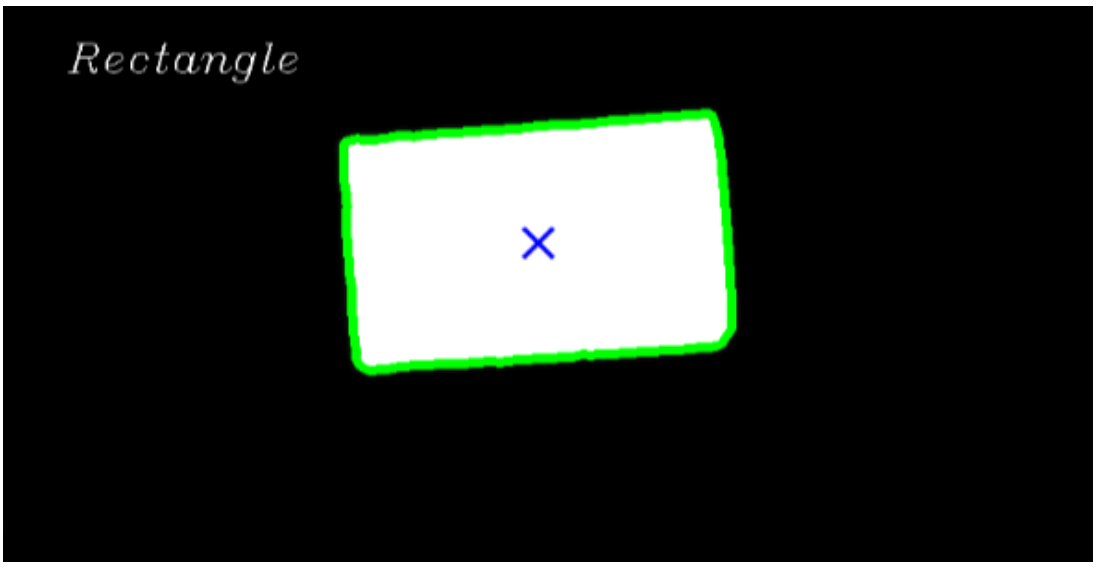

Figure 65: Rectangle

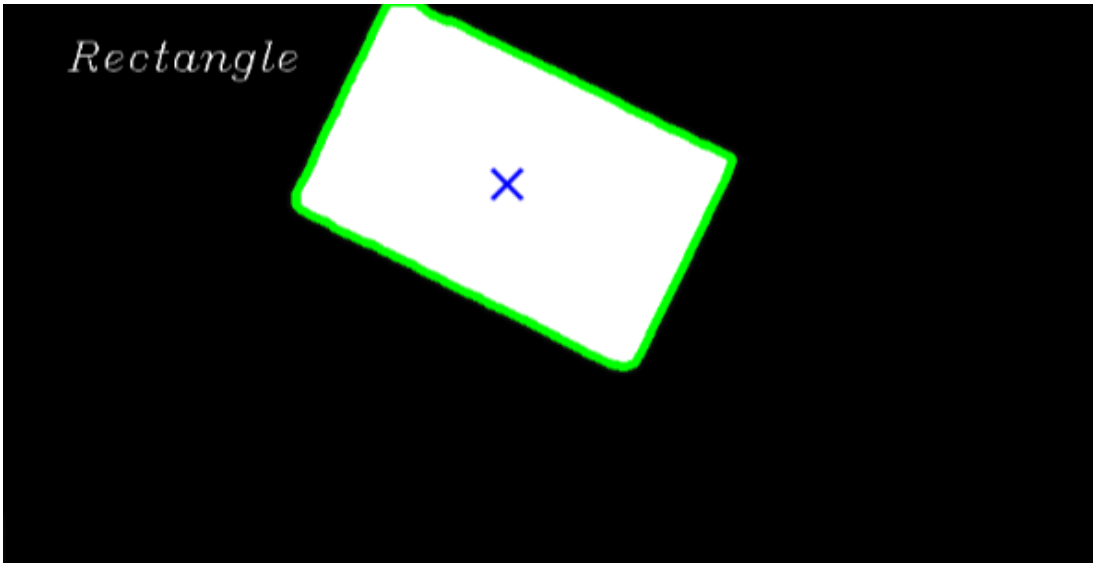

Figure 66: Rectangle 2 rotated

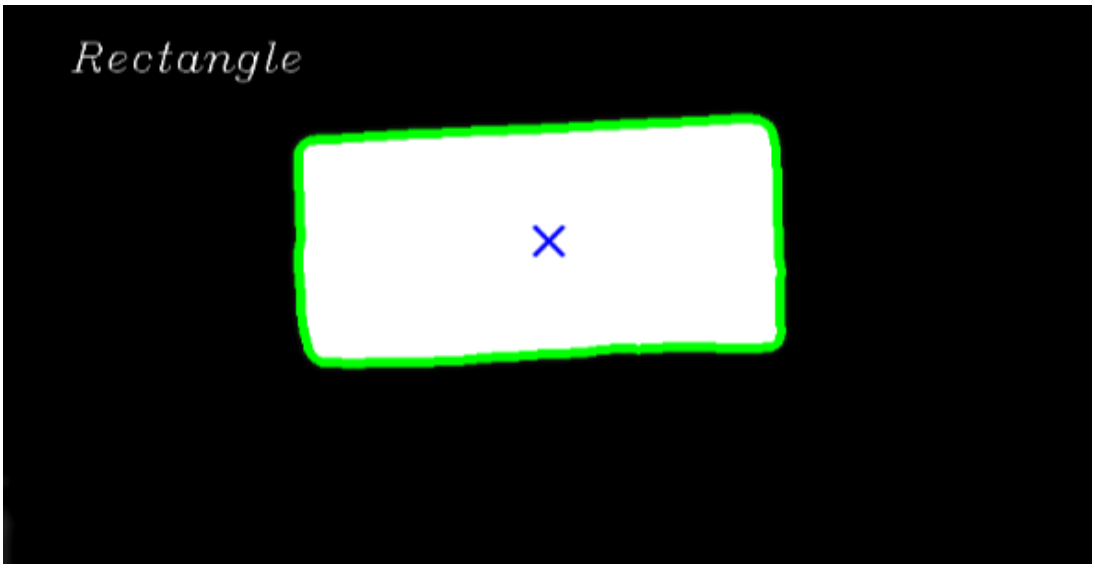

Figure 67: Rectangle 3

The object's center is identified even when it is not completely displayed. Although this may not be precise enough for the robot.

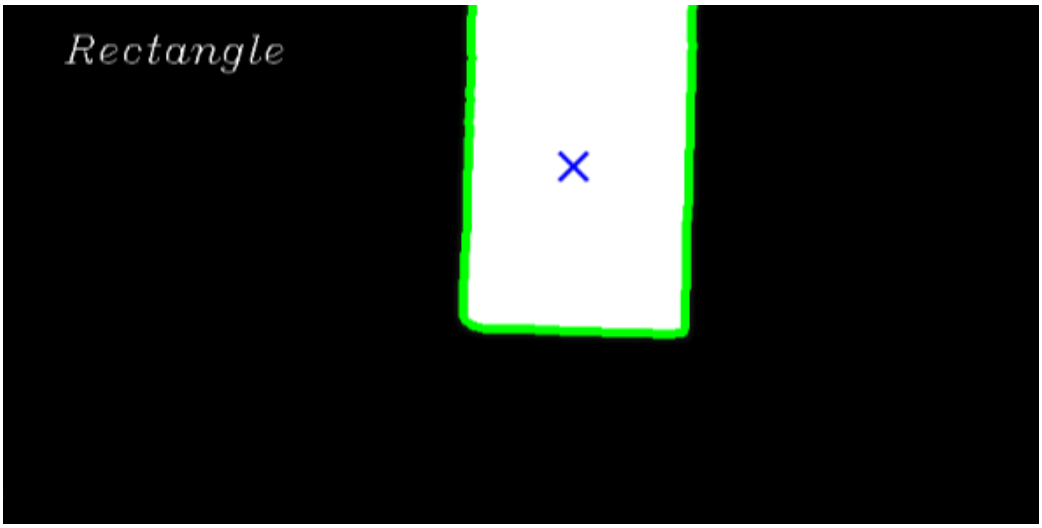

Figure 68: Rectangle out of the ROI

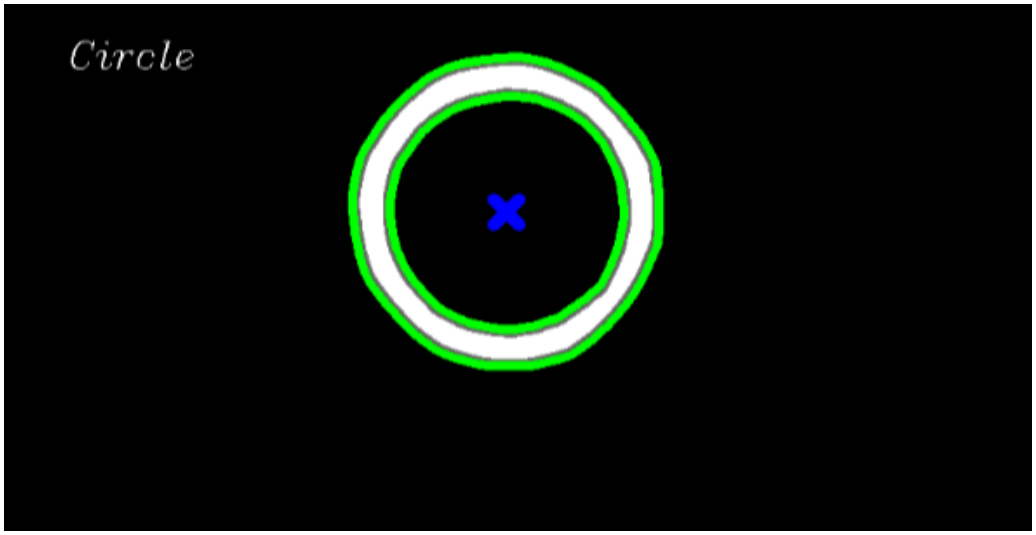

Figure 69: Circle 1 in the center of ROI

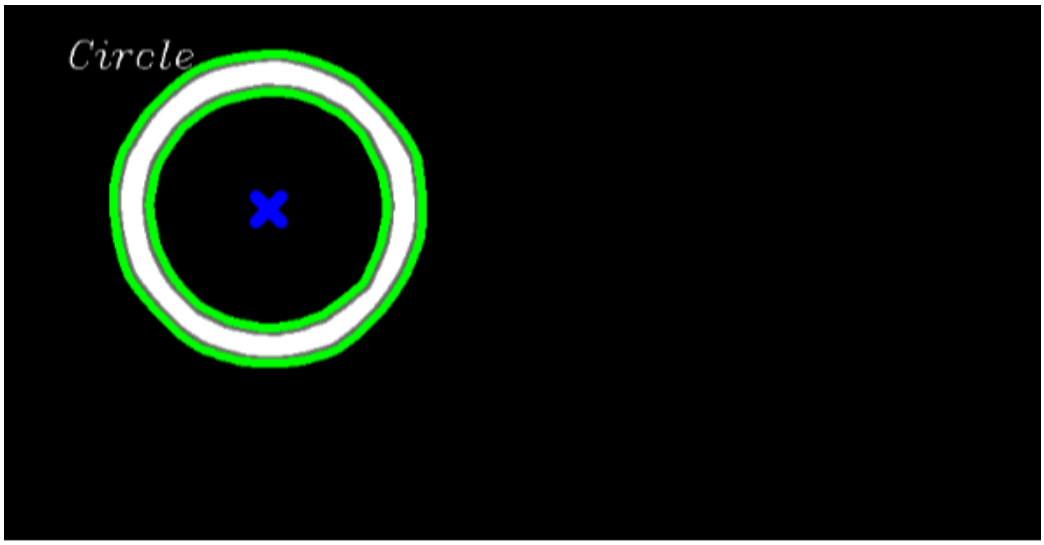

Figure 70:Circle 2 in the left part of ROI

3.2.6 Template Matching

Template matching is a vision technique useful for finding parts of an image that perfectly match a predefined template.

Generally, the algorithm operates by sliding a template image over a source image to determine the position where the template best matches the source image.

In the trials done, see the code in Figure, is used this techinique for recognizing rectangles and circles using a predefined template representing these shapes.

A predefined template, in this context, is an image of an object, either a rectangle or a circle, that matches what we want to recognize in our project. The template image is added to File Source Control and displayed on ADS control using GetCurrentImage().

The function F_VN_MatchTemplateAndEvaluate(), Figure 71, compare the source image from the camera with the template image. If a match is found, it highlights the match location, ipMatches, as the result.

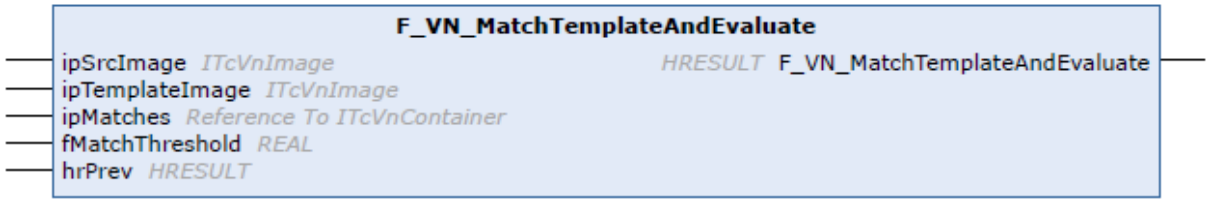

Figure 71: Match Template function

In ipMatches, the coordinates of the object are stored, allowing for the retrieved object to be drawn on the resulting image.

For the rectangle, due to their varying orientations, this algorithm may not always be the best choice; sometimes it successfully recognizes the shape, but in other case, the results differ from expectations.

For the circle, this algorithm performs better because their shape doesn't change orientation.

Finally, this algorithm does not perform as well as the Object Detection methos using DetectBlob() described in the previous section.

```
//use template images taken from directory
hr := fbFileSource.StartAcquisition();
hr := fbFileSource.GetCurrentImage(ipImageTemplate);
hr := F_VN_CopyIntoDisplayableImage(ipImageTemplate, ipImageTemplateDisp, hr);

// In order to draw the contours in green convert the image in RGB
hr := F_VN_ConvertColorSpace(ipImageIn, ipImageRes, TCVN_CST_GRAY_TO_RGB, hr);
hr := F_VN_GaussianFilter(ipImageIn, ipImageIn, 9, 9 ,hr);

//do a conversion in order to do the match on gray images
hr := F_VN_ConvertColorSpace(ipImageTemplate, ipImageTemplate, TCVN_CST_RGB_TO_GRAY, hr);
hr := F_VN_GaussianFilter(ipImageTemplate, ipImageTemplate, 9, 9 ,hr);

//The matches found whose degree of correlation is higher than the specified
//threshold value fMatchThreshold are now located as points in the container ipMatches.
// Each point specifies the top left corner of each occurrence of ipImageTemplate in ipImageIn.
hr := F_VN_MatchTemplateAndEvaluate(ipImageIn, ipImageTemplate, ipMatches, 0.8 , hr);

//save measuments in ordet to rewrite on resulting image
hr := F_VN_GetImageHeight(ipImageTemplate, nHeight, hr);
hr := F_VN_GetImageWidth(ipImageTemplate, nWidth, hr);
hr := F_VN_GetAt_TcVnPoint2_DINT(ipMatches, aPosition, 0, hr);
//in order to access ipMatches
hr := F_VN_GetForwardIterator(ipMatches, ipIterator, hr);

IF SUCCEEDED(hr) AND ipIterator <> 0 THEN
    hr := ipIterator.TcQueryInterface(IID_ITcVnAccess_TcVnPoint2_DINT, ADR(ipAccess));
    IF SUCCEEDED(hr) AND ipAccess <> 0 THEN
        WHILE SUCCEEDED(hr) AND ipIterator.CheckIfEnd() <> S_OK DO
            // access array of DINT
            hr := ipAccess.Get(aPosition);
            // problem with orientation of the object
            hr := F_VN_DrawRectangle(
                        DINT_TO_UDINT(aPosition[0]),
                        DINT_TO_UDINT(aPosition[1]),
                        DINT_TO_UDINT(aPosition[0])+nWidth,
                        DINT_TO_UDINT(aPosition[1])+nHeight,
                        ipImageRes,
                        aColorGreen,
                        5,
                        hr
            );
            hr := F_VN_CopyIntoDisplayableImage(ipImageRes, ipImageResDisp, hr);
            //next ipImatches
            hr := F_VN_IncrementIterator(ipIterator, hr);

        END_WHILE
    END_IF
END_IF
```

Figure 72: Template algorithm implementation

3.2.7 CannyEdgeDetection

The Canny Edge Detector is used for the edge detection, means the boundaries of an object within an image. An edge is defined by the change in pixel intensity. It reduces noise while preserving important features of the original image. The process is divided into several steps:

- Grayscale conversion: the algorithm should be applied to a single channel image to preserve the brightness levels. Thus, the conversion is from RGB(Red, Green, Blue) to Gray.
- Noise reduction: the best filter for noise reduction is the Gaussian Filter which smooths the image using a Gaussian kernel. This kernel slides across the entire image, taking weighted average of the neighboring pixels intensities.

- Gradient calculation: measures the intensity changes for each pixel in a specific direction (x or y of the image). The gradient can be calculated by using the Sobel operator.
- Non-maximum suppression
- Double threshold
- Edge tracking by hysteresis

The function used is F_VN_CannyEdgeDetectionExp. The threshold are set through fThresholdLow and fThresholdHigh range. bL2Gradient is used for the gradient calculation; there are two standards available for this purpose:

- L1 (bL2Gradient is FALSE): The gradient is the sum of absolute values of the gradients. This method is faster.
- L2 (bL2Gradient is TRUE): The gradient is the square root of the sum of the squares of gradients, providing greater accuracy.

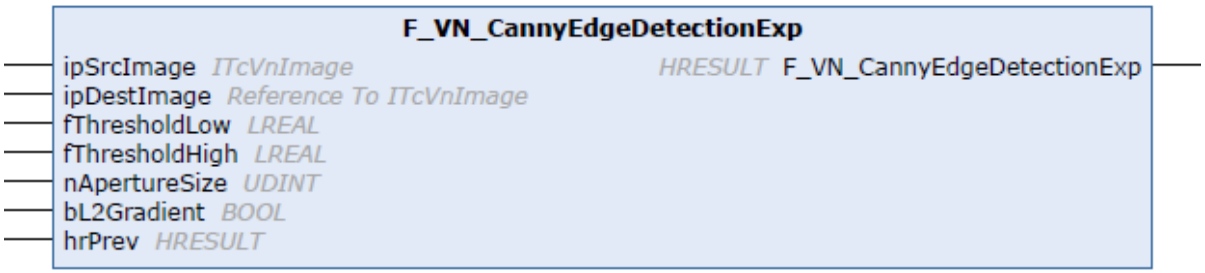

Figure 73: Edge detector

Once the Canny algorithm, see Figure 73, has been applied to the image, the next step is to find contours and then the centers of the figures using the strategy presented in Object Detection.

In conclusion, the algorithm that performs best is the ObjectDetection() with DetectBlobs(), discussed in Section. This is because manually applying different filters provides the opportunity to improve step by step the images and the final results.

```
//found the contour using Canny
//noise remotion
hr := F_VN_GaussianFilter(ipImageIn, ipImageIn, 9, 9 ,hr);
hr := F_VN_MedianFilter( ipImageIn, ipImageIn, 7, hr);

//Otsu calculate the threshold based on the image distribution
hr := F_VN_Threshold(
    ipSrcImage      :=  ipImageIn,
    ipDestImage     :=  ipImageIn,
    fThreshold      :=  0,
    fMaxValue       :=  255,
    eThresholdType  :=  TCVN_TT_OTSU_BINARY,
    hrPrev          :=  hr
);
//Only odd values of 3 or more are valid for nApertureSize.
//L1 faster, L2 accurate
hr := F_VN_CannyEdgeDetectionExp(
    ipSrcImage      :=  ipImageIn,
    ipDestImage     :=  ipImageWork,
    fThresholdLow   :=  fThreshold,
    fThresholdHigh  :=  fMaxValue,
    nApertureSize   :=  5,
    bL2Gradient     :=  TRUE,
    hr
);
hr := F_VN_CopyIntoDisplayableImage(ipImageWork, ipImageWorkDisp, hr);
hr := F_VN_FindContourHierarchyExp(
                        ipSrcImage              :=   ipImageWork,
                        ipContours              :=   ipContoursR,
                        ipHierarchy             :=   ipHierarchyList,
                        eRetrievalMode          :=   eRetrievalMode,
                        eApproximationMethod    :=   eApproximationMethod,
                        aOffset                 :=   aOffset,
                        hrPrev                  :=   hr
);
// In order to draw the contours in blue convert the image in RGB
hr := F_VN_ConvertColorSpace(ipImageWork, ipImageRes, TCVN_CST_GRAY_TO_RGB, hr);
hr := F_VN_DrawContours(
                         ipContours      :=  ipContoursR,
                         nContourIndex   :=  -1,
                         ipDestImage     :=  ipImageRes,
                         aColor          :=  aColorGreen,
                         nThickness      :=  5,
                         hrPrev          :=  hr
            );
hr := F_VN_CopyIntoDisplayableImage(ipImageRes, ipImageResDisp, hr);
```

Figure 74: Algorithm with Canny Edge Detector

Chapter 4: Machine Learning

Instead of using the previous developed filters in TwinCAT, it is possible to create a neural network and perform a comparison for the recognition of rectangles and circles centers.

For this task, Google Colab is used with a GPU(Graphics Processing Units) configuration, designed to handle complex computations efficientiently. However, Google Colab provides a default CPU runtime.

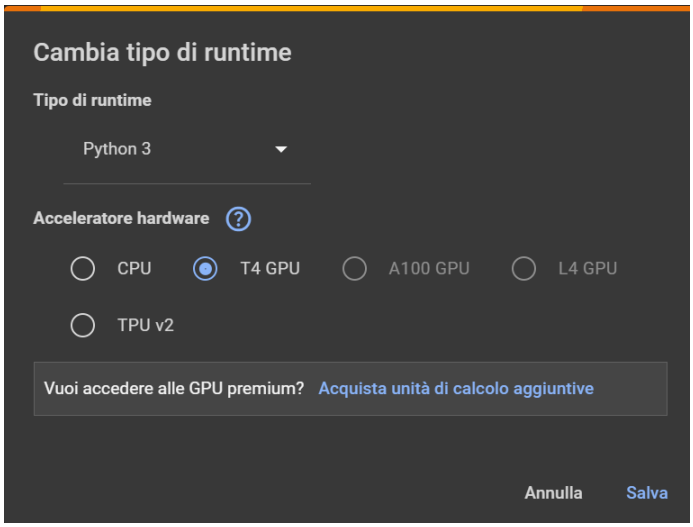

Figure 75: Choose a runtime between CPU, GPU, TPU

The python notebook, saved as a .ipynb file, can be accessed at the following link: https://colab.research.google.com/drive/1D4dwXXG8OBxs_igEdWb5X3a98iKJEbaY .

In Google Colab, the code is divided into sections, which helps to create an organized notebook where each part serves a specific task. The main limitation of this notebook is its limited memory; due to the small amount of RAM, training a neural network will be slow, even with small datasets. Consequently, the number of images used is not ideal but it is the maximum that can be supported given these memory constraints.

This section highlights the process of creating a neural network capable of recognizing the centers of circles and rectangles. The process involves the following steps:

- Creation and preparation of an appropriate dataset
- Division of the data into training, testing and validation set
- Definition of the architecture of a CNN, considering different types of layers
- Definition of the loss function, optimization method and accuracy metrics
- Training of the model
- Testing using images captured by the camera

4.1 Introduction

Firstly, the necessary libraries for this notebook are imported (Figure 76). If a library is missing, the command '!pip install' is the command for the installation.

```
[ ]  1 from PIL import Image, ImageDraw
     2 import random
     3 import imgaug.augmenters as iaa
     4 import numpy as np
     5 import matplotlib.pyplot as plt
     6
     7 import tensorflow as tf
     8 print(tf.__version__)
     9
    10 from tensorflow import keras
    11 from sklearn.model_selection import train_test_split
    12 from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
    13 from tensorflow.keras.optimizers import Adam
    14 from sklearn.utils import class_weight
    15 from tensorflow.keras.utils import to_categorical
    16 from google.colab import files
    17 import cv2
    18 from io import BytesIO
    19 import math
    20 import random
    21 import csv
    22 from matplotlib.patches import Circle
    23 import pandas as pd
    24
2.17.0
```

Figure 76: Import necessary libraries

The main library used is Tensorflow, which can be installed with the command !pip install tensorflow.

Tensorflow is an open-source and free software library designed for high performance computations such as those needed machine learning and artificial intelligence.

As illustrated in the Figure, execution starts from the play symbol located on the left part of the each cell. If all the cells need to be run, there is a specific button 'Run all' shown in Figure 77.

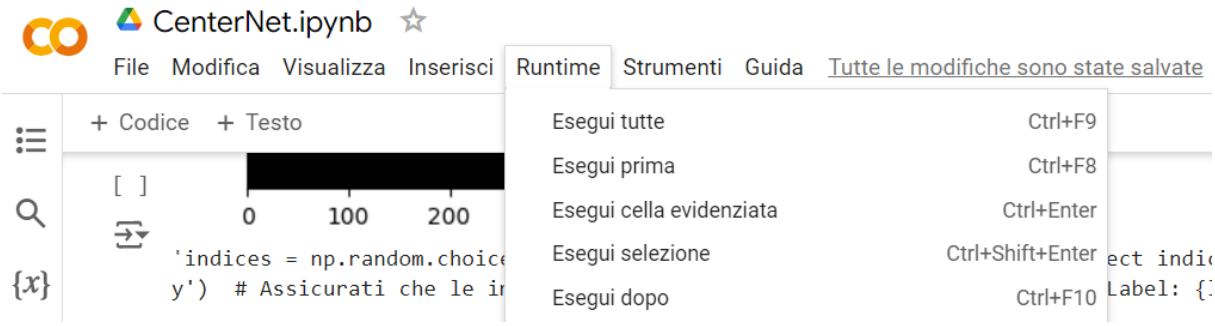

Figure 77: Run all the cells in the notebook

To proceed, there's a useful check to prevent slowing down the training in the subsequent sections: by using the function in the Figure 78, you can ensure that the code will run using the GPU rather than the CPU.

```
1 #use gpu for training
2 print(tf.test.is_gpu_available())
```

Figure 78: test if the cell is executed with GPU

4.2 Dataset generation

A good neural network in the field of computer vision should be trained with a large number of images to perform better.

Therefore, the first step is to generate a substantial dataset of images containing circles and rectangles, along with the respective coordinates of their centers. A large dataset helps prevent the problem of overfitting and enhances the performance of the considered CNN.

The generated images are entirely black except for the geometric shapes.

Two functions are used for shape generation:

- Generate_circles(count, image_width, image_height) is the function for generating circle and its center. The inputs are:
    - Count: the number of images to be generated.
    - Image_width and Image_height: dimensions of the images generated by the camera.

```
[ ]
# 0 stands for circle, 1 stands for rectangle, x-y are the center coordinates
def generate_circles(count,  image_width, image_height):
    images = []
    labels = []
    annotations = []

    # count is the total numer of images
    for i in range(count):
        #set color of the image as black, black as the conveyor
        #img = Image.new('RGB', (image_width, image_height), 'black')
        img = Image.new('L', (image_width, image_height), 'black')
        draw = ImageDraw.Draw(img)

        # Generate random radius, it is calcolated based on the image's dimension
        # 5% of width or height
        min_radius = int(0.05 * min(image_width, image_height))
        # 15% of width or height
        max_radius = int(0.40 * min(image_width, image_height))
        # choose at random btw the radius
        radius = random.randint(min_radius, max_radius)

        # Generate x and y to avoid cirle out of the image
        # center circle
        x = random.randint(radius, image_width - radius)
        y = random.randint(radius, image_height - radius)
```

Figure 79: Generate circles function 1

```
        # define buonding box (left orizontal, left vertical, right orizontal, right vertical)
        draw.ellipse([x - radius, y - radius, x + radius, y + radius], outline='white', width=7, fill='white')

        img = pil_to_tf(img)
        img = tf_to_pil(img)

        img = img.resize((image_width, image_height), Image.Resampling.LANCZOS)

        # useful for center
        labels.append([x, y])

        # useful for recognition of circle
        #labels.append(0)
        images.append(img)
        img.save(f'/content/circle_{i}.png')
        annotations.append({'filename': f'/content/circle_{i}.png', 'label': 0, 'center': [x, y]})
    # Salva le annotazioni in un file CSV
    with open('circle_annotations.csv', 'w', newline='') as csvfile:
        fieldnames = ['filename', 'label', 'center_x', 'center_y']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        for annotation in annotations:
            writer.writerow({'filename': annotation['filename'], 'label': annotation['label'],
                             'center_x': annotation['center'][0], 'center_y': annotation['center'][1]})

    return images, labels
```

Figure 80: Generate circles function 2

The outputs of this function are images and their corresponding labels, which are saved in a .csv file containing the image path with its extension, label and center coordinates.

The process begins with generating a random radius for each circle to ensure diversity in their size.

The minimum radius is calculated as 5% of the smaller dimension of the image, while the maximum radius is the 15% of that same dimension. The radius for each circle is then selected using random.randint to choose the minimum or maximum values.

To ensure that the circles do not extend beyond the image boundaries, correct x and y coordinates are calculated using formula image_width/image_height – radius. This calculation helps place the circle within the image frame properly.

- Generate_rectangles(count, image_width, image_height) is the function used to generate rectangle. The inputs for this function are the same as those for the circle generation.

```
def generate_rectangles(count,  image_width, image_height):
    images = []
    labels = []
    annotations = []

    for i in range(count):
        #set color of the image as black
        #img = Image.new('RGB', (image_width, image_height), 'black')
        img = Image.new('L', (image_width, image_height), 'black')
        draw = ImageDraw.Draw(img)

        # redefinition width btw 70% and 590% of the width image
        rect_width = int(image_width * random.uniform(0.5, 0.7))
        # height btw 10% and 30% dof the height image
        rect_height = int(image_height * random.uniform(0.3, 0.5))

        x0 = random.randint(0, image_width - rect_width)
        y0 = random.randint(0, image_height - rect_height)
        x1 = x0 + rect_width
        y1 = y0 + rect_height
        draw.rectangle([x0, y0, x1, y1], outline = 'white', width=7, fill='white')
        center_x = (x0 + x1) // 2
        center_y = (y0 + y1) // 2
        # Randomly decide whether to rotate the rectangle and by what angle
        # 50% chance to rotate the rectangle
```

Figure 81: Generate rectangles function 1

```
[ ]
        # 50% chance to rotate the rectangle
        if random.choice([True, False]):
            # Rotate by a random angle from 30 to 360 degrees
            angle = random.uniform(30, 360)
            #img = img.rotate(angle, expand=True, fillcolor='black')
            img, center_x, center_y = rotate_image_and_center(img, angle, center_x, center_y, rect_width, rect_height)

        img = pil_to_tf(img)
        img = tf_to_pil(img)
        #resize the image to avoid inconsistency
        img = img.resize((image_width, image_height), Image.Resampling.LANCZOS)

        labels.append([center_x, center_y])
        images.append(img)
        img.save(f'/content/rect_{i}.png')
        annotations.append({'filename': f'/content/rect_{i}.png', 'label': 0, 'center': [center_x, center_y]})
    # Salva le annotazioni in un file CSV
    with open('rect_annotations.csv', 'w', newline='') as csvfile:
        fieldnames = ['filename', 'label', 'center_x', 'center_y']
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()
        for annotation in annotations:
            writer.writerow({'filename': annotation['filename'], 'label': annotation['label'],
                             'center_x': annotation['center'][0], 'center_y': annotation['center'][1]})

    return images, labels
```

Figure 82: Generate rectangles function 2

In the case of rectangles, the generation process starts by defining their width and height. These dimensions are set proportionally to the dimensions of the image, with the width ranging from 50% and 70% of the image width, instead the height from 30% and 50% of the image height. To draw the rectangle, the coordinates x0, y0, x1, y1 need to be calculated, they are the four points representing the top left corner(x0, y0) and the bottom right corner(x1, y1).

Additionally, data augmentation (explained in detail in 4.2 section) includes rotating the rectangle by a random angle between 30 and 360 degrees to enhance the model's robustness.

For each shape the number of images generated is 200.

```
[ ]
# number of images per shape
num_images = 200

# Generate images and labels for both shapes
circle_images, circle_labels = generate_circles(num_images, 800, 600)
rectangle_images, rectangle_labels = generate_rectangles(num_images, 800, 600)

#in images there will be circles and rectangles
images = np.concatenate((circle_images, rectangle_images))
labels = np.concatenate((circle_labels, rectangle_labels))
```

Figure 83: Generate images and labels for circles and rectangles

Initially, trials are conducted with a larger number of images such as 5000 for circles and 5000 for rectangles. However, training is interrupted without yielding results due to the memory constraints, leading to the conclusion that more powerful hardware is needed to handle such a large dataset.

The images and labels generated by generate_circles() and generate_rectangles() functions are saved in circle_images, circle_labels and rectangles_images, rectangles_labels respectively

(Figure 83). The width and height reported in these functions match the output of the camera : 800x600 pixels.

After generating the images and labels, all data is combined using the operator concatenation. A test is conduced to ensure that each image retains the correct label post-concatenation, such as in Figure 84.

```
[ ]
####test########
#from matplotlib.patches import Circle
# Function to display an image with its center marked
def show_with_center(img, center, title):
    fig, ax = plt.subplots()
    ax.imshow(img, cmap='gray')
    # Extract x and y from center
    x, y = center
    # Create a red circle at the center
    circle = Circle((x, y), 10, color='red', fill=False)
    ax.add_patch(circle)
    plt.title(title)
    plt.show()


# Testing rectangles
for i in range(10):
    img = rectangle_images[i]
    label = rectangle_labels[i]
    show_with_center(img, label, f'Rectangle Label: {label}')

# Testing circles
for i in range(10):
    img = circle_images[i]
    label = circle_labels[i]
    show_with_center(img, label, f'Circle Label: {label}')
```

Figure 84: Show the centers of both shapes

To further improve the accuracy of the model a shuffle of images indices is performed (Figure 85). This randomizes the order of circle and rectangles, which enhances training performance. Additionally, in testing phase, it's important to verify that the indices correspond correctly to the images, as mismatches can occur.

```
[ ]
# create an array indices
indices = np.arange(len(images))

# casual index
np.random.shuffle(indices)

# Ora, se stampi la lunghezza degli array, dovrebbero essere uguali
print(f"Numero di immagini dopo il rimescolamento: {len(images)}")
print(f"Numero di etichette dopo il rimescolamento: {len(labels)}")
# reorder the array based on indices
images = images[indices]
labels = labels[indices]

# check lenght after shuffle
print("Length of images:", len(images))
print("Length of labels:", len(labels))

#test images and labels synchronization
indices = np.random.choice(len(images), 5, replace=False)
for index in indices:
    plt.imshow(images[index], cmap='gray')
    plt.title(f'Label: {labels[index]}')
    plt.show()

Numero di immagini dopo il rimescolamento: 400
Numero di etichette dopo il rimescolamento: 400
```

Figure 85: Shuffle images and labels

Subsequently, the output is verified to be correct and appears as shown in the figures below. The coordinates, in square brackets, are displayed in the pixel coordinate system.

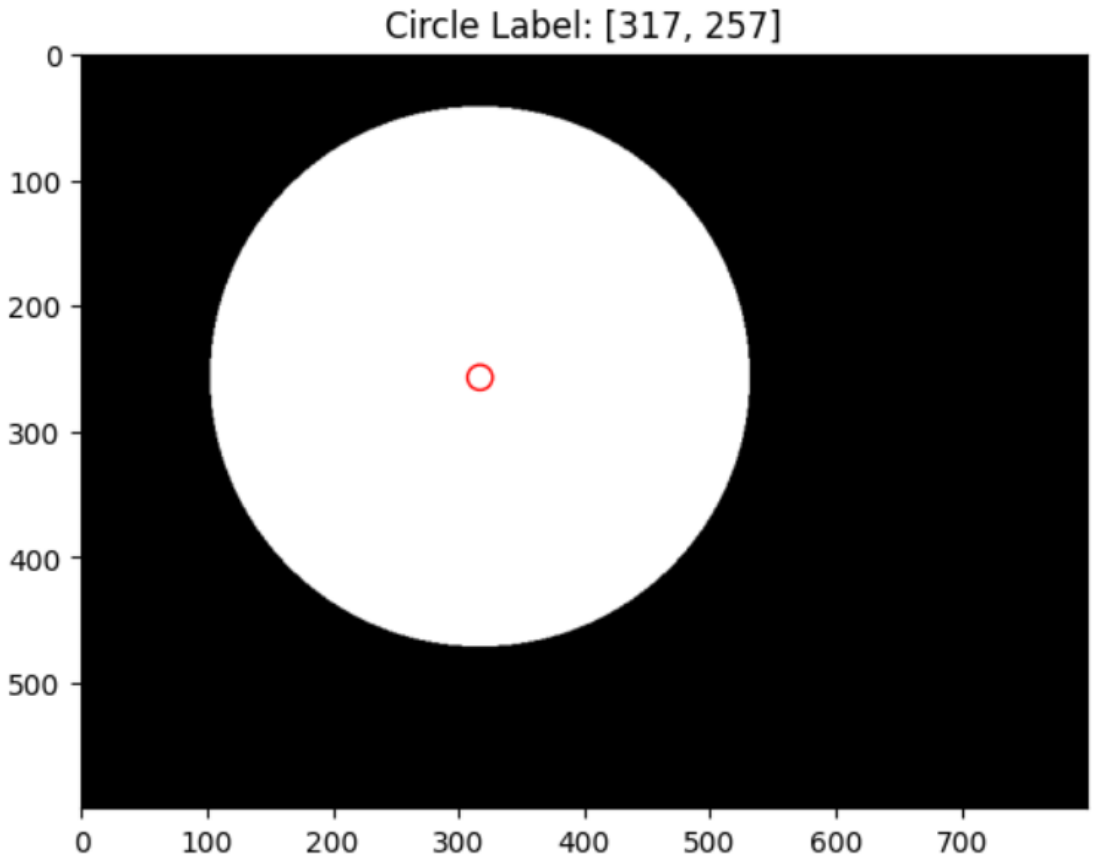

Figure 86: Circle with larger random radius

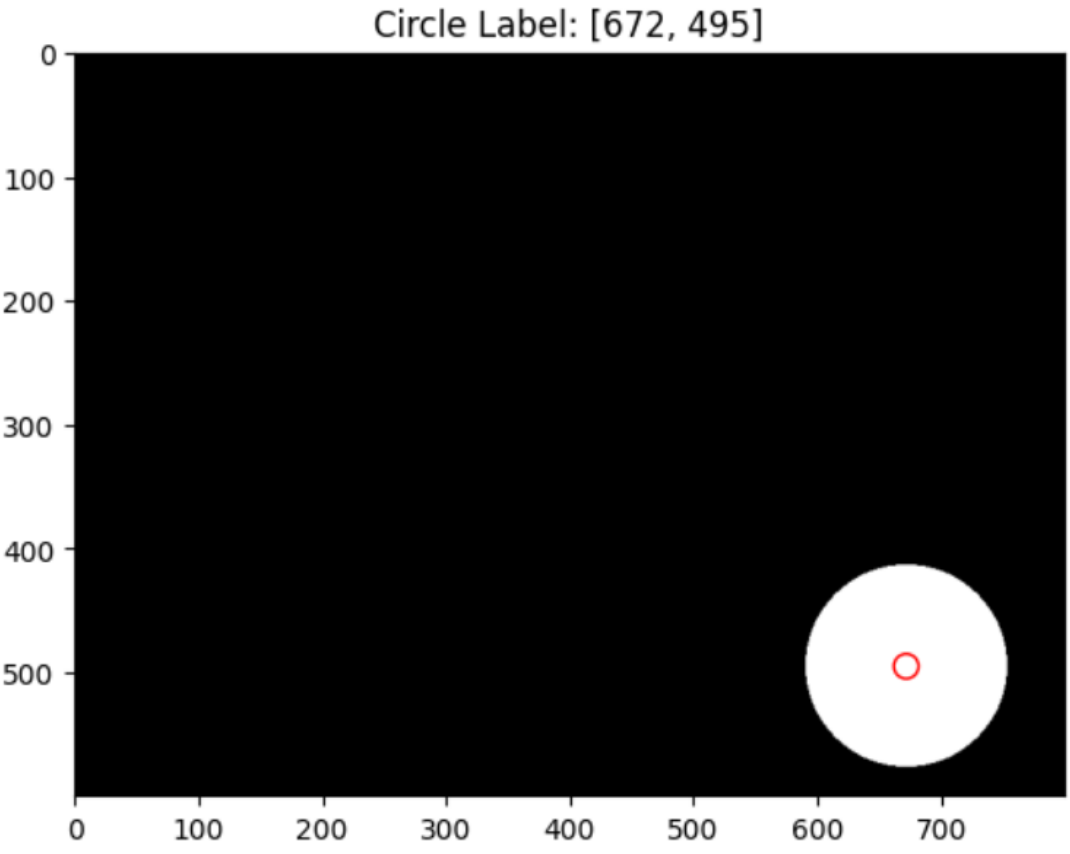

Figure 87: Circle with smaller random radius

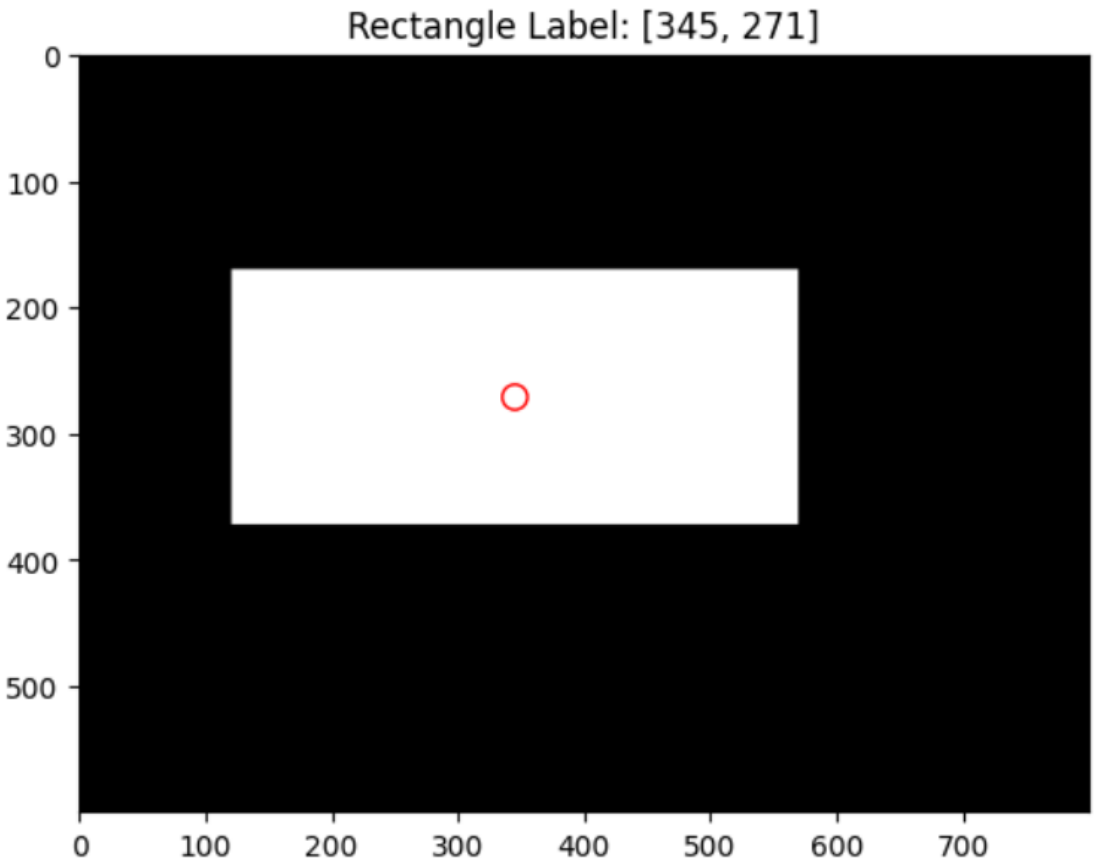

Figure 88: Rectangle in the image's center

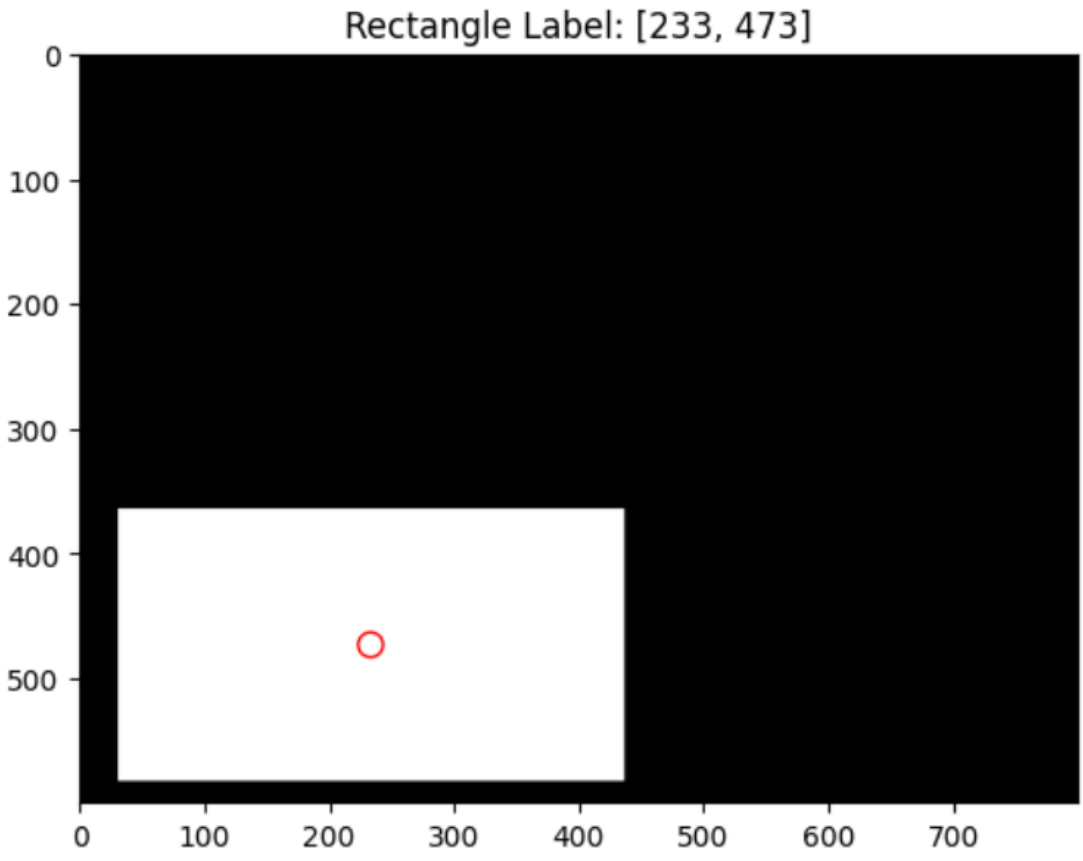

Figure 89: Rectangle in the bottom left of the image

4.2 Data augmentation

Data augmentation is a technique used to artificially increase the size of a training dataset by applying various transformations to the existing data[14]. There are several benefits:

- It adds new data to the existing dataset, increasing efficiency and performance.
- It increase variations in the data, which helps prevent overfitting.
- More data leads to better generalization.

```
[ ]
def rotate_image_and_center(img, angle, center_x, center_y, width, height):
    # Calculate the corners of the rectangle
    corners = [
        (-width / 2, -height / 2),   # Top-left
        (width / 2, -height / 2),    # Top-right
        (width / 2, height / 2),     # Bottom-right
        (-width / 2, height / 2)     # Bottom-left
    ]

    # Calculate new corners after rotation
    rad_angle = math.radians(angle)
    new_corners = [
        (
            center_x + x * math.cos(rad_angle) - y * math.sin(rad_angle),
            center_y + x * math.sin(rad_angle) + y * math.cos(rad_angle)
        )
        for x, y in corners
    ]
```

Figure 90: Data augmentation first part

```
    # Find min and max x, y to get the bounding box
    min_x = min(x for x, y in new_corners)
    max_x = max(x for x, y in new_corners)
    min_y = min(y for x, y in new_corners)
    max_y = max(y for x, y in new_corners)

    # Check if the new bounding box fits within the image
    if min_x < 0 or max_x > img.width or min_y < 0 or max_y > img.height:
        return img, center_x, center_y  # Return original if it doesn't fit

    # Rotate the image
    img = img.rotate(angle, expand=True, center=(center_x, center_y), fillcolor='black')

    # Recalculate the new center after rotation
    new_center_x = (min_x + max_x) / 2
    new_center_y = (min_y + max_y) / 2

    return img, int(new_center_x), int(new_center_y)
```

Figure 91: Data augmentation second part

The data augmentation process describes in this code, Figure 90/91, is contingent on a binary flag, random.choice([True, False]), which randomly decides whether to apply a transformation. For instance, rotations are applied to rectangles in 50% of the cases. This approach allows for the generation of images featuring normal rectangles, rotated rectangles and rectangles of varying sizes.

These steps are followed for rotating a rectangle:

- Calculation of the corners in the original rectangle.
- Rotation of the vertices based on a random angle.
- Definition of the bounding box containing the new rectangle.
- Verification that this bounding box fits within the image dimensions. If it does, the new coordinates of the center are used; otherwise, the original rectangle is retained.

Rotation is not applied to the circle because the change would not be noticeable.

The images generated maintain a balance between circles and rectangles.

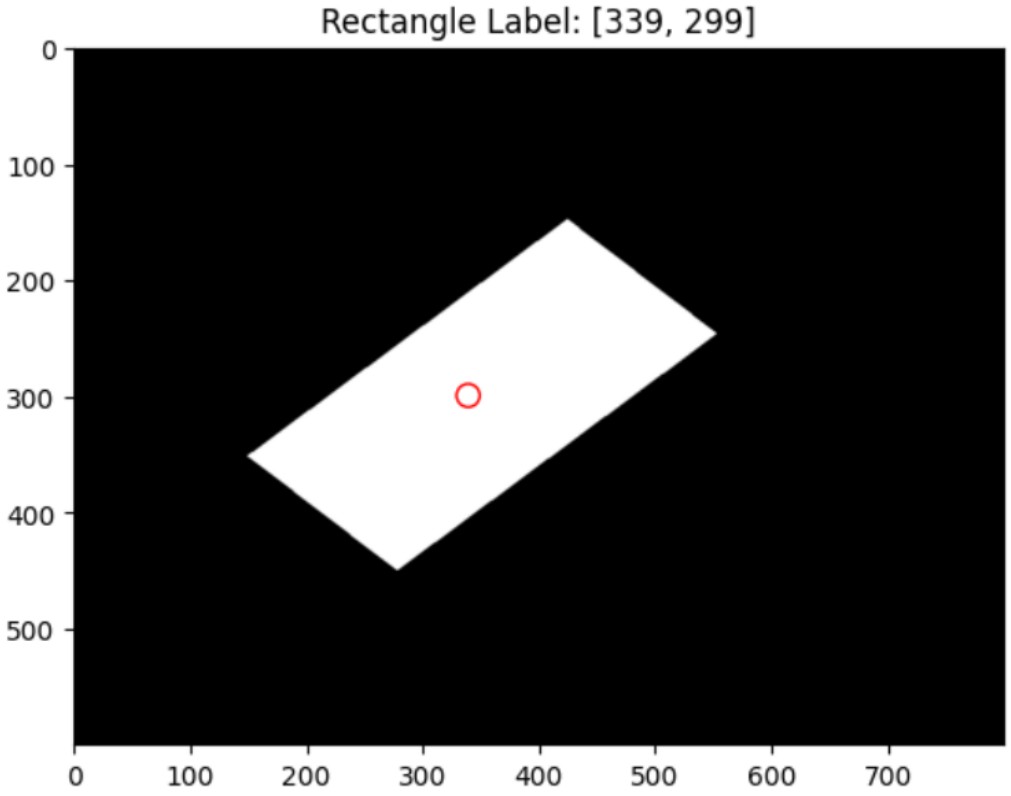

Figure 92: Rectangle rotation 1

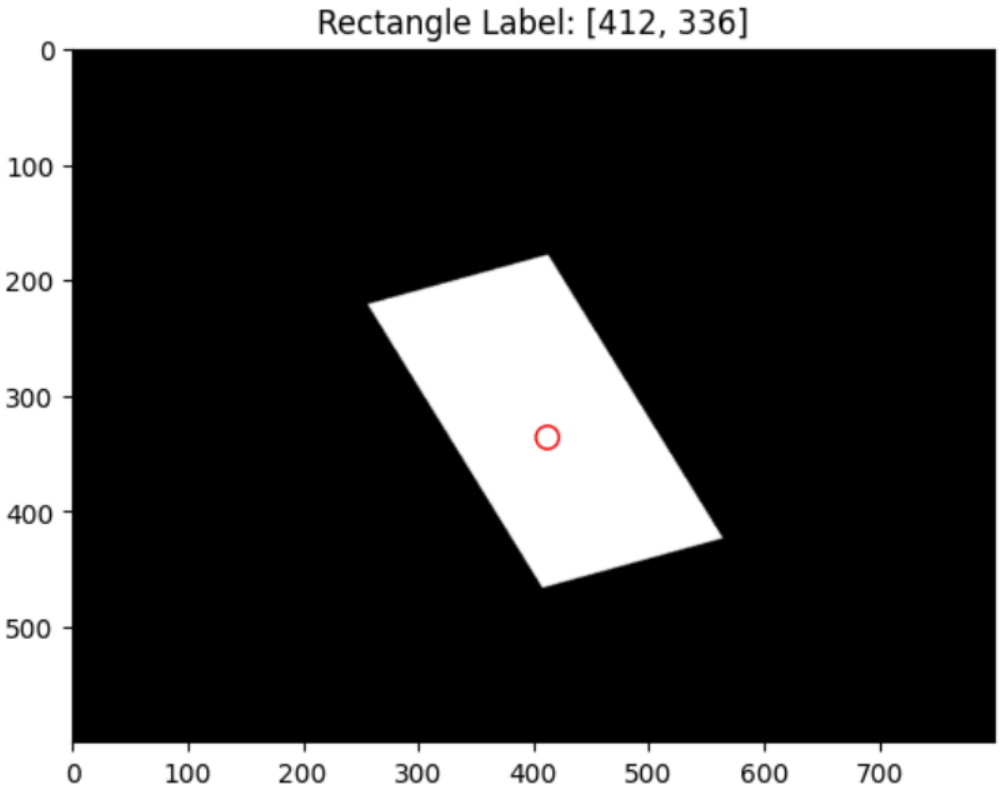

Figure 93: Rectangle rotation 2

4.3 Models

Keras is a high level API in TensorFlow that is useful for machine learning tasks. Models are typically created using tf.keras.models.Sequential, which takes various layers as input.

The choice of layers can significantly impact the model's accuracy, loss and its ability to correctly predict labels.

For this reason, different configurations of the model will be presented, each with unique layer arrangements tested for effectiveness.

The usual procedure begins with loading the dataset, Figure 94, in this case from two CSV files: circle_annotations.csv and rectangle_annotations.csv. These files contain annotations for training the model to recognize and differentiate between circles and rectangles.

```python
# Carica il dataset dal CSV
def load_dataset(csv_file):
    dataset = pd.read_csv(csv_file)
    images = []
    labels = []

    for index, row in dataset.iterrows():
        image_path = row['filename']
        center_x = row['center_x']
        center_y = row['center_y']

        # Carica e preprocessa l'immagine
        image = Image.open(image_path)
        #image = image.resize((800, 600))  # Dimensione compatibile con il modello CenterNet
        #image = np.array(image) / 255.0  # Normalizzazione tra 0 e 1
        images.append(image)

        # Salva le coordinate del centro come etichetta
        labels.append([center_x, center_y])

    return np.array(images), np.array(labels)
```

Figure 94: Load the dataset from CSV file

Images and labels are then concatenated.

```python
# Carica le immagini e le annotazioni per cerchi e rettangoli
circle_images, circle_labels = load_dataset('/content/circle_annotations.csv')
rect_images, rect_labels = load_dataset('/content/rect_annotations.csv')

# Combina i dataset di cerchi e rettangoli
all_images = np.concatenate([circle_images, rect_images], axis=0)
all_labels = np.concatenate([circle_labels, rect_labels], axis=0)

# Verifica le dimensioni del dataset
print(f"Total images: {all_images.shape}, Total labels: {all_labels.shape}")
```

```
Total images: (400, 600, 800), Total labels: (400, 2)
```

Figure 95: Concatenation

For training, it is essential to divide the images into training, test and validation set; in this case the train_test_split function is used with the size of the test of 20% of the total images.

```python
#training model
X_train, X_test, y_train, y_test = train_test_split(all_images, all_labels, test_size=0.2)
```

Figure 96: Split the dataset

Following the dataset split, the next step involves creating a convolutional neural network(CNN).

Typically, a CNN, is structured in layers such as in Figure 97, each one serving a specific purpose:

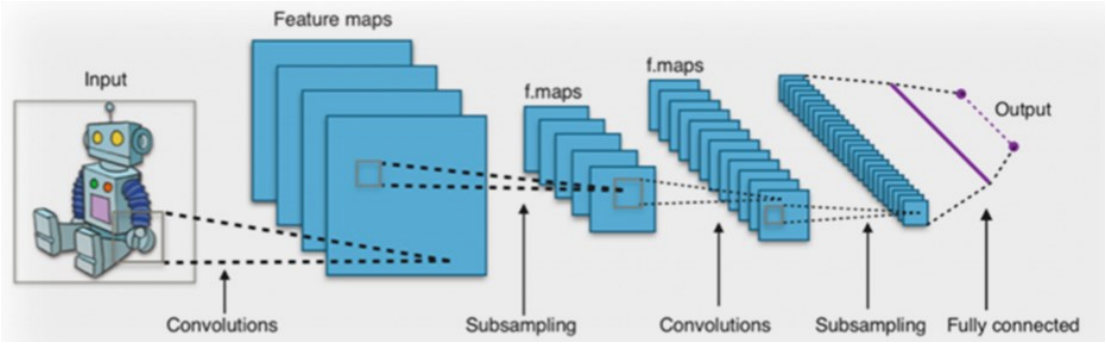

Figure 97: CNN structure

- Input layer: takes the image as input, representing is as a group of pixels.
- Convolutional layer: here, features such as lines, edges and curves are detected. This layer employs convolution operations to extract these features.
- ReLU(Rectified Linear Unit) layer: its input is the output from previous convolutional level. It is an activation function defined as f(x) = max (0,x); this function introduce non linearity to the model, allowing it to learn more complex patterns.
- Pooling layer: reduces dimensionality of the image. While the convolutional layer outputs so dense information, the pooling simplifies these outputs while retaining essential features.

  There are different types of pooling:

  - Max pooling: reduces spatial dimensions by selecting the maximum value from each small window or region.
  - Average pooling: computes the average value for each patch of the feature map.

  For this thesis, max pooling is employed. It reduces the spatial dimensions of features by selecting the maximum value within each region. Specifically, for each small window or region, max pooling selects the maximum value and replaces the original data with it [15].

  It reduces computational load on the network, capturing the essence of visual features.

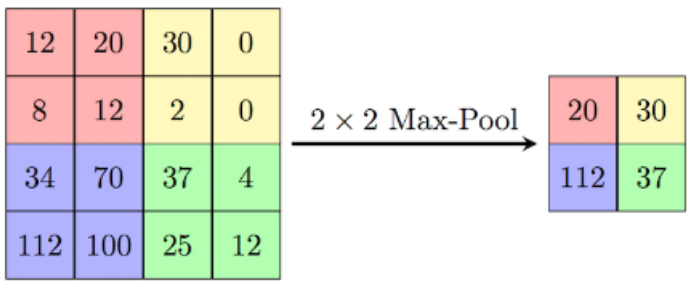

Figure 98: Example of Max Pooling

4.1.1 First model

The initial model architecture, in Figures 99/100, consists of the following layers configured using TensorFlow's Keras API:

- Convolutional layer (tf.keras.layers.Conv2D()): takes an input image shape of 800x600 pixels. It applied a convolution operation using a kernel of size 3x3.
  Initially a smaller number of the filters is used to focus on detecting simple features. As more convolutional layers are added, the number of filters increases to enhance the model's ability to recognize more complex objects.
- Max Pooling Layer (tf.keras.layers.MaxPool2D()): follows the convolutional layer uses a kernel of size 2x2 to perform max pooling as shown in Figure.
- Flatten layer (tf.keras.layers.Flatten()): flattens the input multidimensional output of the previous layer into one-dimensional(1D vector). array to prepare it for input into the dense layers.
- Dense Layer (tf.keras.layers.Dense()): is a fully connected layer where every neuron is connected to all neurons in the previous layer. The layer performs a weighted sum of the inputs, and these results are passed through an activation function.
  In this model, a dense layer is used both in the penultimate layer and also as output layer, where it defines the two coordinates: x and y.
- Dropout layer (tf.keras.layers.Dropout()): randomly sets inputs units to zero during training time at a specified rate, which helps to prevent overfitting. The rate varies from 0 to 1.

```python
#training model
model = tf.keras.models.Sequential([

    # at the beginning the cnn captures contour and edges then simple filter like 32
    # in the next layers in order to capture complex objects it use high filter like 256
    # (filter, filter dimension, activation, image dimension and channel)
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=[800, 600, 1]),
    # (pooling size)
    tf.keras.layers.MaxPooling2D((2, 2)),
    # randomly sets input units to 0 during training time, preventing overfitting

    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),

    tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),

    tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),

    # feature maps in a single vector
    tf.keras.layers.Flatten(),
```

Figure 99: First model 1

```
27
28      # dense
29      tf.keras.layers.Dense(256, activation='relu'),
30      tf.keras.layers.Dropout(0.5),
31
32      #2 stands for x,y coordinates
33      tf.keras.layers.Dense(2, activation='relu')
34  ])
```

Figura 100: First model 2

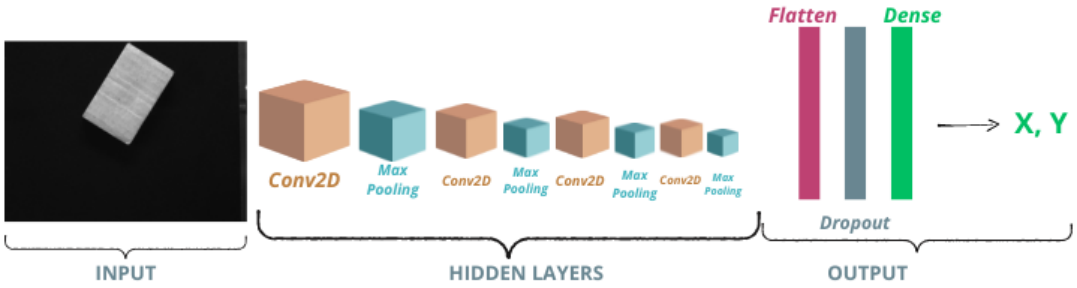

Figure 100a: First model scheme

Actually this model doesn't perform very well, probably due to the limited number of layers.

4.1.2 Second model

The second iteration of the model, Figures 101/102, incorporate dropout layers into the existing architecture. tf.keras.layers.Dropout() is added after different layers. This addition provides several benefits:

- Dropout after Dense Layer (tf.keras.layers.Dense): the layer learn high level feature of the input, introducing Dropout after Dense some neurons are randomly set to zero during the training phase, reducing the probability of overfitting.
- Dropout after Convolutional layer (tf.keras.layers.Conv2D()), the layer captures an huge number of patterns, Dropout prevents the overfitting maintaining features captured.

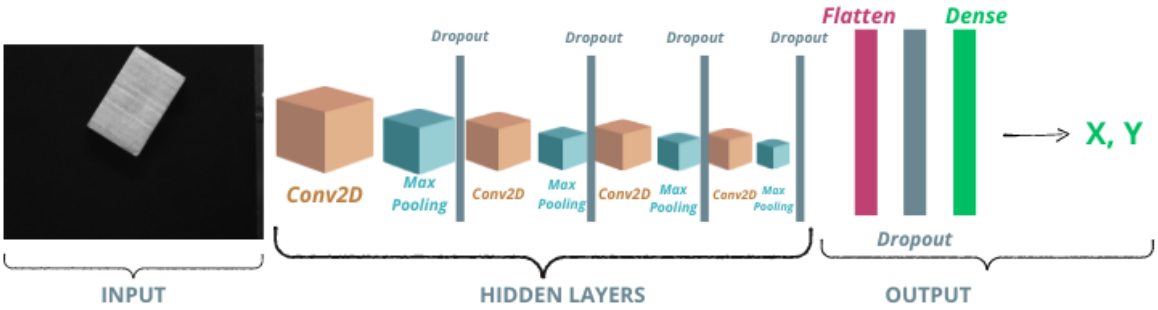

Figure 101a: Second model scheme

```
[ ]
#training model
model = tf.keras.models.Sequential([

    # at the beginning the cnn captures contour and edges then simple filter like 32
    # in the next layers in order to capture complex objects it use high filter like 256
    # (filter, filter dimension, activation, image dimension and channel)
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=[800, 600, 1]),
    # (pooling size)
    tf.keras.layers.MaxPooling2D((2, 2)),
    # randomly sets input units to 0 during training time, preventing overfitting
    tf.keras.layers.Dropout(0.25),

    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(0.25),

    tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(0.25),

    tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(0.25),

    # feature maps in a single vector
    tf.keras.layers.Flatten(),
```

Figure 100: Second model 1

```

    # dense
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.5),

    #2 stands for x,y coordinates
    tf.keras.layers.Dense(2, activation='relu')
])
```

Figure 101: Second model 2

With the addition of Dropout layers, the model appears to perform slightly better because they enhance the network's power, leading to better generalization. However, its performance has not yet met expectations.

4.1.3 Third model

The last model presented, in Figures 103/104, features additional layers, which contribute to improved in the recognition of the circles and rectangles centers. The additions includes the introduction of more powerful and numerous layers, each designed to refine the model's capability to process the image effectively. Here's a breakdown of the additions and its contribute to the model's performance:

- Adding two tf.keras.layers.Conv2D() layers, each with 256 filters, to the final stage of the network, replacing the previous layer with 128 filters, enhance the model's capacity to extract complex features from input.
- Adding tf.keras.layers.BatchNormalization() after each convolutional layer improves the training speed and stability. Batch normalization processes the inputs normalizing them, means adjusting the inputs to have a mean of 0 and a variance of 1. It helps ensure that each layer receives data on similar scale.

```
model = tf.keras.models.Sequential([

    # at the beginning the cnn captures contour and edges then simple filter like 32
    # in the next layers in order to capture complex objects it use high filter like 256
    # (filter, filter dimension, activation, image dimension and channel)
    tf.keras.layers.Conv2D(32, (3, 3), activation='relu', input_shape=[800, 600, 1]),
    tf.keras.layers.BatchNormalization(),
    # (pooling size)
    tf.keras.layers.MaxPooling2D((2, 2)),
    # randomly sets input units to 0 during training time, preventing overfitting
    tf.keras.layers.Dropout(0.25),

    tf.keras.layers.Conv2D(64, (3, 3), activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(0.25),

    tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(0.25),
```

Figure 102: Third model 1

```
    tf.keras.layers.Conv2D(128, (3, 3), activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(0.25),

    # ADDED to improve
    tf.keras.layers.Conv2D(256, (3, 3), activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(0.25),

    tf.keras.layers.Conv2D(256, (3, 3), activation='relu'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(0.25),

    # feature maps in a single vector
    tf.keras.layers.Flatten(),

    # dense
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.5),
```

Figure 103: Third model 2

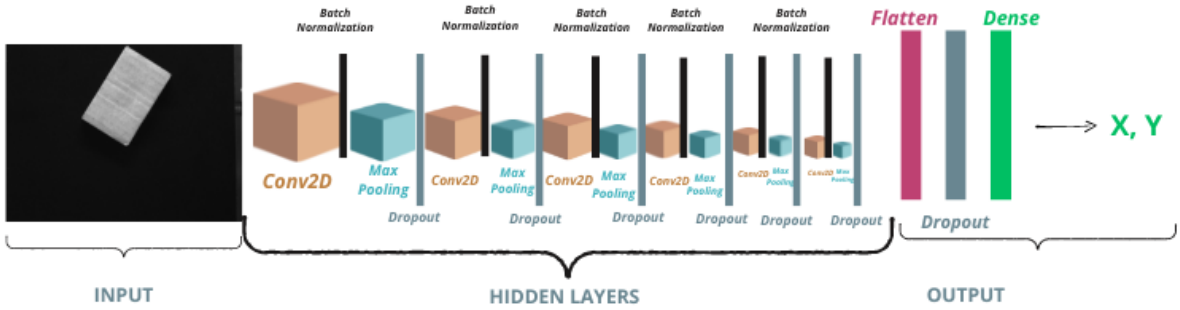

Figure 104a: Third model scheme

The outputs are the x and y coordinates of the center.

In conclusion, the addition of new layers enables the model to learn more detail patterns, which will perform better in terms of coordinate precision.

4.4 Training

At this point, we are going to conclude the preparation process of the CNN by setting additional parameters useful for future training with the method model.compile (Figure 105):

- Optimizer: helps to minimize the loss function. In this specific situation, Adam optimizer is considered, which effectively minimizes the loss function during the training of neural networks. The learning rate is an important hyperparameter, lower learning rate provide stable convergence but result in slower training, instead higher such as in this code speed up training.
- Loss: helps to understand whether the model is performing better or worse over time. Generally, for scenarios involving two or more label classes, cross-entropy is used. The loss used is 'mean_squared_error', common in regression tasks like this. It measure the average of the squares of the errors, which is the average squared difference between the estimated values and the actual value.
- Accuracy: is a metric to measure the percentage of correctly predicted instances compared to the instances in the dataset.

```
    # ADDED to improve
    tf.keras.layers.BatchNormalization(),

    #coordinates, from classification to regression
    tf.keras.layers.Dense(2)
])


model.compile(optimizer=Adam(learning_rate=0.1),
              loss='mean_squared_error',  # Modificato per regressione
              metrics=['mse'])


# Reduce learning rate when a metric has stopped improving
reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.1,
                              patience=10, min_lr=0.00001, verbose=1)

early_stopping = EarlyStopping(monitor='val_loss', patience=5, verbose=1)

# Training
# batch size is the # of examples passed through network for single iteration, better large but limitation for the memory
# epochs # number of times the dataset pass thorugh the network, more epochs more learning. attention to overfitting
model.fit(X_train, y_train, batch_size=32, epochs=20, validation_data=(X_test, y_test), callbacks=[reduce_lr, early_stopping])

```

Figure 104: Training the model

The method model.fit() trains the neural network using the training data generated in the Figure 96 (80% for X_train, 20% for X_test). The training occurs over a specific number of epochs, in this case 20.

An epoch is the passage of the entire dataset through the network. Generally a greater number of epochs improves the results at the end of the training.

The validation set is a collection of data completely new to the model; the images contained should have the same dimensions and color model as those used in the training set. The idea is to compare the predicted labels with the true labels. Then, with the validation set, it is possible to validate the results obtained during the training.

There is a procedure designed to improve performance during training called reduce_lr, which reduces the learning rate when a metric has stopped improving.

To determine if the network performs well, two images saved from the camera used in the laboratory will be uploaded such as in Figure 106. These images are uploaded using files.upload(), and are resized to match the dimensions of the images passed to the network for the training. This ensure consistency in input size, which is crucial for the model to process the images effectively.

```
[ ]  1 # test with loaded images
     2 uploaded = files.upload()
     3
     4 # Load, convert to gray, normalize and resize images
     5 test_images = []
     6 for filename, content in uploaded.items():
     7     image_stream = BytesIO(content)
     8     image = Image.open(image_stream).convert('L')
     9     image = image.resize((800, 600))
    10     image_array = np.array(image)
    11     image_array = image_array.astype('float32') / 255.0
    12     test_images.append(image_array)
    13
    14 test_images = np.array(test_images).reshape((-1, 600, 800, 1))
```

```
Scegli file  Nessun file selezionato    Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.
Saving ml10.PNG to ml10.PNG
Saving o.png to o.png
Saving ml2.PNG to ml2.PNG
Saving ml3.PNG to ml3.PNG
Saving ml4.PNG to ml4.PNG
Saving ml5.PNG to ml5.PNG
Saving ml6.PNG to ml6.PNG
Saving ml7.PNG to ml7.PNG
Saving ml8.PNG to ml8.PNG
Saving ml9.PNG to ml9.PNG
```

Figure 105: upload test images

4.5 Results in Google Colab

The method model.predict is used to make a prediction using the trained model.

```
[ ]  1 #predict the class for the test images
     2 predictions = model.predict(test_images)
     3 predicted_classes = np.argmax(predictions, axis=1)
```

Figure 106: Prediction of the model

By utilizing the plt library from matplotlib, it is possible to display the test images with their predicted coordinates x and y as shown in green in the next figures.

```
[ ]  1 predicted_offsets = []
     2
     3 for i, (img, label) in enumerate(zip(test_images, labels)):
     4     #true_x, true_y = label[0], label[1]
     5     predicted_x, predicted_y = predictions[i][0], predictions[i][1]
     6     #predicted_offsets.append((predicted_x - true_x, predicted_y - true_y))
     7
     8     plt.imshow(img.reshape(600, 800), cmap='gray')
     9     #plt.scatter([true_x], [true_y], color='red', label='True Center')  # True center
    10     plt.scatter([predicted_x], [predicted_y], color='green', label='Predicted Center')  # Predicted center
    11     plt.legend()
    12     plt.show()
```

Figure 107: Display the center predicted

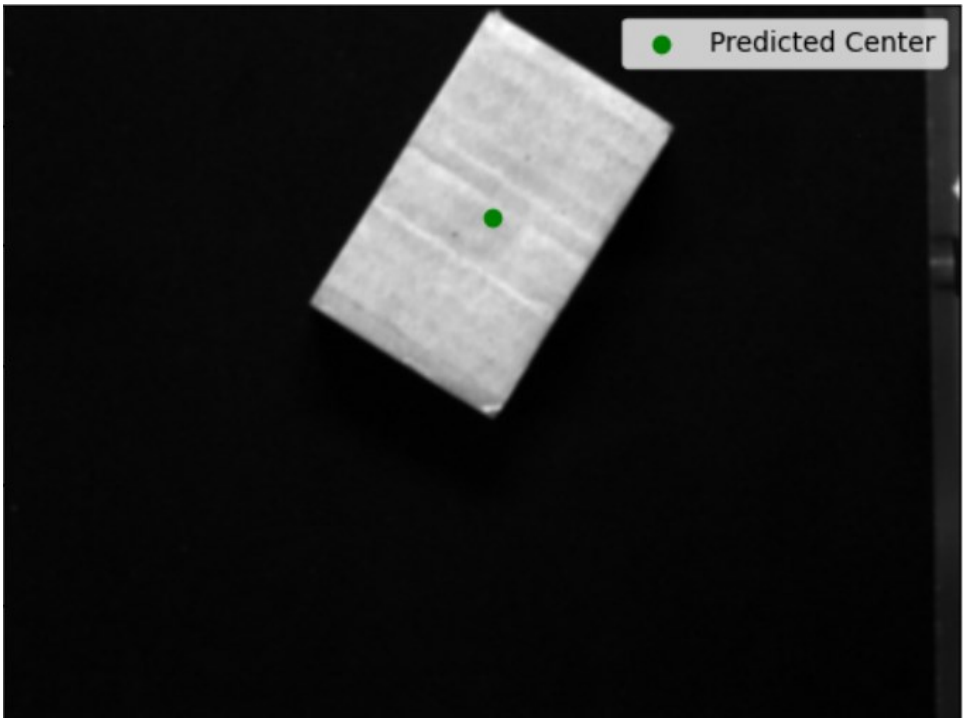

Figure 108: Predicted center 1

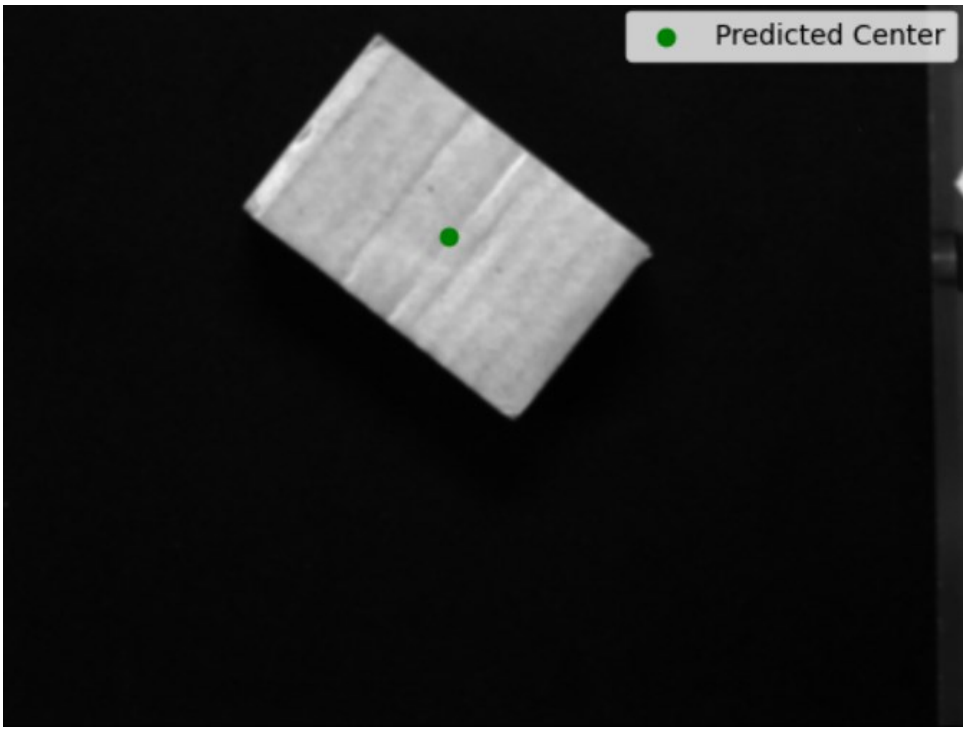

Figure 109: Predicted Center 2

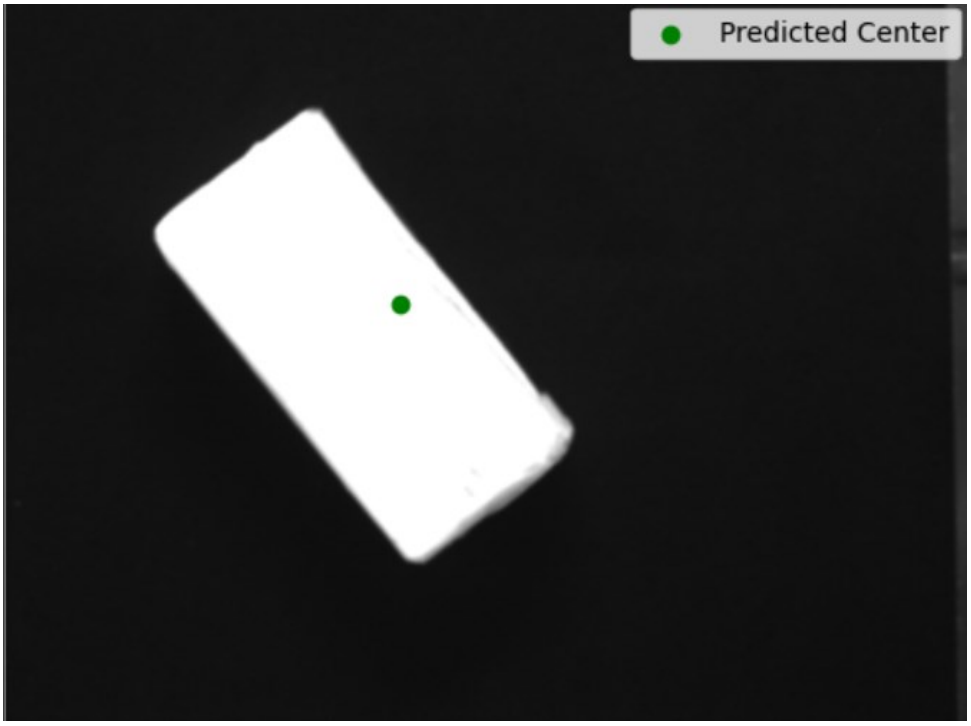

Figure 110: Predicted center 3

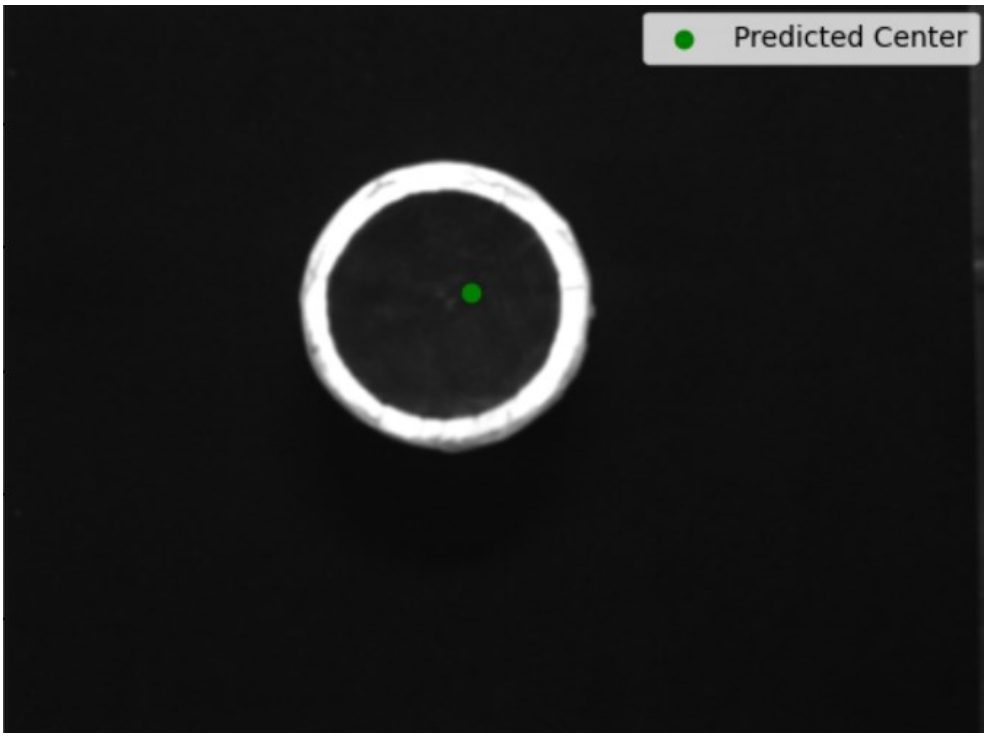

Figure 111: Predicted center 4

Finally, the trained model is saved (Figure 113) and converted to be ONNX format (Figure 114) for loading into TwinCAT.

```
[ ]  1 model.export('/content/model1')
```

Figure 112: Save the model

```
[ ]  1 import tf2onnx
```

```
[ ]  1 !python -m tf2onnx.convert --saved-model /content/model1 --output /content/model11.onnx
```

Figure 113: Convert in onnx format

4.6 Integration in TwinCAT

The final step was integrating the previously developed model into the TwinCAT environment.

Unfortunately, there is no existing package online that facilitate this process, only a Beta Version, though incomplete and lacking a proper documentation.

To use this Version, the installed TwinCAT version was updated from 4024.3 to 4024.6. This update was necessary because only from version 4024.6 is it possible to load machine learning models.

In this Beta Version, it is possible to:

- Read the model with an ONNX extension.
- Preprocess the image in the same manner as those used for network training:
    - Resize the image to 800, 600, matching the camera's resolution.
    - Convert the image into a 4D tensor, formatted as [number of images used, 800, 600, 1] where 1 represents the number of channels.
    - Scale the image by dividing by 255.
- Pass the image through the network to determine the coordinates of the object's center in the image, a process known as Inference.
- The last passage involving extracting results from the image output by the inference process. However, in this Beta Version, there is no existing function to facilitate this extraction. The only available function is suited for classification tasks, not for regression tasks where the output continuously changes.

Chapter 5: Experiments with the robot

The experiments were conducted by integrating the camera vision system with a robotic project, introducing two new modules in the TwinCAT environment:

- Drive manager module: responsible for motor configuration. After scanning, all detected motors are automatically added to I/O module. In this project, there are five motors: four for the robot and one external motor.
- Motion module: provides solutions for various tasks such as kinematic transformations, planar motion and pick-place operations. Following the drive manager's configuration, the axes are automatically defined in the environment. The cartesian axes, however, are manually calculated. This section highlights the coordinates of the conveyor belt with the respect of the robot system.

The MAIN function acts as a state machine to ensure an ordered code execution and to follow the flow of the robot's operations step by step.

The states are the following:

- State 0: is a transition state to State 10, which can be considered as a check state.
- State 10: involves the activation of the axes. The MAIN sends a request for the activation of Axis 1. This process is repeated for all the considered axes.
- State 20: the robot waits for a command. After each state, the machine returns to this state and waits for the next operation.
- State 30: the robot should reach its home position.
- State 40: the robot moves to an intermediate position where it awaits the coordinates of the object.
- State 50: the conveyor is activated and managed as another machine state:
    - State 20: is an initialization of the conveyor.
    - State 30: the velocity setting is defined here; it starts the movement.
    - State 40: is a waiting state. The conveyor waits until an object passes through the photocell. If no object is detected, the system will remain in this state.
    - State 50: when an object is detected, synchronization between the robot and conveyor occurs in order to pick the object.
    - State 60/70: the robot picks up the object.
    - State 80/90: the robot places the object in a specific position.
    - State 100: there is a return to waiting state.
- State 80: is an emergency state, in case of problems, all the axes will be stopped.

This state machine configuration ensures that the robot operates smoothly and effectively, managing tasks sequentially while maintaining synchronization between the conveyor and robotic arms.

5.1 Coordinate systems

Each component of the project, such as the camera, the robot and the photocell, has its own coordinate system, defined based on the position of the work cell. These coordinate systems are established with the respect to a common reference point in the work cell, seeing from the perspective of the computer.

5.1.1 Robot's system

The robot's reference system is centered around the base of the robot, which serves as the reference point or origin for all its movements and operations.

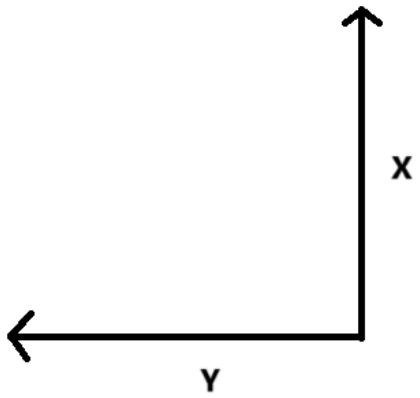

Figure 114: Robot's reference system

5.1.2 Photocell system

This system is rotated 90 degrees with the respect to the robot's base system.

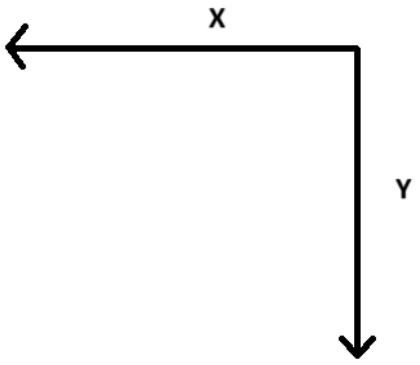

Figure 115: Photocell's reference system

5.1.3 Camera system

The camera reference system axes and the photocell reference system axes are aligned, but the camera reference system center is translated from the photocell reference system by 45 mm both in the x and y direction.

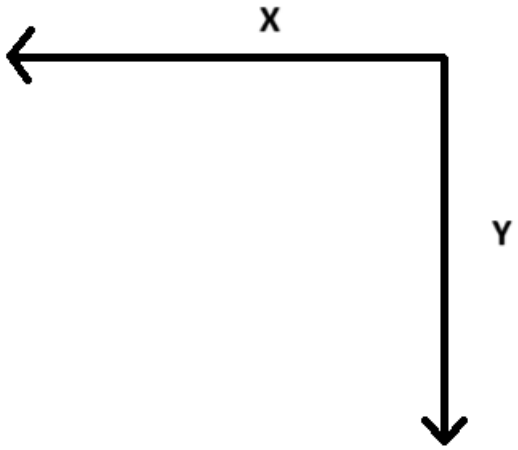

Figure 116: Camera's reference system

5.2 Conveyor velocity

Detection occurs while the conveyor is in motion. Therefore, to obtain high definition images that are not blurred, it is appropriate to calculate the correct velocity in relation to the camera's exposure time.

The conveyor velocity is defined in the activation part with ConvVelocity such as in Figure 118:

```
69  fbPower( Axis := GVL_Axis.conveyor,
70           Enable := TRUE,
71           Enable_Positive := TRUE,
72           Enable_Negative := TRUE,
73           Override := 100);
74
75  IF fbPower.Status THEN
76              ConvVelocity := 40; //20
77              bMoveConveyor := TRUE;
78                  IF moveConveyor.Active THEN
79                      bMoveConveyor := FALSE;
80                      stato_prova := 40;
81                  END_IF
82  END_IF
```

Figure 117: Setting conveyor

The exposure time (or shutter speed) is the time span for which the film of a traditional camera or a sensor of a modern digital camera is actually exposed to the light so as to record a picture.[17]

If the shutter speed is slower(1/500), less light enters the camera, resulting in a clearer image. Conversely, if the shutter speed is faster(1/2) the image may appear blurred (Figure 119).
In this case, the relationship between the conveyor and the shutter speed is: a slow conveyor corresponds to a slow shutter speed, and a fast conveyor corresponds to a fast speed.

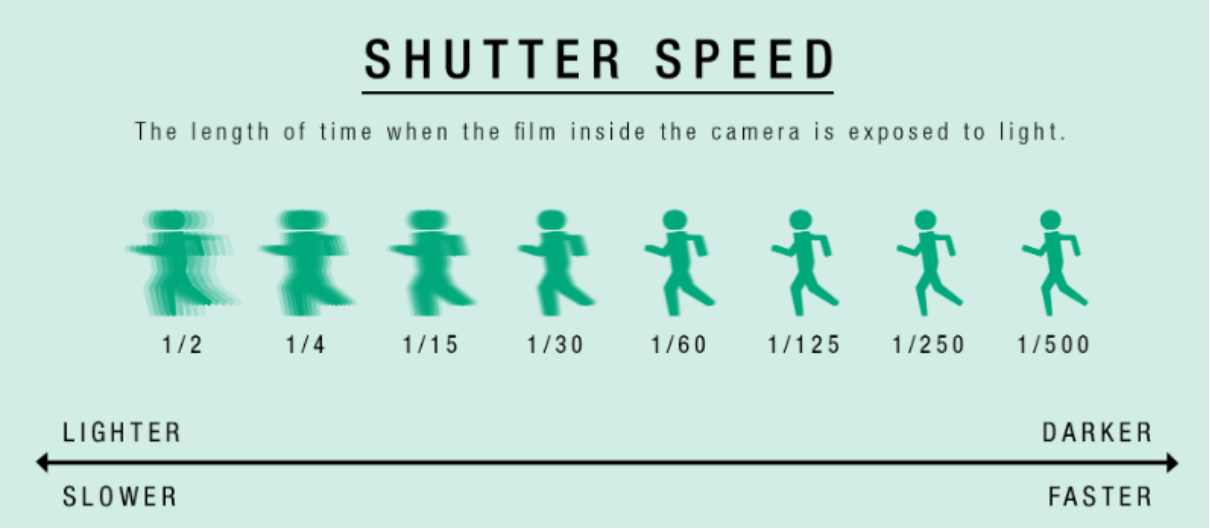

Figure 118: Exposure time example

Given these considerations, adjustments to the exposure time of the camera can be made in the Camera Assistant with the following settings:

- ExposureAuto: should be turned Off, t prevent the camera from automatically adjusting the exposure time during each run.
- ExposureMode: should be set to Timed, by seleceting this mode, you can manually set the exposure time to a specific duration.
- ExposureTimeAbs: is the absolute exposure time.

In this project, with the conveyor velocity set at 40 m/s the exposure time, Figure 120, is of 50354 microseconds(higher exposure). The makoG192B camera can handle higher blur distances, thus a higher exposure time.

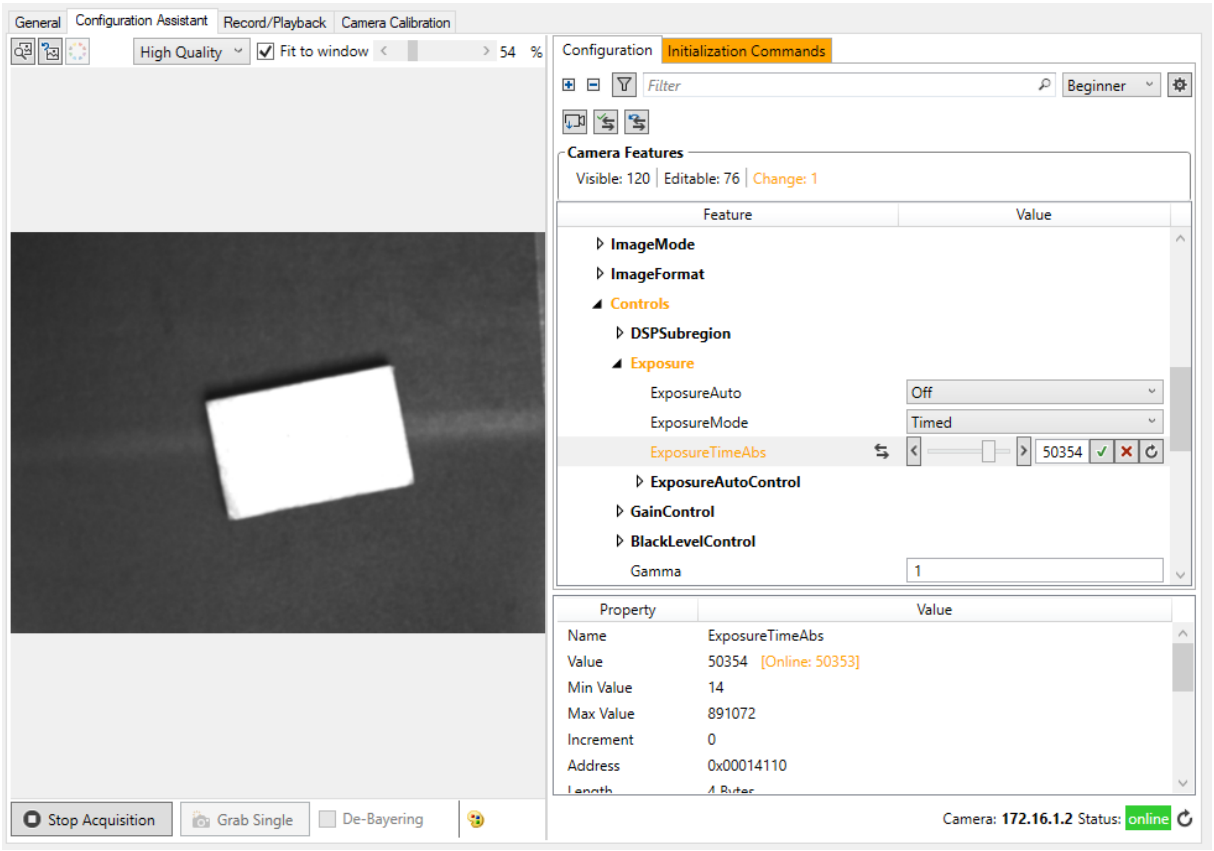

Figure 119: Configuration Assistant, ExposureTime

5.3 Coordinate transformation

The coordinates obtained from ObjectDetection(), see Figure 56 and 60, and transformed by F_VN_TransformCoordinatesImageToWorld_Point() are saved as centerDest. Since centerDest is in camera system, they must be translated with the respect to the photocell. The x-coordinate and y-coordinate are calculated by adding the distance in mm between photocell center and camera center.

InitialObjectPos1 (Figure 121) is external to the state machine, but its value is updated each time an object is detected by the sensor.

```
362   InitialObjectPos1[1] := centerDest[0]+40; // centro fotocellula + centro camera
363   InitialObjectPos1[2] := centerDest[1]+45; //centro fotocellula - centro caamera
364   InitialObjectPos1[3] := 0;
365   InitialObjectPos1[4] := 0;
```

Figure 120: Center of the object after transformation and translation

The position retrieved and saved in InitialObjectPos1 will be passed to mcTrackConveyorbelt (Figure 122), which is now able to track the object's position in real time.

```
374  mcTrackConveyorbelt(
375              AxesGroup := AxesGroupScara,
376              ConveyorBelt := GVL_Axis.conveyor,
377              Execute := bStartTracking,
378              CoordTransform := GVL.oidConveyorBelt,
379              InitialObjectPos := ADR(InitialObjectPos1),
380              InitialObjectPosCount := 4,
381              MasterRefPos := MasterRefPos,
382              Velocity := MC_DEFAULT,
383              Acceleration := MC_DEFAULT,
384              Deceleration := MC_DEFAULT,
385              Jerk := MC_DEFAULT);
```

Figure 121: Track the object from InitialObjectPos1

In conclusion, using a gripper, the object will be picked up from the detected position and placed in the specified location, in this case the table near the conveyor. This is achieved using a 'open'/'close' mechanism. If 'open', the gripper picks up the object; then it closes to secure the object, and upon reaching the specifies location, it opens again to release the object.

Conclusion

This thesis has effectively demonstrated the fundamental role of computer vision in modernizing industrial processes through the use of TwinCAT's integrated environment. The real-time communication enabled by TwinCAT has significantly enhanced operational efficiencies across various manufacturing tasks. The seamless interaction between hardware and software facilitated by this cohesive framework has proven essential for optimizing automation processes, reducing error rates, and increasing overall productivity.

TwinCAT Vision, in particular, has been pivotal in realizing the potential of real-time capable image processing within the TwinCAT 3 runtime environment. The synchronous execution of image processing algorithms with the control system, alongside the ability to run these algorithms in parallel on multiple cores, has minimized processing delays and maximized computational efficiency.

A significant aspect of this thesis focused on the application of object detection in an industrial setting. The precision and reliability with which TwinCAT Vision handled real-time object recognition on conveyor belts highlighted the transformative potential of integrating advanced vision technologies with robotic systems. This integration not only improved the throughput of the production line but also enhanced the quality control mechanisms, ensuring that only items meeting the required standards progressed further in the production process.

In conclusion, the adoption of TwinCAT and TwinCAT Vision within industrial systems represents a robust approach to leveraging advanced computer vision technologies. This integration promises to streamline production lines and elevate the standards of industrial automation, making it a crucial step towards the future of manufacturing efficiency.

Images

Figure 1: Convolutional Neural Network 12
Figure 2: Classification or Regression models 13
Figure 3: Creation of digitized image 15
Figure 4: Pinhole model 16
Figure 5: FoV and focal length 16
Figure 6: Radial and Tangetial distortion 17
Figure 7: XAE and XAR in TwinCAT 3 20
Figure 8: Modules defined in TwinCAT 21
Figure 9: Router Memory, Available Cores (Shared/Isolated) 22
Figure 10: PLC structure and connections 23
Figure 11: Create new project in Visual Studio with integrated TwinCAT 3 24
Figure 12: Add library in References 25
Figure 13: Tc3_Vision is added to the project 26
Figure 14: Vision device 27
Figure 15: Add images in File Source Control 27
Figure 16: Add new item in VISION 30
Figure 17: Computer linked to adapter, adaapter linked to camera 31
Figure 18: Ethernet 31
Figure 19: Property 32
Figure 20: Discover Devices 32
Figure 21: I/O 33
Figure 22: Ethernet informations 33
Figure 23: Camera Assistant Configuration 35
Figure 24: Camera Assistant, Trigger Selector 36
Figure 25: Camera Calibration, load image and calibrate intrinsic and extrinsic 37
Figure 26: Camera Calibration result 38
Figure 27: Image Provider 39
Figure 28: Symbol Initialization 39
Figure 29: Camera state machine 40
Figure 30: ERROR, INITIALIZED, OPENED, START_ACQUISITION 41
Figure 31: ACQUIRING, STOP_ACQUISITION 42
Figure 32: Threshold, Gaussian, Median, HistogramEqualization, MorphologicalOperator 43
Figure 33: Sobel, Laplacian, Bilateral 44
Figure 34: GaussianFilter 45
Figure 35: MedianFilter 45
Figure 36: MorphologicalOperator 45
Figure 37: HistogramEqualization 46
Figure 38: SobelFilter 46
Figure 39: LaplacianFilter 46
Figure 40: BilateralFilter 47
Figure 41:CopyIntoDisplayableImage 47
Figure 42:TransformIntoDisplayableImage 47
Figure 43: ObjectDetection() first part, Blob detection, check contours and circularity, approximate to polygon and draw contours 48
Figure 44: DetectBlobs in the image 49
Figure 45: GetNumberOfElements in the container 49
Figure 46: GetAt_ITcVnContainer, get one element 49
Figure 47: Circularity of each contour 49
Figure 48: Draw found contour 50

Figure 49: Approximation 50
Figure 50: ObjectDetection, second part 50
Figure 51: Extreme point of the contour 51
Figure 52: DrawPointExp 51
Figure 53: DrawLine 51
Figure 54: Conversion in another color space 51
Figure 55: ObjectDetection, third part 52
Figure 56: ObjectDetection, fourth part 52
Figure 57: Trasformation reference systems 53
Figure 58: Put label on image 53
Figure 59: Calculation center of the circle 53
Figure 60: ObjectDetection, fifth part 54
Figure 61: ObjectDetection, sixth part 54
Figure 62: ADS Image Watch 54
Figure 63: Toolbar 55
Figure 64: Other Windows 55
Figure 65: Rectangle 56
Figure 66: Rectangle 2 rotated 56
Figure 67: Rectangle 3 56
Figure 68: Rectangle out of the ROI 57
Figure 69: Circle 1 in the center of ROI 57
Figure 70:Circle 2 in the left part of ROI 57
Figure 71: Match Template function 58
Figure 72: Template algorithm implementation 59
Figure 73: Edge detector 60
Figure 74: Algorithm with Canny Edge Detector 61
Figure 75: Choose a runtime between CPU, GPU, TPU 62
Figure 76: Import necessary libraries 63
Figure 77: Run all the cells in the notebook 63
Figure 78: test if the cell is executed with GPU 64
Figure 79: Generate circles function 1 64
Figure 80: Generate circles function 2 65
Figure 81: Generate rectangles function 1 65
Figure 82: Generate rectangles function 2 66
Figure 83: Generate images and labels for circles and rectangles 66
Figure 84: Show the centers of both shapes 67
Figure 85: Shuffle images and labels 67
Figure 86: Circle with larger random radius 68
Figure 87: Circle with smaller random radius 68
Figure 88: Rectangle in the image's center 69
Figure 89: Rectangle in the bottom left of the image 69
Figure 90: Data augmentation first part 70
Figure 91: Data augmentation second part 70
Figure 92: Rectangle rotation 1 71
Figure 93: Rectangle rotation 2 71
Figure 94: Load the dataset from CSV file 72
Figure 95: Concatenation 72
Figure 96: Split the dataset 72
Figure 97: CNN structure 73
Figure 98: Example of Max Pooling 73
Figure 99: First model 1 74
Figure 101: Second model 1 76

Figure 102: Second model 2 76
Figure 103: Third model 1 77
Figure 104: Third model 2 77
Figure 105: Training the model 78
Figure 106: upload test images 79
Figure 107: Prediction of the model 79
Figure 108: Display the center predicted 80
Figure 109: Predicted center 1 80
Figure 110: Predicted Center 2 80
Figure 111: Predicted center 3 81
Figure 112: Predicted center 4 81
Figure 113: Save the model 82
Figure 114: Convert in onnx format 82
Figure 115: Robot's reference system 84
Figure 116: Photocell's reference system 84
Figure 117: Camera's reference system 85
Figure 118: Setting conveyor 85
Figure 119: Exposure time example 86
Figure 120: Configuration Assistant, ExposureTime 87
Figure 121: Center of the object after transformation and translation 87
Figure 122: Track the object from InitialObjectPos1 88

Sitography

[1] https://www.sas.com/en_us/insights/analytics/computer-vision.html#:~:text=Computer%20vision%20is%20a%20field,to%20what%20they%20%E2%80%9Csee.%E2%80%9D

[2] https://www.automation.com/en-us/articles/march-2023/understanding-role-machine-vision-industry-4

[3] https://www.britannica.com/technology/artificial-intelligence

[4]https://en.wikipedia.org/wiki/Pinhole_camera_model#:~:text=The%20pinhole%20camera%20model%20describes,are%20used%20to%20focus%20light.

[5] https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/

[6] https://www.ibm.com/topics/computer-vision#:~:text=Computer%20vision%20is%20a%20field,they%20see%20defects%20or%20issues.

[7] https://www.unitronicsplc.com/what-is-plc-programmable-logic-controller/

[8] https://www.paessler.com/it-explained/plc

[9] https://itsmaker.it/wp-content/uploads/2018/04/1-TwinCAT-3-Overview-IT-1.pdf

[10] https://www.beckhoff.com/it-it/

[11] https://www.iqsdirectory.com/articles/automation-equipment/industrial-robots.html#:~:text=Common%20applications%20of%20industrial%20robots%20are%20product%20assembly%2C%20machine%20loading,painting%2C%20coating%2C%20and%20inspection.

[12] https://www.educative.io/answers/what-is-canny-edge-detection

[13] https://www.futurelearn.com/info/courses/introduction-to-image-analysis-for-plant-phenotyping/0/steps/297750

[14] https://medium.com/@abhishekjainindore24/data-augmentation-00c72f5f4c54

[15] https://www.nature.com/articles/s41598-024-51258-6#:~:text=Max%20pooling%20is%20a%20commonly,each%20small%20window%20or%20region.

[16] https://computersciencewiki.org/index.php/File:MaxpoolSample2.png

[17] https://www.smartray.com/glossary/exposure-time/#:~:text=The%20exposure%20time%2C%20respectively%20period,time%20is%20given%20in%20seconds.

[18] https://shotkit.com/field-of-view/

[19] https://www.alliedvision.com/en/camera-selector/detail/mako/g-192/

[20] https://www.geeksforgeeks.org/calibratecamera-opencv-in-python/
[21] https://www.javatpoint.com/regression-vs-classification-in-machine-learning
[22] https://www.beckhoff.com/it-it/products/automation/twincat/tfxxxx-twincat-3-functions/tf7xxx-vision/tf7100.html?