Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Control Systems Engineering

3D Data Processing with Deep Learning and Industrial Applications

Master Candidate

Emanuele Bano

Student ID 2090211

Supervisor

Prof. Giovanni Boschetti

University of Padova

Academic Year 2023-24

10 October 2024

Alla mia famiglia

CONTACTS

☞ emanuelebano@gmail.com

ABSTRACT

In this thesis I explore the processing of 3D data and its industrial applications, utilizing both traditional computer vision techniques and modern methods based on deep learning. The ability to sense, perceive, and interpret the surrounding environment by a computer is a challenging task that requires a mathematical framework. While most research has historically focused on 2D data, the recent availability of more affordable 3D sensors and the advancement of powerful deep learning tools have made it possible to tackle tasks that were previously out of reach with standard 2D techniques.

The thesis is divided into three parts. The first part provides an overview of the theory and methods that form the foundation of the applications developed in the subsequent parts. It begins with techniques and sensors for acquiring 3D data, followed by a discussion on the different ways to represent this information. It then delves into high-level 3D computer vision tasks, covering both traditional approaches as well as modern techniques using deep learning networks.

The second part presents a deep learning application that I developed to address a 3D classification task. The network architecture is inspired by the Orientation Boosted Voxel Net, where the network is trained to learn object rotations as an auxiliary task using a combined categorical cross-entropy loss function. The novelty of my design lies in the complete redefinition of the architecture, where I employed skip connections to enable a deeper network, thereby avoiding vanishing gradient problems and facilitating more abstract and effective feature extraction. The full implementation of the dataset, model, network training, and testing was carried out in Python.

The third part of the thesis demonstrates the application of the methods discussed for the design of an industrial system that I developed during my internship at Innova Srl. The aim was to create a general module capable of acquiring point clouds of objects moving on an industrial conveyor belt, followed a specific processing module. An example application is detecting the 3D pose of bread on the conveyor to guide a robotic arm in making cuts that improve cooking properties. A key innovation of the data acquisition module was the replacement of the conventional setup of multiple static 3D profilometers with a new system that utilizes just two profilometers moving perpendicularly to the belt's velocity. This approach significantly reduces costs while introducing challenges related to distortion caused by the relative movement between the profilometers and the objects, as well as the stitching of subsequent scans. The processing component of the application was developed using the MVTec Halcon programming language and was integrated into a unified solution in C# that also manages communication with the sensors' controller. Finally, the thesis illustrates how the processed data from the acquisition module can be used for robotic guidance applications, where 3D surface matching algorithms detect the target object and its pose within the scene and transmit this information to a robotic arm to perform a specif action.

ACKNOWLEDGEMENTS

"At every moment of our life, we must be grateful
to the people who make our journey possible."
"In ogni momento della nostra vita dobbiamo essere grati
alle persone che rendono possibile il nostro cammino."

— William Arthur Ward

I want to start by expressing my heartfelt gratitude to Innova Srl for giving me the chance to work in such a collaborative and dynamic environment. A special thanks goes to Samuele Foscaro and my tutor, Giacomo Pradella, for their patience and invaluable advice throughout my internship. I also want to thank all my colleagues, whose knowledge and support were a constant source of inspiration.

Lastly, I am deeply thankful to Professor Giovanni Boschetti for his time and valuable guidance.

Vorrei poi ringraziare i miei genitori per il supporto incondizionato che mi hanno dato durante tutto il mio percorso, e per i sacrifici fatti senza mai farmeli pesare. Ringrazio anche il resto della mia famiglia, i miei fratelli Milena e Gabriele, e i nonni, presenti e passati. Ognuno di loro mi ha dato una parte di sé, aiutandomi a diventare una persona migliore.

Voglio poi ringraziare le persone che arricchiscono la mia vita, Francesca, gli amici incontrati lungo il mio cammino, Ludovica, Matteo, Giulio, Eli, Alessandra, Marco, Alberto e tanti altri. Contribuite ogni giorno a rendere le giornate più belle e divertenti.

CONTENTS

LIST OF FIGURES

LIST OF CODE SNIPPETS

1 INTRODUCTION

Most of the information humans obtain from the world is in the form of visual data. In fact, vision plays a crucial role in helping humans understand and interact with their environment. The task involves two primary processes: sensing the surrounding environment through the response to light, and perceiving and interpreting the visual information received.

Given the potential of using this type of data, a branch of engineering called Computer Vision has emerged, focusing on exploiting this data to enable vision in computers. However, unlike humans, solving vision tasks is complex for computers, as these tasks must be mathematically framed to be computable. This requires addressing fundamental challenges, such as the mathematical representation of data and its processing through transformations and gradual abstractions during continuous analysis. Additionally, the effectiveness of a computer vision system is significantly influenced by the type of data acquired and the sensors used.

Recent advancements in 3D sensors and 3D data acquisition systems have further increased the research, building on the success of 2D techniques to solve more complex tasks. The development and success of deep learning frameworks have also provided powerful tools to process large amounts of data quickly and with high performance. These advancements are opening up new possibilities across various industries that I intend to showcase in this thesis.

1.1 COMPUTER VISION PIPELINE

The goal of computer vision is to replicate human visual capabilities in order to make meaningful decisions based on the perceived information from visual data.

The typical engineering approach to this challenge involves using any feasible methods and techniques to achieve the desired output from the input data. In particular, both the input and output must be formulated mathematically, allowing any processing to be viewed as a mathematical transformation and thereby making it algorithmically implementable. A complete computer vision application is often divided into multiple steps, forming the pipeline illustrated in Figure 1.

The subsequent steps in a computer vision pipeline are as follows:

- **Data Acquisition**: The first step involves setting up a system to acquire raw data from the target scene. This includes selecting appropriate sensors, which may provide data directly usable by the application (e.g. cameras). In some cases, additional techniques might be necessary to integrate data from multiple sensors to produce a complete 3D information for the subsequent steps (e.g. stereo vision).

- **Data Storage and Representation**: Once the data is acquired, it must be stored in a computer. The method of storage and representation depends on the type of raw data and is crucial for the subsequent analysis and inference stages.In fact, choosing the right representation can enable high-level techniques used later in the pipeline, as it is particulary important in deep learning.
- **Pre-processing**: The stored data must be prepared to be used in the later stages of the pipeline. This step may involve enhancing the data quality, filtering to remove noise, or highlighting specific properties to make the data more suitable for analysis. In deep learning applications, this phase can also include data augmentation to avoid overfitting.
- **Inference and Analysis**: Algorithms are then applied to understand the content of the data and solve the initial tasks. This involves the computation and extraction of features from the data, which are then used by high-level algorithms. This can include traditional methods or modern machine learning techniques that automate the feature extraction and analysis process.
- **Post-processing**: The results from inference and analysis are typically further processed to take action based on the vision outcomes. This step ensures that the final output of the system is usable to perform control actions.

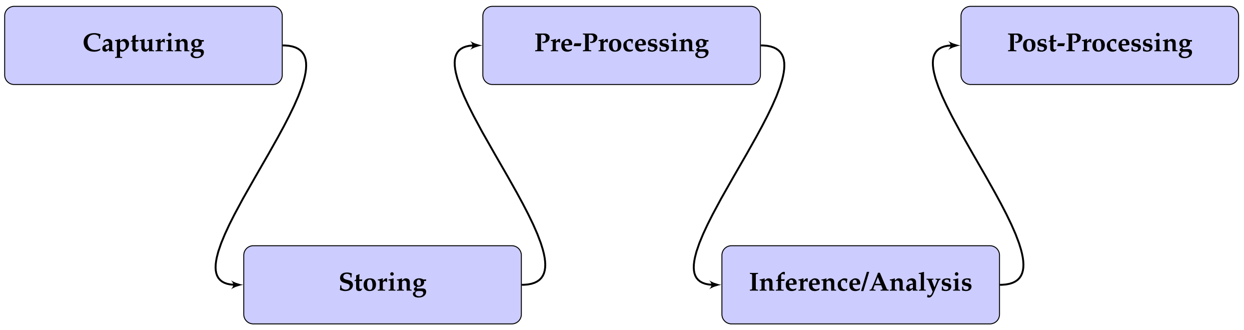


Figure 1: Computer Vision Pipeline

1.2 WHY 3D COMPUTER VISION: FROM 2D TO 3D

Until recently, most computer vision applications focused on 2D data. This is mostly because cameras, the most common sensors for acquiring visual data, project 3D information onto a 2D image plane, losing one dimension. Despite this limitation, 2D vision remains highly useful and is widely applied in various fields, especially in controlled settings where depth information is not needed for the task.

However, the spread of cheaper 3D sensors and algorithmic advancements in deep learning have expanded the possibilities of computer vision. With depth information, 3D computer vision can solve tasks that were not possible using 2D techniques, such as understanding the spatial position of objects and their relationships in space. This capability can significantly improve the accuracy, as 2D systems can be affected by occlusions and most importantly are more dependent on lighting conditions.

For industrial applications the spatial knowledge of objects in the environment is crucial for robotic systems and other automated processes. Nevertheless, the benefits of 3D vision are brought along with increased complexity and higher costs associated with advanced sensors and solutions. Therefore, it is essential to evaluate the need for 3D vision techniques at the start of a project.

	Advantages	**Disadvantages**
2D	• Optimal for planar objects. • Cost effective. • Simpler hence faster.	• Envirnment sensitivity. • Lack of depth information.
3D	• Adaptable to non-rigid objects. • Robust to illumination changes. • Depth information. • High degree of precision.	• Complexity. • Cost.

Table 1: Pros and Cons of 2D and 3D Computer Vision Systems.

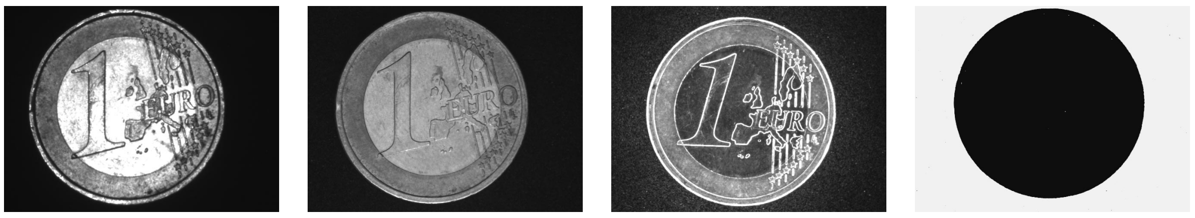
1.3 COMPONENTS OF A COMPUTER VISION SYSTEM

Whether working with 2D or 3D data, the main components of a computer vision system are generally similar. These systems typically include five key elements:

Lighting

Lighting is of critical importante in the success of a computer vision application. Much like human vision, objects must be well illuminated and clearly visible to effectively solve vision tasks. In many systems, lighting is also strategically used to enhance important features of the objects, making the processing tasks easier. An example of this is illustrated in Figure 2.

In 2D applications, changes in lighting can be detrimental to system performance, as they can significantly impact the clarity of edges and features. In contrast, 3D systems often use structured light or laser triangulation, making them more robust to lighting variations.



(a) *Front light.* (b) *Diffuse light.* (c) *Dark field.* (d) *Backlight.*

Figure 2: Example of lighting tecniques and their effect.

Optical Lens

Optical lenses are used to direct light in vision systems. They focus light onto the image sensor (Figure 3) in cameras or redirect light into the environment, as with line scan cameras. The choice of lens significantly impacts the field of view, depth of field, and image resolution, all of which are crucial for capturing high-quality images.

Image Sensor

Typically, a CMOS or CCD sensor is used to convert light into a digital image. In 3D systems, the sensor must also capture depth data, which may require specialized techniques such as time-of-flight or stereo vision.

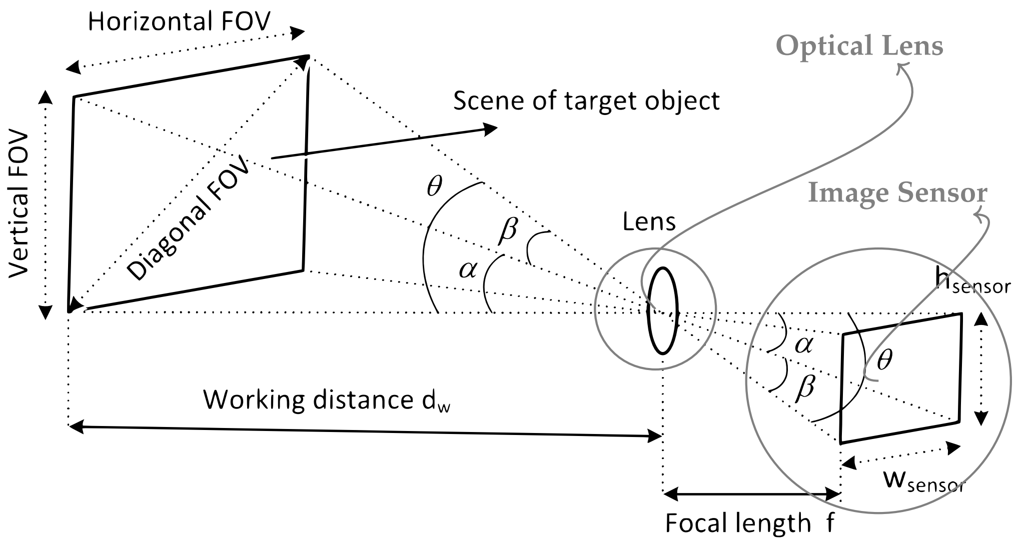


Figure 3: Lens and Image sensor in a camera system Ngo et al., 2019.

Processing Software and Computational Hardware

The processing software interprets the captured images using vision algorithms that manage the stages of storing, pre-processing, analysis, and post-processing, which solve tasks specific to the application like object identification, dimension measurement. In the context of 3D vision, this software must also address the added complexity of processing depth data. The selection of hardware should be precisely aligned with the computational demands of the application to meet the required time constraints.

Communication Interfaces

These interfaces enable communication between the vision system components, such as the connection between sensors and computational hardware, as well as allow the vision system to interact with other machines, robots, or control systems.

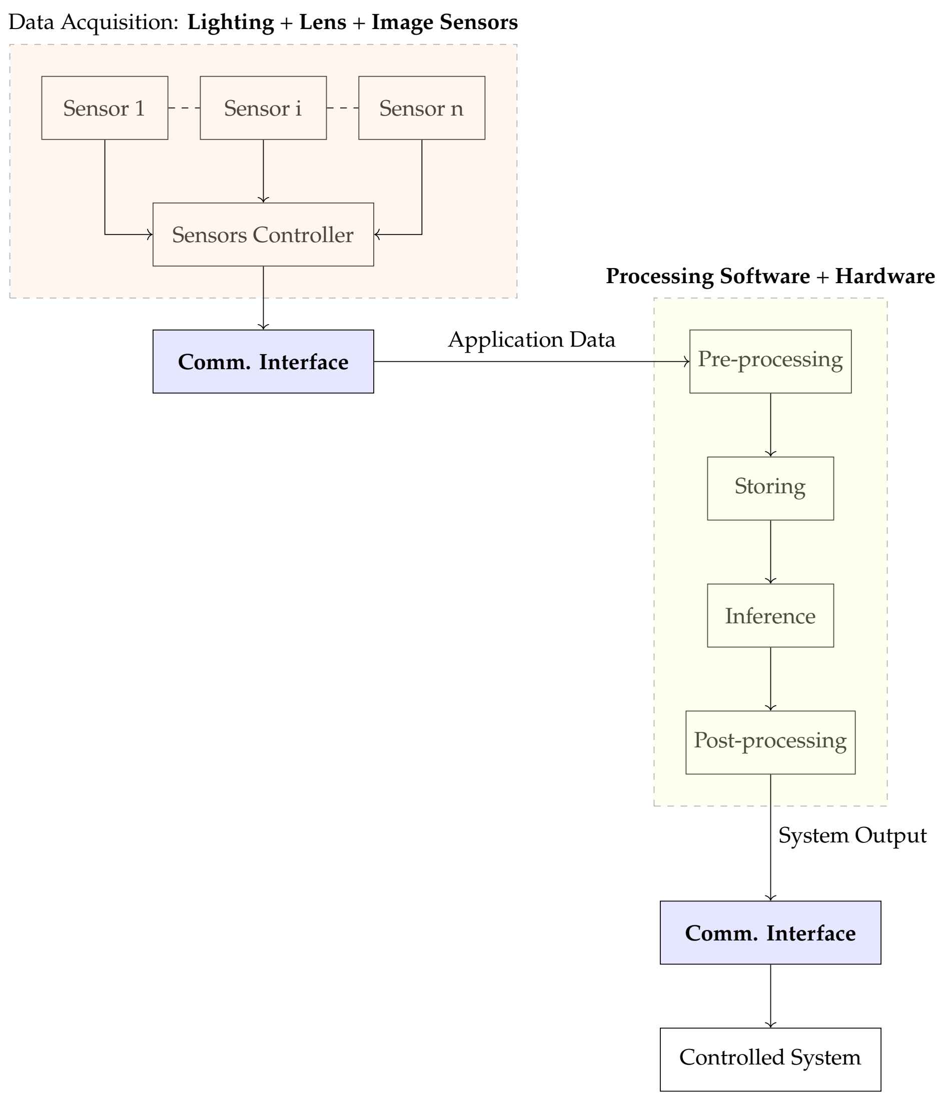


Figure 4: Block diagram illustrating the core components and their connections.

1.4 THESIS OUTLINE

The remainder of the thesis is structured as follows:

- In Chapter2 I review the concepts of 3D pose, the camera pinhole model, and calibration, which are the foundation of the developed applications. 3D data acquisition techniques and sensors used to capture the environment are then introduced, followed by an overview of some of the most commonly used representations for storing and processing this data.
- In Chapter 3 I discuss the transition from low-level perception of the scene to high-level understanding to solve typical tasks that are fundamental in the remainder of the thesis, such as 3D registration and matching. This step can be based on standard methods involving the design of handcrafted features or modern deep learning techniques, with examples provided for both approaches.

These two chapters are fundamental to explain the mathematical framework on which the applications designed in the following chapters are based.

- In Chapter 4 I present the design of a deep learning network for solving a 3D classification task. The design pipeline is explained in detail, focusing on the choices made regarding data augmentation, model architecture, loss function, and optimization algorithms used for training. Along with the explanation, I have provided key sections of the Python code used in the design.

The concepts introduced and the deep learning techniques discussed were crucial in the design of many projects I participated in during my internship at Innova Srl. The combination of these techniques was employed to design industrial applications. In the last part of the thesis, I present the end point of my intership with the design of a personal industrial application.

- In Chapter 5 I provide an overview of the MVTec Halcon programming language, which I utilized in the subsequent chapter. The focus here is on its high-level capabilities for solving industrial applications in the field of 3D machine vision.
- In Chapter 6 I detail the design of an industrial application for acquiring 3D point clouds of objects moving on a conveyor belt, followed by processing in an example application for robot guidance. I present a new design for the acquisition module, along with the challenges that were addressed. The steps of the application are explained in detail, including key code snippets used in the implementation.

Part I

FIRST PART: THEORETICAL BACKGROUND

This part discusses the concepts underlying the subsequent applications. Specifically, it is crucial to develop models of the data acquisition process and to mathematically frame the vision task. The first chapter presents the fundamental concepts of 3D pose, the pinhole camera model, and camera calibration, followed by a discussion of the key techniques for acquiring 3D data. Finally, the representation of this data in relation to subsequent processing is explored. The second chapter provides an overview of how to transition from raw data to a high-level understanding of the content. This is achieved by designing mathematical descriptors of the data that can be used to solve typical inference applications, such as 3D reconstruction and matching. At last, some modern deep learning techniques are discussed.

2 3D SHAPE ACQUISITION AND REPRESENTATION

The first step of a computer vision application involves data acquisition. In both 2D and 3D cases, the acquisition setup typically includes a camera, which may be the only device used to acquire data in the 2D case, or it may be paired with other components or cameras to form 3D sensors in the 3D case. It is then necessary to understand how to model and calibrate the camera, both to obtain the best quality data as possible as well as to determine how we can translate the results referred to an image back to the 3D coordinates of the scene. Before introducing these concepts, it is important to review the core concepts of 3D transformation and pose, and to standardize the notation used throughout the thesis.

2.1 3D TRANSFORMATIONS AND POSES

Given a 3D point P in space, we can represent it as a vector whose coordinates depend on the coordinate system we are considering. In many computer vision applications, we typically consider a world reference system denoted by w, and a camera coordinate system denoted by c (Figure 5).

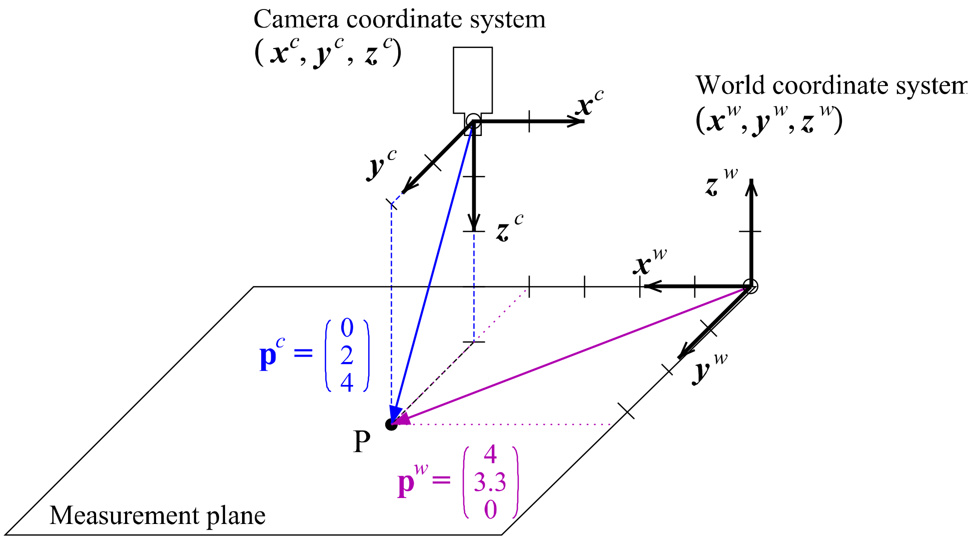


Figure 5: 3D coordinates w.r.t. the coordinate systems.

The 3D point can then be written as:

$$\mathbf{p}^c = \begin{pmatrix} x^c \\ y^c \\ z^c \end{pmatrix} \qquad \mathbf{p}^w = \begin{pmatrix} x^w \\ y^w \\ z^w \end{pmatrix}$$

During the processing of the data, we often need to transform the coordinate system and be able to change the representation of a point from one coordinate system to another. If we consider a coordinate system *c1* and its translated version *c2* from Figure 6, we can see that to obtain the representation of the point Q_2 in the first coordinate system, we can simply add the coordinates of the second coordinate system with respect to the first to the representation of the point in the second coordinate system:

$$\mathbf{q}_2^{c1} = \mathbf{q}_2^{c1} + \mathbf{o}_{c2}^{c1}$$

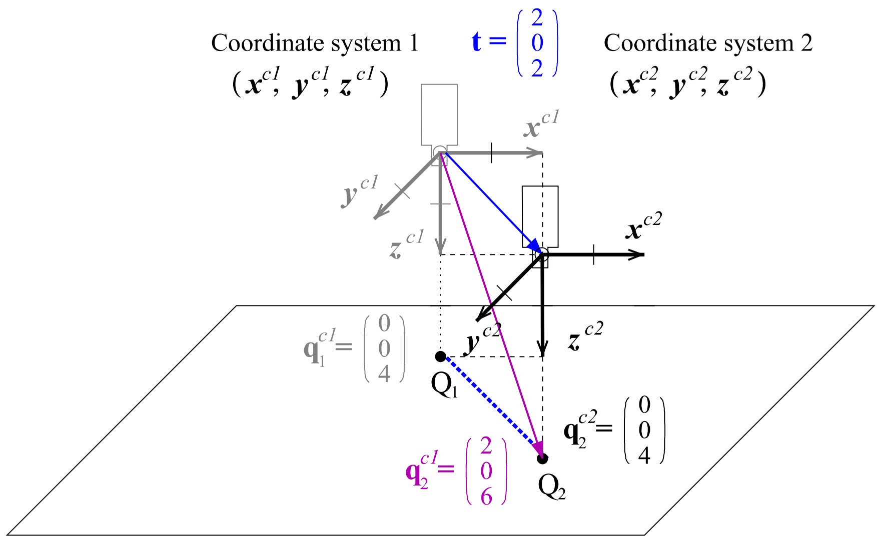

Coordinate system 1
(x^{c1}, y^{c1}, z^{c1})
$\mathbf{t} = \begin{pmatrix} 2 \\ 0 \\ 2 \end{pmatrix}$
Coordinate system 2
(x^{c2}, y^{c2}, z^{c2})
x^{c1}
y^{c1}
z^{c1}
x^{c2}
y^{c2}
z^{c2}
$\mathbf{q}_1^{c1} = \begin{pmatrix} 0 \\ 0 \\ 4 \end{pmatrix}$
Q_1
$\mathbf{q}_2^{c2} = \begin{pmatrix} 0 \\ 0 \\ 4 \end{pmatrix}$
$\mathbf{q}_2^{c1} = \begin{pmatrix} 2 \\ 0 \\ 6 \end{pmatrix}$
Q_2

Figure 6: Coordinate system translation.

Coordinate systems also have a relative rotation with respect to each other. We might then consider the rotated version of a system, as shown in Figure 7. Each rotation is represented by a corresponding rotation matrix. Considering the usual *c1* starting coordinate system and its rotated version *c3*, the rotation matrix is written as:

$$\mathbf{R}_{c3}^{c1} = \begin{bmatrix} \mathbf{x}_{c3}^{c1} & \mathbf{y}_{c3}^{c1} & \mathbf{z}_{c3}^{c1} \end{bmatrix}$$

where the columns are given by the coordinates of the axes of the rotated system with respect to the initial system.

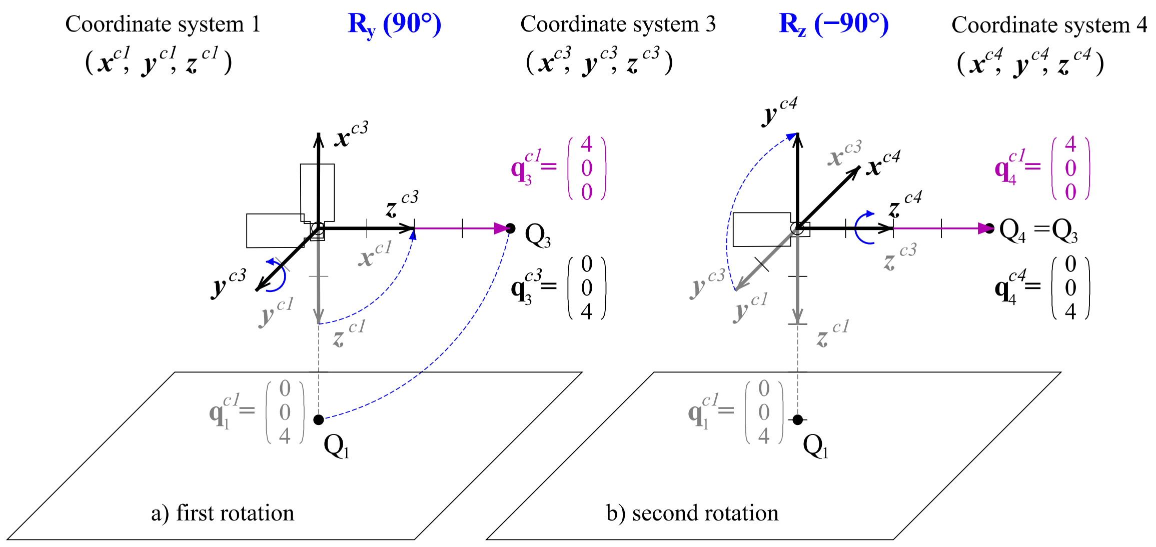


Figure 7: Coordinate system rotation.

A point Q_3 can be then represented in the first coordinate system by right-multiplying the representation in the rotated system by the rotation matrix:

$$\mathbf{q}_3^{c1} = \mathbf{R}_{c3}^{c1}\,\mathbf{q}_3^{c3}$$

Combining a translation and a rotation, we obtain a rigid transformation. The general expression for a point Q_4 in the transformed system $c5$ is then

$$\mathbf{q}_4^{c1} = \mathbf{R}_{c5}^{c1}\,\mathbf{q}_4^{c5} + \mathbf{o}_{c5}^{c1} \tag{1}$$

These expressions can become quite complex when combining multiple rigid transformations; hence, we typically introduce homogeneous coordinates and consider the corresponding homogeneous transformations. The key advantage of homogeneous coordinates is that they allow us to express rigid transformations as linear matrix functions, so that the change in the coordinate representation of points can be written as a matrix homogeneous linear system. In fact, Equation 1 can be rewritten as:

$$\underbrace{\begin{pmatrix} \mathbf{q}_4^{c1} \\ 1 \end{pmatrix}}_{\tilde{\mathbf{q}}_4^{c1}} = \underbrace{\begin{bmatrix} & \mathbf{R}_{c5}^{c1} & & \mathbf{o}_{c5}^{c1} \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{\mathbf{H}_{c5}^{c1}} \underbrace{\begin{pmatrix} \mathbf{q}_4^{c5} \\ 1 \end{pmatrix}}_{\tilde{\mathbf{q}}_4^{c5}} \tag{2}$$

where $\tilde{\mathbf{q}}_4^{c1}$ and $\tilde{\mathbf{q}}4^{c5}$ represent the homogeneous coordinates of the point with respect to the two coordinate systems, and $\mathbf{H}_{c5}^{c1}$ represents the homogeneous transformation.

Finally, this allows us to express the representation change given by n transformations as

$$\tilde{\mathbf{q}}^{c1} = \underbrace{\mathbf{H}_{c2}^{c1} \cdots \mathbf{H}_{cn}^{c(n\text{-}1)}}_{\mathbf{H}_{cn}^{c1}}\,\tilde{\mathbf{q}}^{cn}$$

Each transformation matrix has 12 parameters: 9 parameters for the rotation matrix and 3 parameters for the translation vector. However, the rotation matrix is a redundant representation, as a general rotation can be expressed as a concatenation of three rotations around the axes of the coordinate system. This means that we can represent a rigid transformation using only 6 parameters, forming the concept of 3D *pose*.

2.2 CAMERA MODEL AND CAMERA CALIBRATION

A camera is a device that projects a 3D scene onto a 2D image. The most used ideal model of this device in computer vision is the **pinhole camera model**, shown in Figure 8. In this model, the rays of light coming from the object pass through a point called the pinhole before being captured by an image sensor that forms the image. Of course, in reality, the pinhole model does not exist and it is replaced by a lens, but this model provides a fundamental understanding of the image formation process and its most important parameters.

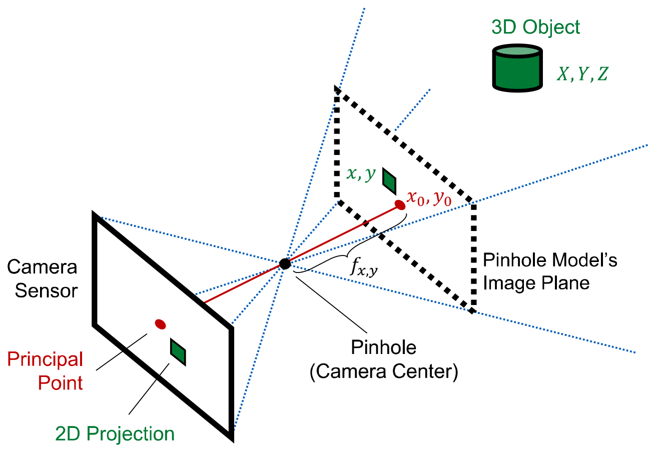


Figure 8: Pinhole camera model.

The aim here is to describe the model that maps 3D world points in standard metric units to the pixel coordinates of an image sensor. To achieve this, we use perspective geometry.

2.2.1 Perspective Projection Camera Model

The mapping from 3D coordinates in the world frame to pixel coordinates in the image can be understood as the concatenation of three successive stages:

1. A rigid transformation that maps points expressed in world coordinates to the same points expressed in the camera frame.

2. A perspective projection from the 3D camera frame coordinates to the 2D image plane.

3. A mapping from the metric image coordinates to the discrete pixel coordinates.

With reference to Figure 9, the scene point P must be transformed in sequence as:

$$\mathbf{p}^w \to \mathbf{p}^c \to \mathbf{q}^i \to (u, v)$$

where $\mathbf{p}^w, \mathbf{p}^c$ are the 3D coordinates of the point in the world and camera frames, respectively, and $\mathbf{q}^i$ and (u, v) are the corresponding 2D coordinates of the point in metric and pixel units in the image plane.

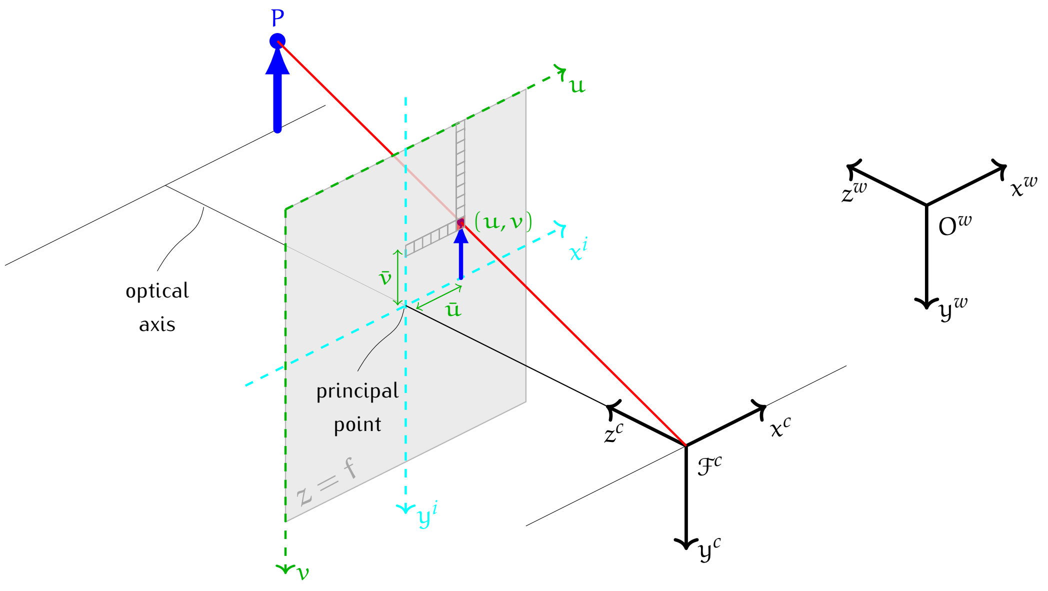


Figure 9: 3D Projection Visualization

The homogeneous coordinates of the scene point with respect to the world frame can be mapped into the coordinates in the camera frame through a 6-degree-of-freedom rototranslation:

$$\begin{pmatrix} x^c \\ y^c \\ z^c \\ 1 \end{pmatrix} = \underbrace{\begin{bmatrix} \mathbf{R}^c_w & \mathbf{t} \\ \mathbf{0}^\top & 1 \end{bmatrix}}_{\mathbf{H}^c_w} \begin{pmatrix} x^w \\ y^w \\ z^w \\ 1 \end{pmatrix}$$

$$\tilde{\mathbf{p}}^c = \mathbf{H}^c_w \, \tilde{\mathbf{p}}^w$$

where $\mathbf{R}^c_w$ represents the rotation and $\mathbf{t}$ the translation of the 4×4 homogeneous matrix $\mathbf{H}^c_w$ that represents the rigid coordinate transformation in the above equation. Observing then the similarity of triangles we get

$$\frac{x^i}{f} = \frac{x^c}{z^c}, \qquad \frac{y^i}{f} = \frac{y^c}{z^c}$$

where (x^i, y^i) is the position in metric units in the image plane and f is the distance of the image plane from the camera center and it's called *focal length*. We can then write in matrix form

$$z^c \begin{pmatrix} x^i \\ y^i \\ 1 \end{pmatrix} = \underbrace{\begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{\mathbf{H}_c^i} \begin{pmatrix} x^c \\ y^c \\ z^c \\ 1 \end{pmatrix}$$

$$z^c\, \tilde{\mathbf{q}}^i = \mathbf{H}_c^i\, \tilde{\mathbf{p}}^c$$

where $\mathbf{H}_c^i$ denotes the 3×4 perspective projection matrix.
Subsequently the image on the image plane is sampled by an image sensor such as a CCD or CMOS device at the location defined by an array of pixels. We then define the number of pixel per unit distance in the two camera frame directions as m_x and m_y and the position of the principal point is modeled with coordinates $[u_0, v_0]^T$ hence we get

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \underbrace{\begin{bmatrix} m_x & 0 & u_0 \\ 0 & m_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{H}_i} \begin{pmatrix} x^i \\ y^i \\ 1 \end{pmatrix}$$

$$\tilde{\mathbf{q}} = \mathbf{H}_i\, \tilde{\mathbf{q}}^i$$

where $\mathbf{H}_i$ denotes the 3×3 projective matrix.
Putting everything together we get

$$z^c \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{bmatrix} m_x & 0 & u_0 \\ 0 & m_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R_w^c & \mathbf{t} \\ \mathbf{o}^T & 1 \end{bmatrix} \begin{pmatrix} x^w \\ y^w \\ z^w \\ 1 \end{pmatrix}$$

$$z^c \overbrace{\begin{pmatrix} u \\ v \\ 1 \end{pmatrix}}^{\substack{\text{homogeneous}\\ \text{image}\\ \text{coordinates}}} = \underbrace{\overbrace{\begin{bmatrix} \alpha_x & 0 & u_0 \\ 0 & \alpha_y & v_0 \\ 0 & 0 & 1 \end{bmatrix}}^{\text{intrinsic camera parameters}}}_{\mathbf{K}} \underbrace{\overbrace{\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix}}^{\substack{\text{extrinsic}\\ \text{camera}\\ \text{parameters}}}}_{[\mathbf{R}|\mathbf{t}]} \overbrace{\begin{pmatrix} x^w \\ y^w \\ z^w \\ 1 \end{pmatrix}}^{\substack{\text{homogeneous}\\ \text{world}\\ \text{coordinates}}} \tag{3}$$

$$z^c\, \tilde{\mathbf{q}} = \mathbf{P}\, \tilde{\mathbf{p}}^w, \quad \mathbf{P} = \mathbf{K}\,[\mathbf{R}|\mathbf{t}] \tag{4}$$

where $\alpha_x = f\, m_x, \alpha_y = f\, m_y$ and the overall transformation matrix **P** is called **camera matrix**.

2.2.2 Distortion

Typical camera may not be accurately represented by the pinhole camera model as there might be lens distortion which disrupts the assumed projective model. The effect of distortion is non-linear and it must be corrected so that the camera can again be modeled as a linear device.

The distortion modifies the image plane coordinates into their distorted version:

$$\mathbf{q}^i \rightarrow \tilde{\mathbf{q}}^i$$

Various types of distortion exist, but typically the most significant component is radial distortion which is radially symmetric and can be modeled using a low-order polynomial such as

$$\underbrace{\begin{pmatrix} \tilde{x}^i \\ \tilde{x}^i \end{pmatrix}}_{\substack{\text{distorted} \\ \text{image} \\ \text{position}}} = \underbrace{\begin{pmatrix} x^i \\ y^i \end{pmatrix}}_{\substack{\text{undistorted} \\ \text{image} \\ \text{position}}} + \begin{pmatrix} x^i \\ y^i \end{pmatrix} (k_1\, r^2 + k_2\, r^4) \tag{5}$$

with $r = \|(x^i, y^i)\|$, and k_1, k_2 being the distortion parameters to be estimated for the specific camera. From this, we can see that distortion increases as we move away from the center of the image. Based on reports in the literature, such as Z. Zhang, 2000, it has been found that any more complex model would not be beneficial for estimation, as it would be comparable to sensor quantization and could also cause greater numerical instability.

2.2.3 Camera Calibration

Camera calibration is an estimation problem that is done to determine the optimal camera parameters that produced a given set of images. This includes both the extrinsic parameters **R** and **t**, as well as the intrinsic parameters of the matrix **K** and the distortion parameters k_1 and k_2. Once all these parameters are known, the expression of the camera matrix **P** can be computed and this allows us to back project any image pixel onto a 3D ray in space using the model of Equation 3.

Pears et al., 2012 presents different types of calibration that differ by precision and usability:

- **Photogrammetric calibration** is a calibration method that requires precise 3D physical knowledge of a scene object. Several images of this object are taken from different positions (e.g. along orthogonal planes) using known translations. This method provides very accurate results but lacks flexibility due to the requirement of the precise knowledge of the scene.
- **Self-calibration** is a calibration method that does not require a calibration target. It is based on the principle that finding the correspondences across three views of the same rigid scene

is sufficient to allow the estimation up to a similarity constraint. This approach is flexible but not always reliable calibrations can be obtained.

- **"Desktop camera calibration"** is a calibration method that involves capturing images of flat calibration grids from various unspecified angles and orientations. This method is an effective compromise between the precision of photogrammetric calibration and the ease of use of self-calibration.

Zhang's calibration method

Zhang's calibration method Z. Zhang, 2000 provides an easy and efficient calibration procedure that can be used in many applications and produces good calibration results. It uses only 2D metric information, unlike photogrammetric methods, and it represents an important step toward integrating 3D computer vision applications into user environments. It requires a calibration pattern that must be attached to a planar plate and at least three images must be taken from different orientations. This can be achieved by freely moving *either* the camera *or* the calibration plate. The procedure is composed by two main steps:

1. For each image of the plate taken from a specific orientation, we don't know the exact 3D positions of the points. Therefore, we first need to estimate the homography from the calibration plane to the image plane.
2. Once we have the estimated homographies for each orientation, we can use them to formulate an estimation problem for the camera parameters. To solve this problem we first formulate a closed-form solution that we use as initialization for a nonlinear refinement based on the maximum likelihood criterion.

Starting from the pinhole model in Equation 3, and without loss of generality, we assume that in the first image, the calibration plate lies on the $z^w = 0$ plane in the world coordinate system. The expression then simplifies to:

$$z^c \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \mathbf{K} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{pmatrix} x^w \\ y^w \\ 0 \\ 1 \end{pmatrix} = \underbrace{\mathbf{K} \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ r_{31} & r_{32} & t_z \end{bmatrix}}_{\mathbf{H}} \begin{pmatrix} x^w \\ y^w \\ 1 \end{pmatrix}$$

therefore a point on the model plane and its image point is related by the 3×3 homography $\mathbf{H} = \begin{bmatrix} \mathbf{h_1} & \mathbf{h_2} & \mathbf{h_3} \end{bmatrix}$ defined up to a scale factor:

$$z^c \tilde{\mathbf{q}} = \mathbf{H}\, \tilde{\mathbf{p}}^w$$

$$\begin{bmatrix} \mathbf{h_1} & \mathbf{h_2} & \mathbf{h_3} \end{bmatrix} = \lambda \mathbf{K} \begin{bmatrix} \mathbf{r_1} & \mathbf{r_2} & \mathbf{t} \end{bmatrix} \tag{6}$$

where $\tilde{\mathbf{q}}, \tilde{\mathbf{p}}^w$ denote the homogeneous pixel coordinates and the compressed metric world coordinates, respectively, and λ is an arbitrary scalar.

As mentioned earlier, before proceeding, we first need to estimate the homography between the considered view and the image plane. This estimate, denoted as $\hat{\mathbf{H}}$, can be found by solving the following minimum log-likelihood problem:

$$\hat{\mathbf{H}} = \min_{\mathbf{H}} \sum_i (\tilde{\mathbf{q}}_i - \mathbf{H}\,\tilde{\mathbf{p}}_i^w)^T (\tilde{\mathbf{q}}_i - \mathbf{H}\,\tilde{\mathbf{p}}_i^w)$$

that can be solved using an algorithm such as Newton's method.

We can then rewrite Equation 6 in terms of the estimated homography:

$$\begin{bmatrix} \hat{\mathbf{h}}_1 & \hat{\mathbf{h}}_2 & \hat{\mathbf{h}}_3 \end{bmatrix} = \lambda\,\mathbf{K} \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{t} \end{bmatrix} \tag{7}$$

We now need to solve for the intrinsic and extrinsic parameters.
Knowing that $\mathbf{r}_1$ and $\mathbf{r}_2$ are orthonormal, the following conditions must be satisfied:

$$\hat{\mathbf{h}}_1^T\,\mathbf{K}^{-T}\,\mathbf{K}^{-1}\,\hat{\mathbf{h}}_2 = 0 \tag{8}$$

$$\hat{\mathbf{h}}_1^T\,\mathbf{K}^{-T}\,\mathbf{K}^{-1}\,\hat{\mathbf{h}}_1 = \hat{\mathbf{h}}_2^T\,\mathbf{K}^{-T}\,\mathbf{K}^{-1}\,\hat{\mathbf{h}}_2 \tag{9}$$

which are two basic constraints on the intrinsic parameters for each view.
We then define

$$\mathbf{B} = \mathbf{K}^{-T}\mathbf{K}^{-1} = \begin{bmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{bmatrix} = \begin{bmatrix} \frac{1}{\alpha_x^2} & 0 & \frac{-u_0\,\alpha_y}{\alpha_x^2\,\alpha_y} \\ 0 & \frac{1}{\alpha_y^2} & -\frac{v_0}{\alpha_y^2} \\ \frac{-u_0\,\alpha_y}{\alpha_x^2\,\alpha_y} & -\frac{v_0}{\alpha_y^2} & \frac{(u_o\,\alpha_y)^2}{\alpha_x^2\,\alpha_y^2} + \frac{v_0^2}{\alpha_y^2} + 1 \end{bmatrix}$$

that is a symmetric matrix that contains the intrinsic camera parameters.

Therefore, if we are able to find an estimate of the vector

$$\mathbf{b} = [B_{11}, B_{12}, B_{22}, B_{13}, B_{23}, B_{33}]^T$$

we can solve for the estimates of the intrinsic camera parameters.
If we let the i-th column of the estimated homography $\hat{\mathbf{H}}$ be $\hat{\mathbf{h}}_i = [h_{i1}, h_{i2}, h_{i3}]^T$ we have

$$\hat{\mathbf{h}}_i{}^T\,\mathbf{B}\,\hat{\mathbf{h}}_j = \mathbf{v}_{ij}^T\,\mathbf{b}$$

$$\mathbf{v}_{ij} = [\hat{h}_{i1}\,\hat{h}_{j1},\ \hat{h}_{i1}\,\hat{h}_{j2} + \hat{h}_{i2}\,\hat{h}_{j1},\ \hat{h}_{i2}\,\hat{h}_{j2},\ \hat{h}_{i3}\,\hat{h}_{j1} + \hat{h}_{i1}\,\hat{h}_{j3},\ \hat{h}_{i3}\,\hat{h}_{j2} + \hat{h}_{i2}\,\hat{h}_{j3},\ \hat{h}_{i3}\,\hat{h}_{j3}]^T$$

so that we can write the constrains of Equation 8 - 9 as a system

$$\begin{bmatrix} \mathbf{v}_{12}^{T} \\ (\mathbf{v}_{11} - \mathbf{v}_{22})^{T} \end{bmatrix} \mathbf{b} = \mathbf{0}$$

Taking multiple images at different orientation and stacking the equations we get

$$\mathbf{V}\,\mathbf{b} = \mathbf{0}, \qquad \mathbf{V} \in \mathbb{R}^{2n \times 6}$$

that has a unique solution for $n \geqslant 3$ given by the eigenvector of $\mathbf{V}^{T}\,\mathbf{V}$ associated with the smallest eigenvalue. Given the estimated intrinsic parameters, the extrinsic parameters can be directly estimated from Equation 7:

$$\begin{cases} \mathbf{r}_1 &= \lambda\,\mathbf{K}^{-1}\,\hat{\mathbf{h}}_1 \\ \mathbf{r}_2 &= \lambda\,\mathbf{K}^{-1}\,\hat{\mathbf{h}}_2 \\ \mathbf{r}_3 &= \mathbf{r}_1 \times \mathbf{r}_2 \\ \mathbf{t} &= \lambda\,\mathbf{K}^{-1}\,\hat{\mathbf{h}}_3 \\ \lambda &= \frac{1}{\|\mathbf{K}^{-1}\,\hat{\mathbf{h}}_1\|} = \frac{1}{\|\mathbf{K}^{-1}\,\hat{\mathbf{h}}_2\|} \end{cases}$$

The given solution is fundamental for providing a good initialization for the high-dimensional maximum likelihood non-linear optimization problem that yields the refined solution and requires a good initialization to converge to the global optimum:

$$\min_{\mathbf{K},\mathbf{R},\mathbf{t}} \sum_{i=1}^{n} \sum_{j=1}^{m} \|\tilde{\mathbf{q}}_{ij} - \mathbf{K}\,[\mathbf{R}|\mathbf{t}]\,\tilde{\mathbf{p}}_{ij}\|^2$$

2.3 3D DATA ACQUISITION SENSORS

When we design a 3D computer vision system, we must choose the right sensors based on the specific application requirements. These requirements may impose constraints on the size of the objects to be captured, the working distance, the operating conditions for the sensor, as well as the accuracy of acquisition and the cost of the sensors. The first important distinction is between *passive* and *active* acquisition systems: passive systems rely only on the environment light sources, while active systems emit waves, typically light, into the environment. This brings some limitations to passive systems, as they are highly dependent on the lighting conditions, while active sensors are generally more accurate and robust to changes of the environment even thought this comes with the drawback of an higher cost. The second main distinction is the operating principle that allows the 3D reconstruction:

- **Triangulation** is based on geometric principles, where the 3D position of a point is determined by finding the intersection of two or more 3D rays coming from different directions, along which the point must lie.

- **Time of Flight** is based on precise timing, where the distance from a point is measured by round trip time that takes a light ray emitted into the environment to be detected with its reflected energy.

- **Deep Learning** methods, instead, estimate the depth component from monocular images by making predictions based on local and global features.

Figure 10 shows a classification of the main 3D data acquisition systems based on their type and operating principle.

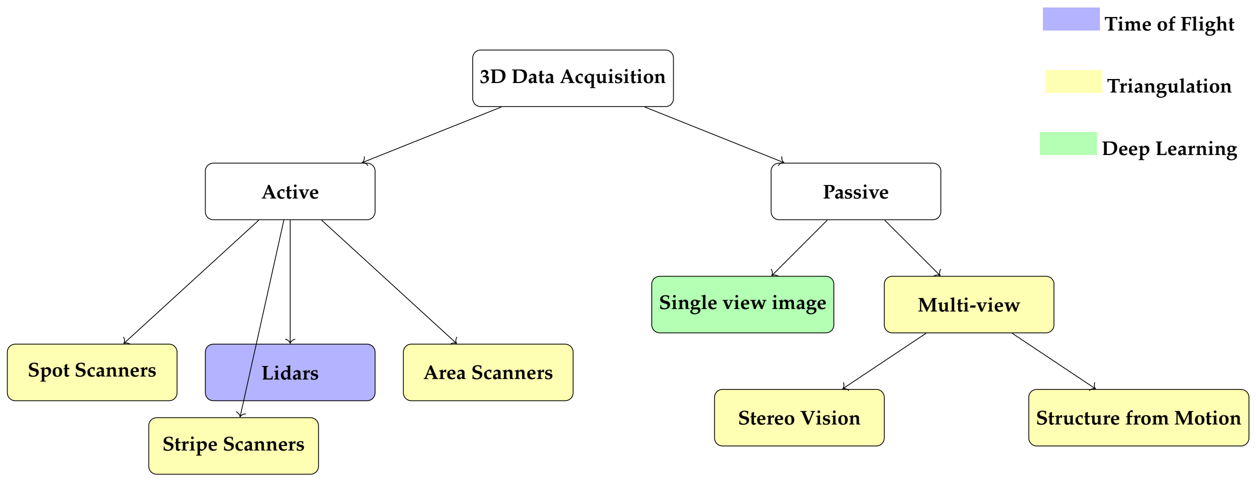


Figure 10: Tentative classification of 3D imaging techniques based on the operating principle.

2.3.1 Passive 3D Imaging Systems

Passive imaging systems rely on ambient light to detect the shape of objects. These systems typically include the use of one or more cameras and are mainly divided into monocular and multi-view systems, depending on the number of images used to reconstruct the 3D scene.

Depth from Focus

Depth from focus techniques are based on single-view images and the principle that cameras have a limited depth of field. Depending on the distance and focus, points in the image appear more or less sharply. By taking multiple images at different object distances, each point will be represented with varying focus. The distance of each point from the camera can be estimated based on the image where the point's focus is at its highest. This concept is illustrated in Figure 11.

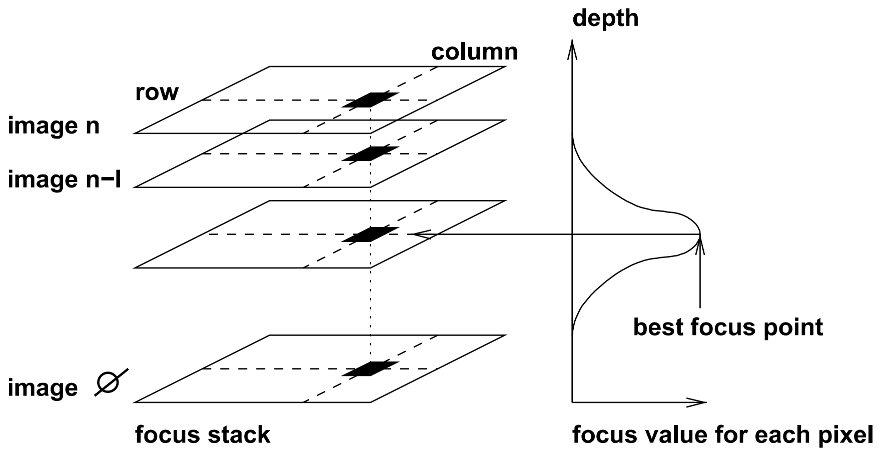


Figure 11: Depth from focus.

Stereo Vision

Stereo vision systems (Pears et al., 2012) are widely used imaging systems that employ two or more cameras to capture the same scene from different spatial positions as shown in Figure 12. A stereo system must solve two main problems:

- **Correspondence problem**: Given a point in one image, what is the corresponding point in the other images?
- **Depth calculation**: Given the two or more corresponding points, how can we calculate the 3D position of this projected points?

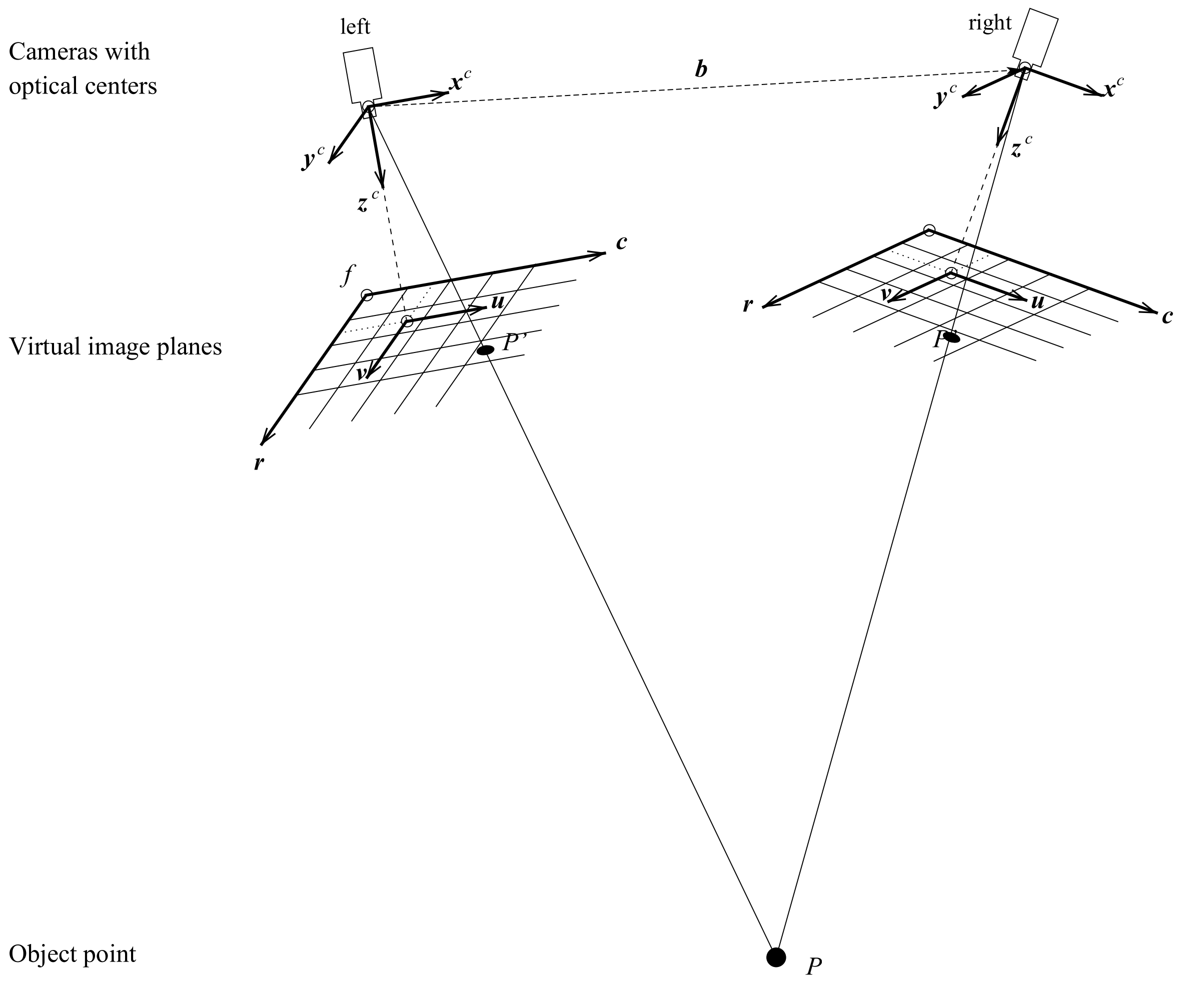


Figure 12: Stereo setup with two cameras.

The depth calculation problem is straightforward to solve. If the cameras are calibrated, we have the pinhole model from Equation 3 for each camera that allows us to calculate the intersection of the 3D rays where the point must lie through a process called triangulation. The correspondence problem is much more complex to solve because, in theory, finding the corresponding point in other images could require searching every pixel, which would be computationally intensive. To simplify the problem, we need to restrict the search space for each pixel, which can be achieved using **epipolar geometry**. This technique restricts the search to a single line in the other image.

Structure from motion SfM

Structure from Motion (SfM) is another reconstruction technique based on triangulation. Unlike stereo vision, which uses multiple stationary cameras, SfM uses a single camera that moves in the space. The correspondences of points are then searched between the images captured during the camera's movement, as illustrated in Figure 13.

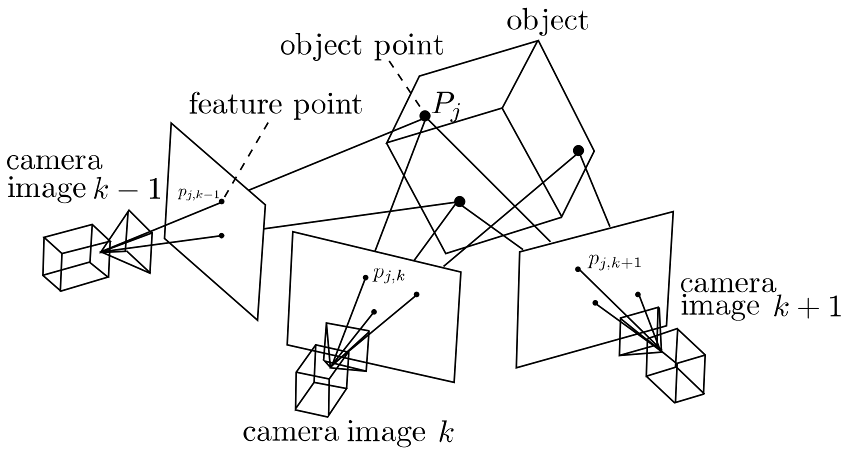


Figure 13: Structure from motion.

2.3.2 Active 3D Imaging Systems

Active imaging systems project light into the environment, providing the significant advantage of being more robust to changes in the operating conditions. Depending on the application, different techniques might be used. For very small objects, interferometry techniques offer high precision. Triangulation is commonly employed in industrial settings for objects ranging from a few centimeters to a few meters. Time of Flight systems are suitable for larger objects, that span from a few meters to several kilometers.

Spot scanners

Spot scanners are 3D sensors that project a ray of light into an object and use triangulation based geometry to calculate the distance of each point on the surface of the object.

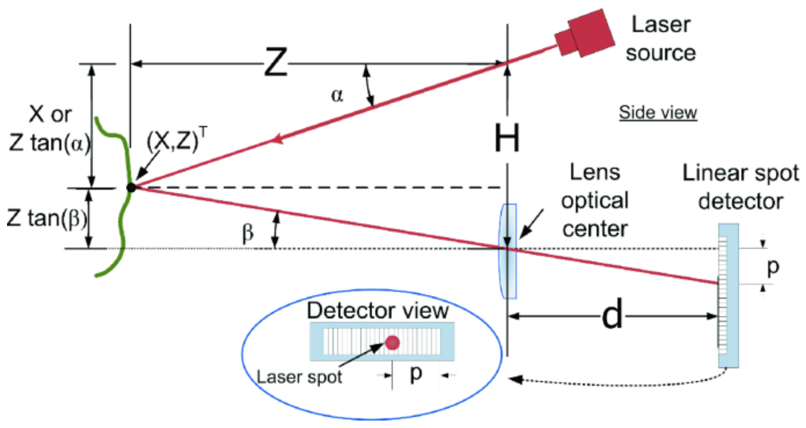


Figure 14: Spot scanners setup Pears et al., 2012

To capture the entire object, a mirror is mechanically rotated to direct the light source in all directions. Figure 14 shows the structure of this type of sensors.

Stripe Scanners

Stripe scanners are highly accurate acquisition systems commonly used in industrial settings. As illustrated in Figure 15, they project a laser stripe into the object's surface, and an image sensor measures the intensity of the reflected light. This information is then used to determine the distance from the object.

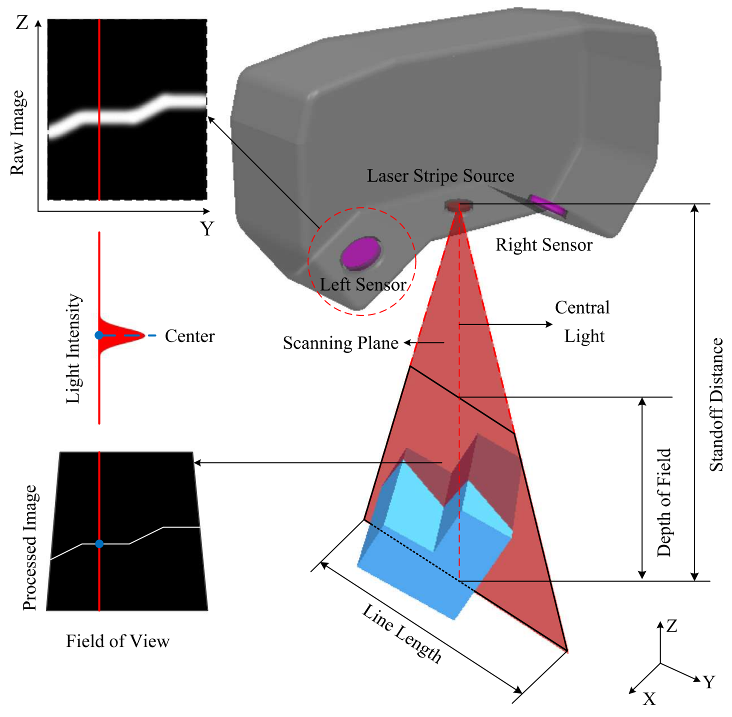


Figure 15: Stripe scanner structure Wang and Feng, 2014.

The output of these sensors is an array of values corresponding to the distance information for each scanned line. Key parameters that define laser scanners include the depth of field and the width of the scan line. To scan an entire object, either the object or the scanner must be moved.

Area-based structured light systems

Structured light systems simplify the correspondence problem in stereo vision by replacing one camera with a projector that casts light patterns into the object's surface. These patterns are distorted by the object's geometry (Figure 16). The camera captures the distorted patterns and analyzes them using the known projected features, making the search for correspondences easier and, most importantly, more robust, especially for objects with smooth or textureless regions.

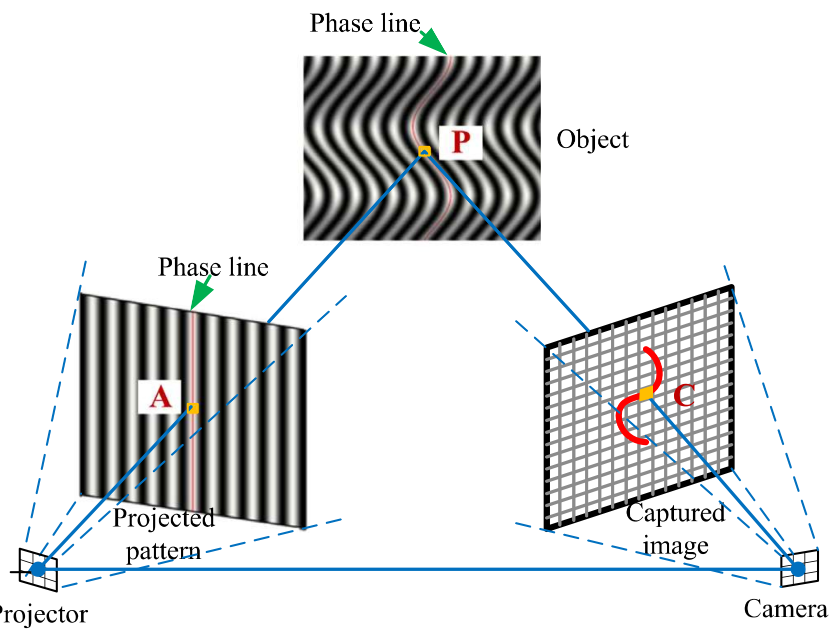


Figure 16: Typical structured light system S. Zhang, 2018.

2.4 3D DATA REPRESENTATION

After the 3D data acquisition process, data can be represented in various forms, some straightforward from the raw data, and others more complex Pears et al., 2012. The choice of this data representation is fundamental for the entire 3D computer vision application as the selection of the processing architecture is often highly dependent on the structure of the input data. In fact, some processing architecture require a specific data representation, as it is typical in deep learning networks. Therefore, it is important to understand the different ways to represent the 3D data, the properties of each representation, and the most efficient ways for converting between different representations.

Out of all the possible representations shown in Figure 17, we can identify three main categories that are commonly used in applications:

- **Raw data** include depth maps and point clouds, which are direct representations obtained from sensors. These typically require further conversion into more structured representations.

- **Surface representation** are continuous representations that model the surface of an object, making them very commonly used in applications for their regularity properties.

- **Volumetric representation** are used to approximate and model the entire volume of an object through quantization, similarly to what is done with 2D images.

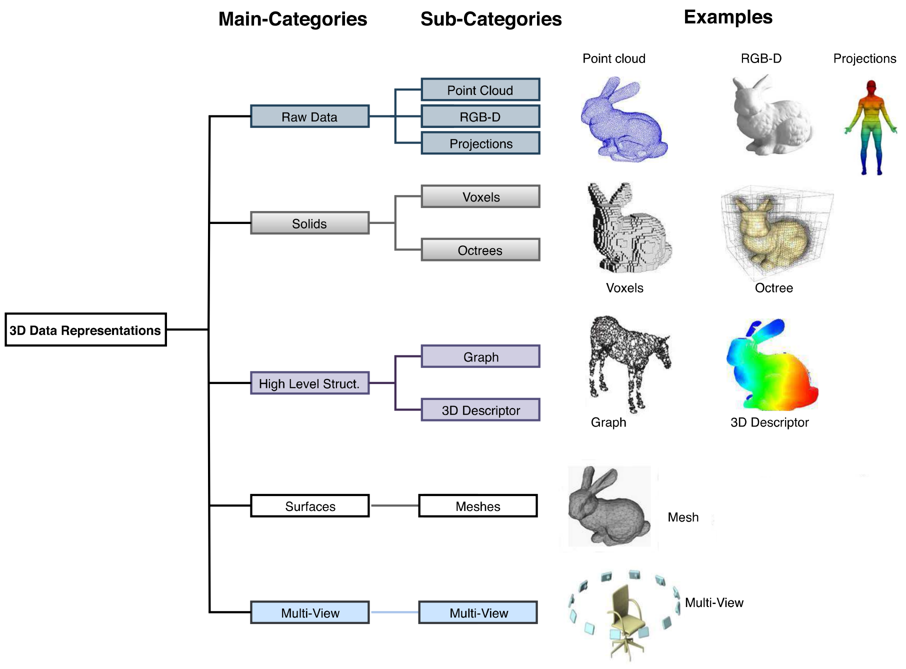


Figure 17: Diagram of the main 3D data representations Gezawa et al., 2020.

2.4.1 Raw Data

Raw data refers to representations that are directly derived from the sensor's output, that typically include depth maps and point clouds. The main advantage of this representation is its simplicity, both in visualization and comprehension. However, this simplicity in structure also brings limitations for many applications as it often requires further processing or transformation into more complex representations.

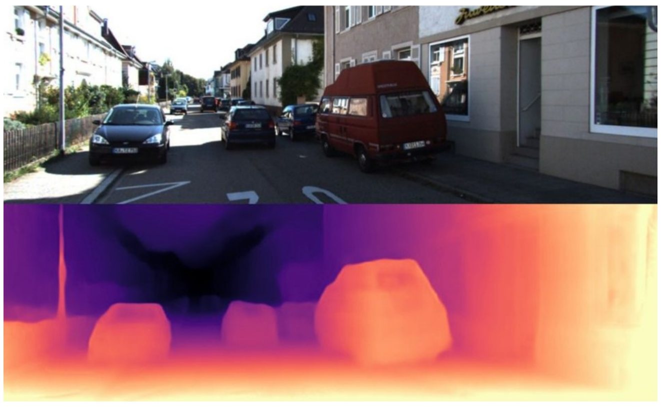
Depth Maps

Depth maps are 2D images where each pixel represents the distance from the camera of the corresponding point in the scene. This representation is compact and easy to visualize and it is usually derived as the direct output of 3D sensors like LiDARs and optical scanners.

To determine the exact 3D coordinates of a point corresponding to a pixel, the camera parameters and the resolution $\Delta x, \Delta y$ along the two directions must be known. With this information, the depth map can be converted into a point cloud, transforming the 2D image into a 3D representation.

Point Clouds

Point Clouds are unordered sets of points $\mathbf{P} = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$ with $\mathbf{v}_i \in \mathbb{R}^3$, which can be stored as an array of $n \times 3$ float values. Point clouds typically come directly from stereo vision and reconstruction techniques, constituting an initial representation in many applications, before being transformed into more complex ones.


(a) *Example of depth map.*

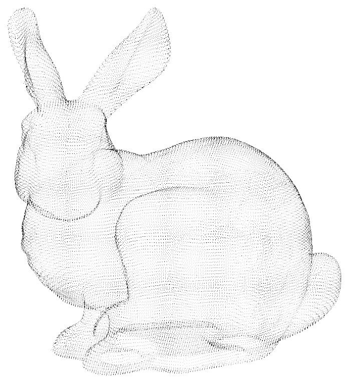

(b) *Example of 3D point cloud (Stanford bunny).*

Figure 18: Raw data examples.

An important characteristic of point clouds that must be accounted for by the processing algorithms is the fact that any permutation of the points leaves the cloud unchanged. This lack of structure presents challenges when using machine learning architectures, as it prevents the use of convolutional layers, which are fundamental components in deep learning architectures.

2.4.2 Surface Representations

Surface representations are the most commonly used representation by 3D computer vision processing algorithms and graphics applications. These representations are usually generated from point clouds or are directly modeled and designed in Computer Aided Design (CAD). Among these, polygonal meshes are the most widely used, where the surface is approximated by polygons.

Meshes

Meshes, unlike point clouds, are ordered sets of vertices and faces connected together. The faces are composed of polygons, most commonly triangles, that approximate the surface of the object.

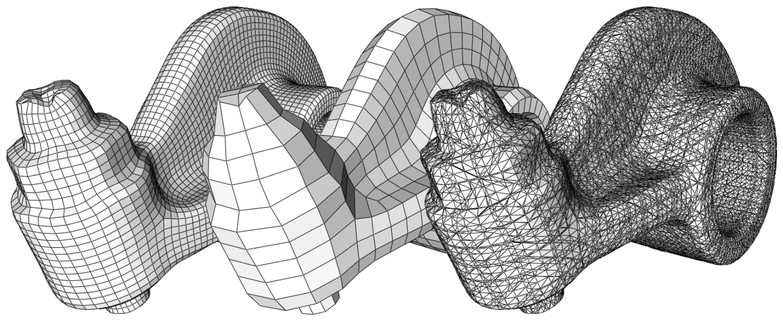
Since the starting point of many applications is a point cloud, but processing algorithms require meshes, the point cloud is often triangulated into a mesh. There exists various triangulation algorithms but the most common one is Delaunay Triangulation P. Su and Scot Drysdale, 1997.

2.4.3 Solid-Based Representations

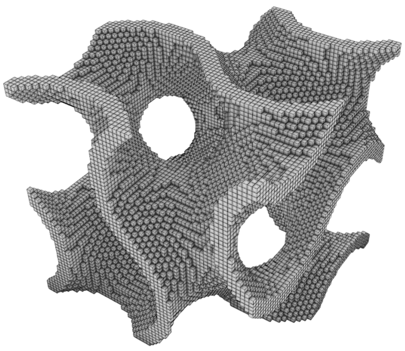
Solid-Based Representations are regular 3D representations that are very useful in many applications as they allow the discretization of the 3D space and thus enabling the extension of 2D techniques.

Voxels

Voxel representations are the most common volumetric representations that approximate the volume using voxels that are the analogous to pixels in 2D images. These representations are typically derived from surface representations and point clouds through a process called voxelization. This type of representation is fundamental in many deep learning applications, as their spatial structure allows the use of convolutional layers and consequently facilitates the extension of many algorithms from the 2D case.



(a) *Example of different 3D polygonal meshes.*



(b) *3D voxel representation.*

Figure 19: Meshes and voxel grids examples.

3 3D SHAPE ANALYSIS AND INFERENCE APPLICATIONS

After data acquisition, 3D computer vision tasks must be able to perceive and interpret the information present in the data to perform analysis and decide on control actions. Some typical applications include object detection, pose estimation and classification within the scene and 3D scene reconstruction from different views. For all these types of applications, it is necessary to have descriptors of local keypoints of the objects. These types of local descriptors represent the transition from low-level processing to high-level understanding of the data.

This chapter begins with an overview of some local features and then demonstrates how they can be used to solve high-level 3D inference applications such as 3D reconstruction and 3D surface matching. The remainder of the chapter reviews some of the deep learning methods used to solve 3D computer vision applications.

3.1 3D LOCAL FEATURES

When analyzing an image or a 3D scene, the first step in understanding its content, beyond the initial perception of the data, is to detect the most salient points and obtain a description of each one. By combining all these local descriptors, we can then process this information globally and match it against known features to achieve a complete understanding. It is therefore essential to mathematically frame the following tasks:

- **Keypoint detection**: detect points that contain rich information about the surface and that are stable, such that the influence of other factors, such as noise, does not impact the detection of the keypoints.

- **Feature description**: for each detected keypoints, compute mathematical quantities that should be descriptive of the local surface and robust to noise.

In the following, I review two methods for solving these tasks that have been widely used in practice.

3.1.1 Keypoint Detection - 3D Harris Keypoints

In Harris and Stephens, 1988, Harris proposed a point detection algorithm for 2D images based on the autocorrelation function to measure local changes in intensity. In the 2D case, the local autocorrelation was defined as:

$$e(x,y) = \sum_{x_i,y_i} W(x_i,y_i)\left[I(x_i+\Delta x, y_i+\Delta y) - I(x_i,y_i)\right]^2$$

where $I(\cdot,\cdot)$ denotes the intensity function, and (x_i, y_i) are the points that define the local area around (x, y) where the Gaussian function W is computed for filtering purposes. Expanding with a first-order Taylor expansion, we get

$$I(x_i+\Delta x, y_i+\Delta y) = I(x_i,y_i) + \frac{\partial I}{\partial x}\Delta x + \frac{\partial I}{\partial y}\Delta y$$

$$e(x,y) = \mathbf{S}\begin{pmatrix} \sum\limits_{x_i,y_i} W(x_i,y_i)\left(\frac{\partial I}{\partial x}\right)^2 & \sum\limits_{x_i,y_i} W(x_i,y_i)\frac{\partial I}{\partial x}\frac{\partial I}{\partial y} \\ \sum\limits_{x_i,y_i} W(x_i,y_i)\frac{\partial I}{\partial x}\frac{\partial I}{\partial y} & \sum\limits_{x_i,y_i} W(x_i,y_i)\left(\frac{\partial I}{\partial y}\right)^2 \end{pmatrix}\mathbf{S}^T = \mathbf{S}\,E(x,y)\,\mathbf{S}^T$$

where $\mathbf{S} = [\Delta x, \Delta y]$.

He then proposed to consider the eigenvalues of the matrix E, which contain most of the local information. To further simplify the operations, for each pixel of the image we can calculate:

$$h(x,y) = \det(E(x,y)) - k(\mathrm{tr}(E(x,y)))^2$$

which gives a similar metric without having to explicitly compute the eigenvalues. This detector has been widely used in 2D applications, so the goal is to extend it to the 3D case of a surface $z = f(x,y)$.

In Sipiran and Bustos, 2011, the authors proposed fitting a paraboloid centered on each vertex v, of the type:

$$z = f(x,y) = \frac{p_1}{2}x^2 + p_2xy + \frac{p_3}{2}y^2 + p_4x + p_5y + p_6$$

Being interested in the derivatives at the point v, we can directly compute:

$$f_x = \left.\frac{\partial f(x,y)}{\partial x}\right|_{x=0} \qquad f_y = \left.\frac{\partial f(x,y)}{\partial y}\right|_{y=0}$$

which should be a good estimate but could be influenced by noise. To address this problem, in line with the concept of stability of the keypoint, Gaussian filtering of the derivatives has been proposed, computing:

$$A = \frac{1}{\sqrt{2\pi}\sigma} \int_{\mathbb{R}^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} f_x(x,y)^2 \, dx dy$$

$$B = \frac{1}{\sqrt{2\pi}\sigma} \int_{\mathbb{R}^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} f_y(x,y)^2 \, dx dy$$

$$C = \frac{1}{\sqrt{2\pi}\sigma} \int_{\mathbb{R}^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} f_x(x,y) \, f_y(x,y) \, dx dy$$

with σ being the standard deviation of the Gaussian filter. Finally, the autocorrelation matrix associated with the vertex v can be computed with:

$$E(v) = \begin{pmatrix} A & C \\ C & B \end{pmatrix}$$

and we use the same operator of the 2D case to evaluate the eigenvalues:

$$h(v) = \det(E(v)) - k(\text{tr}(E(v)))^2$$

After computing the quantity $h(v)$ for all vertices, the keypoints are chosen from the local maxima of the operator h.

3.1.2 3D Local Descriptors - Spin Images

The authors in Johnson and Hebert, 1999 proposed a local descriptor called Spin Images that is widely used in applications. The name refers to the fact that the 3D meshes are described by a set of 2D grids that are rotated around the keypoint normal. The images are obtained by projecting the 3D vertices. Given an keypoint v and its normal $\mathbf{n}$ they define the local quantities:

$$\alpha = \sqrt{\|x - v\|^2 - (\mathbf{n}^T (x - v))^2} \qquad \beta = \mathbf{n}^T (x - v)$$

where x represent the local vertices to project onto the grid. Hence for each keypoint v the spin image is defined by the map

$$S_v : \mathbb{R}^3 \to \mathbb{R}^2$$

$$S_v(x) \to (\alpha, \beta) = \left(\sqrt{\|x - v\|^2 - (\mathbf{n}^T (x - v))^2}, \mathbf{n}^T (x - v) \right)$$

The selection of the projected vertices is controlled by the width of the spin image and by the angle between the normal of the vertex and the normal of the keypoint. After this computation, the local quantities (α, β) are discretized into bins and a bilinear interpolation is performed to ensure robustness to noise.

3.2 3D SHAPE REGISTRATION

In the acquisition process of data from a 3D scene, multiple sensors might be used, each with its own coordinate system, and multiple views of the scene from different perspectives might be produced. 3D shape registration is a fundamental problem in 3D computer vision that aims to bring together all the data into a single coordinate system. Several solutions to this problem have been presented, and the state-of-the-art algorithms for this type of problem are variants of the classical Iterative Closest Point (ICP) algorithm, which will be presented in the section below. This type of algorithm serves as a tool in various high-level applications such as model reconstruction, model fitting, and object recognition.

3.2.1 Registration of Two Views

A general registration problem of n views can be solved by registering two views at a time, one being the current registration of the first i views, and the second being the $i+1$ view. Hence, the registration problem of two views is a fundamental building block for the solution.

Given two partial 3D views $\mathbb{D}$ and $\mathbb{M}$ (typically point clouds or triangulated meshes) of the same object, with the first representing the data view and the second the model view, the registration problem consists in estimating the parameters $\mathbf{a}$ of the transformation $\mathbf{T}$ that best aligns $\mathbb{D}$ to $\mathbb{M}$.

All registration methods differ in the choice of three main aspects (Pears et al., 2012):

1. **The transformation function**, which is typically a rigid transformation determined by a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$. These can generally be defined by a parameter vector $\mathbf{a}$, as in the case of the three rotation angles around the principal axes and the three coordinates of the translation vector.
2. **The error function**, which must be designed such that minimizing this function provides the best parameter vector that solves the registration problem.
3. **The optimization method**, which is used to numerically minimize the error function. General optimization methods might be used, such as Newton's methods, but the state-of-the-art algorithm for the registration specific case is ICP.

Once these aspects are defined, the registration problem is then framed as an optimization problem:

$$\mathbf{a}^* = \arg\min_{\mathbf{a}} E\left(\mathbf{T}(\mathbf{a}, \mathbb{D}), \mathbb{M}\right) \tag{10}$$

where E is the error function that measures the registration error.

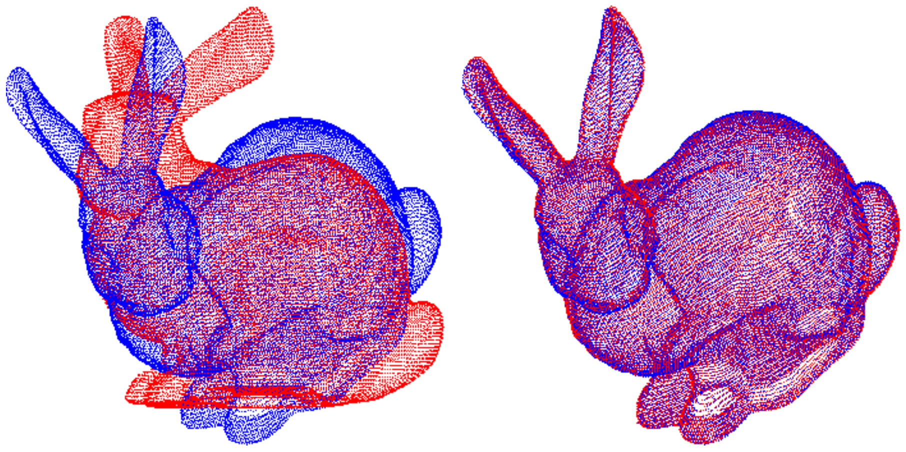


Figure 20: Registration of two views.

The Iterative Closest Point (ICP) Algorithm

The ICP is an iterative algorithm that keeps the model view fixed while transforming the data view by applying a rigid transformation composed of a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$. The two views are assumed to be available in the form of point clouds $\mathbb{M} = \{\mathbf{m}_1, \dots, \mathbf{m}_{N_m}\}$ and $\mathbb{D} = \{\mathbf{d}_1, \dots, \mathbf{d}_{N_d}\}$, which is one of the simplest 3D representations. The error function considered is the sum of squared errors between the model points and the transformed data points:

$$E_{ICP}(\mathbf{a}, \mathbb{D}, \mathbb{M}) = \sum_{i=1}^{N_d} \|(\underbrace{\mathbf{R}\,\mathbf{d}_i + \mathbf{t}}_{\text{transformed data point}}) - \mathbf{m}_j\|^2$$

$$\mathbf{a} = (\mathbf{R}, \mathbf{t}), \quad \mathbf{R} = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{33} \\ a_{31} & a_{32} & a_{33} \end{pmatrix}, \quad \mathbf{t} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix}$$

where $\mathbf{d}_i$ and $\mathbf{m}_j$ are two corresponding points of the model and data clouds.

The optimization problem becomes

$$\arg\min_{\mathbf{R},\mathbf{t}} \sum_{i=1}^{N_d} \|(\mathbf{R}\,\mathbf{d}_i + \mathbf{t}) - \mathbf{m}_j\|^2 \tag{11}$$

The optimization problem in Equation 11 is actually a nested optimization since the corresponding points of the two clouds are not defined a priori, otherwise the problem could be solved analytically. Hence, the procedure fixes a transformation for each iteration, and given the current transformed data view, for each point in this cloud, it calculates the closest point in the model view by minimizing the Euclidean distance between the two points:

$$j = \arg\min_{k \in \{1, \dots, N_m\}} \|(\mathbf{R}\mathbf{d}_i + \mathbf{t}) - \mathbf{m}_k\|^2$$

Once the correspondences are found, the problem in Equation 11 can be solved in closed form. It can be shown that the optimal transformation is the one that aligns the centroid of the model view with the centroid of the data view:

$$\mathbf{d}_c = \frac{1}{N_d}\sum_{i=1}^{N_d}\mathbf{d}_i, \quad \mathbf{m}_c = \frac{1}{N_m}\sum_{i=1}^{N_m}\mathbf{m}_i, \quad \mathbf{d}_c' = \hat{\mathbf{R}}\mathbf{d}_c + \hat{\mathbf{t}} = \mathbf{m}_c$$

We can then rewrite the optimization problem as a minimization with respect to only the rotation matrix **R**:

$$\arg\min_{\mathbf{R}} \sum_{i=1}^{N_d} \|(\mathbf{R}\,\underbrace{(\mathbf{d}_i - \mathbf{d}_c)}_{\mathbf{d}_i'} + \underbrace{(\mathbf{m}_c - \mathbf{m}_j)}_{\mathbf{m}_j'}\|^2 \tag{12}$$

The last formulation can be solved in closed form using the Singular Value Decomposition (SVD) of the cross-covariance matrix:

$$\mathbf{C} = \sum_{i=1}^{N_d} \mathbf{d}_i' \,(\mathbf{m}_j')^T \qquad \mathbf{USV^T} = \mathbf{C} \qquad \text{Singular Value Decomposition}$$

where **U**, **V** are two orthogonal matrices and **S** is the diagonal matrix of the singular values. The optimal solution of Equation 12 is then given by:

$$\hat{\mathbf{R}} = \mathbf{V\,S\,U}^T \qquad \mathbf{S} = \begin{cases} \mathbf{I} & \text{if } \det(\mathbf{U})\det(\mathbf{V}) = 1 \\ \mathrm{Diag}(1, 1, \dots, 1, -1) & \text{if } \det(\mathbf{U})\det(\mathbf{V}) = -1 \end{cases}$$

Finally, the calculation of the optimal translation is given directly by

$$\hat{\mathbf{t}} = \mathbf{m}_c - \hat{\mathbf{R}}\mathbf{d}_c$$

3.3 3D SHAPE MATCHING

Many applications require the capability to automatically locate an object in a 3D scene and determine its coordinates. This capability is useful in many industrial applications where robots must be guided in 3D space to perform actions, but also in social applications, such as guiding blind people, and in autonomous vehicles to recognize road signs, pedestrians, and other vehicles.

In the general setting, we have a 3D template of the object being searched for and a 3D scene where instances of the object might be found multiple times and possibly be partially occluded. The solution to this problem should robustly find the occurrences of the object in the scene, even under conditions that differ from the ideal template, and determine the pose of the found objects.

3.3.1 Least Square Matching

A first solution to the problem is proposed in Akca, 2007. In this case, the 3D matching problem is posed as a least squares problem that can be solved without calculating any 3D features. We assume that we have two surfaces with intensity values $f(x,y,z)$ and $g(x,y,z)$, which are, respectively, the template surface and another instance of the same object. The problem consists of finding the correspondences of the template surface in the searched surface. In the ideal case, we would have

$$f(x,y,z) = g(x,y,z)$$

but due to measurement errors and noise, we need to add a stochastic error component $e(x,y,z)$:

$$f(x,y,z) - e(x,y,z) = g(x,y,z) \tag{13}$$

To solve the problem, we need to frame it as an optimization problem with a cost function to be minimized by a solver. In this specific case, we want to minimize the Euclidean distances between the points on the template and the searched surface. To estimate the location, we find a transformation from an initial position $g^0(x,y,z)$, as we would do in a registration problem.

For this purpose, we expand Equation 13 using the first-order Taylor expansion:

$$-e(x,y,z) = \frac{\partial g^0(x,y,z)}{\partial x}\,dx + \frac{\partial g^0(x,y,z)}{\partial y}\,dy + \frac{\partial g^0(x,y,z)}{\partial z}\,dz - \left(f(x,y,z) - g^0(x,y,z)\right)$$

with

$$g_x = \frac{\partial g^0(x,y,z)}{\partial x}, \quad g_y = \frac{\partial g^0(x,y,z)}{\partial y}, \quad g_z = \frac{\partial g^0(x,y,z)}{\partial z}$$

which are the first-order derivatives, and dx, dy, dz are the differential components of the 3D transformation.

Expanding further, in general, we get a 3D similarity transformation of the type:

$$-\mathbf{e} = \mathbf{A}\,\mathbf{x} - \mathbf{l}, \quad \mathbf{P} \tag{14}$$

where $\mathbf{A}$ is the design matrix, $\mathbf{P} = \mathbf{P}_{11}$ is the associated a priori weight matrix, $\mathbf{x}$ is the vector of parameters of the transformation to be estimated, and $\mathbf{l} = f(x,y,z) - g^0(x,y,z)$ represents the distance between corresponding points on the two surfaces, with the corresponding model of the noise:

$$e \sim \mathcal{N}(0, \sigma_0\,\mathbf{Q}_{11}), \quad \sigma_0\,\mathbf{Q}_{11} = \sigma_0\,\mathbf{P}_{11}^{-1} = \mathbf{K}_{11} = \mathbb{E}[e\,e^T]$$

The parameters are introduced into the system as observable:

$$-\mathbf{e}_b = \mathbf{I}\,\mathbf{x} - \mathbf{l}_b, \quad \mathbf{P}_b \tag{15}$$

where $\mathbf{P}_b$ is the associated weight matrix and $\mathbf{l_b}$ is the observation vector.

The solution to the system of Equations 14–15 is given by the least squares solution:

$$\hat{\mathbf{x}} = (\mathbf{A}^T \mathbf{P} \mathbf{A} + \mathbf{P_b})^{-1} (\mathbf{A}^T \mathbf{P} \mathbf{l} + \mathbf{P_b} \mathbf{l_b}), \quad \text{solution vector}$$

$$\hat{\sigma}_0^2 = \frac{\mathbf{v}^T \mathbf{P} \mathbf{v} + \mathbf{v}_b^T \mathbf{P}_b \mathbf{v}_b}{r}, \quad \text{variance factor}$$

$$\mathbf{v} = \mathbf{A} \hat{\mathbf{x}} - \mathbf{l}, \quad \text{residual vector for surface observations}$$

$$\mathbf{v}_b = \mathbf{I} \hat{\mathbf{x}} - \mathbf{l_b}, \quad \text{residual vector for additional observations}$$

that is estimated with the covariance matrix:

$$\mathbf{K}_{xx} = \hat{\sigma}_0^2 \mathbf{Q}_{xx} = \hat{\sigma}_0^2 \mathbf{N}^{-1} = \hat{\sigma}_0^2 (\mathbf{A}^T \mathbf{P} \mathbf{A} + \mathbf{P}_b)^{-1}$$

3.3.2 Surface Matching through 3D Features

Another solution to the 3D surface matching problem is presented in Drost et al., 2010, where the proposed algorithm extracts 3D feature points from the template and the search surfaces. During runtime, the features are efficiently queried and compared, and a voting approach is used to estimate the pose, followed by an ICP refinement. This algorithm represents the foundation of state-of-the-art methods to solve this problem and it's the one that will be used in the industrial application of this thesis. The block diagram of the algorithm is shown in Figure 21.

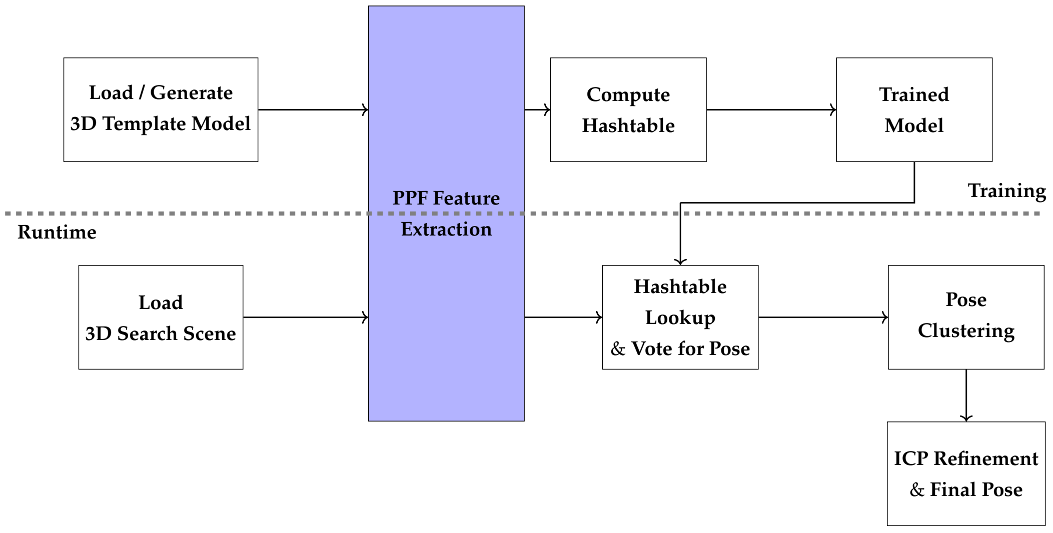


Figure 21: Block scheme of the algorithm.

The algorithm assumes that both the template and the scene object are represented as sets of points with their associated normals, which can be computed from any point cloud or mesh representation. We denote the points in the scene as $\mathbf{s}_i \in \mathcal{S}$ and the points in the template model as $\mathbf{m}_i \in \mathcal{M}$. In the offline training phase, the goal is to build a global descriptor of the template that we can use to find matches. To construct this descriptor, the algorithm calculates a feature for each pair of points that describes their relative positions and orientations. Given two points, $\mathbf{m}_1$ and $\mathbf{m}_2$, and their corresponding normals, $\mathbf{n}_1$ and $\mathbf{n}_2$, with $\mathbf{d} = \mathbf{m}_2 - \mathbf{m}_1$, the feature $\mathbf{F}$ is defined as:

$$\mathbf{F}(\mathbf{m}_1, \mathbf{m}_2) = \left(\|\mathbf{d}\|, \angle(\mathbf{n}_1, \mathbf{d}), \angle(\mathbf{n}_2, \mathbf{d}), \angle(\mathbf{n}_1, \mathbf{n}_2)\right) \tag{16}$$

where $\angle(\mathbf{a}, \mathbf{b}) \in [0, \pi]$ denotes the angle between the two vectors.

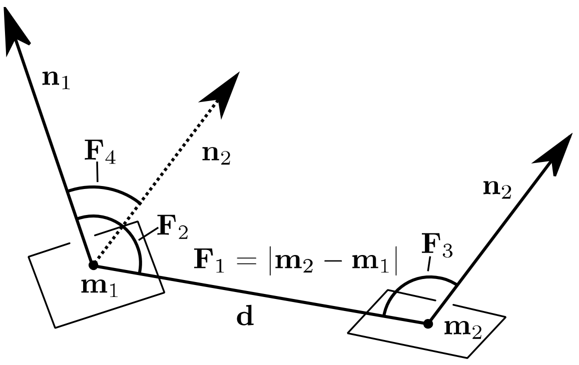


Figure 22: Feature of a point pair. Each feature is given by four components $\mathbf{F}_i$ as represented in the figure.

To build the global descriptor, the feature vector $\mathbf{F}$ is calculated for all point pairs $\mathbf{m}_i, \mathbf{m}_j \in \mathcal{M}$ on the template model surface. The distances and angles are then sampled in steps of $\mathbf{d}_{\text{dist}}, \mathbf{d}_{\text{angle}} = 2\pi/n_{\text{angle}}$, and the identical discretized feature vectors are grouped together. The global descriptor is a mapping from the sampled point pair feature space to the model:

$$L : \mathcal{Z}^4 \to A \subset \mathcal{M}^2$$

where the feature vectors from Equation 16 are mapped to the set A of all pairs $(\mathbf{m}_i, \mathbf{m}_j) \in \mathcal{M}^2$ that have an identical feature vector. Figure 23 illustrates this mapping with an example.

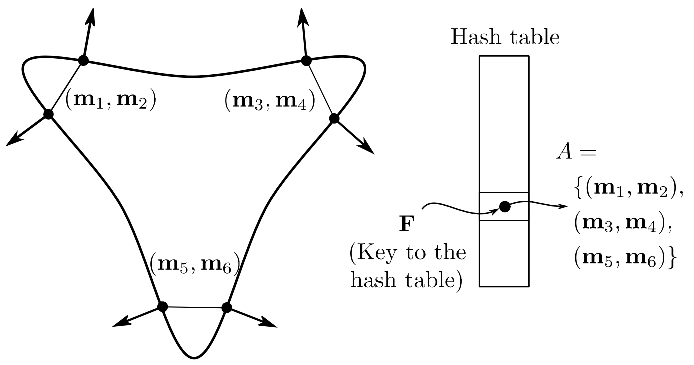

$(\mathbf{m}_1, \mathbf{m}_2)$
$(\mathbf{m}_3, \mathbf{m}_4)$
$(\mathbf{m}_5, \mathbf{m}_6)$
Hash table
$\mathbf{F}$
(Key to the hash table)
$A = \{(\mathbf{m}_1, \mathbf{m}_2), (\mathbf{m}_3, \mathbf{m}_4), (\mathbf{m}_5, \mathbf{m}_6)\}$

Figure 23: Global model descriptor implemented with a hash table.

In the algorithm, this mapping is implemented using a hash table that efficiently queries the features of a scene point pair and responds in constant time with the set of template model point pairs that have a similar feature.

In the ideal case, for any point $\mathbf{s}_r \in \mathcal{S}$, there should be a corresponding point $\mathbf{m}_r \in \mathcal{M}$ in the template model. Once this correspondence is found, aligning the two objects would be achieved by aligning the two points and then rotating the scene around the normal of $\mathbf{s}_r$ by an angle α. The pair $(\mathbf{m}_r, \alpha)$ is referred to as the local coordinates of the model with respect to $\mathbf{s}_r$. After constructing the hash table, the algorithm matches two pairs $(\mathbf{s}_r, \mathbf{s}_i) \in \mathcal{S}^2$ and $(\mathbf{m}_r, \mathbf{m}_i) \in \mathcal{M}^2$ that share the same discretized feature vector $\mathbf{F}$. The alignment transformation from the model coordinates to the scene coordinates is given by (Figure 24):

$$\mathbf{s}_i = T_{s\to g}^{-1} \, R_x(\alpha) \, T_{m\to g} \, \mathbf{m}_i \tag{17}$$

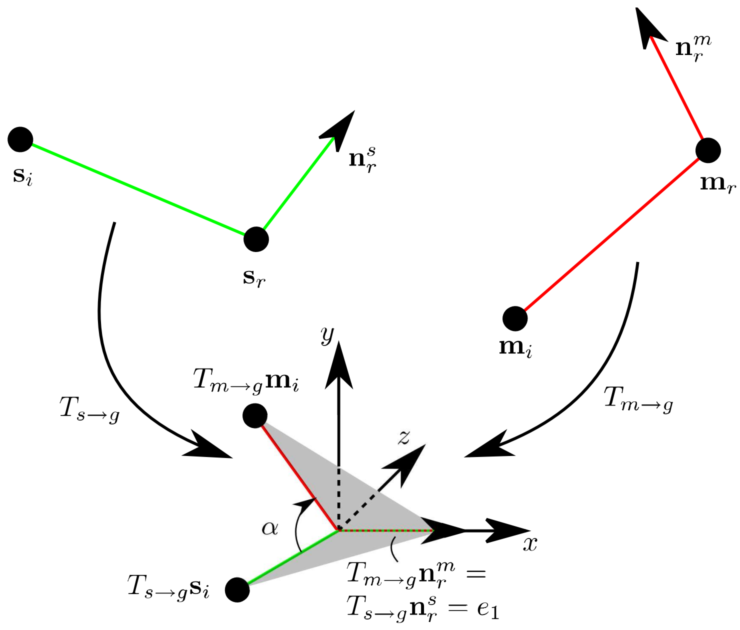


Figure 24: Transformation of template-scene coordinates.

Given a point $\mathbf{s}_r$, the algorithm tries to find the best local coordinates that maximize the number of points in the scene that align with the model using a voting scheme. Once the optimal coordinates are identified, the global pose of the object can be determined. To achieve this, for each scene pair $(\mathbf{s}_r, \mathbf{s}_i)$, the corresponding pair feature $\mathbf{F}_s(\mathbf{s}_r, \mathbf{s}_i)$ is computed and used as a query to the hash table to find all template model pairs that share the same discretized feature. For each of these correspondences, the local coordinate α is computed by solving Equation 17. After α is calculated, a vote is cast and collected in the accumulator space. Each reference point then returns the peak of its accumulator array, representing a set of possible object poses.

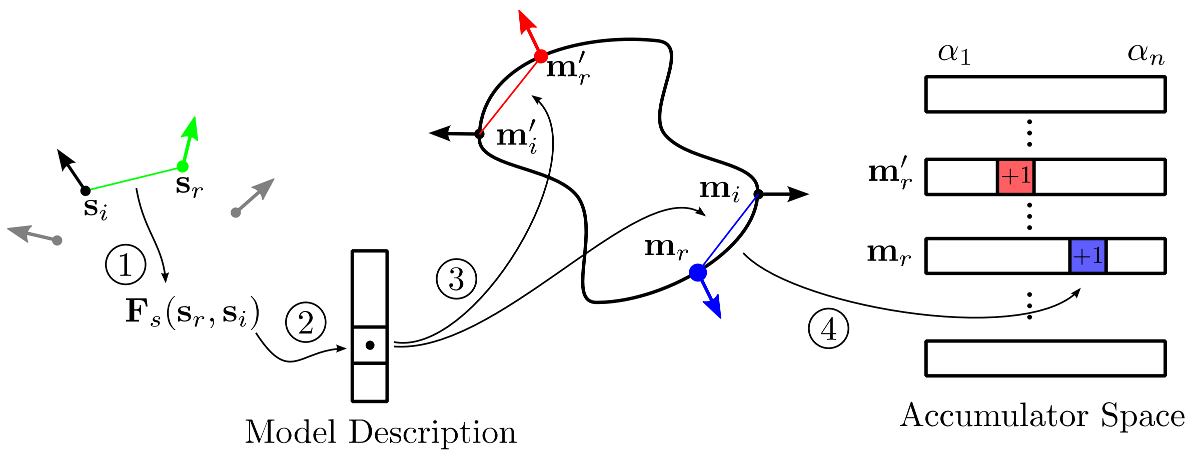
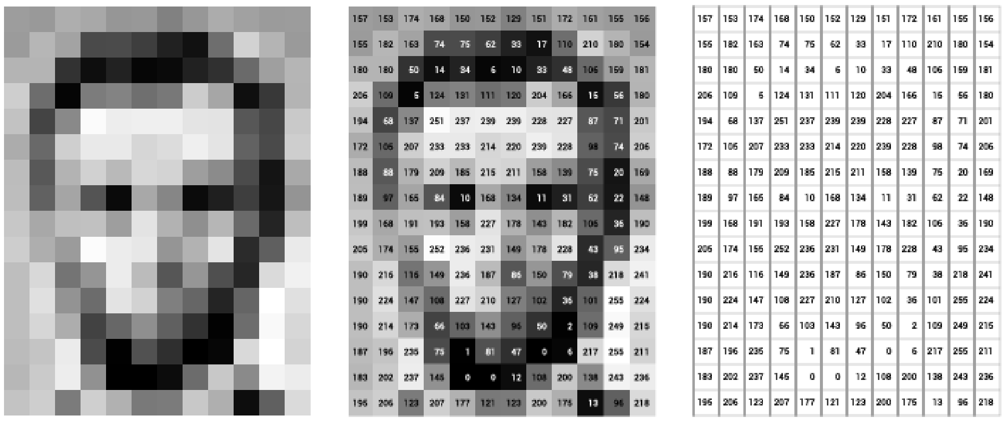


Figure 25: Voting scheme for determining object poses.

Finally, to enhance the robustness of the algorithm, these poses are filtered using pose clustering, and the final pose is refined through an ICP stage.

3.4 DEEP LEARNING ON 3D DATA

Deep learning is a branch of machine learning that, compared with traditional machine learning methods, avoids the need for manually designing features. The current main deep learning methods are based on **neural networks**, which have the ability to directly learn patterns from training data without manually designing the features to be extracted. This takes advantage of large amounts of data and can provide an end-to-end solution for many computer vision tasks. In recent years, deep learning has shown state-of-the-art results in tasks that use various types of data such as images, text, and audio. This success, coupled with the growth of available 3D data, has led to the rise of a new field that aims to apply deep learning to 3D data. Fundamental to this rise was the fact that 3D sensors became progressively cheaper, leading to the availability of large-scale 3D datasets with annotated data.
However, unlike traditional data that has a unique or dominant representation, 3D data has many representations (point clouds, volumetric, polygonal meshes, etc.).



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Figure 26: Unique representation of 2D images.

This represents a fundamental issue, as the most common type of neural network uses convolutional layers that exploit the regularity of the data. Therefore, we cannot directly utilize the full power of conventional neural networks for the most common types of 3D data, such as point clouds and meshes. On the other side, regular structures for 3D data, such as volumetric representations like voxels, can be directly used with 3D kernels. However, this presents further challenges in terms of computational demands.

This inspires two main branches of methods to consume and analyze 3D data using neural networks. One branch first converts the irregular structures into regular structures and then applies convolutional-based neural networks with 3D kernels. The other branch directly uses the irregular data, taking advantage of the permutation invariance of the points in the cloud.

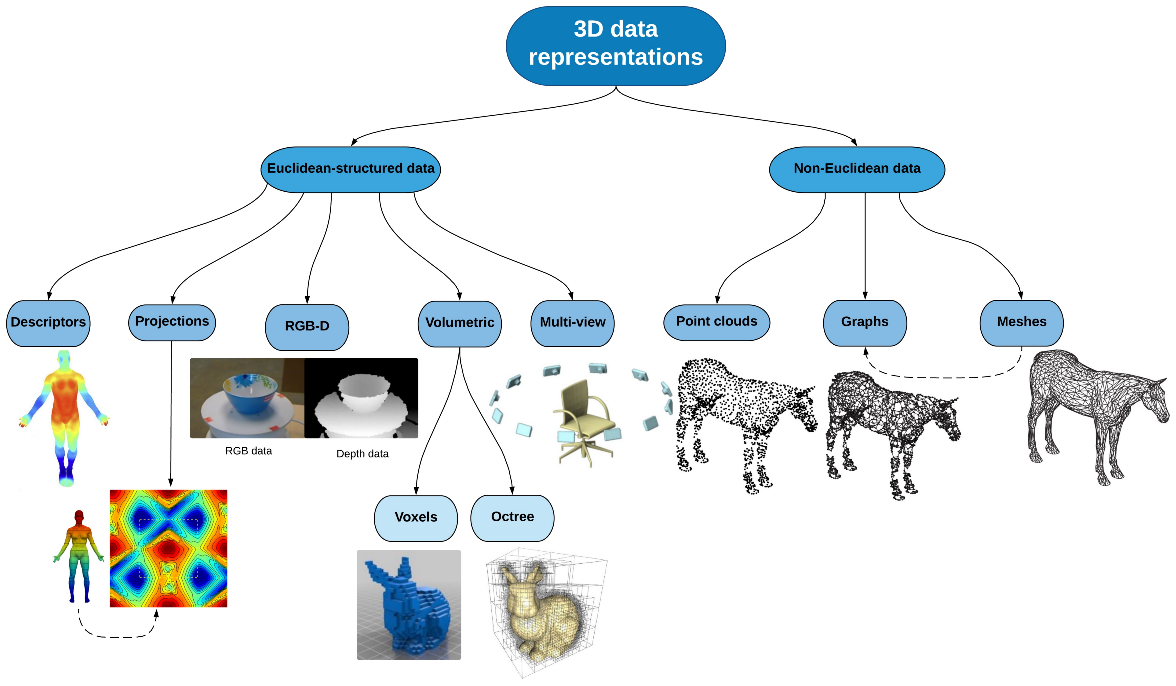


Figure 27: Regular vs Irregular representations for 3D data Ahmed et al., 2019.

3.4.1 Deep learning tasks

Deep learning can be used to solve a lot of applications that use 3D data. We can divide these applications into:

1. **3D analysis** where the 3D data is fed to the neural network to analyze its properties. Some typical tasks include classification, semantic segmentation, and detection of objects within a scene.

2. **3D synthesis** where the neural network is used to generate 3D data from lower-dimensional representations or from embeddings produced by an encoder network. Some typical tasks include shape completion and shape modeling.

3. **3D assisted image understanding** where the 3D data is not directly used as input to the network but is instead used as additional information to assist tasks that would be difficult to solve otherwise.

In this thesis, I will concentrate as an example on the 3D analysis case, in particular to solve classification tasks.

3.4.2 Deep Learning on Regularly Structured Data

Sensor data typically arrives in the form of point clouds or depth maps, which can be transformed into meshes through triangulation techniques. To utilize convolutional neural networks (CNNs) for learning deep features from this type of irregular data, it first needs to be converted into a regular structure. The two main methods to operate this conversion are the projection or rendending of the 3D object from multiple viewpoints to generate 2D images and the voxelization process. In the projection or rendering method, the 3D object is taken from various angles and the 2D craeted images can be used as inputs to traditional CNNs, and finally the outputs from each viewpoint can then be combined using a view pooling technique to infer properties of the original 3D model.

The second method, voxelization, involves quantization of 3D point clouds or meshes into a volumetric grid. The simplest form of voxelization creates a binary occupancy grid, where each voxel is assigned a value of zero or one that indicates the absence or presence of points within the voxel's space. Once the voxel grid is created, CNNs can be applied directly using 3D kernels.

Multi-view CNN

Before the spread of 3D sensors, a big challenge in computer vision was inferring 3D object properties from 2D images. With access to 3D data, one effective approach is to voxelize the data and use volumetric CNNs or employ neural networks that can process irregular data directly. Another proven method is rendering the 3D object from multiple viewpoints to create a set of 2D images, which can then be analyzed using conventional 2D CNNs.

This approach has shown good results due to the relative efficiency of 2D representations compared to 3D ones H. Su et al., 2015. In order to actually use voxel grids and train deep neural networks with reasonable computational power, it's usually necessary to use a coarse grid. Instead, a single viewpoint can utilize much higher resolution images and still require less training time. In addition, the 2D image representation allows us to leverage the wide research on image descriptors, access very large labeled image datasets, and use numerous pre-trained CNN architectures designed for images.

While the advantages of 2D image representation are less evident when we have available large volumes of 3D data and substantial computational power, it remains a viable option in many scenarios.

The general pipeline of a multi-view CNN (Figure 28) begins with the generation of multiple views of a 3D shape using a rendering engine. Each of these views is then processed by a CNN to extract features and produce a 2D descriptors for each view. These view descriptors are subsequently passed to a view pooling layer, which combines the features from the different views to create a compact representation of the 3D shape. This descriptor can then be used as input to a neural network to process the global information and afterwards the output of this network is flattened and fed to a classifier, which in the simplest case is a dense layer with a softmax output function.

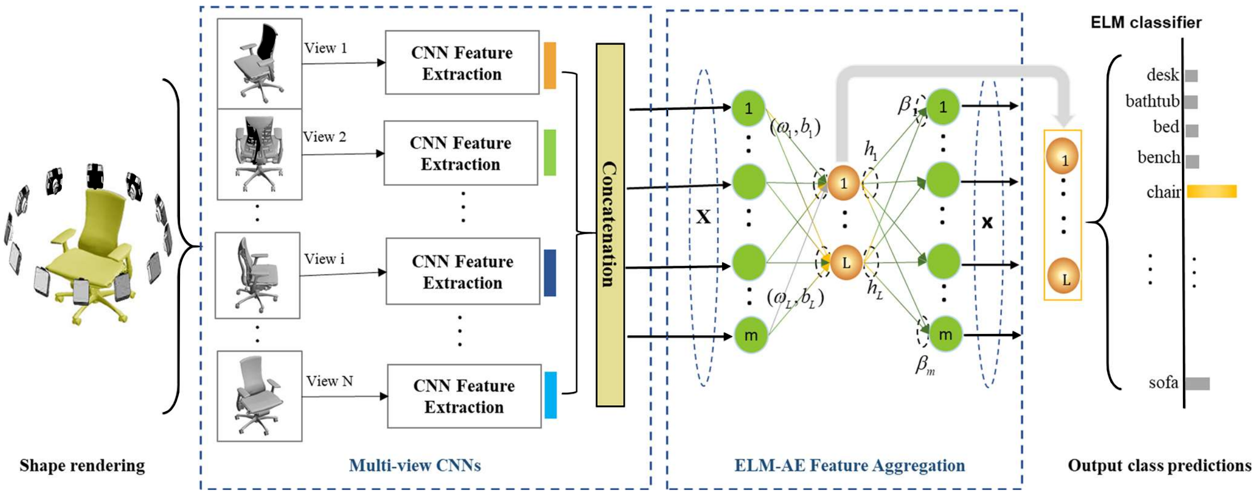


Figure 28: Multi view CNN Yang et al., 2018

Volumetric CNN

Volumetric CNN are an alternative way of using the 3D data as input to the neural network. In order to exploit the power of convolutional layers for feature extraction, the irregular data coming from sensors must be first turned into a regular grid through a process called **voxelization**. There are many ways of creating a voxel grid depending on the input data:

1. **discrete "deterministic" hit grids** are the simplest form of voxelization, which is done by computing a binary grid based on whether or not there is a point in a voxel. If we also know the pose of the sensor (for example, a LiDAR or a depth camera) that produced the point cloud, we can encode the voxel as free, occupied, or unknown through 3D ray tracing.

2. **discrete "probabilistic" hit grids** refers to the case in which the occupancy grid is computed using a probabilistic model.

3. **continuos density grids** consider each voxel to have a continuous density, representing the likelihood that the voxel would obstruct a sensor beam.

In Thrun, 2003 the author shows how to acquire occupancy grids with mobile robots.
Let m be the occupancy grid map that is estimated from the sensor measurements. Let $z_1, \dots, z_T$ be the measurements from time 1 to time T. Each measurement carries information about the occupancy of many grid cells.
The problem becomes:

How can we find the probability of occupancy of each cell m $p(m|z_1, \dots, z_T)$ ***given the measurements*** $z_1, \dots, z_T$***?***

The one dimensional problem corresponds to the estimation $p(m_{x,y}|z_1,\ldots,z_T)$. For numerical reasons we estimate instead the *log-odds*:

$$l^T_{x,y} = \log \frac{p(m_{x,y}|z_1,\ldots,z_T)}{1-p(m_{x,y}|z_1,\ldots,z_T)} \quad \Rightarrow p(m_{x,y}|z_1,\ldots,z_T) = 1-[e^{l^T_{x,y}}]^{-1}$$

Using Bayes rule we can estimate the log-odds at any time t:

$$p(m_{x,y}|z_1,\ldots,z_T) = \frac{p(z_t|z_1,\ldots,z_{t-1},m_{x,y})\,p(m_{x,y}|z_1,\ldots,z_{t-1})}{p(z_t|z_1,\ldots,z_{t-1})} \tag{18}$$

Using the common static world assumption

$$p(z_t|z_1,\ldots,z_{t-1},m) = p(z_t|m)$$

so that we can simply Equation 18

$$p(m_{x,y}|z_1,\ldots,z_T) = \frac{p(z_t|m_{x,y})\,p(m_{x,y}|z_1,\ldots,z_{t-1})}{p(z_t|z_1,\ldots,z_{t-1})}$$

Applying once again Bayer we get the probability that $m_{x,y}$ is occupied

$$p(m_{x,y}|z_1,\ldots,z_T) = \frac{p(m_{x,y}|z_t)\,p(z_t)\,p(m_{x,y}|z_1,\ldots,z_{t-1})}{p(m_{x,y})\,p(z_t|z_1,\ldots,z_{t-1})} \tag{19}$$

Analogously we can get the cell is free denoted by $\bar{m}_{x,y}$

$$p(\bar{m}_{x,y}|z_1,\ldots,z_T) = \frac{p(\bar{m}_{x,y}|z_t)\,p(z_t)\,p(\bar{m}_{x,y}|z_1,\ldots,z_{t-1})}{p(\bar{m}_{x,y})\,p(z_t|z_1,\ldots,z_{t-1})} \tag{20}$$

We can eliminate some term by dividing Equation 19 by Equation 20

$$\frac{p(m_{x,y}|z_1,\ldots,z_T)}{p(\bar{m}_{x,y}|z_1,\ldots,z_T)} = \frac{p(m_{x,y}|z_t)}{p(\bar{m}_{x,y}|z_t)}\,\frac{p(\bar{m}_{x,y})}{p(m_{x,y})}\,\frac{p(m_{x,y}|z_1,\ldots,z_{t-1})}{p(\bar{m}_{x,y}|z_1,\ldots,z_{t-1})}$$

and using the normalization property of probability distributions we can rewrite as follows:

$$\frac{p(m_{x,y}|z_1,\ldots,z_T)}{1-p(m_{x,y}|z_1,\ldots,z_T)} = \frac{p(m_{x,y}|z_t)}{1-p(m_{x,y}|z_t)}\,\frac{1-p(m_{x,y})}{p(m_{x,y})}\,\frac{p(m_{x,y}|z_1,\ldots,z_{t-1})}{1-p(m_{x,y}|z_1,\ldots,z_{t-1})}$$

taking the logarithm we get the target log-odds:

$$\log\frac{p(m_{x,y}|z_1,\ldots,z_T)}{1-p(m_{x,y}|z_1,\ldots,z_T)} = \log\frac{p(m_{x,y}|z_t)}{1-p(m_{x,y}|z_t)} + \log\frac{1-p(m_{x,y})}{p(m_{x,y})}$$
$$+ \log\frac{p(m_{x,y}|z_1,\ldots,z_{t-1})}{1-p(m_{x,y}|z_1,\ldots,z_{t-1})}$$

Substituting the log-odds we get the recursive formulation:

$$l_{x,y}^{t} = \log \frac{p(m_{x,y}|z_t)}{1 - p(m_{x,y}|z_t)} + \log \frac{1 - p(m_{x,y})}{p(m_{x,y})} + l_{x,y}^{t-1}$$

with initialization

$$l_{x,y}^{0} = \log \frac{p(m_{x,y})}{1 - p(m_{x,y})}$$

Finally solving in closed form

$$\boxed{l_{x,y}^{T} = (T-1) \log \frac{1 - p(m_{x,y})}{p(m_{x,y})} + \sum_{t=1}^{T} \frac{p(m_{x,y}|z_t)}{1 - p(m_{x,y}|z_t)} \qquad (21)}$$

where $p(m_{x,y}|z_t)$ requires an inverse sensor model that maps the sensor measurements to its causes.
Once the voxel grid is obtained, it is relatively straightforward to extend 2D convolutional network techniques to 3D data. However, it is not obvious which architecture will yield optimal performance, and the volumetric representation can become computationally intensive. This must be taken into condiration for the design of the architecture.

One of the most knows architectures that uses voxel grids for shape classification is VoxNet Maturana and Scherer, 2015.

This architecture fundamentally focuses on extracting hierarchical features from the 3D grid input to obtain a global descriptor that can be used by the classification network.
The network (Figure 29) takes an input point cloud and transforms it into a low resolution occupancy grid with dimensions of 32x32x32. It then applies rotation and translation augmentation. The baseline architecture consists of a convolutional layer with 3D filters followed by max pooling. The output is then flattened and passed to a feedforward layer for classification. The softmax function with categorical cross-entropy loss is used as the output function.

In Chapter 4, I will present the end-to-end implementation of a modified VoxNet structure.

3.4.3 Deep Learning on Point Clouds

The majority of 3D data from sensors or 3D acquisition techniques is produced in the form of point clouds. Therefore, point clouds represent a particularly important type of representation for 3D data. As discussed in previous sections, many neural networks convert the irregular structure of point clouds into a regular format. However, this approach imposes constraints on the resolution and computational power that has led to the development of a new type of neural network that directly uses point clouds and takes into consideration their permutation invariance. One of the most used architectures of this type is PointNet Charles et al., 2017, which provides a unified framework for classification and part segmentation.

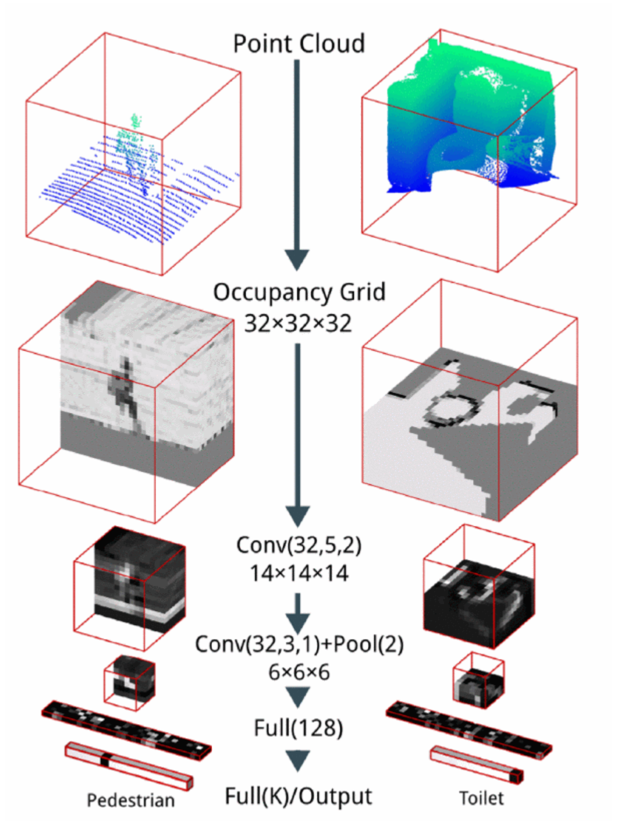


Figure 29: VoxNet structure Maturana and Scherer, 2015

Even if point clouds are a straightforward representation, designing a neural network architecture that can utilize them in a efficient way presents three main challenges:

1. The model must be designed to account for the fact that a point cloud is a set of points that remains invariant under permutations.

2. The points belong to an Euclidean space with a defined distance metric. This means that they are not isolated, and the relationships between adjacent points are important. The network should then be able to capture these local structures.

3. Since point clouds are geometric objects, the model must produce labels that are invariant to transformations of the input, such as rotations and translations.

The key design element for this approach is the symmetric function, max pooling. The network learns to select important parts of the point clouds and encodes that information into features, which are then processed by fully connected layers.

A point cloud is represented by a subset of 3D Euclidian points $\{P_i | i = 1, \ldots, n\}$ where each point P_i is a vector of at least three coordinates (x, y, z). To create a transformation invariant to permutation the idea of the authors Charles et al., 2017 is to approximate a function applied to a set of points with a symmetric function on transformed elements in the set:

$$f(\{x_1, \ldots, x_n\}) \approx g(h(x_1), \ldots, h(x_n))$$
$$f : 2^{\mathbb{R}^N} \rightarrow \mathbb{R}, \quad h : \mathbb{R}^N \rightarrow \mathbb{R}^K, \quad g : \underbrace{\mathbb{R}^K \times \cdots \times \mathbb{R}^K}_{n} \rightarrow \mathbb{R}$$

where $g(\cdot)$ is a symmetric function.

The core component of PointNet tries to approximate the function h using a multi-layer perceptron network, while the function g is approximated using a combination of a univariate function and a max pooling operation.

To achieve invariance to transformations, PointNet uses a spatial transformer network to convert the data into a canonical form for processing. By training the T-net along with the main network, it learns to apply geometric transformations that align point clouds with good accuracy, resulting into much easier subsequent classification tasks. This concept can be extended to the alignment in the feature space by using another alignment network that predicts a feature transformation matrix to align features from different input point clouds. However, the feature space transformation matrix has a much higher dimension, making the optimization more challenging. Therefore, PointNet includes a regularization term that constrains the feature transformation matrix to remain close to an orthogonal matrix:

$$L_{reg} = \|I - A A^T\|_F^2$$

where A is the feature transform learned by the network. This regularization is proven to make the optimization faster and more stable.

In Figure 30 we can see the full architecture of PointNet.

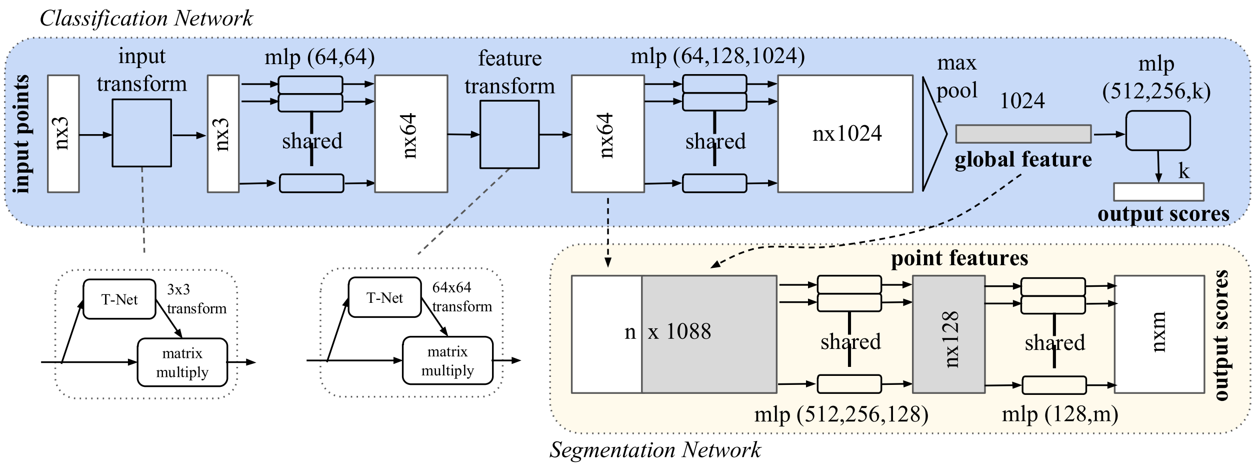


Figure 30: PointNet structure Charles et al., 2017

Part II

SECOND PART: DEEP LEARNING APPLICATION USING 3D DATA

This part shows the design of a deep learning application for 3D classification. The design is inspired by the Orientation Boosted Voxel Net, but the entire architecture of the network has been changed and skip connections were introduced. The implementation in Python from the augmentation of the data, the creation of the dataset, the model and the training and evalution of the network is provided in its key parts.

4 RESIDUAL ORIENTATION-BOOSTED CNN FOR 3D CLASSIFICATION

In this chapter I present a novel 3D classification network architecture inspired by the Orientation Boosted Voxel Net (ORION), in which I incorporate skip connections similar to the 2D ResNet architecture. Utilizing the ModelNet10 dataset, the network processes aligned 3D CAD models, which are augmented by rotating each mesh and voxelizing them into binary grids. The architecture is designed to classify both the object and its rotation, utilizing a combined categorical cross-entropy loss for both tasks. Skip connections are chosen to enable deeper network design without vanishing gradient problems, leading to more abstract and effective feature extraction. During testing, a voting approach is used, where each input is rotated multiple times, and the final classification is based on the label with the highest overall sum class probability from these rotations. This strategy is used to enhance the network's robustness against challenging perspectives. The experimental results demonstrate that the proposed architecture achieves 92.3% accuracy on the test set, surpassing both my implementation of the original and extended ORION models. This improvement shows the efficacy of using skip connections in 3D data processing and serves as a study for future more elaborated networks.

This project was implemented in PyTorch using the torch and Open3D libraries. It is organized into files to follow a scalable approach. Specifically, I provide a class file for dataset creation and a class file for the models definition. Additionally, I provide three executable files to be run in sequence: one file to voxelize and augment the initial dataset offline, one file to train the model, and one file to evaluate its performance.

4.1 INTRODUCTION AND RELATED WORK

In recent years, deep learning has demonstrated state-of-the-art results in tasks involving various types of data such as images, text, and audio. This success, combined with the increasing availability of 3D data, has led to the emergence of a new field focused on applying deep learning to 3D data. A key factor in this development has been the decreasing cost of 3D sensors, resulting in very large 3D datasets with annotated data becoming more accessible. However, unlike traditional data with a unique or dominant representation, 3D data comes in multiple forms (e.g., point clouds, volumetric data, polygonal meshes), some of which are irregular. This represents a primary challenge, as the most common type of neural network uses convolutional layers that rely on the regularity of the data.

This challenge has given rise to two main approaches for processing and analyzing 3D data with neural networks:

- One approach converts the irregular structures into regular structures and then applies convolutional neural networks with 3D kernels.
- The other approach directly uses the irregular data such as point clouds, taking into account the permutation invariance of the points in the cloud.

Even if point clouds are a straightforward representation, designing a neural network architecture that can effectively utilize them presents many challenges, as the model must be invariant to permutations of the points as well as transformations such as rotations and translations of the point cloud. On the other side, converting these irregular structures into a regular grid allows the extension of 2D convolutional network techniques to 3D data in a relatively simple manner. However, it is not clear which architecture will yield optimal performance, and the volumetric representation can become computationally intensive.

In this thesis, I present a novel approach for 3D object classification that draws inspiration from the Orientation Boosted Voxel Net network Sedaghat et al., 2017, while implementing a new architecture that incorporates skip connections similar to those used in the 2D ResNet He et al., 2016 architecture.

I start by using the ModelNet10 dataset, composed of aligned 3D CAD models divided into 10 classes, and I perform data augmentation by rotating each mesh 12 times in steps of 30°. Each rotated mesh is then voxelized into a 32x32x32 binary grid and fed to the network, which outputs both the class label and the rotation class through a softmax output function with a combined categorical entropy loss for both tasks. The expectation is that forcing the network to also learn the orientation of the object will help the classification task, and by using skip connections, the network can be made deeper without encountering the vanishing gradient problem and thus producing more abstract features that can improve the network's accuracy.

After this, during test time, each input is rotated multiple times and fed to the network, producing multiple output labels for the same object. The chosen label is the one that maximizes the class sum probability according to the softmax output function. This approach has the goal to provide the network with the full view of the object to avoid misclassifications due to challenging perspectives.

4.2 PROCESSING PIPELINE

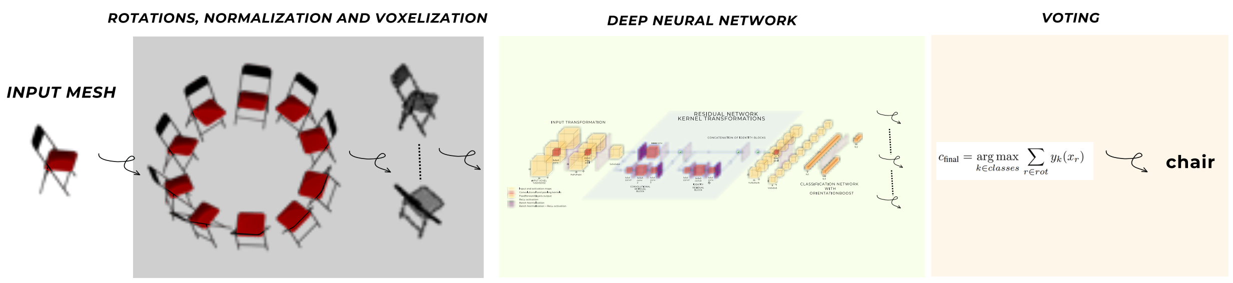

ROTATIONS, NORMALIZATION AND VOXELIZATION
DEEP NEURAL NETWORK
VOTING
INPUT MESH
RESIDUAL NETWORK KERNEL TRANSFORMATIONS
INPUT TRANSFORMATION
CLASSIFICATION NETWORK WITH ORIENTATIONBOOST
$c_{\text{final}} = \arg\max_{k \in classes} \sum_{r \in rot} y_k(x_r)$
chair

Figure 31: Processing pipeline

Pre-processing

Data plays a crucial role in the accuracy of a machine learning model. In particular, the data must be as representative as possible of the entire input distribution, and the samples should be drawn independently from each other whenever possible. For this study, the input data consisted of 3D meshes taken from the ModelNet10 dataset. The meshes for each class were manually aligned around the z-axis by the authors. The first preprocessing step was to take each mesh and create rotated copies around the z-axis with fixed increments. This is useful for both:

- Ensuring that the input rotation distribution is well-sampled across the entire dataset, creating the labels to solve the rotation task.
- Augmenting the initial dataset with more data, which can also help with the class label task.

Each rotated mesh is then normalized to make it fit the unit cube, helping to avoid the input covariance shift, making the training more robust and faster. The normalized meshes are then voxelized and saved separately in a voxelized dataset that will be used for the training. This helps to reduce the memory occupation and speeds up the training since the transformation process is done offline. The data is then sampled randomly in batches to make each batch as i.i.d. as possible.

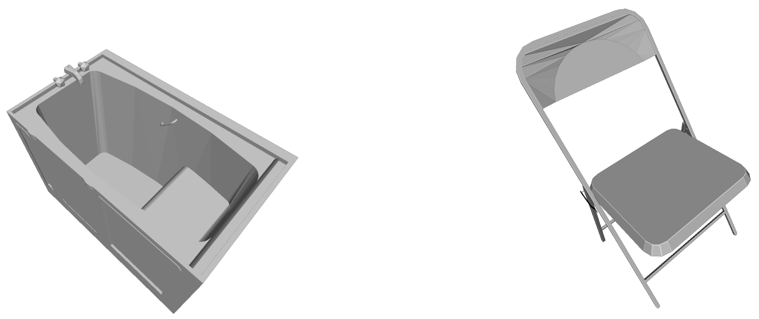
Models, Training and Evaluation

The model architecture consists of input transformation layers that end with a max pooling layer to help with invariance with respect to the translation of the objects inside the grid. The feature maps are then processed by a 3D residual network with skip connections, further processed with average pooling, and then flattened to be fed to an orientation boosted classification network that produces both the class and rotation labels. To prevent overfitting, dropout layers were used. During testing, I implemented a voting approach where each input is rotated to different angles around the z-axis, and the label corresponding to the highest sum of outputs across all orientations is taken.

4.3 DATASET

Input Dataset

The starting point dataset I used is ModelNet10, which contains 4,899 synthetic CAD models from 10 different classes. The dataset is split into 3,991 (80%) shapes for training and 908 (20%) shapes for testing. All the objects within a class are manually aligned by the authors. The provided objects are in OFF files that can be read and encapsulated in a mesh model using Open3D functions.



(a) *Bathtub object.* **(b)** *Chair object.*

Figure 32: Example of input meshes taken from ModelNet10.

Data Augmentation

From each input mesh, I created 12 rotations around the z-axis with increments of 30°. This procedure produces the orientation labels used for training and augments the initial dataset while maintaining the same class proportions. During training, the imbalance in the number of points for each class is compensated by using different weights for each class in the loss function.

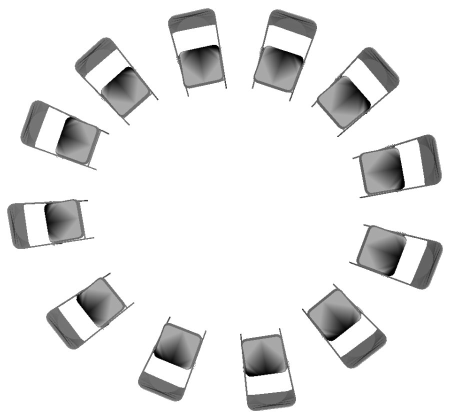


Figure 33: Augmentation of the chair object.

Voxelization

Each rotated mesh is then normalized to fit the unit cube to avoid input covariance shift. The normalized mesh is then voxelized. There are many ways to perform this process, depending on the nature of the input data. These methods are mainly:

- Discrete "deterministic" hit grids: This is the simplest form of voxelization, done by computing a binary grid based on whether or not there is a point in a voxel.
- Discrete and continuous "probabilistic" grids: In this case, a probabilistic model is used to produce either a binary grid or a continuous grid. For example, with sensor data, each voxel could be considered to have a continuous density, representing the likelihood that the voxel would obstruct a sensor beam.

In this case, using synthetic data, I employed a deterministic approach with Open3D functions, creating binary grids.

4.4 LEARNING FRAMEWORK

This section presents all the details of the network implementation, beginning with the architecture and including the activation functions used, as well as the training loss and the optimization algorithm used. Figure 34 illustrates the entire architecture, and Table 2 lists all the hyperparameters of the layers.

4.4.1 Skip connections

Deep neural networks have constituted a big breakthrough for many applications for their ability to automatically extract meaningful features from large amounts of data and process them in short time obtaining good results. Building on this success, we would like to implement even deeper neural networks for several reasons:

- Deeper networks have greater capacity and hierarchical structure.
- Features computed by deeper layers are more abstract and, therefore, generally more effective at discriminating complex structures.
- The efficacy of deeper networks is supported by empirical evidence.

Given the significance of depth, we aim to stack more layers. However, this leads to the problem of vanishing gradients, as the network is trained using backpropagation and the gradient must be propagated through all the activation functions of the layers. This makes the training of deep neural networks challenging to the point where the addition of more layers and the increase of network depth instead of constituting an improvement can negatively impact the performance of the network.

This is evident in Figure 35, where the authors He et al., 2016 have shown that the training loss and accuracy become worse when identity layers are added to a network to increase its depth. These identity layers, which theoretically should not affect the network's output, instead lead to a decline in the performance.

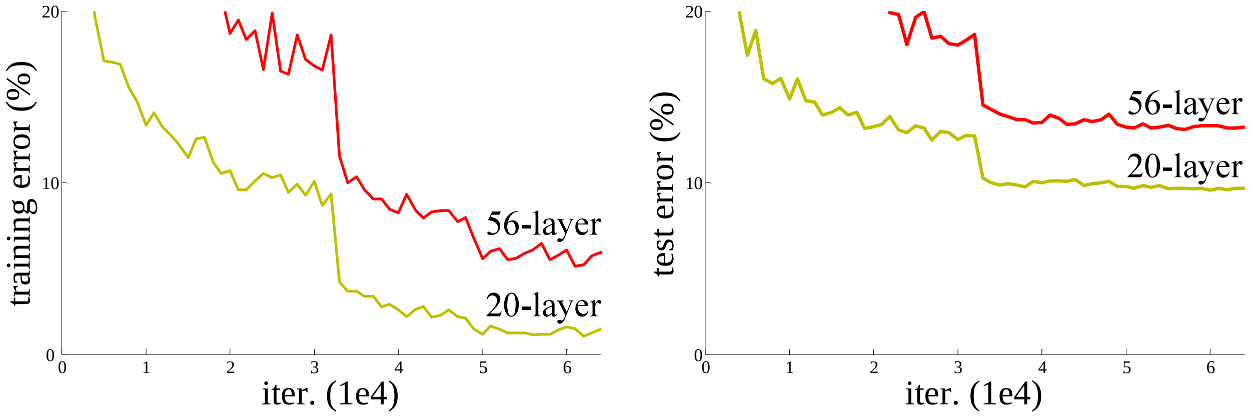


Figure 35: Vanishing gradient with deeper neural networks He et al., 2016.

Several solutions have been shown to help with this problem, such as:

- The choice of the nonlinearity of the activation functions.
- Batch normalization.
- Proper initialization of the weights.

Even if these techniques are helpful, they are not sufficient to completely solve the problem.

Recently, a breakthrough for solving the vanishing gradient problem was proposed with the introduction of skip connections He et al., 2016. This approach forces the network to learn the residual with respect to the input, which is typically a small quantity:

$$\mathcal{F}(\mathbf{x}) = \mathbf{o} - \mathbf{x}$$

Figure 35 shows the proposed residual unit.

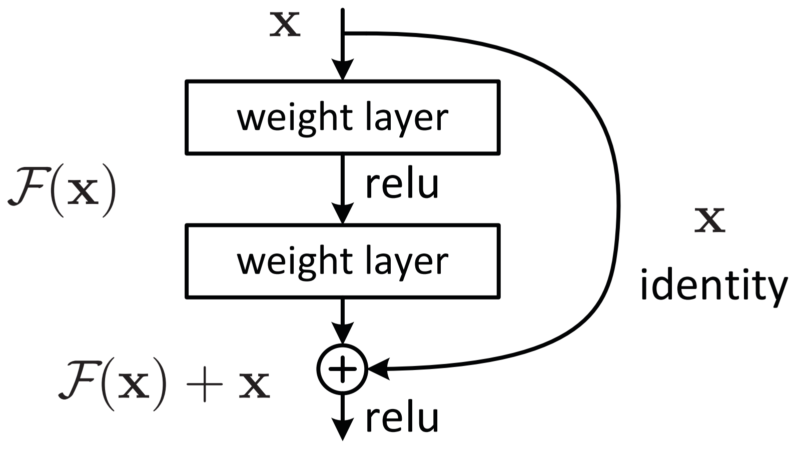


Figure 36: Residual block He et al., 2016.

Each residual unit can be expressed as:

$$x^{l+1} = f\left(\mathcal{F}(\mathbf{x}^l, \mathcal{W}^l) + h(\mathbf{x}^l)\right)$$

where $\mathbf{x}^l$ is the input to the l-layer, $\mathcal{W}^l$ is the set of weights of the l-layer, $\mathbf{x}^{l+1}$ is the input to the $(l+1)$-layer and

$$f(\cdot) = \text{ReLU}(\cdot), \qquad h(\mathbf{x}) = \mathbf{x}, \qquad \mathcal{F}(\mathbf{x}^l, \mathcal{W}^l) = \text{ network layer}$$

Further studies have shown that bringing the nonlinearity inside the network layer can further improve the gradient flow. The best choice of the function is then obtained by setting $f(\cdot)$ and $h(\cdot)$ to the identity function, so the layer can be written as follows:

$$\mathbf{x}^{l+1} = \mathcal{F}(\mathbf{x}^l, \mathcal{W}^l) + \mathbf{x}^l$$

From the equation above, one can derive the expression of the output of the network $\mathbf{x}_L$ as a function of the input to layer l:

$$\mathbf{x}_L = \mathbf{x}_l + \sum_{i=l}^{L-1} \mathcal{F}(\mathbf{x}^i, \mathcal{W}^i)$$

from which the expression of the derivative of the loss computed at layer L with respect to $\mathbf{x}_l$ is

$$\frac{\partial \mathcal{L}}{\partial \mathbf{x}_l} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_L}\frac{\partial \mathbf{x}_L}{\partial \mathbf{x}_l} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}_L}\left(1 + \frac{\partial}{\partial \mathbf{x}_l}\sum_{i=l}^{L-1} \mathcal{F}(\mathbf{x}^i, \mathcal{W}^i)\right)$$

From the red-highlighted term, we can see that it is unlikely for a minibatch to completely cancel out the gradient. This implies that the gradient of the layer l does not vanish, even when the weights are small.

In Figure 37 we can see that using residual blocks reverses the behavior observed in Figure 35. With residual blocks, deeper networks not only fit the training set better but also achieve improved accuracy during inference.

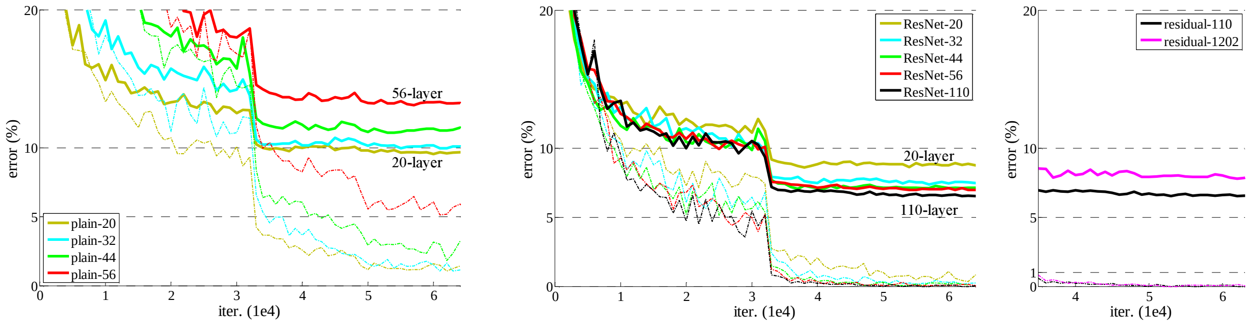


Figure 37: Improved accuracy with skip layers He et al., 2016.

4.4.2 Model Architecture

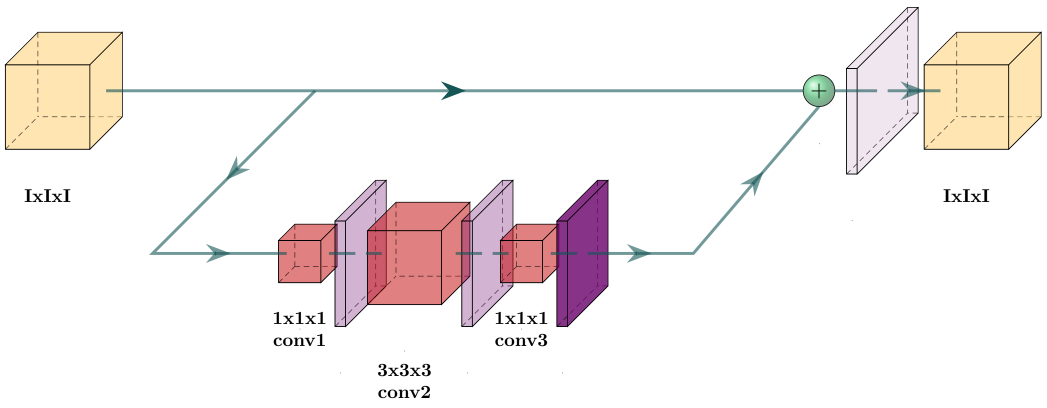
	Conv1	Pool1	ConvRes1	IdRes1	ConvRes2	IdRes2	Pool2	fc1	fc2	fc3
# of filters	64		(64,64,128)	(64,64,128)	(64,64,128)	(64,64,128)				
kernel size	(3,3,3)	(2,2,2)	(3,3,3)	(3,3,3)	(3,3,3)	(3,3,3)	(2,2,2)			
# of repetitions			1	2	1	3				
stride	1	2					2			
padding	0	0				0				
batch norm	✓	×					×	×	×	×
# of outputs								1024	10	120
dropout ratio	×	×	0.2	×	0.2	×	×	0.1	×	×

Table 2: Details of the architecture.

Layers, output function and voting

The network takes as input a voxel grid, which is first transformed by a convolutional layer and a max pooling layer. These layers render the architecture invariant to small translations of the input within the grid. The novelty of the architecture is the use of skip layers. There are two main blocks used:

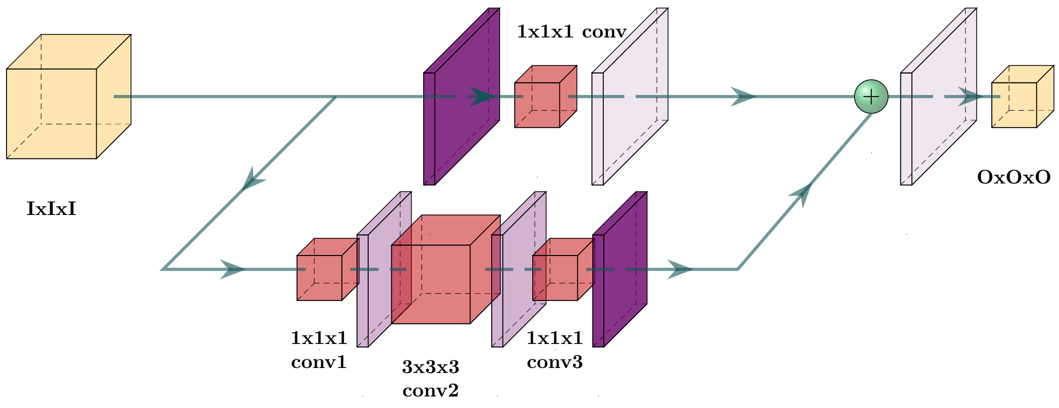
- **Identity residual block**: The main path consists of three convolutional layers that output activation maps of the same size as the input, allowing them to be added in a sum block to learn the residual and favor gradient propagation through the network. To achieve the same size as the input, the first and third kernels are of dimension $1 \times 1 \times 1$ with unitary stride, while the second kernel has a dimension that can be changed using padding of half the input dimension and stride equal to one.



- **Convolutional residual block**: The main path remains the same as the identity block, with the only difference being that the first kernel now has a stride of two, so that the output dimension is the integer division of the input size by two.

$$O = \left\lfloor \frac{I-1}{2} + 1 \right\rfloor$$

  The shortcut path also has a one-dimensional kernel with stride two, so the two outputs can be summed in the usual manner.



The output of the residual network is then further processed with average pooling and at last flattened to be fed as input to a feedforward classification network with an orientation boost. This means that the output of the initial feedforward layers is given as input to two separate feedforward layers, which produce the final output of the network through the softmax function:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

During inferce time, each input is rotated 12 times and we take as output label:

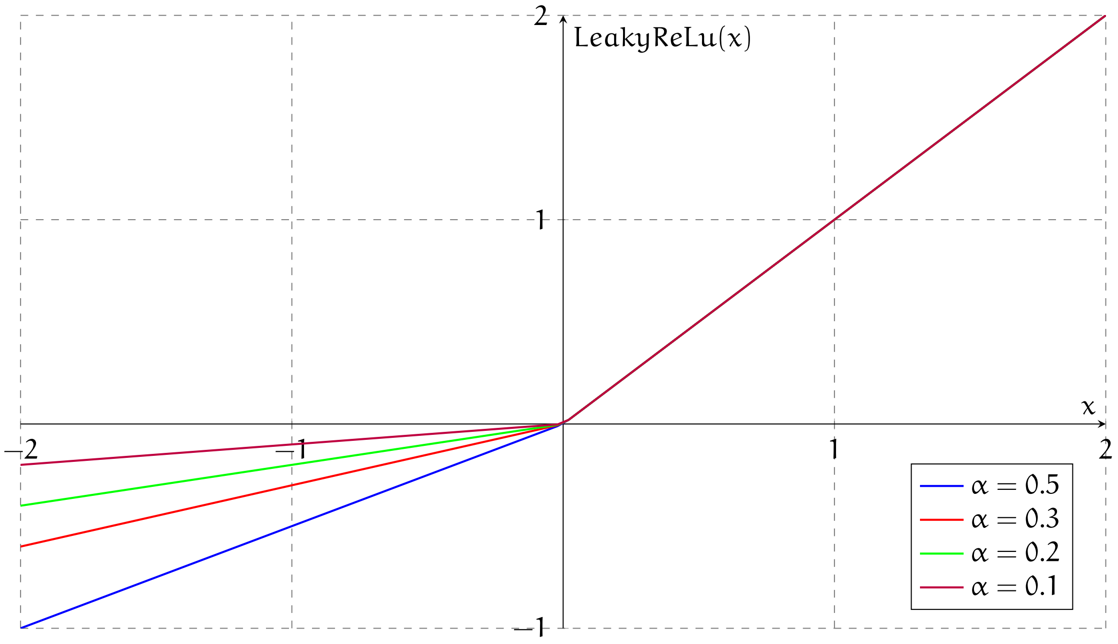
$$c_{\text{final}} = \underset{k \in classes}{\arg\max} \sum_{r \in rot} y_k(x_r)$$

This makes the output more robust to misleading views of the input.

Activation function

The activation function used for the training is the Leaky ReLu function with parameter 0.1 to avoid dying neurons due to bad initialization or bias.

$$\mathrm{LeakyReLu}(x) = \begin{cases} x & \text{if } x > 0, \\ \alpha x & \text{if } x \leqslant 0. \end{cases}$$



Regularization

Given the large number of parameters in the network, it is important to avoid overfitting to prevent the network from following the noise in the training data as if it was part of the useful features, which can lead to bad generalization to new data. A very common countermeasure to prevent overfitting is dataset augmentation, which enlarges the dataset and that I implemented during the voxelization process. On top of that, model validation was used to detect overfitting at each epoch by plotting the training-validation curves. The main regularization strategy used in the network was **model ensembling** through the implementation of dropout layers. During training, dropout layers randomly removes some neurons along with their incoming and outgoing connections at each epoch, as illustrated in Figure 38. This is done by dropping each neuron with a probability p, which can be tuned.

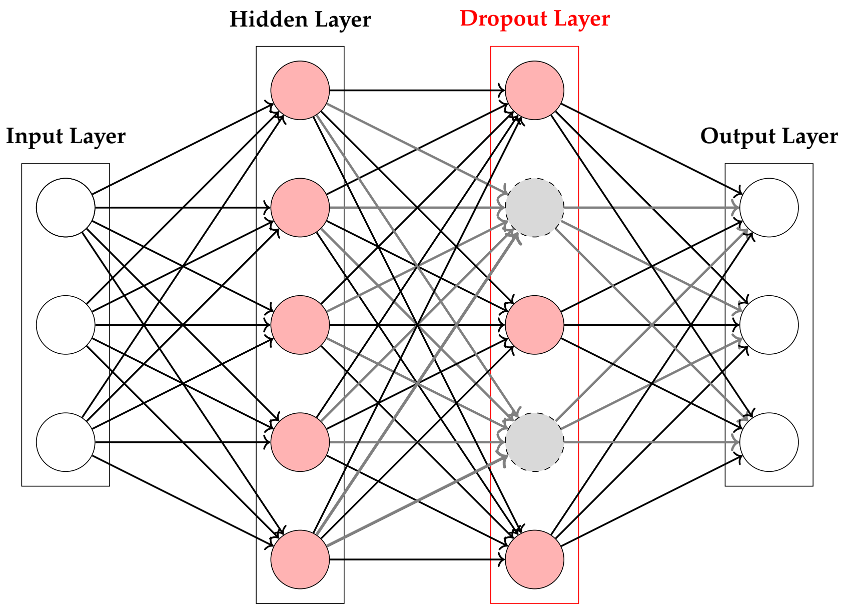


Figure 38: Dropout layer for regularization during training.

This strategy approximates the optimal Bayesian model ensembling

$$\mathbf{y} = \int F(\mathbf{x}, \mathcal{W})\, p(\mathcal{W}|\mathcal{D})\, d\mathcal{W}$$

by considering different networks with shared weights that are trained for only one iteration.

Another regularization term is given by the use of skip connections. In fact, the authors in Xu et al., 2024 demonstrated that skip connections can help to smooth the loss function, facilitating the converge to the minima proving once again their benefits in the training of a neural network.

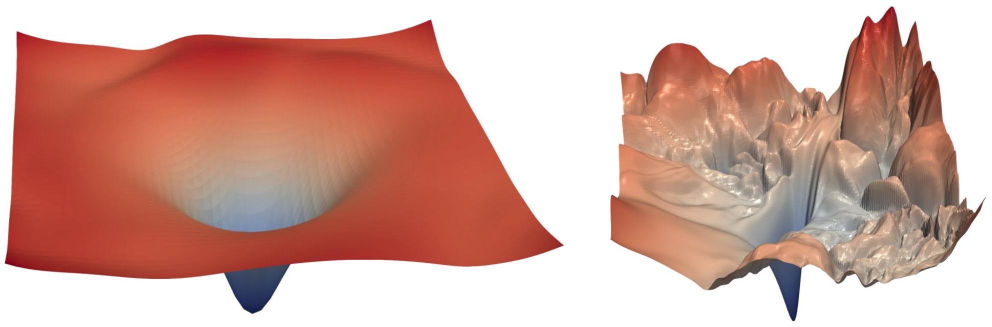


Figure 39: Loss surface of ResNet-56 with and without skip connections Xu et al., 2024.

4.5 OPTIMIZATION ALGORITHMS AND BACKPROPAGATION

After defining the architecture of the network, training is framed as an optimization problem where a loss function must be minimized numerically. To achieve this, it is first necessary to choose an optimization algorithm with good convergence characteristics, and secondly, to efficiently compute the gradient of the network at each step.

Stochastic Gradient Descent

One of the most used optimization algorithms in the past is Stochastic Gradient Descent, which has been proposed as a modification of the Gradient Descent algorithm to ease computation and introduce stochasticity to possibly escape local minima. For this type of algorithm, a random batch of training samples is used to compute an unbiased estimate of the gradient, and the weights are updated according to:

$$\mathbf{w}^{i+1} = \mathbf{w}^{i} - \frac{\eta^{(i)}}{K} \sum_{k=1}^{K} \nabla \mathcal{L}(\mathbf{x}_k, t_k; \mathbf{w}^i)$$

Over the years, SGD has shown many problems such as ill-conditioning, long training times for neural networks, and high dependence of the results on the learning rate, as shown in Soydaner, 2020. To address these issues, modifications of this algorithm were introduced to reduce the oscillations of the update.

Stochastic Gradient Descent with Momentum (Nesterov)

The concept of Momentum was introduced to make the updates more stable and to accelerate convergence to a minimum, especially along flat surfaces. In this case, a new quantity ν is introduced to accumulate the contributions of past gradients towards the mean direction, so that radical changes in direction due to particular batches can be dampened. A parameter $\alpha \in [0, 1)$ is introduced to tune the decay of past contributions. A Nesterov momentum refers to the computation of the gradient of the loss already previewing the future update based on the momentum, further increasing the convergence speed. The update rules are given by:

$$\mathbf{v}^{i+1} = \alpha \mathbf{v}^{i} - \frac{\eta^{(i)}}{K} \sum_{k=1}^{K} \nabla \mathcal{L}(\mathbf{x}_k, t_k; \mathbf{w}^i + \alpha \mathbf{v}^i)$$

$$\mathbf{w}^{i+1} = \mathbf{w}^{i} + \mathbf{v}^{i+1}$$

Adaptive Learning Rate: AdaGrad and RMSprop

Another type of optimization algorithm dynamically changes the learning rate across iterations and also applies different learning rates to each weight. The two main algorithms of this type are AdaGrad and RMSprop. These algorithms reduce the learning rate for weights frequently updated in past iterations while increasing the learning rate for weights that have had only small changes. In order to quantify the changes in the gradient, the mean of the gradient over the batch at each iteration i is first computed, and two quantities $\alpha_m^{(i)}, r_m^{(i)}$ are introduced to memorize the gradient changes for each weight:

$$\begin{cases} \alpha_m^{(i)} = \sum_{j=1}^{i} \left(g_m^{(j)}\right)^2 = \alpha_m^{(i-1)} + \left(g_m^{(i)}\right)^2 & \textbf{AdaGrad} \\ r_m^{(i)} = \beta\, r_m^{(i-1)} + (1-\beta)\left(g_m^{(i)}\right)^2 & \textbf{RMSprop} \end{cases}$$

The two quantities have a similar meaning, with the difference that RMSprop introduces a moving average with parameter β to control the decay of the contributions from earlier iterations. These quantities are then used to compute the dynamic learning rates for each parameter:

$$\begin{cases} \eta_m^{(i)} = \frac{\eta}{\sqrt{\alpha_m^{(i)} + \epsilon}} & \textbf{AdaGrad} \\ \eta_m^{(i)} = \frac{\eta}{\sqrt{r_m^{(i)} + \epsilon}} & \textbf{RMSprop} \end{cases}$$

These learning rates are then used for the update rule:

$$w_m^{(i+1)} = w_m^{(i)} - \eta_m^{(i)}\, g_m^{(i)}$$

Both algorithms are effective in many applications, but RMSprop typically addresses the problem in AdaGrad where the weights decrease monotonically, causing learning to stop after many iterations.

Adaptive Moment Estimation (Adam)

Adam is one of the most commonly used optimization algorithms in deep learning. The algorithm uses both momentum updates from estimates of the first-order moment of the gradient and learning rate updates from the second-order moment of the gradient. The two moments are computed as:

$$\begin{cases} \xi_m^{(i)} = \beta_1\, \xi_m^{(i-1)} + (1-\beta_1)\, g_m^{(i)} & \textbf{First order moment} \\ \psi_m^{(i)} = \beta_2\, \psi_m^{(i-1)} + (1-\beta_2)\left(g_m^{(i)}\right)^2 & \textbf{Second order moment} \end{cases}$$

Since the initialization of the moments is typically set to zero, these are biased estimates. While this bias diminishes after several iterations due to the decay in the moving average, it can be significant in the first few iterations. Adam compensates for this bias by using unbiased estimates:

$$\hat{\xi}_m^{(i)} = \frac{\xi_m^{(i)}}{(1-\beta_1^{(i)})} \qquad \hat{\psi}_m^{(i)} = \frac{\psi_m^{(i)}}{(1-\beta_2^{(i)})}$$

The weights are then updated according to the usual rule:

$$w_m^{(i+1)} = w_m^{(i)} - \eta \frac{\hat{\xi}_m^{(i)}}{\hat{\psi}_m^{(i)} + \epsilon} g_m^{(i)}$$

Adam has demonstrated many advantages and is known for its superior convergence properties in many deep learning applications. In particular, it requires little tuning of the initial learning rate and is computationally efficient.

Figure 40 shows an example of trajectories of some of the mentioned algorithms on a loss surface with a saddle point. In particular, we can see the problems of Gradient Descent, which gets stuck in the saddle point, and the efficiency of the Adam optimizer, which is able to avoid the saddle point directly, moving towards the minimum.

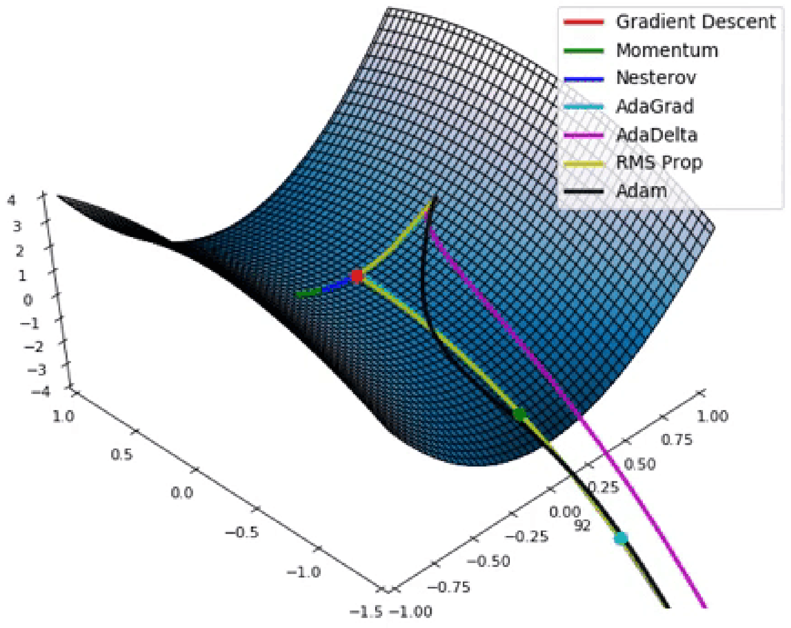


Figure 40: Comparison of the trajectory of some of the most used optimization algorithms.

Backpropagation

All the optimization algorithms include the calculation of the gradient over a batch of the loss function of the network with respect to all weights in the definition of the update rule. Since the number of weights in a network is typically very large, it is crucial to compute the gradient efficiently. The best way to do this is through numeric differentiation, which in the context of neural networks is called Backpropagation. The main idea here is to recursively relate the calculation of the gradient with respect to one weight to the calculation of the gradient of the weights in the subsequent layers. Specifically, an error message is propagated from the end of the network to the first layers. In a feedforward network, this leads to:

$$\delta_j^{(l)} \triangleq \frac{\partial \mathcal{L}}{\partial a_j^{(l)}} = \sum_k \frac{\partial \mathcal{L}}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}}$$

So that the error message can be recursively computed as:

$$\delta_j^{(l)} = f'(a_j^{(l)}) \sum_k \delta_k^{(l+1)} w_{k,j}^{(l+1)}$$

And the gradient is efficiently updated by:

$$\frac{\partial \mathcal{L}}{\partial w_{j,i}^{(l)}} = \delta_j^{(l)} o_i^{(l-1)} \qquad \frac{\partial \mathcal{L}}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

4.6 MODEL TRAINING

The network is then trained using a loss function that incorporates the orientation-boost that forces the network to learn the object's orientation as a secondary task. This can be done using a combination of categorical cross-entropy loss for each task, weighted by the parameter γ:

$$\mathcal{L}(\mathcal{D}, \mathcal{W}) = -\sum_{n=1}^{N} \left[\sum_{q=1}^{10} t_{n,q}^{\text{class}} \log y_{n,q}^{\text{class}} + \gamma \sum_{p=1}^{120} t_{n,p}^{\text{rot}} \log y_{n,p}^{\text{rot}} \right]$$

In this specific case, I chose $\gamma = 0.5$ and observed that the accuracy of the class label was not greatly influenced by this specific value.

After this, several optimization algorithms were tested using different learning rates. The algorithm that produced the best and most stable results was the Adam optimizer with an initial learning rate of 10^{-3}.

4.7 RESULTS

The final architecture of the model was built starting from an implementation of the ORION network, which was used as a baseline. Extending the ResNet concept to 3D data was not straightforward, as I had to account for relatively small voxel grids and limited computational power. Particularly important was the choice of the activation function that I changed from the ReLU function to the Leaky ReLU function. This was helpful to avoid the dying neurons problem that was affecting the convergence of the model. I also found that the model achieved similar performance within a discrete range of hyperparameters, such as learning rate, batch size, and the parameter γ. I experimented with different types of optimizers, confirming that the Adam optimizer provided the most stable updates and converged the fastest overall.

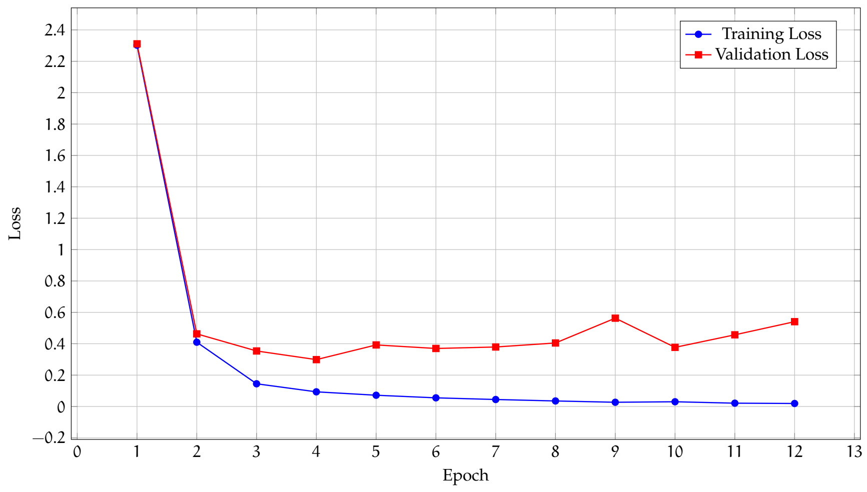


Figure 41: Train-validation class loss plot.

Even with the large number of layers used, the model was able to converge very quickly within 10 epochs to achieve the best accuracy on the validation set. Training even more the model was slower and did not improve the validation accuracy, as I checked by testing the validation loss and accuracy at the end of each epoch to update the best model. This showed that the model successfully avoided overfitting despite the large number of parameters, thanks to the extended augmented dataset and the use of dropout layers, whose impact grows the deeper the layer is placed within the network.

Figure 42 shows the confusion matrix of the residual model on the test set provided by the ModelNet10 dataset. The matrix is almost diagonal and this indicates that most of the predicted objects are correctly classified.

As expected, the network had the greatest difficulty to distinguish between similar object shapes, such as *desks* and *tables* and *dressers* and *nightstands*.

During testing, the voting approach was shown to significantly improve the accuracy of the network across all designs by almost 2%. The accuracies of all the trained models were compared, beginning with the baseline ORION model, passing through the extended ORION model from the paper, and concluding with my implementation incorporating skip layers. The trained model achieved an accuracy of **92.3%** on the ModelNet10 test dataset, surpassing both my original ORION network and the modified implementation, therefore demonstrating the potential of skip layers for 3D data.

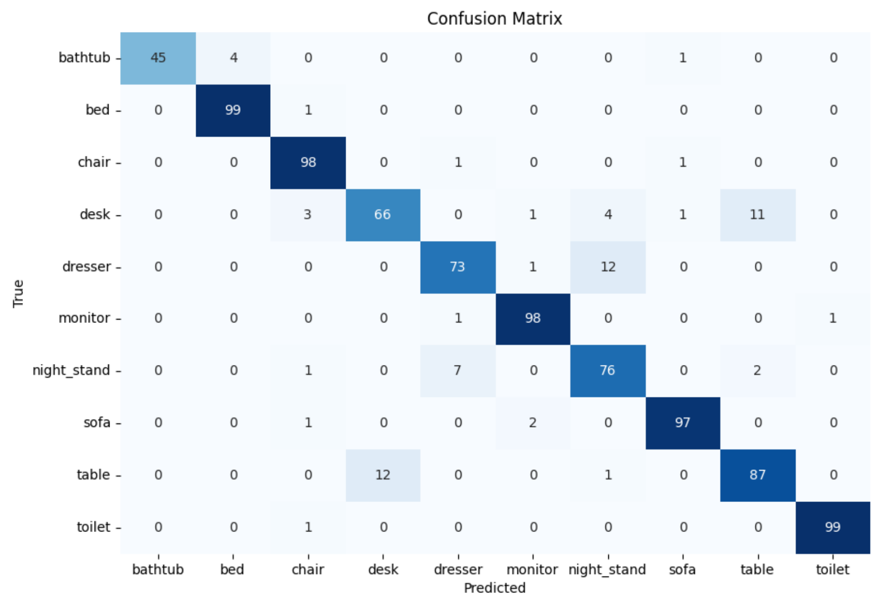


Figure 42: Confusion matrix on the ModelNet10 test dataset

4.8 CONCLUDING REMARKS AND FUTURE WORK

The solution proposed in this thesis shows good accuracy and presents several advantages. Even more importantly of the achieved accuracy, various new approaches were implemented starting from the baseline model that showed constant accuracy improvements and this provides a tested basis for future architectures. In particular, the use of skip layers can be explored more, where interesting research can be done for long skip connections and alternative implementations of the convolutional and identity blocks. For example, a technique the was proven useful in the 2D domain and that could be a possible extension was placing the activation function before the sum block. Additionally, more advanced voxelization and augmentation techniques could be tested, such as also adding translations and rotations around the x-axis. Exploring different numbers of rotations for each class could also prove beneficial, as certain classes may gain from a larger or smaller range of possible rotations.

Finally, the initial resolution of the voxel grid and the number of layers can be tuned based on the memory and computational capacity of the hardware used for training.

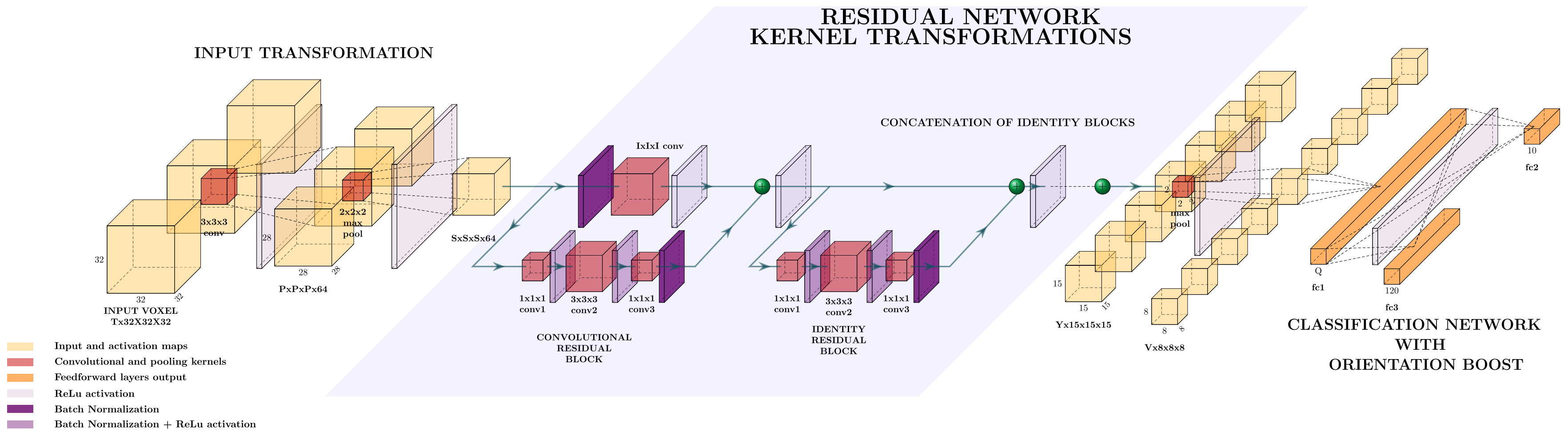


Figure 34: My deep neural network architecture for 3D classification.

A APPENDIX

In the following I present the key parts of the code that I developed to implement the proposed deep network architecture. Only key parts are shown that represent the main ideas. The whole code was written in Python using Pytorch libraries.

Voxelization and Augmentation

```python
'''
1. open the .off files and use the open3d function to read the mesh
2. create #augmentation_factor rotated meshes, making sure that the
rotations are random but still unique for the same mesh
3. voxelize all the rotated meshes choosing the voxel_size
4. save all the voxel grids into a .ply file concatenating in the name
the degrees of the rotation. Also save this file in the corresponding
folder of the rotation (if not existant, just create it)
'''
def augment_and_voxelize_o3d(input_address, output_address, augmentation_factor,
num_rotation_classes, voxel_grid_size = 32, padding = 0) :
    #open off file of the mesh and create the mesh
    mesh = o3d.io.read_triangle_mesh(input_address)
    #pick #augmentation_factor random but unique numbers for the rotations
    rand_rotation_class_list = random.sample(range(num_rotation_classes), augmentation_factor)
    #the list of the random rotations sampled
    theta_list = [((2*i)/num_rotation_classes) * math.pi for i in rand_rotation_class_list]
    #---------creation of the folder for the rotation class------------
    #split the output address into folder address and file name
    head_tail_address = os.path.split(output_address)
    #create the #num_rotation_classes folders if non existant
    for i in range(num_rotation_classes):
        angle = i * int(360/num_rotation_classes)
        output_address_directory = head_tail_address[0] + '/' + str(angle)
        if not os.path.exists(output_address_directory):
            os.makedirs(output_address_directory)
    #--------end creation of the folder for the rotation class---------
    for theta in theta_list:
        #-----apply the random rotation---------
        rot_matrix = np.array([[ math.cos(theta), -math.sin(theta),    0],
                               [ math.sin(theta),  math.cos(theta),    0],
                              [0,                              0,      1]])
        mesh.rotate(rot_matrix)
        #------normalize the mesh---------------
        mesh.scale(1/np.max(mesh.get_max_bound()-mesh.get_min_bound()),center=mesh.get_center())
        #-------------------------voxelize-------------------------
        #the voxelization procedure is based on the open3d tutorial
        VOXEL_SIZE = 1/(voxel_grid_size - (1 + padding*2))
        #voxelize the normalized mesh using open3d function
        voxel_grid = o3d.geometry.VoxelGrid.create_from_triangle_mesh(mesh,
        ↪  voxel_size=VOXEL_SIZE)
        #----------------------save results-------------------------
        #rotation degrees
        degrees_rotation = int((theta / (2*math.pi)) * 360)
        #final output address of the file
        output_address_rotated = head_tail_address[0] + '/' + str(degrees_rotation) + '/' +
        ↪  head_tail_address[1] + "_" + str(degrees_rotation) + ".ply"
        #save the voxel grid in a .ply file
        o3d.io.write_voxel_grid(output_address_rotated, voxel_grid, write_ascii=False, compressed
        ↪  = False)
        #------------------------------------------------------------
''' read an open3d grid object and return a binary voxel grid as a tensor
1. retrive all the voxels in the voxel grid using voxel_grid.get_voxels()
2. map(function, iterable) applies the function to each element in the
iterable and returns a map object
3. choose as function the lambda function lambda x: x.grid_index to
extract the 'grid_index' attribute from each voxel object
4. list(map(....)) transform the returned map object into a list -->
output is a list of 'grid_index' values for all voxels
5. np.array(list(...)) transform the returned list into a np.array for
efficient operations '''
def read_voxel_grid(grid, voxel_grid_size = 32):
    np_voxels = np.array(list(map(lambda x: x.grid_index, grid.get_voxels())))
    np_voxelGrid = np.zeros((voxel_grid_size, voxel_grid_size, voxel_grid_size))
    for j in range(len(np_voxels)):
        x = np_voxels[j][0]
        y = np_voxels[j][1]
        z = np_voxels[j][2]
        np_voxelGrid[x,y,z] = 1
    return torch.from_numpy(np.array([[np_voxelGrid]])).float()
```

Definition of the residual blocks

```
#-----------------------------DEFINITION OF THE RESIDUAL BLOCKS--------------------------------
'''this is the general processing path of both the identity and the convolutional layers'''
class MainPath(nn.Module):
    def __init__(self, in_channels, filters, kernel_size, stride=1):
        super().__init__()
        #dimension of the filters
        F1, F2, F3 = filters
        self.main_path = nn.Sequential(
            #1x1x1 filter
            nn.Conv3d(in_channels, F1, kernel_size=1, stride=stride),
            nn.BatchNorm3d(F1),
            nn.LeakyReLU(0.1),
            #middle convolution with tunable dimension
            nn.Conv3d(F1, F2, kernel_size=kernel_size, padding=kernel_size//2),
            nn.BatchNorm3d(F2),
            nn.LeakyReLU(0.1),
            #1x1x1 filter
            nn.Conv3d(F2, F3, kernel_size=1),
            nn.BatchNorm3d(F3),
        )
        self.apply(self._init_weights)
    '''weights initialization function'''
    def _init_weights(self, module):
        if isinstance(module, torch.nn.Linear):
            #xavier initialization
            torch.nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                module.bias.data.zero_()
        if isinstance(module, torch.nn.Conv3d):
            torch.nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                module.bias.data.zero_()
    def forward(self, x):
        y = self.main_path(x)
        return y
'''identity block that keeps the same dimension of the input'''
class IdentityBlock(MainPath):
    def __init__(self, in_channels, filters, kernel_size):
        #the input size is preserved using stride = 1 (as default in the main path)
        super().__init__(in_channels, filters, kernel_size)
        self.activation = nn.LeakyReLU(0.1)
    def forward(self, x):
        y =self.activation(self.main_path(x) + x)
        return y
'''this set-up gives as ouput dimension floor(in_dimension - 1/2) + 1'''
class ConvolutionalBlock(MainPath):
    def __init__(self, in_channels, filters, kernel_size):
        super().__init__(in_channels, filters, kernel_size, stride=2)
        self.relu = nn.ReLU()
        #shortcut path that uses a convolution to half the size of the input
        self.shortcut_path = nn.Sequential(
            nn.Conv3d(in_channels, filters[2], kernel_size=1, stride=2),
            nn.BatchNorm3d(filters[2])
        )
        self.apply(self._init_weights)
    '''weights initialization function'''
    def _init_weights(self, module):
        if isinstance(module, torch.nn.Linear):
            torch.nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                module.bias.data.zero_()
        if isinstance(module, torch.nn.Conv3d):
            torch.nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                module.bias.data.zero_()
    def forward(self, x):
        y = nn.LeakyReLU(0.1)(self.main_path(x) + self.shortcut_path(x))
        return y
#-----------------------------------------------------------------------------------------------
```

Definition of the architecture of the deep network

```
class ResidualNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.network = nn.Sequential(
            #input (1,32,32,32)
            nn.Conv3d(1, 64, kernel_size=3, stride=1),
            #output(64, 30,30,30)
            nn.BatchNorm3d(64),
            #output(64, 30,30,30)
            nn.MaxPool3d(kernel_size=2, stride=2),
            #output(64, 15,15,15)
            ConvolutionalBlock(64, [64, 64, 128], kernel_size=3),
            #output (128, 8,8,8)
            nn.Dropout(0.2),
            IdentityBlock(128, [64, 64, 128], kernel_size=3),
            #output (128, 8,8,8)
            IdentityBlock(128, [64, 64, 128], kernel_size=3),
            #output (128, 8,8,8)
            ConvolutionalBlock(128, [64, 64, 128], kernel_size=3),
            nn.Dropout(0.2),
            #output (128, 4,4,4)
            IdentityBlock(128, [64, 64, 128], kernel_size=3),
            #output (128, 4,4,4)
            IdentityBlock(128, [64, 64, 128], kernel_size=3),
            #output (128, 4,4,4)
            IdentityBlock(128, [64, 64, 128], kernel_size=3),
            #output (128, 4,4,4)
            nn.AvgPool3d(kernel_size=2, stride=2)
            #output (128, 2,2,2)
        )
        self.classification_layer = nn.Linear(128*(2**3), 128)
        self.fc2 = nn.Linear(128, 10)
        self.fc3 = nn.Linear(128, 120)
        self.apply(self._init_weights)
    def forward(self, x):
        #y = self.network(x).reshape((x.shape[0], -1))
        y = nn.Flatten()(self.network(x))
        y = nn.LeakyReLU(0.1)(self.classification_layer(y)) #LeakyReLU(0.1)
        return (self.fc2(y), self.fc3(y))
    def _init_weights(self, module):
        if isinstance(module, torch.nn.Linear):
            torch.nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                module.bias.data.zero_()
        if isinstance(module, torch.nn.Conv3d):
            torch.nn.init.xavier_uniform_(module.weight)
            if module.bias is not None:
                module.bias.data.zero_()
```

Training loop

```
for epoch in range(EPOCHS):
    running_loss = 0
    total_loss = 0
    total_class_loss = 0
    #set model to training mode
    model.train(True)
    #iterate the validation dataloader
    for i, data in enumerate(train_dataloader):
        #get data sample and split it into input binary grid and label
        inputs, labels = data
        #split the label into class label and rotation label
        labels_class, labels_rot = labels
        labels_class = labels_class.int()
        labels_rot = labels_rot.int()
        '''the labels class and label rotation and two integer numbers, one
        from 0 to (#numclasses - 1) and one from 0 to (#numrotations - 1).
        The output of the network is a vector of #numclasses
        +#numrotations float numbers. So in order to create the correct
        target output I have to create a tensor for the classes  of
        dimension BATCH_SIZE x #numclasses (one vector for each sample in
        the batch) and one tensor for the rotations and set the correct
        index corresponding to the label to 1'''
        labels_class_tensor = torch.zeros(len(labels_class), NUM_CLASSES).float()
        labels_rotations_tensor = torch.zeros(len(labels_class), NUM_ROTATIONS).float()
        for j in range(len(labels_class)):
            labels_class_tensor[j, labels_class[j]] = 1
            labels_rotations_tensor[j, labels_rot[j]] = 1
        #--------------------optimization step----------------------
        optimizer.zero_grad()
        #get the full output of the model
        outputs = model(inputs)
        #split into output class and output rotation
        outputs_class, outputs_rot = outputs
        #calculate loss and gradient voxelnetLoss already applies the
        softmax the the outputs of the network
        loss = voxelnetLoss(outputs_class, outputs_rot, labels_class_tensor,
        ↪  labels_rotations_tensor, weight_class, GAMMA)
        loss.backward()
        '''Monitor and log gradients
        for name, param in model_Orion.named_parameters():
            if param.grad is not None:
                writer.add_scalar(f'gradients/{name}',
                param.grad.norm().item(), epoch * len(train_dataloader) + i) '''
        '''avoid gradient clipping (opposite of vanishing gradient)
        max_grad_norm = 4
        torch.nn.utils.clip_grad_norm_(model_Orion.parameters(), max_grad_norm)'''
        #update model weights
        optimizer.step()
        #-----------------------------------------------------------------
        running_loss += loss.item()
        total_loss += loss.item()
        # print statistics every 32 mini-batches
        if i % 32 == 32 - 1:
            print('[Epoch: %d, Batch: %4d / %4d], loss: %.3f' %
            (epoch + 1, i + 1, len(train_dataloader), running_loss / len(labels_class)))
            running_loss = 0.0
    #---------------------EVALUATE LOSSES AND PLOT---------------------
    training_loss.append(total_class_loss/len(train_dataloader))
    validation_loss.append(helper.calculate_loss(test_dataloader, model_Orion, num_classes =
    ↪  NUM_CLASSES, num_rot = NUM_ROTATIONS, gamma = 0.5)/len(test_dataloader))
    helper.plot_losses(training_loss, validation_loss)
    #-----------------------------------------------------------------
    current_val = evaluate_validation_accuracy(model, test_dataloader)
    validation_accuracy.append(current_val)
    print('Valid accuracy: ' + str(current_val))
    #-----------------------------------------------------------------
    #save the model that achieves the best validation loss
    if current_val > best_val:
        best_val = current_val
```

Part III

THIRD PART: INDUSTRIAL APPLICATION

This part integrates all the concepts from the first two parts for the design of an industrial application. The first chapter provides an overview of the used programming language MVTec Halcon, highlighting the high-level algorithms used in industrial applications. The second chapter presents the development of an industrial application designed during my internship at Innova Srl.

5 BUILDING A 3D COMPUTER VISION APPLICATION USING HALCON

MVTec Halcon is a proprietary software that provides interfaces, algorithms, and structures that can be used to solve end-to-end computer vision tasks. The fact that the many of the algorithms provided are state-of-the-art algorithms designed for vision applications makes Halcon a powerful tool to do rapid prototyping and efficient application development in significantly less time compared to using open-source languages like C++ with OpenCV libraries. This makes it a valuable option for many automated industrial systems that rely on vision technology.

Typically, most Halcon applications are prototyped in the interactive programming environment HDevelop, which has its own programming language. The code developed in HDevelop can be exported and automatically translated into other programming languages, or more efficiently rewritten in the final application language using the Halcon libraries.

Figure 43 shows the Halcon architecture.

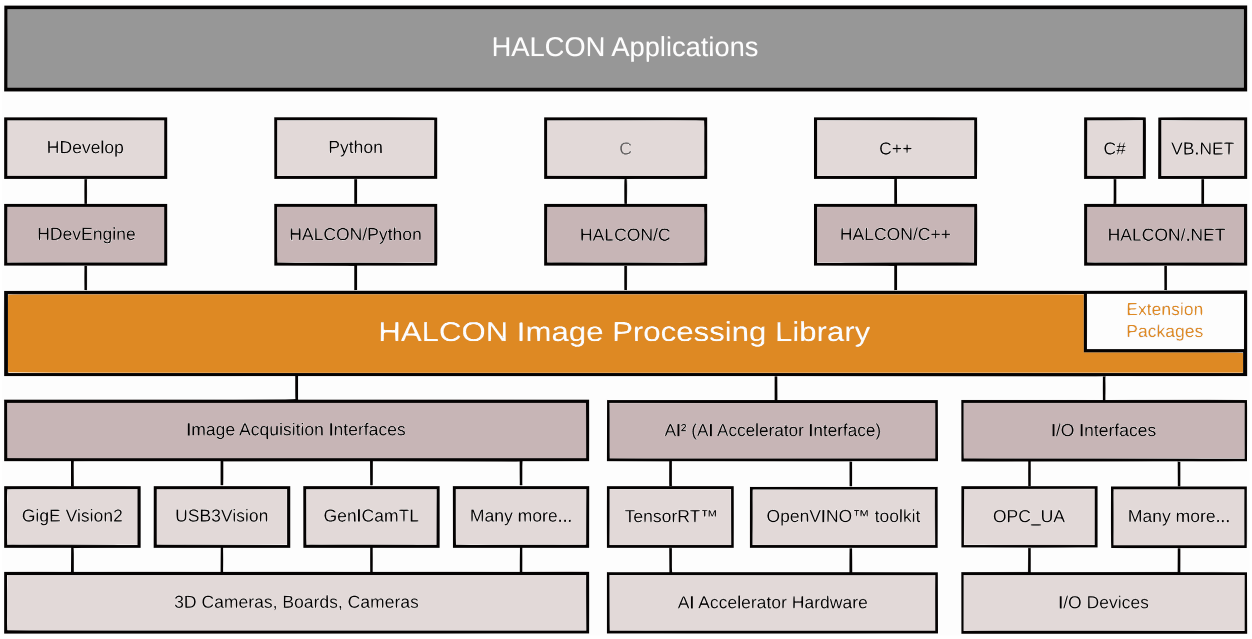


Figure 43: Halcon Architecture MVTec, n.d.(b).

5.1 3D OBJECT MODELS

A 3D object model is a data structure used to represent a three-dimensional data. It can be obtained in various ways and contain different information depending on the type of model. This is important because, depending on the application, different types of object models might be needed, as not every operation can be applied to every object.

In particular, 3D object models can be:

- created by specifying the parameters of particular shapes or surface parametrizations, or by directly specifying the points that lie on the surface;
- obtained from Computer-Aided Design (CAD) data;
- derived directly from sensor data or through a 3D reconstruction approach.

Figure 44 illustrates the various methods for obtaining a 3D model depending on the generation or acquisition technique used.

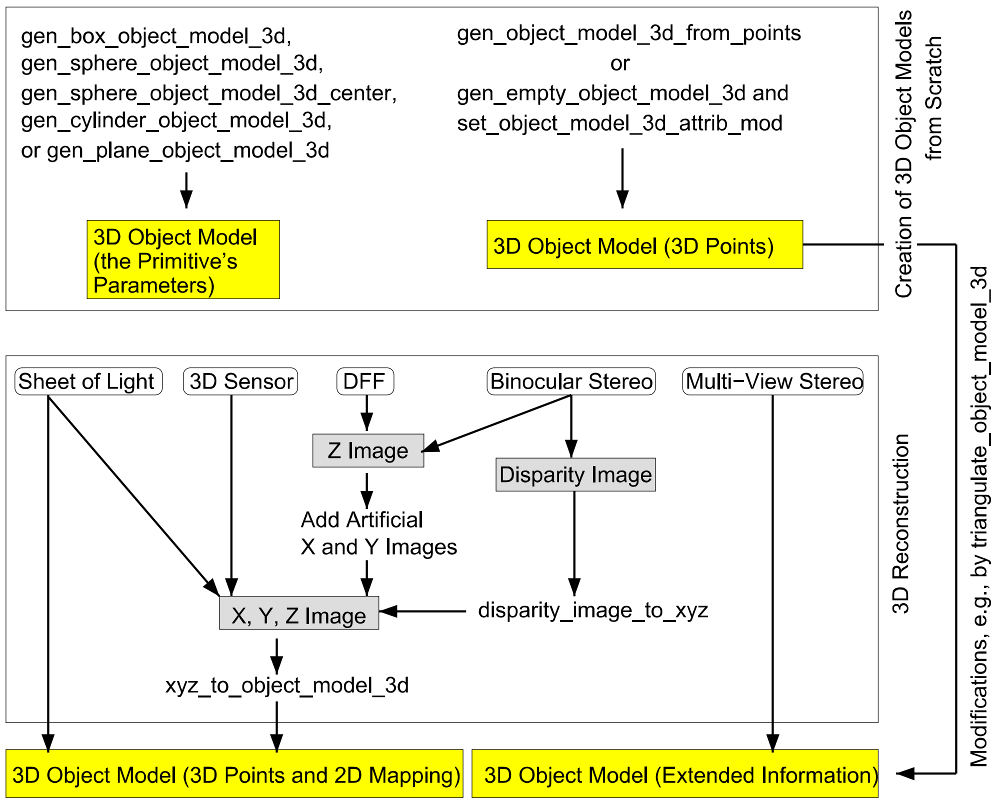

gen_box_object_model_3d,
gen_sphere_object_model_3d,
gen_sphere_object_model_3d_center,
gen_cylinder_object_model_3d,
or gen_plane_object_model_3d
3D Object Model (the Primitive's Parameters)
gen_object_model_3d_from_points
or
gen_empty_object_model_3d and
set_object_model_3d_attrib_mod
3D Object Model (3D Points)
Creation of 3D Object Models from Scratch
Sheet of Light
3D Sensor
DFF
Binocular Stereo
Multi−View Stereo
Z Image
Disparity Image
Add Artificial X and Y Images
X, Y, Z Image
disparity_image_to_xyz
xyz_to_object_model_3d
3D Reconstruction
Modifications, e.g., by triangulate_object_model_3d
3D Object Model (3D Points and 2D Mapping)
3D Object Model (Extended Information)

Figure 44: Overview of the 3D object models in Halcon.

Object models typically need to be prepared before processing, for example to ensure efficient data access and to add attributes or convert them into the required representation for the processing algorithm. Some of these operations include generating point clouds from sensor parameters and depth maps, triangulating point clouds to create meshes, and subsequently calculating the surface normals used in the matching algorithms.

5.2 CAMERA CALIBRATION AND MEASUREMENTS ON A PLANE

Many computer vision applications require camera calibration, both to accurately reconstruct the positions of the object points and to make reliable measurements directly from 2D/3D data.

In order to achieve the maximum accuracy, it is very important to precisely determine the parameters of the camera model for the specific camera that is being used, as even small changes in the parameters for the same type of camera can have big impact on the quality of the results.

To perform camera calibration in Halcon the following steps are needed:

1. **Create a calibration plate descriptor** by specifying the pattern and the coordinates of the marks that have to be detected.
2. **Create a calibration model** by specifying the number and type of cameras used, and by defining the parameters in the optimization (for example choosing the distortion model) while providing initial values for these parameters to initialize the optimization algorithm.
3. **Collect the calibration data** by moving the calibration plate to different positions and orientations.
4. **Locate the calibration plate in the images** by extracting the salient points.
5. **Perform the actual calibration** by running the optimization algorithm.

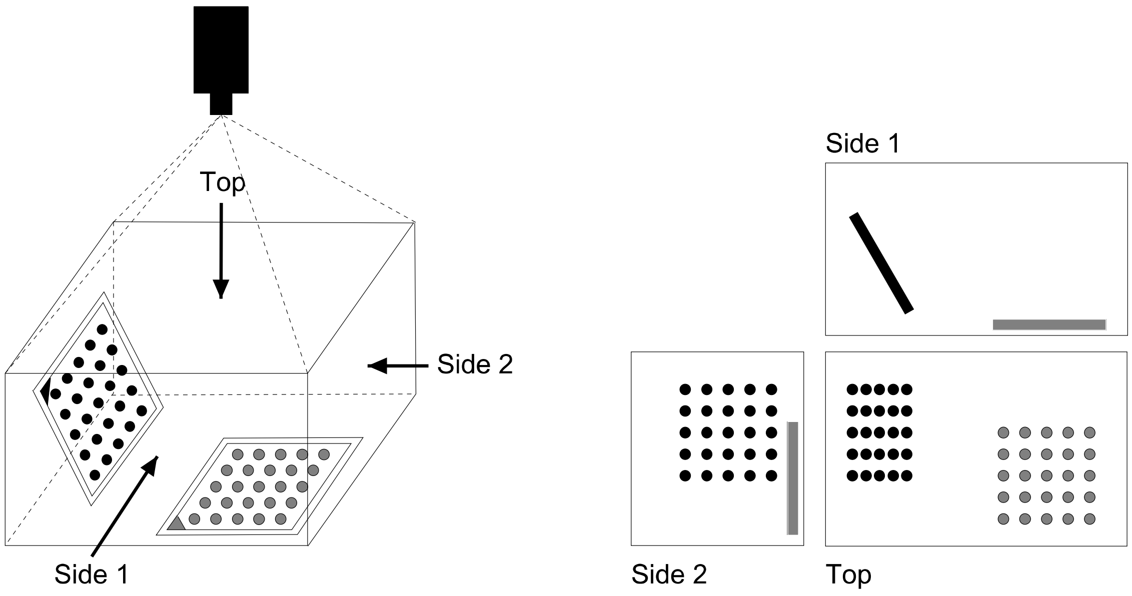


Figure 45: Example of how to take the calibration images MVTec, n.d.(c).

The results of the calibration include the intrinsic camera parameters as well as the pose of the calibration plate in each of the calibration images. To determine the coordinates with respect to the world reference frame, a further rigid transformation is needed as it is represented in Figure 46.

$$\begin{pmatrix} x^w \\ y^w \\ z^w \\ 1 \end{pmatrix} = \mathbf{H}^w_{cp}\,\mathbf{H}^{cp}_c\,\mathbf{H}^c_i \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \mathbf{H}^w_{cp} \begin{pmatrix} x^{cp} \\ y^{cp} \\ z^{cp} \\ 1 \end{pmatrix}$$

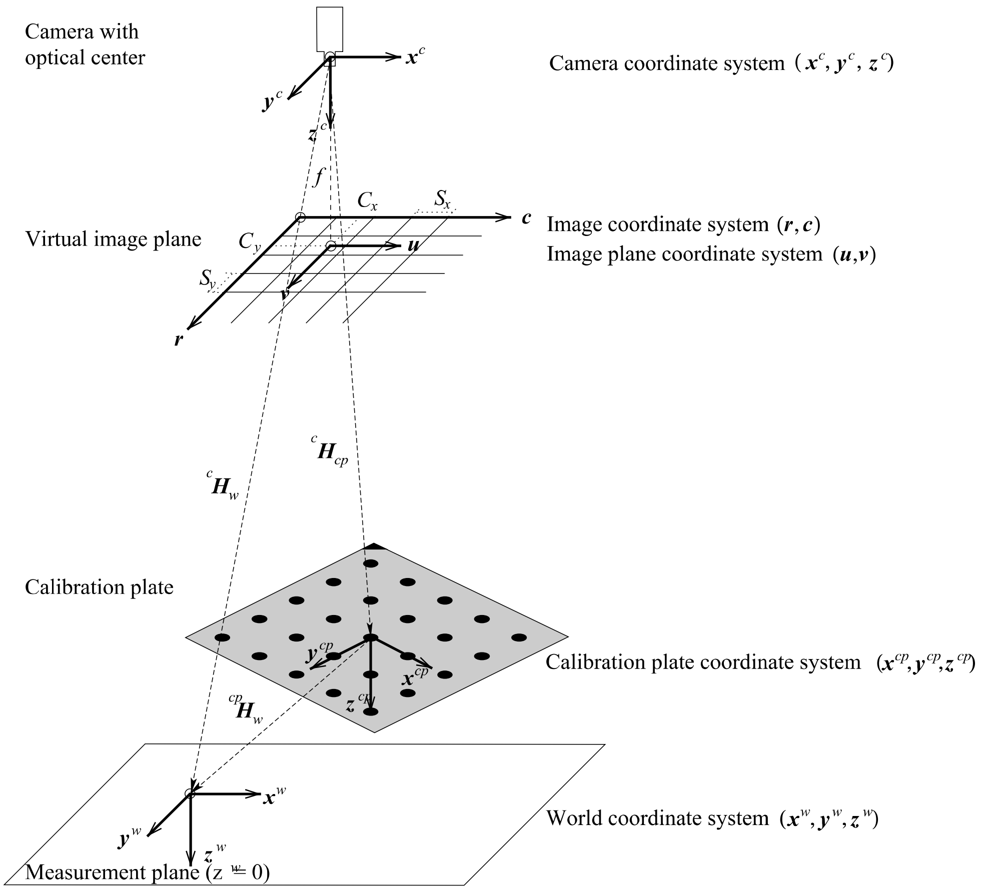

Camera with optical center
Camera coordinate system (x^c, y^c, z^c)
x^c
y^c
z^c
f
C_x
S_x
c
Virtual image plane
C_y
S_y
u
v
r
Image coordinate system (r, c)
Image plane coordinate system (u,v)
${}^cH_{cp}$
${}^cH_w$
Calibration plate
y^{cp}
x^{cp}
z^{cp}
Calibration plate coordinate system (x^{cp},y^{cp},z^{cp})
${}^{cp}H_w$
x^w
y^w
z^w
World coordinate system (x^w,y^w,z^w)
Measurement plane $(z^w = 0)$

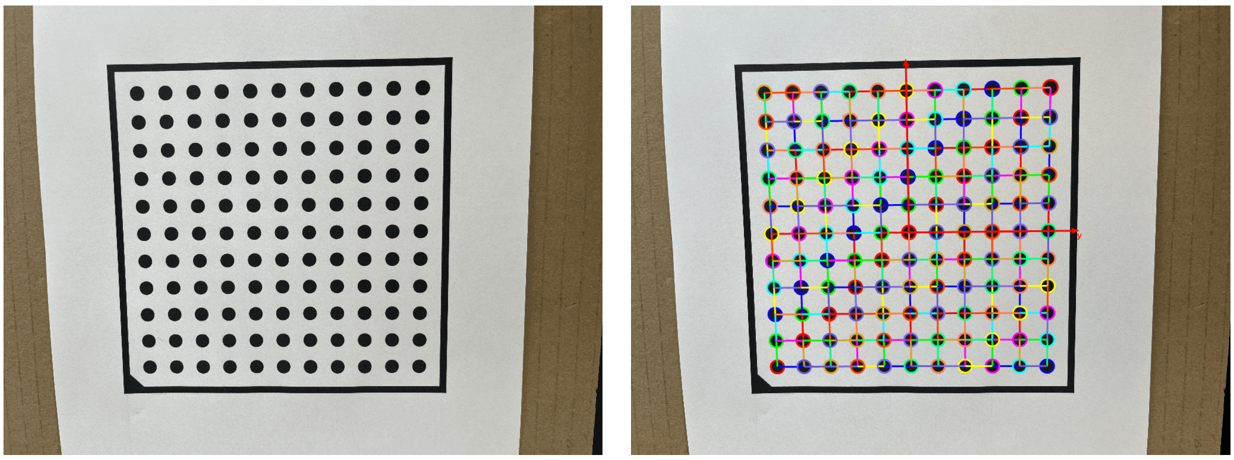
Figure 46: Relation between calibration and measurement plane.

To test the results of the camera calibration and perform some measurements on a specified plane, I implemented the calibration code in Halcon using my personal camera. For testing, I printed a calibration pattern and attached it to a flat surface. By capturing several images at different plane orientations, I then performed the calibration with the code below.

```
CalTabDescrFile := 'C:/Users/39348/Desktop/Tesi Halcon/caltab_pcLab.descr'
*initialization
gen_cam_par_area_scan_division (0.0269743,49.9899,8.28539e-06,8.3e-06,2003.16,1532.15,
4032,3024, StartCamPar)
*create calibration object
```

```
create_calib_data ('calibration_object', 1, 1, CalibDataID)
set_calib_data_cam_param (CalibDataID, 0, [], CamParam)
set_calib_data_calib_object (CalibDataID, 0, CalTabDescrFile)
list_image_files ('C:/Users/39348/Desktop/Tesi Halcon/images_calibration_phone',
'default', [], ImageFiles)
for I := 0 to |ImageFiles|-1 by 1
    read_image (Image, ImageFiles[I])
    * Find the calibration plate
    find_calib_object (Image, CalibDataID, 0, 0, I, [], [])
    get_calib_data (CalibDataID, 'camera', 0, 'init_params', StartCamPar)
    get_calib_data_observ_points (CalibDataID, 0, 0, I, Row, Column, Index, Pose)
    get_calib_data_observ_contours (Contours, CalibDataID, 'caltab', 0, 0, I)
    gen_cross_contour_xld (Cross, Row, Column, 6, 0.785398)
endfor
calibrate_cameras (CalibDataID, Error)
get_calib_data (CalibDataID, 'camera', 0, 'params', CamParam)
*measurements part
for I := 0 to |ImageFiles|-1 by 1
    read_image (Image, ImageFiles[I])
    get_image_size (Image, Width1, Height)
    * Now, measure the size of the black border of the plate
    get_measure_positions (Image, PlateRegion, CalibDataID, I, Distance, Phi,
    RowCenter, ColumnCenter)
    gen_rectangle2_contour_xld (Rectangle, RowCenter, ColumnCenter,
    Phi, Distance x 0.52, 8)
    gen_measure_rectangle2 (RowCenter, ColumnCenter, Phi, Distance x 0.52, 8, Width1,
    Height, 'nearest_neighbor', MeasureHandle)
    measure_pos (Image, MeasureHandle, 1, 40, 'all', 'all', RowEdge, ColumnEdge,
    Amplitude, Distance1)
    Rows := [RowEdge[0],RowEdge[|RowEdge| - 1]]
    Columns := [ColumnEdge[0],ColumnEdge[|RowEdge| - 1]]
    gen_cross_contour_xld (Cross, Rows, Columns, 16, Phi)
    * Transform the two border points into the world coordinate system
    get_calib_data (CalibDataID, 'calib_obj_pose', [0,I], 'pose', Pose)
    image_points_to_world_plane (CamParam, Pose, Rows, Columns, 'm', SX, SY)
    distance_pp (SY[0], SX[0], SY[1], SX[1], Width)
    * Now, measure the size of the calibration marks
    * Extract the ellipses in the image
    erosion_circle (PlateRegion, ROI, 17.5)
    reduce_domain (Image, ROI, ImageReduced)
    edges_sub_pix (ImageReduced, Edges, 'canny', 1, 20, 60)
    select_contours_xld (Edges, SelectedEdges, 'contour_length', 20, 99999999, -0.5, 0.5)
    * Fit ellipses to extracted edges
    fit_ellipse_contour_xld (SelectedEdges, 'fitzgibbon', -1, 2, 0, 200, 3, 2, Row,
    Column, Phi, Radius1, Radius2, StartPhi, EndPhi, PointOrder)
    MeanRadius1 := mean(Radius1)
    MeanRadius2 := mean(Radius2)
    DevRadius1 := deviation(Radius1)
    DevRadius2 := deviation(Radius2)
    * Transform the ellipses to world coordinates, where they should be circles
    * and convert the circles from meters to millimeters so that we can see them.
    contour_to_world_plane_xld (SelectedEdges, WorldCircles, CamParam, Pose, 'mm')
    * Fit ellipses to the circles in world coordinates
    fit_ellipse_contour_xld (WorldCircles, 'fitzgibbon', -1, 2, 0, 200, 3, 2, Row,
    Column, Phi, RadiusW1, RadiusW2, StartPhi, EndPhi, PointOrder)
    MeanRadiusW1 := mean(RadiusW1)
    MeanRadiusW2 := mean(RadiusW2)
    DevRadiusW1 := deviation(RadiusW1)
    DevRadiusW2 := deviation(RadiusW2)
endfor
**do measuraments of the circle in the last image
read_image (Image, ImageFiles[|ImageFiles|-1])
get_image_size (Image, Width1, Height)
decompose3(Image, ImageR, ImageG, ImageB)
edges_sub_pix (Image, Edges, 'canny', 1, 20, 60)
threshold (ImageR, Regions, 140, 141)
connection (Regions, ConnectedRegions)
```

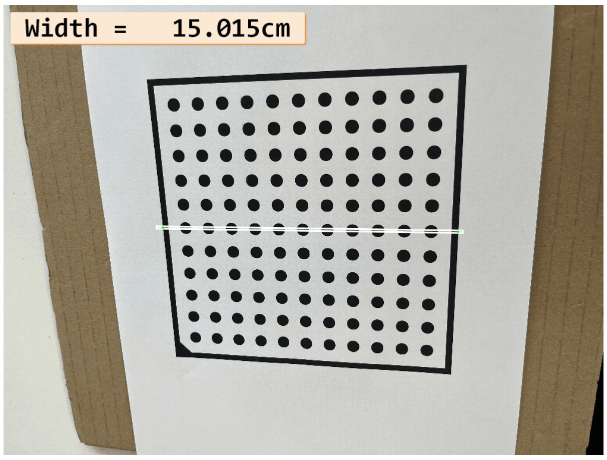
Figure 47 shows the detection of the pattern and the pose of the calibration object.



(a) *Printed calibration plate attached to a plate.* (b) *Results of the calibrated camera and pose of the plate.*

Figure 47: Example of calibration in Halcon of my personal phone camera.

After the camera calibration was performed, the parameters were tested by measuring the width of the calibration plate and the radius of the circles in the pattern directly from the images. The results were very accurate, as shown in Figure 48.



(a) *Width measurement.*

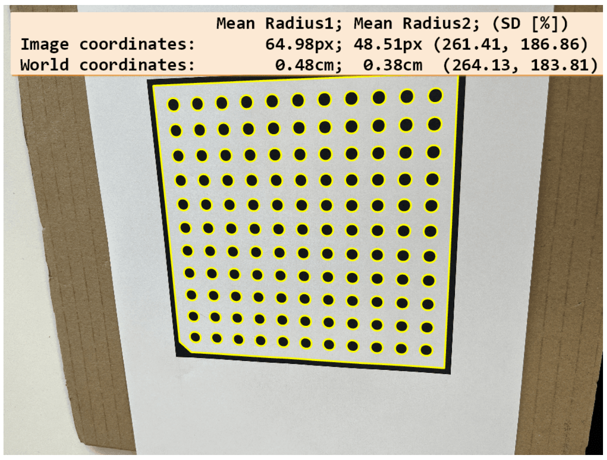

Mean Radius1; Mean Radius2; (SD [%])
Image coordinates: 64.98px; 48.51px (261.41, 186.86)
World coordinates: 0.48cm; 0.38cm (264.13, 183.81)
(b) *Radius measurement.*

Figure 48: Example of measurements in Halcon with my calibrated phone camera.

5.3 3D MATCHING AND POSE ESTIMATION

As shown in Section 3.3.2, pose estimation and object identification within 3D scenes are important tasks in many industrial applications.

Halcon implements a wide range of state-of-the-art algorithms for matching 3D objects that differ based on the input data type and method.

The two main types of matching implemented are:

- **Shape-based 3D matching**, in which the 3D template model is rendered from different views and then searched within 2D images taken with a calibrated camera.
- **Surface-based 3D matching**, which is the most advanced type of matching, where the 3D template model is directly searched in the 3D scene. The scene must be represented as a surface mesh, which may be the result of the triangulation of a raw point cloud.

Surface-based 3D matching

Surface-based 3D matching is the most powerful type of matching and is the one that has the biggest impact in industrial applications where, thanks to 3D sensors, there is access to a 3D representation of the scene. Halcon implements state-of-the-art algorithms based on 3D features, reviewed in Section 3.3.2. The steps required to actually perform the matching are:

1. **Create the template object model** by loading the 3D template and computing the point normals needed for the matching algorithm.

2. **Load the 3D scene and prepare it for matching**, optimizing the data for matching and optionally triangulating and computing the normals if not already done.
3. **Perform the matching** by running the matching algorithm, finding the occurrences of the template in the scene along with their pose.

In Chapter 6, I will demonstrate how this is implemented in an industrial application designed by me.

5.4 ROBOT VISION

One of the main industrial applications that uses 3D vision is robot vision, where 3D sensors are used to guide robotic arms in order to perform actions within the environment. Typically, the sensor coupled with the robot is a 3D camera, which can either be mounted on the robotic arm and move with it or be statically mounted at a fixed point. In either case, a special calibration is needed since it is necessary to translate the coordinates from the camera system into the coordinates of the robotic system. This type of calibration is called end-eye calibration (Figure 49).

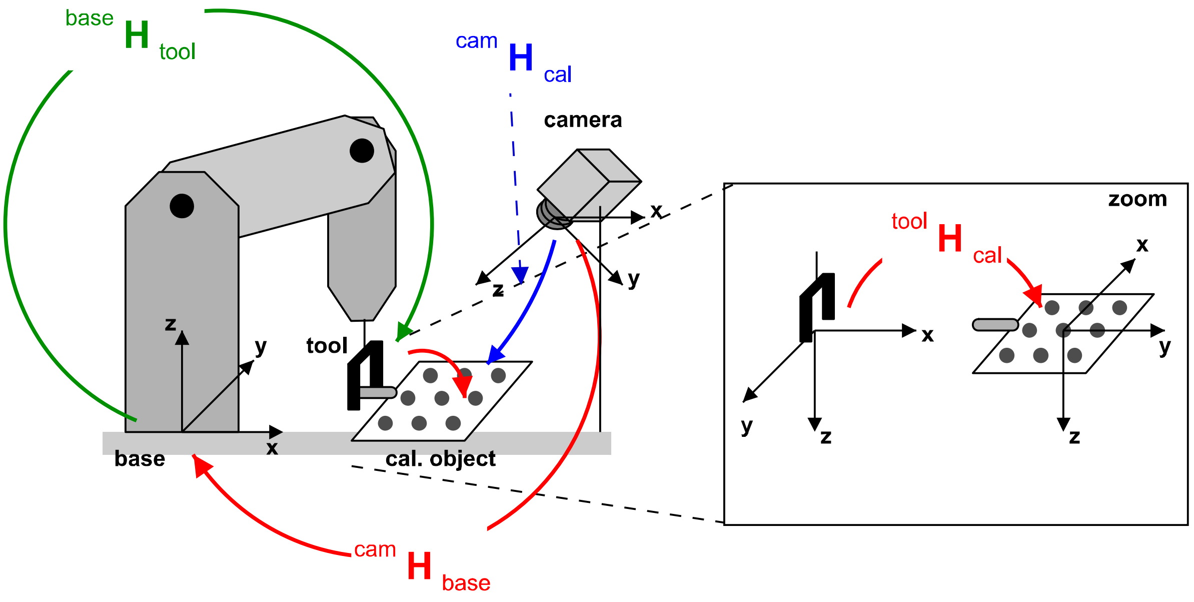


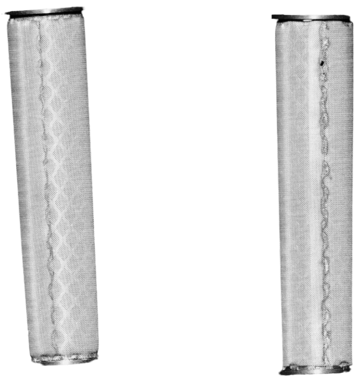
Figure 49: Transformations between frames with a stationary camera.

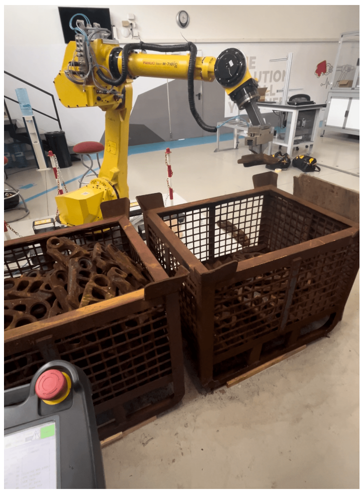
To perform the calibration, the calibration plate must be moved by the robot during the acquisition of the 3D camera. During the process, the transformations between the robot and the camera, as well as between the robot and the calibration object, are estimated:

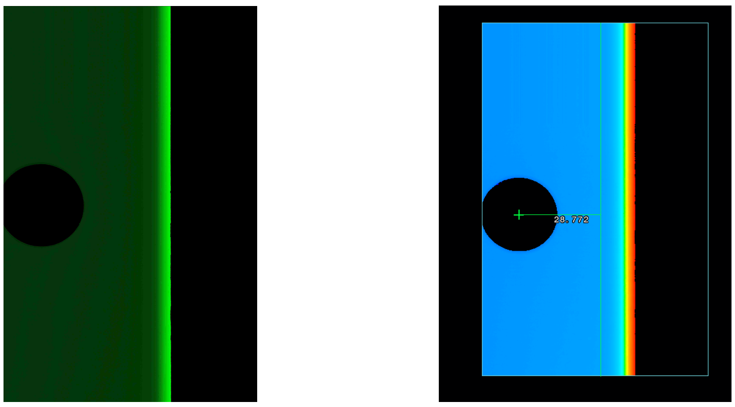
$$\mathbf{H}^{\text{cam}}_{\text{cal}} = \mathbf{H}^{\text{cam}}_{\text{base}} \underbrace{\mathbf{H}^{\text{base}}_{\text{tool}}}_{\text{known}} \mathbf{H}^{\text{tool}}_{\text{cal}}$$

5.5 SOME RELATED PROJECTS AT INNOVA SRL

During my internship at Innova Srl, I had the opportunity to participate in various industrial applications that utilized 3D computer vision. These experiences were important to understand how to apply the discussed concepts in practice, learning from the expertise of other engineering colleagues. The methods applied ranged from low-level data acquisition techniques to high-level inference through standard and deep learning methods. Figure 50 shows some of these mentioned applications. In particular, Figure 50a illustrates the use of deep learning techniques to detect defects in the welding of tubes, Figure 50b shows a pick-and-place robotic application used to move objects from one box to another with the help of a 3D camera to acquire the point cloud, and Figure 50c shows an application where holes in industrial parts were detected along with their distance to some reference points.


(a) *Welding defect detection.*


(b) *Pick and place application.*



(c) *Detection of the distance of the circle from the curve in a industrial object.*

Figure 50: Some applications at Innova Srl.

6 3D SYSTEM FOR ROBOT GUIDANCE AND PRODUCTS ANALYSIS

During my internship at Innova Srl, I was presented with an open problem that consisted in the development of a 3D acquisition system for an industrial application. This application involved objects moving on a conveyor belt and the subsequent analysis of the registered objects for a general purpose robot guidance system. The goal was to design a 3D vision system for the acquisition of the point clouds of general objects on the conveyor belt that needed to as precise, reliable, easy to set up and cost effective as possible. The designed system would then be applied to various 3D industrial applications, such as detecting the pose of bread moving on a conveyor for guiding a robotic arm. This arm would then cut the top surface of the bread before it proceeds through an industrial oven for better cooking (Figure 51).

I then proceeded to propose my own solution to the problem, implementing and testing all the components that make up the final application.

6.1 INTRODUCTION TO THE APPLICATION

The application can be fundamentally divided into two main components:

- The **data acquisition module**, which must handle the unprocessed data from the sensors and manage the communication of that data from the sensor controller to the PC that integrates the whole application. This module is general and can be applied to all sorts of objects moving on a conveyor belt.
- A **high-level module** dependent on the application for the post-processing of the acquired 3D point clouds. In the example application, this involves detecting the bread and its pose and guiding a robotic arm on the surface of the detected bread.

In the example application, the detailed steps are:

1. Comparative analysis of commercially available technologies for 3D image acquisition and choice of the sensors used.
2. Development of an application in C# for the communication of the controller of the sensors with the PC;
3. Integration of HALCON libraries in C# to create a unified solution, executable on the PC.

4. Low processing of the acquired images in HALCON, removing the undercut and the distortion due to the relative movement of the objects with respect to the conveyor belt and the reconstruction of the point cloud with the correct coordinates in the coordinate system of the belt.

5. Triangulation of the obtained point cloud, and pose estimation in HALCON.

6. Communication of the pose to a robot system and movement of the robotic arm in the desired position.

7. Integration of the solution into a Windows .Net form as a graphics interface that allows the visualization and the real-time adjustment of the operational parameters, as well as the preview of the analysis results.

The whole application is represented graphically in Figure 52.
For this particular application, I chose to perform all the computer vision processing using HALCON integrated with the C# language. This setup allows to handle the communication with the controller using dedicated libraries and to manage the socket communication with the robot.

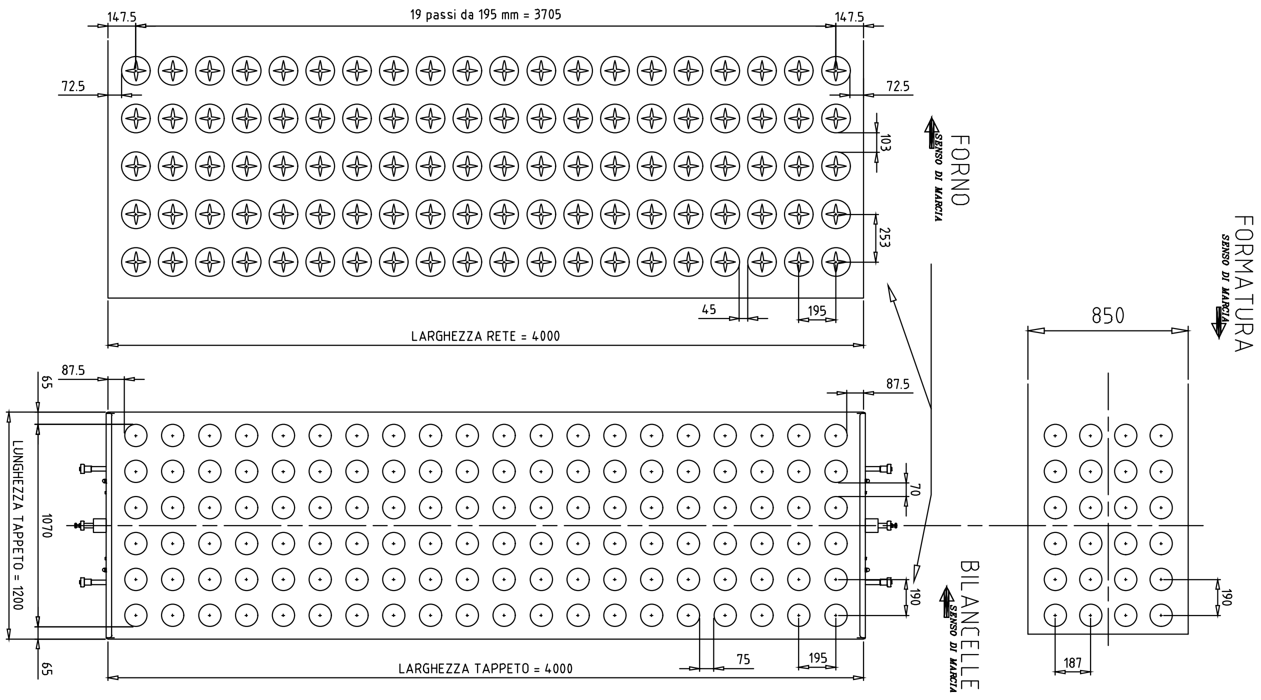


Figure 51: Layout of the example application

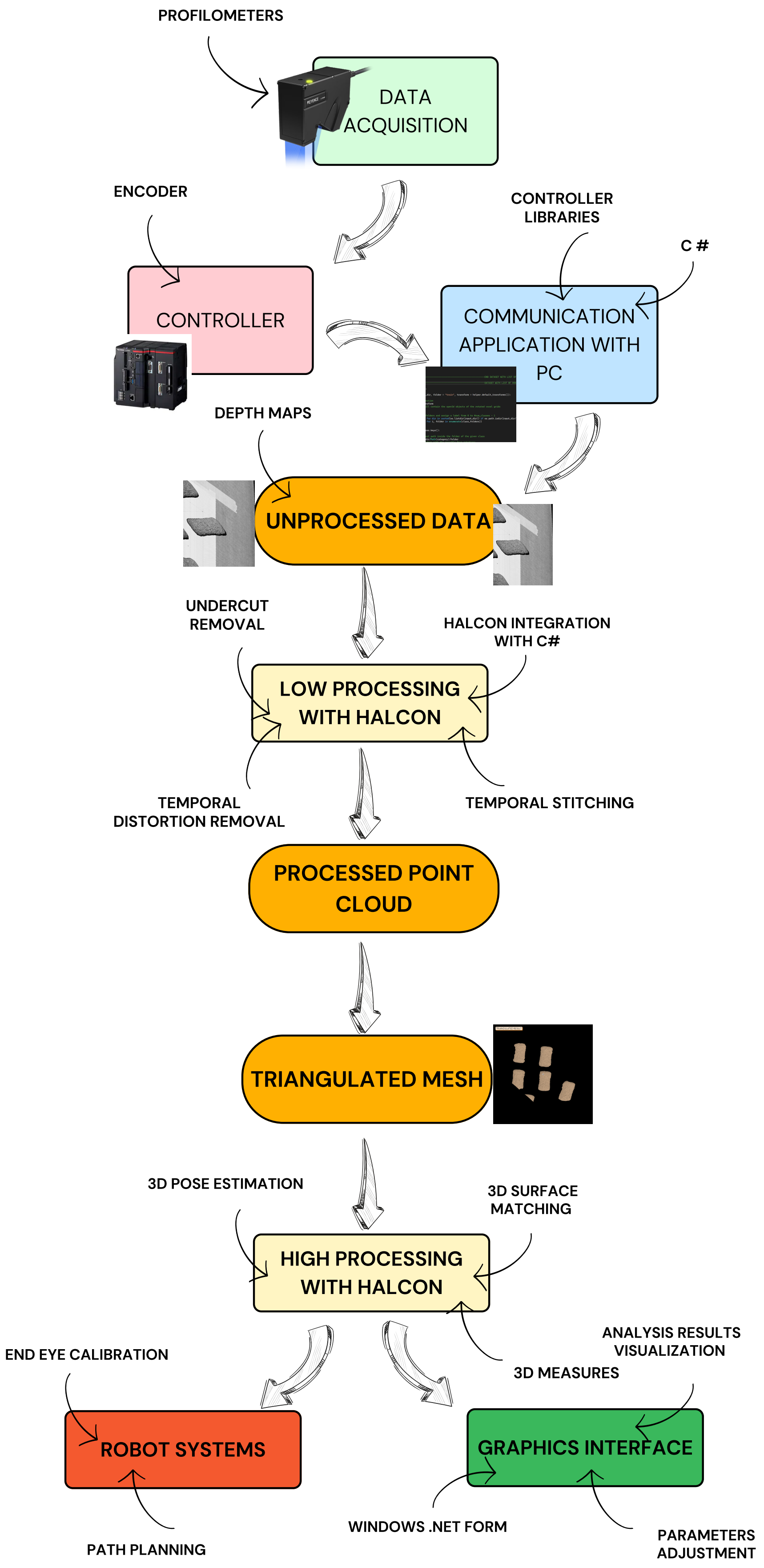


Figure 52: Visual Representation of the application's steps

6.2 DATA ACQUISITION

The first part of the application consists of choosing the right sensor to be used in the acquisition model. This sensor should be selected according to the application context and requirements, also considering the cost. The main requirements for the target industrial applications are:

- The data acquisition technique should be able to work with objects moving on a conveyor belt.
- The object's size to be considered ranges from millimeters to meters.
- The system must be as robust as possible to changes in conditions.
- The system should be easily configurable and require minimal adjustments over time.
- The system should be accurate and have a resolution of at least millimeters.
- The data acquisition should be a fast process.

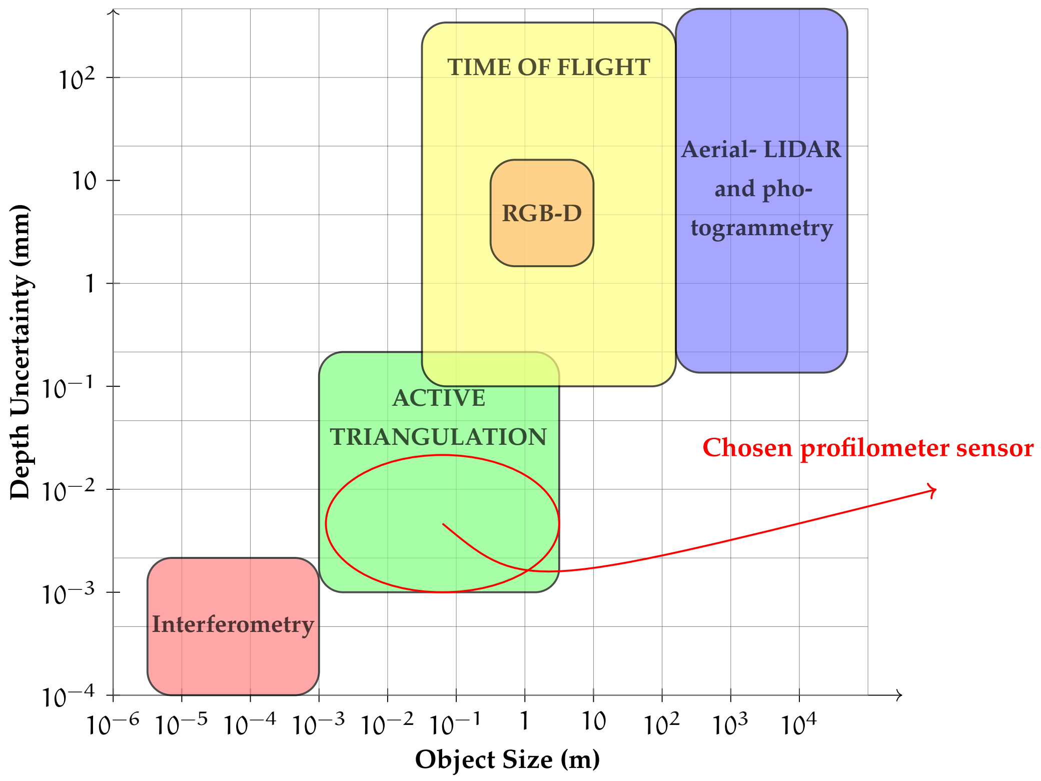


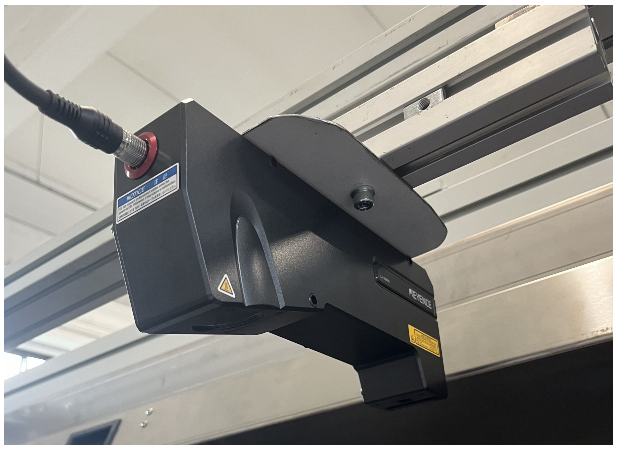
Figure 53: Graph showing 3D data acquisition characteristics based on object size and accuracy. Data taken from Pears et al., 2012.

Based on the given requirements, passive acquisition techniques were not considered since they cannot be considered reliable and accurate enough, as shown in Table 3.

Table 3: Differences of active and passive data acquisition sensors.

	Advantages	Disadvantages
Passive	• Can capture complete images. • Low cost.	• Cannot capture 3D structure in smooth, textureless regions. • Highly dependent on illumination conditions. • Computationally intensive.
Active	• Can determine 3D features also in smooth, textureless regions. • More robust to illumination changes. • Good accuracy.	• Higher cost than passive systems. • They generally require multiple acquisitions to get the full image.

Considering only the active sensors, we need to take into account the working area of each technique, as illustrated in Figure 53. The best sensor that met all the specifications was the stripe scanner, which is based on active triangulation techniques. In particular, all tests were conducted using the Keyence LJ-X8300 profilometer.


(a) *Keyence profilometer.*

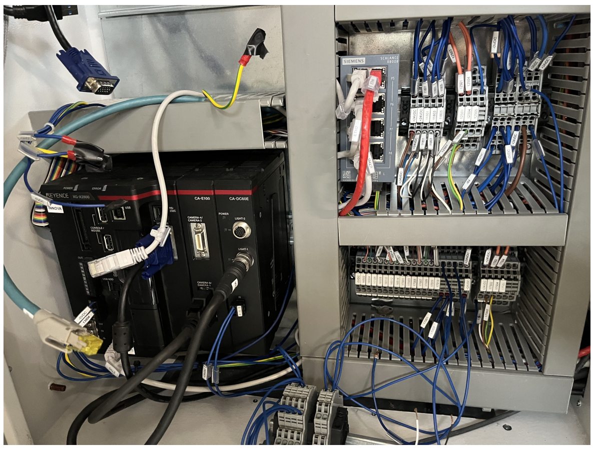

(b) *Controller and alimentation.*

Figure 54: Hardware configuration of the data acquisition module. On the left is the Keyence profilometer, and on the right is the controller with its connections.

This system poses the following main challenges:

- Reconstruct the complete images from the single line scans.
- Reduce the number of profilometers needed to acquire the entire conveyor belt for cost reasons.
- Minimize the problem of dead zones.
- Remove distortions and accurately reconstruct 3D positions.

Reconstruction of the Depth Image and Point Cloud from Single Line Scans

The chosen profilometer provides, at each sampling time t_k, an array of integer values using 16 bits that correspond to the z distance of the surface point from the profilometer. The first bit is used as a flag for dead zone detection, so to convert each of these values $V(i)$ into a metric distance, we use:

$$z_{t_k}(i) = V(i) \times \underbrace{\frac{Z_s}{2^{15}-1}}_{\textbf{z-resolution } r_z} \tag{22}$$

where Z_s is the depth field of the profilometer used and $V(i)$ is the integer value of the sampled point. This calculation must be repeated for each cell in the output array at each sampling time. To reconstruct the entire object, the profilometer must be moved in a direction while sampling the object. To calculate the resolution along the axis of movement, an encoder is needed to provide the velocity of the profilometer at each instant:

- If we keep the sampling frequency f constant, the resolution along the axis of movement x is not constant. Given the velocity measure of the encoder ν_k between two sampling instants, the x_k position at the sampling instant k is given by

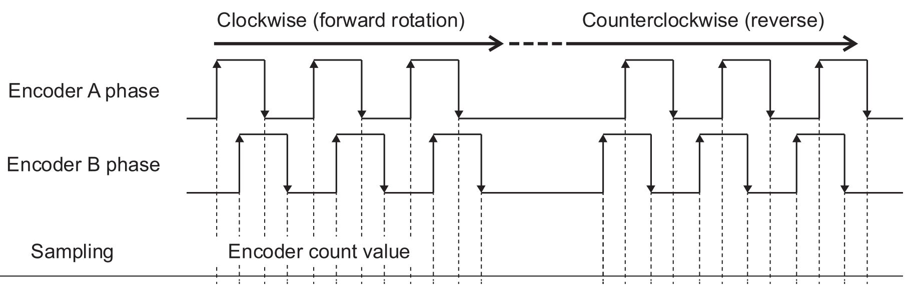
$$x_k = x_{k-1} + \frac{\nu_k}{f} \tag{23}$$

- Given the velocity measurements from the encoder, we can set a constant resolution r_x along the x-axis by adjusting the sampling frequency over time, taking into account that the profilometer can achieve up to a maximum frequency. In this case, the position x_k is given by:

$$x_k = x_{k-1} + r_x$$

This second approach is the most effective because it ensures uniform sampling of the object along the direction of movement, preventing distorted portions caused by variations in velocity and reducing the likelihood of poor results.

To achieve this, we must first consider the characteristics of the encoder. Specifically, the encoder sends pulses to the profilometer, and we need to calculate the moving distance corresponding to each pulse, which depends on the number of encoder ticks, the diameter of the pulley installed on the encoder, and the sampling specifications (Figure 55). Once the moving distance per pulse is determined, we can calculate the number of pulses to wait before triggering the line scan to maintain a constant resolution. Naturally, the number of pulses received by the encoder within a given time period will depend on the movement's velocity. Uniform sampling is achievable up to the point where the maximum frequency of the profilometer is reached.



Detection direction	Sampling	Encoder count value
None	1X	1 2 3 4 5 6
	2X	1 2 3 4 5 6 7 8 9 10 11 12
	4X	1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Yes (CW direction)	1X	1 2 3 3 3 3
	2X	1 2 3 4 5 6 6 6 6 6 6 6
	4X	1 2 3 4 5 6 7 8 9 10 11 12 12 12 12 12 12 12 12 12 12 12 12 12
Yes (both directions)	1X	1 2 3 2 1 0
	2X	1 2 3 4 5 6 5 4 3 2 1 0
	4X	1 2 3 4 5 6 7 8 9 10 11 12 11 10 9 8 7 6 5 4 3 2 1 0

Figure 55: Encoder phases and relationship between sampling and direction detection Keyence, n.d.

Chosen the sampling factor k, and given the length of the encoder's pulley L_{enc} and the number of steps N_{enc} in the encoder for a full spin we first calculate the moving distance Δx per pulse that is given by:

$$\Delta x = \frac{L_{enc}}{N_{enc} \times k}$$

so the number of pulses N_{pulses} to wait for the target resolution r_x is given by

$$N_{pulses} = \frac{r_x}{\Delta x}$$

Along the line scan direction y, the resolution r_y is constant and is given by the field of view of the profilometer L and the number of points N for each scan, so the position $y_k(i)$ of the i-th pixel at time k is given by:

$$y_k(i) = i \times \underbrace{\frac{L}{N}}_{\textbf{y-resolution } r_y}$$

The characteristics of the Keyence profilometer used can be found in the datasheet, as partially shown in Figure 56.

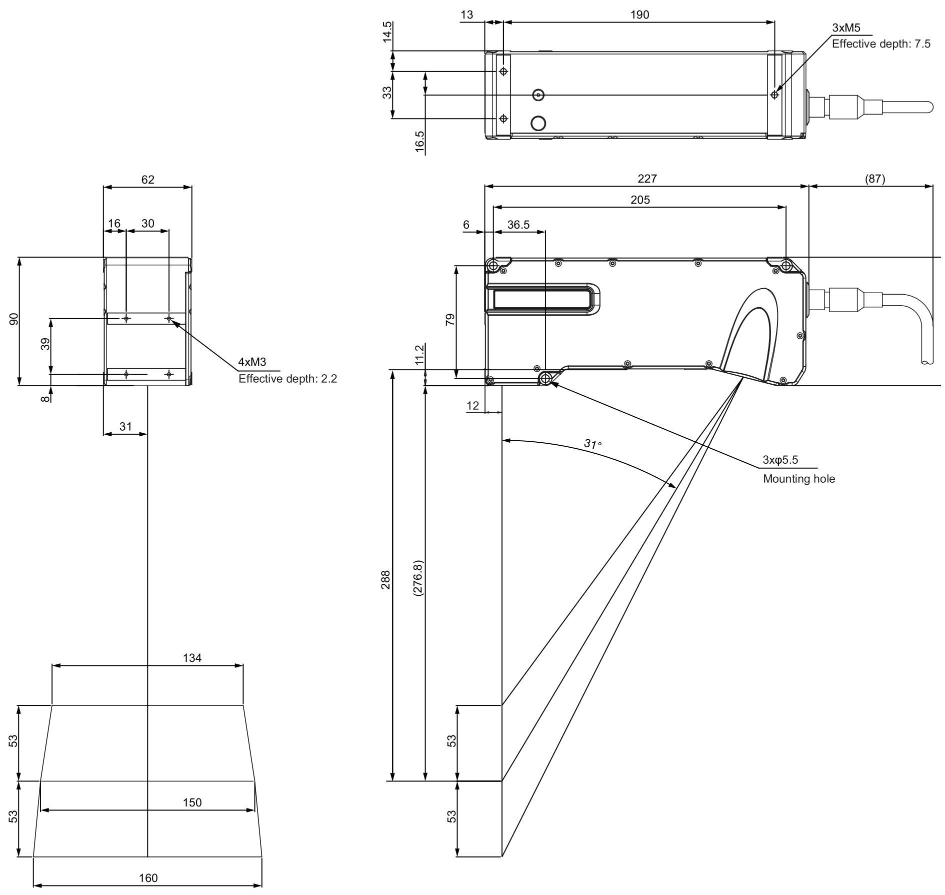


Figure 56: Profilometer LJ-X8300 by Keyence used for the data acquisition Keyence, n.d.

To reconstruct the point cloud efficiently, we create three support images, one for each metric coordinate. To obtain the z-image, we simply stitch together the profilometer's line scan output and apply the transformation from Equation 22. For the y-image, we generate a gradient with a constant resolution r_y. The x-image can be created similarly to the y-image in the case of constant resolution r_x, by generating a gradient in the opposite direction. However, in the case of varying frequency, each row must be computed separately using Equation 23.

Number of profilometers reduction

The chosen profilometer sensor meets all the requirements for the application, but it comes with a high cost. This issue is even more alarming in the target application, where the conveyor belt width typically ranges from 4 to 5 meters, while most profilometers on the market have a field of view of only up to 1 meter. Consequently, the most common approach would be to use multiple profilometers fixed along the direction perpendicular to the belt's velocity, with each covering a small portion of the belt. Although this approach is effective and easy to implement, it significantly increases the overall cost due to the need for multiple sensors.

To reduce the number of profilometers, we could use a single profilometer with a scanning direction parallel to the velocity of the conveyor belt, moving along an axis perpendicular to the belt's velocity. However, this approach introduces several challenges:

- While the profilometer moves perpendicular to the belt, the objects on the belt continue moving forward. As a result, the stitching of the scan lines produces depth images that are temporally distorted due to the relative movement between the objects and the profilometer. These distorted images must be processed and corrected before applying the previously described procedure, otherwise the resulting coordinates will be inaccurate.
- Consecutive scans must overlap on a portion of the conveyor belt to ensure that all objects are captured. Additionally, stitching these scans together to produce a final image is a non trivial task.

The proposed setup is illustrated in Figure 57.

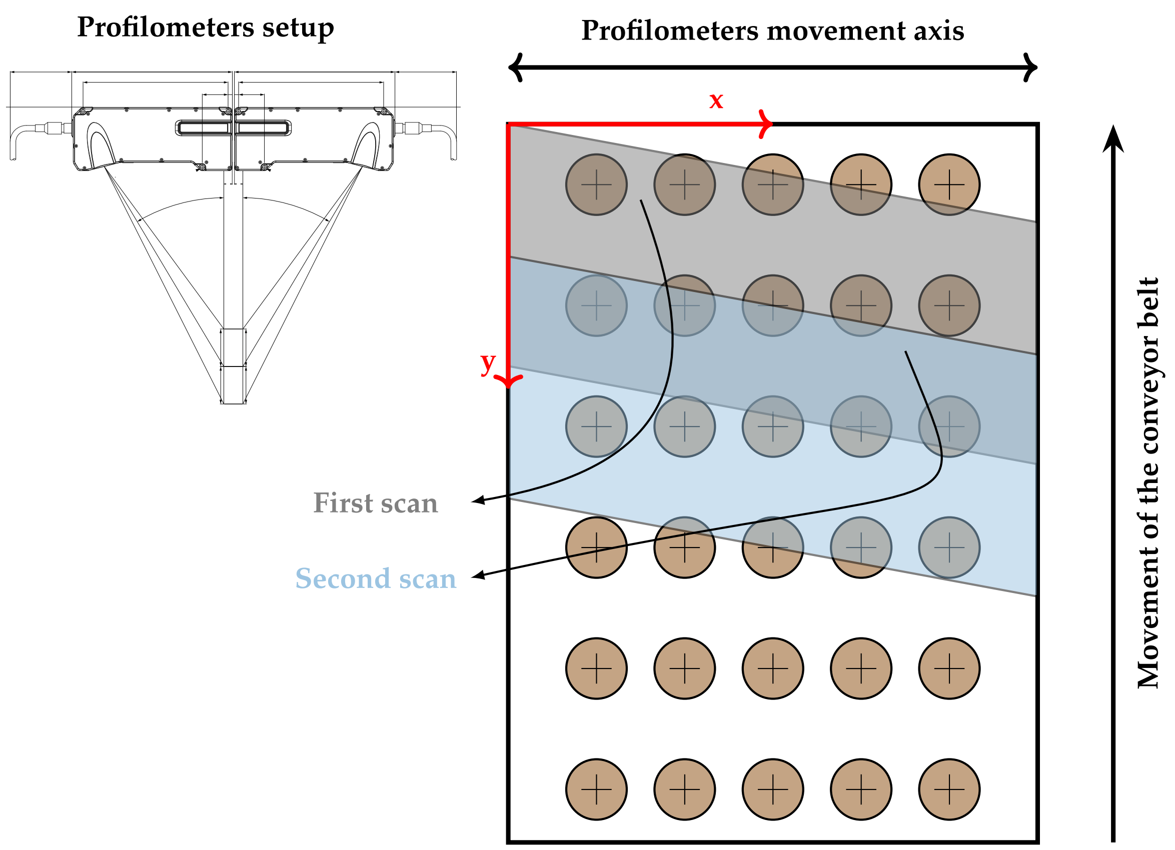
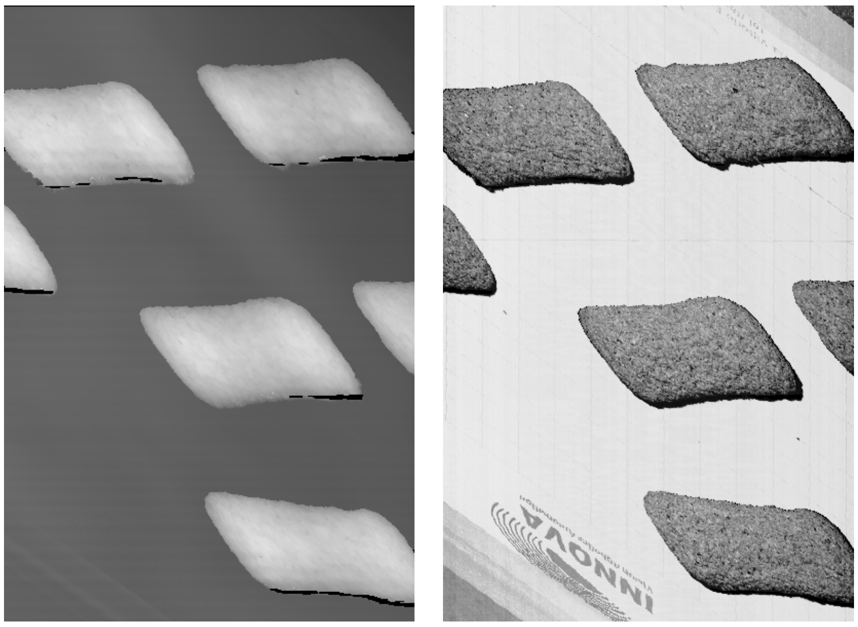


Figure 57: Subsequent scan area taken by the profilometers in relation to time t_0.

One important consideration is that the conveyor belt generally moves at a uniform velocity, which is monitored by an encoder. The area covered by a single scan, relative to a fixed position of the object on the belt, forms a trapezoidal shape, with the inclination of its edges depending on the relative proportions of the conveyor belt's velocity and the profilometer's velocity if the profilometer moves at a uniform speed. However, if the profilometer's velocity is not uniform, the shape of the covered area becomes more complex, making the reconstruction of accurate coordinates more challenging. In such cases, the velocity of the profilometer at each instant must be known, which requires another encoder on the profilometer's axis.

To address this, it is crucial to control the profilometer's velocity over the target area, allowing for a longer axis of movement outside the belt to accelerate and decelerate the profilometer. If this is not feasible, the procedure outlined in this thesis must be applied at each encoder tick, effectively linearizing the behavior over short time intervals and applying the proposed method.

To test all the designs in this thesis, I used a batch of Pavesini biscuits as the test objects, which were easily available and had long shapes that effectively showcased the results. As previously mentioned, this module is designed to be reproducible in various applications of different natures, making the choice of the test object irrelevant.
Figure 58 shows the result of the simple stitching of the scan lines using the proposed design. The need for further processing is evident, particularly in addressing distortions and dead zones.



(a) *Depth image* (b) *Intensity image*

Figure 58: Images generated by stitching all the scan lines of a profilometer without processing. The distortion of the shape due to the relative movement is clearly visible.

Profilometers setup to remove dead zones

Although the proposed design significantly reduces the number of profilometers needed, using only one can present problems, as evidenced by the results shown in Figure 58. In particular, dead zones are evident at the borders of the objects, where depth and intensity data are missing. This occurs because part of the light emitted by the sensor is intercepted by the object's surface (Figure 59), preventing it from reaching the receiver.

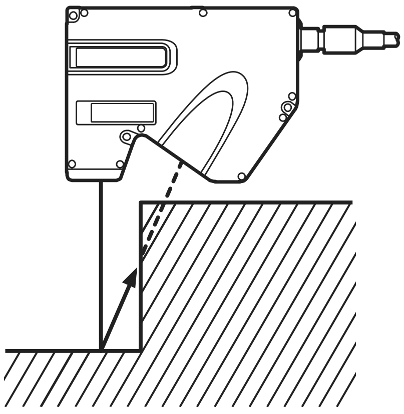


Figure 59: Illustration of the dead zone problem.

This can be ameliorated using another profilometer that is rotated of 180 degrees that scans the same area of the first one as in Figure 60.
Using this design, when one profilometer encors in dead zones, the other one is most likely able to detect the surface. The data coming from the two profilometer must then be processed in the application and the two images must be matched and the data must be combined to produce a final output that excludes the dead zones.

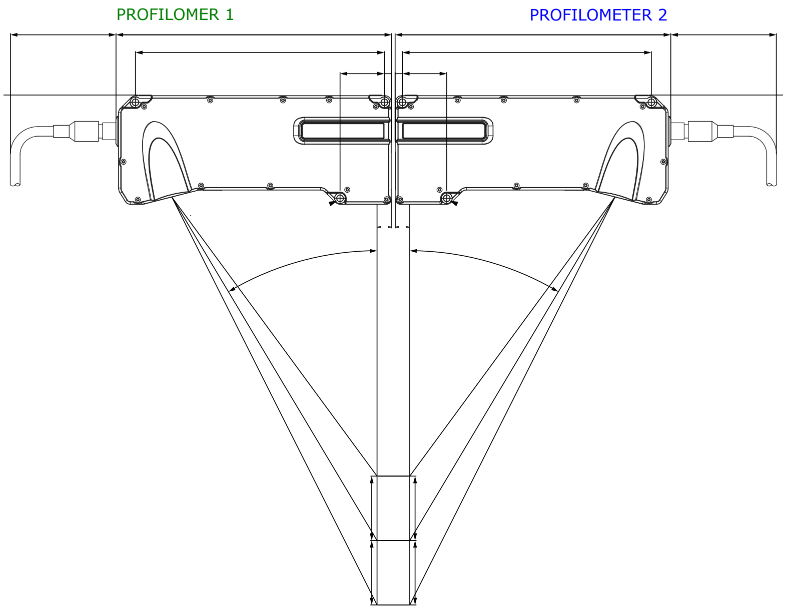


Figure 60: Profilometers setup to avoid dead zones.

Figure 61 shows the result of processing that combines the images from the two profilometers and removes the dead zones in the direction of movement. The same processing is applied to the depth maps.

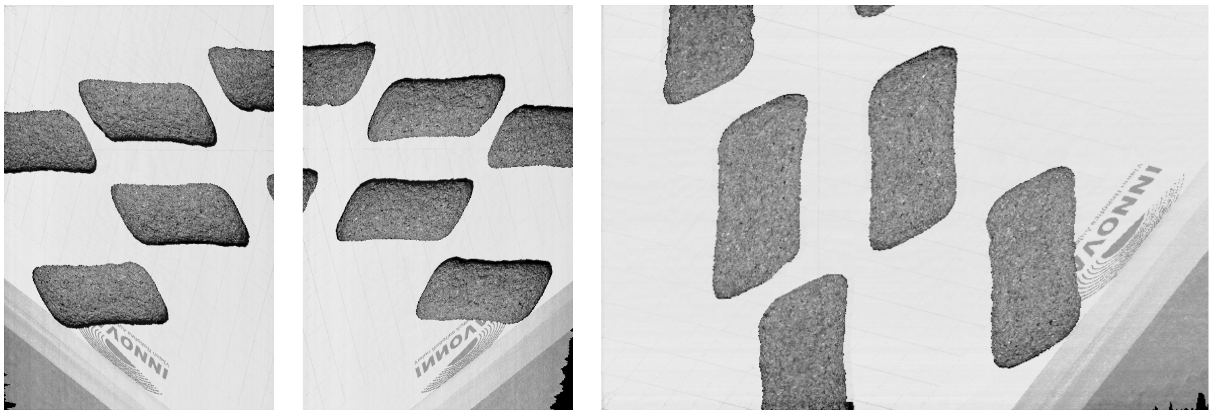

(a) *Left profilometer.* (b) *Right profilometer.* (c) *Processed image to remove the dead zones.*

Figure 61: Removal of the dead zones.

6.3 PC COMMUNICATION WITH THE CONTROLLER

The profilometers used are connected to and controlled by a controller unit (Figure 62), which is primarily responsible for triggering data acquisition and buffering the data. The data from the sensors must be integrated into a C# application running on a PC, where processing will be carried out using HALCON software. Therefore, it is necessary to develop a communication software that manages the transfer of the depth data from the controller unit to the PC application.

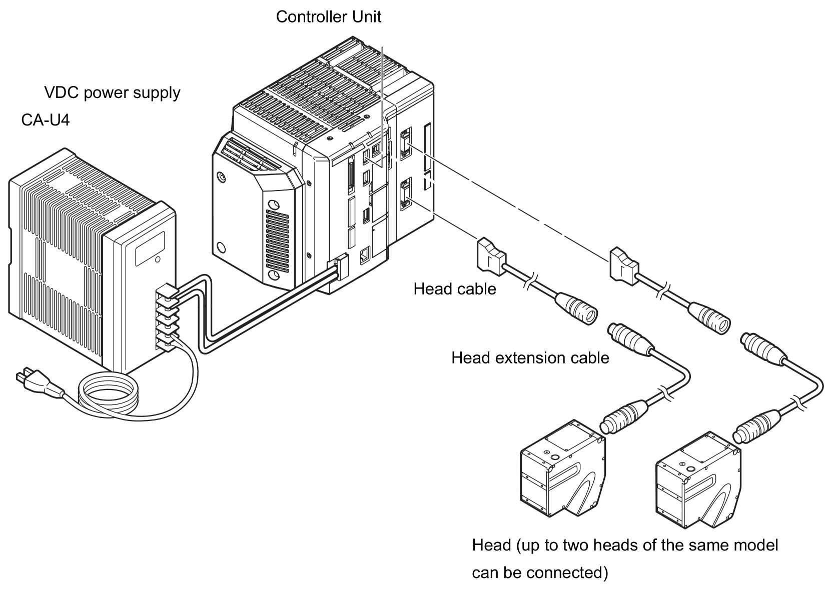


Figure 62: System configuration Keyence, n.d.

Figure 63 shows the data flow.

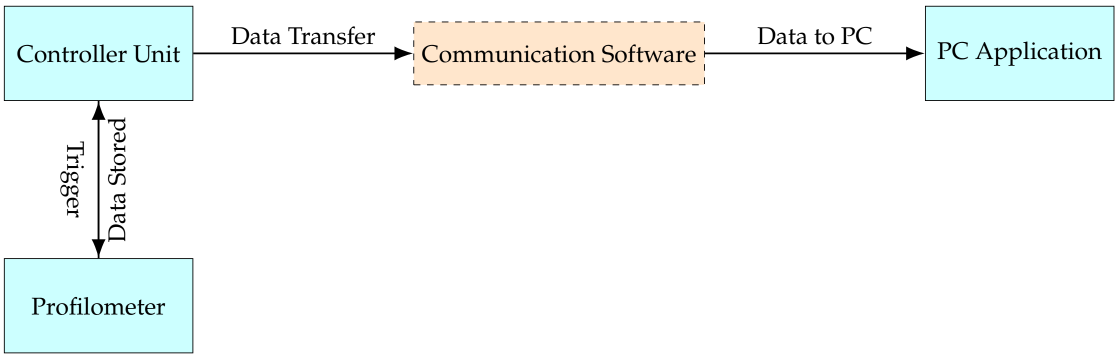


Figure 63: Data flow in the system with the communication application.

The communication software is built on the native functions of the Keyence libraries for the controller. When high-speed communication is initiated, the profile data is stored in the internal buffer until a predefined number of profiles are accumulated, triggering a callback function that stores the data within a thread internal to the application. The entire communication software is quite extensive, and in the appendix I present the core function responsible for acquiring the profile data using the mentioned functions.

6.4 LOW PROCESSING OF THE IMAGES USING HALCON

Once the raw data is acquired and integrated into the PC application, HALCON is used to process it and address the challenges that arised from the acquisition design, in particular:

1. Removing the distortion of the depth image caused by the relative movement between the conveyor belt and the object, and subsequently reconstructing the correct point cloud of the single scan with accurate metric positions.
2. Stitching the point clouds from consecutive scans and creating a continuous mesh through triangulation, which will be used for the inference and analysis stages of the application.

6.4.1 Removal of the time distortion

To undistort the image, each profile acquired at a given time t must be corrected with an offset that increases as the profilometer moves closer to the end of the belt. This adjustment accounts for the fact that the object observed at time $t + \Delta t$ has moved, so simply stitching the line scans together would place it in a forward position. However, we aim to determine the correct positions of the entire belt scan area as they were at the initial time of the scan. By establishing these correct initial positions and knowing the start time, we can accurately predict the objects' positions at any given time, using the belt's velocity provided by the encoder.

To compute the offset, we first consider that this distortion primarily affects the y-coordinates, assuming uniform velocities. In this case, the offset between two consecutive scans remains constant. Therefore, to calculate the offset for the i-th scan, we simply determine the offset between two scans and then multiply this offset by i to obtain the correct y-coordinates.
Since the x-resolution r_x is fixed by the procedure described above, and the profilometer velocity v_x is known, we can calculate the fixed time interval between two line scans as:

$$\Delta t = \frac{r_x}{v_x}$$

During this time interval Δt, the conveyor belt moves with a constant velocity v_y, allowing us to calculate the offset Δy between two scans:

$$\Delta y = v_y \, \Delta t = v_y \, \frac{r_x}{v_x}$$

Thus, the offset to be added to the i-th scan in metric units is:

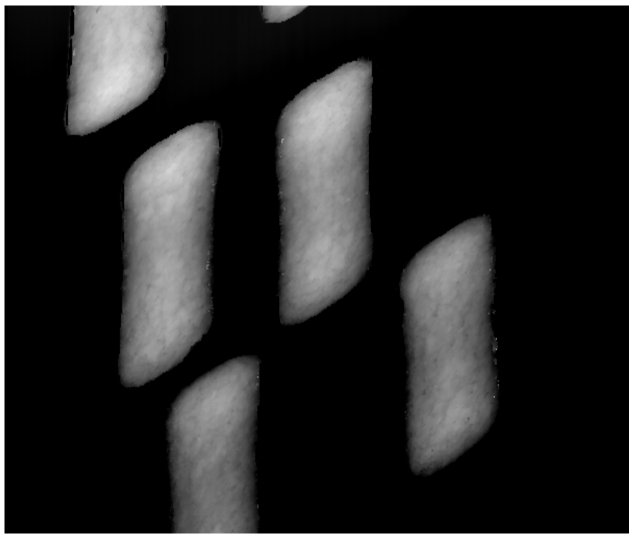
$$\text{offset}_m^i = i \, \Delta y = i \, v_y \, \frac{r_x}{v_x} = i \, r_x \underbrace{\frac{v_y}{v_x}}_{\tan(\alpha)} \tag{24}$$

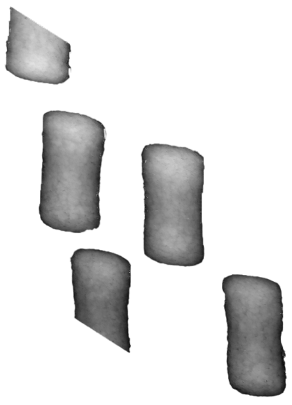
Here, the velocities v_x and v_y can be interpreted as the components of a relative velocity v between the objects and the belt, with α representing the angle formed between the v and the v_x component. This interpretation also allows to test the design without a moving conveyor belt, by simply moving the profilometer over a fixed arrangement of objects at a given angle.
If we also want to adjust the depth and intensity images to visually verify that this procedure works, we can calculate the y-offset for the i-th column as follows:

$$\text{offset}_{\text{pix}}^i = i \, \Delta y = i \, v_y \, \frac{r_x}{v_x} = i \, \frac{r_x}{r_y} \, \frac{v_y}{v_x}$$

After the point cloud is processed, it is further transformed into a mesh through triangulation for the subsequent steps of the application.

Figure 64 shows the result of this processing. In the appendix, I provide the code for both the correction of the depth/intensity images and the point cloud.


(a) *Raw depth data with dead zones removed.*


(b) *Processed depth data with position correction.*

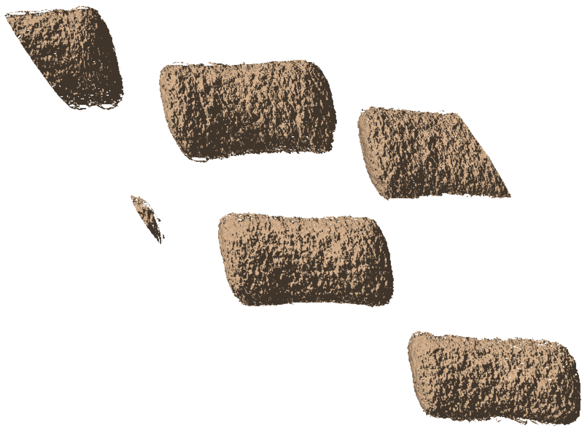

(c) *3D triangulated mesh result.*

Figure 64: Result of processing starting from the raw data to the triangulated mesh.

6.4.2 Time Stitching and Creation of the Point Cloud

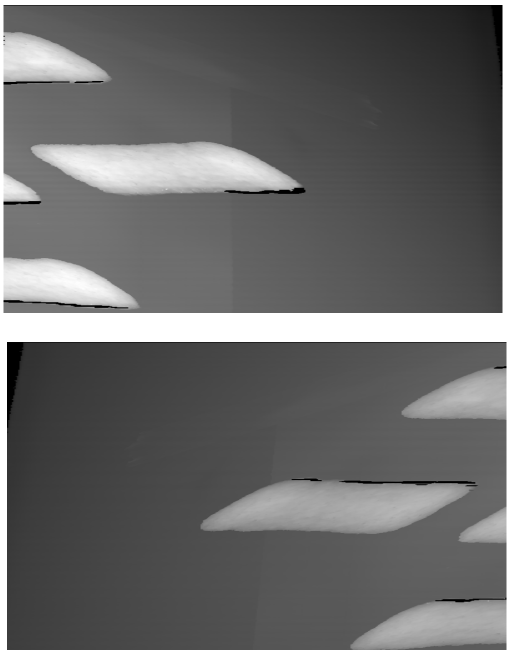
In general, a single scan can cover an area with full objects and only partial objects at the borders. Therefore, it is necessary to stitch together data coming from two consecutive scans to fully detect all the objects throughout the entire process. This can be achieved by directly using the corrected point clouds from the two scans and modifying the second scan to align every position with the initial starting instant of the first scan. This is done by adding a constant offset to the y-coordinates of the second point cloud. The offset depends on the constant velocity of the belt and the time taken by the profilometer to complete one scan and return to the initial position.

Given the time T_c of one cycle of the profilometer, and the belt velocity v_y this is simply given by:

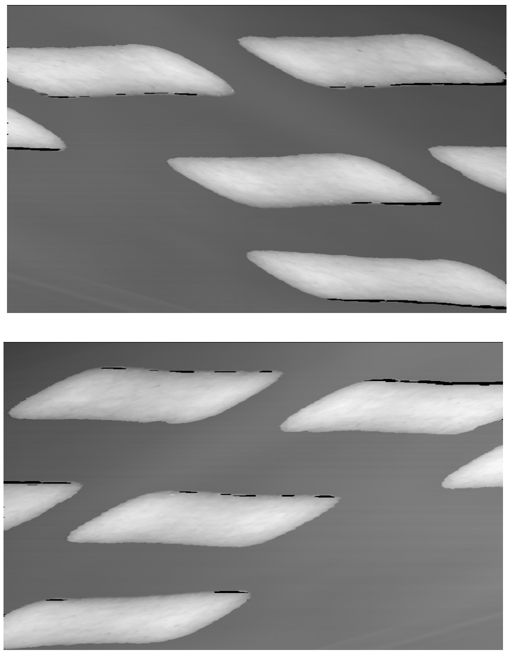
$$\text{offset}_t = v_y \, T_c$$

I now present the full results of the entire process in Figure 65. To recap the entire process, the following images are shown:

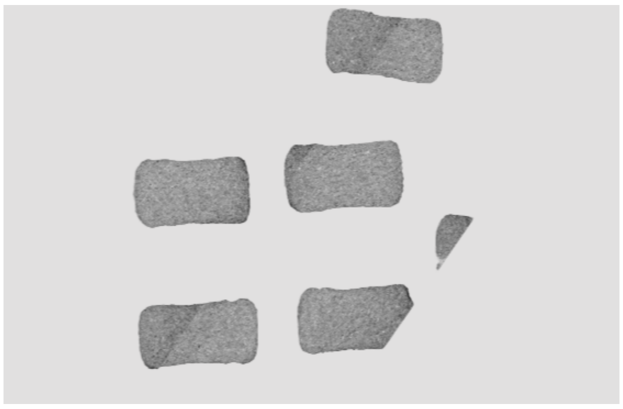
- The raw depth images of two subsequent scans from the left profilometer, obtained by stitching together the line scans (Figure 65a).
- The raw depth images of two subsequent scans from the right profilometer, obtained by stitching together the line scans (Figure 65b).
- The reconstructed intensity image (that is used for visualization) after removing the dead zones, correcting the coordinates, and stitching the two subsequent scans with the computed offset (Figure 65c).
- The final triangulated mesh that is the result of all the processing steps and that is used as the starting point of the next phases of the application (Figure 65d).



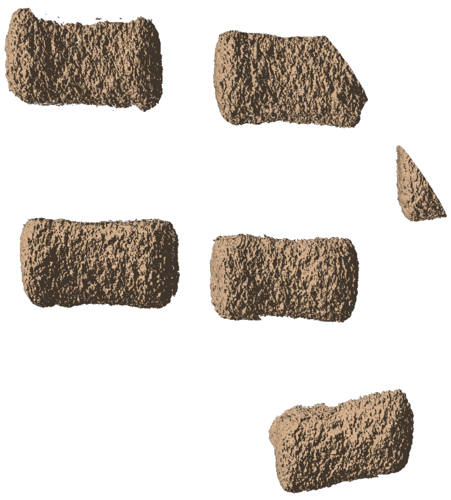
(a) *Raw depth data from left profilometer.*



(b) *Raw depth data from right profilometer.*



(c) *Reconstructed intensity image after all the processing.*



(d) *Final 3D mesh of the two subsequent scans.*

Figure 65: Result of time stitching.

6.5 HIGH PROCESSING USING HALCON

The developed data acquisition module is general and independent of the specific application, making it applicable to a wide range of cases involving objects of different natures. Once the point cloud or triangulated mesh is obtained, the subsequent steps in the application depend on the specific goals of the application. However, in many of these applications, such as picking objects from a conveyor belt and performing actions on their surfaces, it is necessary to recognize and locate the objects on the belt, including determining their pose. This allows to analyze the characteristics of the objects in terms of conformity or defects and/or guiding a robotic arm to perform actions such as picking or cutting the surface of the object, as in the example application involving bread. Finally, I will show how the results of this process can be communicated to a robotic arm to perform specific actions.

Here, I shows the implementation of the most interesting part of identifying the objects on the belt along with their pose, continuing with the test case of the Pavesini objects to maintain the continuity. Additional analyses of the objects can always be incorporated, depending on the specific targets of the application.

To achieve this, I apply surface matching in HALCON with the obtained mesh. The necessary steps are:

- Creation of a triangulated template of the object to be used for searching correspondences within any mesh produced by the scans at any given time.

- Determination of the keypoints on the template and the mesh where the object is being searched, followed by matching to find the correspondences and locate the objects.

It is worth noting that this process could also be performed using other techniques, such as deep learning. However, the time requirements of the application and the well established efficacy of the chosen method led to the decision to use this approach.

6.5.1 Creation of the Surface Model

In order to locate the object within the search area, it is first necessary to create a template that will be used as a reference for this object. This can be done accurately using CAD models or by scanning and triangulating a single object. Alternatively, we can use a full scan from the profilometers as a reference and extract the template from it. To do this, we first need to remove the background and segment the mesh by computing the connected regions. The results of this segmentation are shown in Figure 66.

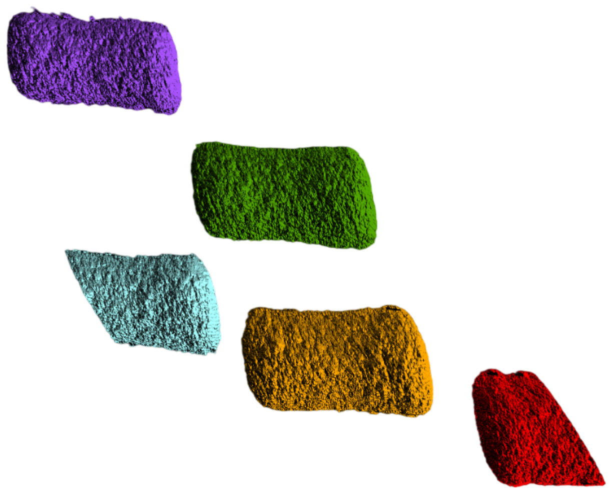


Figure 66: Segmentation.

6.5.2 3D surface matching and Pose Estimation

Once the template is obtained, we need to compute the keypoints in both the template and the search mesh. This step is essential to sample the important features of the object that identify it and to calculate the pose of the object by determining the relative positions of these keypoints. Figure 67 shows the results of the keypoint calculation on a sample mesh.

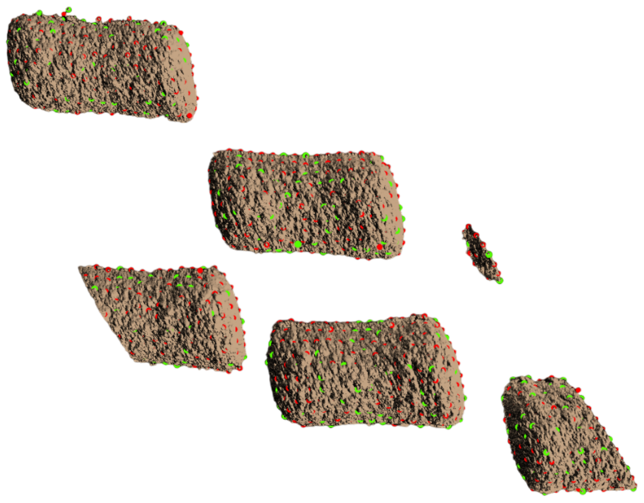


Figure 67: Keypoints detection.

Subsequently, keypoint matching is performed. For each matching keypoint, the optimal pose is computed by pairing the sampled points that have similar distances and relative orientations. A matching score is then output to define the accuracy of the matching. Given the rigid transformation $\mathbf{T_k}$ between the template and the k-th matched object, we can represent the template within the search mesh by performing for each match:

$$\mathbf{H_k} = \begin{pmatrix} \mathbf{R_k} & \mathbf{t_k} \\ \mathbf{o}^\top & 1 \end{pmatrix} \qquad \begin{pmatrix} x_k^{scene} \\ y_k^{scene} \\ z_k^{scene} \end{pmatrix} = \mathbf{H_k} \begin{pmatrix} x^{temp} \\ y^{temp} \\ z^{temp} \end{pmatrix} \qquad \mathbf{R_k} \in \mathbb{R}^{3\times3},\ \mathbf{t_k} \in \mathbb{R}^3$$

Figure 68 shows the results of the surface matching.

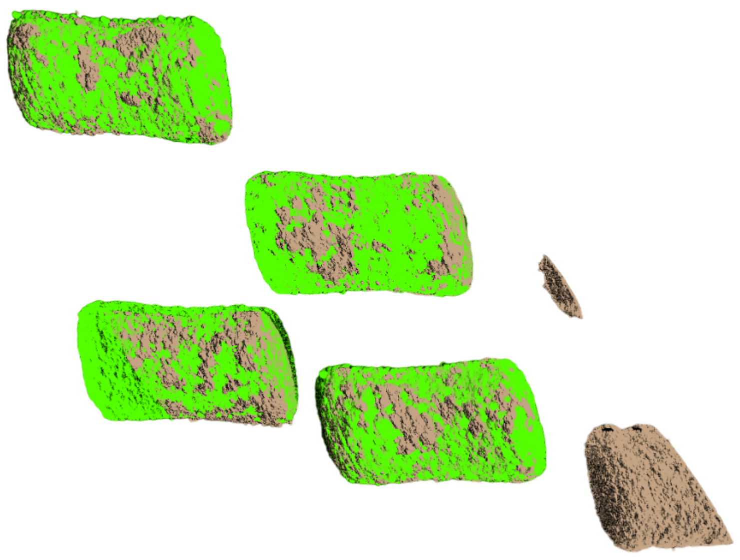


Figure 68: 3D surface matching results. The full target objects inside the mesh are recognized and located with good accuracy.

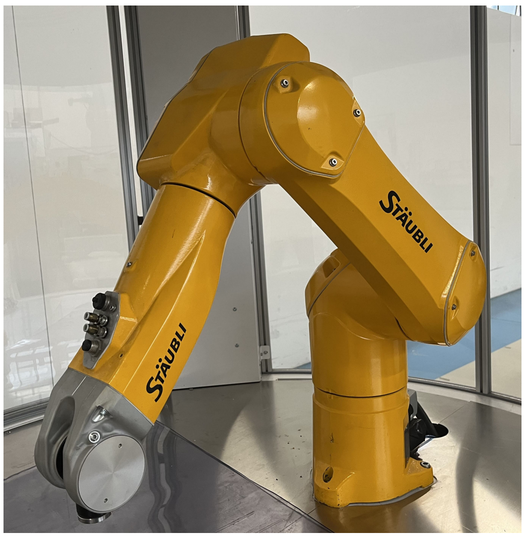
6.6 COMMUNICATION WITH THE ROBOTIC ARM

After the 3D matching, the target objects are detected within the triangulated mesh, along with their pose in the profilometer's reference system. To translate these results into the robotic arm's coordinate system, a known rigid transformation must be applied, which is computed based on the mechanical setup of the application:

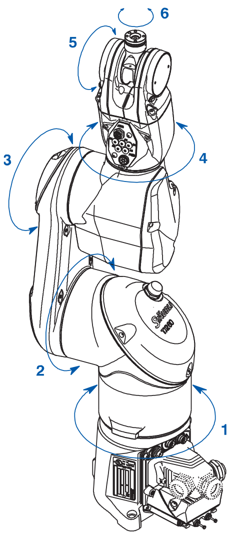
$$\begin{pmatrix} x^{rob} \\ y^{rob} \\ z^{rob} \end{pmatrix} = \mathbf{H}^{rob}_{prof} \begin{pmatrix} x^{prof} \\ y^{prof} \\ z^{prof} \end{pmatrix}$$

Once the coordinates are transformed into the robotic system, they can be communicated to the robotic arm controller so that we can compute the robot's trajectory and perform the required action. The communication is done through a socket connection. This connection was tested with a Staubli robotic arm shown in Figure 69.

```
*open the connection with the specified port at the given IP
open_socket_connect ('192.168.0.50', 1111, 'protocol', 'TCP', Socket)
*send the pose as a string
send_data (Socket, 'z', PoseToSend, [])
```



(a) *Robotic arm.*



(b) *Joints.*

Figure 69: TX2-60 L Staubli robotic arm used at Innova Srl.

6.7 CONCLUSIONS, FUTURE WORK AND EXTENSIONS

The project presented was entirely designed by myself and it represents a design idea that has been implemented in parts to verify its applicability. The main innovation is represented by the acquisition module, where the reduction of the number of profilometers needed can significantly help to reduce the overall cost of the application. To make an idea of the scale of this improvement, typically, at the current date of writing this thesis, the cost of a single profilometer required for this project is around 20,000 euros. Hence, reducing the number from 5-6 to 2-1 profilometers can be significant, especially for the fact that this acquisition system is designed to be as general as possible and applicable to a variety of application in a industrial setting with conveyor belts. The individual components of the design were tested during the internship and the next step is the mechanical production of the system and its final testings. This mechanical design has already been started according to the design idea proposed in this thesis, and Figure 70 shows a rendered image of the top part of the 3D CAD model of the prototype of the presented profilometers setup, with the moving axis mounted above the conveyor belt.

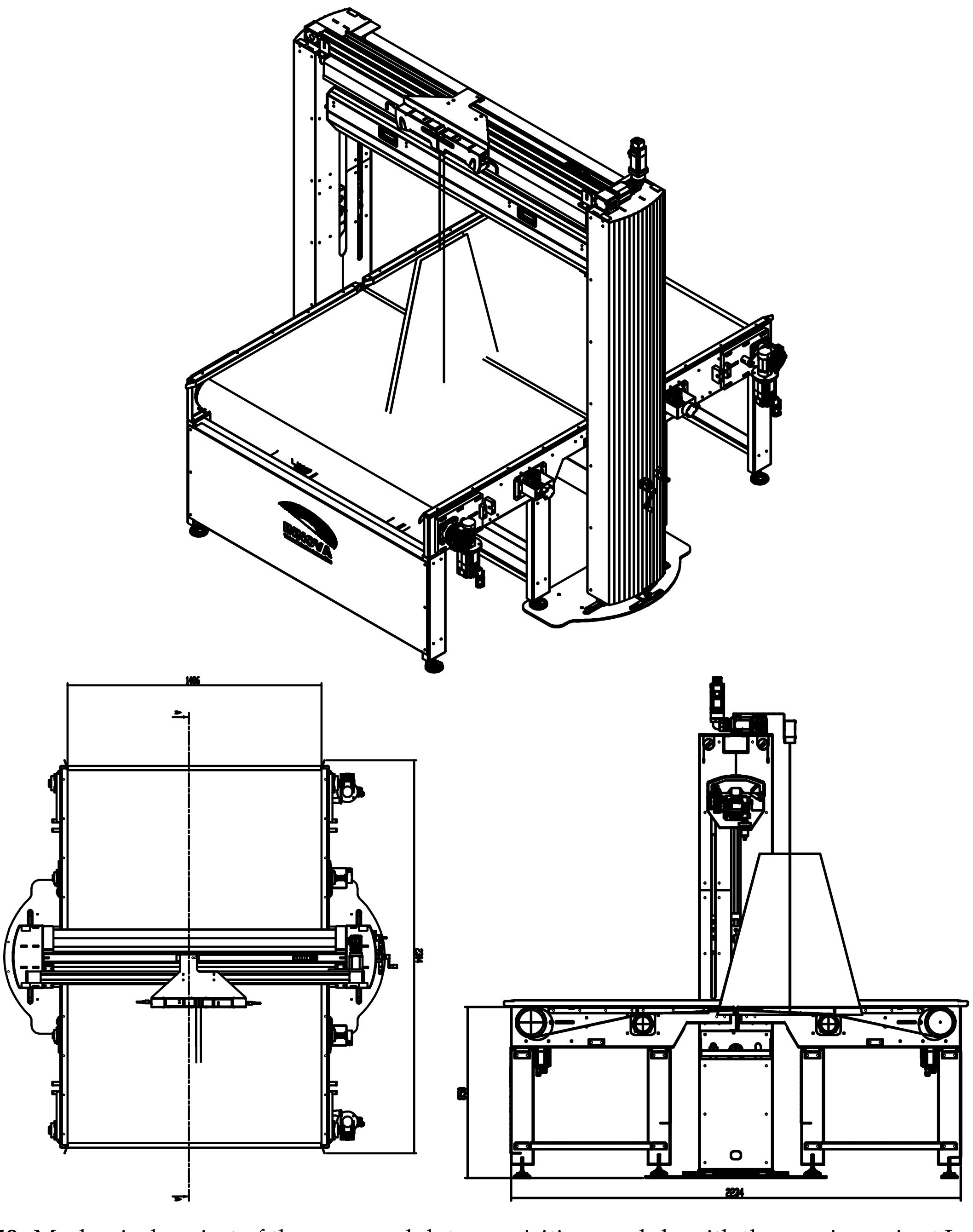


Figure 70: Mechanical project of the proposed data acquisition module with the moving axis at Innova Srl.

The first application in which this design will be used, is the cited application for the bread cutting before cooking. In this case the high-processing module proposed in this thesis with the 3D matching and the pose estimation is ready for use. The only necessary modification is of course the replacement of the template model with a 3D scan of the specific bread used. Additional needed analysis can be added in the same framework of the C# application. I have also shown in this thesis how to communicate the pose of the object to the robotic arm. Further work will be needed for the trajectory planning.

On top of this, I have already started the design of a simple graphics interface to preview in real time the results of the analysis, as well as to change the operating parameters. This interface is designed to be integrated in the same C# application, using the .NET Windows Form framework. All the added analysis will have to be incorporated into the provided baseline form.

B APPENDIX

In the following I present snippets of the code that I developed for the industrial application of Chapter 6. The goal of this is to show how to reproduce the results that were obtained in this thesis, as well as the actual sofware implementation of the proposed design. Of course, this does not entirely constitute the whole application, but only small key parts are presented. In particular, the two different displays indicate code written in Halcon or C#.

Removal of dead zones

```
read_image (IntensityNotFlipped, 'C:/Users/39348/right.png')
read_image (IntensityFlipped, 'C:/Users/39348/left.png')
*****************************************ALIGN IMAGES***********************************************
*get the coordinates of all points in the image
get_region_points (IntensityFlipped, Rows, Columns)
*retrieve the corresponding grayvalue of all pixels
get_grayval (IntensityFlipped, Rows, Columns, Grayval)
*invert the rows indices (this will allow to flip the image)
tuple_inverse (Rows, Rows)
get_image_size (IntensityFlipped, Width, Height)
*set the flipped value of the pixels
set_grayval (IntensityFlipped, Height-1-Rows, Width-1-Columns, Grayval)
**************************************DEAD ZONES REMOVAL*********************************************
*set the two mechanical offsets
Offset1 := 70, Offset2 := 12
gen_rectangle1 (Rectangle, 0, 0, Height-Offset2, Width-Offset1)
gen_rectangle1 (Rectangle2, Offset2-1, Offset1-1, Height, Width)
*retrieve the grayvalues of the pixel inside the rectangles
reduce_domain (IntensityFlipped, Rectangle, ImageReduced1)
reduce_domain (IntensityNotFlipped, Rectangle2, ImageReduced2)
get_region_points (ImageReduced1, Rows1, Columns1)
get_region_points (ImageReduced2, Rows2, Columns2)
get_grayval (ImageReduced1, Rows1, Columns1, GrayvalNotFlipped)
get_grayval (ImageReduced2, Rows2, Columns2, GrayvalFlipped)
*pick the maximum between each pair of grayvalues to remove the dead zone
tuple_max2 (GrayvalNotFlipped, GrayvalFlipped, Max)
*and set the chosen values in the correct position
set_grayval (MatchedMax, Rows2, Columns2, Max)
```

Communication with the controller in C#

This code represents the main function to start the acquisition of the data from the controller to the PC application and it's part of a much bigger class that handles the communication.

```csharp
public static int Acquire(int deviceId, List<ushort> heightImage, List<ushort> luminanceImage,
↪  SetParam setParam, ref GetParam getParam)
{
    int yDataNum = setParam.YLineNum;
    int timeoutMs = setParam.TimeoutMs;
    int useExternalBatchStart = setParam.UseExternalBatchStart;
    ushort zUnit = 0;
    _heightBuf[deviceId] = new List<ushort>();
    _luminanceBuf[deviceId] = new List<ushort>();
    //Initialize
    int errCode = NativeMethods.LJX8IF_InitializeHighSpeedDataCommunicationSimpleArray(
    deviceId, ref _ethernetConfig[deviceId], (ushort)_highSpeedPortNo[deviceId],
    Callback, (uint)yDataNum, (uint)deviceId);
    Console.WriteLine(@"[@(Acquire) Initialize HighSpeed](0x{0:x})", errCode);
    //PreStart
    var startReq = new LJX8IF_HIGH_SPEED_PRE_START_REQUEST {bySendPosition = Convert.ToByte(2)};
    var profileInfo = new LJX8IF_PROFILE_INFO();
    errCode = NativeMethods.LJX8IF_PreStartHighSpeedDataCommunication(deviceId, ref startReq, ref
    ↪  profileInfo);
    Console.WriteLine(@"[@(Acquire) PreStart](0x{0:x})", errCode);
    //zUnit
    errCode = NativeMethods.LJX8IF_GetZUnitSimpleArray(deviceId, ref zUnit);
    if (errCode != 0 || zUnit == 0)
    {
        Console.WriteLine(@"Failed to acquire zUnit.");
               return errCode;
    }
    //Start HighSpeed
    _imageAvailable[deviceId] = 0;
    _lastImageSizeHeight[deviceId] = 0;
    errCode = NativeMethods.LJX8IF_StartHighSpeedDataCommunication(deviceId);
    Console.WriteLine(@"[@(Acquire) Start HighSpeed](0x{0:x})", errCode);
    //StartMeasure(Batch Start)
    if (useExternalBatchStart > 0) {}
    else
    {
        errCode = NativeMethods.LJX8IF_StartMeasure(deviceId);
        Console.WriteLine(@"[@(Acquire) Measure Start(Batch Start)](0x{0:x})", errCode);
    }
    // Acquire. Polling to confirm complete.
    // Or wait until a timeout occurs.
    Console.WriteLine(@" [@(Acquire) acquiring image...]");
    DateTime startDt = DateTime.UtcNow;
    TimeSpan ts;
    while (true)
        {
            ts = DateTime.UtcNow - startDt;
            if (timeoutMs < ts.TotalMilliseconds)
            {
                break;
            }
            if (_imageAvailable[deviceId] == 1) break;
        }
        if (_imageAvailable[deviceId] != 1)
        {
            Console.WriteLine(@" [@(Acquire) timeout]");
            //Stop HighSpeed
            errCode = NativeMethods.LJX8IF_StopHighSpeedDataCommunication(deviceId);
            Console.WriteLine(@"[@(Acquire) Stop HighSpeed](0x{0:x})", errCode);
            return (int)Rc.ErrTimeout;
        }
        Console.WriteLine(@" [@(Acquire) done]");
        //Stop HighSpeed
        errCode = NativeMethods.LJX8IF_StopHighSpeedDataCommunication(deviceId);
        Console.WriteLine(@"[@(Acquire) Stop HighSpeed](0x{0:x})", errCode);
        //-----------------------------------------------------------------------
        //  Organize parameters related to acquired image
        //-----------------------------------------------------------------------
        _getParam[deviceId].LuminanceEnabled = profileInfo.byLuminanceOutput;
        _getParam[deviceId].XPointNum = profileInfo.nProfileDataCount;
        _getParam[deviceId].YLinenumAcquired = _lastImageSizeHeight[deviceId];
        _getParam[deviceId].XPitchUm = profileInfo.lXPitch / 100.0f;
        _getParam[deviceId].YPitchUm = setParam.YPitchUm;
        _getParam[deviceId].ZPitchUm = zUnit / 100.0f;

        getParam = _getParam[deviceId];
        //-----------------------------------------------------------------------
        //  Copy internal buffer to user buffer
        //-----------------------------------------------------------------------
        heightImage.AddRange(_heightBuf[deviceId]);
        if (profileInfo.byLuminanceOutput > 0)
        {
            luminanceImage.AddRange(_luminanceBuf[deviceId]);
        }
        return (int)Rc.Ok;
}
```

Correction of the depth image and point cloud

```
**************************************DEPTH IMAGE ADJUSTMENT******************************************
*in the profilometer settings to decide the resolution
number_lines_profilometer := 1200
*this is due to interpolation
points_per_line := 4
*settings of the encoder, how many impulses it outputs over a full spin
impulses_per_spin := 10000
*how much space is covered over a full spin of the encoder [mm]
space_per_spin := 190
*how many impulses it takes to get a line (settings of the profilometer)
impulses_per_line := 10
*how many degrees are spanned over an impulse
degrees_per_impulse := 360.0/impulses_per_spin
*how much space in [mm] is covered over an impulse
space_per_impulse := (space_per_spin x degrees_per_impulse)/360.0
*how much space in [mm] is covered to take a line
space_per_line := space_per_impulse x impulses_per_line
*full space spanned over the all image in the axis of movements of the
*profilometer [mm]
space_spanned := space_per_line x number_lines_profilometer
*resolution over the axis of movement of the profilometer [mm]
ResolutionY := (space_per_line)/points_per_line
*resolution over axis perpendicular to the conveyor belt [micro m]
ResolutionZ := 3.24
read_image (Matchedmax, 'C:/Users/39348/matchedmax_heights.png')
rotate_image (Matchedmax, Image, 90, 'constant')
*******************************PROCEDURE TO ANTI-TRANSFORM THE IMAGE*********************************
*[degrees]
angle_velocity := 45
*the offset in pixel is referred to the resolution at the belt level
belt_level_intensity := 0
*this is for the lj-x8300 profilometer by Keyence
heights_difference := 53
FOV := [134,150,160]
calculate_resx (belt_level_intensity, heights_difference, FOV, ResolutionZ, ResolutionX_belt)
*computation of offset in [mm] (this is constant at each height)
offset_mm := ResolutionY x cos(rad(angle_velocity))
*computation of offset in pixels (this changes at various heights but here
*i consider only the belt level)
offset_pixel := offset_mm / ResolutionX_belt
*total offset for the last row
total_offset_pixels := offset_pixel x number_lines_profilometer x points_per_line
gen_image_const(OutputImage, 'byte', number_lines_profilometer x points_per_line,
3200+round(total_offset_pixels))
get_image_size (OutputImage, Width, Height)
Rows := []
```

```
Columns := []
ColumnGrayval := []
for I := 0 to Width-1 by 1
    gen_rectangle1 (Rectangle, 0, I, Height-1, I)
    reduce_domain (Image, Rectangle, ImageReduced)
    get_region_points (ImageReduced, Rows1, Columns1)
    get_grayval (ImageReduced, Rows1, Columns1, ColumnGrayval1)
    Rows := [Rows, Rows1 + round(offset_pixel x I)]
    Columns := [Columns, Columns1]
    ColumnGrayval := [ColumnGrayval, ColumnGrayval1]
endfor
set_grayval (OutputImage, Rows, Columns, ColumnGrayval)
write_image(Image, 'png', 0, 'not_transformed')
write_image (OutputImage, 'png', 0, 'transformed')
***********************************END DEPTH IMAGE ADJUSTMENT*****************************************
```

```
**************************************POINT CLOUD ADJUSTMENT******************************************
read_image (Z_NotTransformed, 'C:/Users/not_transformed_heights.png')
*this is for the lj-x8300 profilometer by Keyence
ResolutionX := 0.05
ResolutionY := 0.0475
ResolutionZ := 3.24
*heights_difference := 53
*FOV := [134,150,160]
******************************CREATION OF THE POINT CLOUD NOT TRANSFORMED****************************
*convert the type of the depth image
convert_image_type (Z_NotTransformed, Z_NotTransformed, 'real')
*create new image of z coordinates using the ResolutionZ
get_image_size (Z_NotTransformed, Width, Height)
gen_image_const (ConstImage, 'real', Width, Height)
paint_region (ConstImage, ConstImage, ConstImage, 1, 'fill')
div_image (Z_NotTransformed, ConstImage, Z_NotTransformed, 0.001, 0)
div_image (Z_NotTransformed, ConstImage, Z_NotTransformed, ResolutionZ, 0)
*generate the two gradient images for the x and y coordinates
gen_image_surface_first_order (X_NotTransformed, 'real', 0, ResolutionX, 0, 0, 0, Width, Height)
gen_image_surface_first_order (Y_NotTransformed, 'real', ResolutionY, 0, 0,0, 0, Width, Height)
*create cloud point
xyz_to_object_model_3d (X_NotTransformed, Y_NotTransformed,
Z_NotTransformed, CloudPoint_NotTransformed)
*save not transformed point cloud
write_object_model_3d (CloudPoint_NotTransformed, 'ply',
'NotTransformed_PC', [], [])
*visualize obtained point cloud
create_pose (0, 0, 0, 0, 0, 0, 'Rp+T', 'gba', 'point', Pose)
dev_open_window (0, 0, 500, 500, 'black', WindowHandle)
visualize_object_model_3d(WindowHandle,
CloudPoint_NotTransformed, [], [], [], [], [], [], [], PoseOut)
*select point only above the belt
select_points_object_model_3d (CloudPoint_NotTransformed, 'point_coord_z',
0.5, 80, CloudPoint_NotTransformed_Thresholded)
*save point cloud with removed belt
write_object_model_3d (CloudPoint_NotTransformed_Thresholded, 'ply',
'NotTransformed_withoutBelt_PC', [], [])
*visualize point cloud without belt level points
create_pose (0, 0, 0, 0, 0, 0, 'Rp+T', 'gba', 'point', Pose)
dev_open_window (0, 0, 500, 500, 'black', WindowHandle)
visualize_object_model_3d(WindowHandle,
CloudPoint_NotTransformed_Thresholded, [], [], [], [], [], [], [], PoseOut)
*****************************CREATION OF THE POINT CLOUD  TRANSFORMED*********************************
offset_mm :=  0.0335876
gen_image_surface_first_order (X_Transformed, 'real', 0, ResolutionX, 0, 0, 0, Width, Height)
gen_image_surface_first_order (Y_Transformed, 'real', ResolutionY,
offset_mm, 0, 0, 0, Width, Height)
*create cloud point
xyz_to_object_model_3d (X_Transformed, Y_Transformed, Z_NotTransformed,
CloudPoint_Transformed)
*select point only above the belt
select_points_object_model_3d (CloudPoint_Transformed, 'point_coord_z', 0.5,
80, CloudPoint_Transformed_Thresholded)
*save point cloud transformed with removed belt
write_object_model_3d (CloudPoint_Transformed_Thresholded, 'ply',
'Transformed_withoutBelt_PC', [], [])
*visualize the transformed point cloud
create_pose (0, 0, 0, 0, 0, 0, 'Rp+T', 'gba', 'point', Pose)
dev_open_window (0, 0, 500, 500, 'black', WindowHandle)
visualize_object_model_3d(WindowHandle, CloudPoint_Transformed_Thresholded,
[], [], [], [], [], [], [], PoseOut)
*triangulate the point cloud
triangulate_object_model_3d (CloudPoint_Transformed_Thresholded, 'greedy',
[], [], TriangulatedObjectModel3D, Information)
```

Temporal Stitching

```
read_image (Z_NotTransformed1, 'C:/Users/not_transformed_heights_1.png')
read_image (Z_NotTransformed2, 'C:/Users/not_transformed_heights_2.png')
*this is for the lj-x8300 profilometer by Keyence
ResolutionX := 0.05
ResolutionY := 0.0475
ResolutionZ := 3.24
**************************CREATION OF THE POINT CLOUD  STICHTED TRANSFORMED*************************
*this is calculated according to the procedure in the test case
offset_mm :=  0.0335876
*convert the type of the depth imageS
```

```
convert_image_type (Z_NotTransformed1, Z_NotTransformed1, 'real')
convert_image_type (Z_NotTransformed2, Z_NotTransformed2, 'real')
*create new image of z coordinates using the ResolutionZ for first image
get_image_size (Z_NotTransformed1, Width, Height)
gen_image_const (ConstImage, 'real', Width, Height)
paint_region (ConstImage, ConstImage, ConstImage, 1, 'fill')
div_image (Z_NotTransformed1, ConstImage, Z_NotTransformed1, 0.001, 0)
div_image (Z_NotTransformed1, ConstImage, Z_NotTransformed1, ResolutionZ, 0)
*create new image of z coordinates using the ResolutionZ for second image
get_image_size (Z_NotTransformed2, Width, Height)
gen_image_const (ConstImage, 'real', Width, Height)
paint_region (ConstImage, ConstImage, ConstImage, 1, 'fill')
div_image (Z_NotTransformed2, ConstImage, Z_NotTransformed2, 0.001, 0)
div_image (Z_NotTransformed2, ConstImage, Z_NotTransformed2, ResolutionZ, 0)
*generate the two gradient images for the x and y coordinates(first image)
gen_image_surface_first_order (X_Transformed1, 'real', 0, ResolutionX, 0,
0, 0, Width, Height)
gen_image_surface_first_order (Y_Transformed1, 'real', ResolutionY,
offset_mm, 0, 0, 0, Width, Height)
*generate the two gradient images for the x and y coordinates(second image)
offset_subsequent := 115
gen_image_surface_first_order (X_Transformed2, 'real', 0, ResolutionX, 0,
0, 0, Width, Height)
gen_image_surface_first_order (Y_Transformed2, 'real', ResolutionY,
offset_mm, offset_subsequent, 0, 0, Width, Height)
**generate X,Y,Z images of the subsequent point cloud
concat_obj (X_Transformed1, X_Transformed2, X_concat)
concat_obj (Y_Transformed1, Y_Transformed2, Y_concat)
concat_obj (Z_NotTransformed1, Z_NotTransformed2, Z_concat)
tile_images_offset (X_concat, X, [0, Height], [0, 0], [0,0], [0,0],
[Height - 1, Height - 1], [Width -1, Width - 1], Width, Height*2)
tile_images_offset (Y_concat, Y, [0, Height], [0, 0], [0,0], [0,0],
[Height - 1, Height - 1], [Width -1, Width - 1], Width, Height*2)
tile_images_offset (Z_concat, Z, [0, Height], [0, 0], [0,0], [0,0],
[Height - 1, Height - 1], [Width -1, Width - 1], Width, Height*2)
xyz_to_object_model_3d (X, Y, Z, CloudPoint_Transformed)
*select point only above the belt
belt_level_mm := 3500*0.001*ResolutionZ
select_points_object_model_3d (CloudPoint_Transformed, 'point_coord_z',
belt_level_mm, belt_level_mm + 8, CloudPoint_Transformed_Thresholded)
*triangulate the point cloud
triangulate_object_model_3d (CloudPoint_Transformed_Thresholded, 'greedy',
[], [], TriangulatedObjectModel3D, Information)
```

3D Matching and Pose estimation

```
read_object_model_3d ('C:/Tesi/mesh.ply', 'mm', [], [], ObjectModel3D, Status)
*segment the 3d object
connection_object_model_3d (ObjectModel3D, 'distance_3d', 0.002,ObjectModel3DConnected)
*cumpute the surface normals needed for the matching
surface_normals_object_model_3d (ObjectModel3DConnected[0], 'mls', [], [], ObjectModel3DNormals)
* Create the surface model from the reference view
create_surface_model (ObjectModel3DNormals, 0.03, [], [], SFM)
* compute keypoint, match and get the results
find_surface_model (SFM, ObjectModel3D, 0.05, 0.3, 0.2, 'true', 'num_matches', 10, Pose,
Score, SurfaceMatchingResultID)
ObjectModel3DResult := []
    for Index2 := 0 to |Score| - 1 by 1
        if (Score[Index2] < 0.35)
            continue
        endif
        CPose := Pose[Index2 * 7:Index2 * 7 + 6]
        rigid_trans_object_model_3d (ObjectModel3DNormals, CPose, ObjectModel3DRigidTrans)
        ObjectModel3DResult := [ObjectModel3DResult,ObjectModel3DRigidTrans]
    endfor
* Visualize matching scene and key points
Message := 'Original scene points (white)'
Message[1] := 'Sampled scene points (cyan)'
Message[2] := 'Key points (yellow)'
get_surface_matching_result (SurfaceMatchingResultID, 'sampled_scene', [], SampledScene)
get_surface_matching_result (SurfaceMatchingResultID, 'key_points', [], KeyPoints)
dev_clear_window ()
dev_open_window (0, 0, 1024, 1024, 'black', WindowHandle)
visualize_object_model_3d (WindowHandle, [ObjectModel3D,SampledScene,KeyPoints], [], [],
['color_' + [0, 1, 2],'point_size_' + [0, 1, 2]], ['gray', 'cyan', 'yellow', 1.0, 3.0,
5.0], Message, [], Instructions, PoseOut)
dump_window_image (Result, WindowHandle)
* Visualize result(s)
Message := 'Scene: '
Message[1] := 'Found ' + |ObjectModel3DResult| + ' object(s) '
ScoreString := sum(Score$'.2f' + ' / ')
Message[2] := 'Score(s): ' + ScoreString{0:strlen(ScoreString) - 4}
NumResult := |ObjectModel3DResult|
tuple_gen_const (NumResult, 'green', Colors)
tuple_gen_const (NumResult, 'circle', Shapes)
tuple_gen_const (NumResult, 3, Radii)
Indices := [1:NumResult]
dev_clear_window ()
dev_open_window (0, 0, 1024, 1024, 'black', WindowHandle)
visualize_object_model_3d (WindowHandle,[ObjectModel3D,ObjectModel3DResult],[],
PoseOut,['color_'+[0,Indices],'point_size_0'],['gray',Colors,1.0],Message,[],Instructions, PoseOut)
```

BIBLIOGRAPHY

Ahmed, Eman, Alexandre Saint, Abd El Rahman Shabayek, Kseniya Cherenkova, Rig Das, Gleb Gusev, Djamila Aouada, and Bjorn Ottersten

2019 *A survey on Deep Learning Advances on Different 3D Data Representations*, arXiv: 1808.01462 [cs.CV], https://arxiv.org/abs/1808.01462.

Akca, Devrim

2007 “Matching of 3D surfaces and their intensities”, *ISPRS Journal of Photogrammetry and Remote Sensing*, 62 (June 2007), pp. 112-121, DOI: 10.1016/j.isprsjprs.2006.06.001.

Charles, R. Qi, Hao Su, Mo Kaichun, and Leonidas J. Guibas

2017 “PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation”, in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 77-85, DOI: 10.1109/CVPR.2017.16.

Drost, Bertram, Markus Ulrich, Nassir Navab, and Slobodan Ilic

2010 “Model globally, match locally: Efficient and robust 3D object recognition”, in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 998-1005, DOI: 10.1109/CVPR.2010.5540108.

Gezawa, Abubakar Sulaiman, Yan Zhang, Qicong Wang, and Lei Yunqi

2020 “A Review on Deep Learning Approaches for 3D Data Representations in Retrieval and Classifications”, *IEEE Access*, 8, pp. 57566-57593, DOI: 10.1109/ACCESS.2020.2982196.

Harris, Christopher G. and M. J. Stephens

1988 “A Combined Corner and Edge Detector”, in *Alvey Vision Conference*, https://api.semanticscholar.org/CorpusID:1694378.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun

2016 “Deep Residual Learning for Image Recognition”, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770-778, DOI: 10.1109/CVPR.2016.90.

Johnson, A.E. and M. Hebert

1999 “Using spin images for efficient object recognition in cluttered 3D scenes”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 21, 5, pp. 433-449, DOI: 10.1109/34.765655.

Keyence

n.d. *LJ-X8000 Series User’s Manual (3D mode)*, English, version Version 2.0, Keyence, 810 pp., https://www.keyence.eu/support/user/measure/lj-x8000/manual/.

Maturana, Daniel and Sebastian Scherer

2015 "VoxNet: A 3D Convolutional Neural Network for real-time object recognition", *IEEE*, DOI: 10.1109/IROS.2015.7353481.

MVTec

n.d.(a) *HDevelop User's Guide*, English, version Version 24.05.0.0, MVTec Software GmbH, 344 pp., https://www.mvtec.com/fileadmin/Redaktion/mvtec.com/products/halcon/documentation/manuals/hdevelop_users_guide.pdf.

n.d.(b) *Quick Guide*, English, version Version 24.05.0.0, MVTec Software GmbH, 19 pp., https://www.mvtec.com/fileadmin/Redaktion/mvtec.com/products/halcon/documentation/manuals/quick_guide.pdf.

n.d.(c) *Solution Guide III-3C 3D Vision*, English, version Version 24.05.0.0, MVTec Software GmbH, 238 pp., https://www.mvtec.com/fileadmin/Redaktion/mvtec.com/products/halcon/documentation/solution_guide/solution_guide_iii_c_3d_vision.pdf.

Ngo, Trung-Thanh, Asatilla Abdukhakimov, and Dong-Seong Kim

2019 "Long-Range Wireless Tethering Selfie Camera System Using Wireless Sensor Networks", *IEEE Access*, PP (Aug. 2019), pp. 1-1, DOI: 10.1109/ACCESS.2019.2933402.

Pears, Nicholas Edwin, Yonghuai Liu, and Peter Bunting

2012 (eds.), *3D Imaging, Analysis and Applications*, Springer, ISBN: 978-1-4471-4062-7.

Sedaghat, Nima, Mohammadreza Zolfaghari, Ehsan Amiri, and Thomas Brox

2017 *Orientation-boosted Voxel Nets for 3D Object Recognition*, arXiv: 1604.03351 [cs.CV], https://arxiv.org/abs/1604.03351.

Sipiran, Ivan and Benjamin Bustos

2011 "Harris 3D: A robust extension of the Harris operator for interest point detection on 3D meshes", *The Visual Computer*, 27 (Nov. 2011), pp. 963-976, DOI: 10.1007/s00371-011-0610-y.

Soydaner, Derya

2020 "A Comparison of Optimization Algorithms for Deep Learning", *International Journal of Pattern Recognition and Artificial Intelligence*, 34, 13 (Apr. 2020), p. 2052013, ISSN: 1793-6381, DOI: 10.1142/s0218001420520138, http://dx.doi.org/10.1142/S0218001420520138.

Su, Hang, Subhransu Maji, Evangelos Kalogerakis, and Erik Learned-Miller

2015 "Multi-view Convolutional Neural Networks for 3D Shape Recognition", in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 945-953, DOI: 10.1109/ICCV.2015.114.

Su, Peter and Robert L. Scot Drysdale

1997 "A comparison of sequential Delaunay triangulation algorithms", *Computational Geometry*, 7, 5, 11th ACM Symposium on Computational Geometry, pp. 361-385, ISSN: 0925-7721, DOI: https://doi.org/10.1016/S0925-7721(96)00025-9, https://www.sciencedirect.com/science/article/pii/S0925772196000259.

Thrun, S.

2003 "Learning occupancy grid maps with forward sensor models", *Auton. Robots*, DOI: https://doi.org/10.1023/A:1025584807625.

Wang, Yutao and Hsi-Yung Feng

2014 "Modeling outlier formation in scanning reflective surfaces using a laser stripe scanner", *Measurement*, 57, pp. 108-121, ISSN: 0263-2241, DOI: https://doi.org/10.1016/j.measurement.2014.08.010, https://www.sciencedirect.com/science/article/pii/S026322411400325X.

Xu, Guoping, Xiaxia Wang, Xinglong Wu, Xuesong Leng, and Yongchao Xu

2024 *Development of Skip Connection in Deep Neural Networks for Computer Vision and Medical Image Analysis: A Survey*, arXiv: 2405.01725 [eess.IV], https://arxiv.org/abs/2405.01725.

Yang, ZX., L Tang, Zhang, and K. et al.

2018 "Multi-View CNN Feature Aggregation with ELM Auto-Encoder for 3D Shape Recognition." *Cognitive Computation*, DOI: https://doi.org/10.1007/s12559-018-9598-1.

Zhang, Yu-Jin

2023 (ed.), *3-D Computer Vision. Principles, Algorithms and Applications*, Springer Singapore, ISBN: 978-981-19-7580-6.

Zhang, Song

2018 "High-speed 3D shape measurement with structured light methods: A review", *Optics and Lasers in Engineering*, 106, pp. 119-131, ISSN: 0143-8166, DOI: https://doi.org/10.1016/j.optlaseng.2018.02.017, https://www.sciencedirect.com/science/article/pii/S0143816617313246.

Zhang, Zhengyou

2000 "A Flexible New Technique for Camera Calibration", *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22 (Dec. 2000), pp. 1330-1334, DOI: 10.1109/34.888718.