NTNU – Trondheim
Norwegian University of
Science and Technology

Robotic Assembly Using a 3D Camera

Martin Morland
Geir Ole Tysse

Subsea Technology
Submission date: June 2015
Supervisor: Olav Egeland, IPK

Norwegian University of Science and Technology
Department of Production and Quality Engineering

Robotic assembly using a 3D camera

Martin Morland & Geir Ole Tysse

Spring 2015

PROJECT / MASTER THESIS
Department of Production and Quality Engineering
Norwegian University of Science and Technology

Supervisor 1: Professor Olav Egeland

NTNU

Fakultet for ingeniørvitenskap og teknologi
Institutt for produksjons- og kvalitetsteknikk

Vår dato	Vår referanse
22.01.2015	OEG

MASTEROPPGAVE VÅREN 2015

Martin Morland og Geir Ole Tysse

Tittel: Robotisert montasje ved bruk av 3D-kamera

Tittel (engelsk): Robotic Assembly Using a 3D Camera

Oppgavens tekst:
Ved robotisert montasje i småserieproduksjon kan det være hensiktsmessig å identifisere delenes posisjon og orientering med robotsyn for å sikre korrekt griping og sammensetning. I industriell produksjon med deler i metall vil det være problemer med refleksjon av lys fra metalloverflater, og dette gjør det vanskelig på benytte *key-point descriptors* som SIFT og SURF med vanlige 2D-kamera i stereoarrangement. Videre er stereosyn relativt komplisert å implementere og krever gjenre spesialtilpasning til den konkrete anvendelsen. Det er derfor interessant å studere bruk av 3D-kamera, som er enklere å bruke, og som gir punktskyer som kan være mer robuste enn stereodata. I denne oppgaven skal dette gjøres for sammensetning av bildeler i et eksperiment i instituttets robotlaboratorium. Det skal benyttes et enkelt Asus 3D-kamera.

1. Presenter geometrisk algebra og gi et enkelt eksempel på hvordan dette kan brukes i oppgaven.
2. Beskriv kalibrering av interne og eksterne parametere for et 3D-kamera.
3. Beskriv hvordan et 3D-kamera kan benyttes til å bestemme posisjon og orientering av et objekt.
4. Hvis hvordan orientering om lengdeaksen for en sylindrisk del kan bestemmes med et vanlig 2D-kamera.
5. Lag en demonstrasjon av robotisert montasje ved bruk av 3D-kamera.

Oppgave utlevert: 2015-01-22
Innlevering: 2015-06-10
Utført ved Institutt for produksjons- og kvalitetsteknikk

Trondheim, 2015-01-22

Olav Egeland

Professor Olav Egeland
Faglærer

Preface

This is the concluding Master's thesis in the study program Subsea Technology at NTNU. The work was carried out between January and June 2015. The reader should have some elementary knowledge about robotic systems and computer vision.

Trondheim, 2015-06-10

Martin Morland and Geir Ole Tysse

Acknowledgment

We would like to thank our supervisor Olav Egeland for the lectures given in geometric algebra. We would also like to thank Ph.D. candidate Adam Leon Kleppe for getting us started using Ubuntu, ROS and the robots in this project.

O.N.

Summary and Conclusions

In this thesis we explain how two parts from the automotive industry can be assembled using an RGB-D camera providing colour (RGB) and depth (D) as a combined RGB-D image, and two KUKA Agilus manipulators with six revolute joints. The RGB-D camera is presented and calibrated, and AR-tags were used to efficiently transform the coordinate frame of the RGB-D camera to a table with the parts. In relation to this, it is shown how geometric algebra can be used to make an orthonormal coordinate frame on the table. Several different methods are presented and evaluated for obtaining the position of the parts, based on data from the RGB-D camera. This includes, a C++ program for segmentation of the scene based on the orientation of the surface normals in the point cloud, and a program based on the geometry of the parts. A method for determining the orientation of the parts, using SIFT (Scale Invariant Feature Transform) and homography based on data from a 2D camera is proposed. We present ROS (Robot Operating System) and how it interacts with the RGB-D camera, 2D camera and the robots. In addition, the forward kinematics of the robots are explained and implemented in ROS.

The experiment was performed as follows: An RGB-D camera positioned at a random place with the AR-tags in the field of view. An algorithm automatically determines the pose of the RGB-D camera with respect to a known coordinate frame. The RGB-D camera was used to measure the coordinates of the parts with respect to the known coordinate frame. A robot with a 2D camera connected to the end-effector, moved to a pose where the 2D camera was vertically above the parts, one at a time. Data from the 2D camera were used in an algorithm for estimating the orientation of the parts. A robot with a gripper connected to the end-effector used the position and orientation of the parts to assemble them.

The results obtained were adequate to perform a simplified assembly task for the parts which require an accuracy of 4 mm. Two demonstration videos have been made. One video demonstrates the estimation of the orientation of the parts. The other video demonstrates how the position of the parts are estimated and how the robot performs the simplified assembly task.

Sammendrag

Denne rapporten forklarer hvordan to deler fra bilindustrien kan monteres sammen ved bruk av et RGB-D-kamera, som kombinerer farge (RGB) og dybde (D) sammen til et RGB-D-bilde, og to KUKA Agilus manipulatorer med seks roterende ledd. RGB-D-kameraet blir presentert og kalibrert, og AR-merker ble brukt til å effektivt transformere koordinatsystemet til RGB-D-kameraet til et bord med delene på. Med tanke på dette presenteres det også hvordan geometrisk algebra kan benyttes for å lage en ortonormal koordinatsystem på bordet. Flere forskjellige metoder for å innhente posisjonen til delene presenteres og evalueres basert på data fra RGB-D-kameraet. Blant annet, et C++ program som segmenterer scenen basert på orienteringen til overflatenormalene i punktskyen, og et program som baserer seg på delenes geometri. En metode for å bestemme orienteringen til delene, ved å bruke SIFT og homografi basert på data fra 2D-kameraet blir foreslått. Vi presenterer ROS (Robot Operating System) og måten det samhandler med RGB-D-kameraet, 2D-kameraet og robotene. I tillegg, blir robotdirektekinematikken forklart og implementert i ROS.

Eksperimentet ble utført på følgende måte: Et RGB-D-kamera plasseres et tilfeldig sted med AR-merkene på bordet i synsfeltet. En algoritme beregner automatisk posituren til RGB-D-kameraet med hensyn på et kjent koordinatsystem. RGB-D-kameraet ble brukt til å måle koordinatene til delene med i forhold til det kjente koordinatsystemet. En robot med et 2D-kamera festet til end-effektoren flytter 2D kameraet til en positur som er vinkelrett over en av delene, etter tur. Data fra 2D-kameraet benyttes i en algoritme for å estimere orienteringen til delene. En robot med en "griper" påmontert end-effektoren, benytter posisjonen og orienteringen til delene for å montere de sammen.

Resultatene som ble oppnådd var tilstrekkelig for å utføre en forenklet montering av delene, som krevde en nøyaktighet på 4 mm. To demonstrasjonsvideoer har blitt laget. En video demonstrerer hvordan orienteringen bestemmes. Den andre viser hvordan posisjonen til delene blir funnet og hvordan den forenklede monteringen utføres.

Contents

Chapter 1

Introduction

1.1 Background

Applications of robots in the industry have widened to ameliorate the complexity of factory automation systems. An increasing interest in regard to robotic material handling have been to reduce the cost of installation of a factory automation system while improving productivity [7]. When using grasping robotic systems to assemble parts, it is convenient to identify the position and orientation of the parts. This task can be solved by obtaining depth images of the operational area of the robot. Conventional 2D cameras in stereo arrangement are able to capture depth information in images, but it is relatively complicated to implement and may need special adjustment for the specific task. Low-cost consumer depth cameras have risen to widespread prevalence across many areas of 3D computer vision [17]. Recent RGB-D cameras (3D cameras), such as the Microsoft Kinect or Asus Xtion PRO LIVE, acquire both visual information (RGB) like regular camera systems, along with per-pixel depth information (D) at high frame rates. In terms of real-time performance and general depth accuracy, RGB-D cameras outperform common stereo vision approaches on texture-less object surfaces [17].

Other research groups have developed systems for detecting and grasping objects using a depth camera. A prominent example is found in a paper by Kim et al. [7], where both 3D point cloud and CAD data are segmented to simple primitives as plane, cylinder and cone for object recognition, pose estimation and object grasping. Stückler et al. [27] describe efficient methods for tackling everyday mobile manipulation tasks that require object pick-up. They have developed scene segmentation methods that process depth images in real-time at high frame rates for 3D perception of objects on planar surfaces. Lin and Setiawan [10] uses Scale Invariant Feature Transform (SIFT) for recognition and Support Vector Machine (SVM) to determine the orientation of an object, using a stereo camera.

In this thesis we will study the use of RGB-D camera in robotic assembly of automotive parts, more particularly a piston and a cylinder as shown in Figure 1.1. The intrinsic parameters of the RGB-D camera is calibrated and an efficient method to transform the coordinate frame of the RGB-D camera on to a table using AR-tags proposed. For this purpose, geometric algebra can be used to easily create an orthogonal coordinate frame. Different approaches for measuring the coordinates of the parts from an RGB-D camera are explained. Analysis of accuracy and repeatability related to these measurements are performed. This includes a scene segmentation method based on the orientation of the surface normals. We propose a method for determining the orientation around the length axis of the objects using a 2D camera, SIFT and homography. The kinematics for assembling the parts with a robot is explained. ROS (Robot Operating System) and how it is applied to solve the kinematics is presented.

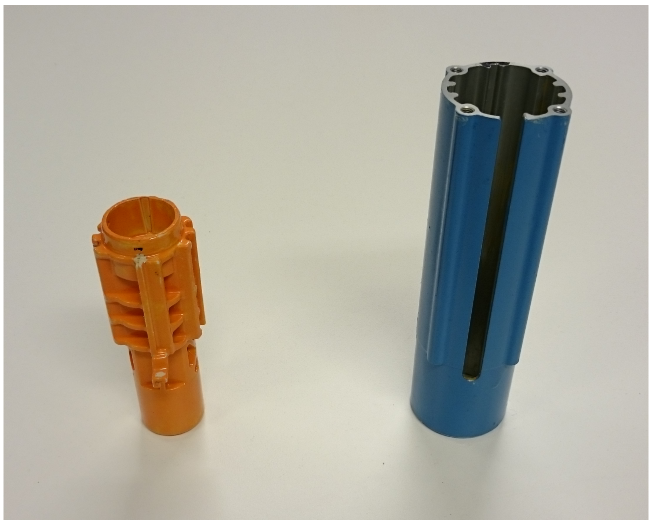


Figure 1.1: The the piston (orange) and cylinder (blue).

The practical work has been performed in the lab at the Department of Production and Quality Engineering, NTNU. Two KUKA Agilus robot cells with six revolute joints and an RGB-D camera (Asus Xtion PRO LIVE) were used in the process of assembling the piston and the cylinder.

1.2 Structure of the Report

The rest of the report is organized as follows:

- Chapter 2 gives an introduction to geometric algebra.
- Chapter 3 contains information about the RGB-D camera and calibration.
- Chapter 4 contains theoretical background information relevant for different aspects of computer vision used in the thesis.
- Chapter 5 contains experiments conducted and analysis of the results.
- Chapter 6 addresses the forward kinematics and the homogeneous transformations between robot and camera.
- Chapter 7 presents the implementation in ROS.
- Chapter 8 presents the concluding remarks in form of discussion, conclusion and recommendations for further work.

Chapter 2

Geometric Algebra

This chapter will give an introduction to geometric algebra.

Geometric algebra recognise lines, areas, volumes and hyper-volumes as structures with *magnitude* and *orientation*. The foundation of geometric algebra is the *geometric product* [30].

2.0.1 Geometric product

Two vectors a and b in a 2D Euclidean space $\mathbb{R}^2$, can be described as

$$a = a_1e_1 + a_2e_2 \quad , \quad b = b_1e_1 + b_2e_2$$

where a_i and b_i are components with respect to the unit basis vectors e_1 and e_2.

The geometric product of a and b is

$$ab = \underbrace{a \cdot b}_{\textit{inner product}} + \underbrace{a \wedge b}_{\textit{outer product}} \tag{2.1}$$

The inner product of equation (2.1) is

$$\begin{aligned} a \cdot b &= (a_1e_1 + a_2e_2) \cdot (b_1e_1 + b_2e_2) \\ &= a_1b_1e_1 \cdot e_1 + a_1b_2e_1 \cdot e_2 + a_2b_1e_1 \cdot e_2 + a_2b_2e_2 \cdot e_2 \\ &= a_1b_1 + a_2b_2 \end{aligned} \tag{2.2}$$

which gives a scalar. For vectors $a, b \in \mathbb{R}^n$, their inner product is just the same as their scalar product, where

$$e_i \cdot e_j = 0 \quad , \quad e_i \cdot e_i = 1$$

The outer product of equation (2.1) is

$$\begin{aligned} a \wedge b &= (a_1e_1 + a_2e_2) \wedge (b_1e_1 + b_2e_2) \\ &= a_1b_1e_1 \wedge e_1 + a_1b_2e_1 \wedge e_2 + a_2b_1e_2 \wedge e_1 + a_2b_2e_2 \wedge e_2 \\ &= (a_1b_2 - a_2b_1)e_1 \wedge e_2 \end{aligned} \tag{2.3}$$

where

$$e_i \wedge e_i = 0 \quad , \quad e_i \wedge e_j = -e_j \wedge e_i$$

Equation (2.3) results in a scalar term $a_1b_2 - a_2b_1$ that defines a signed area, whilst the unit *bivector* term $e_1 \wedge e_2$ identifies a plane associated with the area. The orientation of the plane is defined by its bivector components [30], [28]. The bivector term $e_1 \wedge e_2$ means that orientation of the surface is anticlockwise, as depicted by the directed circle in figure (2.1).

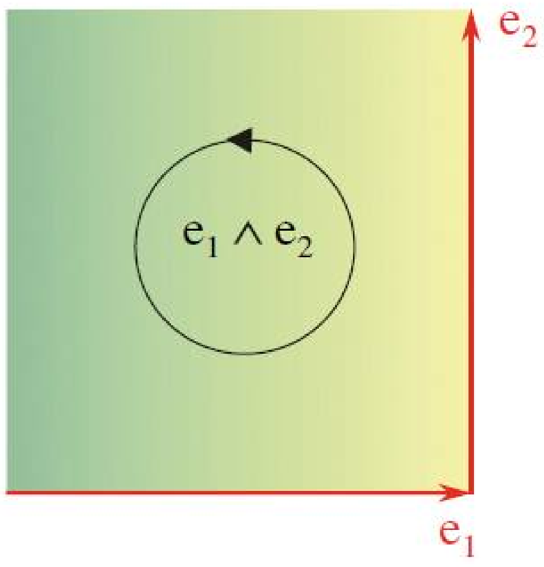


Figure 2.1: Orientation of $e_1 \wedge e_2$ [30].

For two identical basis vectors, the geometric product is

$$e_ie_i = e_i \cdot e_i + e_i \wedge e_i = e_i^2 = 1$$

and the product of two orthogonal basis vectors is

$$e_ie_j = e_i \cdot e_j + e_i \wedge e_j = e_i \wedge e_j$$

The geometric product of two vectors a and b

$$a = a_1e_1 + a_2e_2 + a_3e_3 \quad , \quad b = b_1e_1 + b_2e_2 + b_3e_3$$

in a 3D Euclidean space $\mathbb{R}^3$ with basis vectors e_1, e_2 and e_3, is

$$ab = \underbrace{a \cdot b}_{\text{inner product}} + \underbrace{a \wedge b}_{\text{outer product}} \tag{2.4}$$

where

$$\begin{aligned} a \cdot b &= (a_1e_1 + a_2e_2 + a_3e_3) \cdot (b_1e_1 + b_2e_2 + b_3e_3) \\ a \cdot b &= a_1b_1 + a_2b_2 + a_3b_3 \end{aligned} \tag{2.5}$$

and

$$\begin{aligned} a \wedge b &= (a_1e_1 + a_2e_2 + a_3e_3) \wedge (b_1e_1 + b_2e_2 + b_3e_3) \\ a \wedge b &= (a_1b_2 - a_2b_1)e_1 \wedge e_2 + (a_2b_3 - a_3b_2)e_2 \wedge e_3 + (a_3b_1 - a_1b_3)e_3 \wedge e_1 \\ a \wedge b &= (a_1b_2 - a_2b_1)e_1e_2 + (a_2b_3 - a_3b_2)e_2e_3 + (a_3b_1 - a_1b_3)e_3e_1 \end{aligned} \tag{2.6}$$

The outer product of k linearly independent vectors is called a k-blade with the following order:

$$\begin{aligned} \text{scalar} &= 0 - \text{blade} \\ \text{vector} &= 1 - \text{blade} \\ \text{bivector} &= 2 - \text{blade} \\ \text{trivector} &= 3 - \text{blade} \\ \text{quadvector} &= 4 - \text{blade} \end{aligned}$$

where k identifies the grade of a blade [30]. Table 2.1 shows the k-blades in a 3D Euclidean space $\mathbb{R}^3$. In such graded algebras it is tradition to call the highest grade element a *pseudoscalar*

k	basis k-blades	total
0 (scalar)	$\{1\}$	1
1 (vector)	$\{e_1, e_2, e_3\}$	3
2 (bivector)	$\{e_{12}, e_{23}, e_{31}\}$	3
3 (trivector)	$\{e_{123}\}$	1

Table 2.1: k-blades in $\mathbb{R}^3$.

I [30]. In $\mathbb{R}^3$, the pseudoscalar is the trivector $I = e_1e_2e_3 = e_{123}$.

2.0.2 The Outer Product Null Space

The outer product null space $\mathbb{NO}(a)$ of a vector a in Euclidean space $\mathbb{R}^3$ is given by the set of vectors $x \in \mathbb{R}^3$ that satisfy $x \wedge a = 0$. That is,

$$\mathbb{NO}(a) = \{x \in \mathbb{R}^3 : x \wedge a = 0\} = \{x = \alpha a : \alpha \in \mathbb{R}\}$$

The outer product null space of a vector a is a line through the origin with direction of a, because $x \wedge a$ is zero if x is linearly dependent on a [28] [3].

Given a $2-$*blade* $a \wedge b$, where a and b are two vectors in Euclidean space $\mathbb{R}^3$. The outer product null space $\mathbb{NO}(a \wedge b)$ is

$$\mathbb{NO}(a \wedge b) = \{x \in \mathbb{R}^3 : x \wedge (a \wedge b) = 0\} = \{x = \alpha a + \beta b : \alpha, \beta \in \mathbb{R}\}$$

the set of vectors x satisfying $x \wedge (a \wedge b) = 0$. This is a set of vectors x that are in the plane defined by the bivector $a \wedge b$.

In Figure 2.2 the green plane is $\mathbb{NO}(a \wedge b)$, where a and b are two vectors in $\mathbb{R}^3$. The plane goes through origin of the Euclidean space $\mathbb{R}^3$.

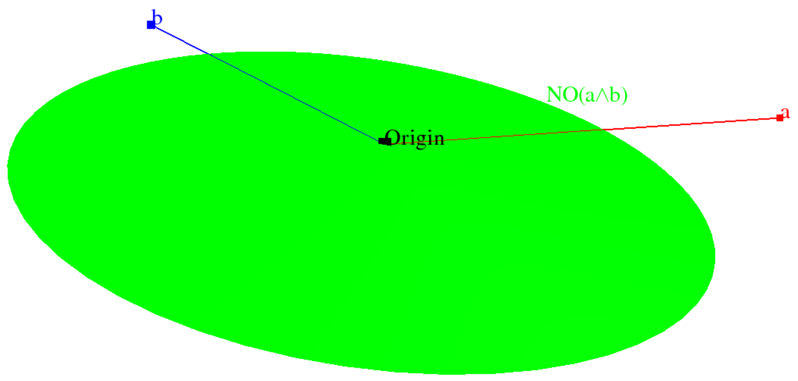


Figure 2.2: $\mathbb{NO}(a \wedge b)$.

2.0.3 The Inner Product Null Space

The inner product null space $\mathbb{NI}(a)$ of a vector a in Euclidean space $\mathbb{R}^3$ is given by the set of vectors $x \in \mathbb{R}^3$ that satisfy $x \cdot a = 0$, that is

$$\mathbb{NI}(a) = \{x \in \mathbb{R}^3 : x \cdot a = 0\}$$

This implies that the set of vectors x are orthogonal to a, which makes $\mathbb{NI}(a)$ a plane that has a as the normal vector [28][3]. This is shown in Figure 2.3.

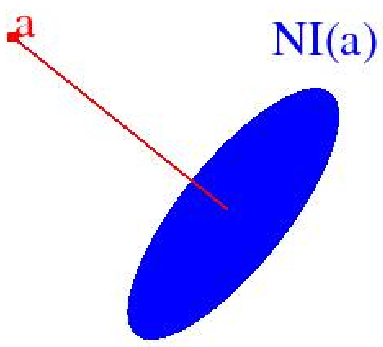


Figure 2.3: $\mathbb{NI}(a)$.

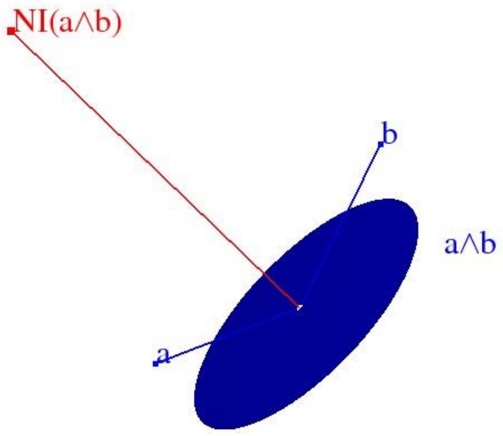


Figure 2.4: $\mathbb{NI}(a \wedge b)$.

The $\mathbb{NI}(a \wedge b)$ of a blade $a \wedge b$ is given by the set of vectors x that satisfy $x \cdot (a \wedge b) = 0$, that is,

$$\mathbb{NI}(a \wedge b) = \{x \in \mathbb{R}^3 : x \cdot (a \wedge b) = 0\}$$

This implies that x must be orthogonal to the plane defined by $a \wedge b$. This is shown in Figure 2.4.

2.0.4 The Dual

Let $\{e_1, e_2, e_3\}$ denote again the orthonormal basis of $\mathbb{R}^3$. The $\mathbb{NI}(e_1)$ is

$$\mathbb{NI}(e_1) = \{\alpha e_2 + \beta e_3 : \alpha, \beta \in \mathbb{R}\}$$

the plane spanned by e_2 and e_3.

The $\mathbb{NO}(e_2 \wedge e_3)$ is also

$$\mathbb{NO}(e_2 \wedge e_3) = \{\alpha e_2 + \beta e_3 : \alpha, \beta \in \mathbb{R}\}$$

which implies that it is relation between the concepts of the $\mathbb{NO}$ and $\mathbb{NI}$ [28]. This relation is *duality.*

Consider a vector a

$$a = a_1 e_1 + a_2 e_2$$

in a 2D Euclidean space $\mathbb{R}^2$ with basis vectors e_1 and e_2. The vector a in geometric product with the pseudoscalar I ($I = e_{12}$ in $\mathbb{R}^2$) gives

$$\begin{aligned} aI &= (a_1e_1 + a_2e_2)e_{12} \\ &= a_1e_{112} + a_2e_{212} \\ &= -a_2e_1 + a_1e_2 \end{aligned}$$

a vector $-a_2e_1 + a_1e_2$ that has been rotated anticlockwise 90°. This is shown in Figure 2.5. The vector $-a_2e_1 + a_1e_2$ in geometric product with the inverse of the pseudoscalar $I^{-1} = e_{21}$ gives

$$\begin{aligned} (-a_2e_1 + a_1e_2)e_{21} &= -a_2e_{121} + a_1e_{221} \\ &= a_2e_2 + a_1e_1 \end{aligned}$$

the original vector a. The geometric product of the vector $-a_2e_1 + a_1e_2$ and the inverse of the pseudoscalar I^{-1}, rotate the vector $-a_2e_1 + a_1e_2$ 90° clockwise back to the original vector a.

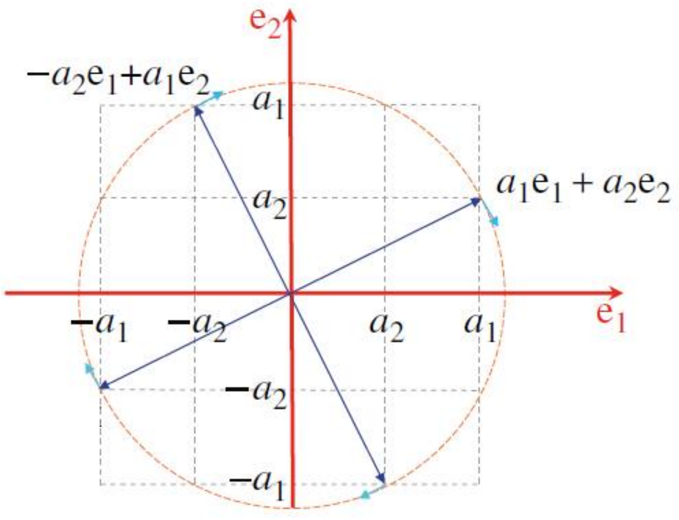


Figure 2.5: Rotation of a in e_{12} [30].

The *dual* a^* of a vector a is

$$a^* = aI^{-1}$$

In Euclidean space $\mathbb{R}^3$, the dual of the basis vectors e_1, e_2 and e_3 are

$$\begin{aligned} e_1^* &= e_1e_{321} = -e_{1123} = -e_{23} = -e_2 \wedge e_3 \\ e_2^* &= e_2e_{321} = -e_{3221} = -e_{31} = -e_3 \wedge e_1 \\ e_3^* &= e_3e_{321} = -e_{3312} = -e_{12} = -e_1 \wedge e_2 \end{aligned}$$

The dual of the unit bivectors $e_1 \wedge e_2$, $e_2 \wedge e_3$ and $e_3 \wedge e_1$ are

$$
\begin{aligned}
(e_1 \wedge e_2)^* &= e_{12}e_{321} = e_{11223} = e_3 \\
(e_2 \wedge e_3)^* &= e_{23}e_{321} = e_{33221} = e_1 \\
(e_3 \wedge e_1)^* &= e_{31}e_{321} = e_{33112} = e_2
\end{aligned}
$$

[30][28]

In Euclidean space $\mathbb{R}^3$ the unit bivectors $e_1 \wedge e_2$ is in the x y plane. The dual $(e_1 \wedge e_2)^* = e_3$ is the unit vector in the z direction, and the normal to the plane. This is shown in Figure 2.6.

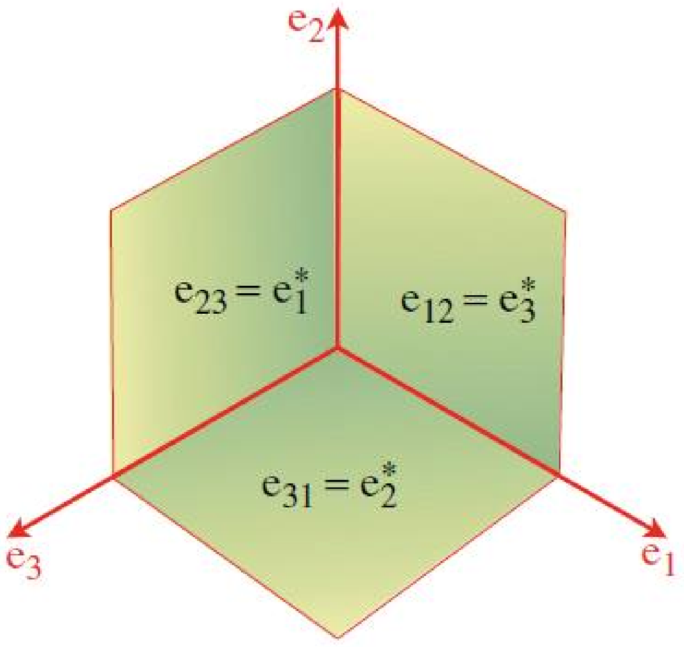


Figure 2.6: Dual [30].

2.0.5 The Vector-Bivector Geometric Product

As already mentioned, a pseudoscalar $I = e_{12}$ in $\mathbb{R}^2$ rotates a vector 90° when it is in geometric product with the vector.

A vector a in $\mathbb{R}^3$

$$a = a_1e_1 + a_2e_2 + a_3e_3$$

in geometric product with a bivector $e_1 \wedge e_2$ gives

$$\begin{aligned} e_{12}a &= a_1e_{121} + a_2e_{12}e_2 + a_3e_{12}e_3 \\ &= a_1e_{121} + a_2e_{122} + a_3e_{123} \\ &= a_2e_1 - a_1e_2 + a_3e_{123} \end{aligned}$$

a vector $a_2e_1 - a_1e_2$ and a trivector a_3e_{123}. These are shown in Figure 2.7. The component of the vector a contained within the bivector has been rotated clockwise 90° while the vector component orthogonal to the bivector has created a volume. Post-multiplying the vector a by the same bivector will give a vector that has been rotated anti-clockwise 90° and the same trivector.

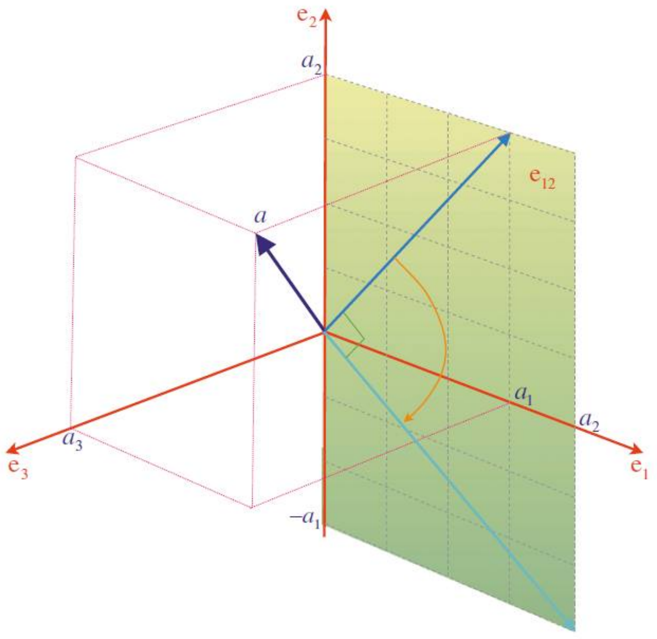


Figure 2.7: Geometric product of a vector a and a bivector $e_1 \wedge e_2$ [30].

If the vector a lies in the plane defined by the bivector $e_1 \wedge e_2$, that is

$$a = a_1e_1 + a_2e_2 + 0e_3$$

then the geometric product of the vector a and the bivector would result in a vector that has been rotated 90° in the plane, and none trivector. The geometric product

$$ae_{12} = \underbrace{a \cdot e_{12}}_{\text{inner product}} + \underbrace{a \wedge e_{12}}_{\text{outer product}}$$

gives a outer product $a \wedge e_{12} = 0$, since the vector a lies in the plane defined by $e_1 \wedge e_2$. This means that the inner product part $a \cdot e_{12}$ is the operation which take the component of the vector a contained within the bivector, and rotate it 90°.

Chapter 3

Camera

3.1 Central Projection Model

The central projection model explains fundamental geometry to project three-dimensional points, curves and surfaces on a two-dimensional plane, the image plane. It models a perspective transformation where the image plane is placed in front of the lens. Light rays reflected from objects in the world penetrates the image plane before they converge in the camera center as shown in Figure 3.1 [2]. A point $P = (X, Y, Z)^{\mathrm{T}}$ with Cartesian coordinates in world is projected on to the image plane

$$x = \frac{fX}{Z} \qquad , \qquad y = \frac{fY}{Z} \tag{3.1}$$

at the point $p = (x, y)^{\mathrm{T}}$ given in Cartesian image coordinates. f is the distance from the camera center to the principal point, known as the focal length. This transformation from three dimensions to two dimensions is called a projective transformation. The consequence of loosing a dimension is that angles and distances are not preserved when transformed in to the image plane. A point on the image plane $p = (x, y)^{\mathrm{T}}$ can be represented by homogeneous coordinates $\tilde{p} = (x, y, 1)^{\mathrm{T}}$. A point in the image plane expressed in homogeneous coordinates $\tilde{p} = (\tilde{x}, \tilde{y}, \tilde{z})^{\mathrm{T}}$ can also be represented by Cartesian coordinates like so

$$x = \frac{\tilde{x}}{\tilde{z}} \qquad , \qquad y = \frac{\tilde{y}}{\tilde{z}} \tag{3.2}$$

A point in the image plane expressed in homogeneous coordinates $p = (\tilde{x}, \tilde{y}, \tilde{z})^{\mathrm{T}}$ represents the same point with

$$p = \left(\frac{\tilde{x}}{\tilde{z}}, \frac{\tilde{y}}{\tilde{z}}\right)$$

Equation 3.1 can be derived to

$$\tilde{x} = fX \qquad , \qquad \tilde{y} = fY \qquad , \qquad \tilde{z} = Z \tag{3.3}$$

where $P = (X, Y, Z)^{\mathrm{T}}$ is a point in the world [2]. The use of equation (3.1) with homogeneous coordinates makes a linear transformation which is beneficial in calculations.

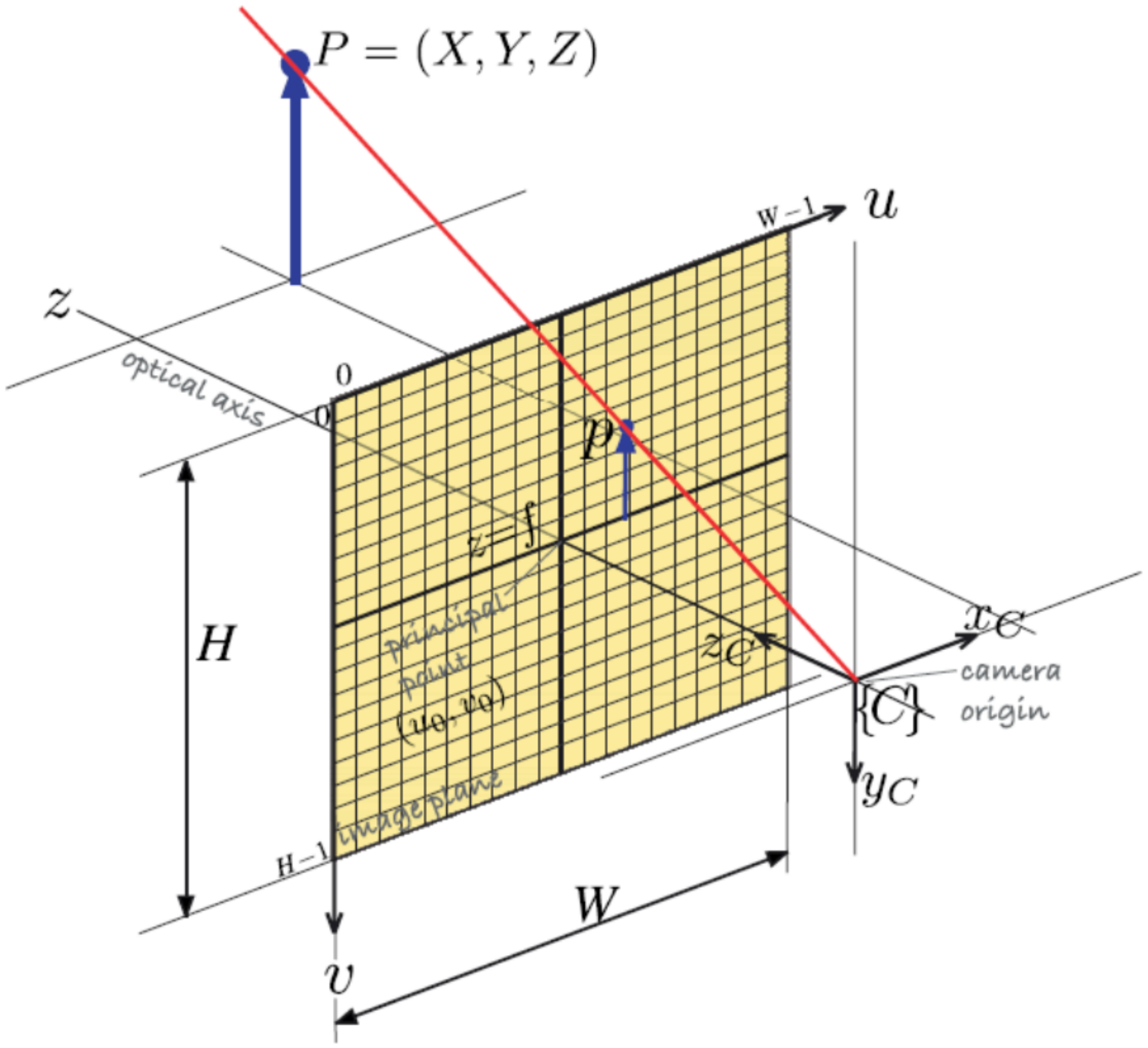


Figure 3.1: The central projection model [2].

In a digital camera, like the Asus Xtion Pro Live and the Logitech HD Pro Webcam C920, the image plane consists of a grid of pixels as shown in Figure 3.1. The pixel coordinates are (u,v) where u represents the horizontal axis and v represents the vertical axis. The coordinates of a point in the world must be scaled down to fit in the image plane. A physical image plane represented in Cartesian coordinates, has its origin in the point where the optical axis penetrates the the image plane. A digital image plane with pixel coordinates on the other hand, have its origin in the upper left corner [2]. A transformation from Cartesian image coordinates (x, y) to pixel coordinates (u,v) is

$$u = \frac{x}{\rho_w} + u_0 \quad , \quad v = \frac{y}{\rho_h} + v_0 \tag{3.4}$$

where ρ_w and ρ_h is the width and height of a pixel respectively, and u_0 and v_0 are the pixel coordinates where the optical axis penetrates the image plane. The relation between pixel coordinates (u, v) and homogeneous pixel coordinates $(\tilde{u}, \tilde{v}, \tilde{w})$ is

$$u = \frac{\tilde{u}}{\tilde{w}} \quad , \quad v = \frac{\tilde{v}}{\tilde{w}} \tag{3.5}$$

Homogeneous pixel coordinates can be expressed by

$$\tilde{u} = \frac{\tilde{x}}{\rho_w} + u_0 \quad , \quad \tilde{v} = \frac{\tilde{y}}{\rho_h} + v_0 \quad , \quad \tilde{w} = \tilde{z} \tag{3.6}$$

Equation (3.6) and equation (3.1) can be derived to

$$\tilde{u} = \frac{fX}{\rho_w} + u_0 \quad , \quad \tilde{v} = \frac{fY}{\rho_h} + v_0 \quad , \quad \tilde{w} = Z \tag{3.7}$$

where the point $P = (X, Y, Z)$ is given with respect to the position and orientation of camera. Equation (3.7) can be expressed by matrices like so

$$\underbrace{\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix}}_{\tilde{p}} = \underbrace{\begin{bmatrix} \frac{f}{\rho_w} & 0 & u_0 \\ 0 & \frac{f}{\rho_h} & v_0 \\ 0 & 0 & 1 \end{bmatrix}}_{K} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{P_0} \underbrace{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}}_{{}^{C}\tilde{P}} \tag{3.8}$$

where K is a camera parameter matrix with the intrinsic camera parameters $(f, \rho_w, \rho_h, u_0, v_0)$. ${}^{C}\tilde{P}$ is the point P with respect to the camera [2].

A point P with respect to a known world coordinate frame $\{0\}$, is shown in Figure 3.2.

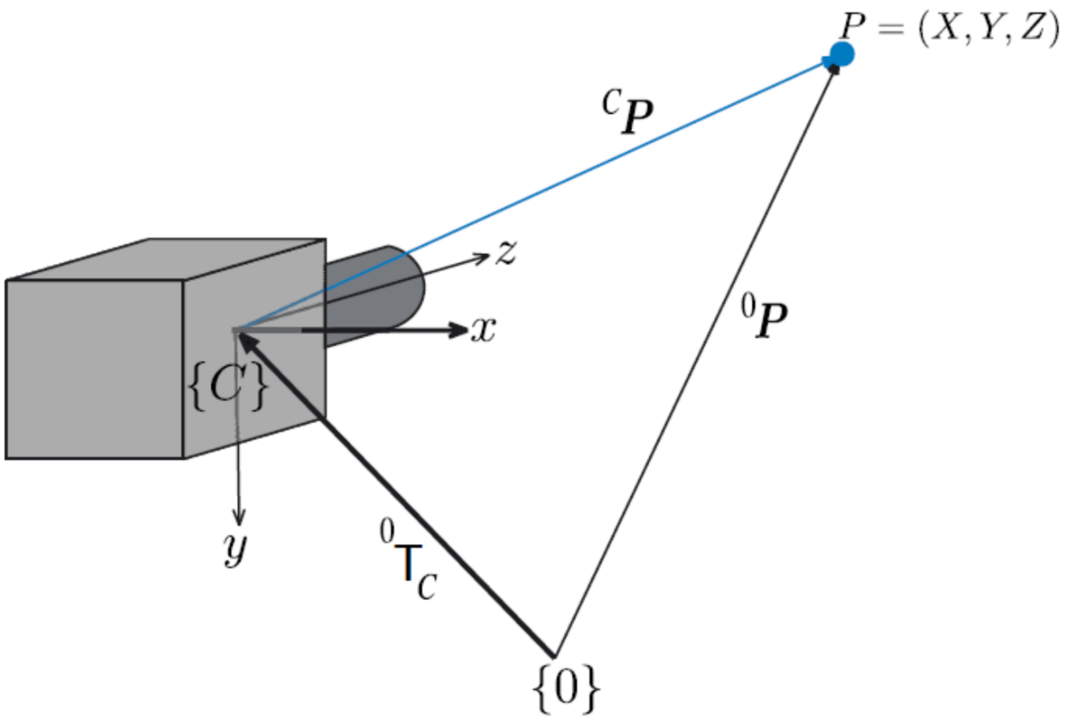


Figure 3.2: Relation between the camera and a point in the world [2].

${}^{0}P$ represents the point given in the coordinate frame $\{0\}$. The point P with respect to the camera frame $\{C\}$ is

$$^{C}\tilde{P} = T_0^C \, {}^{0}\tilde{P} \tag{3.9}$$

where T_0^C is the transformation from $\{C\}$ to $\{0\}$. The inverse of T_0^C is the transformation from $\{0\}$ to $\{C\}$ T_C^0 as shown in Equation 3.10.

$$T_0^C = (T_C^0)^{-1} = \begin{bmatrix} (R_C^0)^{\mathrm{T}} & -(R_C^0)^{\mathrm{T}} t_{0C}^0 \\ 0^{\mathrm{T}} & 1 \end{bmatrix} \tag{3.10}$$

Equation 3.8 and Equation 3.10 can be combined to

$$\underbrace{\begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix}}_{\tilde{p}} = \underbrace{\begin{bmatrix} \frac{f}{\rho_w} & 0 & u_0 \\ 0 & \frac{f}{\rho_h} & v_0 \\ 0 & 0 & 1 \end{bmatrix}}_{K} \underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}}_{P_0} \underbrace{\begin{bmatrix} R_0^C & t_{C0}^C \\ 0^{\mathrm{T}} & 1 \end{bmatrix}}_{T_0^C} \underbrace{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}}_{^{0}\tilde{P}} \tag{3.11}$$

where $\tilde{p} = (\tilde{u}, \tilde{v}, \tilde{w})$ is a pixel represented in homogeneous coordinates [2]. The pixel $\tilde{p} = (\tilde{u}, \tilde{v}, \tilde{w})$ represented in pixel coordinates $p = (u, v)$ is

$$u = \frac{\tilde{u}}{\tilde{w}} \quad , \quad v = \frac{\tilde{v}}{\tilde{w}} \tag{3.12}$$

So that a point $^{0}P = (X, Y, Z)$ in world is projected on to a image plane as a pixel $p = (u, v)$.

3.2 RGB-D Camera

As a normal 2D camera captures an image, a depth camera captures a three-dimensional point cloud. A point cloud is a set of data points in a three-dimensional coordinate system. These points are usually defined by X, Y, and Z coordinates, and they often represent the external surface of an object.

The RGB-D camera used in this project is the Asus Xtion PRO LIVE. It is a structured light based RGB-D (Red Green Blue - Depth) camera as shown in Figure 3.3. Structured light-based depth estimation can be viewed as a specific case of a stereo-vision system where a projector replaces one of the two cameras [26]. Asus Xtion PRO LIVE consists of an infrared camera, a RGB camera and a projector projecting an infrared active pattern.

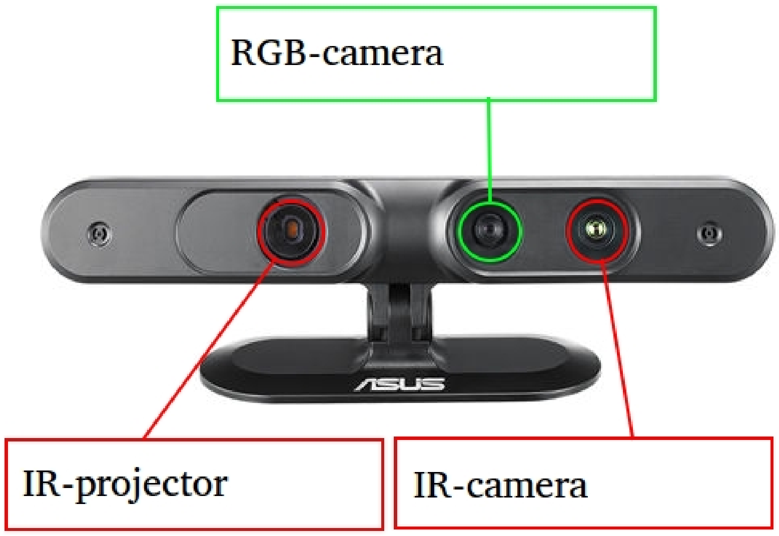


Figure 3.3: Asus Xtion PRO LIVE.

The infrared camera obtains the depth by analyzing the deformation of the pattern projected on to the scene by the projector. Figure 3.4 shows how the light pattern is observed from the IR camera. The lines look straight when projected onto a wall, but are distorted when projected onto people, furniture, or other uneven surfaces.

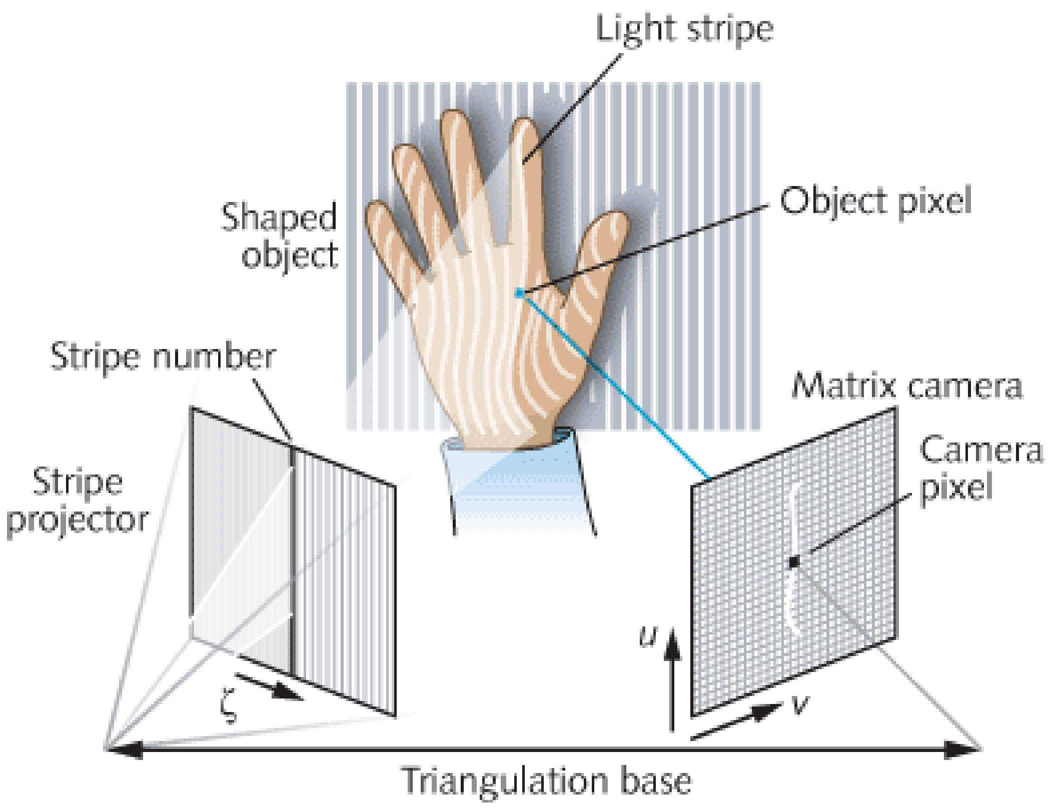


Figure 3.4: Structured light camera [5].

The RGB camera captures the color information and can be aligned with the IR camera by estimating the extrinsic parameters between them. This makes capturing of three-dimensional color point clouds (3D images) possible. Figure 3.5 shows how a three-dimensional point cloud is captured using a structured light camera.

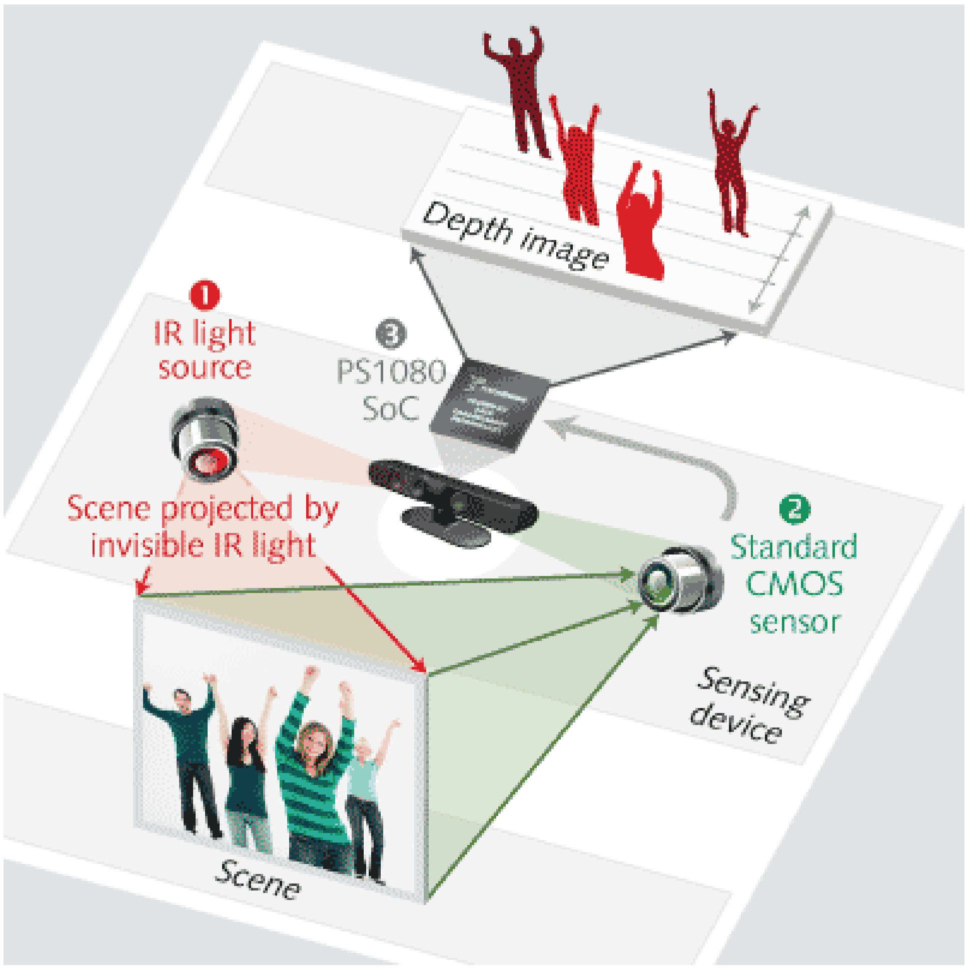


Figure 3.5: Depth image captured with a structured light camera [5].

Structured light based RGB-D cameras have a 60° field of view, which is far less than specialized cameras and laser scanners commonly used for 3D mapping (180°). They also provide depth and color information with less depth precision ($\approx$ 3 cm at 3 m depth), and their depth limit is typically 5 m [8]. Another weakness is that the camera does not detect glass doors, mirrors or shiny surfaces for instance. This proved to be a problem for detection of the shiny aluminium cylinder, especially if it was close to the IR-projector. Figure 3.6 shows the point cloud captured by the RGB-D camera. There are only a few points along the edges of the cylinder.

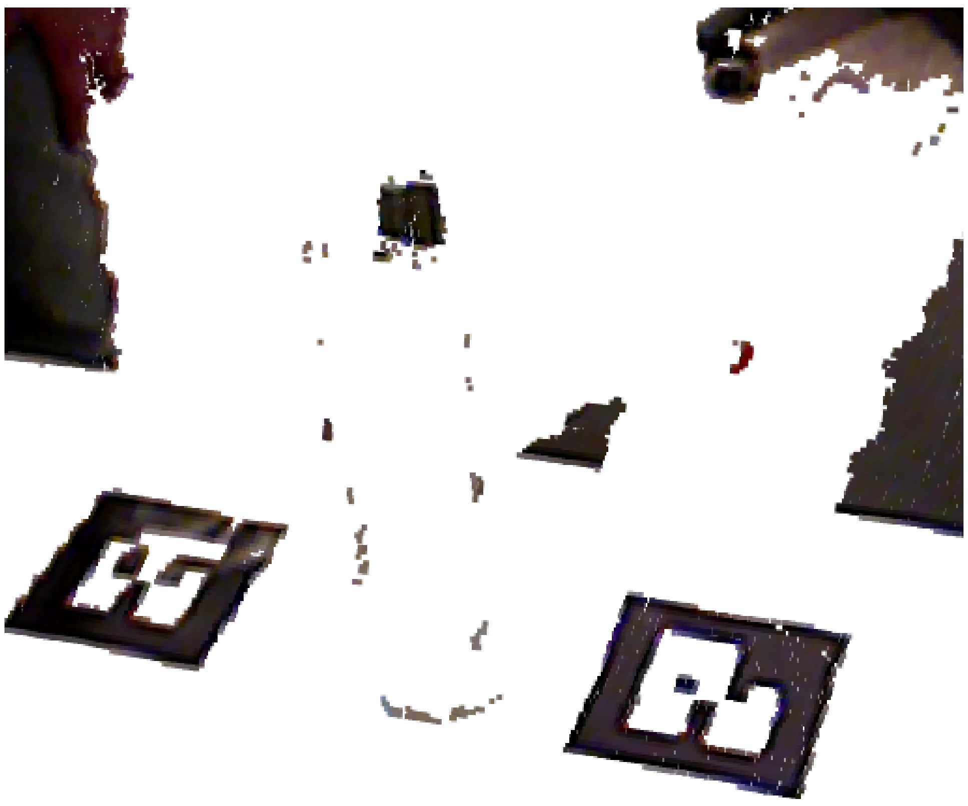


Figure 3.6: The shiny surface of the cylinder reflects the IR pattern so that most points on the cylinder does not show in the point cloud.

To make the camera detect the points, the cylinder was painted blue. As Figure 3.7 shows, the points on the cylinder are now detected much better because the IR pattern is not deflected.

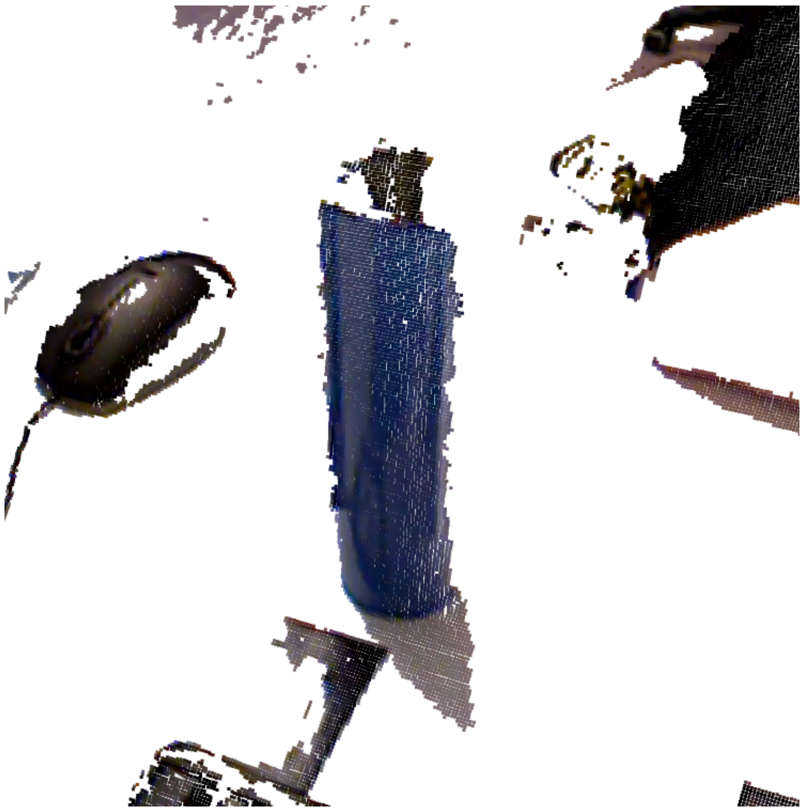


Figure 3.7: The blue cylinder is detected well by the RGB-D camera.

At the macro level, structured light systems are better than other 3D vision technologies at delivering high levels of accuracy with less depth noise in an indoor environment [8].

3.2.1 Calibration of the Asus Xtion PRO LIVE

As for any other sensor, and since the manufacturer's parameters (e.g., focal length) might change between different cameras of the same type, calibration is necessary to increase the camera's sensing accuracy.

The intrinsic camera parameters were found by running the *Cameracalibrator* function in *OpenNI* (Open Natural Interaction) in ROS, Figure 3.8. In order to calibrate the RGB camera using Cameracalibrator, a large chessboard with known dimensions was used. Calibration uses the interior vertex points of the chessboard pattern as known points. The chessboard was then moved around in front of the camera in different orientations and poses while the camera automatic captured pictures for calibration. When the Cameracalibrator had enough good pictures to calibrate, the calibration result could be computed and saved. The saved intrinsic parameters were then used instead of the default parameters from OpenNI. This calibration technique proved to be an efficient way to calibrate the camera as the whole process only lasted about 20 minutes.

The IR camera was calibrated the same way, but with the IR projector covered by a post'it note. The reason for this is that the infrared pattern makes it difficult to detect the chessboard corners accurately in the IR image. Covering the IR projector improves the reliability of the chessboard corner detection. An ideal solution is to block the projector completely and provide a separate IR light source.

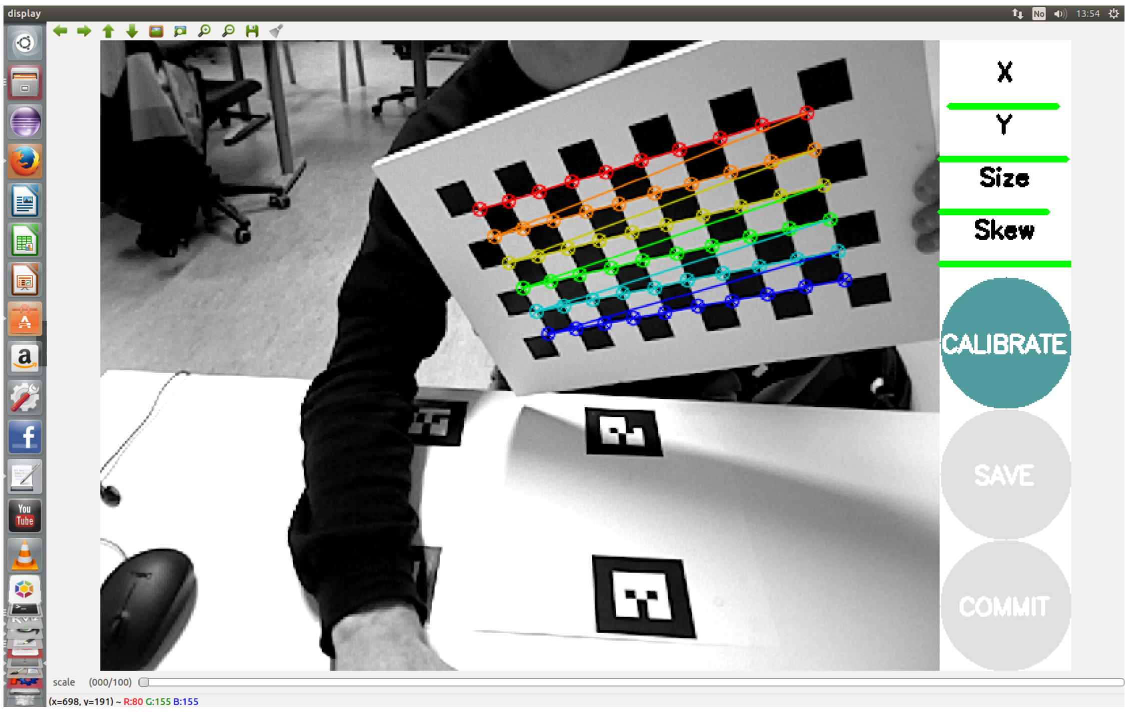


Figure 3.8: Cameracalibrator interface.

The resulting intrinsic parameters for both the RGB and IR camera was given as a camera parameter matrix K_{ir} for the IR camera and K_{rgb} for the RGB camera.

The depth data d from the IR camera can be estimated from regression techniques with the linear function [16]:

$$F(d) = ad + b \tag{3.13}$$

where a and b are constants from the camera. The depth Z is

$$Z = \frac{1.0}{F(d)} \tag{3.14}$$

A 3D-point $(X, Y, Z)^{\mathrm{T}}$ in the world can be approximated from the depth image (u, v, Z) by using the intrinsic parameters of the IR camera shown in Equation 3.15.

$$K_{\mathrm{ir}} = \begin{bmatrix} f_x & 0 & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.15}$$

where $f_x = f/\rho_w$ og $f_y = f/\rho_h$. Then the X, Y and Z coordinates can be calculated as

$$X = \frac{Z(u - u_0)}{f_x} \tag{3.16}$$

$$X = \frac{Z(v - v_0)}{f_y} \tag{3.17}$$

$$Z = \text{depth} \tag{3.18}$$

where u and v are pixels of the depth image and Z is the distance between the object and the IR camera as shown in Figure 3.9 [16].

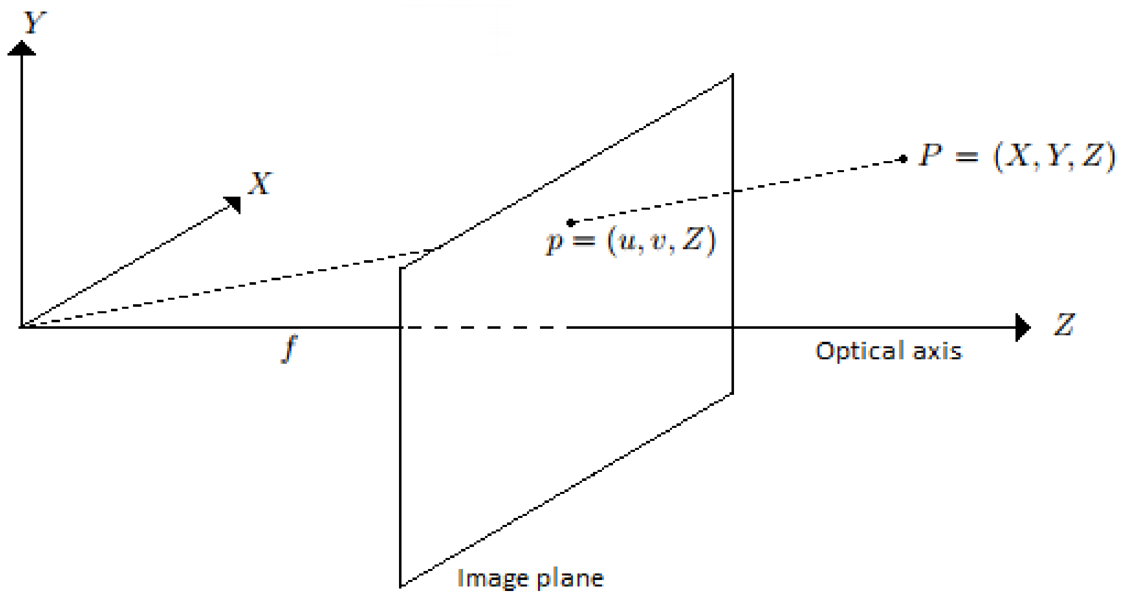


Figure 3.9: A point in the world $P = (X, Y, Z)^{\mathrm{T}}$ projected on to a corresponding point in the image plane $p = (u, v, Z)^{\mathrm{T}}$.

In this project, the default extrinsic camera parameters were used. The extrinsic camera parameters consists of the rotation matrix $R_{\text{ir}}^{\text{rgb}}$ from RGB camera to IR camera and a translation vector $t_{\text{ir}}^{\text{rgb}}$ from RGB camera to IR camera. Together they make the homogeneous transformation matrix $T_{\text{ir}}^{\text{rgb}}$ which can be used to map the images from the IR camera on to the images from the RGB camera. It is possible to compute the depth of the color image from the depth image provided by the IR camera. Consider a pixel $p_{\text{ir}} = (u_{\text{ir}}, v_{\text{ir}})$ in the IR image. The 3D point $P_{\text{ir}} = (X_{\text{ir}}, Y_{\text{ir}}, Z_{\text{ir}})$ corresponding to p_{ir} can be computed by back-projecting p_{ir} in the IR camera's coordinate system [16].

$$P_{\text{ir}} = K_{\text{ir}}^{-1} p_{\text{ir}} \tag{3.19}$$

P_{ir} can be transformed to the RGB camera's coordinate frame through a relative transformation $R_{\text{ir}}^{\text{rgb}}$ and $t_{\text{ir}}^{\text{rgb}}$ like so

$$\underbrace{\begin{bmatrix} X_{\text{rgb}} \\ Y_{\text{rgb}} \\ Z_{\text{rgb}} \end{bmatrix}}_{\tilde{P}_{\text{rgb}}} = R_{\text{ir}}^{\text{rgb}} P_{\text{ir}} + t_{\text{ir}}^{\text{rgb}} \tag{3.20}$$

Figure 3.10 shows the transformation $T_{\text{ir}}^{\text{rgb}}$ from the RGB camera frame $\{C_{\text{rgb}}\}$ to the IR camera frame $\{C_{\text{ir}}\}$.

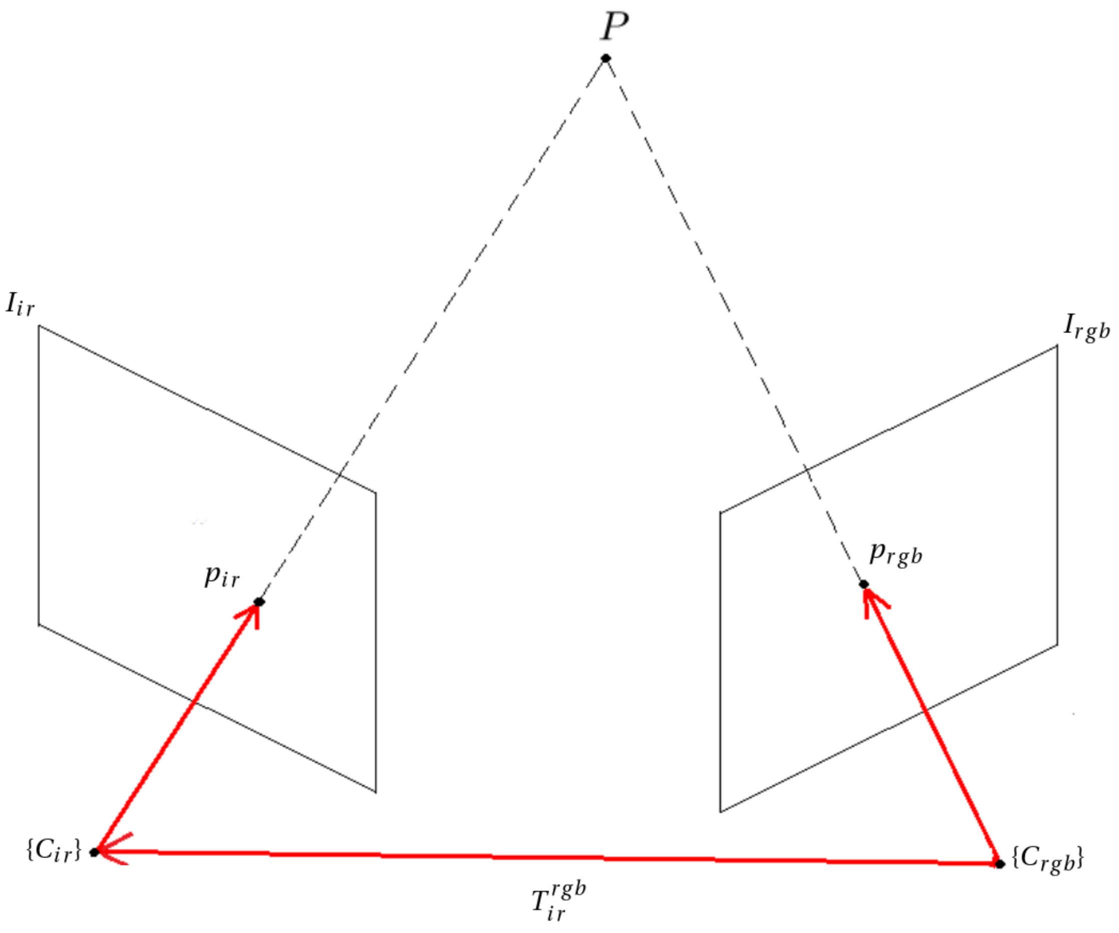


Figure 3.10: Transformation from IR to RGB frame.

Equation (3.21) and equation (3.22) can be used to map the depth images onto the color images [16].

$$u_{\text{rgb}} = \frac{X_{\text{rgb}} f_x}{Z_{\text{rgb}}} + u_0 \tag{3.21}$$

$$v_{\text{rgb}} = \frac{Y_{\text{rgb}} f_y}{Z_{\text{rgb}}} + v_0 \tag{3.22}$$

$p_{\text{rgb}} = (u_{\text{rgb}}, v_{\text{rgb}})$ is a pixel in the image plane of the RGB-camera corresponding to P_{ir} in equation (3.19). f_x and f_y are focal lengths from K_{rgb} and $P_{\text{rgb}} = (X_{\text{rgb}}, Y_{\text{rgb}}, Z_{\text{rgb}})^T$ is given by equation (3.20).

To see if the newly acquired calibration parameters were different than the default camera parameters included in the driver provided by OpenNI, an experiment was performed. An object was placed directly in front of the cameras optical axis. The Z coordinate of the closest point were then written to a log file. This result was then compared with the log file from the same experiment but with the default calibration parameters. The result showed that the depth registered only differed with a few micro meters between the result from calibrated parameters and the default parameters.

Chapter 4

Computer Vision

This chapter contains some theoretical background information relevant for the computer vision part of this thesis.

4.1 Relevant Theoretical Background Information for 2D Camera

Estimating the orientation of the objects is a vital part in the process of assembling the cylinder and the piston.

It was of interest to use a 2D camera to determine the orientation of the parts. This can be achieved by comparing measurements obtained from training images of the parts with measurements from an image stream from a 2D camera. This measurement is called image *features*. Professor David Lowe's SIFT (Scale Invariant Feature Transform) is a method to detect image features that are invariant to image scale and rotation. In other words SIFT is robust when the distance to and the orientation of the objects in the image stream, are different compared to what the objects are in the training images [2].

A feature is a highly distinctive region of an image that can be reliably located in other images of the same scene [20]. This allows a single feature from a training image to be correctly matched with high probability against a feature in the image stream [11].

4.1.1 SIFT

The first stage of feature detection is to identify locations and scales that can be repeatably assigned under different views of the same object. Consider two images of the same object. The object has varying distance to the camera in the images. Detecting same locations on the object in both images, can be accomplished by searching for stable features across all possible scales, using a continuous function of scale known as *scale space*. The scale space of an input image $I(u, v)$ in *convolution* (*) of a Gaussian function $G(u, v, \sigma)$, generate a blurred output image $L(u, v, \sigma)$

$$L(u, v, \sigma) = G(u, v, \sigma) * I(u, v) \tag{4.1}$$

An example of an input image $I(u, v)$ and a output image $L(u, v, 1)$, are shown in Figure 4.1

and Figure 4.2.

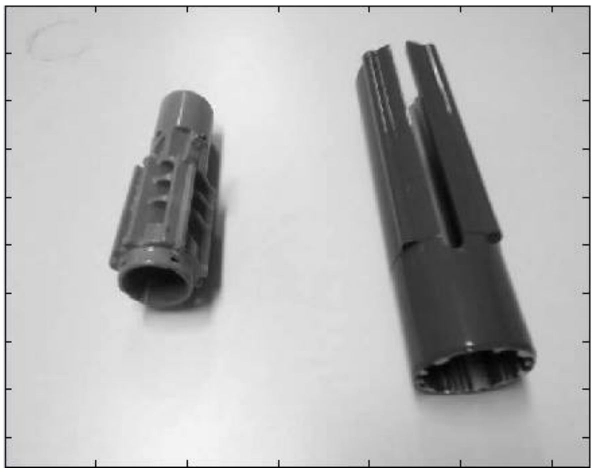


Figure 4.1: Input image $I(u, v)$

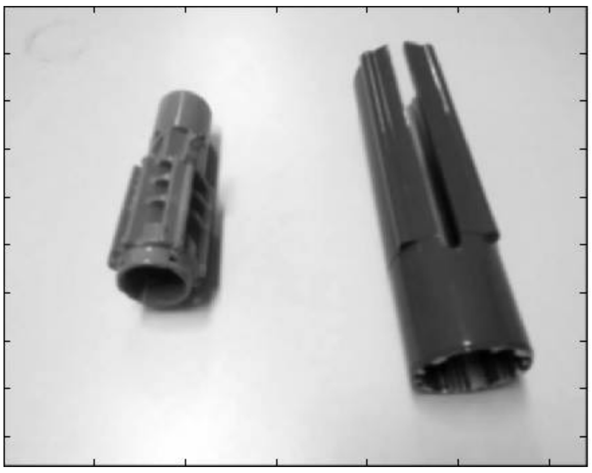


Figure 4.2: Output image $L(u, v, 1)$

Convolution is a linear spatial operator. For every output pixel (u, v) in $L(u, v)$ the corresponding window of pixels from $I(u, v)$ is multiplied element-wise with a kernel [2]. In Figure 4.2 this kernel is the Gaussian function

$$G(u, v, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(u^2+v^2)/2\sigma^2} \tag{4.2}$$

In Figure 4.3 a Gaussian kernel with $\sigma = 1$ and $\sigma = 1.2$ is shown.

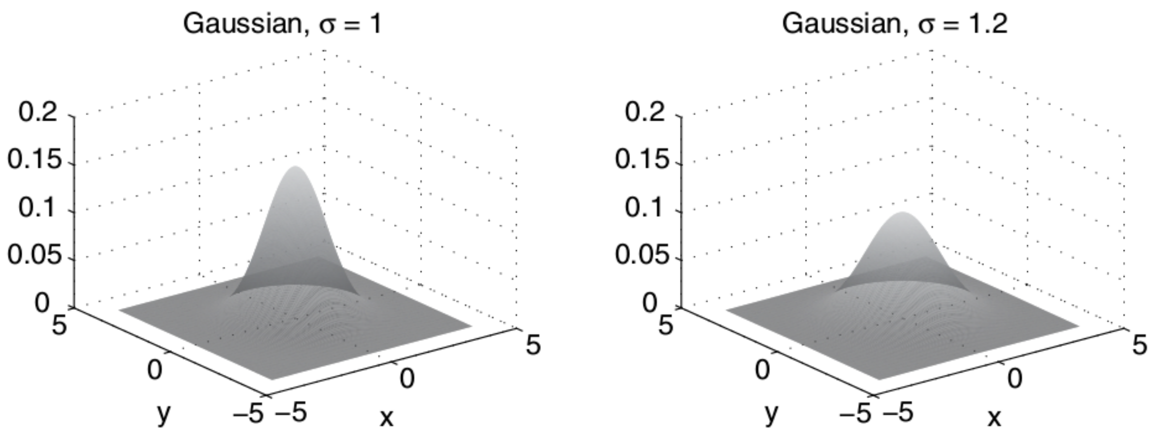


Figure 4.3: Gaussian kernel with $\sigma = 1$ and $\sigma = 1.2$ [20]

.

For detecting stable features, SIFT uses scale-space extrema in a continuous difference-of-Gaussian function convolved with the input image

$$\begin{aligned} D(u, v, \sigma) &= (G(u, v, k\sigma) - G(u, v, \sigma)) * I(u, v) \\ &= L(u, v, k\sigma) - L(u, v, \sigma) \end{aligned} \tag{4.3}$$

SIFT has an octave structure for computing the scale-space difference-of-Gaussian (see Figure 4.4). An octave of a Gaussian scale space is a pair of images whose σ differ by a factor of k. From equation (4.3) $k = \frac{1}{s}$ where each set of $s+1$ images results in an octave. Lowe suggest $s = 3$ and $\sigma = 1.6$ [20]. Once a complete octave has been processed, the next octave re-sample the Gaussian image that has twice the initial value of σ by taking every second pixel (u, v) in each row and column [20][11].

In Figure 4.4 this process is shown. The white images represent the Gaussian blurred images with the same sequence of scales applied to octaves of images of different resolutions. The Gaussian images at each octave has half the resolution of the images in the octave above. The gray images represents the difference-of-Gaussian images.

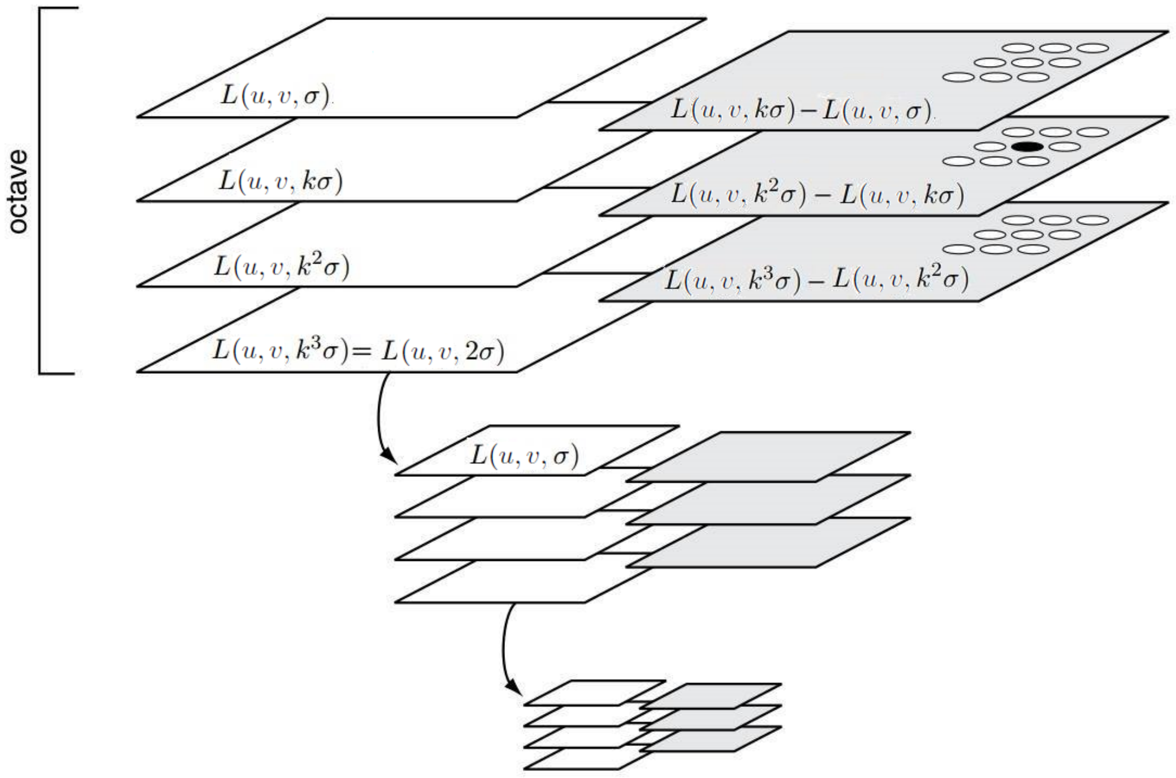

octave
$L(u,v,\sigma)$
$L(u,v,k\sigma)$
$L(u,v,k^2\sigma)$
$L(u,v,k^3\sigma)= L(u,v,2\sigma)$
$L(u,v,k\sigma)-L(u,v,\sigma)$
$L(u,v,k^2\sigma)-L(u,v,k\sigma)$
$L(u,v,k^3\sigma)-L(u,v,k^2\sigma)$
$L(u,v,\sigma)$

Figure 4.4: Scale-space [20].

The key quantity of difference-of-Gaussian feature detection is the difference of adjacent Gaussians applied to the input image $I(u, v)$. The difference-of-Gaussian detection seek for maxima and minima in both the spatial and scale dimensions to detect features. In the difference-of-Gaussian images, local maxima and minima are detected by comparing a pixel to its (9+8+9=) 26 neighbors in 3×3 regions at the current and adjacent scales. A candidate for a *keypoint* is accepted if it is larger or smaller than all of its eight neighbors in the current image and the nine neighbors in the scale above and below [11]. This is shown in the top right corner in Figure 4.4 and in Figure 4.5.

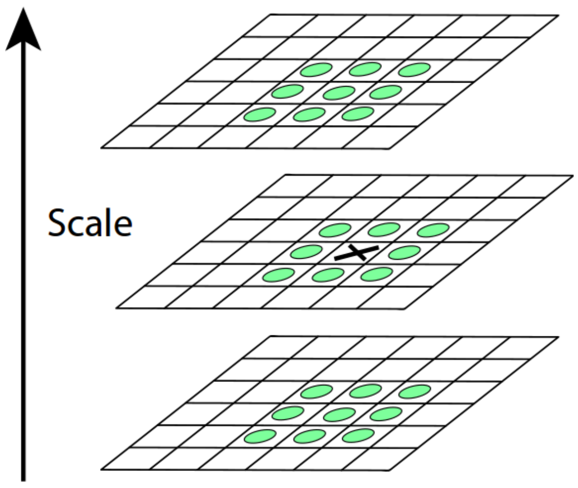


Figure 4.5: Scale-space difference-of-Gaussian [11].

Figure 4.6 shows an example of a difference-of-Gaussian image generated from the input image in Figure 4.1.

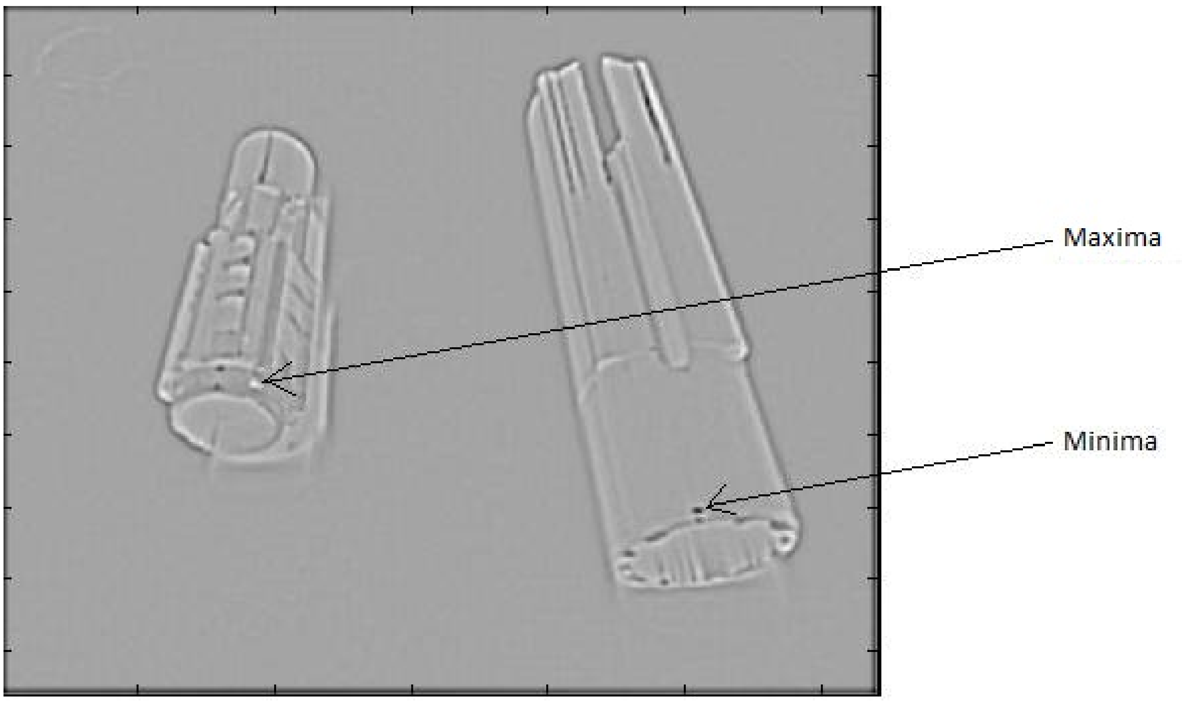


Figure 4.6: Difference-of-Gaussian image.

For some of the candidates of keypoints, there could be some *outliers* because of noise. Example of outliers can be candidates along an edge or low contrast.

An approximation of the difference-of-Gaussian function D by a Taylor series is

$$D(\boldsymbol{x}) = D + \frac{\partial D^{\mathrm{T}}}{\partial \boldsymbol{x}}\boldsymbol{x} + \frac{1}{2}\boldsymbol{x}^{\mathrm{T}}\frac{\partial^2 D}{\partial \boldsymbol{x}^2}\boldsymbol{x} \quad , \quad \boldsymbol{x} = (x, y, \sigma)^{\mathrm{T}} \tag{4.4}$$

and an extrema $\hat{\boldsymbol{x}}$ is located at

$$\boldsymbol{x} = -\frac{\partial^2 D^{-1}}{\partial \boldsymbol{x}^2}\frac{\partial D}{\partial \boldsymbol{x}} \tag{4.5}$$

The value of $D(\hat{\boldsymbol{x}})$ at the location of an extrema $\hat{\boldsymbol{x}}$ in equation (4.4), must be lower than a threshold to be accepted as an *inlier*. This process is useful for rejecting outliers that are unstable extrema with low contrast.

The *Hessian* matrix H of the difference-of-Gaussian function D is

$$H = \begin{bmatrix} D_{uu} & D_{uv} \\ D_{uv} & D_{vv} \end{bmatrix} \tag{4.6}$$

where $D_{uu} = \frac{\partial^2 D}{\partial u \partial u}$, $D_{uv} = \frac{\partial^2 D}{\partial u \partial v}$ and $D_{vv} = \frac{\partial^2 D}{\partial v \partial v}$. Further outlier rejection can be done by evaluating the *trace* (Tr) and determinant of H

$$\frac{\mathrm{Tr(H)}^2}{\mathrm{Det(H)}} = \frac{(r+1)^2}{r} \quad , \quad r = \frac{\lambda_1}{\lambda_2} \tag{4.7}$$

where

$$\begin{aligned} \mathrm{Tr(H)} &= D_{uu} + D_{vv} = \lambda_1 + \lambda_2 \\ \mathrm{Det(H)} &= D_{uu}D_{vv} - (D_{uv})^2 = \lambda_1\lambda_2 \end{aligned} \tag{4.8}$$

The parameter r in equation (4.7), must be smaller than a threshold to be accepted as a keypoint. This process is useful for rejecting outliers that are unstable extrema along an edge. At this stage, keypoint have been localised with a pixel (u, v) and scale [11].

As mentioned, the SIFT features are invariant of orientation. To achieve this orientation invariance, each feature has to be given an orientation assignment. The scale of the keypoint is used to select the Gaussian smoothed image, L, with the closest scale to the scale of the keypoint, so that all computations are performed in a scale-invariant manner. For the images $L(u, v)$ at this scale, the gradient magnitude, $m(u, v)$, and orientation, $\theta(u, v)$, is computed

$$\begin{aligned} m(u,v) &= \sqrt{(L(u+1,v) - L(u-1,v))^2 + (L(u,v+1) - L(u,v-1))^2} \\ \theta(u,v) &= \tan^{-1}\left(\frac{L(u,v+1) - L(u,v-1)}{L(u+1,v) - L(u-1,v)}\right) \end{aligned} \tag{4.9}$$

and used for creating a histogram of the gradient orientation within the scale-covariant circle region around the keypoint. The orientation histogram has 36 bins (one bin for each ten degrees) covering the 360 ° range of orientations. The orientation histogram is weighted by its gradient magnitude and by a Gaussian-weighted circular window with 1.5σ where σ is the scale of the keypoint. That is, for each pixel (u, v) in the region around the keypoint (u_0, v_0, σ), the quantity $m(u, v)G(u - u_0, v - v_0, 1.5\sigma)$ is the quantity incrementing in the bin corresponding to $\theta(u, v)$ [20].

The highest peak in the histogram and other local peaks that is within 80% of the highest peak, is used to create the orientation of the keypoint. At this stage, the keypoint has an image location (u, v), scale σ and orientation θ. These parameters are the SIFT feature for a keypoint [11].

The next step is to describe the feature with a 128 number vector called a *SIFT descriptor*.

In Figure 4.7 the input image from Figure 4.1 is shown with a feature. The location of the keypoint is in the center of the green circle, the size of the green circle is corresponding to the scale of the feature, and the green line indicate the orientation of the feature. The descriptor is computed from the pixels inside the blue square in Figure 4.8. The orientation of the square in Figure 4.8 is the same as the orientation of the feature in Figure 4.7. Each side of the square is a multiple of the feature's scale [20].

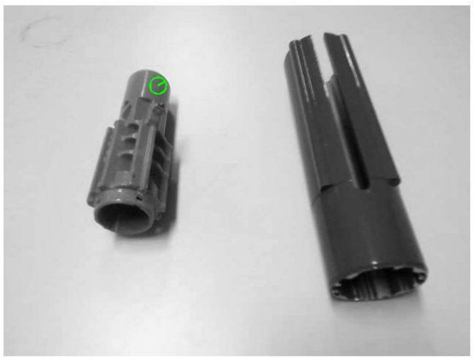


Figure 4.7: Input image $I(u, v)$ with one feature.

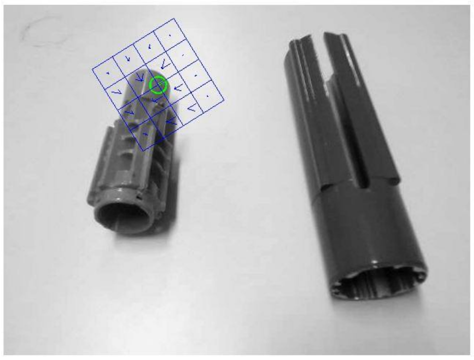


Figure 4.8: Input image $I(u, v)$ with one descriptor and feature.

The image pixels inside the square are smothered with the appropriate Gaussian for the feature's scale. Then the gradient magnitude and orientation at each pixel is estimated, as shown on the left in Figure 4.9. These are weighted by a Gaussian window (blue circle in Figure 4.9), subdivided into a 4×4 grid of smaller squares, and used to create a histogram of eight gradient orientations within each grid square in Figure 4.8. To the right in Figure 4.9 this is also shown.

So for the descriptor in Figure 4.8 there are 8 gradient orientations within each of the 4×4 squares, which makes a $4 \times 4 \times 8 = 128$ element vectors for each feature. The vector is then normalized to unit length. This normalized vector is the descriptor for the feature in Figure 4.7.

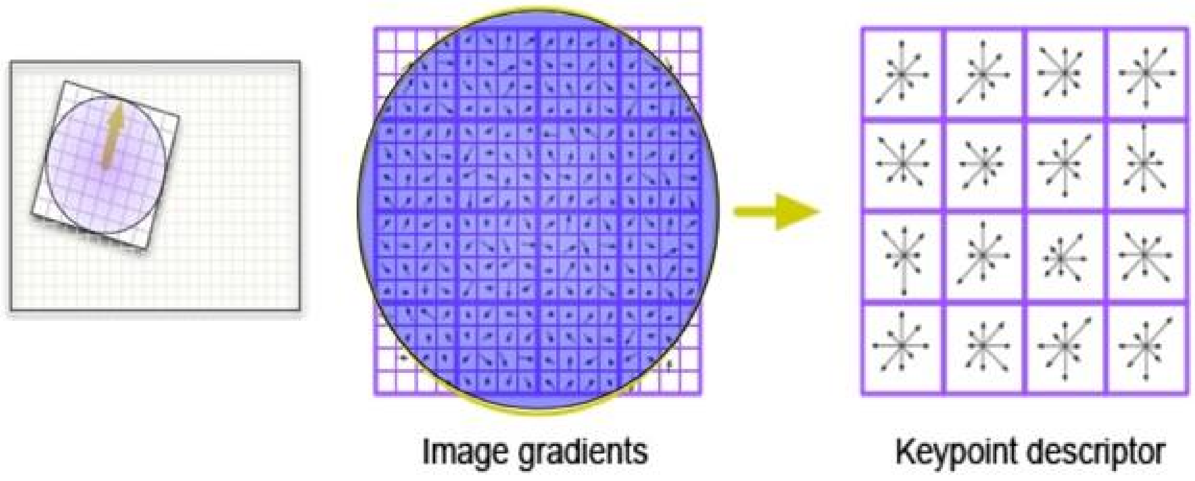


Figure 4.9: Computing a descriptor [1].

In Figure 4.10 there is an image I_o of an object and an image I_s of a scene, with some estimated SIFT features (the circles) in both images.

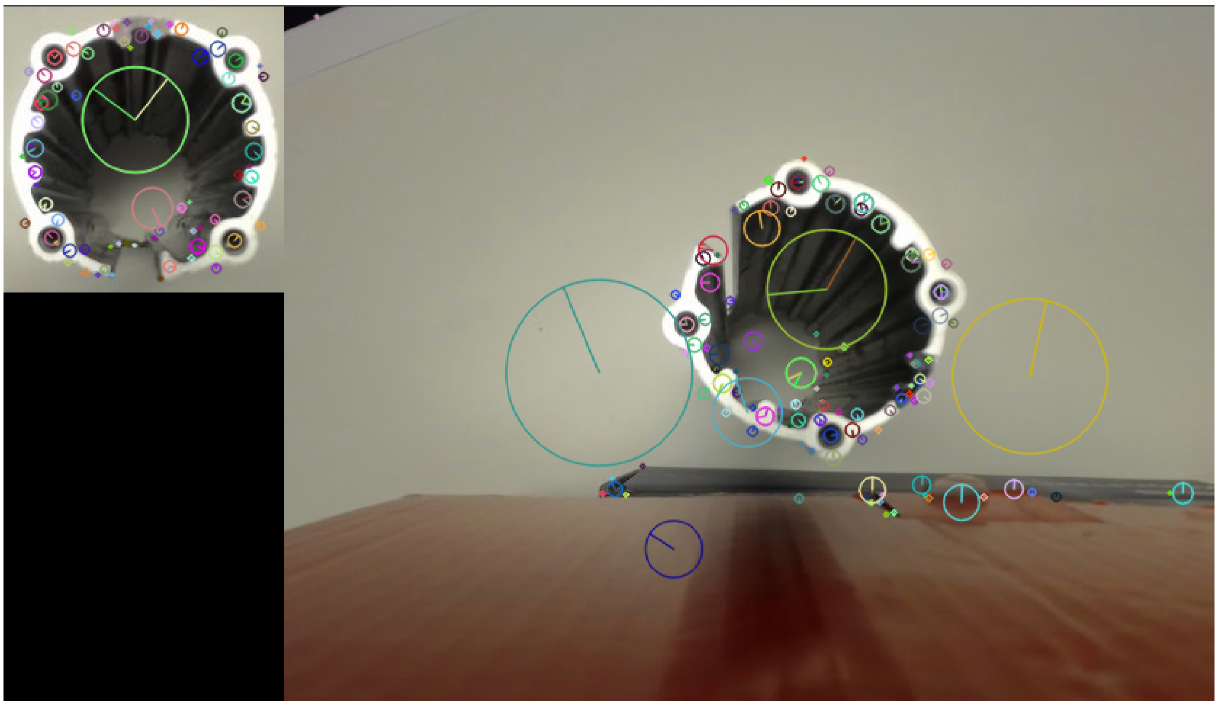


Figure 4.10: From left: Image I_o of a object and an image I_s of a scene. The circles are features.

In I_o there has been computed a set of descriptors $O = o^1, ..., o^{N_1}$ for the features, and in I_s there has been computed a set of descriptors $S = s^1, ..., s^{N_2}$ for the features. The next step is to generate a list of *feature correspondences* $\{(c_o^j, c_s^j), j = 1, ..., N_3\}$, which says that $o^{c_o^j} \in O$ and $s^{c_s^j} \in S$ are a match. Finding the feature correspondences, can be done by estimating the Euclidean distance between descriptors in I_s and I_o and then using the method of nearest neighbor. The euclidean distance D between a descriptor o from the set O in I_o and s from the set S in I_s is

$$D(o,s) = \sqrt{\sum_{i=1}^{128}(o_i - s_i)^2} \tag{4.10}$$

(each descriptor vector has 128 element). The match to a descriptor o from a set of descriptors in S is computed by use of nearest neighbor

$$s^* = \arg\min_{s\in S} D(o,s) \tag{4.11}$$

A descriptor o and s^* are accepted as a match if $D(o,s^*)$ is below a specified threshold [20].

From the list of feature correspondences, the *corresponding keypoint* are extracted. In the rest of this chapter, the notation $p = (u,v)^T$ is used for a keypoint in I_o and $p' = (u',v')^T$ is used for a keypoint in I_s. A pair of corresponding keypoints p and p', is the location of a pair of matched features.

In Figure 4.11 some of the corresponding keypoints from Figure 4.10 are shown.

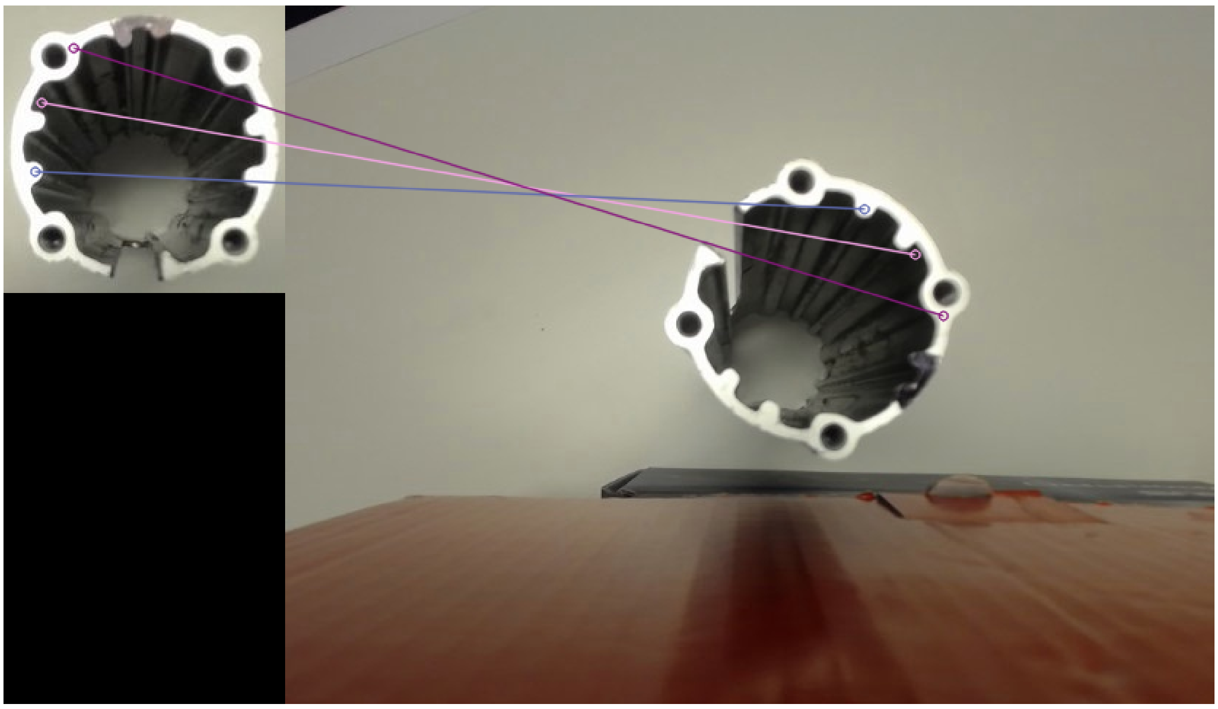


Figure 4.11: From left: Image I_o with three keypoints (circles) and an image I_s with three keypoints (circles). The lines indicates that the keypoints are corresponding.

4.1.2 Homography

The goal is to find the orientation of the object (in I_o) in an image stream (continuous updating of I_s). The set of corresponding keypoints (from the set of matched features) can be used to find a perspective transformation between the object in I_o and I_s. The projective transformation or homography H is estimated

$$\underbrace{\begin{bmatrix} u_i' \\ v_i' \\ 1 \end{bmatrix}}_{\tilde{p_i}'} = \underbrace{\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}}_{homography} \underbrace{\begin{bmatrix} u_i \\ v_i \\ 1 \end{bmatrix}}_{\tilde{p_i}} \tag{4.12}$$

so that the back-projection error

$$\sum_{i=1} \left(u_i' - \frac{h_{11}u_i + h_{12}v_i + h_{13}}{h_{31}u_i + h_{32}v_i + h_{33}} \right)^2 + \left(v_i' - \frac{h_{21}u_i + h_{22}v_i + h_{23}}{h_{31}u_i + h_{32}v_i + h_{33}} \right)^2 \tag{4.13}$$

is minimized [15]. In equation (4.12), p_i' are the keypoints $\{(u_1', v_1')^T, ..., (u_n', v_n')^T\}$ in I_s and p_i are the keypoints $\{(u_1, v_1)^T, ..., (u_n, v_n)^T\}$ in I_o. However, if not all of the point pairs $\{p_i, p_i'\}$ are corresponding, the homography matrix can be incorrect.

This is why it is of interest to also include a robust iterative method called RANSAC (RANdom SAmple Consensus) in the estimation of the homography matrix. RANSAC will try many different random subsets of corresponding point pairs (of four pairs each) and the homography matrix H is estimated on behalf of numbers of *inliers* that fits to the homography matrix. The best subset is then used to estimate the final homography matrix in equation (4.12). The homography is determined up to scale. Therefore it is normalized so that $h_{33} = 1$ [15][4].

The projective transformation or homography, maps points in a a training/object image on the image planes in the image stream. That is, it maps the training/object image plane to the image stream. The orientation of the mapped plane in the image stream is used to determine the orientation of the object.

4.2 Relevant Theoretical Background Information for RGB-D Camera

This chapter contains some theoretical background information relevant for the RGB-D camera, followed by an explanation of the different programs for for obtaining the position of the piston and the cylinder.

4.2.1 AR-tags

AR is an abbreviation for augmented reality and means to make virtual objects appear to enter the real world. AR-markers or AR-tags look like QR-codes and a particular pattern can be recognized in the image from the camera and represent the location of a virtual object. The three-dimensional orientation of the tag can be registered as well as the position of the tag. It is also possible to make digital artificial objects appear on the surface of the tag.

In this project the ROS package ar_track_alvar was used to make a coordinate system which is constant regardless of the position of the camera. Figure 4.12 shows AR-tags, and how they are seen from the camera on the computer screen.

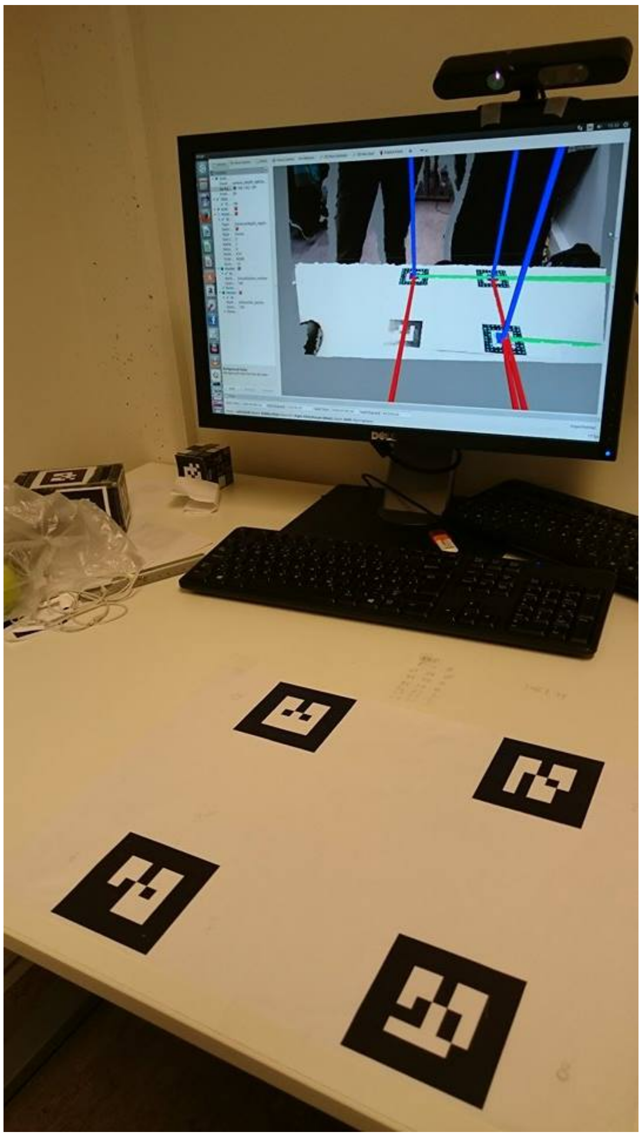


Figure 4.12: AR-tags.

The computer recognizes the AR-tags viewed form the camera and creates an artificial coordinate frame with the origin in the center of three of the four AR-tags.

4.2.2 Estimating Surface Normals in a Point Cloud

Local geometric features such as surface normal or curvature at a point form a fundamental basis for extracting semantic information from 3D sensor data. This section explains how to use approximations to infer the surface normals from a point cloud dataset [23].

For a *query point* p_q its neighboring points $\mathcal{P}^k = \{p_1, p_2..p_k\}$ is used in the estimation of a surface normal at p_q. The problem of determining the normal to a point on a surface can be approximated by the problem of estimating the normal of a plane tangent to the surface, which in turn becomes a least-square plane fitting estimation problem in $\mathcal{P}^k$. The plane is represented as a point x and a normal n. The distances d_i from the points p_i, in the neighborhood $\mathcal{P}^k$, to the plane is $d_i = (p_i - x) \cdot n$. The point x and the normal n are computed in a least-square sense $d_i = 0$. That is, the point x is determined as the 3D centroid of the neighboring points $\mathcal{P}^k$

$$x = \frac{1}{k} \cdot \sum_{i=1}^{k} p_i \tag{4.14}$$

and the normal n is given by analyzing the eigenvalues λ_j and eigenvectors v_j (or PCA – Principal Component Analysis) of the covariance matrix $\mathcal{C}$

$$C = \frac{1}{k} \sum_{i=1}^{k} (p_i - x) \cdot (p_i - x)^T \ , \quad C \cdot \mathsf{v}_j = \lambda_j \cdot \mathsf{v}_j \ , \quad j \in \{0, 1, 2\} \tag{4.15}$$

The eigenvectors v_j form an orthogonal frame, corresponding to the principal components of the query points neighbours. If $0 \leq \lambda_0 \leq \lambda_1 \leq \lambda_2$, the eigenvector v_0, corresponding to the smallest eigenvalue λ_0 can be used as an estimate of a normal $+n = \{n_x, n_y, n_z\}$ or $-n$ [23]. The ratio between the smallest eigenvalue λ_0 and the sum of eigenvalues provides an estimate of the local curvature [6].

There is in general no mathematical way to solve for the sign of the normal $+n$ or $-n$ using PCA. The orientation computed is ambiguous and not consistently over an entire point cloud dataset. Figure 4.13 shows a side view of a plane where the normals are pointing both upwards and downwards.

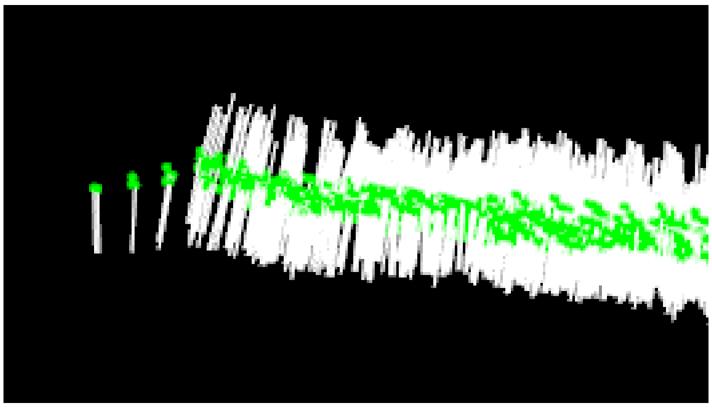


Figure 4.13: Surface normals have ambiguous direction.

The right orientation of the surface normals can be found if the viewpoint V_p is known. In order for the normals n_i to be oriented consistently towards the viewpoint, the following equation needs to be satisfied:

$$n_i \cdot (\mathsf{v}_p - p_i) > 0 \tag{4.16}$$

where p_i is the corresponding point to the normal n_i.

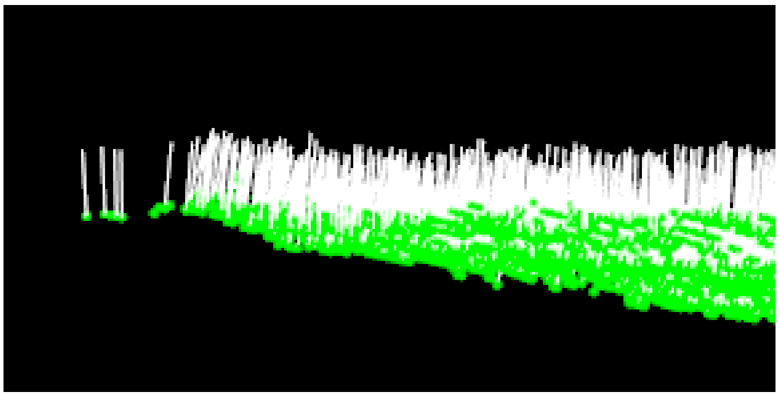


Figure 4.14: Surface normals are oriented in the same direction.

As already mentioned, the surface normal at a point needs to be estimated from the surrounding point neighborhood $\mathcal{P}^k = \{p_1, p_2..p_k\}$ support of the point (also called k-neighborhood). To estimate the nearest neighbours $\mathcal{P}^k$, an appropriate scale factor is set either by choosing the radius r of the sphere containing neighbouring points or the number of k nearest neighbors to use for the feature estimation. A reasonably chosen scale factor results in estimated surface normals that are approximately perpendicular for planar surfaces. If the scale factor is too big, and the set of neighbors is larger covering points from adjacent surfaces, the estimated point feature representations get distorted, with rotated surface normals at the edges of planar surfaces, and smeared edges and suppressed fine details [23]. Figure 4.15 shows a point cloud with $r = 0.1$, which uses all neighbors in a sphere of radius 10 cm around each query point p_q. The surface normals are pointing downwards and are almost parallel to the surface normals of the table when they should be pointing in the horizontal direction. The scale factor is too high.

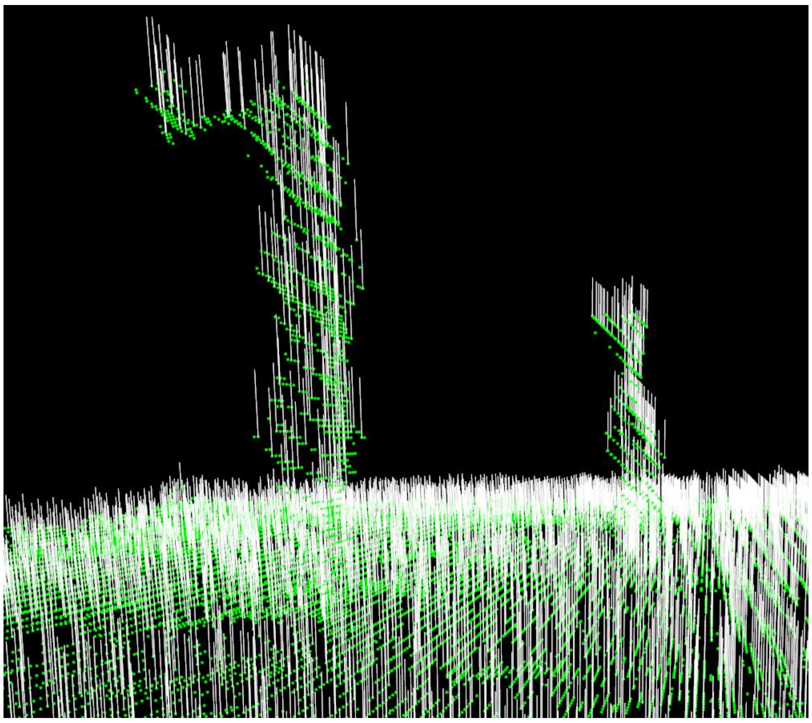


Figure 4.15: Surface normals, $r = 0.3$.

Figure 4.16 shows a point cloud with $r = 0.03$. Here the normals are pointing more in the horizontal direction, which is what we are looking for.

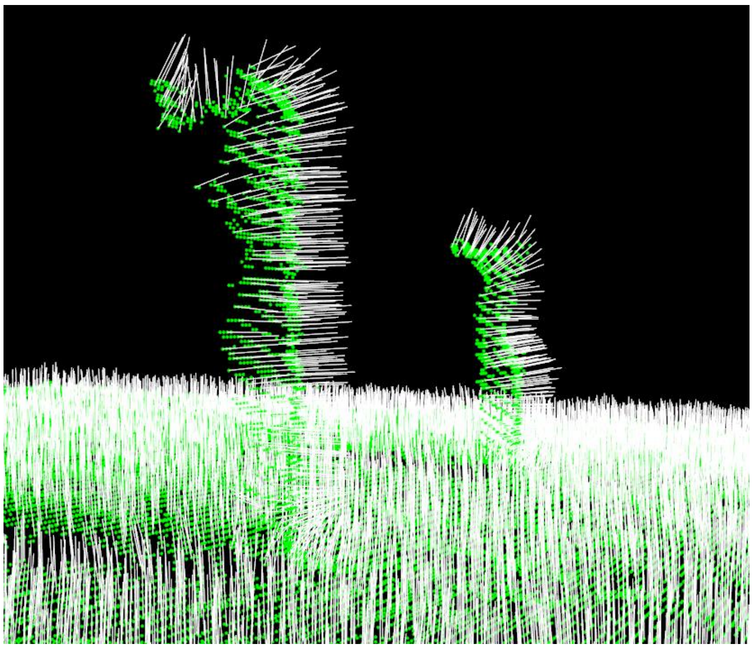


Figure 4.16: Surface normals, $r = 0.03$.

After some testing and adjustment $r = 0.04$ was chosen. The result is shown in Figure 4.17.

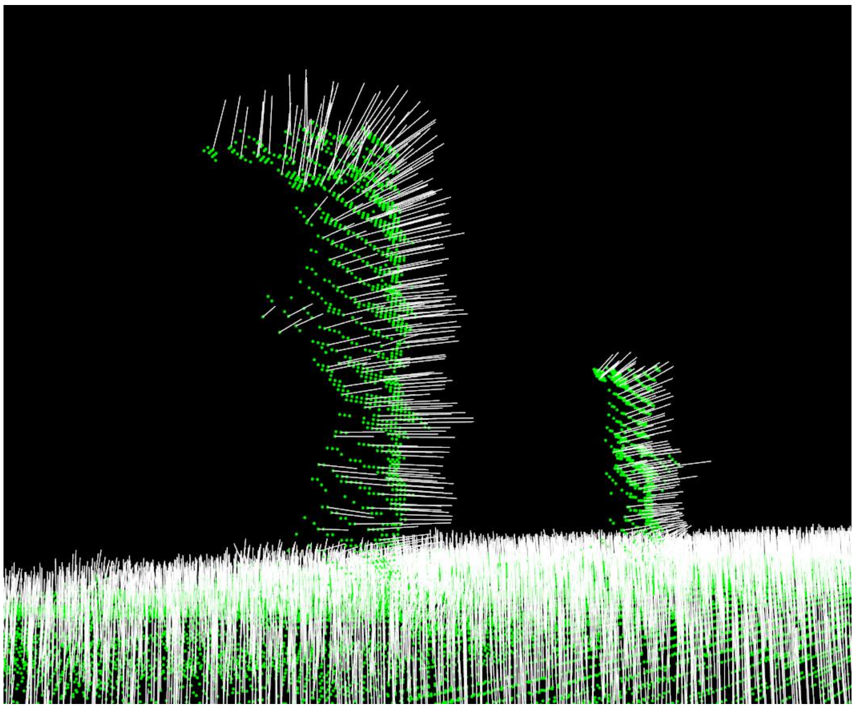


Figure 4.17: Surface normals, $r = 0.04$.

The figure clearly shows that the estimated surface normals are approximately perpendicular for the plane and for the objects on it.

The scale factor to chose, depends on the detail level needed in the cloud. Simply put, if the curvature at the edge between the handle of a mug and the cylindrical part is important, the scale factor needs to be small enough to capture those details, and large otherwise [23].

Estimation of surface normals can be summarized by the following:

1. Get the nearest neighbors of point cloud P
2. Compute the surface normal n of the point p
3. Check if n is consistently oriented towards the viewpoint and flip otherwise

4.2.3 PFH - Point Feature Histogram

The characterization of the surface geometry around a point is given through the formulation of a Point Feature Histogram (PFH) 3D feature. This feature space is extremely useful for the problem of finding point correspondences.

Surface normals and curvature estimates are basic in their representation of the geometry around a specific point, as point feature representations. Though extremely fast and easy to compute, they cannot capture too much detail, as they approximate the geometry of a point's neighbouring points (k-neighborhood) with only a few values [23]. The consequence of this is that the informative characteristics gets reduced because most scenes contains many points with the same or very similar feature values.

Point feature histograms (PFH) are robust multi-dimensional features which describe the local geometry around a point p for 3D point cloud datasets, and they rely on 3D data (X, Y, Z) as well as surface normals. The goal of the PFH formulation is to encode a point's k-neighborhood geometrical properties by generalizing the mean curvature around the point using a multi-dimensional histogram of values. This highly dimensional hyperspace provides an informative signature for the feature representation, is invariant to the 6D pose of the underlying surface, and copes very well with different sampling densities or noise levels present in the neighborhood [23].

The neighborhood $\mathcal{P}^{k_2}$ used in PFH has larger neighborhood than the neighborhood $\mathcal{P}^k$ used for estimating the surface normals. That is, it is more points in the neighborhood $\mathcal{P}^{k_2}$ than $\mathcal{P}^k$.

PFH represents the relationship between the points in $\mathcal{P}^{k_2}$ and their estimated surface normals.

In Chapter 4.2.2 it was described how to estimate a surface normal n for a point p. Consider two points p_i and p_j in the neighborhood $\mathcal{P}^{k_2}$. To compute the relative difference between two points p_i and p_j and their surface normals n_i and n_j, at either p_i or p_j, a fixed *Darboux coordinate frame* is defined. This is shown in Figure 4.18, where

$$p_s = p_i \quad , \quad n_s = n_i \quad , \quad p_t = n_j \quad , \quad n_t = n_j$$

or

$$p_s = p_j \quad , \quad n_s = n_j \quad , \quad p_t = n_i \quad , \quad n_t = n_i$$

The point p_s is chosen such that the angle between its associated normal n_s and the line connecting the p_i and p_j is minimal.

The Darboux coordinate frame has the axes U, V and W where:

$$U = n_s \tag{4.17}$$

$$V = U \times \frac{(p_t - p_s)}{d} \tag{4.18}$$

$$W = U \times V \tag{4.19}$$

where $d = \|p_t - p_s\|_2$ is the Euclidean distance between p_s and p_t. [23].

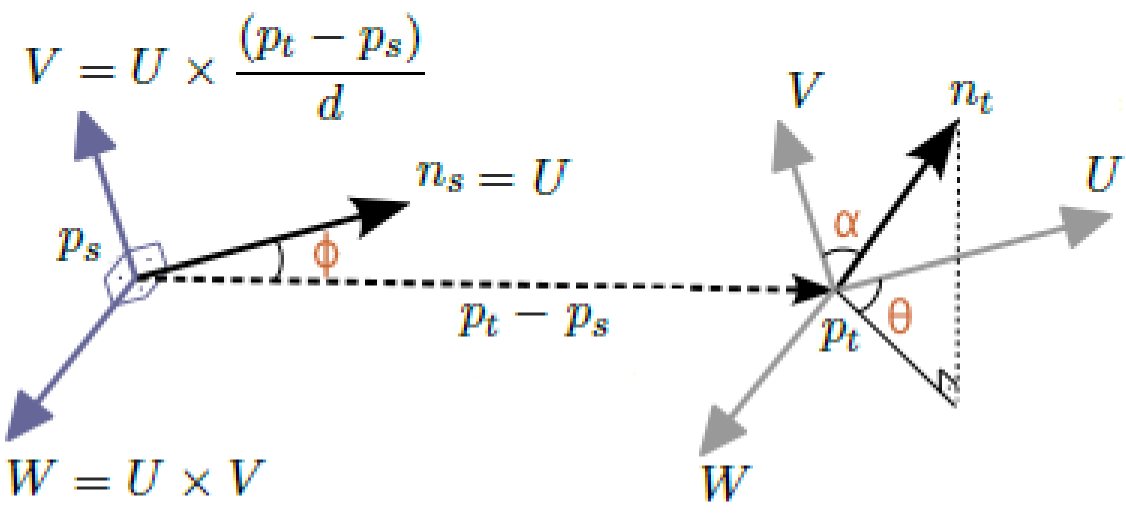


Figure 4.18: A graphical representation of the Darboux frame and the angular PFH features for a pair of points p_s and p_t with their associated normals n_s and n_t [23].

The angular variation of the two normals n_s and n_t can be expressed as follows

$$\begin{aligned} \alpha &= V \cdot n_t \\ \phi &= U \cdot \frac{(p_t - p_s)}{d} \\ \theta &= \tan^{-1}(W \cdot n_t, U \cdot n_t) \end{aligned} \tag{4.20}$$

For each pair of points p_i and p_j in the neighborhood $\mathcal{P}^{k_2}$, the angular variations α, ϕ and θ of their normals and the euclidean distance between those points d are computed. This gives the quadruplet $\langle \alpha, \phi, \theta, d \rangle$ for each pair of points. This reduces the 12 values (X, Y, Z, n_x, n_y, n_z for each point) of the two points and their normals to four values [23].

Figure 4.19 presents an influence region diagram of the PFH computation for a query point p_q. p_q (red), in the middle of the circle (or sphere in 3D) is interconnected with its k neighbours (blue) in a mesh.

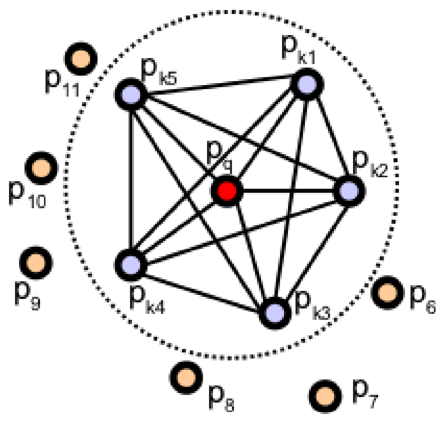


Figure 4.19: The influence region diagram for a Point Feature Histogram. The query point (red) and its k-neighbors (blue) are fully interconnected in a mesh [23].

To create the final PFH representation for the query point p_i, the set of all quadruplets is binned into a histogram. The binning process divides each feature's value range into b subdivisions, and counts the number of occurrences in each sub-interval. Since three out of the four features in the quadruplet are measure of the angles between normals, their values can be normalized to the same interval on the trigonometric circle. A binning example is to divide each feature interval into the same number of equal parts, and therefore create a histogram with b^4 bins in a fully correlated space. In this space, a histogram bin increment corresponds to a point having certain values for all its four features. Figure 4.20 presents examples of Point Feature Histograms representations for different points in a cloud.

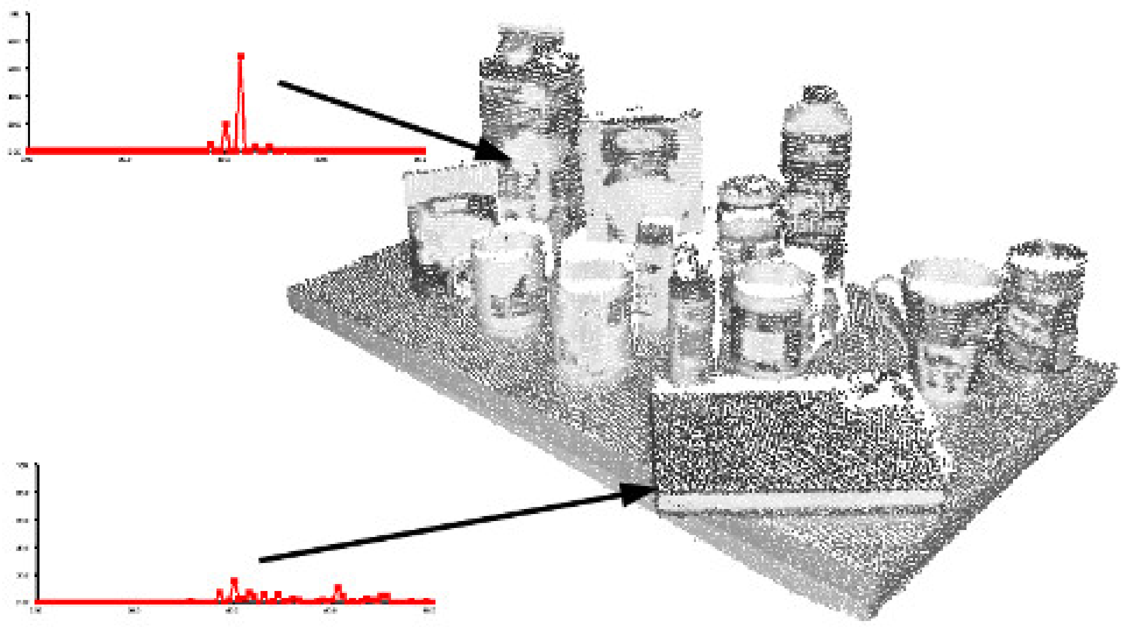


Figure 4.20: Examples of Point Feature Histograms representations for different points in a cloud [23].

In some cases, the fourth feature, d, does not present an extreme significance for 2.5D datasets captured with one RGB-D camera, as the distance between neighboring points increases from the viewpoint. Therefore, omitting d for scans where the local point density influences this feature dimension has proved to be beneficial [23].

PFH can be summarized as follows:

1. Create normals for all points in point cloud P.
2. Estimate features for a point p_i in P: The set of k-neighbours in the radius r around the point p_i (p_{ik}) is taken. Four features are calculated between two points. The corresponding bin is incremented by 1. A point feature histogram (PFH) is generated.
3. The resulting set of histograms can be compared to those of other point clouds in order to find correspondences.

4.2.4 FPFH - Fast Point Feature Histogram

For real-time or near real-time applications, the computation of Point Feature Histograms in dense point neighborhoods can represent one of the major bottlenecks. The theoretical computational complexity of the PFH for a given point cloud P with n points is $O(n \cdot k^2)$, where k is the number of neighbors for each point p in P [23]. Fast point feature histograms (FPFH) is a simplified version of point feature histograms (PFH) that reduces the computational complexity of the algorithm to $O(n \cdot k)$, while still retaining most of the discriminative power of the PFH.

First a set of tuples α, ϕ and θ are computed (in equation (4.20)) between itself and its neighbours, for each query point p_q, as described in PFH. This is called the simplified point feature histogram (SPFH). Next, the k neighbors are re-determined for each point, and the neighboring SPFH values are used to weight the final histogram of p_q (called FPFH) as follows [23]:

$$\mathrm{FPFH(p_q)} = \mathrm{SPFH(p_q)} + \frac{1}{\mathrm{k}} \sum_{\mathrm{i=1}}^{\mathrm{k}} \frac{1}{\omega_\mathrm{k}} \cdot \mathrm{SPFH(p_k)} \tag{4.21}$$

where the weight ω_k represents a distance between the query point p_q and a neighbor point p_k in some given metric space. Figure 4.21 presents the influence region diagram for a k-neighborhood set centered at p_q.

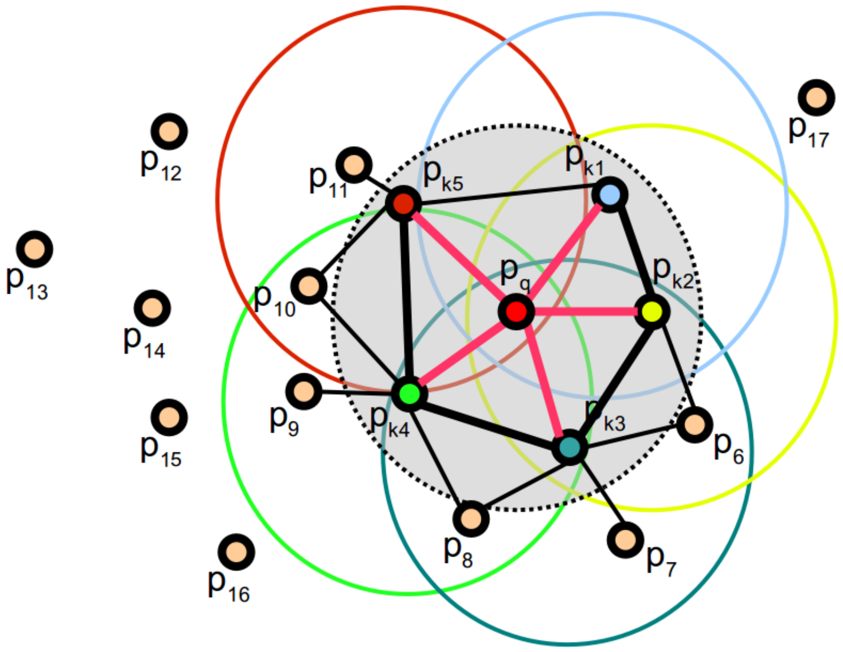


Figure 4.21: The influence region diagram for a Fast Point Feature Histogram. Each query point (red) is connected only to its direct k-neighbors (enclosed by the gray circle). Each direct neighbor is connected to its own neighbors and the resulted histograms are weighted together with the histogram of the query point to form the FPFH. The connections marked with thicker lines will contribute to the FPFH twice [23].

The algorithm first estimates its SPFH values for a given query point p_q by creating pairs between itself and its neighbors (illustrated using red lines). This is repeated for all the points in the dataset, followed by a re-weighting of the SPFH values of p_q using the SPFH values of its p_k neighbors. This creates the FPFH for p_q [23]. The extra FPFH connections, as a result from the additional weighting scheme, are shown with black lines. The connections marked with thicker lines will be counted twice.

FPFH can be summarized as follows:

1. Create normals for all points in P
2. Estimate features for a point p_i in P: The set of k-neighbours in the radius r around the point p_i (p_{ik}) is taken. Three features are calculated between two points (only between p_i and its neighbours). The corresponding bin is incremented by 1. A simple point feature histogram (SPFH) is generated.
3. To reach more points and connections (up to 2 times r) the SPFH of the neighbours are added weighted according to their spatial distance as a last step.
4. The resulting set of histograms can be compared to those of other point clouds in order to find correspondences.

4.3 Program Code for Depth Camera

This section explains the principle behind some of the program code used on the data obtained by the depth camera.

4.3.1 Aligning Object Templates to a Point Cloud

To be able to get the pose of an object with respect to a known coordinate frame, some of the example code from Point Cloud Library (PCL) called template_alignment was used [18]. Template_alignment will, as the name suggests, be used to perform template alignment (also referred to as template fitting/matching/registration). A template is typically a small group of pixels or points that represents a known part of a larger object or scene. The pose of the object in a scene, can be found by registering a template (reference cloud) to a new point cloud (target cloud). That is, the program estimates a transformation T_t^r from the pose of the object in the reference cloud to the objects' pose in the target cloud.

How the program works

The code reads the cloud from the camera and then computes the cloud's surface normals. To do so, the radius r_n that defines each point's neighborhood must be specified. Then, the feature search radius r_f is specified so the FPFH-descriptors can be computed.

Once all the descriptors of the two point clouds (target and reference) have been computed, a particular version of the RANSAC algorithm (RANdom SAmple Consensus) is used to find a raw alignment between the clouds [29]. This version is called SAC-IA (SAmple Consensus - Initial Alignment) and takes a template as input and aligns it to the target cloud (i.e., the cloud to which the templates will be aligned).

SAC-IA tries to maintain the same geometric relations of the correspondences as the greedy initial alignment method described in [25], without having to try all combinations of a limited set of correspondences (corresponding points). Instead, a large number of correspondence candidates are sampled in a sample s. The SAC-IA computes the transformation T_t^r defined by the sample points and their correspondences and compute an error metric for the point cloud that computes the quality of the transformation.

The SAC-IA takes three parameters:

- The minimum distances d_{min} between samples. That is, the s samples are selected only if their pairwise distances are greater than d_{min}.
- The maximum distance threshold c_{max} between two corresponding points in reference- and target cloud. If the correspondences have larger distances than c_{max}, the points will be ignored in the alignment process.
- The maximum number of iterations the internal optimization should run for.

The given template is set as the SAC-IA algorithm's source cloud and then it aligns the source to the target [24].

4.3.2 Region Growing Segmentation

The purpose of the segmentation is to group points with similar features into segments. Segments are smooth surfaces that are achieved by grouping neighboring points with similarity measures, such as the direction of a locally estimated surface normal.

Point clouds can be segmented into multiple surfaces. The surface growing algorithm starts from the optimal seed points and extend surfaces to neighboring points based on a pre-defined criteria [13]. Selection of the seed point is an important step and the final segmentation results depend on it. First a plane equation, as shown in Figure 4.22, is defined for each seed point and its neighboring points in order to find the optimal seed point. Then, the residuals (orthogonal distances of the point clouds to the best fitted plane) are computed [13]. Figure 4.23 shows a representation of surface normals and segmentation parameters. The point within the fitted plane with lowest square sum of residuals is considered the optimal seed point.

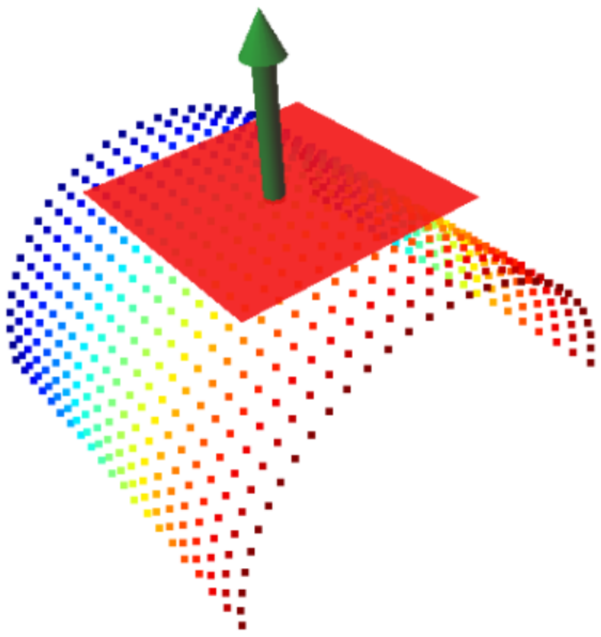


Figure 4.22: Normal estimation by fitting a plane to the points in the neighborhood [19].

The candidate point with a orthogonal distance to the fitted plane (residual) which is below the predefined threshold is accepted as a new surface point. To increase the efficiency of the program while using low accurate point clouds, the plane equation is updated after adding the new candidate point to the corresponding surface points. The neighborhood threshold k and residual threshold r_{th} are used to determine the smoothness of the fitted plane [13]. Secondly, the local surface normal at each point is compared with its neighboring points. The neighboring points are accepted if the angle between the surface normal and normal at the neighboring point is below the pre-defined threshold θ_{th}.

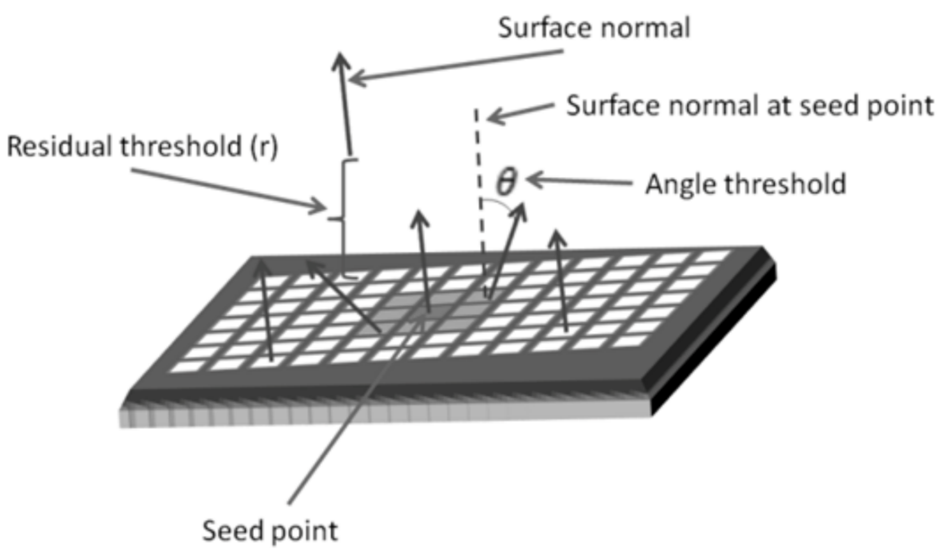


Figure 4.23: Representation of surface normals and segmentation parameters [13].

The process of region growing proceeds in the following steps [19]:

1. Specify a residual threshold r_{th}.
2. Define a smoothness threshold θ_{th} in terms of the angle between the normals of the current seed and its neighbors.
3. If all the points already have been segmented go to step 7. Otherwise select the point with the minimum residual as the current seed.
4. Select the neighboring points of the current seed by using k-nearest neighbours with specified parameters. Add the points that have normals with smaller angle than the angle threshold to the current region. Add the points whose residuals are less than r_{th} to the list of potential seed points.
5. If the potential seed point list is not empty, set the current seed to the next available seed, and go to step 4.
6. Add the current region to the segmentation and go to step 3.
7. Return the segmentation result.

A segmented point cloud of the cylinder to the left and the piston to the right is shown in Figure 4.24.

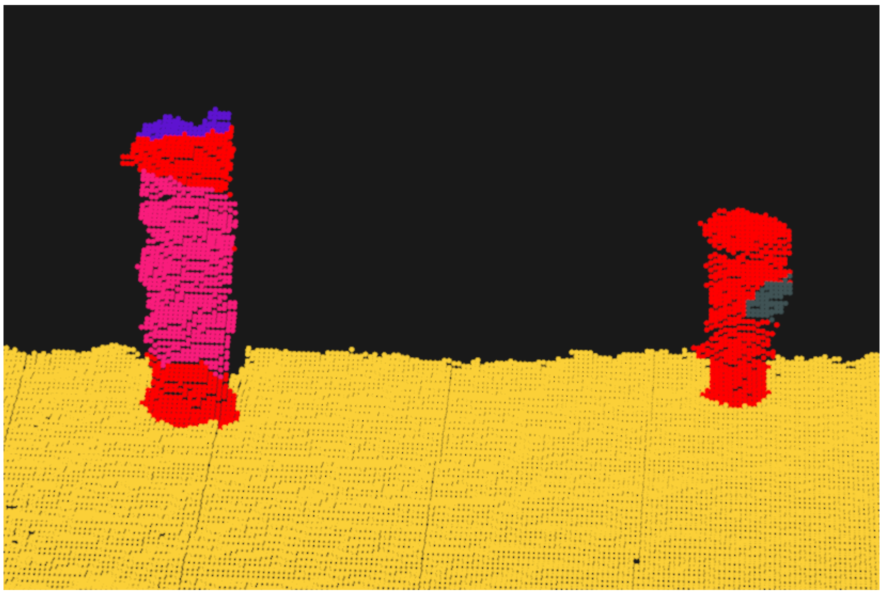


Figure 4.24: A segmented point cloud.

The segment grown on the cylinder is colored pink and the segment grown on the piston is colored dark grey. There is also a purple segment on the top of the cylinder. The color of the segments is randomly generated every time the viewer is run, but the unsegmented points are always red. The table, colored yellow, is separated from the objects by the difference of the angle between the surface normals. If a point's normal has a larger angle difference than the specified threshold θ_{th}, the points are not segmented. The flat property of the table makes it a good fit for the plane equation giving the points on the table low residual values. The piston on the other hand is not very well segmented since the plane equation is not idle for the strong curvature of the piston. The growing region starts its growth where the residuals have the lowest value. For a cylinder shaped object the residual value is high near the top edge and the bottom edge of the object. The segmentation for the piston does not always start from the right side as shown in Figure 4.24, but seed points will always start its growth from somewhere in the middle of the height of the piston if the parameters stays the same. Figure 4.25 shows a segmented point cloud with the same parameters.

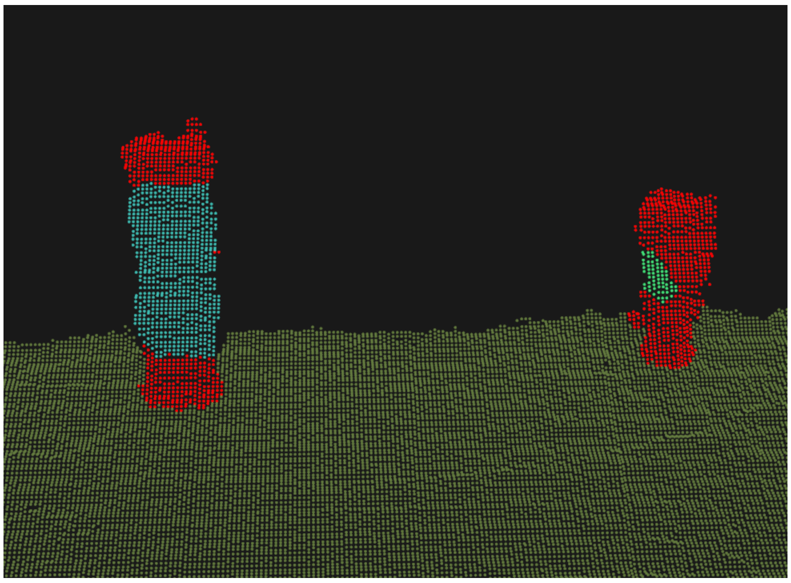


Figure 4.25: A segmented point cloud.

Here the segmentation have started its growth from the left side on the piston. A better segmentation of the piston can be obtained by increasing the residual threshold as shown in Figure 4.26.

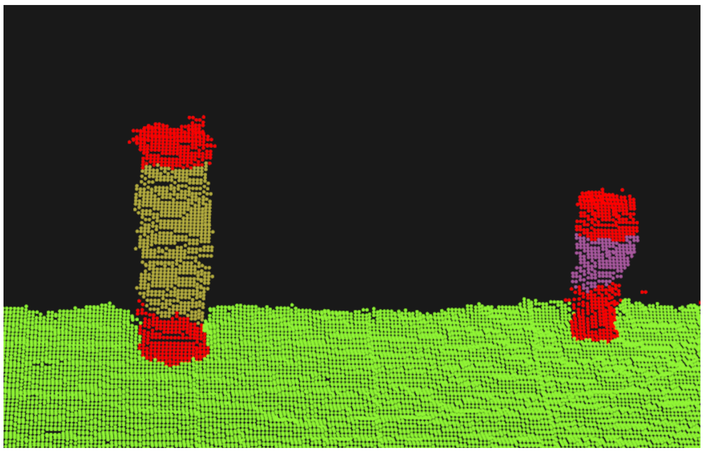


Figure 4.26: A segmented point cloud.

Here the residual threshold is higher so more seed points are found on the piston.

If the k-nearest neighbours is increased, the the segmented area also increases. Figure 4.27 shows how the point cloud is segmented when searching in a larger neighbourhood.

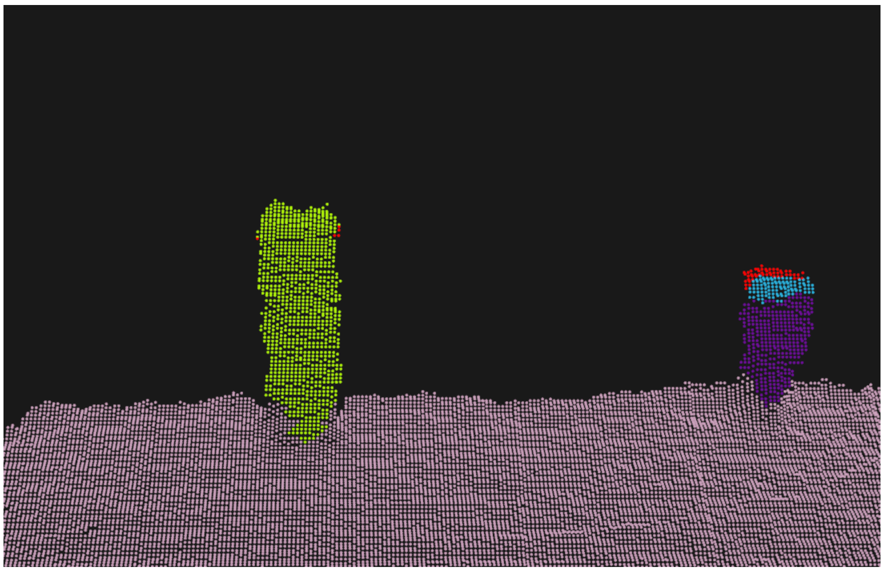


Figure 4.27: A segmented point cloud.

Searching in a larger neighbourhood increases the chance of finding an acceptable seed point to grow from.

4.3.3 Statistical Removal of Outliers

Noisy points in the point cloud complicates the estimation of local point cloud characteristics such as surface normals or curvature changes, leading to erroneous values. Some of these irregularities can be solved by performing a statistical analysis on each point's neighborhood, and trimming those which do not meet a certain criteria.

Luckily PCL [18] has developed a filter that removes sparse outliers. The filter is based on the computation of the distribution of point to neighbors distances in the input dataset. The mean distance from a point to all its neighbours is computed for each point. By assuming that the resulted distribution is Gaussian with a mean and a standard deviation, all points whose mean distances are outside an interval defined by the global distances mean and standard deviation can be considered as outliers and trimmed from the dataset.

In this thesis the filter was implemented in the following way. First the excess points (Figure 4.29) are removed from the point cloud (Figure 4.28), leaving only the points of interest (Figure 4.30) in the cloud.

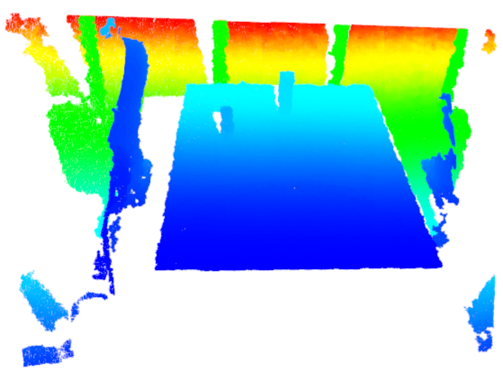


Figure 4.28: Unfiltered point cloud.

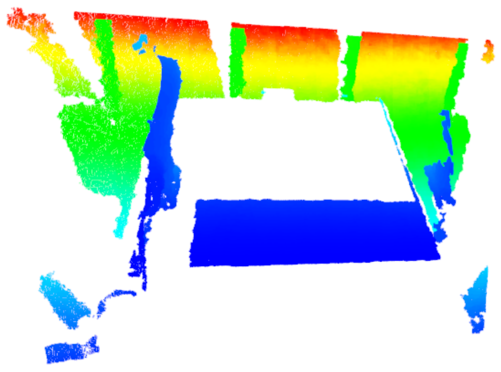


Figure 4.29: Removed part of point cloud.

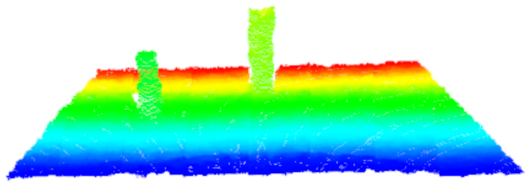


Figure 4.30: Remaining part of point cloud.

Figure 4.31 shows a side view of the cloud in Figure 4.30. By running the statistical outlier removal filter on this cloud, noisy points can be removed from the cloud, leaving only the cloud in Figure 4.33. The trimmed outliers are shown in Figure 4.32.

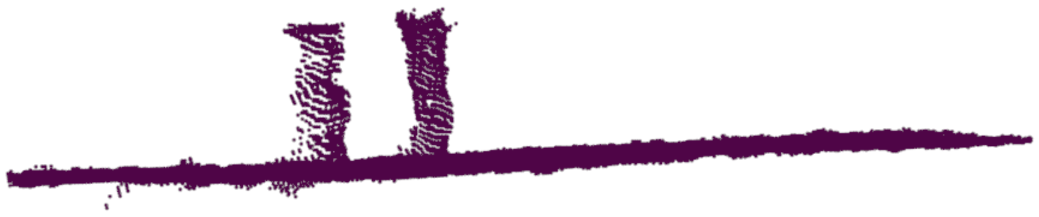


Figure 4.31: Side view of the point cloud.

Figure 4.32: Noisy points that are trimmed away by the filter.

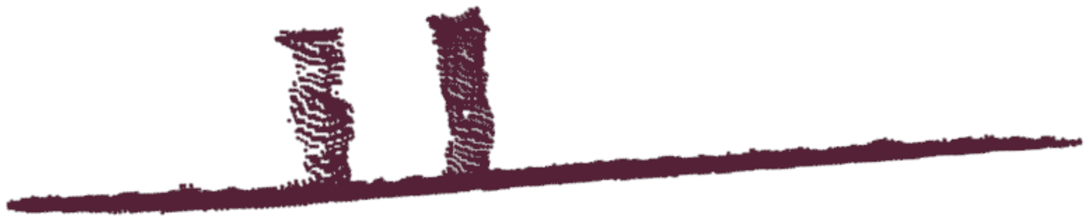


Figure 4.33: Filtered point cloud.

Chapter 5

Computer Vision Experiments and Results

5.1 Experiments based on the RGB-D Camera

5.1.1 Calibration of AR-tag Coordinate Frame

This section explains how the coordinate frame of the camera is transformed on to the table. Three AR-tags, ID0, ID2 and ID3 were placed on the table 90 degrees to each other, forming an X Y plane aligned with the table as shown in Figure 5.1. The Z axis has its origin in the AR-tag in the middle (ID3) and is set to be orthogonal to the X Y plane.

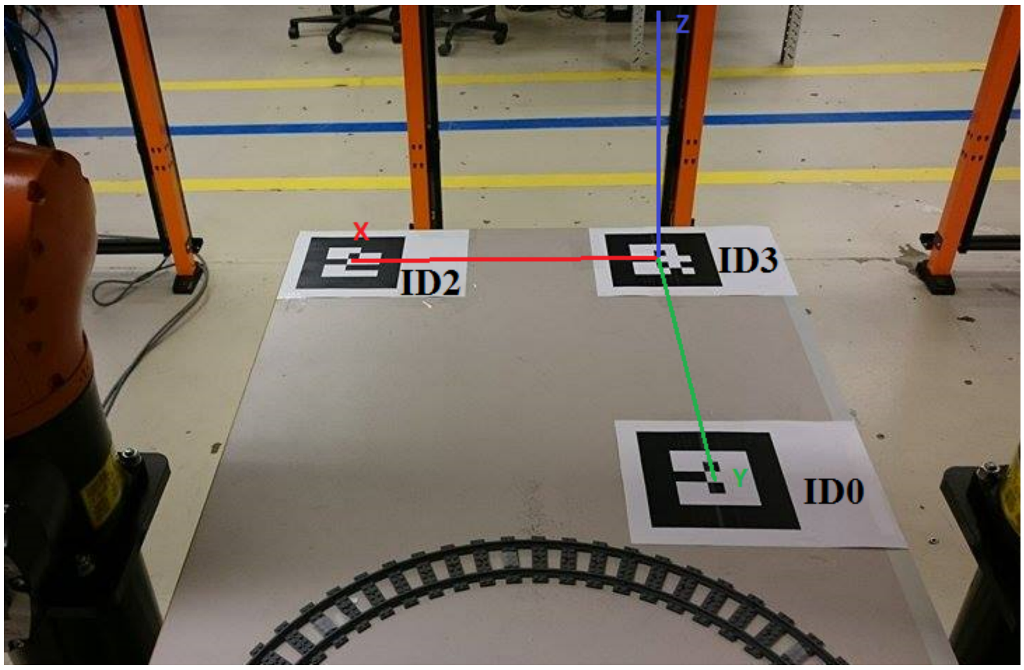


Figure 5.1: X Y plane is defined between the AR-tags. X axis is red, Y axis is green and Z axis is blue.

The center points $p_i = (x_i, y_i, z_i)^T$ of each AR-tag IDi, $i \in \{0, 2, 3\}$ were used in the estimation of transformation T_a^c from camera to the AR-tags on the table. A problem that occurred was that the coordinates x_i, y_i and z_i of the AR-tags in the image stream moved randomly with an amplitude of up to 5 millimeters for each frame the camera captured. This was not acceptable because the work to be done by the robot required high accuracy. Therefore 50 sample points were taken for each of the coordinates x_i, y_i and z_i for each AR-tag. The median of the samples was used to determine the center points p_i for each AR-tag IDi, $i \in \{0, 2, 3\}$.

Figure 5.2, Figure 5.3 and Figure 5.4 shows the difference between the coordinates x_3, y_3 and z_3 of the AR-tag ID3 fed directly from the camera and the median of 50 frame samples, for x_3, y_3 and z_3 respectively. The red curve shows the live coordinate value from the camera and the blue curve shows the improved coordinate value, the median of 50 samples.

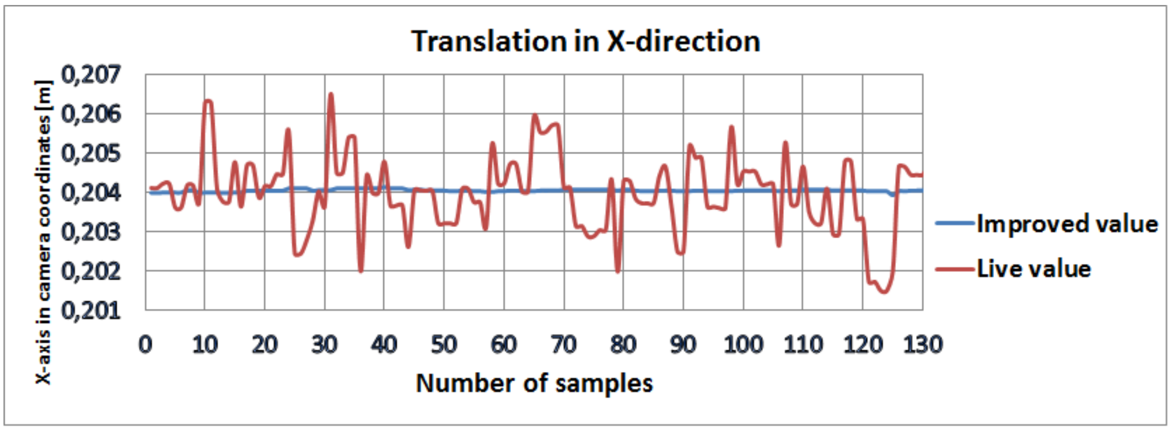


Figure 5.2: The difference between the live value and the improved value of the x_3 coordinate, 1.6 m from the camera.

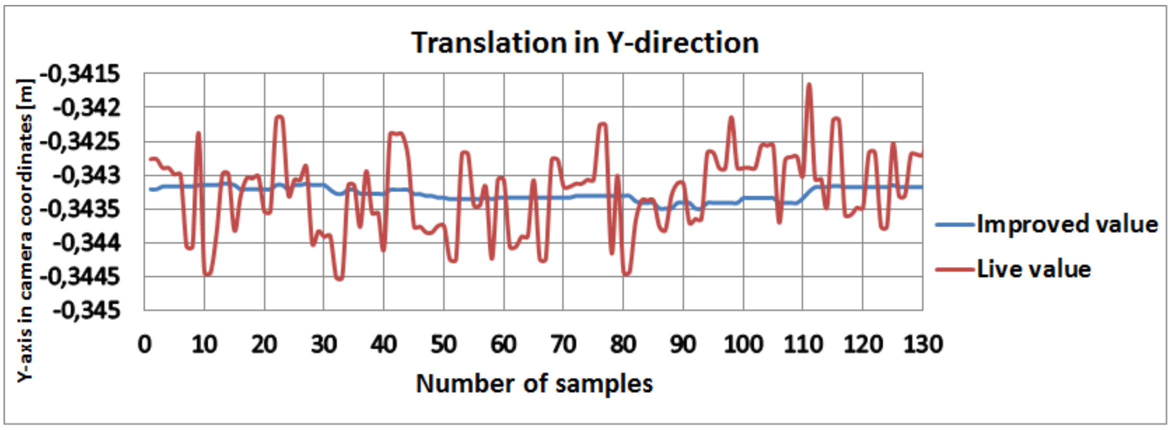


Figure 5.3: The difference between the live value and the improved value of the y_3 coordinate, 1.6 m from the camera.

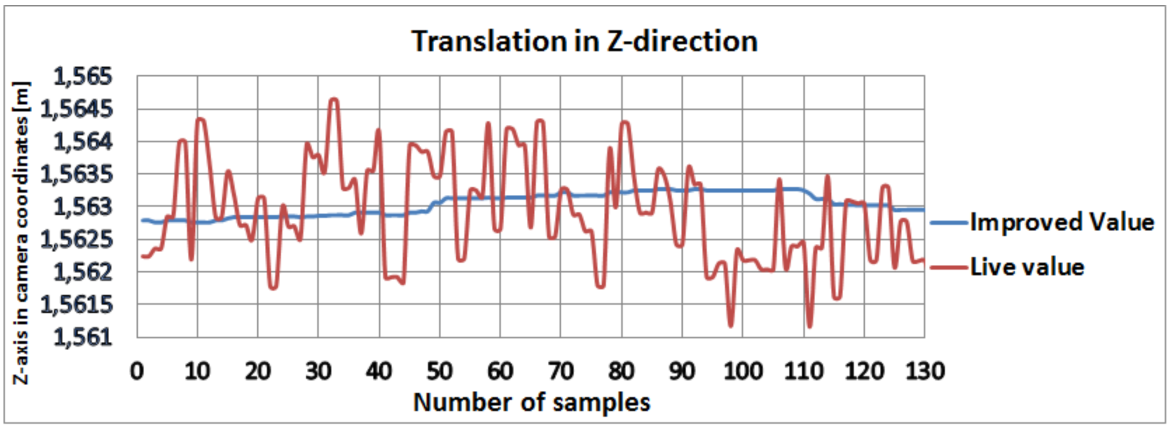


Figure 5.4: The difference between the live value and the improved value of the z_3 coordinate, 1.6 m from the camera.

The graphs shows that the improved value is much more stable and accurate than the live value.

After the improved coordinates are computed, the program defines the axes of the coordinate frame. First, a unit vector X is defined along the X axis from $p_3 = (x_3, y_3, z_3)^{\mathrm{T}}$ (AR-tag ID3) to $p_2 = (x_2, y_2, z_2)^{\mathrm{T}}$ (AR-tag ID2):

$$X = \frac{p_2 - p_3}{\sqrt{(x_2 - x_3)^2 + (y_2 - y_3)^2 + (z_2 - z_3)^2}} \tag{5.1}$$

Next, a unit vector Y is defined from $p_3 = (x_3, y_3, z_3)^{\mathrm{T}}$ (AR-tag ID3) to $p_0 = (x_0, y_0, z_0)^{\mathrm{T}}$ (AR-tag ID0):

$$Y = \frac{p_0 - p_3}{\sqrt{(x_0 - x_3)^2 + (y_0 - y_3)^2 + (z_0 - z_3)^2}} \tag{5.2}$$

And lastly a unit vector Z is calculated as the cross product between X and Y, rising orthonormal upwards from the X Y plane:

$$Z = X \times Y \tag{5.3}$$

The three unit vectors X, Y and Z is than inserted into a 3×3 rotation matrix like so:

$$R = \begin{bmatrix} X & Y & Z \end{bmatrix} \tag{5.4}$$

Next, the program repeatedly checks if the determinant of the rotation matrix R is 1, which means that the X, Y and Z axes are orthonormal to each other. When R is 1 ($1 > R > 0.9999$), it is inserted into a homogeneous transformation matrix T_a^c, which can be used to transform the coordinate frame of the camera c to the AR-tag a.

$$T_a^c = \begin{bmatrix} R & p_3 \\ 000 & 1 \end{bmatrix} \tag{5.5}$$

The center point of ID3, $p_3 = (x_3, y_3, z_3)^T$, is the origin of the coordinate frame.

For the 50 sample points of each of the coordinates x_i, y_i and z_i value for each AR-tag, the mode of each sample was also used to make a rotation matrix. Three rotation matrices were estimated, one with the median of the samples, one with the mode of the samples and one with live values from x_i, y_i and z_i value for each AR-tag. Figure 5.5 shows the result of the determinant values of these rotation matrices.

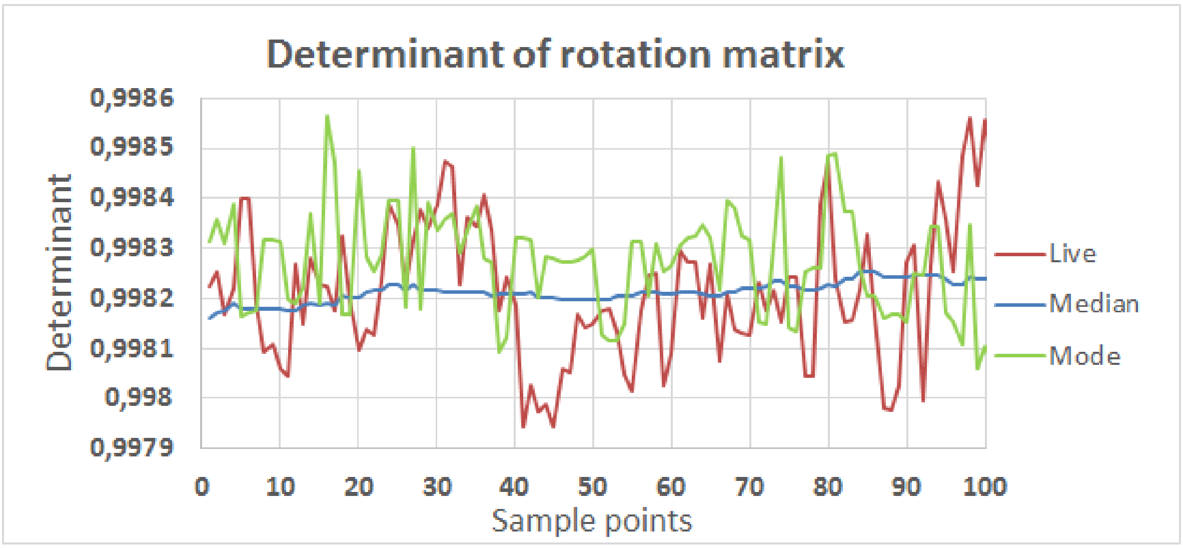


Figure 5.5: The determinant values of the rotation matrix acquired from AR-tag 1.60 m from the camera.

The first rotation matrix to obtain a determinant value between 0.9999 and 1 from either the median- or the mode-value is used in the transformation. In the diagram in Figure 5.5 none of the values would be approved as they are not 0.9999. In order to get the determinant to become 0.9999, one AR-tag had to be moved more than one centimeter away from perpendicular position to the other tags. This was not acceptable, so further experiments had to be conducted.

To see if the accuracy would increased if the AR-tags were closer to the camera, another experiment was conducted. The AR-tags were positioned 0.45 m closer to the camera as shown in Figure 5.6.

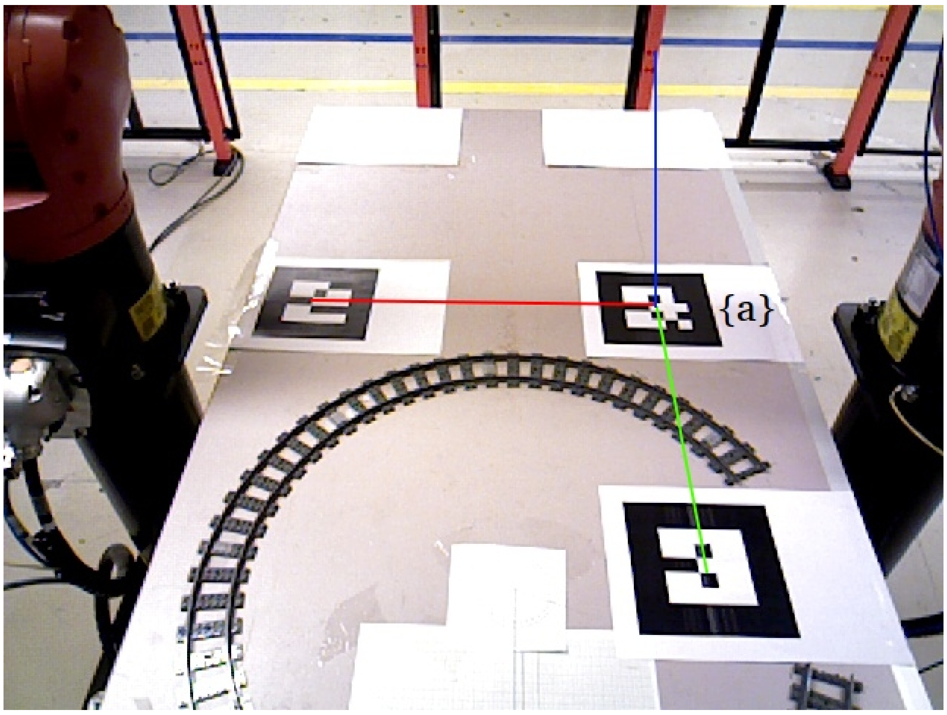


Figure 5.6: AR-tag is 1.15 m from the camera.

The translation p_3 in T_a^c is shown in Figure 5.7, Figure 5.8 and Figure 5.9 for the x_3, y_3 and z_3 coordinate respectively.

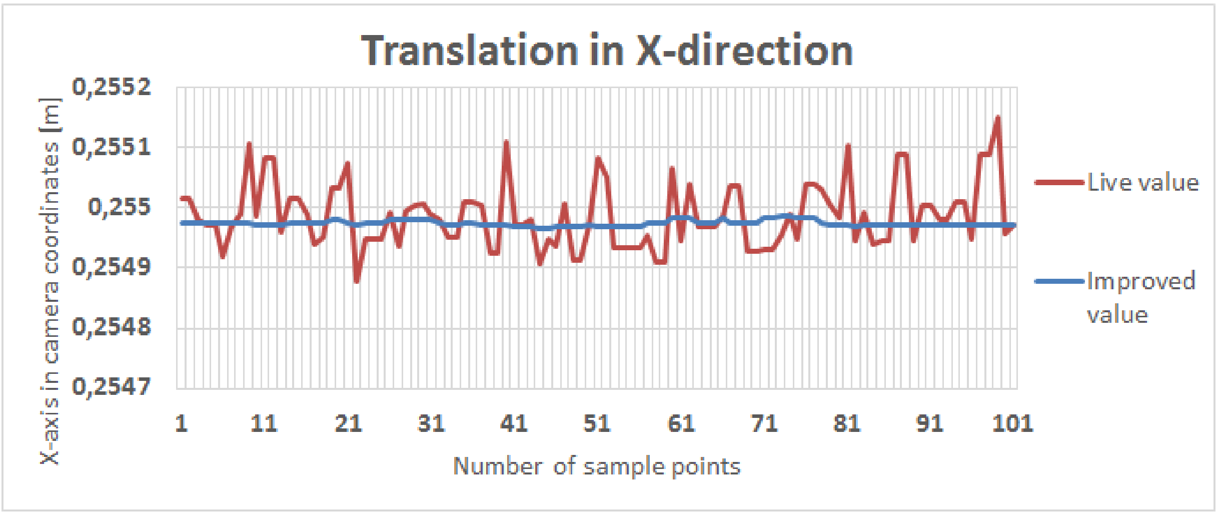


Figure 5.7: The difference between the live value and the improved value of the x_3 coordinate, 1.15 m from the camera.

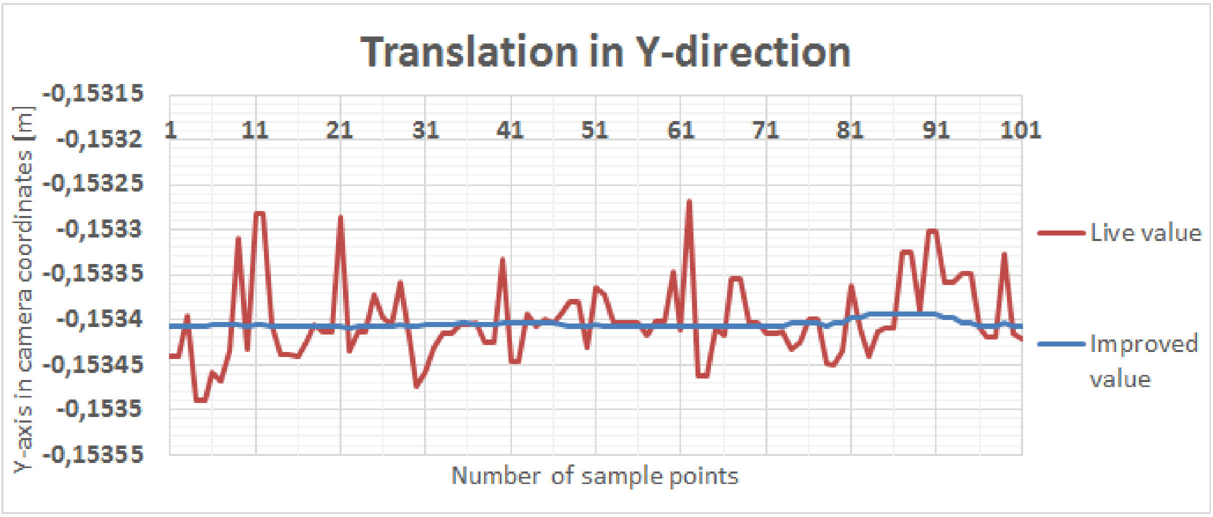


Figure 5.8: The difference between the live value and the improved value of the y_3 coordinate, 1.15 m from the camera.

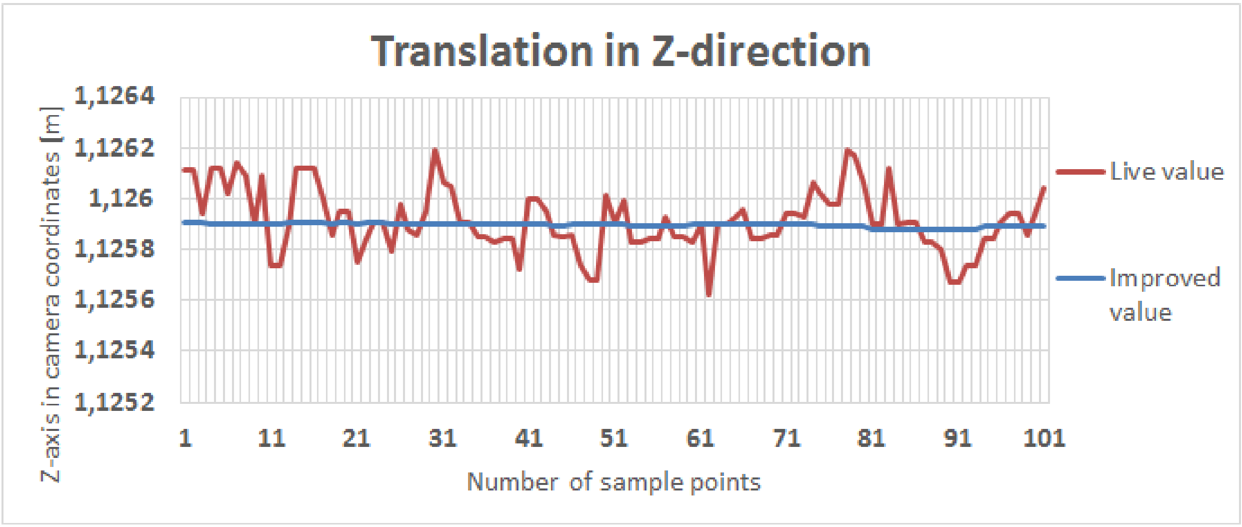


Figure 5.9: The difference between the live value and the improved value of the z_3 coordinate, 1.15 m from the camera.

Figure 5.10 shows the difference between the maximum and minimum value for the new translation p_3 approximately 0.45 m closer to the camera than the original translation.

New	X-live	Y-live	Z-live	X-improved	Y-Improved	Z-improved
Max	0.255151	-0.15327	1.12619	0.254986	-0.15339	1.12591
Min	0.254879	-0.15349	1.12562	0.254967	-0.15341	1.12558
Max – Min	0.000272	0.00022	0.00057	0.000019	0.00002	0.00033

Figure 5.10: The variation of the highest value to the lowest value for the new translation 1.15 m from the camera.

Compared to the original translation, shown in Figure 5.11, the accuracy has improved greatly.

Original	X-live	Y-live	Z-live	X-improved	Y-Improved	Z-improved
Max	0.206503	-0.34214	1.56462	0.20414	-0.343126	1.56327
Min	0.201989	-0.344519	1.56117	0.203991	-0.343494	1.56276
Max – Min	0.004514	0.002379	0.00345	0.000149	0.000368	0.00051

Figure 5.11: The variation of the highest value to the lowest value for the original translation 1.60 m from the camera.

The difference between the original and the new translation is shown in Figure 5.12.

	X-live	Y-live	Z-live	X-improved	Y-Improved	Z-improved
Original variation	0.004514	0.002379	0.00345	0.000149	0.000368	0.00051
New variation	0.000272	0.00022	0.00057	0.000019	0.00002	0.00033
Difference	0.004242	0.002159	0.00288	0.00013	0.000348	0.00018

Figure 5.12: The difference between the original and the new translation.

This shows that better accuracy can be achieved by moving the camera closer to the AR-tag.

The resulting values of the determinant of the new rotation matrix acquired 1.15 m from the camera is shown in the diagram in Figure 5.13.

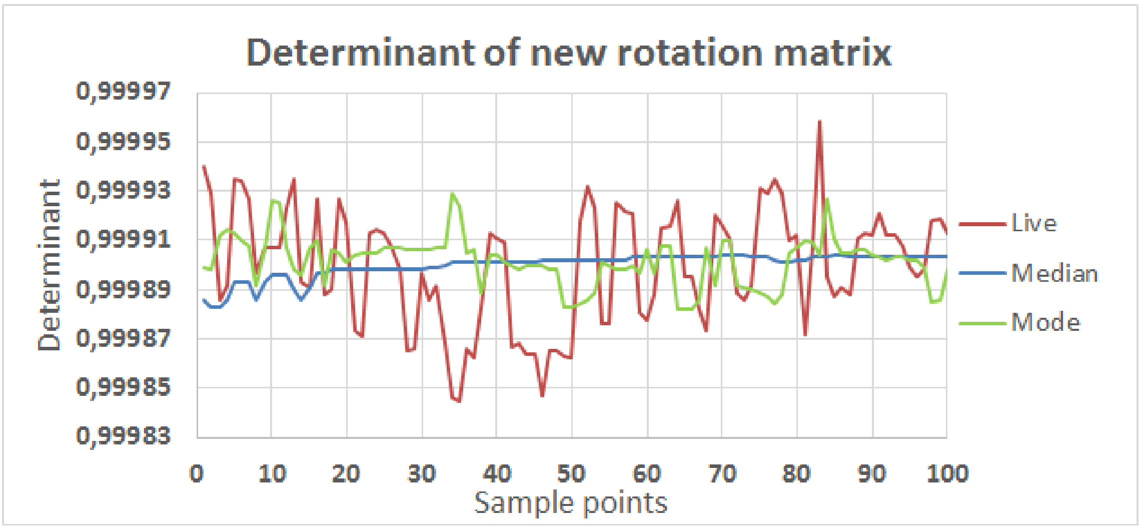


Figure 5.13: The determinant values of the new rotation matrix acquired from an AR-tag 1.15 m from the camera.

The new rotation matrix produced better results than the rotation matrix acquired from 1.60 m from the camera and here the AR-tags were perpendicular to each other.

This result gives us an efficient method to calibrate the camera to a flat surface, and if the camera is moved, all it takes is for the three AR-tags to be in the field of view of the camera. The computation process takes about 25 seconds, but it is only needed once per camera position.

Calibration of AR-tag-Coordinate Frame using Geometric Algebra

A unit vector X (normalized vector from p_3 to p_2) was estimated to be

$$X = (-0.9995, 0.0006, 0.0309)$$

and a unit vector Y (normalized vector from p_3 to p_0) was estimated to be

$$Y = (-0.0120, 0.6838, -0.7296)$$

Defining a bivector $X \wedge Y$ aligned with the table, where

$$\begin{aligned} X &= x_1e_1 + x_2e_2 + x_3e_3 \\ &= -0.9995e_1 + 0.0006e_2 + 0.0309e_3 \end{aligned}$$

and

$$\begin{aligned} Y &= y_1e_1 + y_2e_2 + y_3e_3 \\ &= -0.0120e_1 + 0.6838e_2 - 0.7296e_3 \end{aligned}$$

is

$$X \wedge Y = (x_1e_1 + x_2e_2 + x_3e_3) \wedge (y_1e_1 + y_2e_2 + y_3e_3) = -0.0216e_{23} - 0.73e_{31} - 0.683e_{12}$$

Determine a new Y axis Y_{new} with use of the vector X and bivector $X \wedge Y$ is done by

$$\begin{aligned} Y_{new} &= X \cdot (X \wedge Y) \\ &= (-0.9995e_1 + 0.0006e_2 + 0.0309e_3) \cdot (-0.0216e_{23} - 0.73e_{31} - 0.683e_{12}) \\ &= -0.0221e_1 + 0.684e_2 + 0.729e_3 \end{aligned}$$

were Y_{new} is X rotated 90° anti-clockwise in the bivector $X \wedge Y$.

The new Z axis Z_{new} is then the normal to $X \wedge Y$

$$\begin{aligned} Z_{new} &= (X \wedge Y)^* \\ &= (X \wedge Y)I^{-1} \quad \text{where} \quad I^{-1} = e_3 \wedge e_2 \wedge e_1 \\ &= (-0.0216e_{23} - 0.73e_{31} - 0.683e_{12})e_{321} \\ &= -0.0216e_1 - 0.73e_2 - 0.683e_3 \end{aligned}$$

The AR-tag ID3, center point p_3, was located at

$$p_3 = (0.2293, 0.0011, 1.2589)^{\mathrm{T}}$$

The transformation matrix T_a^c based on geometric algebra is

$$T_a^c = \begin{bmatrix} X^{\mathrm{T}} & Y_{new}^{\mathrm{T}} & Z_{new}^{\mathrm{T}} & p_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_a^c = \begin{bmatrix} -0.9995 & -0.0221 & -0.0216 & 0.2293 \\ 0.0006 & 0.684 & -0.73 & 0.0011 \\ 0.0309 & 0.729 & -0.683 & 1.2589 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

A summary of the geometric algebra calculations is illustrated in Figure 5.14.

Since geometric algebra let us rotate a vector 90° in a plane, the AR-tag ID0 does not necessarily have to be perpendicular to AR-tag ID3. The only information needed is an X axis defined with ID2 and ID3, as well as ID0 somewhere on the plane of the table.

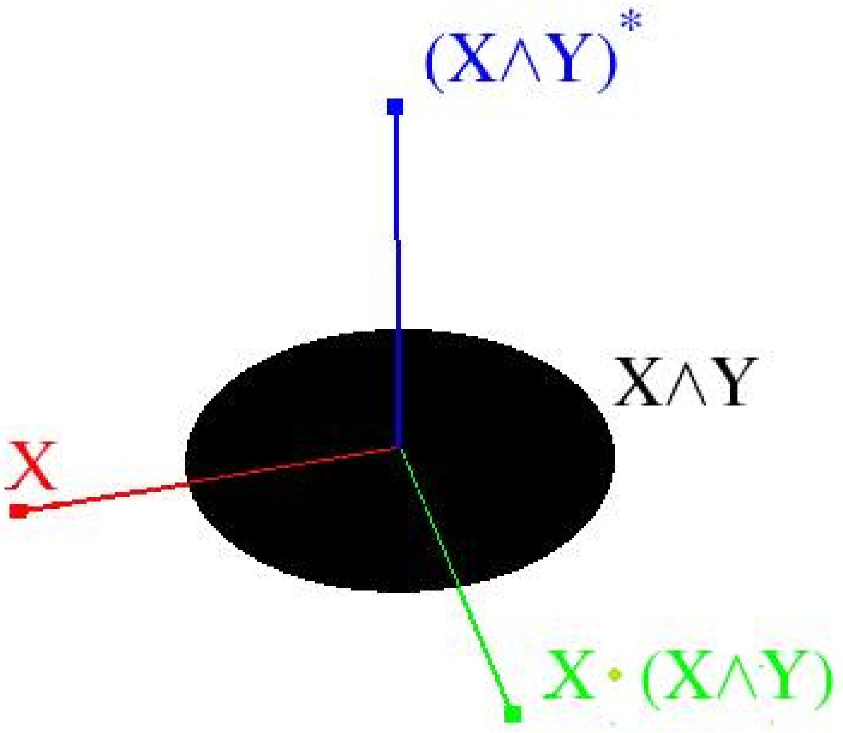


Figure 5.14: Coordinate system {a} estimated with geometric algebra.

5.1.2 Calibration of the Table Coordinate Frame

Transformation from the depth camera to the AR-tags T_a^c has already been described in Chapter 5.1.1. Transformation from the AR-tags $\{a\}$ to the corner of table $\{t\}$, T_t^a, is

$$T_t^a = \begin{bmatrix} 1 & 0 & 0 & -0.1905 \\ 0 & 1 & 0 & -0.604 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.6}$$

and the transformation from the table $\{t\}$ to the RGB-D camera T_c^t is

$$T_c^t = (T_t^a)^{-1}(T_a^c)^{-1} \tag{5.7}$$

In Figure 5.15 and Figure 5.16 the different poses $\{a\}$ and $\{t\}$ are shown.

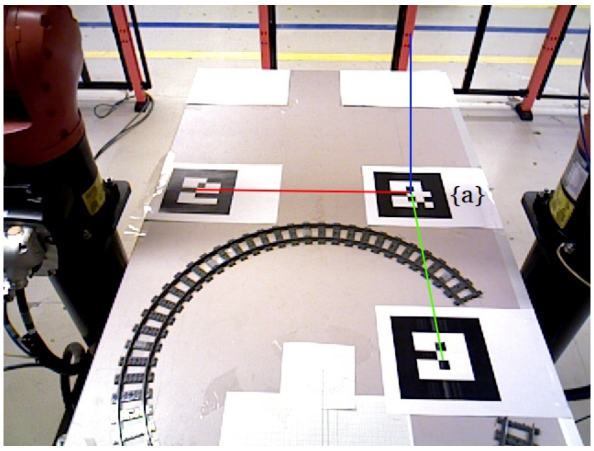


Figure 5.15: The coordinate system $\{a\}$.

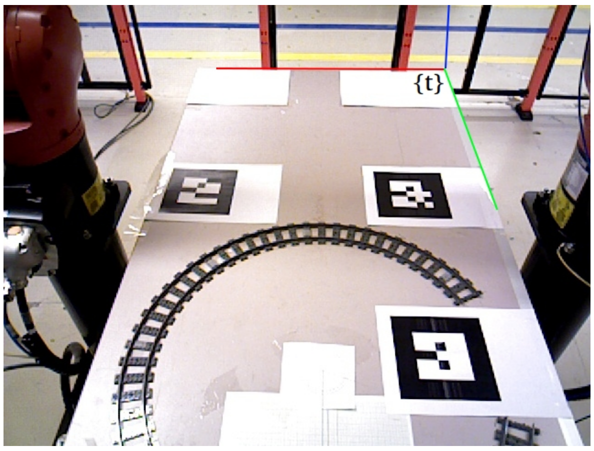


Figure 5.16: The coordinate system $\{t\}$.

The points p^c in the point cloud from the depth camera are transformed to points p^t given with respect to the table coordinate frame $\{t\}$ like so

$$\underbrace{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}}_{\tilde{p}^t} = T_c^t \underbrace{\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}}_{\tilde{p}^c} \tag{5.8}$$

In Figure 5.17 the process so far is shown.

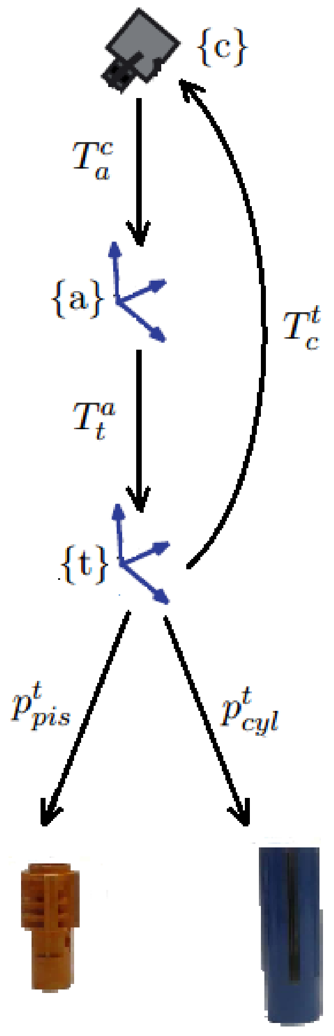


Figure 5.17: Transformation from depth camera $\{c\}$ to the cylinder and the piston.

In Figure 5.18 the pose of the table frame $\{t\}$ is shown. Even if the default transformation $T_{\text{ir}}^{\text{rgb}}$ from IR camera to RGB camera was used, the final pose of table $\{t\}$ is correct as shown in Figure 5.18.

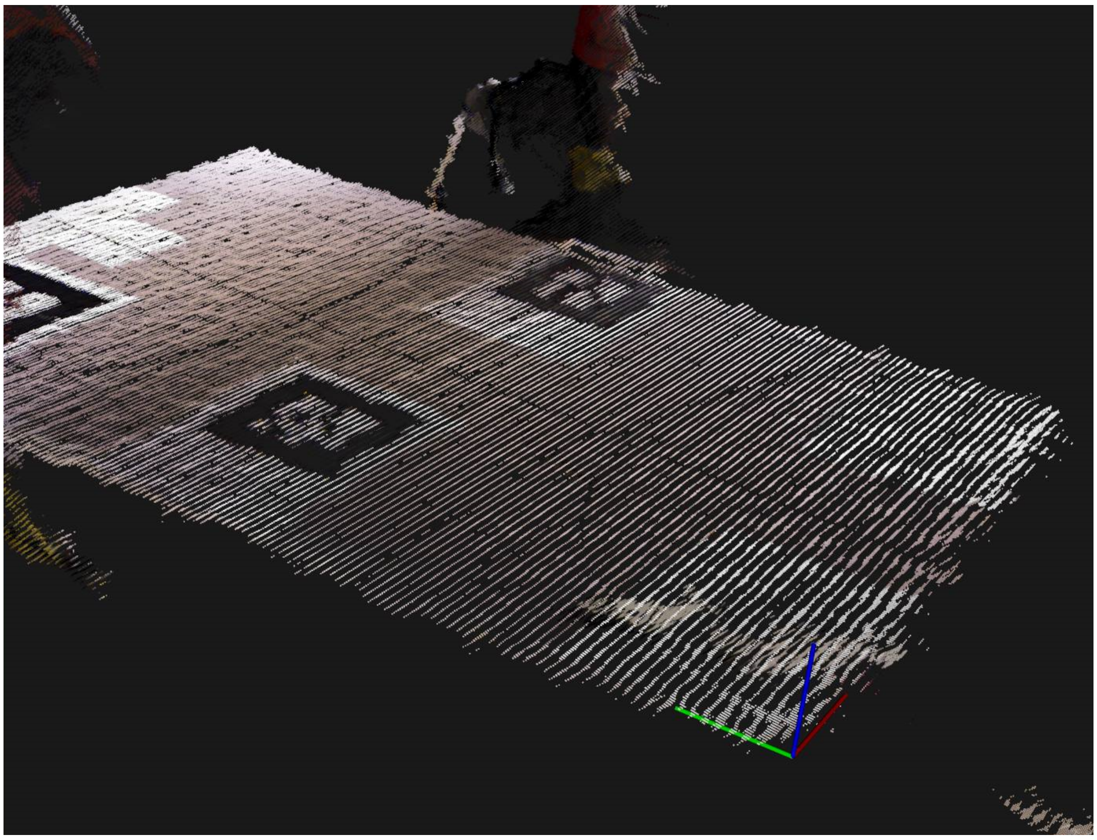


Figure 5.18: The pose of the table $\{t\}$ is shown in the point cloud.

This coordinate frame is convenient when taking measurements and it is easy to relate to.

5.1.3 Template Alignment

This section explains how the code, Template alignment explained in Chapter 4.3.1, was used to locate the pose of the parts. A template cloud was made for the piston and another one for the cylinder. The two templates were then used in the code to align them to the target cloud.

The following parameters were set:

- Estimation of normal search radius r_n
- Feature search radius r_f (FPFH)
- Minimum sample distance d_{min} (SAC-IA)
- Maximum correspondence distance c_{max} (SAC-IA)

Default parameters were set to:

- $r_n = 0.02$
- $r_f = 0.02$
- $d_{\text{min}} = 0.05$
- $c_{\text{max}} = 0.0001$

These parameters resulted in a template alignment as shown in Figure 5.19. The purple template is representing the piston and the brown template is representing the cylinder.

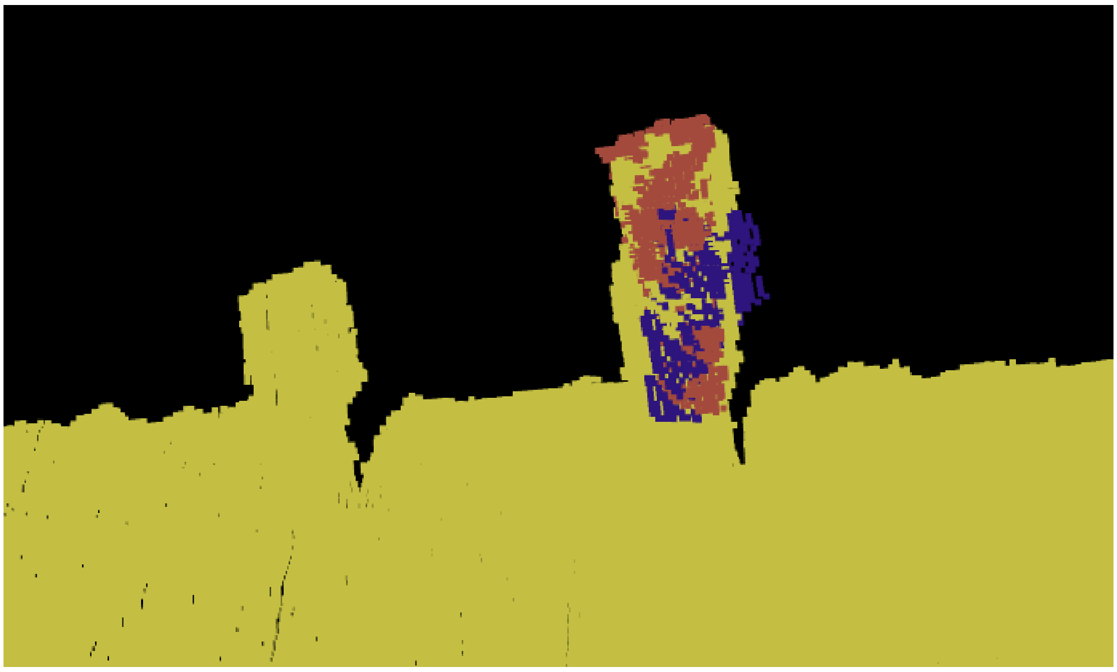


Figure 5.19: Template alignment with initial parameter values.

As shown in Figure 5.19 the initial parameter values did not align the templates correctly to the point cloud.

The best result was obtained by running two independent template alignment programs, one for each of the object. The following parameters were chosen for the program aligning the cylinder:

- $r_n = 0.023$
- $r_f = 0.034$

and for the piston:

- $r_n = 0.02$
- $r_f = 0.025$

Common for both programs:

- $d_{\text{min}} = 0.025$
- $c_{\text{max}} = 0.00005$

These parameter values resulted in an alignment as shown in Figure 5.20.

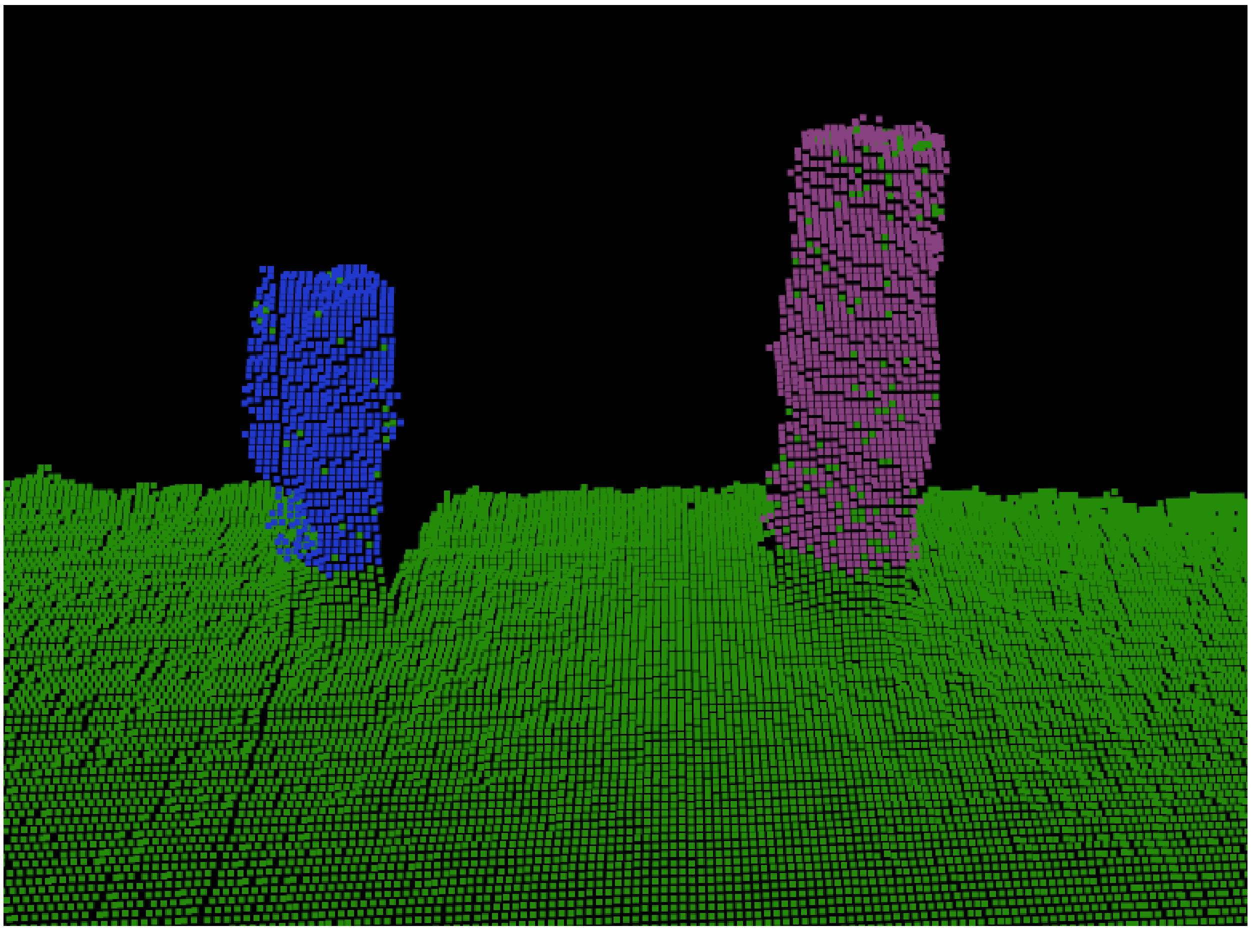


Figure 5.20: Template alignment with new parameter values.

The program estimated a homogeneous transformation from the template to the target cloud. The pose of the objects in the point cloud in Figure 5.20, are the same as the objects pose in the templates. Both the transformation, from the cylinder template to the target cloud and the transformation form the piston template to the target cloud were estimated to be identity matrix, which is correct.

Running template alignment with live point clouds as input cloud from the camera gave different transformations every time. The alignment of the templates to the live point cloud is shown in Figure 5.21. The objects have the same pose as in Figure 5.20. As shown, the templates have not been aligned as good as in Figure 5.20.

The rotation of the piston template to the live target cloud was estimated to be

$$R = \begin{bmatrix} 0.993 & -0.109 & -0.041 \\ 0.111 & 0.993 & 0.045 \\ 0.036 & -0.050 & 0.998 \end{bmatrix} \tag{5.9}$$

and the cylinder template to the live target cloud

$$R = \begin{bmatrix} 0.991 & 0.093 & 0.094 \\ -0.099 & 0.994 & 0.054 \\ -0.089 & -0.063 & 0.994 \end{bmatrix} \tag{5.10}$$

There was also some movement in the translation part of both the homogeneous transformations. Attempts were made to merge multiple point clouds and use statistical outlier removal, without getting a more stable result.

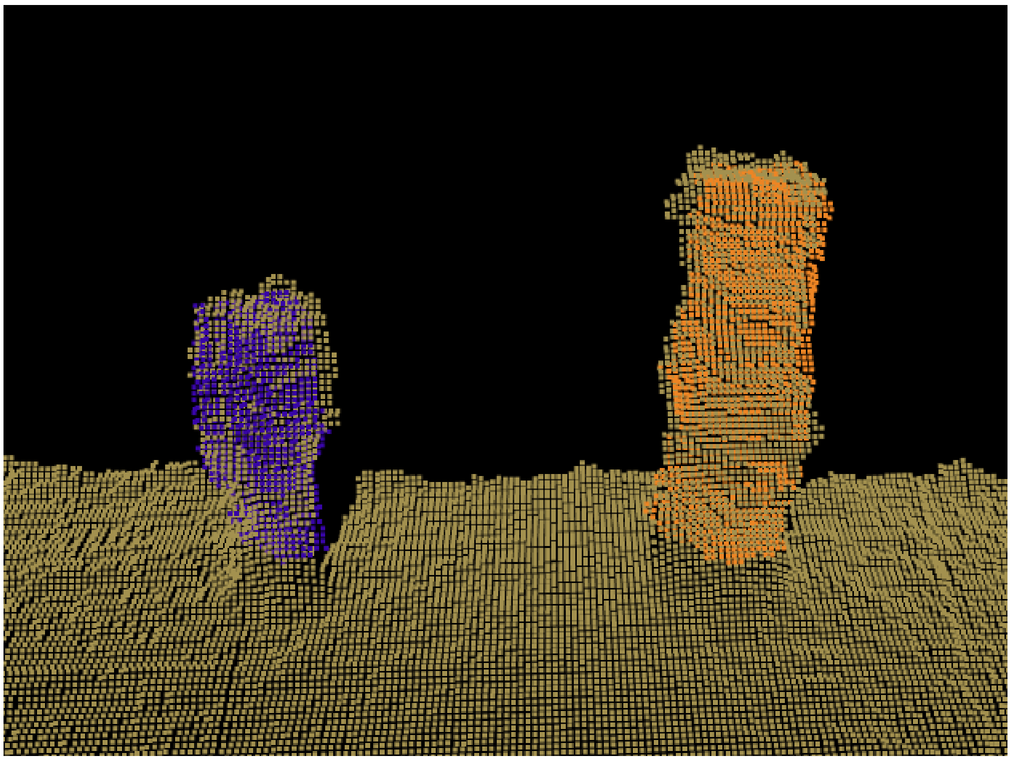


Figure 5.21: Template alignment with a live point cloud.

Next, the objects were oriented 45°. The alignments are shown in Figure 5.22. Until this point there have only been used one template for each object. Therefore, a test with 8 templates (of the objects oriented 0°, 45°, 90°, 135°, 180°, 225°, 270° and 315°) captured from different angles of each object was also conducted. The program uses the template (among the 8 templates) with the best *alignment score*. The best alignment score is a representation of how many of the points in the template that were aligned in the point cloud. This proved to work sometimes, but it was more unstable and the program used different templates for each point cloud captured when the objects were in the same pose. Also, the computational time of the program increased when 8 templates were used for each part.

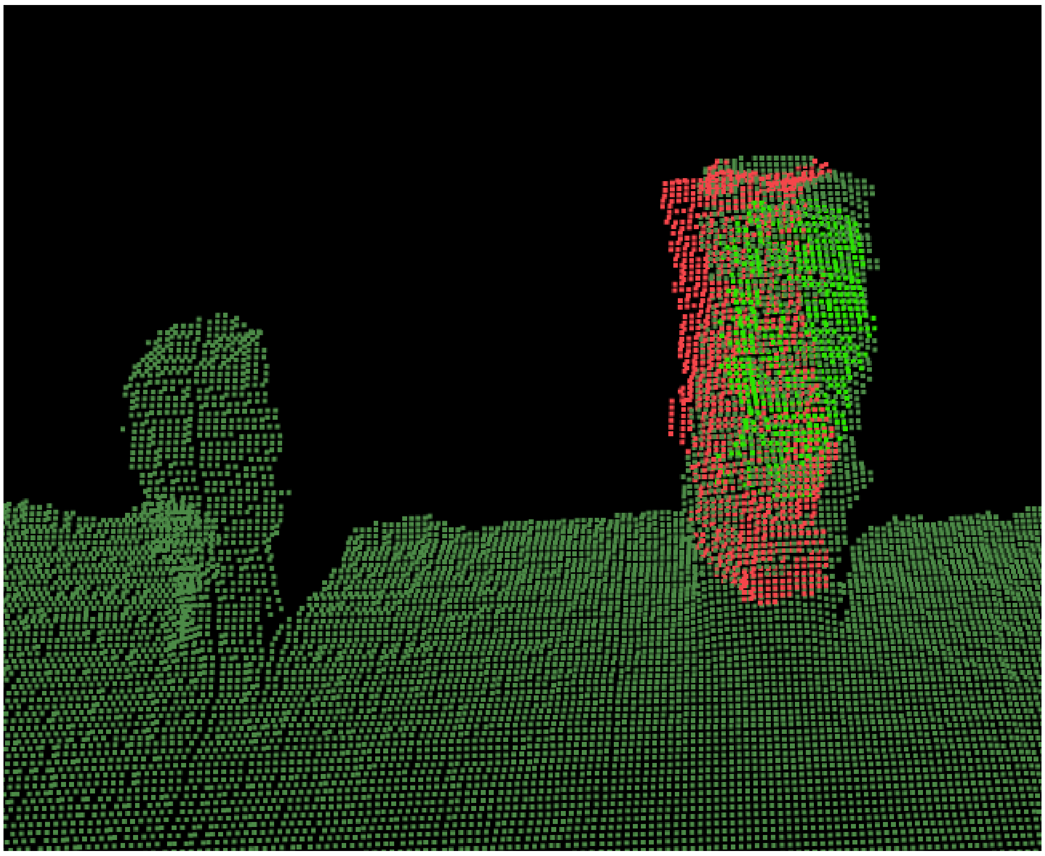


Figure 5.22: Template alignment with 45° oriented objects.

The objects were then moved closer to the camera and oriented 0° (the same as in the templates). The template alignment with two different point clouds are shown in Figure 5.23 and Figure 5.24. As shown, the template alignment differed for every point in the point cloud.

The rotation of the piston template to the target cloud in Figure 5.24 was estimated to be;

$$R = \begin{bmatrix} 0.991 & -0.103 & 0.085 \\ 0.100 & 0.994 & 0.043 \\ -0.089 & -0.034 & 0.995 \end{bmatrix} \tag{5.11}$$

and the cylinder template to the same target cloud

$$R = \begin{bmatrix} 0.996 & 0.054 & 0.073 \\ -0.055 & 0.998 & 0.005 \\ -0.072 & -0.008 & 0.997 \end{bmatrix} \tag{5.12}$$

As shown, the rotation matrices were close to identity matrix. The translation part of the transformations were off with some millimeters. The exception were the cylinder template translation in Y direction, which was off with almost one centimeter.

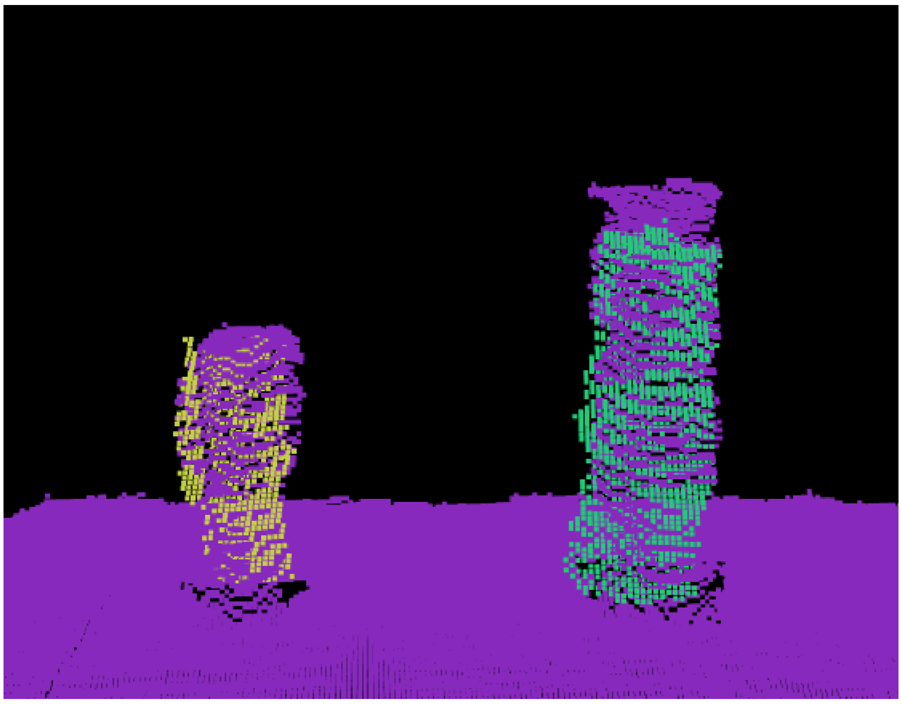


Figure 5.23: Templates alignment with 0° oriented objects closer to camera.

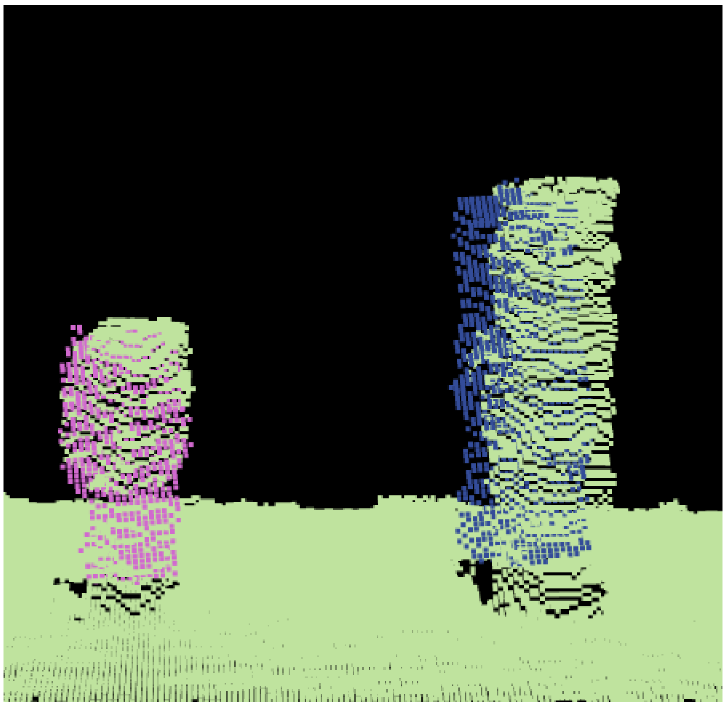


Figure 5.24: Templates alignment with 0° oriented objects closer to camera.

As mentioned, the program did not work better when more templates of each objects were used. It was therefore determined to find the maximal orientation of the objects in the point cloud, while still beeing able to align the template. The objects could be oriented with a maximum of 30°. Figure 5.25 shows the template alignment when the objects were oriented 40°.

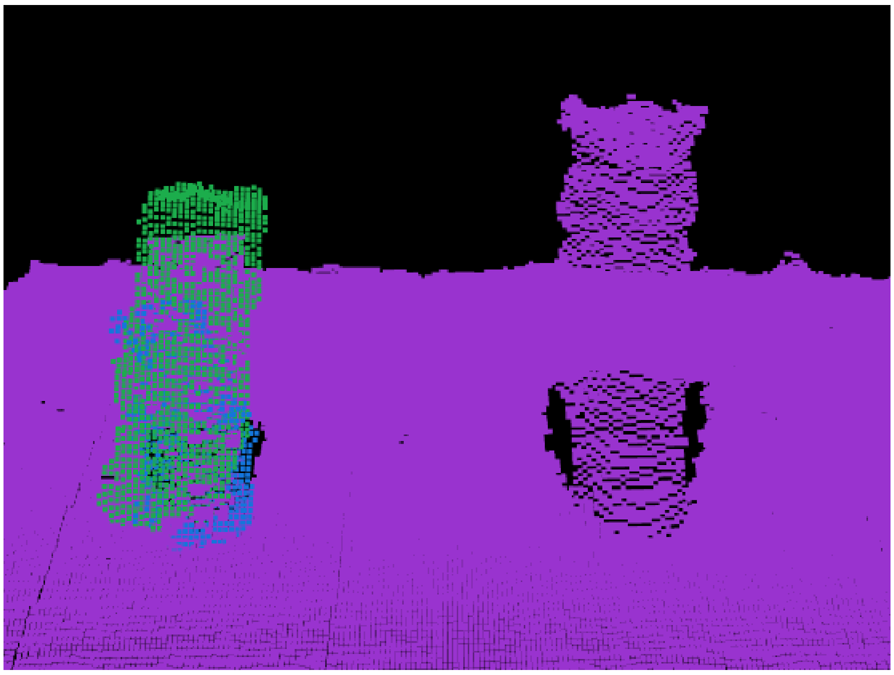


Figure 5.25: The alignment with 40° oriented objects closer to camera.

Still, as the objects were oriented from 0 to 30°, the rotation matrices did not correspond to the true orientations of the cylinder and the piston. Also the translation part was off with one cm. The algorithm used approximately 25 seconds when running template alignment for the cylinder and piston on a point cloud that only did not include the table.

The template alignment method worked for locating the objects with a maximum error of 1.2 cm, considering that the objects had a maximum orientation of 30° with respect to the orientations of the parts in the templates. The orientation part was difficult to estimate, since the geometry of the objects made it difficult to distinguish between different orientations. Also, the RGB-D camera produces a slightly different point cloud for every frame it captures so the alignment differs for every time the program is run.

5.1.4 Region Growing Segmentation

After having tested different ways of determining the position of the piston and the cylinder with respect to the coordinate frame on the table, one of the best results was obtained by a modified version of Region growing segmentation explained in section 4.3.2. This method was not ideal since it was meant for flat and smooth surfaces, but what we were looking for was a segment of the points on the side facing the camera. Region growing segmentation proved to be stable and produced relatively good results repeatedly.

The program reads the point cloud live from the depth camera. Next, the program runs a NaN filter on the point cloud to filter out the NaN values that indicate space locations where the sensor had troubles detecting depth values. This could occur on reflecting surfaces (i.e. metal), surfaces too far away or in areas which are occluded (shadows). Then, the points in the point cloud are transformed from the camera frame $\{c\}$ to the table frame $\{t\}$. Next, the program filters out all the points from the cloud except the points on the table or directly above it.

The surface normals are computed with a scale factor of $r = 0.04$ m. This value gave good results as shown in Figure 5.26, where the normals on the front of the objects are easily distinguished from the normals on the table.

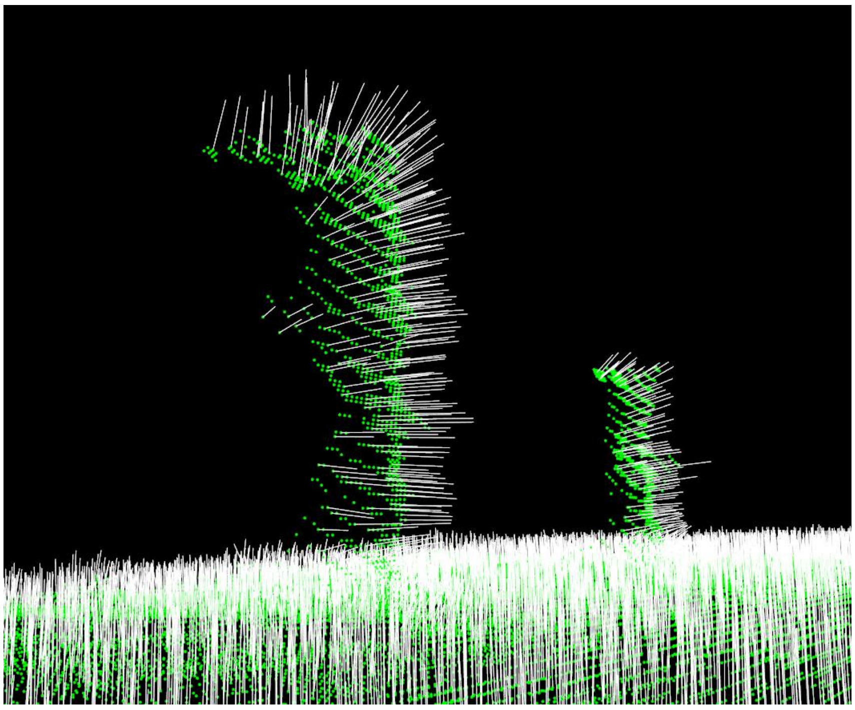


Figure 5.26: Surface normals computed with $r = 0.04$.

When the surface normals are computed the segmentation parameters can be set. The residual threshold r_{th} was set to $r_{th} = 150$, the neighborhood threshold k was set to $k = 20$ and the smoothness threshold θ_{th} was set to $\theta_{th} = 70°$. The resulting segmentation is shown in Figure 5.27.

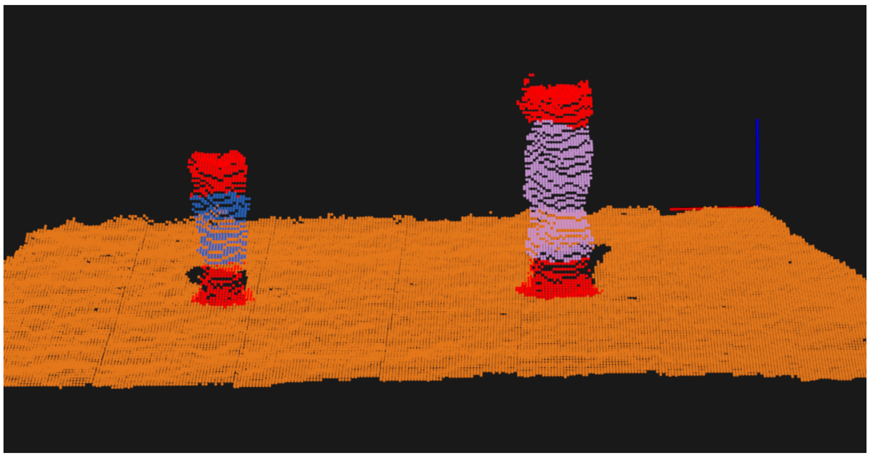


Figure 5.27: Segmentation result. Each segment grown in the cloud gets a random color. Red points are always unsegmented.

The blue segment represents the piston and the purple segment represents the cylinder. There are a lot of noise in the point cloud. The noisy points are normally filtered out with a filter that statistically removes outliers as mentioned in Chapter 4.3.3. This cloud has been captured without the filter to better visualize the scene. The table had not been visible if the filter had been activated. The resulting segments is distinguished by by the number of points and the height of the point cloud. An initial suggestion for the X and Y coordinates with respect to the table frame $\{t\}$, is then calculated from the median of the coordinates of the points in the segments.

The Z coordinate is calculated based on the height the gripper is going to grip the object at. From the points p^t in the point cloud, the coordinates (X, Y) of the piston $p^t_{pis} = (X, Y, 0.10)^{\mathrm{T}}$ and the cylinder $p^t_{cyl} = (X, Y, 0.17)^{\mathrm{T}}$ are extracted.

Figure 5.28 shows the repeatability in the segmentation even if the objects are further away from the camera.

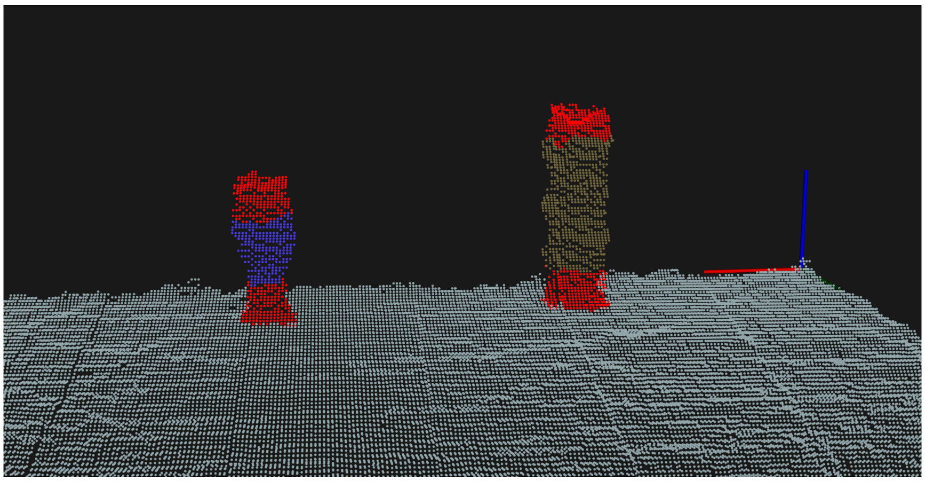


Figure 5.28: Stable segmentation results.

5.1.5 Segmentation based on known Geometry

Two programs were made for estimation of the coordinates of the parts based on known geometry of the parts. One program uses one point cloud and another program uses four merged point clouds. The known geometry of the objects are:

- The cylinder is 18.5 cm tall and has a maximum radius of 5 cm.
- The piston is 12.5 cm tall and has a maximum radius of 4.5 cm.

The programs runs as follows:

1. Transform point cloud/point clouds to table coordinates $\{t\}$.
2. Statistical removal of outliers.
3. Search for a point p on the table that has $Z > 15$ cm.
4. If there are more than 30 points in a cubic volume around p as shown in Figure 5.29 with $Z > 3$ cm, save this point. Else, do point 3 again.
5. Segment the points located within a cubic volume around p shown as the blue cube in Figure 5.29 (3 cm $< Z <$ 12 cm).
6. This segment is saved as the cylinder segment.
7. Search for a new point p that is 4 cm $< Z <$ 11 cm and has not already been segmented.
8. If there are more than 10 points in a cubic volume around the new p, save this point, else do point number 7 again.
9. Segment the points that are in a cubic volume around p with 3 cm $< Z <$ 8.5 cm.
10. This segment is saved as the piston segment.

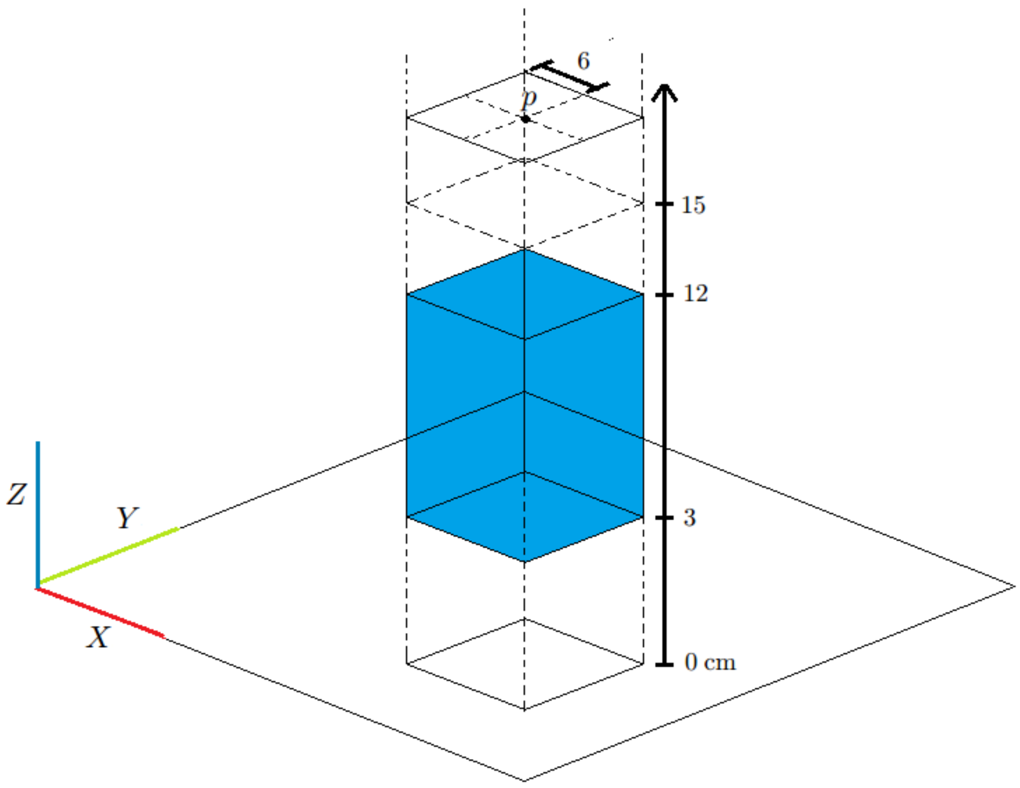


Figure 5.29: Segmentation and search algorithm for the cylinder.

In Figure 5.30 these segments are shown in the program that uses one point cloud at a time.

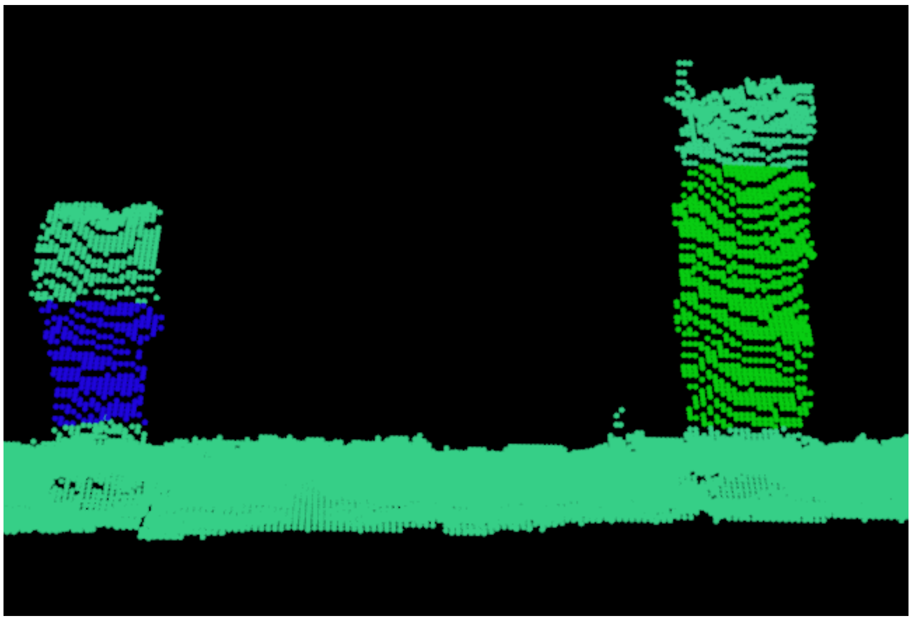


Figure 5.30: One point cloud and segmentation of the cylinder and the piston.

In Figure 5.31 the same segments are shown in the program that uses four point clouds at a time. The segments consist of a more dense point cloud.

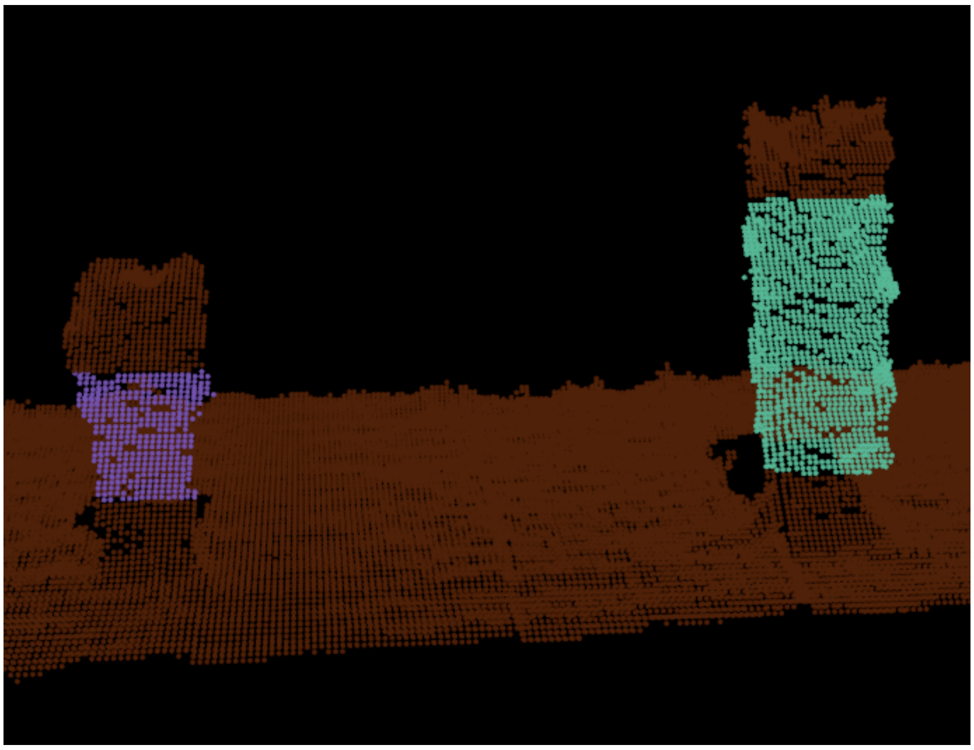


Figure 5.31: Four merged point clouds and segmentation of the cylinder and the piston.

Estimating the coordinates of the cylinder and the piston from the segments, is done in the same way as described in Chapter 5.1.4.

5.1.6 Analysis

Evaluating the Programs for locating the Objects

There are three working programs for estimating the coordinates of the piston and cylinder. These programs are:

- Segmentation based on known geometry using one point cloud.
- Segmentation based on known geometry using 4 point clouds.
- Region growing segmentation.

These programs were tested individually for locating the coordinates of the cylinder and the piston. The tests gave approximately the same result for all the programs. The differences between the programs were the computational time to estimate the coordinates, and the repeatability of the estimations.

Figure 5.32 and Figure 5.33 shows the result from a test where the cylinder was placed at two different locations $p^t_{\text{cyl}} = (0.25, 0.5)$ and $p^t_{\text{cyl}} = (0.25, 0.1)$ in the table coordinate frame $\{t\}$. For each program there were sampled 29 coordinates for each location of the cylinder. As shown in Figure 5.32 and Figure 5.33, the program *Segmentation based on known geometry using 4 point clouds* is the program that estimates the most stable coordinates and *Segmentation based on known geometry using one point cloud* is the least stable.

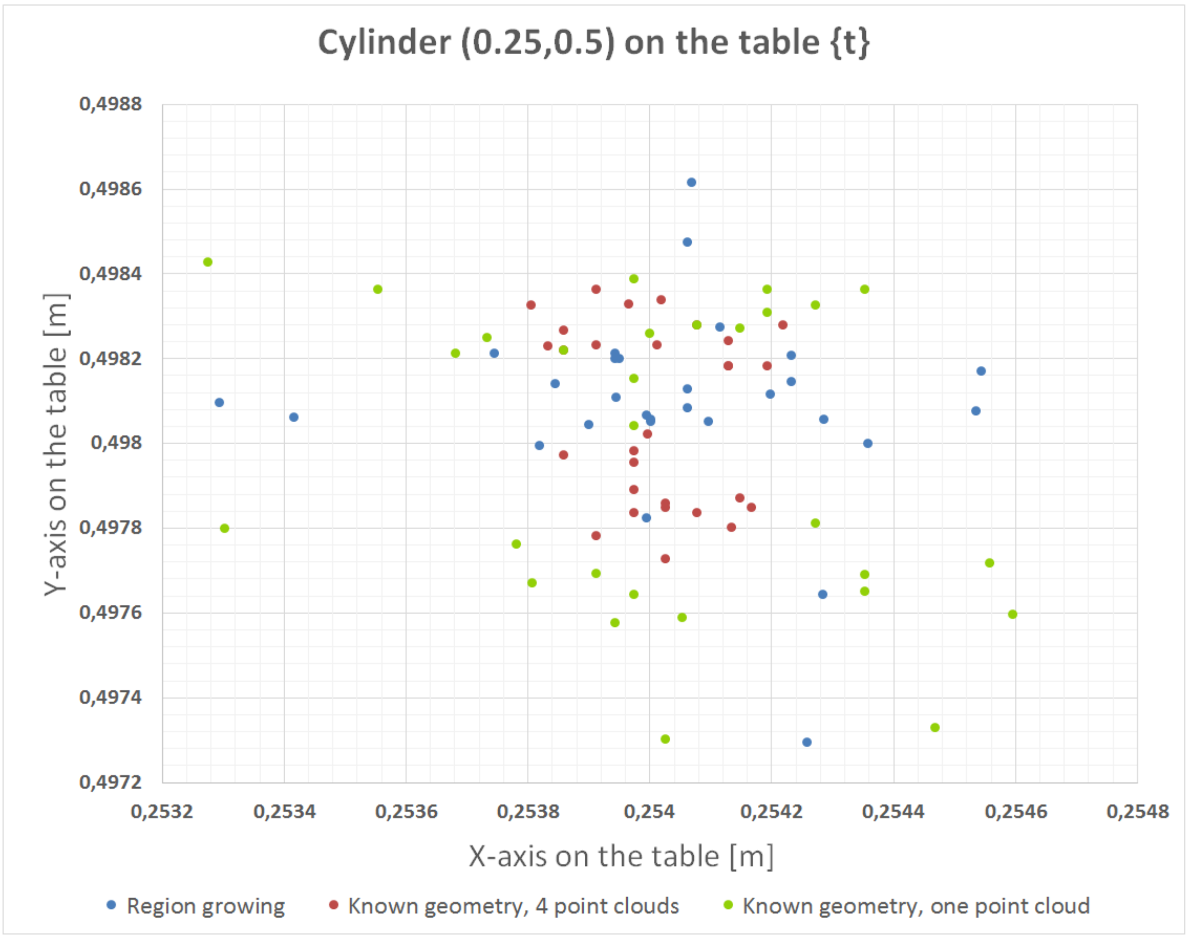


Figure 5.32: 29 samples of the coordinates of the cylinder located at $(0.25, 0.5)$ in $\{t\}$.

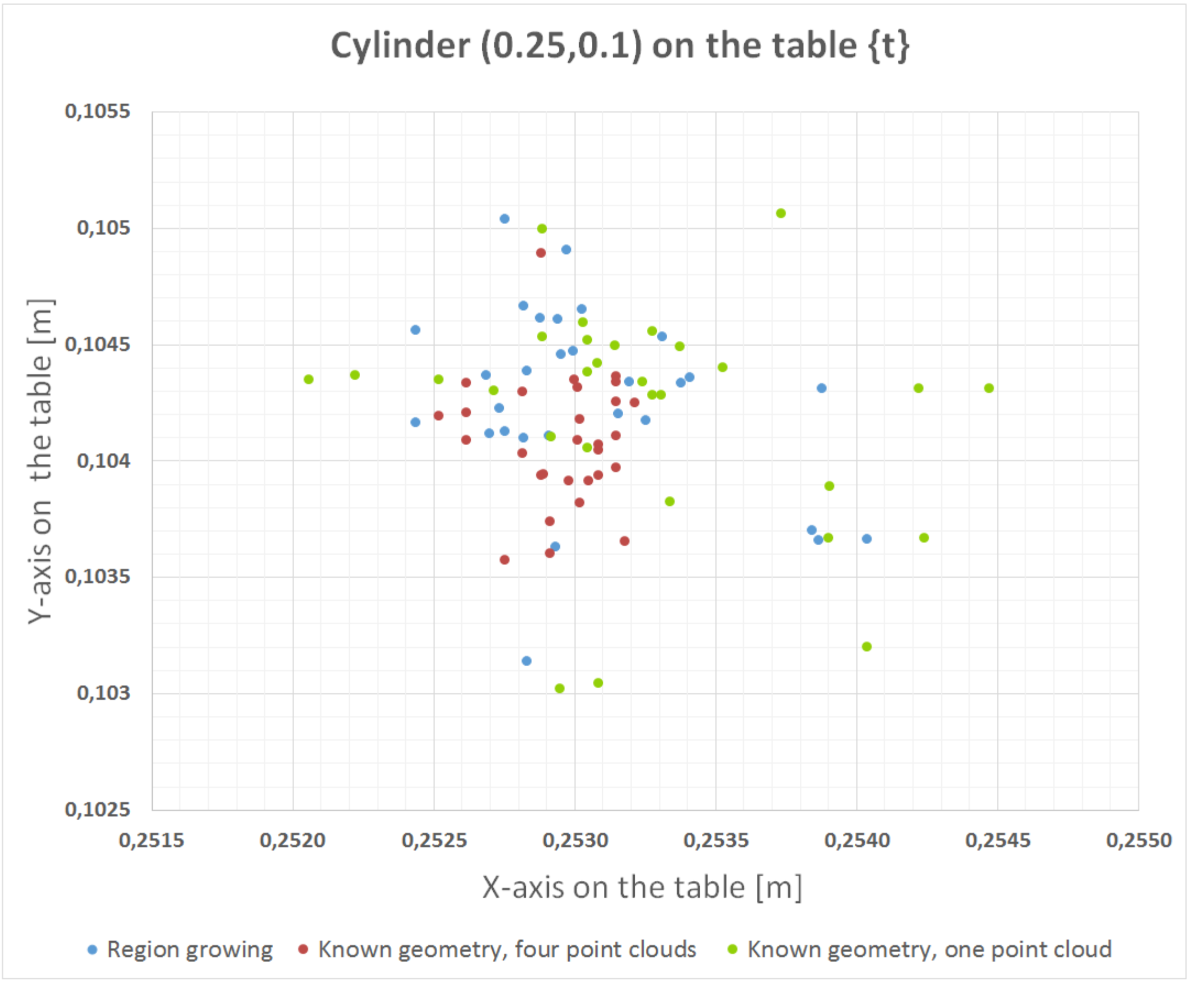
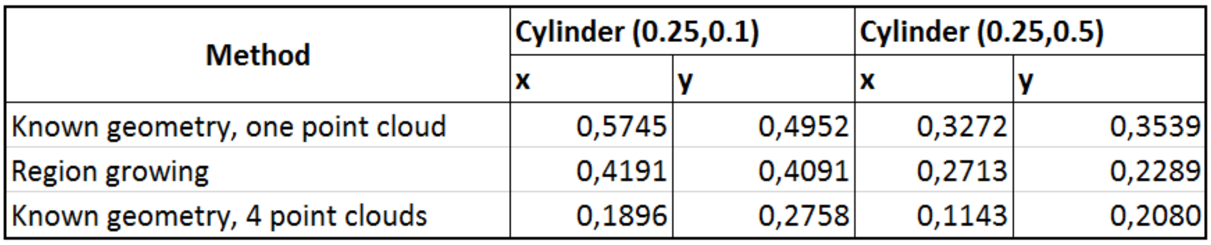


Figure 5.33: 29 samples of the coordinates of the cylinder located at $(0.25, 0.1)$ in $\{t\}$.

The standard deviations with 95% confidence level of the 29 samples, for each of the three different programs, are shown in Figure 5.34.

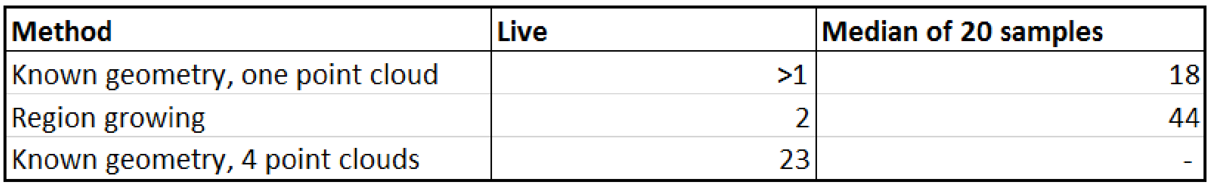


Method	Cylinder (0.25,0.1)		Cylinder (0.25,0.5)	
	x	y	x	y
Known geometry, one point cloud	0,5745	0,4952	0,3272	0,3539
Region growing	0,4191	0,4091	0,2713	0,2289
Known geometry, 4 point clouds	0,1896	0,2758	0,1143	0,2080

Figure 5.34: Standard deviations (95% confidence level) in millimeters.

As shown in Figure 5.34 the standard deviations varies with the distance from camera. That is, the standard deviations increased the further away from camera the cylinder was. The point (0.25,0.1) is further away from the camera than the point (0.25,0.5). Still, the program *Segmentation based on known geometry using one point cloud*, which is the least stable, had at the most a standard deviaton (with 95% confidence level) of ±0.57 mm in X direction and ±0.50 mm in the Y direction. This was still acceptable for performing a simplified assembly task of the parts, assuming that the mean of the samples gives the correct coordinates.

In Figure 5.35 the computation time for the different programs are shown. For the programs *Segmentation based on known geometry using one point cloud* and *Region growing segmentation* there were also implemented an option for sampling 20 coordinates of both parts, and then use the median of those samples as the coordinates.



Method	Live	Median of 20 samples
Known geometry, one point cloud	>1	18
Region growing	2	44
Known geometry, 4 point clouds	23	-

Figure 5.35: Computation time in seconds for the different programs.

Assembling the parts required high accuracy in the measurements of the coordinates. It was therefore of interest to eliminate errors, like the standard deviations, in the estimate of the coordinates. That is why the median of 20 sampled coordinates for the programs *Segmentation based on known geometry using one point cloud* and *Region growing segmentation* were implemented as an option. The median of 20 sampled coordinates with the program *Segmentation based on known geometry using one point cloud* was used in the demo video where a robot assembles the parts. If the demand for fast production is the goal, the *Region growing segmentation* would be a suitable choice.

Computed Pose VS. Measured Pose

This section will demonstrate the accuracy of the position of the objects seen from the camera with respect to the table coordinate frame. The program code used in this experiment is based on the *Segmentation based on known geometry using one point cloud* with median of 20 sampled coordinates. The objects to detect is the piston and the cylinder, which were placed in nine different positions on the table. For each position, the coordinates in the X Y plane on the table $\{t\}$ were measured with a tape measure. These coordinates were then compared with the coordinates from the computer program based on the *Segmentation based on known geometry using one point cloud*. Figure 5.36 and Figure 5.37 shows the difference between the measured coordinates and the computed coordinates for the piston and the cylinder respectively. The axis of the diagram is the X axis and Y axis of the table frame $\{t\}$ in the corner of the table and the unit is in meters. The diagrams clearly show that there is a relatively constant error between the actual coordinates and the computed coordinates.

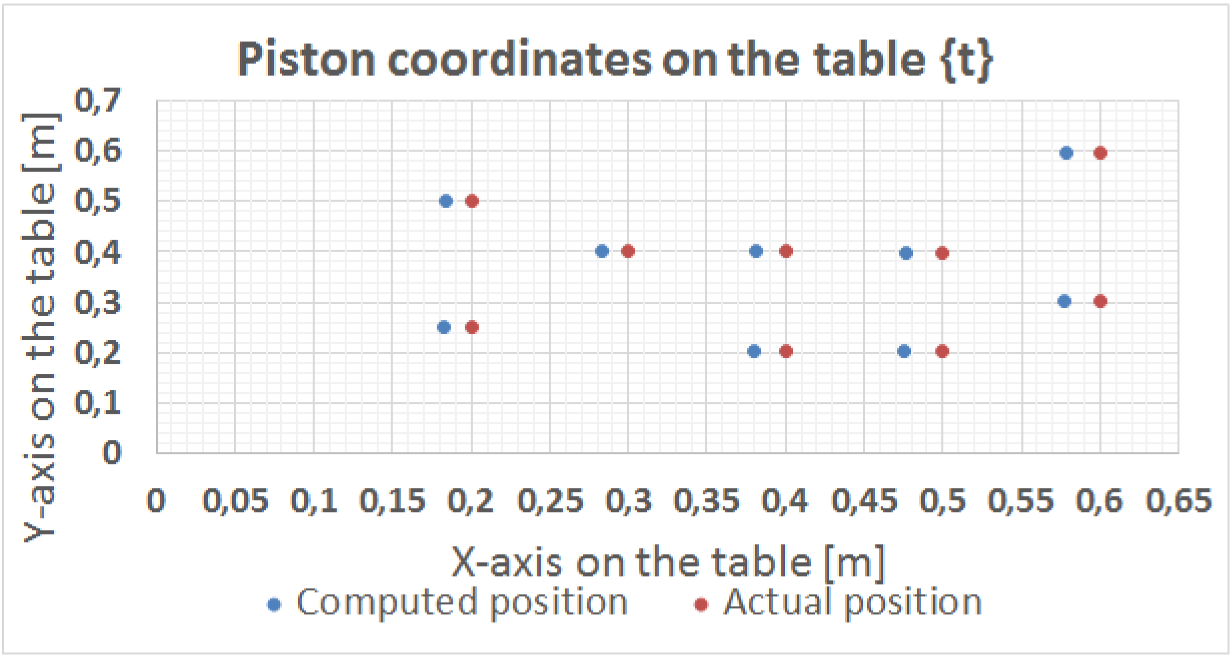


Figure 5.36: Raw coordinates for piston.

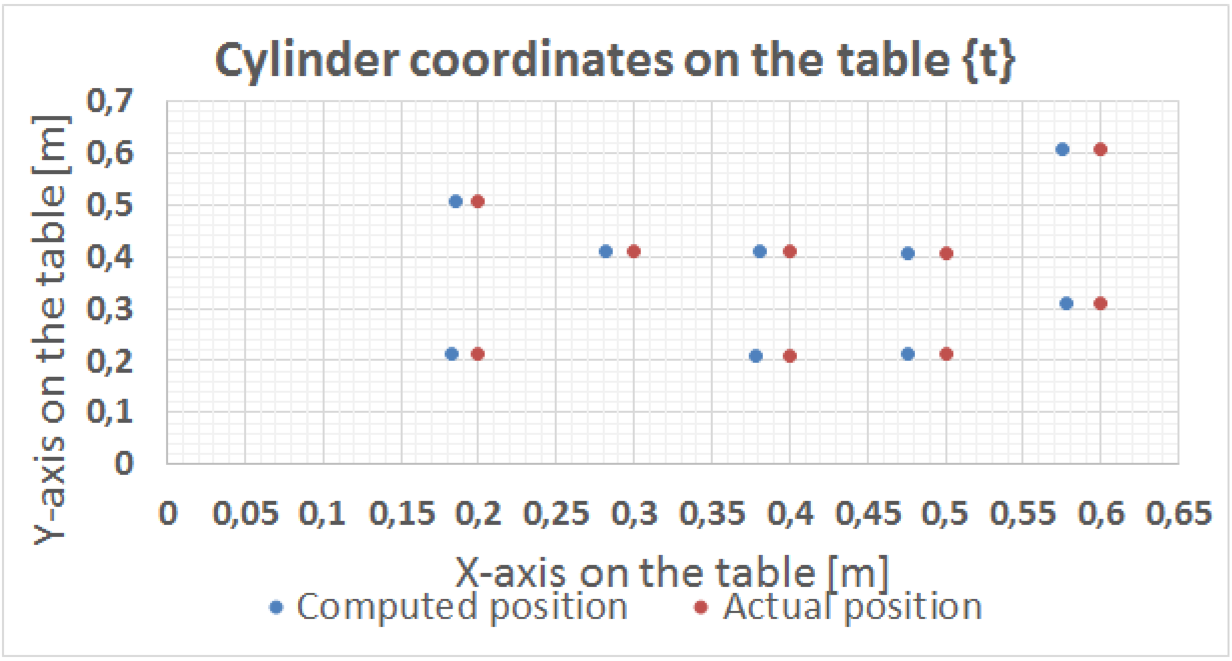


Figure 5.37: Raw coordinates for cylinder.

To compensate for the error, the average error between the actual position and the computed position for the nine samples, were subtracted. The result is shown in Figure 5.38 and Figure 5.39 for the piston and the cylinder respectively.

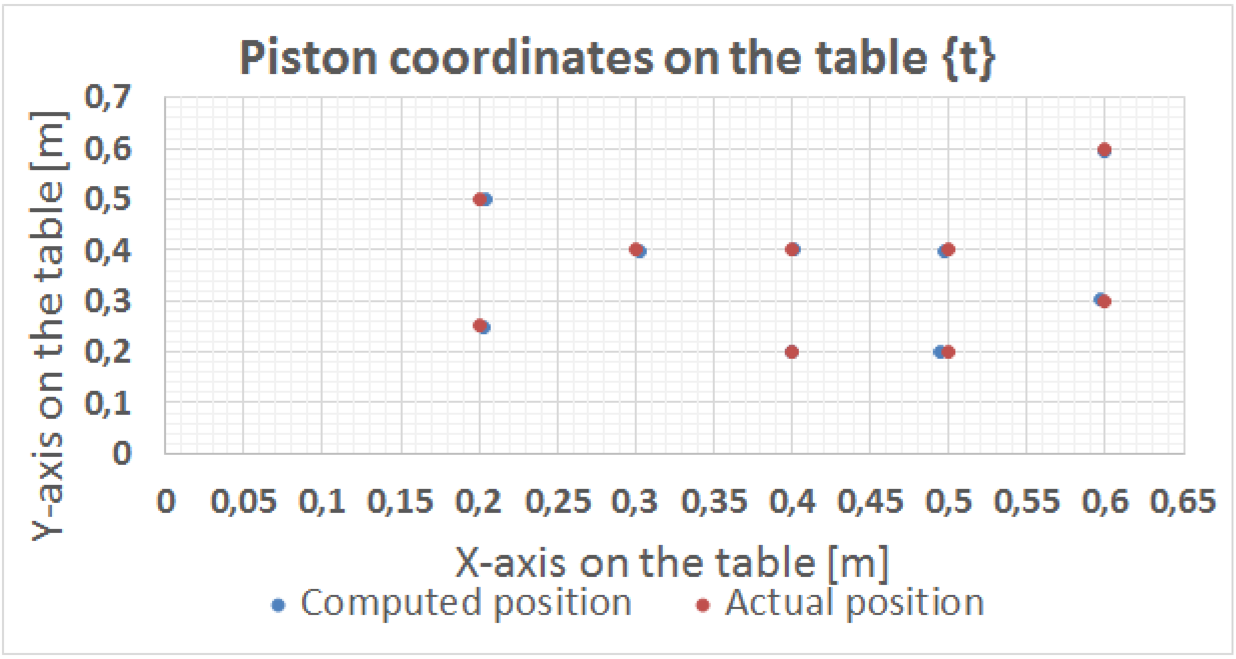


Figure 5.38: Improved coordinates for piston.

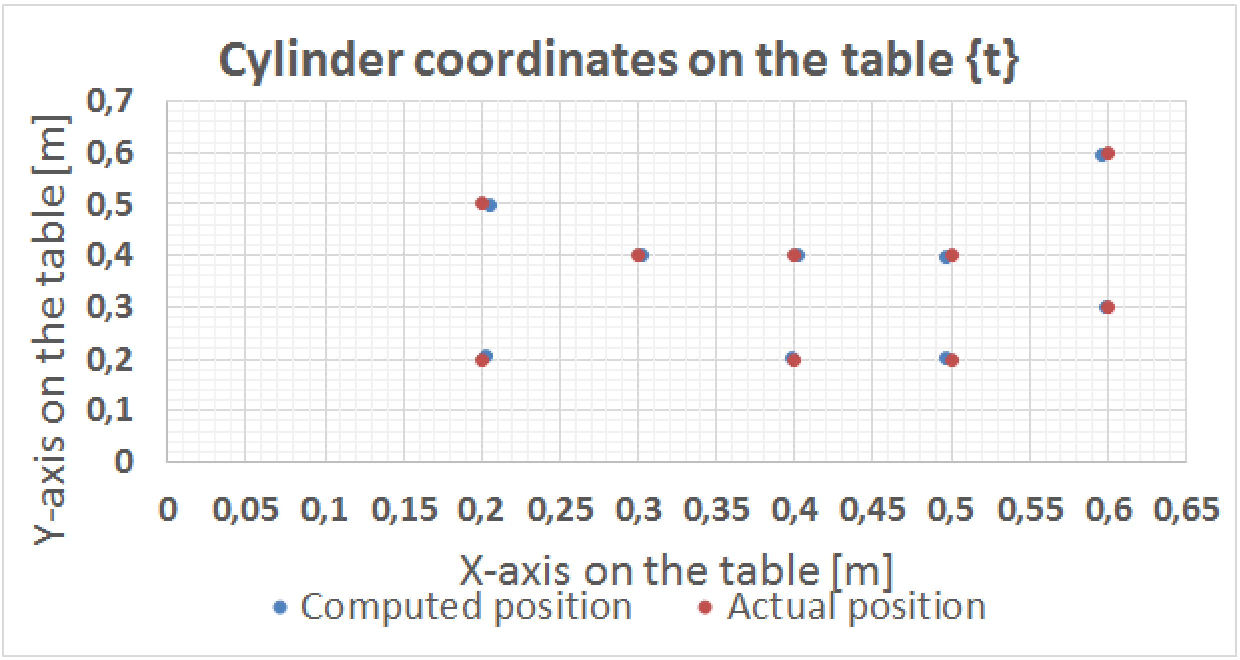


Figure 5.39: Improved coordinates for cylinder.

The result shows that the improved coordinates are much more accurate than before. For the nine samples the piston's average error was 0.0026 m in X direction and 0.0016 m in Y direction. The cylinder's average error was 0.0029 m in X direction and 0.0022 m in the Y direction. That is, an average error of 3.05 mm for the piston and 3.64 mm for the cylinder. The results obtained were not adequate for the original assembly task which require sub-millimeter accuracy, but it was possible to perform a simplified assembly task requiring 4 mm accuracy. As previously mentioned, the other methods, *Segmentation based on known geometry using 4 point clouds* and *Region growing segmentation* did not produce more accurate coordinates for the parts.

In this test the camera was approximately 1.5 m away from the objects so the results will most likely improve if the camera is mounted closer to the objects of interest. It is also likely that a more thorough and comprehensive calibration technique for the RGB-D camera will increase the accuracy. Also, a calibration of transformation $T_{\text{ir}}^{\text{rgb}}$ from RGB camera to IR camera could have improved the transformation T_a^c from RGB-D camera to the AR-tag pose. This is because the AR-tags are recognized by the RGB camera and the parts location (X, Y, Z) are given by the IR camera.

5.2 Experiments based on the 2D Camera

This chapter presents a program that uses a 2D camera (Logitech HD Pro Webcam C920) to estimate the orientations of the piston and the cylinder. For this purpose a OpenCV program [14] was used and further developed.

5.2.1 Orientation of the Objects

The program has four *training images* of each object (the piston and the cylinder). The training images are shown in Figure 5.40 and Figure 5.41 for the piston and the cylinder respectively.

Four training images was necessary because the geometry of the objects made it difficult to estimate an orientation with one, two and three training images. The training images $I_{o,0}$, $I_{o,90}$, $I_{o,180}$ and $I_{o,270}$ represents the objects when they are oriented 0°, 90°, 180° and 270°.

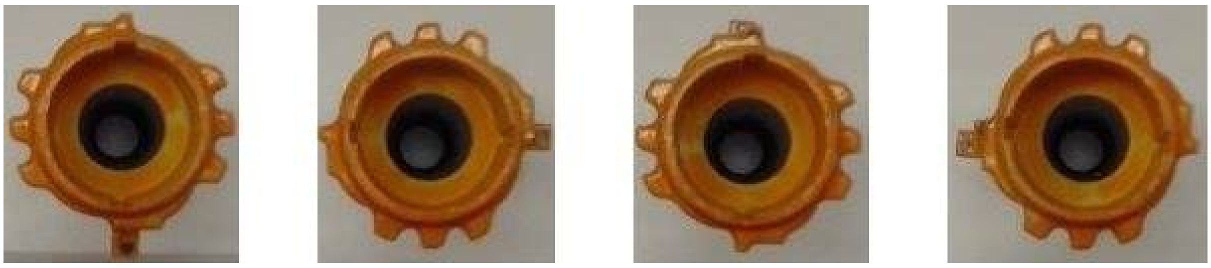


Figure 5.40: From left: The training images $I_{o,0}$, $I_{o,90}$, $I_{o,180}$ and $I_{o,270}$ for the piston.

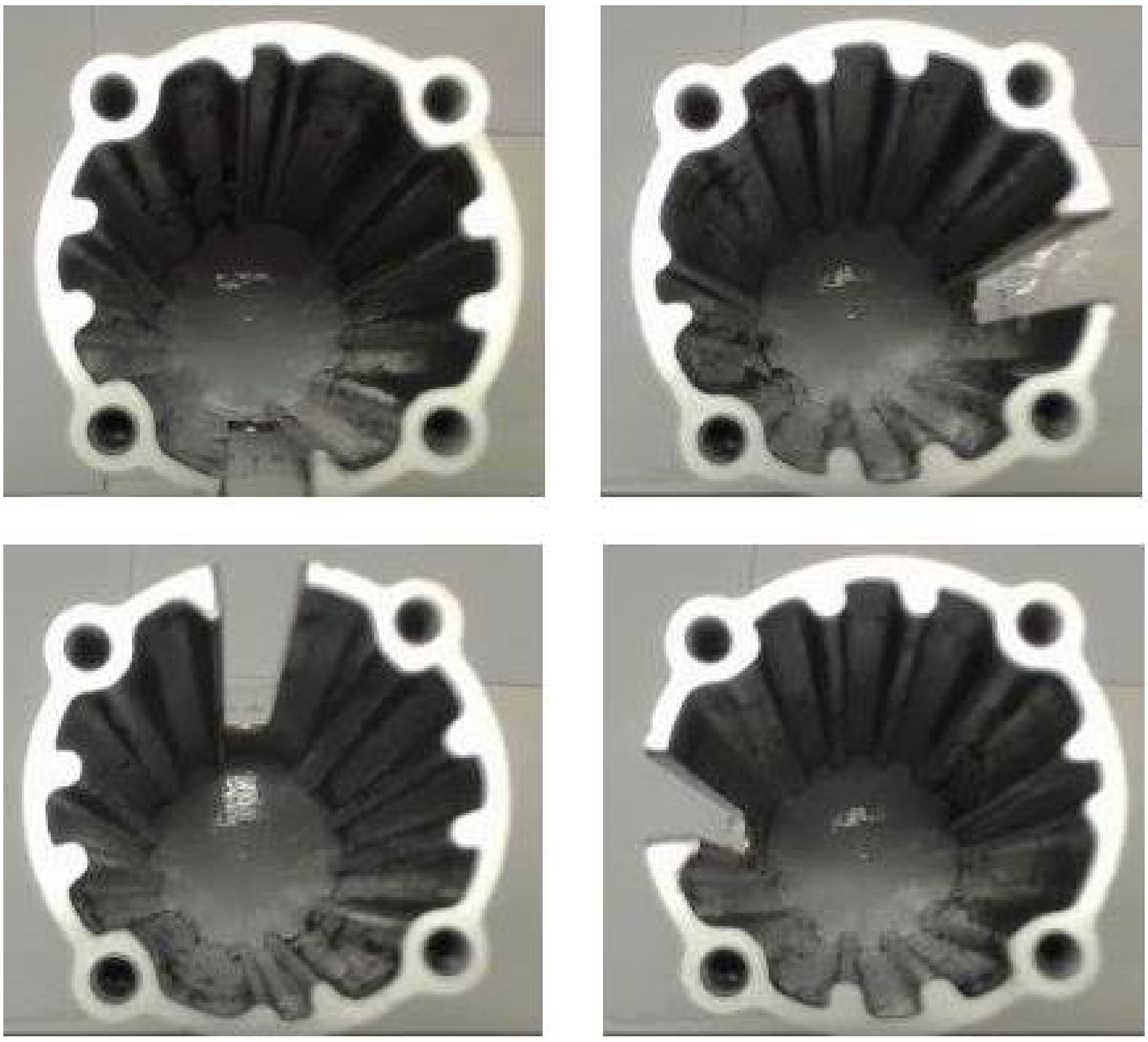


Figure 5.41: From upper left to right: The training images $I_{o,0}$, $I_{o,90}$, $I_{o,180}$ and $I_{o,270}$ for the cylinder.

SIFT features are extracted from the training images and the live image stream from the 2D camera and then stored in a database. The threshold for the Hessian matrix in equation (4.7) is set so it approves a maximum of 200 features in each image.

An example of extracted features from two image streams, are shown in Figure 5.43 and Figure 5.43.

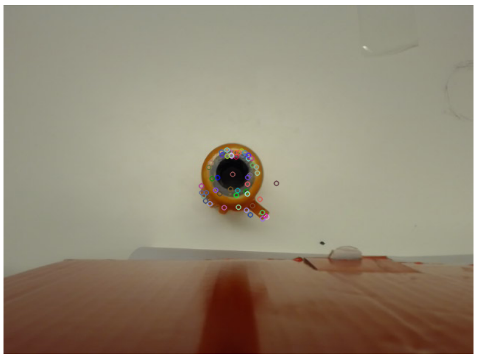


Figure 5.42: Features from the image stream.

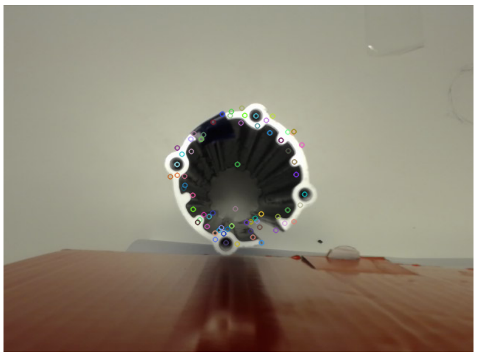


Figure 5.43: Features from the image stream.

The rest of this chapter, presents the program for determining the orientation of the cylinder.

Examples of accepted matched features in the training image $I_{o,0}$ in Figure 5.41 and the image stream in Figure 5.43, are shown in Figure 5.44. The matched features have been accepted based on having a Euclidean distance less than 210. As shown in Figure 5.44 not all matched features are consistent.

As mentioned, using RANSAC will improve the estimation of the homography matrix. For the example shown in Figure 5.44, the normalized homography matrix was

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$$

$$H = \begin{bmatrix} 0.8974 & 0.4594 & 183.271 \\ -0.4714 & 0.8793 & 201.616 \\ 4.958 5\dot{1}0^{-5} & -3.908 8\dot{1}0^{-5} & 1 \end{bmatrix} \tag{5.13}$$

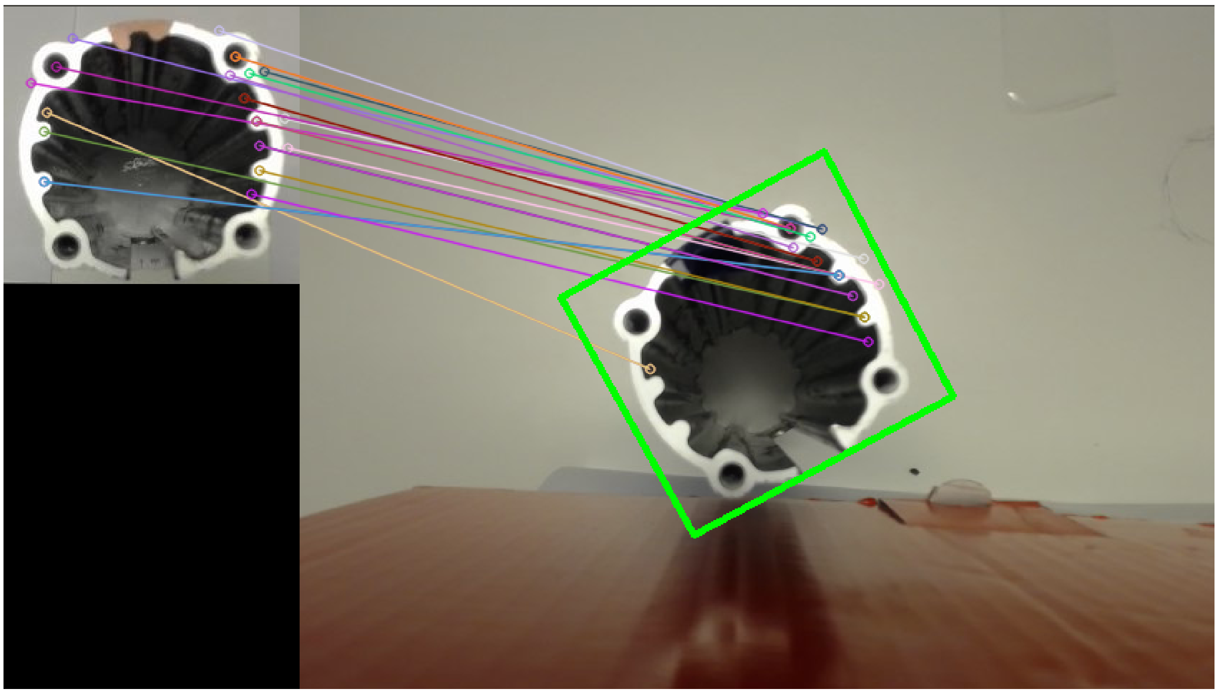


Figure 5.44: Homography to find the orientation of the cylinder.

The camera is perpendicular to the cylinder in the training images and the image stream. The program is designed so the cylinder could be placed in a specified region in the field of view of the camera, as explained in the demo video. The cylinder can be oriented in any direction in the image stream, and there will be an *affine homography* between the training images and the image stream. That is, the projection of the corner points $p = (u, v)^T$ in the training images, to the corner points $p' = (u', v')^T$ (the corners of the green square in Figure 5.44) in the image stream

$$u' = \frac{h_{11}u + h_{12}v + h_{13}}{h_{31}u + h_{32}v + h_{33}} \qquad v' = \frac{h_{21}u + h_{22}v + h_{23}}{h_{31}u + h_{32}v + h_{33}} \tag{5.14}$$

is an affine transformation with $h_{31} = h_{32} = 0$ and $h_{33} = 1$. The homography in equation (5.13) has $h_{31} \approx h_{32} \approx 0$ and $h_{33} = 1$. The program is designed only to approve a homography matrix if it is an affine transformation. This projective transformation or homography projects the training images into a projectively equivalent figure in the image stream [4].

Examples of corner points (the corners of the image) in the training image $I_{o,0}$ mapped to an image in the image stream, are shown in Figure 5.45. These mapped corner points are visulized as p_0, p_1, p_2 and p_3.

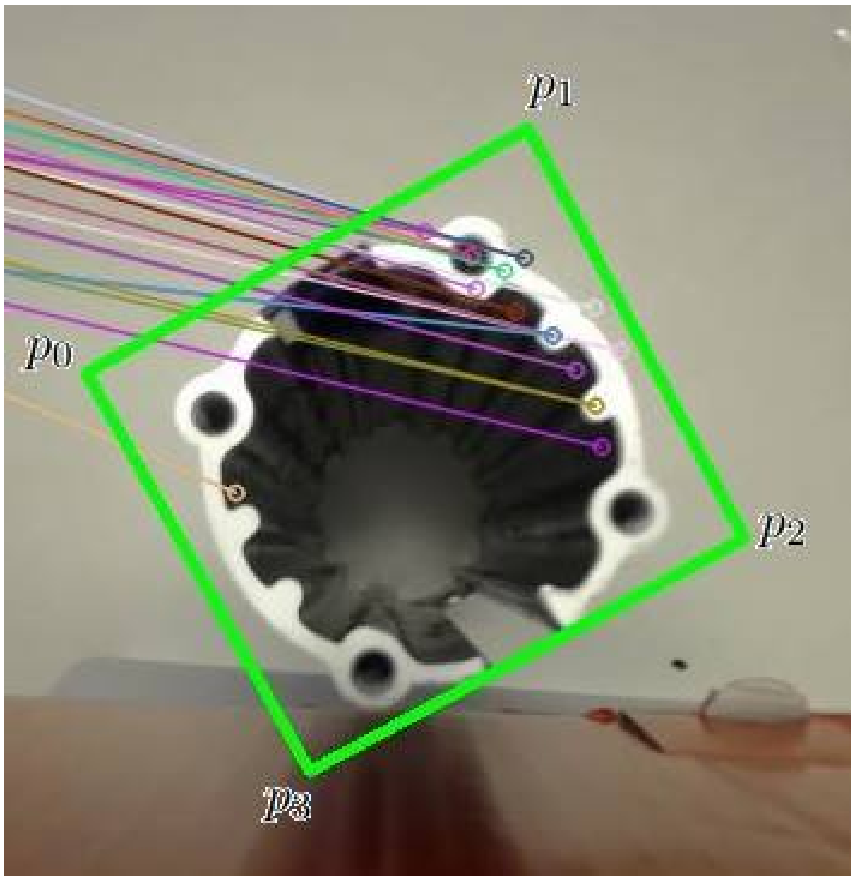


Figure 5.45: Corner points of the affine transformation.

The rotation angle of the cylinder in Figure 5.45 is

$$\begin{aligned}\theta_{p_{12}} &= \cos^{-1}\left(\frac{v_2 - v_1}{\sqrt{(u_2 - u_1)^2 + (v_2 - v_1)^2}}\right) \\ \theta_{p_{03}} &= \cos^{-1}\left(\frac{v_3 - v_0}{\sqrt{(u_3 - u_0)^2 + (v_3 - v_0)^2}}\right)\end{aligned} \tag{5.15}$$

where $p_i = (u_i, v_i)^T$ and θ_0 is the estimated orientation of the cylinder from the homography matrix between $I_{o,0}$ and the image stream. If $\theta_{p_{12}} \approx \theta_{p_{03}}$, something it should be since the plane is an affine transformation, then $\theta_0 = (\theta_{p_{12}} + \theta_{p_{03}})/2$ is an approved orientation.

The program estimates a homography matrix and a θ for each of the training images. That is, θ_0 for the orientation between $I_{o,0}$ and the image stream, θ_{90} for the orientation between $I_{o,90}$ and the image stream, θ_{180} for the orientation between $I_{o,180}$ and the image stream, θ_{270} for the orientation between $I_{o,270}$ and the image stream. The estimated θ_{90}, θ_{180} and θ_{270} are computed with use of different points than in the example shown in equation (5.15).

The program estimates which θ_i, where $i \in \{0, 90, 180, 270\}$, that has a stable value, and which value that dominates in the different θ_i.

Figure 5.46 shows the projective transformation of the corner points (corners of the green square) from the training images ($I_{o,0}$, $I_{o,90}$, $I_{o,180}$ and $I_{o,270}$) to the image stream. In this case, the homography matrix for $I_{o,90}$ and $I_{o,270}$ were rejected and θ_{90} and θ_{270} were not estimated. $\theta_0 = 30.674°$ and $\theta_{180} = 210.942°$ were both stable. That is, the program still does not know what the actual orientation of the cylinder is. For the program to be able to estimate the true orientation of the cylinder, the area on the opposite side of the guiding "slide track" on the cylinder have been marked with a dark color as shown in Figure 5.46.

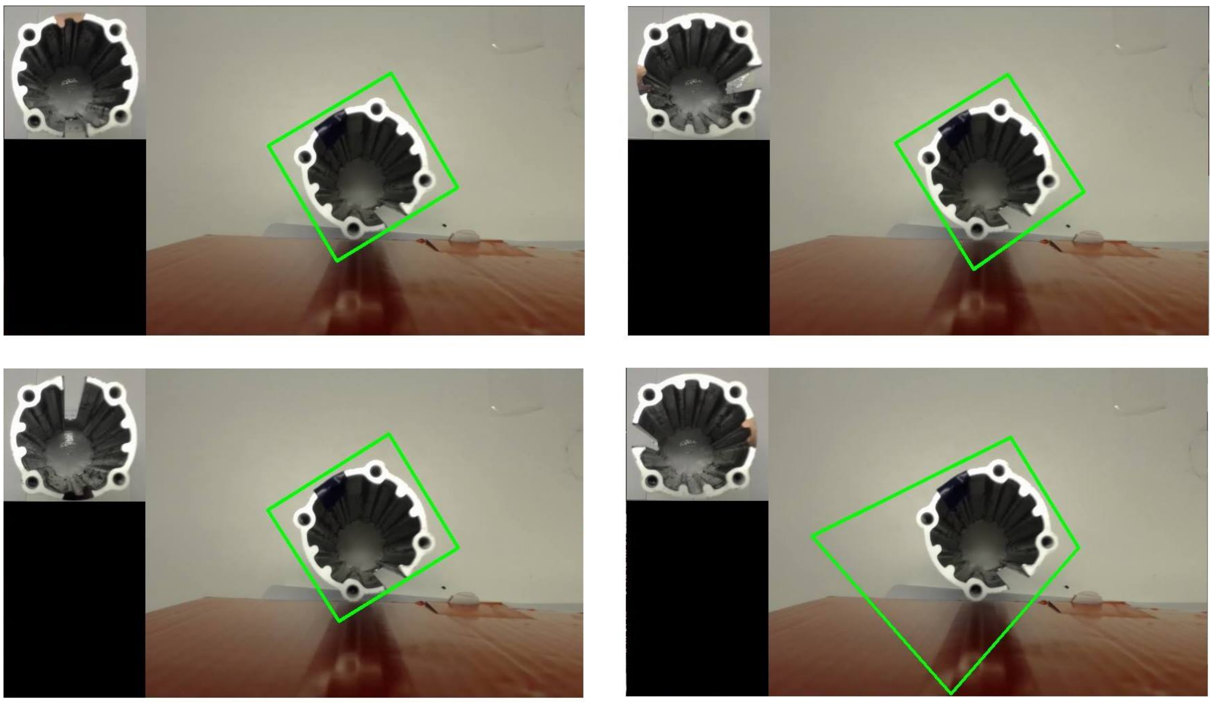


Figure 5.46: The projective transformation of the corner points in the training images to the image stream.

The program reads the grey scale value of three pixels p_{180} around the edge of the cylinder at θ_{180} and three pixels p_0 around θ_0, in this case. The pixels p_0 are shown as green squares in Figure 5.47 and the three pixels p_{180} are shown Figure 5.48.

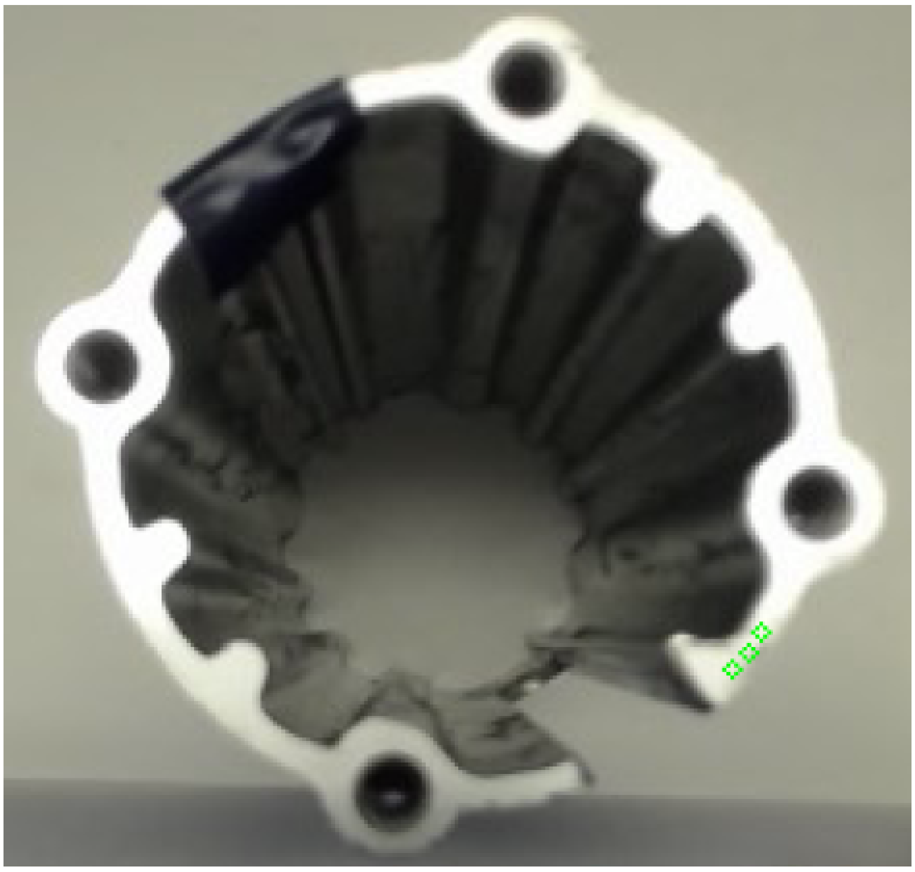


Figure 5.47: The three pixles p_0 around the edge of the cylinder with orientation equal θ_0.

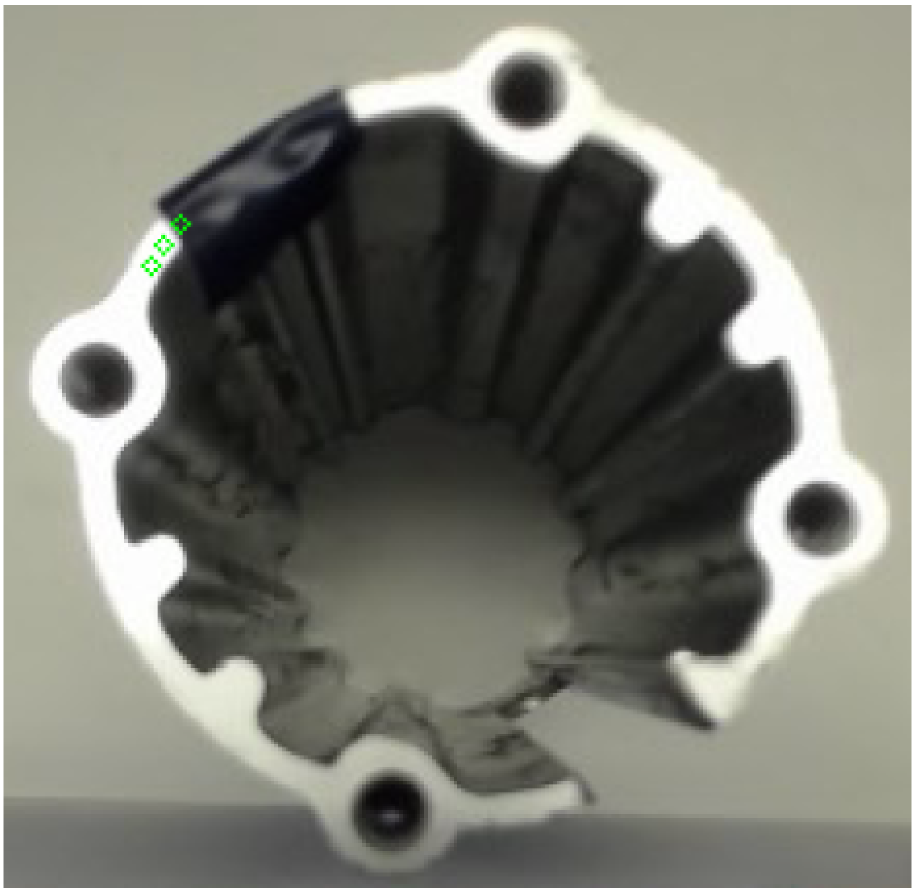


Figure 5.48: The three pixles p_{180} around the edge of the cylinder with orientation equal θ_{180}.

The program need information about the radius r of the cylinder and the piston, and the distance Z from the camera center to the top of the objects, to be able to find these pixels.

These pixels represented in normalized image coordinates $p = (x, y)^T$ are

$$x = \frac{fX}{Z} \quad , \quad y = \frac{fY}{Z} \tag{5.16}$$

where $X = r\cos(\theta_0 + \delta\theta)$ for the three pixels p_0 and $X = r\cos(\theta_{180} + \delta\theta)$ for the three pixels p_{180}. The parameters for the cylinder are $r = 0.024$ and $Z = 0.155$. The focal length $f = 1$ and $\delta\theta = 19°, 23°, 27°$. This is how the six pixels p_0 and p_{180} (in normalized image coordinates) are obtained. Transforming these pixels $p = (x, y)^{\mathrm{T}}$ to pixel coordinates $p = (u, v)^{\mathrm{T}}$ are done by

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5.17}$$

where K is the calibrated camera parameter matrix for the 2D camera

$$K = \begin{bmatrix} 540.927271 & 0 & 320 \\ 0 & 540.264582 & 240 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.18}$$

The grey scale values for the pixels p_{180} were 51, 251 and 250, and the grey scale values for the pixels concerning p_0 were 231, 246 and 230. The program estimates which of the pixels p_0 or p_{180} that have the most similar grey scale values. In this case the pixels p_0 have the most similar grey scale values (231, 246 and 230) which make the program accept θ_0 as the true orientation of the cylinder. The program then list an array with the last 7 "true" orientations, and if the estimated orientations are stable, the program approve an orientation θ for the cylinder. The final orientation θ of the cylinder is equal to the median of the 7 last "true" estimated orientations.

The program estimates that $\theta = 30.679°$. This is shown in Figure 5.49 and the green line visualize the estimated orientation $\theta = 30.679°$.

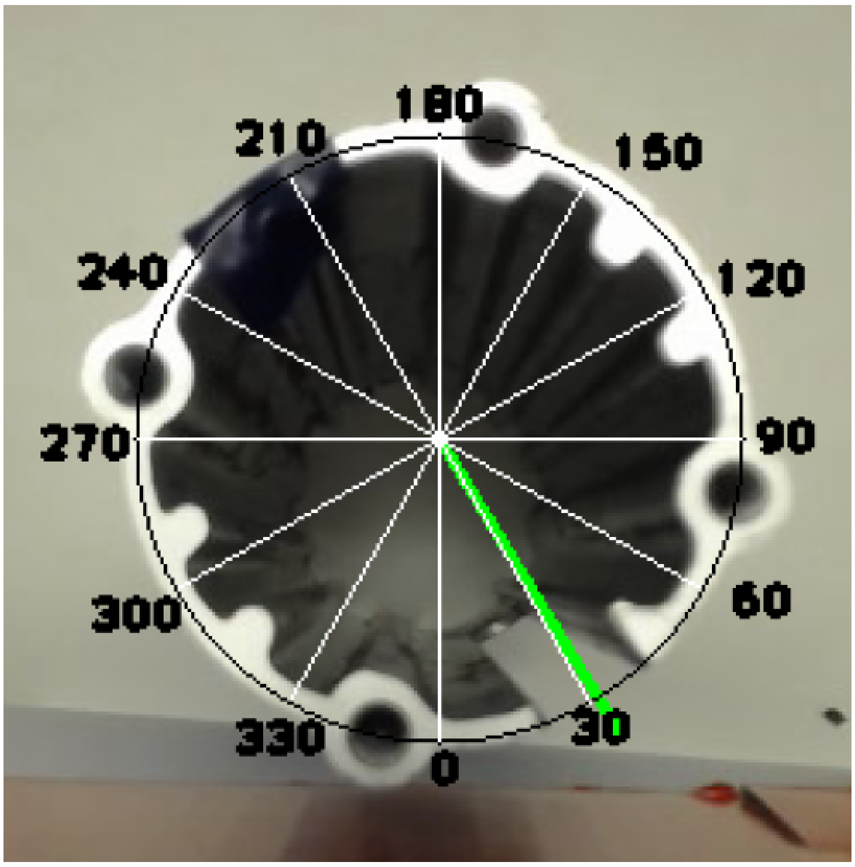


Figure 5.49: Orientation $\theta = 30.679°$ of the cylinder.

The program for the piston is quite similar to the program that have just been explained except for a few differences. The program for the piston has different training images, distance Z between the camera center and the object, the radius of the object, how to choose pixels to distinguish between orientations that are true and false (180° phase shifted) and the visualization part in the program.

A video for demonstrating how these programs work have also been made. In the rest of the thesis, the notations θ_c is used for the orientation of the cylinder and θ_p for the orientation of the piston. That is, these programs estimates a θ_p and a θ_c.

It was documented good accuracy of the estimated orientations θ_p and θ_c. Still, the programs are depending on the parts being in accurate positions with respect to the 2D camera. That is, with a $Z = 0.155$ for the cylinder and $Z = 0.205$ for the piston in equation (5.16), which was used in the demo video, the cylinder could have a maximum error (in the $X\ Y$ plane) of 2 mm and the piston 4 mm, from the center of the image stream. These maximum allowable errors were measured with a tape measure. Adjusting the distance between the camera and the parts to $Z = 0.055$ for the cylinder and $Z = 0.115$ for the piston, allowed the maximum error to be approximately 4.5 mm for the cylinder and 6 mm for the piston.

Chapter 6

Kinematics

This chapter presents the kinematics used in this project.

Two identical KUKA AGILUS KR 6 R900 SIXX manipulators have been used in this project. One of the manipulators is used for grasping the objects, while the other manipulator is used for estimating the orientations of the objects. In Figure 6.1 the manipulator used for grasping the objects is shown. This manipulator has a gripper connected to the end-effector.

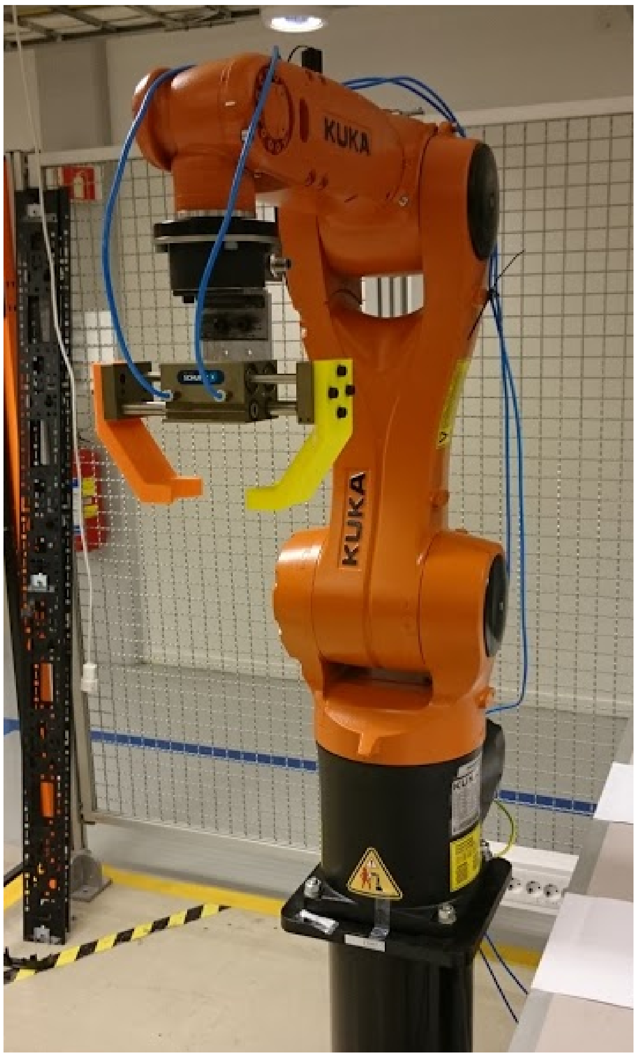


Figure 6.1: Manipulator with a gripper connected to end-effector.

In Figure 6.2 the manipulator used for estimating the orientations of the objects is shown. This manipulator has a 2D camera connected to the end-effector.

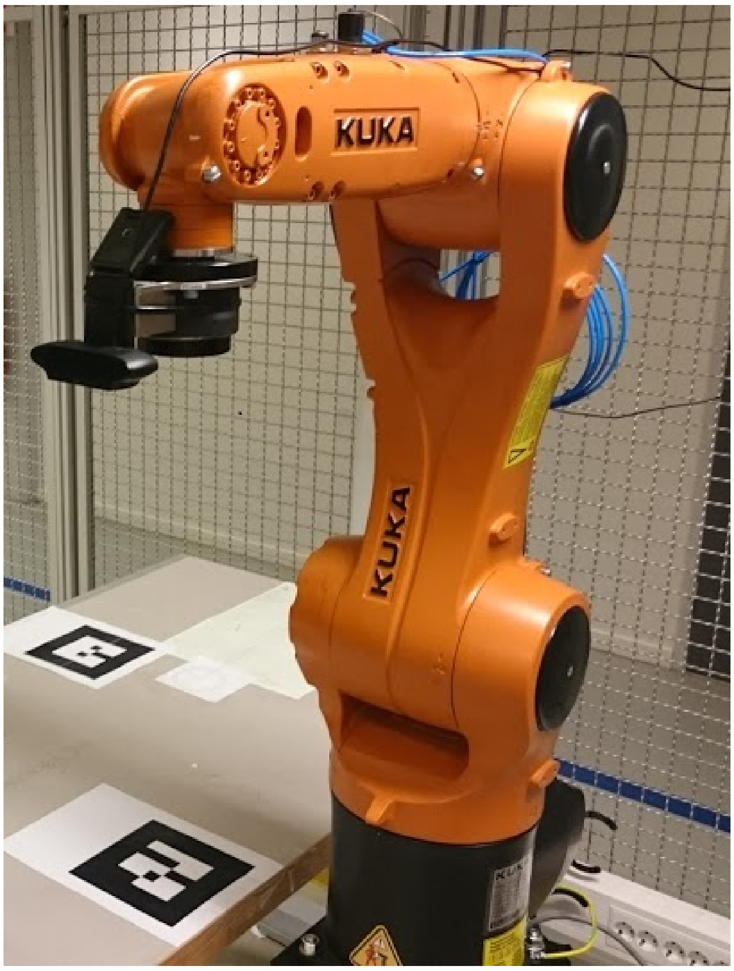


Figure 6.2: Manipulator with a 2D camera connected to end-effector.

Only forward kinematics have been used since the program operates in world coordinates. Solving of the inverse kinematics is described in Chapter 7.

6.1 Robot Forward Kinematics

A serial-link manipulator comprises a set of bodies, called links, in a chain and connected by joints. Each joint have one degree of freedom, either translation (a prismatic joint) or rotational (a revolute joint).

A systematic way of describing the geometry of a serial chain of links and joints was proposed by Denavit and Hartenberg and is known today as Denavit-Hartenberg notation.

A KUKA Agilus manipulator has 6 revolute joints numbered from 1 to 6, there are 7 links, numbered from 0 to 6. The direction of rotation of the joints, are shown in Figure 6.3.

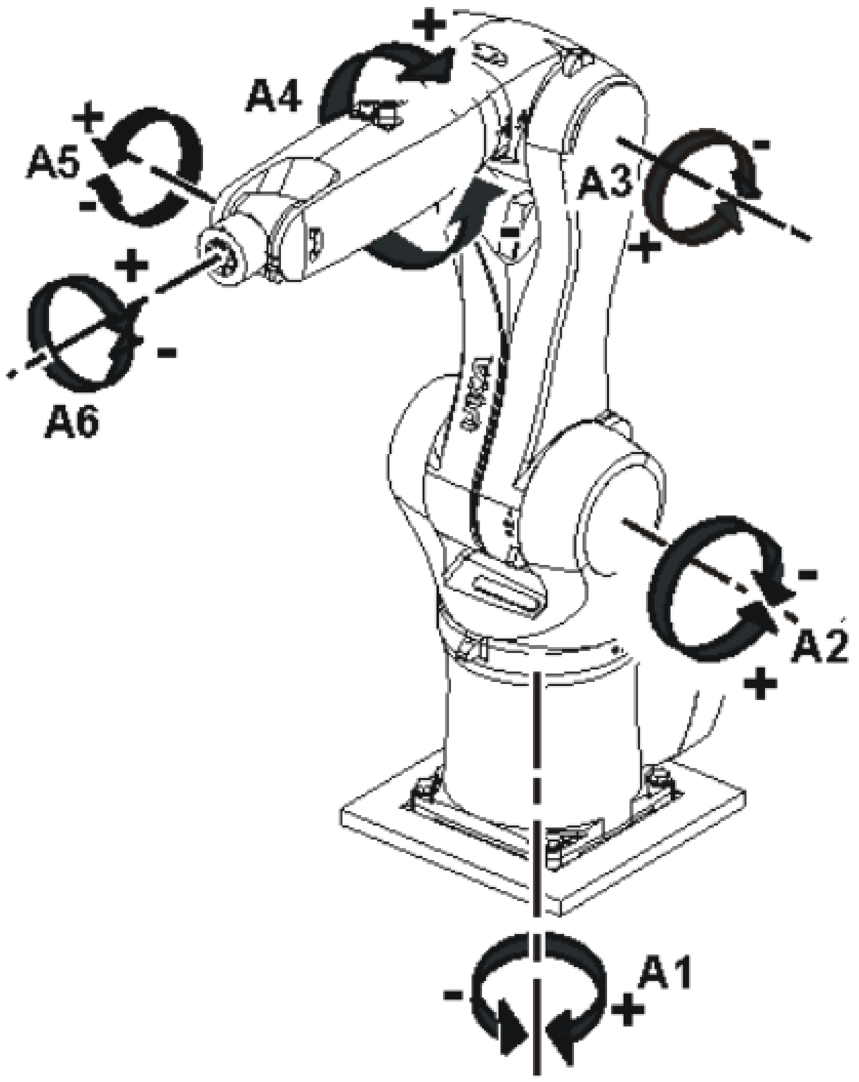


Figure 6.3: Direction of rotation of robot axes [21]

Link 0 is the base of the manipulator and link 6 carries the end effector. Joint j connects link j-1 to link j and therefore joint j moves link j. This is shown in Figure 6.4. A link can be specified by its length a_j and its twist α_j. The link offset d_j is the distance from one link coordinate frame to the next along the axis of the joint. The joint angle θ_j is the rotation of one link with respect to the next about the joint axis. The coordinate frame $\{j\}$ is attached to the end of link j. The axis of joint j is aligned with the Z axis [2].

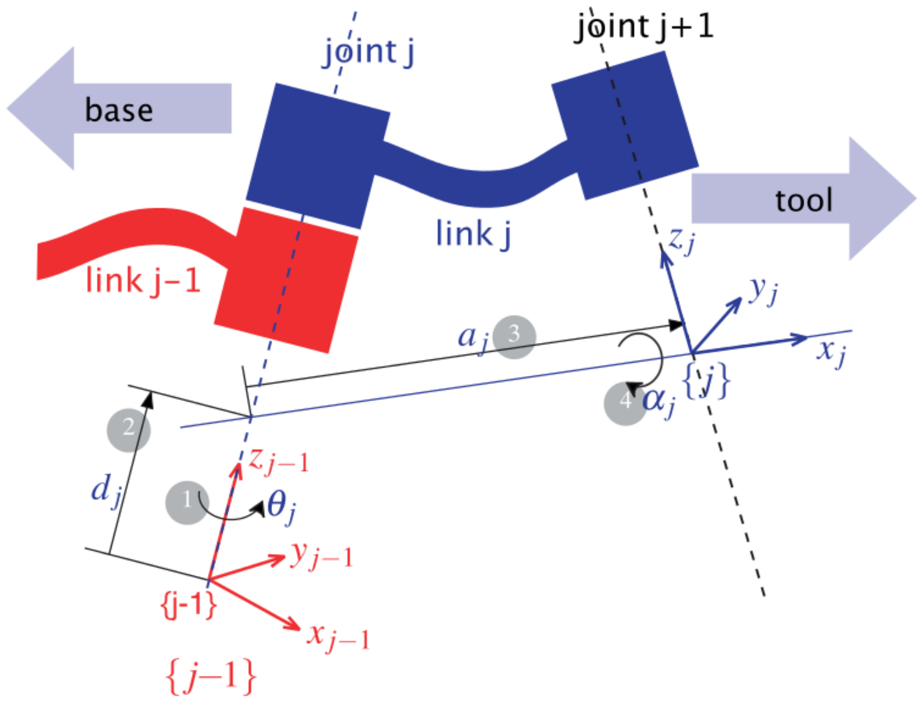


Figure 6.4: Definition of standard Denavit and Hartenberg link parameters [2].

These link and joint parameters are known as Denavit-Hartenberg parameters [2]. The Denavit-Hartenberg parameters for the KUKA Agilus is shown in Table 6.1 and in Figure 6.5. The joint variables θ_j^* where $j \in \{1...6\}$ are the different joint angles between the x_{j-1} and x_j axes about the z_{j-1} axis in Figure 6.4.

Link	θ_j[rad]	d_j[mm]	a_{j-1}[mm]	α_{j-1}[rad]
1	θ_1^*	-400	-25	$\frac{-\pi}{2}$
2	θ_2^*	0	315	0
3	θ_3^*	0	35	$\frac{\pi}{2}$
4	θ_4^*	-365	0	$\frac{-\pi}{2}$
5	θ_5^*	0	0	$\frac{-\pi}{2}$
6	θ_6^*	-80	0	π

Table 6.1: Denavit-Hartenberg parameters for KUKA Agilus.

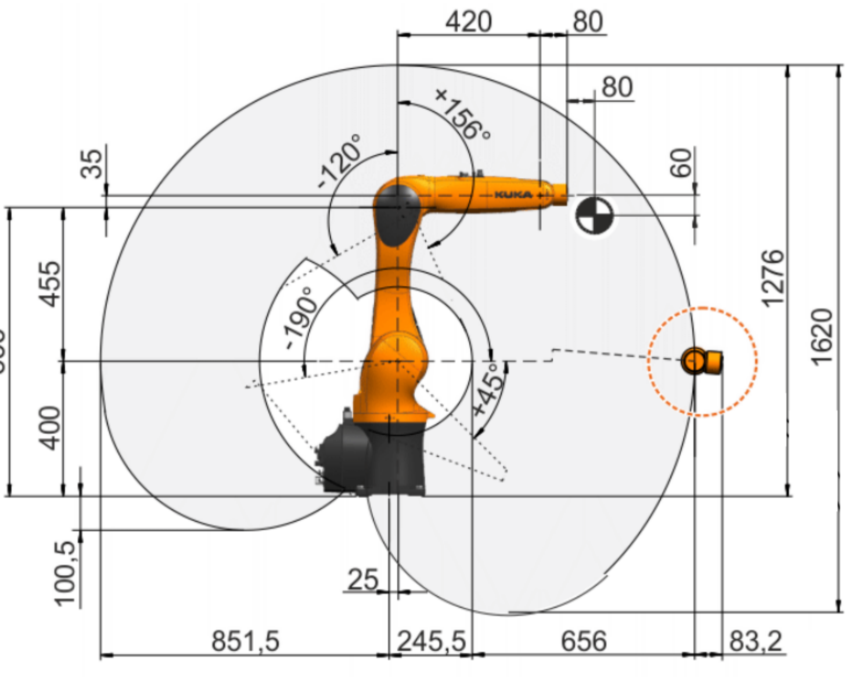


Figure 6.5: KUKA Agilus KR 6 R900 sixx [21].

The transformation from link coordinate frame $\{j-1\}$ to frame $\{j\}$ is defined as

$$T_{\mathrm{j}}^{\mathrm{j-1}}(\theta_j, d_j, a_j, \alpha_j) = T_{Rz}(\theta_j)T_z(d_j)T_x(a_j)T_{Rx}(\alpha_j) \tag{6.1}$$

which can be expanded as

$$T_j^{j-1} = \begin{bmatrix} \cos\theta_j & -\sin\theta_j\cos\alpha_j & \sin\theta_j\sin\alpha_j & a_j\cos\theta_j \\ \sin\theta_j & \cos\theta_j\cos\alpha_j & -\cos\theta_j\sin\alpha_j & a_j\sin\theta_j \\ 0 & \sin\alpha_j & \cos\alpha_j & d_j \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.2}$$

The forward kinematics T_e^0 is expressed as a function with the end-effectors pose with respect to the base of the robot $\{0\}$, as a function of the joint variables

$$T_{\mathrm{e}}^0(\theta_1^*, \theta_2^*, \theta_3^*, \theta_4^*, \theta_5^*, \theta_6^*)$$

Using homogeneous transformations, this is the product of the individual link transformation matrices given by equation (6.2)

$$T_{\mathrm{e}}^0 = T_1^0 T_2^1 T_3^2 T_4^3 T_5^4 T_6^5. \tag{6.3}$$

In ROS (Robot Operating System), both robots operates in a common world coordinate system $\{w\}$. That is, there have been implemented transformations from a common world coordinate system $\{w\}$ to both robots' bases. In ROS, both poses of the end-effector operates with respect to world coordinate system $\{w\}$.

6.2 Kinematics Result

6.2.1 Robot with a 2D Camera Connected to End-Effector

The robot with a 2D camera attached to the end effector, is the robot that are used in the orientation analysis described in Chapter 5.2.1. The orientation analysis is run when the robot's end effector is at a certain pose with respect to the objects.

Pose of the Cylinder

The transformation $T^{e}_{c_{2D}}$ from the end-effector to the 2D camera $\{c_{2D}\}$ is

$$T^{e}_{c_{2D}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -0.028 \\ 0 & 0 & 1 & 0.01 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.4}$$

Figure 6.6 shows the pose of the 2D camera $\{c_{2D}\}$ when the program for estimating the orientation θ_c of the cylinder is run. θ_c and θ_p are the orientations around the Z axis of the objects, which were described in Chapter 5.2. The Z axis of the objects is parallel to the Z axis of the table coordinate system $\{t\}$ shown in Figure 6.6.

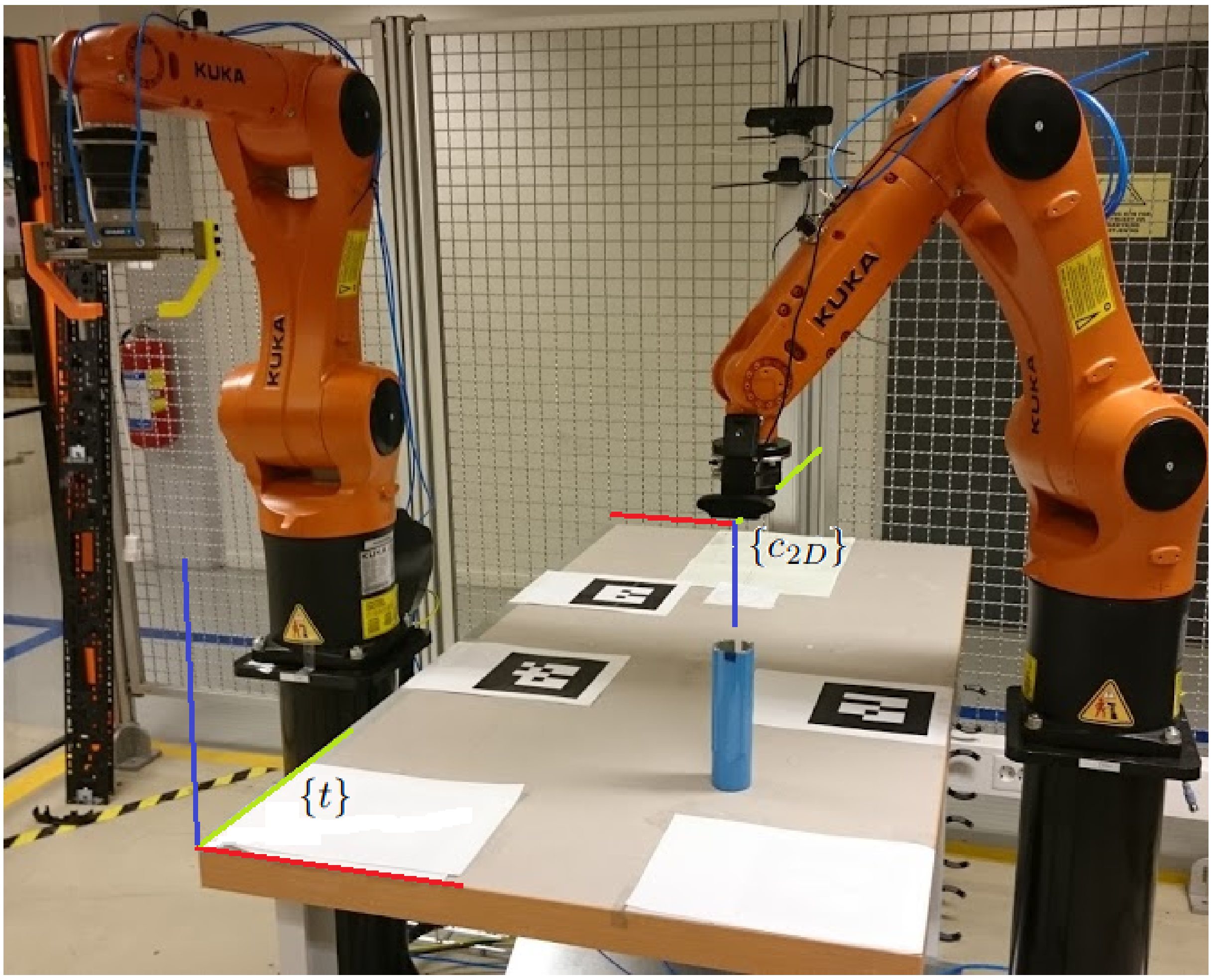


Figure 6.6: The table coordinate system $\{t\}$ and the 2D camera coordinate system $\{c_{2D}\}$.

The rotation matrix $R^t_{c_{2D}}$ from the table $\{t\}$ to the 2D camera $\{c_{2D}\}$ is a 180° rotation around the Y axis as shown in Figure 6.6.

$$R^t_{c_{2D}} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{6.5}$$

As shown in the Figure 6.7, the transformation ${}^{\theta_c}T^t_{c_{2D}}$ from table $\{t\}$ to $\{c_{2D}\}$ when running the program for estimating θ_c is

$${}^{\theta_c}T^t_{c_{2D}} = \begin{bmatrix} R^t_{c_{2D}} & p^t_{\text{cyl}} + \delta p_{\text{cyl}} \\ 0^T & 1 \end{bmatrix} \tag{6.6}$$

where $\delta p_{\text{cyl}} = (0, 0, 0.055)^{\text{T}}$ is the distance between the top of the cylinder and optical center of the 2D camera. The coordinates p^t_{cyl} of the cylinder with respect to the table pose $\{t\}$ was estimated in Chapter 5.1.4.

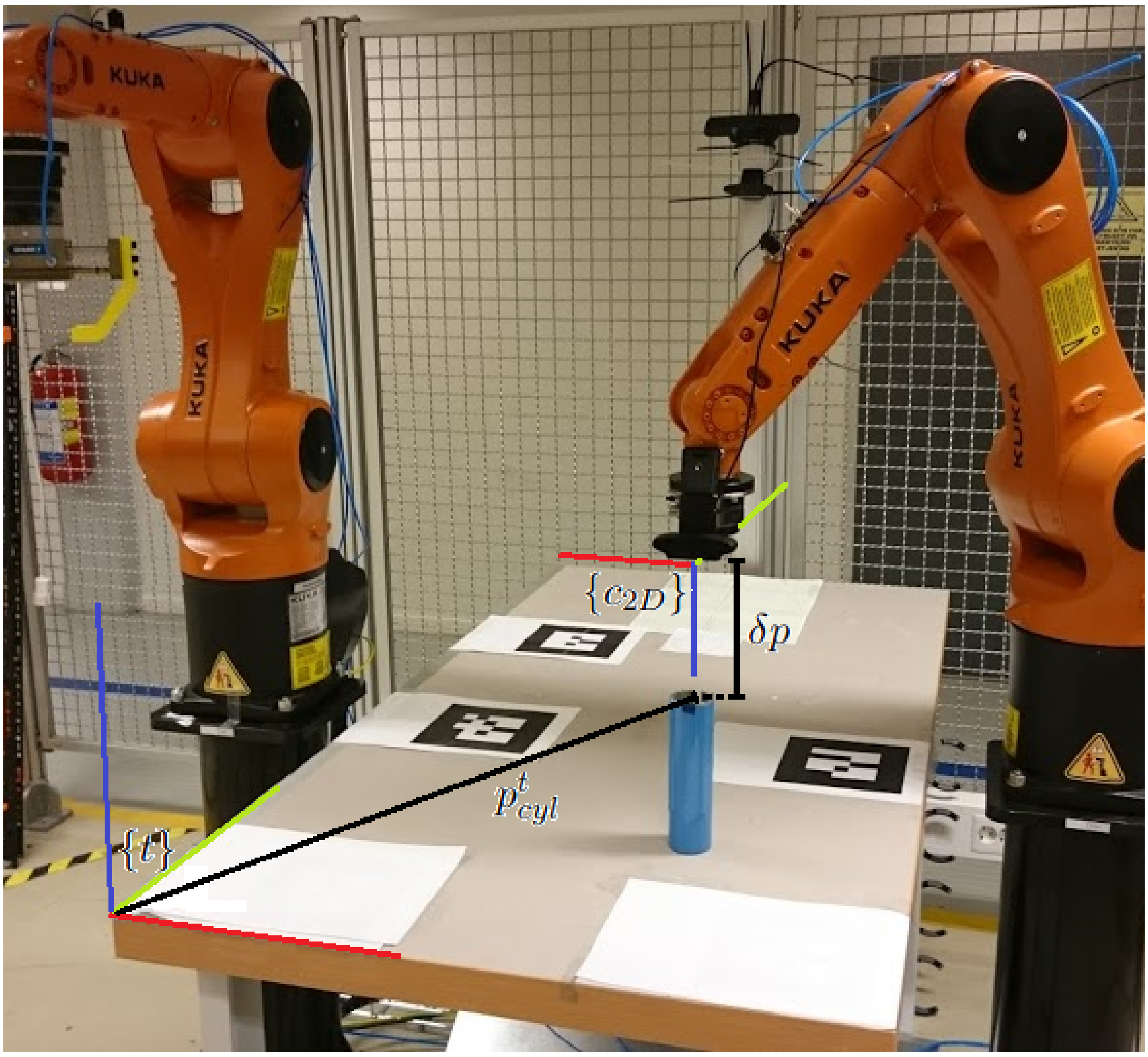


Figure 6.7: Transformation $T^t_{c_{2D}}$ from table pose $\{t\}$ to 2D camera pose $\{c_{2D}\}$.

The transformation T^w_t from world $\{w\}$ (the coordinate system both robots are operating in) to table $\{t\}$ was measured to be a rotation $R_z(90°)$ and a translation $t = (0.67562, -0.39624, 0.938)$.

$$T^w_t = \begin{bmatrix} 0 & -1 & 0 & 0.67562 \\ 1 & 0 & 0 & -0.39624 \\ 0 & 0 & 1 & 0.938 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.7}$$

The end-effector pose $T^w_{e,\theta c}$ when doing the orientation analysis of the cylinder is

$$T^w_{e,\theta_c} = T^w_t \, {}^{\theta_c}T^t_{c_{2D}} (T^e_{c_{2D}})^{-1} \tag{6.8}$$

were $T^e_{c_{2D}}$ was estimested in equation (6.4), ${}^{\theta_c}T^t_{c_{2D}}$ in equation (6.6) and T^w_t in equation (6.7).

When the robot's end effector is at T^w_{e,θ_c} the program for estimating θ_c is run. In Figure 6.8 the X and Y axis of the pose of the cylinder is shown. The rotation matrix R^t_{cyl} from table $\{t\}$ to cylinder is an identity matrix when $\theta_c = 0°$.

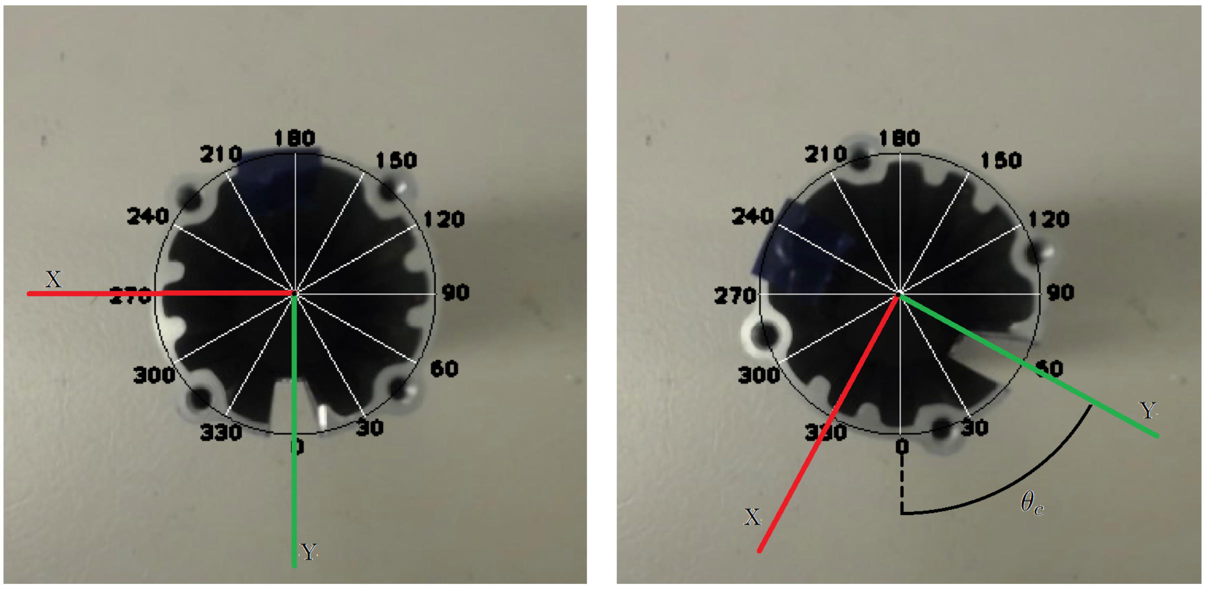


Figure 6.8: From left: Orientation of the cylinder is $\theta_c = 0°$ and $\theta_c = 60°$.

Since the Z axis of table $\{t\}$ and the cylinder (and piston) is parallel, the rotation matrix R^t_{cyl} is a rotation θ_c around the Z axis

$$R^t_{\text{cyl}} = R_z(\theta_c) \tag{6.9}$$

The pose of the cylinder T^t_{cyl} with respect to the table $\{t\}$ is

$$T^t_{\text{cyl}} = \begin{bmatrix} R^t_{\text{cyl}} & p^t_{\text{cyl}} \\ 0^{\text{T}} & 1 \end{bmatrix} \tag{6.10}$$

The pose of the cylinder T^t_{cyl} with respect to the world coordinate system $\{w\}$ is

$$T^w_{\text{cyl}} = T^w_t T^t_{\text{cyl}} \tag{6.11}$$

were T^w_t was estimated in equation (6.7) and T^t_{cyl} in equation (6.10).

T^w_{cyl} is the pose of the cylinder with respect to coordinate system both robots are operating in.

Pose of the Piston

The end-effector pose $T^w_{e,\theta p}$ when doing the orientation θ_p analysis of the piston is

$$T^w_{\mathrm{e},\theta p} = T^w_t \underbrace{\begin{bmatrix} R^t_{c_{2D}} & p^t_{\mathrm{pis}} + \delta p_{\mathrm{pis}} \\ 0^{\mathrm{T}} & 1 \end{bmatrix}}_{^{\theta_p}T^t_{c_{2D}}} (T^{\mathrm{e}}_{c_{2D}})^{-1} \tag{6.12}$$

where $\delta p_{\mathrm{pis}} = (0, 0, 0.115)$ is the distance between the top of the piston and the optical center of the 2D camera and $^{\theta_c}T^t_{c_{2D}}$ is the transformation from table $\{t\}$ to $\{c_{2D}\}$ when running the program for estimating θ_p. The transformation T^w_t was estimated in equation (6.8), $T^e_{c_{2D}}$ in equation (6.4) and the rotation matrix $R^t_{c_{2D}}$ in equation (6.5). The coordinates p^t_{pis} of the piston with respect to the table $\{t\}$ was estimated in Chapter 5.1.4.

When the end-effector is at the pose $T^w_{e,\theta p}$ the program for estimating the orientation θ_p of the piston is run.

The pose T^w_{pis} of the piston with respect to world $\{w\}$ is

$$T^w_{\mathrm{pis}} = T^w_t \begin{bmatrix} R_z(\theta_{\mathrm{pis}}) & p^t_{\mathrm{pis}} \\ 0^{\mathrm{T}} & 1 \end{bmatrix}$$

The transformations in equations (6.12) and (6.8) are estimated forward kinematics for the robot with 2D camera.

6.2.2 Robot with a Gripper Connected to End-Effector

The pose of the parts T^w_{pis} and T^w_{cyl} are estimated and the robot with a gripper connected to the end-effector, have enough information to assemble the parts.

The transformation T^g_e from gripper to end-effector is

$$T^g_e = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -0.16 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.13}$$

In Figure 6.9 the pose of the gripper $\{g\}$ is shown. The transformation T^{cyl}_g from the cylinder to the gripper (when the gripper is grabbing the cylinder) is a 180° rotation around the Y axis

$$T^{\mathrm{cyl}}_g = \begin{bmatrix} R_y(180^\circ) & 0 \\ 0^{\mathrm{T}} & 1 \end{bmatrix} \tag{6.14}$$

The same applies to the transformation T^{pis}_g from the piston to the gripper

$$T_g^{\text{pis}} = \begin{bmatrix} R_y(180°) & 0 \\ 0^{\text{T}} & 1 \end{bmatrix} \tag{6.15}$$

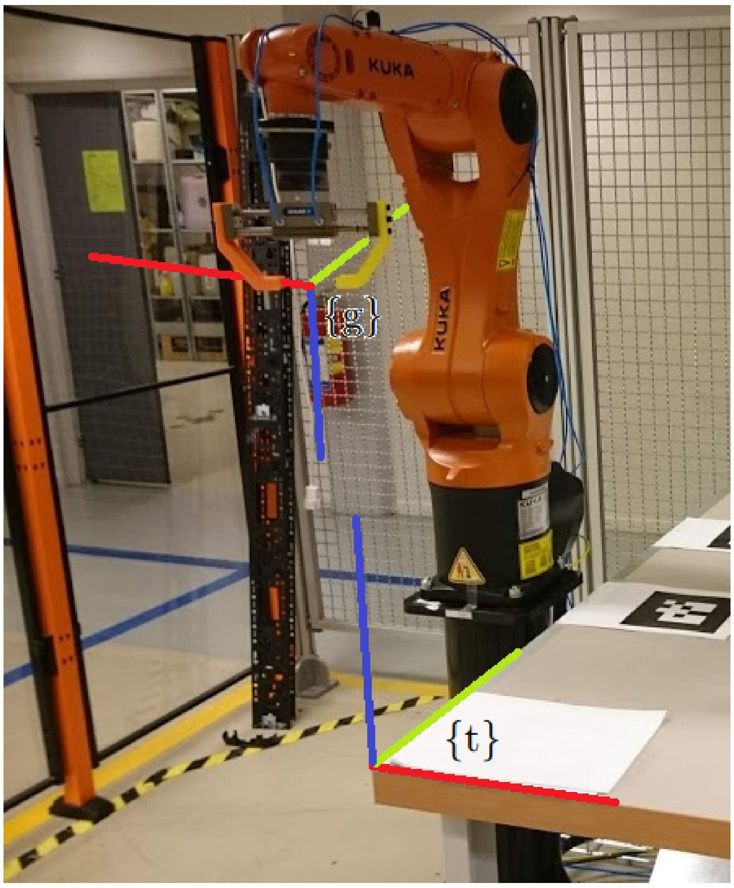


Figure 6.9: The robot gripper pose $\{g\}$ and the table pose $\{t\}$.

The transformation T_e^{cyl} from the cylinder to end effector (when the gripper is grabbing the cylinder) is

$$T_e^{\text{cyl}} = T_g^{\text{cyl}} T_e^g \tag{6.16}$$

and transformation T_e^{pis} from the piston to end effector is

$$T_e^{\text{pis}} = T_g^{\text{pis}} T_e^g \tag{6.17}$$

The final pose $T_{e_{\text{pis}}}^w$ of the end-effector with respect to world $\{w\}$, when the gripper is grabbing the piston is

$$T_{e_{\text{pis}}}^w = T_{\text{pis}}^w T_e^{\text{pis}} \tag{6.18}$$

and when the gripper $T^w_{e_{\mathrm{cyl}}}$ is grabbing the cylinder is

$$T^w_{e_{\mathrm{cyl}}} = T^w_{\mathrm{cyl}} T^{\mathrm{cyl}}_e \tag{6.19}$$

This is shown in Figure 6.10. The transformations in equations (6.18) and (6.19) are the estimated forward kinematics for the robot with a gripper connected to end-effector.

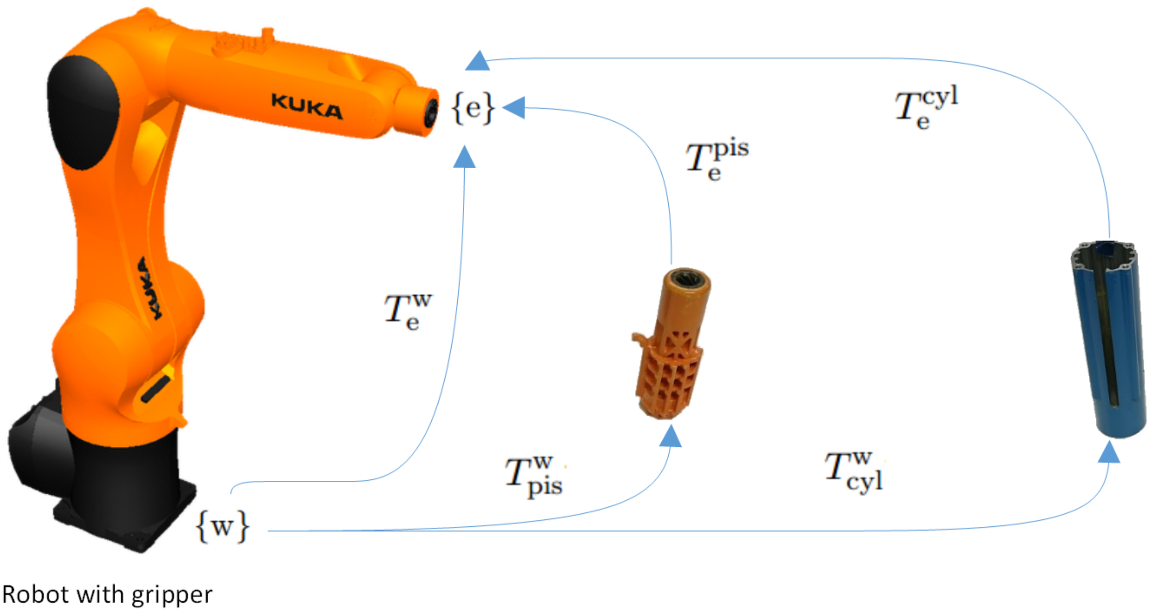

{e}
$T_{\mathrm{e}}^{\mathrm{cyl}}$
$T_{\mathrm{e}}^{\mathrm{pis}}$
$T_{\mathrm{e}}^{\mathrm{w}}$
$T_{\mathrm{pis}}^{\mathrm{w}}$
$T_{\mathrm{cyl}}^{\mathrm{w}}$
{w}
Robot with gripper

Figure 6.10: Transformations between the robot and the parts (cylinder and piston).

6.2.3 The Kinematics Sequence

The sequence of the program runs as follows:

Robot with 2D camera:

1. $T^w_e = T^w_{e,\theta c}$ (equation (6.8))
2. Run algorithm for estimating the cylinder orientation θ_c
3. $T^w_e = T^w_{e,\theta p}$ (equation (6.12))
4. Run algorithm for estimating the piston orientation θ_p
5. Run robot to home position

Robot with gripper:

1. gripper open
2. $T^w_e = \delta T\, T^w_{e_{\mathrm{pis}}}$
3. $T^w_e = T^w_{e_{\mathrm{pis}}}$ (equation (6.18))
4. gripper close

5. $T_e^w = \delta T \, T_{e_\text{pis}}^w$
6. $T_e^w = \delta T \, T_{e_\text{cyl}}^w$
7. $T_e^w = T_{e_\text{cyl}}^w$ (equation (6.19))

where

$$\delta T = \begin{bmatrix} I & t \\ 0^\text{T} & 1 \end{bmatrix} \quad \text{and} \quad t = [0, 0, 0.1]^\text{T}$$

There have been made a video demonstrating the sequence of the robot assembling the parts with θ_c and θ_p set equal $0°$.

In Figure 6.11 the two robots are shown.

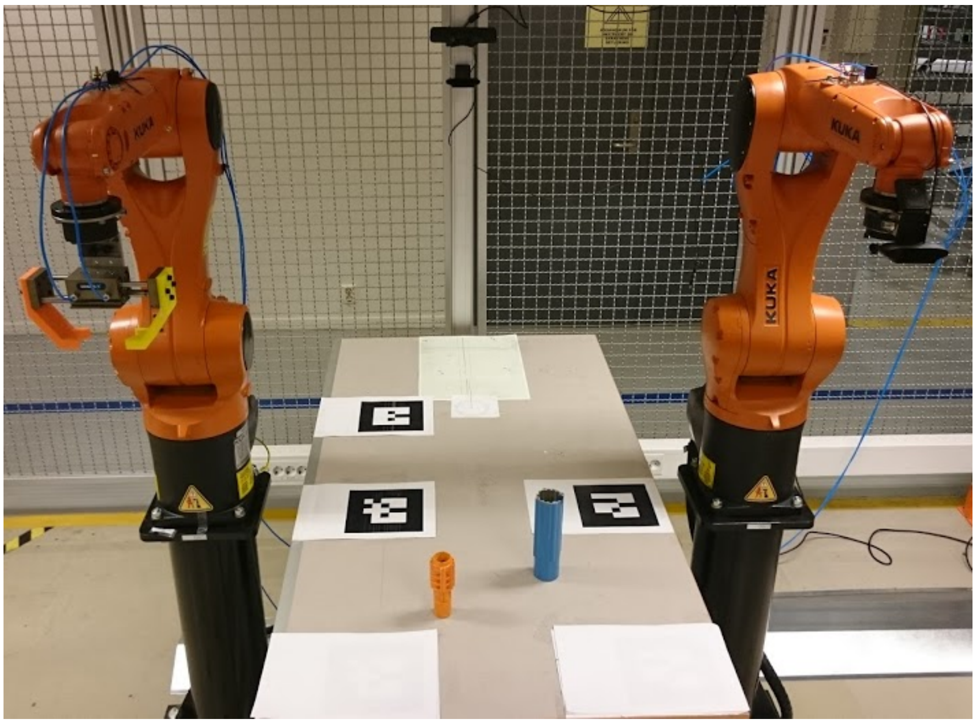


Figure 6.11: The two manipulators and their work space.

Chapter 7

ROS

ROS (Robot Operating System) is a framework for writing robot software. It is a collection of tools, libraries, and conventions for creating robot behavior across a wide variety of robotic platforms [22].

7.1 The Robots

Figure 7.1 shows a KUKA AGILUS KR 6 R900 SIXX robot. The robot consist of a manipulator (1), a teach pendant (2), connection cable for the teach pendant (3), a KUKA KR C4 controller (4), data cable (5) and motor cable (6).

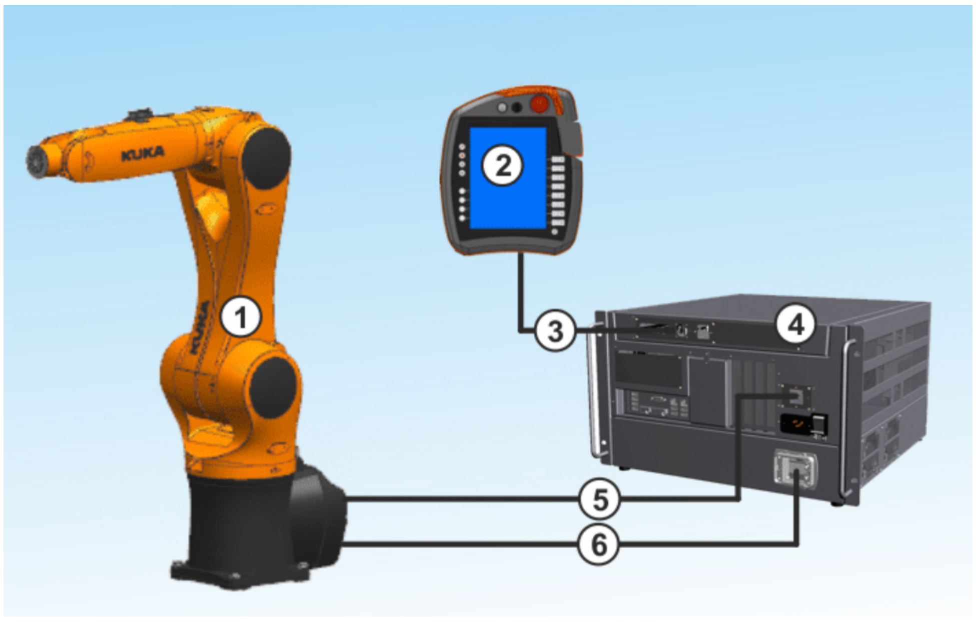


Figure 7.1: Description of the Kuka Agilus robot [9].

The KUKA Robot Sensor Interface (RSI) is the program that passes information between the KUKA KR C4 controller and the computer with ROS. This information is passed over the Ethernet UDP/IP protocol as XML (Extensible Markup Language) strings.

Figure 7.2 shows the information flow for the robot with a 2D camera connected to end-effector.

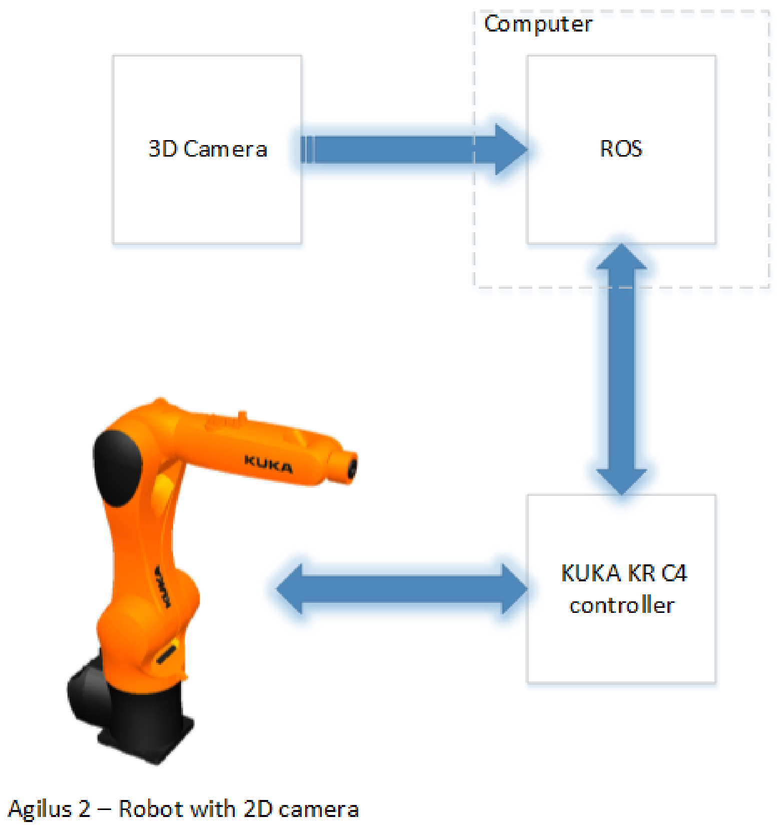


Figure 7.2: Robot with a 2D camera connected to end-effector.

Figure 7.3 shows the information flow for the robot with a gripper connected to end-effector.

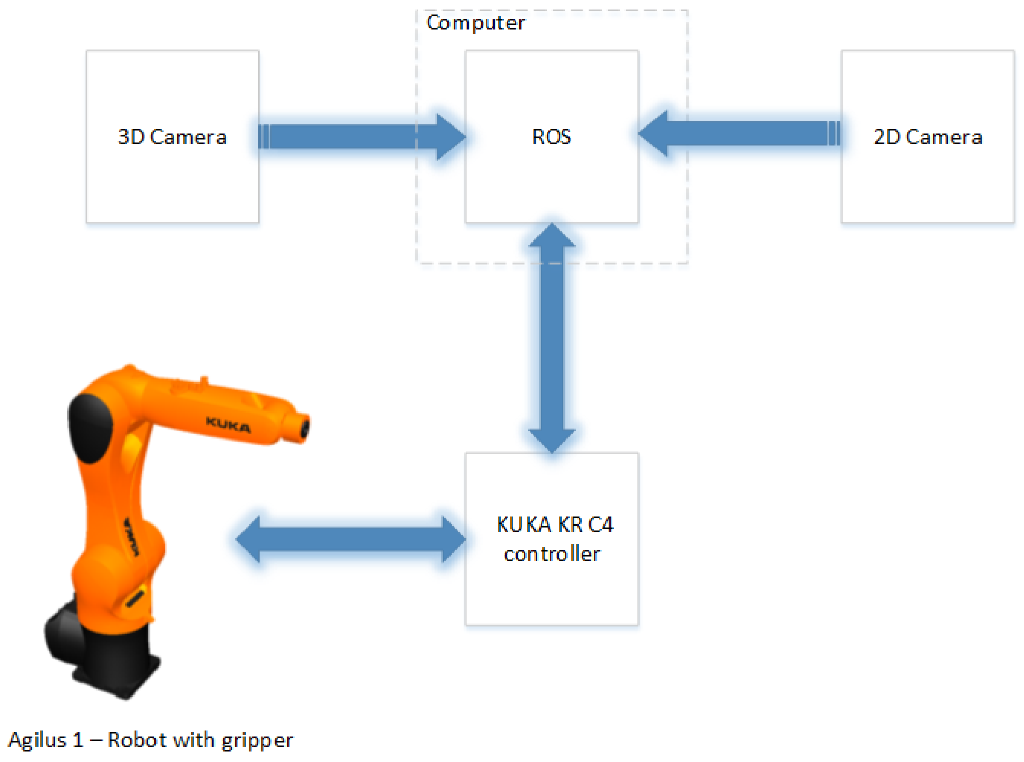


Figure 7.3: Robot with a gripper connected to end-effector.

7.2 Implementation of RGB-D Camera

Software in ROS is organized in *packages*. A package might contain among others, ROS *nodes*, a ROS-independent library, a dataset and configuration files. A node is an executable that uses ROS to communicate with other nodes [22].

In this project the ROS packages *Openni_camera*, *Openni2_launch* and *Ar_track_alvar* have been used in relation with the depth camera. *openni_camera* provides a ROS interface to the ASUS Xtion Pro Live. *openni2_launch* contains launch files for using OpenNI-compliant devices in ROS and *ar_track_alvar* is a ROS wrapper for Alvar, an open source AR tag tracking library [22].

Figure 7.4 shows a simplified version of the ROS program for estimating coordinates of the objects. The ellipses are ROS nodes and the rectangles represents ROS topics. A node can publish messages to a topic as well as subscribe to a topic to receive messages. A message is a ROS data type [22]. In this chapter, names for ROS nodes and ROS topics have been simplifies in contrast to the names in the program.

The program runs as follows:

1. The node *3D Camera* publishes a point cloud to the topic *Point cloud*.
2. The node *AR TAG* subscribes to *Point cloud* and publishes the center points of the AR-tags on the topic *TAG coordinates*.
3. The node *TF point cloud* subscribes to *TAG coordinates* and publishes a homogeneous transformation from camera frame to table frame on the topic *TF camera table*.
4. The node *Coordinates objects* subscribes to *Point cloud* and *TF point cloud*, and publishes the coordinates of the objects with respect to table coordinate system on the topics *Coordinates cylinder* and *Coordinates piston*.

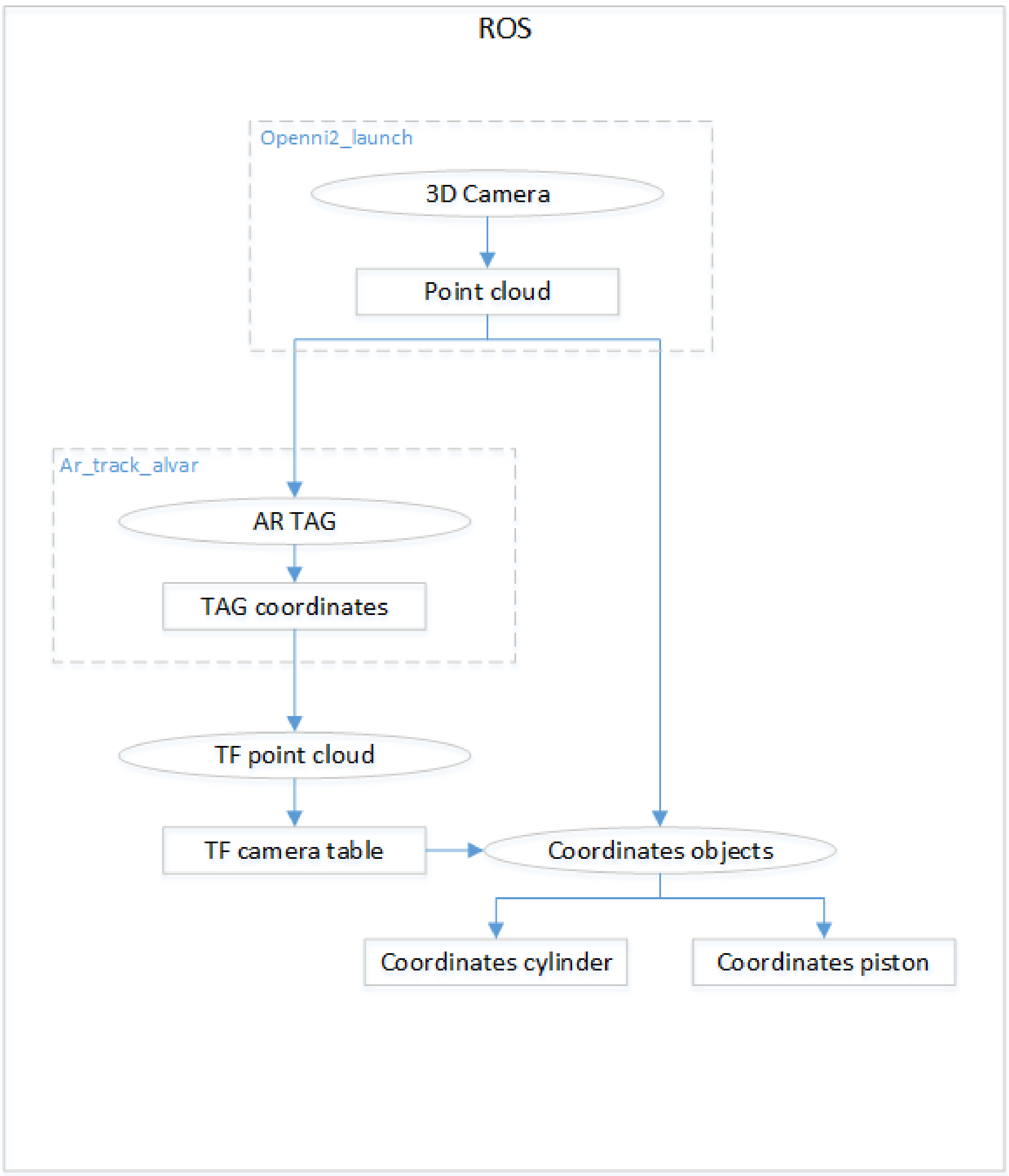


Figure 7.4: RGB-D camera in ROS.

7.3 Implementation of Robot with a 2D Camera Connected to End-Effector

Figure 7.5 shows a simplified version of the ROS program for running the robot with 2D camera (The names used in the figure are also simplified). The node *Agilus 2* is the interface to the KUKA KR C4 controller. *Joint trajectory ag2* provides an action interface for tracking trajectory execution.

MoveIt! is a program used for planing and execution of trajectories. It will among others look on joint limits, kinematics and motion planning. MoveIt! talks to the robot through the ROS topics and ROS actions. It communicates with the robot to get current state information (positions of the joints, etc.) to get point clouds and other sensor data from the robot sensors and to talk to the controllers on the robot. [12]. ROS actions is used when someone would like to send a request to a node and also receive a reply to the request. ROS actions also gives the ability to cancel the request if it takes a long time to execute. The ROS actions are visualized (in Figure 7.5) as boxes that are connected to lines with arrows in both ends.

The program runs as follows:

1. The node *Agilus 2* publihes information about the joint angles to the topic *Joint states*.

2. The node *Forward kinematics* subscribes to the topic *Joint states* and publishes the 3D poses of the robot links to the topic *TF*.

3. The node *Moveit* subscribes to the topics *Joint states* and *TF* to determine the current state where each joint of the robot is and the robot's pose.

4. A user defined request is published from the node *Pose objects* to the action topic *New pose end effector*. This user defined request is the kinematics used in points number 1 and 3 in the kinematic sequence "Robot with 2D camera" in Chapter 6.2.3. The topics *Coordinates cylinder* and *Coordinates piston* are the same as the topics shown in Figure 7.4.

5. The node *Moveit* talks to *New pose end effector*, and the KUKA KR C4 controller through the ROS action interface *Joint trajectory ag2*.

MoveIt! will generate a desired trajectory in response to the transformations on *New pose end effector*. Since the robot operates in joint coordinates, the inverse kinematics must be solved. MoveIt! uses a plug-in infrastructure. The default inverse kinematics plug-in for MoveIt! is configured using a numerical jacobian-based solver. This plug-in is automatically configured in MoveIt! [12]. The action topic *Joint trajectory ag2* passes trajectory goals to the node *Agilus 2*, and reports success when they have finished executing. It can also enforce constraints on the trajectory, and abort trajectory execution when the constraints are violated.

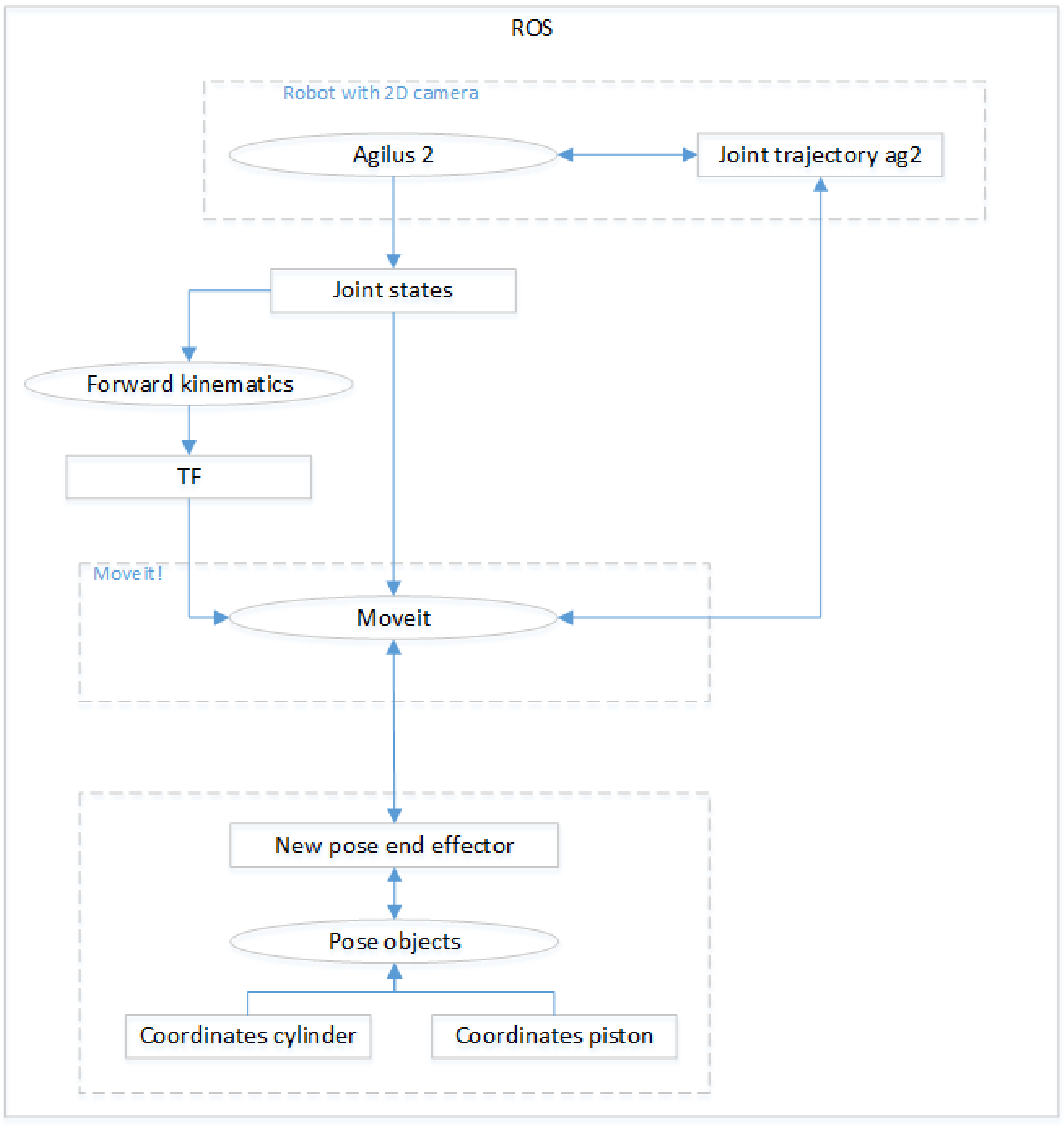


Figure 7.5: Robot with a 2D Camera connected to end-effector.

7.4 Implementation of 2D Camera

The ROS package *usb_camera* was used as ROS interface with the 2D camera. The program for the 2D camera is illustrated in Figure 7.6 and runs as follows:

1. The node *2D camera* publishes images from the 2D camera on the topic *2D image*.
2. The node *Orientation objects* subscribes to *2D image* and publishes the orientations θ_c and θ_p on the topics *Orientation cylinder* and *Orientation piston*.
3. The node *Pose objects* subscribes to the topics *Orientation piston*, *Orientation cylinder*, *Coordinates piston* and *Coordinates cylinder*. On the basis of this information, the node estimates T^w_{cyl} and T^w_{pis}.

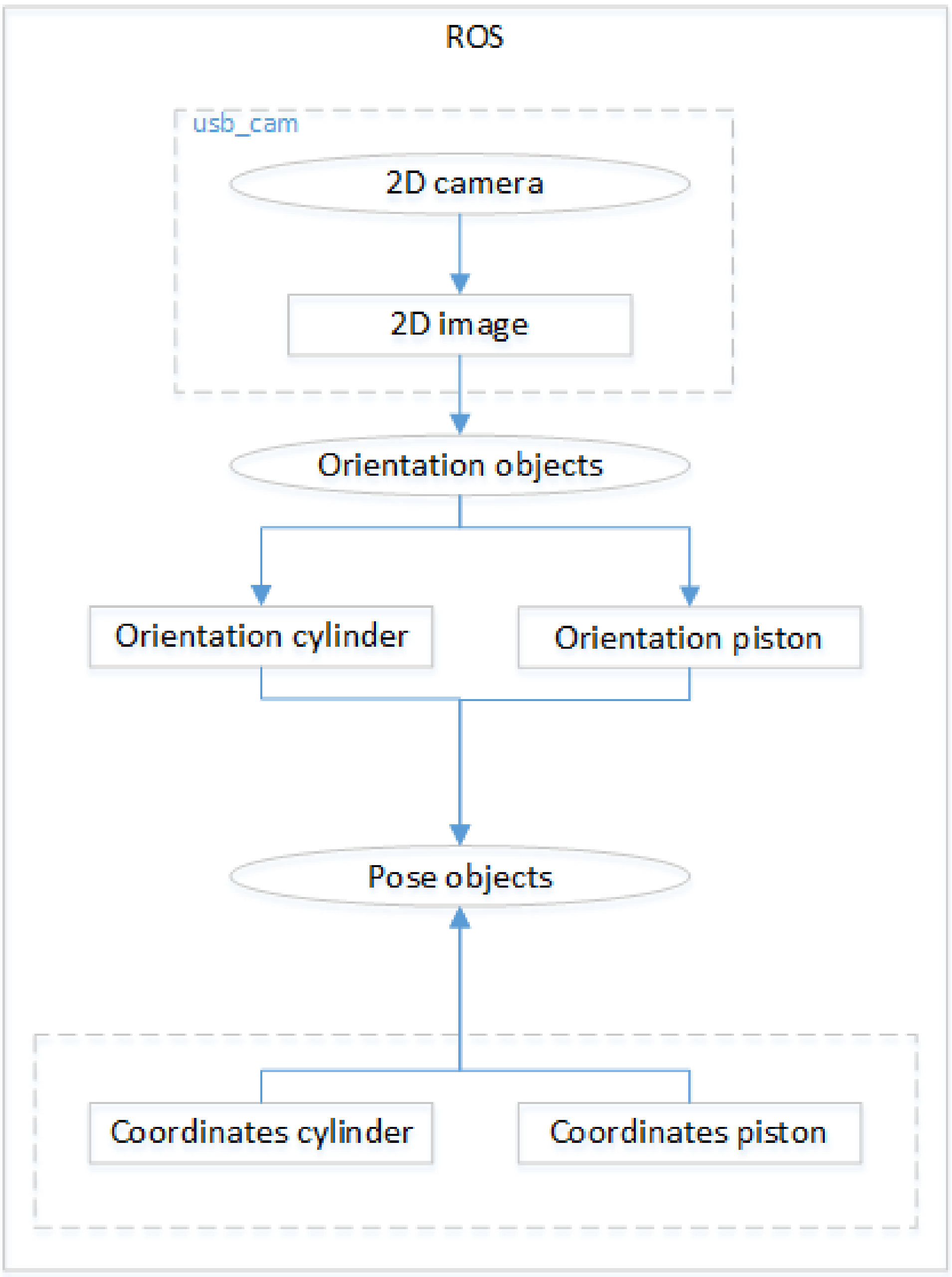


Figure 7.6: 2D camera in ROS

7.5 Implementation of Robot with a Gripper Connected to End-Effector

In Figure 7.7 the robot with gripper is shown. The ROS program for this robot is the same as for the robot with the 2D camera. The node *Pose objects* publishes the kinematics estimated in points 2, 3, 5, 6 and 7 in the kinematics sequence ”Robot with gripper” in Chapter 6.2.3, to the topic *New pose end effector*.

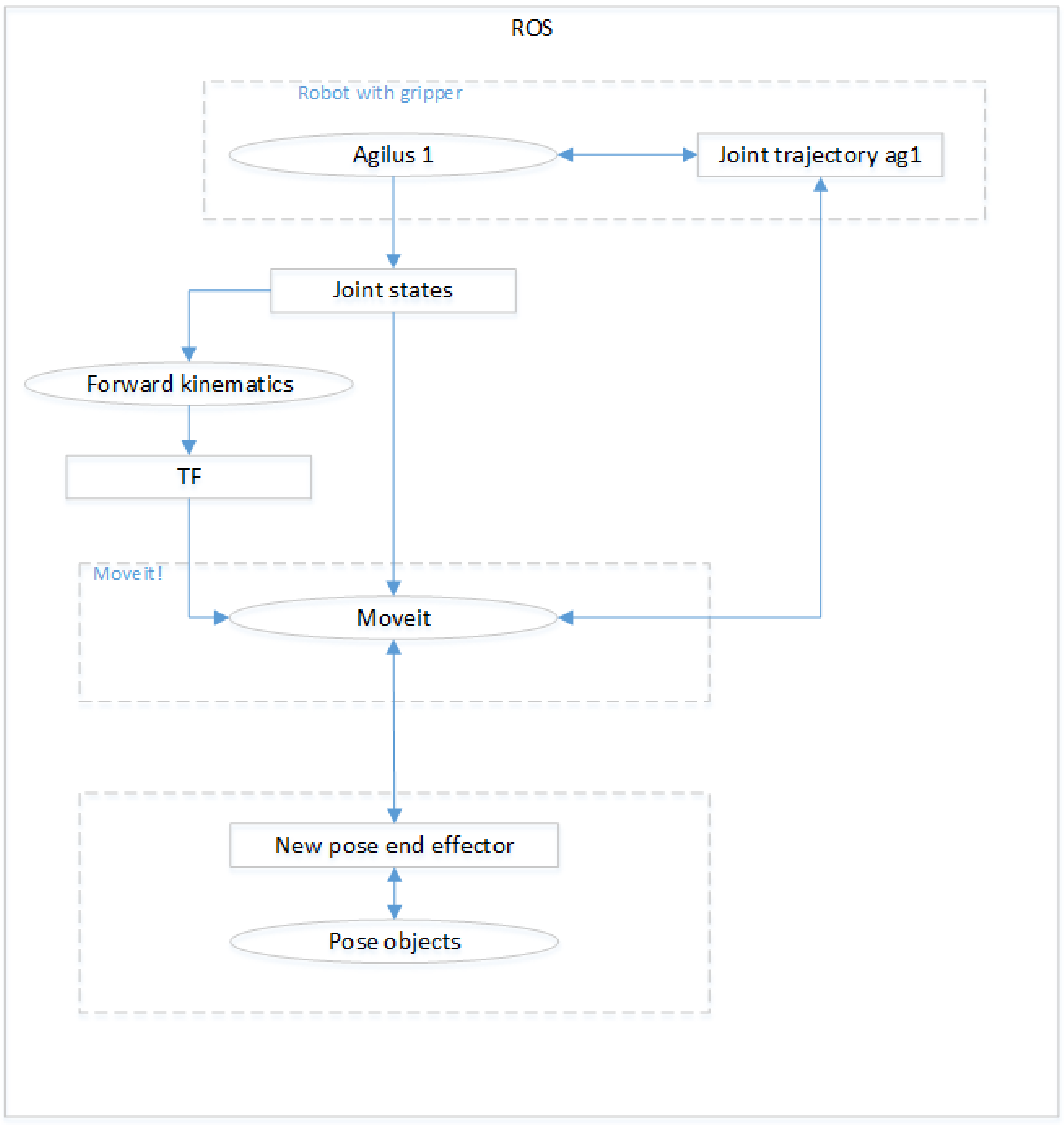


Figure 7.7: Robot with a gripper connected to end-effector

PhD. candidate Adam Leon Kleppe has configured Moveit! and the interfaces for the robots in ROS.

Chapter 8

Summary and Recommendations for Further Work

8.1 Discussion

Estimating the coordinates of the parts using the RGB-D camera proved to have an accuracy of about 4 mm which was sufficient to perform a simplified assembly task as shown in the video. The program have been further improved since the video was made, introducing a second robot with a 2D camera for estimating the orientation.

Using SIFT and homography on data from the 2D camera to estimate the orientation of the parts proved to work when the objects were placed in a region of stable illumination. The programs developed were quite complex since the geometry of the parts made them difficult to work with. The programs require the 2D camera to be vertically above the part, which can be challenging if the estimated coordinates of the parts are not correct. The maximum possible deviation of the position of the 2D camera was found to be 4.5 mm for the cylinder and 6 mm for the piston. The accuracy of the coordinates of the parts were estimated to an average error of 3.05 mm and 3.64 mm for the piston and the cylinder, respectively. The physical positions were measured using a tape measure with millimeter resolution, which adds some uncertainty. Also, the effect the illumination from the light source in the ceiling had on the objects, changed when the parts were moved around on the table. This created difficulties for the program to estimate the orientation of the parts when they were positioned on certain locations on the table.

A calibrated transformation from the IR camera to the RGB camera could have improved the calibration of the transformation T_a^c from camera to AR-tags. Still, the transformation from RGB camera to the IR camera was not calibrated due to limited information about how to do this for the ASUS Xtion PRO LIVE. Methods have been developed for calibrating this transformation for the Microsoft Kinect, but they proved not to work for the ASUS Xtion PRO LIVE. Different methods for estimating the transformation from RGB camera to the IR camera were tested without getting better results. We also tried to use different methods for localizing the objects based on the RGB data, but the RGB data for the parts was not perfectly aligned with the depth image from the IR camera so we decided to mainly use the depth data from the IR camera in the experiments. The AR-tags are the only part of the project where the RGB data have been used. The estimation of the coordinates could have been more accurate if the camera had been mounted closer to the objects of interest.

The template alignment method could have been used in combination with a method called ICP (Iterative Closest Point). The ICP algorithm can determine if the aligned template (after template alignment) is just a rigid transformation of the template by minimizing the distances between the points in the template and the corresponding points in the target cloud, and rigidly transforming them. This was not done since the template alignment program had problems to distinguish the parts.

8.2 Summary and Conclusions

In this thesis we have presented how two parts from the automotive industry can be assembled using an RGB-D camera, a 2D camera and two KUKA Agilus manipulators with six revolute joints. The RGB-D camera have been presented and calibrated, and AR-tags have been used to efficiently transform the coordinate frame of the RGB-D camera to a table. In relation to this, the use of geometric algebra to make an orthonormal coordinate frame on the table was presented. Several different methods for obtaining the position of the parts, based on data from the RGB-D camera, have been presented and evaluated. This includes a C++ program segmenting the scene based on the orientation of the surface normals in the point cloud, and a program based on known geometry of the parts. The accuracy of the coordinates of the parts were estimated to an average error of 3.05 mm and 3.64 mm for the piston and the cylinder respectively. The results obtained were not adequate for the original assembly task which require sub-millimeter accuracy, but it was possible to perform a simplified assembly task requiring 4 mm accuracy. A method for determining the orientation of the parts, using SIFT and homography, based on data from a 2D camera was presented. Two demonstration videos were made, one determining the orientation, and another one where the parts are assembled using a robot.

The final operation runs as follows:

1. The RGB-D camera is positioned at a random place with the AR-tags in the field of view.
2. An algorithm automatically determines the pose of the RGB-D camera with respect to a known coordinate frame.
3. Data from the RGB-D camera is used to estimate the coordinates of the parts with respect to the known coordinate frame.
4. A robot with a 2D camera connected to the end-effector, moves to a pose where the 2D camera is vertically above the parts, one at a time.
5. Data from the 2D camera is used in an algorithm to estimate the orientation of the parts.
6. A robot with a gripper connected to the end-effector uses the information about the positions and orientations of the parts to assemble them.

8.3 Recommendations for Further Work

For future work we would recommend to have one separate computer for the computer vision part and one for robot control. There were problems where the robot would stop in the middle of a trajectory. The computer did computations for both computer vision and the robots simultaneously, and the computer could not keep up with the updated information from the controllers. This lead the program to interrupt the trajectory since it did not get the vital information about trajectory goals from the controllers in time.

We also recommend to develop a method for calibrating the transformation between the IR camera and the RGB camera in the Asus Xtion PRO LIVE. This could improve the estimations of the coordinates of the parts.

Using several RGB-D cameras with different view points and a more comprehensive calibration technique for the RGB-D camera could also improve the result. Using several RGB-D cameras will generate a more dense point cloud and get a point cloud of the entire parts as opposed to one side of the parts, as done in this project.

For the orientation part, we would recommend to store a set of training images of different locations of the parts. Depending on what coordinates the RGB-D camera estimates for the parts, the program can then choose the training images most appropriate for that location. We would also recommend to use the 2D camera to find the final coordinates of the parts. That is, use RGB-D camera to find approximately coordinates, and then run a robot with 2D camera perpendicular to the parts and use the homography matrix to determine more accurate coordinates.

The computational time for the different programs used in this project could most likely be reduced if the program code was optimized.

Appendix A

Source Code

The C++ programs explained in the Chapter ”ROS” are in the appendix. They are also in the digital appendix. There are 8 programs: three for estimating the coordinates of the parts, two for estimating the orientations and two for the kinematics.

```

//———————————————————————————————————————————————————————
//———Node run for publishing transformation from camera to table ——————————
//———————————————————————————————————————————————————————
//———————————————————————————————————————————————————————
//——————————————Authors: Geir Ole Tysse and Martin Morland .—————————
//——————————————————————NTNU 2015————————————————————————
//———————————————————————————————————————————————————————

#include "ros/ros.h"
#include <vector>
#include <geometry_msgs/Pose.h>
#include <geometry_msgs/Vector3.h>
#include <std_msgs/ColorRGBA.h>
#include <geometry_msgs/Point.h>
#include <std_msgs/ColorRGBA.h>
#include "visualization_msgs/Marker.h"
#include <iostream>
#include <complex>
#include "/usr/include/math.h"
#include <fstream>
#include <sstream>
#include "std_msgs/String.h"
#include "std_msgs/Float64MultiArray.h"
#include <stdio.h>
#include <stdlib.h>
#include <Eigen/Eigen>
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/Int32MultiArray.h"
#include <armadillo>

using namespace std;

string argument_old_tags ("table_old_tags");
string argument_new_tags ("table_new_tags");
int use_old_or_new_tags=0;
int ready_for_publish=0;
int number_of_extracted_coordinates_from_id0;
int number_of_extracted_coordinates_from_id2;
int number_of_extracted_coordinates_from_id3;
std_msgs::Float64MultiArray Transformation_in_array_list;
int size=50;
std::vector<float> vec_0;
std::vector<float> vec_3;
std::vector<float> vec_3_0;
 std::vector<float> vec_2_0;
float id0_x_median;
float id0_y_median;
float id0_z_median;
float id2_x_median;
float id2_y_median;
float id2_z_median;
float id3_x_median;
float id3_y_median;
float id3_z_median;
float id0_x_mode;
float id0_y_mode;
float id0_z_mode;
float id2_x_mode;
float id2_y_mode;
float id2_z_mode;
float id3_x_mode;
float id3_y_mode;
float id3_z_mode;
float id0_x[50];
float id0_y[50];
float id0_z[50];
float id2_x[50];
float id2_y[50];
float id2_z[50];
float id3_x[50];
float id3_y[50];
float id3_z[50];
  int id_nr;
   geometry_msgs::Pose AR_tags;
 arma::mat Transformation_camera_AR_tags(4,4);
arma::mat Transformation_AR_tags_table(4,4) ;
arma::mat Transformation_camera_table(4,4) ;
 arma::fcolvec translation;
  arma::fcolvec X_axis_median;
 arma::fcolvec Y_axis_median;
 arma::fcolvec Z_axis_median;
 arma::fcolvec vector_along_x_axis_median;
 arma::fcolvec vector_along_y_axis_median;
 arma::fcolvec vector_along_x_axis_mode;
 arma::fcolvec vector_along_y_axis_mode;
   arma::fcolvec X_axis_mode;
 arma::fcolvec Y_axis_mode;
 arma::fcolvec Z_axis_mode;
  arma::fcolvec vector_along_x_axis;
 arma::fcolvec vector_along_y_axis;
   arma::fcolvec X_axis;
 arma::fcolvec Y_axis;
 arma::fcolvec Z_axis;


//-- Mode estimation
float GetMode(float daArray[], int iSize) {
    int* ipRepetition = new int[iSize];
    for (int i = 0; i < iSize; ++i) {
        ipRepetition[i] = 0;
        int j = 0;
        bool bFound = false;
        while ((j < i) && (daArray[i] != daArray[j])) {
            if (daArray[i] != daArray[j]) {
                ++j;
            }
        }
        ++(ipRepetition[j]);
    }
    int iMaxRepeat = 0;
    for (int i = 1; i < iSize; ++i) {
        if (ipRepetition[i] > ipRepetition[iMaxRepeat]) {
            iMaxRepeat = i;
        }
    }
    delete [] ipRepetition;
    return daArray[iMaxRepeat];
}


```

```

// -- Median estimation
float GetMedian(float daArray[], int iSize) {
    float* dpSorted = new float[iSize];
    for (int i = 0; i < iSize; ++i) {
        dpSorted[i] = daArray[i];
    }
    for (int i = iSize - 1; i > 0; --i) {
        for (int j = 0; j < i; ++j) {
            if (dpSorted[j] > dpSorted[j+1]) {
                float dTemp = dpSorted[j];
                dpSorted[j] = dpSorted[j+1];
                dpSorted[j+1] = dTemp;
            }
        }
    }
    float dMedian = 0.0;
    if ((iSize % 2) == 0) {
        dMedian = (dpSorted[iSize/2] + dpSorted[(iSize/2) - 1])/2.0;
    } else {
        dMedian = dpSorted[iSize/2];
    }
    delete [] dpSorted;
    return dMedian;
}




// -- The class
//------------------------------------
class SubscribeAndPublish
{
public:
  SubscribeAndPublish()
  {

    //--- Subscribe points from the AR-tags and publish transformation from camera to ↩
        table
   pub = n_.advertise<std_msgs::Float64MultiArray>("Transformation_from_camera_to_table↩
      ", 1000);
   sub = n_.subscribe("visualization_marker", 1, &SubscribeAndPublish::callback, this);

  }

  void callback(const visualization_msgs::Marker::ConstPtr& msg)
  {

// -- Points from the AR-tags
  AR_tags= msg->pose;
  id_nr= msg->id;


  //-- Extract the coordinates of ID0 and save the 50 last coordinates
  if(id_nr==0 ){
  if(!((AR_tags.position.x != AR_tags.position.x) || (AR_tags.position.y != AR_tags.↩
      position.y) || (AR_tags.position.z != AR_tags.position.z)))
  {
    number_of_extracted_coordinates_from_id0++;
     for(int x = size-1;x>=0;x--){
    id0_x[x+1]=id0_x[x];
    id0_y[x+1]=id0_y[x];
    id0_z[x+1]=id0_z[x];
      if(x==0){
    id0_x[x]=AR_tags.position.x;
    id0_y[x]=AR_tags.position.y;
    id0_z[x]=AR_tags.position.z;
     }
     }

//-- Mode and Median of 50 last coordinates
if(number_of_extracted_coordinates_from_id0 >49)
{
//-- Median of the points
id0_x_median=GetMedian(id0_x,50);
id0_y_median=GetMedian(id0_y,50);
id0_z_median=GetMedian(id0_z,50);

//-- Mode of the points
id0_x_mode=GetMode(id0_x,50);
id0_y_mode=GetMode(id0_y,50);
id0_z_mode=GetMode(id0_z,50);
  }
  }
  }



  //-- Extract the coordinates of ID2 and save the 50 last coordinates
  if(id_nr==2){
    if(!((AR_tags.position.x != AR_tags.position.x) || (AR_tags.position.y != AR_tags.↩
        position.y) || (AR_tags.position.z != AR_tags.position.z)))
  {
    number_of_extracted_coordinates_from_id2++;
     for(int x = size-1;x>=0;x--){
    id2_x[x+1]=id2_x[x];
    id2_y[x+1]=id2_y[x];
    id2_z[x+1]=id2_z[x];
      if(x==0){
    id2_x[x]=AR_tags.position.x;
    id2_y[x]=AR_tags.position.y;
    id2_z[x]=AR_tags.position.z;
  }
     }

//-- Mode and Median of 50 last coordinates
if(number_of_extracted_coordinates_from_id2 >49)
{
//-- Median of the points
id2_x_median=GetMedian(id2_x,50);
id2_y_median=GetMedian(id2_y,50);
id2_z_median=GetMedian(id2_z,50);

//-- Mode of the points
id2_x_mode=GetMode(id2_x,50);
id2_y_mode=GetMode(id2_y,50);
id2_z_mode=GetMode(id2_z,50);
}
}
}


```

```

  //-- Extract the coordinates of ID3 and save the 50 last coordinates
  if(id_nr==3){
    if(!((AR_tags.position.x != AR_tags.position.x) || (AR_tags.position.y != AR_tags.↩
        position.y) || (AR_tags.position.z != AR_tags.position.z)))
  {
    number_of_extracted_coordinates_from_id3++;
    for(int x = size-1;x>=0;x--){
    id3_x[x+1]=id3_x[x];
    id3_y[x+1]=id3_y[x];
    id3_z[x+1]=id3_z[x];
      if(x==0){
    id3_x[x]=AR_tags.position.x;
    id3_y[x]=AR_tags.position.y;
    id3_z[x]=AR_tags.position.z;
  }
     }

 //-- Mode and Median of 50 last coordinates
if(number_of_extracted_coordinates_from_id3>49)
{
//-- Median of the points
id3_x_median=GetMedian(id3_x,50);
id3_y_median=GetMedian(id3_y,50);
id3_z_median=GetMedian(id3_z,50);

//-- Mode of the points
id3_x_mode=GetMode(id3_x,50);
id3_y_mode=GetMode(id3_y,50);
id3_z_mode=GetMode(id3_z,50);
}
  }
  }



  if(number_of_extracted_coordinates_from_id3>49 && ↩
      number_of_extracted_coordinates_from_id2>49 && ↩
      number_of_extracted_coordinates_from_id0>49)
    {
       //-- Translation
    translation << id3_x_median << id3_y_median <<id3_z_median;


    vector_along_y_axis_median << id0_x_median-id3_x_median << id0_y_median-↩
        id3_y_median <<id0_z_median-id3_z_median;
    vector_along_x_axis_median << id2_x_median-id3_x_median << id2_y_median-↩
        id3_y_median << id2_z_median-id3_z_median;
    vector_along_y_axis_mode << id0_x_mode-id3_x_mode << id0_y_mode-id3_y_mode <<↩
        id0_z_mode-id3_z_mode;
    vector_along_x_axis_mode << id2_x_mode-id3_x_mode << id2_y_mode-id3_y_mode << ↩
        id2_z_mode-id3_z_mode;
    vector_along_y_axis << id0_x[49]-id3_x[49] << id0_y[49]-id3_y[49] <<id0_z[49]-id3_z↩
        [49];
    vector_along_x_axis<< id2_x[49]-id3_x[49] << id2_y[49]-id3_y[49] << id2_z[49]-id3_z↩
        [49];


    //-- The axes estimated with median of points
    Y_axis_median = arma::normalise(vector_along_y_axis_median);
    X_axis_median = arma::normalise(vector_along_x_axis_median);
    Z_axis_median = arma::cross(X_axis_median,Y_axis_median);

    //-- The axes estimated with mode of points
    Y_axis_mode = arma::normalise(vector_along_y_axis_mode);
    X_axis_mode = arma::normalise(vector_along_x_axis_mode);
    Z_axis_mode = arma::cross(X_axis_mode,Y_axis_mode);

    //-- The axes estimated with live points
    Y_axis = arma::normalise(vector_along_y_axis);
    X_axis = arma::normalise(vector_along_x_axis);
    Z_axis = arma::cross(X_axis,Y_axis);



//-- Rotation matrix from median of points
 Eigen::Matrix3f Rotation_matrix_median;
Rotation_matrix_median << X_axis_median[0], Y_axis_median[0], Z_axis_median[0],
   X_axis_median[1] ,Y_axis_median[1] ,Z_axis_median[1],
  X_axis_median[2],Y_axis_median[2], Z_axis_median[2];

//-- Rotation matrix from mode of points
Eigen::Matrix3f Rotation_matrix_mode;
Rotation_matrix_mode << X_axis_mode[0], Y_axis_mode[0], Z_axis_mode[0],
X_axis_mode[1] ,Y_axis_mode[1] ,Z_axis_mode[1],
X_axis_mode[2],Y_axis_mode[2], Z_axis_mode[2];

//-- Rotation matrix from live points
Eigen::Matrix3f Rotation_matrix_live;
Rotation_matrix_live << X_axis[0], Y_axis[0], Z_axis[0],
X_axis_mode[1] ,Y_axis[1] ,Z_axis[1],
X_axis[2],Y_axis[2], Z_axis[2];




//-- Determine which of the rotation matrices that are the final rotation matrix
if((abs(Rotation_matrix_median.determinant()-1) < 0.0005 || abs(Rotation_matrix_mode.↩
    determinant()-1) < 0.0005)&&ready_for_publish==0)
{
  ready_for_publish=1;
  if(abs(Rotation_matrix_median.determinant()-1)<abs(Rotation_matrix_mode.determinant()↩
      -1))
  {
    //-- Transformation from camera to AR-tags
Transformation_camera_AR_tags << X_axis_median[0] << Y_axis_median[0] << Z_axis_median↩
    [0] << translation[0] << arma::endr
  << X_axis_median[1] << Y_axis_median[1] << Z_axis_median[1] << translation[1] << arma↩
      ::endr
  << X_axis_median[2] << Y_axis_median[2] << Z_axis_median[2] << translation[2] << arma↩
      ::endr
<< 0 << 0 << 0 << 1 << arma::endr ;
  }
  if (abs(Rotation_matrix_median.determinant()-1)>abs(Rotation_matrix_mode.determinant↩
      ()-1))
  {
     //-- Transformation from camera to AR-tags
    Transformation_camera_AR_tags << X_axis_mode[0] << Y_axis_mode[0] << Z_axis_mode[0]↩
         << translation[0] << arma::endr
  << X_axis_mode[1] << Y_axis_mode[1] << Z_axis_mode[1] << translation[1] << arma::endr
  << X_axis_mode[2] << Y_axis_mode[2] << Z_axis_mode[2] << translation[2] << arma::endr
<< 0 << 0 << 0 << 1 << arma::endr ;
```

```

  }


 //−− Transformation from AR−tags to table (old tags)
if(use_old_or_new_tags==1)
{

Transformation_AR_tags_table << 1<< 0 << 0 << −0.1905 << arma::endr
  << 0 << 1 << 0<< −0.104<< arma::endr
  << 0 << 0 << 1<< 0 << arma::endr
<< 0 << 0 << 0 << 1 << arma::endr ;

Transformation_camera_table=Transformation_camera_AR_tags∗Transformation_AR_tags_table;

  Transformation_in_array_list.data.clear();
for (int i = 0; i < 16; i++)
{
//assign array a random number between 0 and 255.
Transformation_in_array_list.data.push_back(Transformation_camera_table[i]);
}

  }

 //−− Transformation from AR−tags to table (new tags)
  if(use_old_or_new_tags==2)
{
Transformation_AR_tags_table << 1<< 0 << 0 << −0.1905 << arma::endr
  << 0 << 1 << 0<< −0.604<< arma::endr
  << 0 << 0 << 1<< 0 << arma::endr
<< 0 << 0 << 0 << 1 << arma::endr ;

Transformation_camera_table=Transformation_camera_AR_tags∗Transformation_AR_tags_table;

  Transformation_in_array_list.data.clear();
for (int i = 0; i < 16; i++)
{
//−− Transformation ready for publishing
Transformation_in_array_list.data.push_back(Transformation_camera_table[i]);
}
  }
else{
   //−− Transformation from AR−tags to table (old tags)

  Transformation_in_array_list.data.clear();
for (int i = 0; i < 16; i++)
{
//−− Transformation ready for publishing
Transformation_in_array_list.data.push_back(Transformation_camera_AR_tags[i]);
}
}
    }
    }



//−− Publish transformation from camera to table
 if(ready_for_publish==1)
 {
 pub.publish(Transformation_in_array_list);
     std::cout << "Transformation published!" << endl;

 }
  }



private:
  ros::NodeHandle n_;
  ros::Publisher pub;
  ros::Subscriber sub;

};

int main(int argc, char **argv)
{
if(argc !=1)
  {

    if(argv[1]==argument_old_tags)
    {
      use_old_or_new_tags=1;
      cout << "Align coordinatesystem to the table based on old AR-tags" << endl;
    }

    if(argv[1]==argument_new_tags)
    {
      use_old_or_new_tags=2;
      cout << "Align coordinatesystem to the table based on new AR-tags" << endl;
    }
   }

  //Initiate ROS.
  ros::init(argc, argv, "Node_Transformation_from_camera_to_table");
  //Create an object of class SubscribeAndPublish that will take care of everything
  SubscribeAndPublish SAPObject;
  ros::spin();

  return 0;
}
```

```
//-------------------------------------------------------------------------
//-----Node run for Segmentation based on known geometry, one point clouds-----
//-------------------------------------------------------------------------
//--------------------------------------------------------------------------
//---------------------Authors: Geir Ole Tysse and Martin Morland .---------------
//----------------------------------NTNU 2015----------------------------------
//--------------------------------------------------------------------------

#include <ros/ros.h>
#include <pcl_ros/point_cloud.h>
#include <ros/ros.h>
#include <iostream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <boost/foreach.hpp>
#include <sensor_msgs/PointCloud2.h>
#include <limits>
#include <fstream>
#include <vector>
#include <Eigen/Core>
#include <pcl/point_cloud.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/features/normal_3d.h>
#include <pcl/features/fpfh.h>
#include <pcl/registration/ia_ransac.h>
#include <armadillo>
#include <sstream>
#include "std_msgs/String.h"
#include "std_msgs/Float64MultiArray.h"
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/Int32MultiArray.h"
#include <armadillo>
#include <pcl/filters/filter.h>
#include <pcl/filters/statistical_outlier_removal.h>

using namespace std;
using namespace arma;

float list_with_cylinder_coordinates_y[30];
float list_with_cylinder_coordinates_x[30];
float list_with_piston_coordinates_y[30];
float list_with_piston_coordinates_x[30];
int number_of_coordinates_estimated=0;
int Transformation_camera_table_ready=0;
   mat Transformation_table_camera(4,4);
   mat Transformation_camera_table(4,4);
  rowvec r;
  colvec temporary_point_with_respect_to_table;
  colvec temporary_point;
  std_msgs::Float64MultiArray publishing_coordinates_cylinder;
  std_msgs::Float64MultiArray publishing_coordinates_piston;


// --- Get the transformation from camera to table
    void callbacknh(const std_msgs::Float64MultiArray::ConstPtr& msg)
   {
     Transformation_camera_table << msg->data[0] << msg->data[4] << msg->data[8] << msg->data[12] << arma::endr
     << msg->data[1] << msg->data[5] << msg->data[9] << msg->data[13] << arma::endr
     << msg->data[2] << msg->data[6] << msg->data[10] << msg->data[14] << arma::endr
     << msg->data[3] << msg->data[7] << msg->data[11] << msg->data[15] << arma::endr;
    Transformation_camera_table_ready =1;

   }

//-- Remove NAN-points
void filterOutNans(const pcl::PointCloud<pcl::PointXYZ>& _inCloud, pcl::PointCloud<pcl::PointXYZ>& _outCloud) {
    // Filter out NaNs from data
    pcl::PassThrough<pcl::PointXYZ> NANRemover;
    NANRemover.setInputCloud(_inCloud.makeShared());
    NANRemover.setFilterFieldName("z");
    NANRemover.setFilterLimits(-0.01, 10.0); /// Get valid range in kinect point cloud data
    NANRemover.filter(_outCloud);
}



//-- Estimate the median
float GetMedian(float daArray[], int iSize) {
    double* dpSorted = new double[iSize];
    for (int i = 0; i < iSize; ++i) {
        dpSorted[i] = daArray[i];
    }
    for (int i = iSize - 1; i > 0; --i) {
        for (int j = 0; j < i; ++j) {
            if (dpSorted[j] > dpSorted[j+1]) {
                double dTemp = dpSorted[j];
                dpSorted[j] = dpSorted[j+1];
                dpSorted[j+1] = dTemp;
            }
        }
    }
    double dMedian = 0.0;
    if ((iSize % 2) == 0) {
        dMedian = (dpSorted[iSize/2] + dpSorted[(iSize/2) - 1])/2.0;
    } else {
        dMedian = dpSorted[iSize/2];
    }
    delete [] dpSorted;
    return dMedian;
}


typedef pcl::PointCloud<pcl::PointXYZ> PointCloud;
pcl::PointCloud<pcl::PointXYZ>  cloud;
   pcl::PointCloud<pcl::PointXYZ> cloud_table;
pcl::PointCloud<pcl::PointXYZ> cloud_cylinder;
pcl::PointCloud<pcl::PointXYZ> cloud_piston;
pcl::PointCloud<pcl::PointXYZ> cloud_from_camera;


//-- The class
//---------------
class SubscribeAndPublish
```

```
{
public:
  SubscribeAndPublish()
  {
   publisher_cylinder = n_.advertise<std_msgs::Float64MultiArray>("cylinder_coordinates↩
       ", 1);
   publisher_piston = n_.advertise<std_msgs::Float64MultiArray>("piston_coordinates", ↩
       1);
    subscriber_point_cloud = n_.subscribe<PointCloud>("/camera/depth_registered/points"↩
        , 1, &SubscribeAndPublish::callback, this);

  }


  void callback(const PointCloud::ConstPtr& msg)
{

  //-- Start to run code when transformation from camera to table has been subscribed
      if(Transformation_camera_table_ready > 0)
   {

//-- Point cloud from camera
      cloud_from_camera  =*msg;

//-- Filters out nan-points
    filterOutNans(cloud_from_camera, cloud);

//-- Statistical outliers removal
  pcl::StatisticalOutlierRemoval<pcl::PointXYZ> statistical_filter;
  statistical_filter.setInputCloud(boost::make_shared<pcl::PointCloud<pcl::PointXYZ> >(↩
      cloud));
  statistical_filter.setMeanK(80);
  statistical_filter.setStddevMulThresh(2.9);
  statistical_filter.filter(cloud);

//-- Transform point cloud to points given with respect to table coordinates
        temporary_point << 1 << 1 << 1 << 1;
        Transformation_table_camera=inv(Transformation_camera_table);
        float cylinder_point_x_coordinate=0;
        float cylinder_point_y_coordinate=0;
        int cloud_table_size=0;
      for (size_t i = 0; i < cloud.points.size (); ++i)
  {
    //-- Transform every point in point cloud to be given with (X,Y,Z) coordinates with↩
        respect to table
   temporary_point << cloud.points[i].x << cloud.points[i].y << cloud.points[i].z << 1;
        temporary_point_with_respect_to_table = Transformation_table_camera* ↩
            temporary_point;
         r = temporary_point_with_respect_to_table.t();
    cloud.points[i].x = r(0);
    cloud.points[i].y = r(1);
    cloud.points[i].z = r(2);

    //-- Count number of points on table
     if((cloud.points[i].z > -0.1)&&(cloud.points[i].x > 0)&&(cloud.points[i].x < 0.7)↩
         &&(cloud.points[i].y > -0.1)&&(cloud.points[i].y < 0.7))
    {
      cloud_table_size++;
    }
  }


  //-- Point cloud on table
   cloud_table.width = cloud_table_size;
    cloud_table.height = cloud.height;
    cloud_table.is_dense = cloud.is_dense;
  cloud_table.points.resize (cloud_table.width * cloud_table.height);
  int index_for_points_in_cloud_table=0;

       for (size_t i = 0; i < cloud.points.size (); ++i)
  {
     if((cloud.points[i].z > -0.1)&&(cloud.points[i].x > 0)&&(cloud.points[i].x < 0.7)↩
         &&(cloud.points[i].y > -0.1)&&(cloud.points[i].y < 0.7))
    {
      // -- A cloud consisting of points on table
     cloud_table.points[i].x = cloud.points[i].x;
     cloud_table.points[i].y = cloud.points[i].y;
     cloud_table.points[i].z = cloud.points[i].z;
    index_for_points_in_cloud_table++;

    //-- Extract a point on the cylinder
    if((cloud.points[i].z > 0.15)&&(cloud.points[i].z < 0.22))
    {
     cylinder_point_x_coordinate= cloud.points[i].x;
     cylinder_point_y_coordinate = cloud.points[i].y;
    }
    }
  }



      //-- Extract a point on the piston
         float piston_point_x_coordinate=0;
     float piston_point_y_coordinate=0;

    for (size_t i = 0; i < cloud_table.points.size (); ++i)
  {
     if(((abs(cloud_table.points[i].x-cylinder_point_x_coordinate)>0.12) || (abs(↩
         cloud_table.points[i].y-cylinder_point_y_coordinate)>0.12))&&(cloud_table.↩
         points[i].z>0.03)&&(cloud_table.points[i].z<0.15))
    {
    piston_point_x_coordinate = cloud_table.points[i].x;
    piston_point_y_coordinate = cloud_table.points[i].y;
    }
  }




 //-- Count number of points on the cylinder and the piston
  int number_of_points_cylinder=0;
  int number_of_points_piston=0;
    for (size_t i = 0; i < cloud_table.points.size (); ++i)
  {
    if( (abs(cloud_table.points[i].x-cylinder_point_x_coordinate)<0.07) && (abs(↩
        cloud_table.points[i].y-cylinder_point_y_coordinate)<0.07)&&(cloud_table.points↩
        [i].z>0.03)&&(cloud_table.points[i].z<0.14))
    {
      number_of_points_cylinder++;
    }
  if( (abs(cloud_table.points[i].x-piston_point_x_coordinate)<0.06) && (abs(cloud_table↩
      .points[i].y-piston_point_y_coordinate)<0.06)&&(cloud_table.points[i].z>0.03)&&(↩
      cloud_table.points[i].z<0.08))
    {
      number_of_points_piston++;
    }
  }


//-- Define a cloud consisting of the cylinder and a cloud consisting of the piston
      cloud_cylinder.width = number_of_points_cylinder;
    cloud_cylinder.height = cloud_table.height;
    cloud_cylinder.is_dense = cloud_table.is_dense;
  cloud_cylinder.points.resize (cloud_cylinder.width * cloud_cylinder.height);

   cloud_piston.width = number_of_points_piston;
    cloud_piston.height = cloud_table.height;
    cloud_piston.is_dense = cloud_table.is_dense;
  cloud_piston.points.resize (cloud_piston.width * cloud_piston.height);

int index_for_points_in_cloud_cylinder=0;
int index_for_points_in_cloud_piston=0;

//--- A cloud consisting of the cylinder and a cloud consisting of the piston
    for (size_t i = 0; i < cloud_table.points.size (); ++i)
  {

    if( (abs(cloud_table.points[i].x-cylinder_point_x_coordinate)<0.07) && (abs(↩
        cloud_table.points[i].y-cylinder_point_y_coordinate)<0.07)&&(cloud_table.points↩
        [i].z>0.03)&&(cloud_table.points[i].z<0.14))
    {
    cloud_cylinder.points[index_for_points_in_cloud_cylinder].x = cloud_table.points[i↩
        ].x;
    cloud_cylinder.points[index_for_points_in_cloud_cylinder].y = cloud_table.points[i↩
        ].y ;
    cloud_cylinder.points[index_for_points_in_cloud_cylinder].z = cloud_table.points[i↩
        ].z;
      index_for_points_in_cloud_cylinder++;
    }
    else
    {
    if( (abs(cloud_table.points[i].x-piston_point_x_coordinate)<0.06) && (abs(↩
        cloud_table.points[i].y-piston_point_y_coordinate)<0.06)&&(cloud_table.points[i↩
        ].z>0.03)&&(cloud_table.points[i].z<0.08))
    {
    cloud_piston.points[index_for_points_in_cloud_piston].x = cloud_table.points[i].x;
    cloud_piston.points[index_for_points_in_cloud_piston].y = cloud_table.points[i].y ;
    cloud_piston.points[index_for_points_in_cloud_piston].z = cloud_table.points[i].z;
      index_for_points_in_cloud_piston++;
    }
    }
  }


 //-- Save pcd-files of the point clouds
   pcl::io::savePCDFileASCII ("/home/minions/pcl/build/pcd/cylinder.pcd", cloud_piston)↩
       ;
   pcl::io::savePCDFileASCII ("/home/minions/pcl/build/pcd/piston.pcd", cloud_cylinder)↩
       ;
   pcl::io::savePCDFileASCII ("/home/minions/pcl/build/pcd/table.pcd", cloud_table);


 //-- List the coordinates of the points in point clouds of the piston and the cylinder
```

```
float array_cylinder_x_coordinates[index_for_points_in_cloud_cylinder];
float array_cylinder_y_coordinates[index_for_points_in_cloud_cylinder];

for(size_t i =0;i<index_for_points_in_cloud_cylinder;i++)
{
array_cylinder_x_coordinates[i]= cloud_cylinder.points[i].x ;
array_cylinder_y_coordinates[i]= cloud_cylinder.points[i].y ;
}

float array_piston_x_coordinates[index_for_points_in_cloud_piston];
float array_piston_y_coordinates[index_for_points_in_cloud_piston];

for(size_t i =0;i<index_for_points_in_cloud_cylinder;i++)
{
array_piston_x_coordinates[i]= cloud_piston.points[i].x ;
array_piston_y_coordinates[i]= cloud_piston.points[i].y ;
}




//-- Coordinates of the piston and the cylinder
float piston_coordinate_x = GetMedian(array_piston_x_coordinates,↩
    index_for_points_in_cloud_piston)-0.011023;
float piston_coordinate_y = GetMedian(array_piston_y_coordinates,↩
    index_for_points_in_cloud_piston)-0.021165;
float cylinder_coordinate_x = GetMedian(array_cylinder_x_coordinates,↩
    index_for_points_in_cloud_cylinder)-0.009983;
float cylinder_coordinate_y = GetMedian(array_cylinder_y_coordinates,↩
    index_for_points_in_cloud_cylinder)-0.026002;


//-- Save the last 20 coordinates for both parts
 for(int x = 20-1;x>=0;x--){
list_with_cylinder_coordinates_x[x+1]=list_with_cylinder_coordinates_x[x];
list_with_cylinder_coordinates_y[x+1]=list_with_cylinder_coordinates_y[x];
list_with_piston_coordinates_x[x+1]=list_with_piston_coordinates_x[x];
list_with_piston_coordinates_y[x+1]=list_with_piston_coordinates_y[x];
      if(x==0){
list_with_cylinder_coordinates_x[x]=cylinder_coordinate_x;
list_with_cylinder_coordinates_y[x]=cylinder_coordinate_y;
list_with_piston_coordinates_x[x]=piston_coordinate_x;
list_with_piston_coordinates_y[x]=piston_coordinate_y;
     }
}

number_of_coordinates_estimated++;

if(number_of_coordinates_estimated>20)
{
//-- The estimated coordinates for the cylinder and the piston
float final_cylinder_coordinate_x=GetMedian(list_with_cylinder_coordinates_x,20);
float final_cylinder_coordinate_y=GetMedian(list_with_cylinder_coordinates_y,20);
float final_piston_coordinate_x=GetMedian(list_with_piston_coordinates_x,20);
float final_piston_coordinate_y=GetMedian(list_with_piston_coordinates_y,20);

  //-- Publish cylinder coordinates
publishing_coordinates_cylinder.data.clear();
     for (int i = 0; i < 3; i++)
{
  if(i==0)
  {
publishing_coordinates_cylinder.data.push_back(final_cylinder_coordinate_x);
  }
  if(i==1)
  {
    publishing_coordinates_cylinder.data.push_back(final_cylinder_coordinate_y);
  }
  if(i==2)
  {
  publishing_coordinates_cylinder.data.push_back(0.17);
  }
}

 publisher_cylinder.publish(publishing_coordinates_cylinder);


  //-- Publish piston coordinates
 publishing_coordinates_piston.data.clear();
        for (int i = 0; i < 3; i++)
{
  if(i==0)
  {
publishing_coordinates_piston.data.push_back(final_piston_coordinate_x);
  }
  if(i==1)
  {
    publishing_coordinates_piston.data.push_back(final_piston_coordinate_y);
  }
  if(i==2)
  {
  publishing_coordinates_piston.data.push_back(0.10);
  }
}
 publisher_piston.publish(publishing_coordinates_piston);

}
  }
}


private:
  ros::NodeHandle n_;
  ros::Publisher publisher_cylinder;
  ros::Publisher publisher_piston;
  ros::Subscriber subscriber_point_cloud;


};

int main(int argc, char **argv)
{

  //Initiate ROS. The node name is "Pointcloud"
  ros::init(argc, argv, "Pointcloud");
  ros::NodeHandle nh;

    //-- Subscribe Transformation from camera to table
ros::Subscriber subnh = nh.subscribe<std_msgs::Float64MultiArray>("↩
    Transformation_from_camera_to_table", 1, callbacknh);

//Create an object of class SubscribeAndPublish that will take care of everything
```

```
  SubscribeAndPublish SAPObject;
  ros::spin();

  return 0;
}
```

```
//------------------------------------------------------------------------
//-----Node run for Segmentation based on known geometry, four point clouds-----
//------------------------------------------------------------------------
//------------------------------------------------------------------------
//-------------------Authors: Geir Ole Tysse and Martin Morland .------------
//------------------------------NTNU 2015----------------------------------
//------------------------------------------------------------------------

#include <ros/ros.h>
#include <pcl_ros/point_cloud.h>
#include <ros/ros.h>
#include <iostream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <boost/foreach.hpp>
#include <sensor_msgs/PointCloud2.h>
#include <limits>
#include <fstream>
#include <vector>
#include <Eigen/Core>
#include <pcl/point_cloud.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/features/normal_3d.h>
#include <pcl/features/fpfh.h>
#include <pcl/registration/ia_ransac.h>
#include <armadillo>
#include <sstream>
#include "std_msgs/String.h"
#include "std_msgs/Float64MultiArray.h"
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/Int32MultiArray.h"
#include <armadillo>
#include <pcl/filters/filter.h>
#include <pcl/filters/statistical_outlier_removal.h>

using namespace std;
using namespace arma;

int index_for_merging_clouds=0;
float list_with_cylinder_coordinates_y[30];
float list_with_cylinder_coordinates_x[30];
float list_with_piston_coordinates_y[30];
float list_with_piston_coordinates_x[30];
int number_of_coordinates_estimated=0;
int Transformation_camera_table_ready=0;
   mat Transformation_table_camera(4,4);
   mat Transformation_camera_table(4,4);
  rowvec r;
  colvec temporary_point_with_respect_to_table;
  colvec temporary_point;
  std_msgs::Float64MultiArray publishing_coordinates_cylinder;
  std_msgs::Float64MultiArray publishing_coordinates_piston;


// --- Get the transformation from camera to table
```

```
    void callbacknh(const std_msgs::Float64MultiArray::ConstPtr& msg)
   {
     Transformation_camera_table << msg->data[0] << msg->data[4] << msg->data[8] << msg->data[12] << arma::endr
     << msg->data[1] << msg->data[5] << msg->data[9] << msg->data[13] << arma::endr
     << msg->data[2] << msg->data[6] << msg->data[10] << msg->data[14] << arma::endr
     << msg->data[3] << msg->data[7] << msg->data[11] << msg->data[15] << arma::endr;
    Transformation_camera_table_ready =1;

   }

//-- Remove NAN-points
void filterOutNans(const pcl::PointCloud<pcl::PointXYZ>& _inCloud, pcl::PointCloud<pcl::PointXYZ>& _outCloud) {
    pcl::PassThrough<pcl::PointXYZ> NANRemover;
    NANRemover.setInputCloud(_inCloud.makeShared());
    NANRemover.setFilterFieldName("z");
    NANRemover.setFilterLimits(-0.01, 10.0);
    NANRemover.filter(_outCloud);
}



//-- Estimate the median
float GetMedian(float daArray[], int iSize) {
    double* dpSorted = new double[iSize];
    for (int i = 0; i < iSize; ++i) {
        dpSorted[i] = daArray[i];
    }
    for (int i = iSize - 1; i > 0; --i) {
        for (int j = 0; j < i; ++j) {
            if (dpSorted[j] > dpSorted[j+1]) {
                double dTemp = dpSorted[j];
                dpSorted[j] = dpSorted[j+1];
                dpSorted[j+1] = dTemp;
            }
        }
    }
    double dMedian = 0.0;
    if ((iSize % 2) == 0) {
        dMedian = (dpSorted[iSize/2] + dpSorted[(iSize/2) - 1])/2.0;
    } else {
        dMedian = dpSorted[iSize/2];
    }
    delete [] dpSorted;
    return dMedian;
}


typedef pcl::PointCloud<pcl::PointXYZ> PointCloud;
pcl::PointCloud<pcl::PointXYZ>  cloud;
pcl::PointCloud<pcl::PointXYZ>  first_cloud;
pcl::PointCloud<pcl::PointXYZ>  second_cloud;
pcl::PointCloud<pcl::PointXYZ>  third_cloud;
pcl::PointCloud<pcl::PointXYZ>  fourth_cloud;
   pcl::PointCloud<pcl::PointXYZ> cloud_table;
pcl::PointCloud<pcl::PointXYZ> cloud_cylinder;
pcl::PointCloud<pcl::PointXYZ> cloud_piston;


//-- The class
```

```
//-------------
class SubscribeAndPublish
{
public:
  SubscribeAndPublish()
  {
   publisher_cylinder = n_.advertise<std_msgs::Float64MultiArray>("cylinder_coordinates↩
       ", 1);
   publisher_piston = n_.advertise<std_msgs::Float64MultiArray>("piston_coordinates", ↩
       1);
    subscriber_point_cloud = n_.subscribe<PointCloud>("/camera/depth_registered/points"↩
        , 1, &SubscribeAndPublish::callback, this);

  }


  void callback(const PointCloud::ConstPtr& msg)
{

  //-- Start to run code when transformation from camera to table has been subscribed
      if(Transformation_camera_table_ready > 0)
   {

 unsigned int microseconds=500000;


 //-- Save four point clouds
   if(index_for_merging_clouds==0)
     {

     first_cloud  =*msg;
     usleep(microseconds);
     }

      if(index_for_merging_clouds==1)
     {
    second_cloud=*msg;
     usleep(microseconds);
     }

       if(index_for_merging_clouds==2)
     {
      third_cloud=*msg;
     usleep(microseconds);
     }

       if(index_for_merging_clouds==3)
     {
     fourth_cloud=*msg;
     usleep(microseconds);
     }
  index_for_merging_clouds++;



  if(index_for_merging_clouds>3)
  {
    index_for_merging_clouds=0;

//-- Four merged point clouds from camera
   pcl::PointCloud<pcl::PointXYZ>  merged_four_clouds  = first_cloud;
   merged_four_clouds += second_cloud;
    merged_four_clouds += third_cloud;
    merged_four_clouds += fourth_cloud;

//-- Filters out nan-points
    filterOutNans(merged_four_clouds, cloud);

//-- Statistical outliers removal
  pcl::StatisticalOutlierRemoval<pcl::PointXYZ> statistical_filter;
  statistical_filter.setInputCloud(boost::make_shared<pcl::PointCloud<pcl::PointXYZ> >(↩
      cloud));
  statistical_filter.setMeanK(80);
  statistical_filter.setStddevMulThresh(2.9);
  statistical_filter.filter(cloud);

//-- Transform point cloud to points given with respect to table coordinates
       temporary_point << 1 << 1 << 1 << 1;
       Transformation_table_camera=inv(Transformation_camera_table);
       float cylinder_point_x_coordinate=0;
       float cylinder_point_y_coordinate=0;
       int cloud_table_size=0;
     for (size_t i = 0; i < cloud.points.size (); ++i)
  {
    //-- Transform every point in point cloud to be given with (X,Y,Z) coordinates with↩
        respect to table
   temporary_point << cloud.points[i].x << cloud.points[i].y << cloud.points[i].z << 1;
       temporary_point_with_respect_to_table = Transformation_table_camera* ↩
           temporary_point;
        r = temporary_point_with_respect_to_table.t();
    cloud.points[i].x = r(0);
    cloud.points[i].y = r(1);
    cloud.points[i].z = r(2);

    //-- Count number of points on table
     if((cloud.points[i].z > -0.1)&&(cloud.points[i].x > 0)&&(cloud.points[i].x < 0.7)↩
         &&(cloud.points[i].y > -0.1)&&(cloud.points[i].y < 0.7))
    {
      cloud_table_size++;
    }
  }


  //-- Point cloud on table
   cloud_table.width = cloud_table_size;
    cloud_table.height = cloud.height;
    cloud_table.is_dense = cloud.is_dense;
  cloud_table.points.resize (cloud_table.width * cloud_table.height);
  int index_for_points_in_cloud_table=0;

       for (size_t i = 0; i < cloud.points.size (); ++i)
  {
     if((cloud.points[i].z > -0.1)&&(cloud.points[i].x > 0)&&(cloud.points[i].x < 0.7)↩
         &&(cloud.points[i].y > -0.1)&&(cloud.points[i].y < 0.7))
    {
      // -- A cloud consisting of points on table
     cloud_table.points[i].x = cloud.points[i].x;
     cloud_table.points[i].y = cloud.points[i].y;
     cloud_table.points[i].z = cloud.points[i].z;
    index_for_points_in_cloud_table++;

    //-- Extract a point on the cylinder
    if((cloud.points[i].z > 0.15)&&(cloud.points[i].z < 0.22))
    {
     cylinder_point_x_coordinate= cloud.points[i].x;
     cylinder_point_y_coordinate = cloud.points[i].y;
    }
    }
  }



      //-- Extract a point on the piston
         float piston_point_x_coordinate=0;
     float piston_point_y_coordinate=0;

    for (size_t i = 0; i < cloud_table.points.size (); ++i)
  {
     if(((abs(cloud_table.points[i].x-cylinder_point_x_coordinate)>0.12) || (abs(↩
         cloud_table.points[i].y-cylinder_point_y_coordinate)>0.12))&&(cloud_table.↩
         points[i].z>0.03)&&(cloud_table.points[i].z<0.15))
    {
    piston_point_x_coordinate = cloud_table.points[i].x;
    piston_point_y_coordinate = cloud_table.points[i].y;
    }
  }




 //-- Count number of points on the cylinder and the piston
  int number_of_points_cylinder=0;
  int number_of_points_piston=0;
    for (size_t i = 0; i < cloud_table.points.size (); ++i)
  {
    if( (abs(cloud_table.points[i].x-cylinder_point_x_coordinate)<0.07) && (abs(↩
        cloud_table.points[i].y-cylinder_point_y_coordinate)<0.07)&&(cloud_table.points↩
        [i].z>0.03)&&(cloud_table.points[i].z<0.14))
    {
      number_of_points_cylinder++;
    }
  if( (abs(cloud_table.points[i].x-piston_point_x_coordinate)<0.06) && (abs(cloud_table↩
      .points[i].y-piston_point_y_coordinate)<0.06)&&(cloud_table.points[i].z>0.03)&&(↩
      cloud_table.points[i].z<0.08))
    {
      number_of_points_piston++;
    }
  }


//-- Define a cloud consisting of the cylinder and a cloud consisting of the piston
      cloud_cylinder.width = number_of_points_cylinder;
    cloud_cylinder.height = cloud_table.height;
    cloud_cylinder.is_dense = cloud_table.is_dense;
  cloud_cylinder.points.resize (cloud_cylinder.width * cloud_cylinder.height);

   cloud_piston.width = number_of_points_piston;
    cloud_piston.height = cloud_table.height;
    cloud_piston.is_dense = cloud_table.is_dense;
  cloud_piston.points.resize (cloud_piston.width * cloud_piston.height);

int index_for_points_in_cloud_cylinder=0;
int index_for_points_in_cloud_piston=0;
```

```

//--- A cloud consisting of the cylinder and a cloud consisting of the piston
    for (size_t i = 0; i < cloud_table.points.size (); ++i)
  {

    if( (abs(cloud_table.points[i].x-cylinder_point_x_coordinate)<0.07) && (abs(↩
        cloud_table.points[i].y-cylinder_point_y_coordinate)<0.07)&&(cloud_table.points↩
        [i].z>0.03)&&(cloud_table.points[i].z<0.14))
    {
    cloud_cylinder.points[index_for_points_in_cloud_cylinder].x = cloud_table.points[i↩
        ].x;
    cloud_cylinder.points[index_for_points_in_cloud_cylinder].y = cloud_table.points[i↩
        ].y ;
    cloud_cylinder.points[index_for_points_in_cloud_cylinder].z = cloud_table.points[i↩
        ].z;
      index_for_points_in_cloud_cylinder++;
    }
    else
    {
    if( (abs(cloud_table.points[i].x-piston_point_x_coordinate)<0.06) && (abs(↩
        cloud_table.points[i].y-piston_point_y_coordinate)<0.06)&&(cloud_table.points[i↩
        ].z>0.03)&&(cloud_table.points[i].z<0.08))
    {
    cloud_piston.points[index_for_points_in_cloud_piston].x = cloud_table.points[i].x;
    cloud_piston.points[index_for_points_in_cloud_piston].y = cloud_table.points[i].y ;
    cloud_piston.points[index_for_points_in_cloud_piston].z = cloud_table.points[i].z;
      index_for_points_in_cloud_piston++;
    }
    }
  }


 //-- Save pcd-files of the point clouds
   pcl::io::savePCDFileASCII ("/home/minions/pcl/build/pcd/cylinder.pcd", cloud_piston)↩
       ;
   pcl::io::savePCDFileASCII ("/home/minions/pcl/build/pcd/piston.pcd", cloud_cylinder)↩
       ;
   pcl::io::savePCDFileASCII ("/home/minions/pcl/build/pcd/table.pcd", cloud_table);


 //-- List the coordinates of the points in point clouds of the piston and the cylinder
float array_cylinder_x_coordinates[index_for_points_in_cloud_cylinder];
float array_cylinder_y_coordinates[index_for_points_in_cloud_cylinder];

for(size_t i =0;i<index_for_points_in_cloud_cylinder;i++)
{
array_cylinder_x_coordinates[i]= cloud_cylinder.points[i].x ;
array_cylinder_y_coordinates[i]= cloud_cylinder.points[i].y ;
}

float array_piston_x_coordinates[index_for_points_in_cloud_piston];
float array_piston_y_coordinates[index_for_points_in_cloud_piston];

for(size_t i =0;i<index_for_points_in_cloud_cylinder;i++)
{
array_piston_x_coordinates[i]= cloud_piston.points[i].x ;
array_piston_y_coordinates[i]= cloud_piston.points[i].y ;
}
```

```

//-- Coordinates of the piston and the cylinder
float piston_coordinate_x = GetMedian(array_piston_x_coordinates,↩
    index_for_points_in_cloud_piston)-0.011023;
float piston_coordinate_y = GetMedian(array_piston_y_coordinates,↩
    index_for_points_in_cloud_piston)-0.021165;
float cylinder_coordinate_x = GetMedian(array_cylinder_x_coordinates,↩
    index_for_points_in_cloud_cylinder)-0.009983;
float cylinder_coordinate_y = GetMedian(array_cylinder_y_coordinates,↩
    index_for_points_in_cloud_cylinder)-0.026002;


//-- Save the last 20 coordinates for both parts
 for(int x = 20-1;x>=0;x--){
list_with_cylinder_coordinates_x[x+1]=list_with_cylinder_coordinates_x[x];
list_with_cylinder_coordinates_y[x+1]=list_with_cylinder_coordinates_y[x];
list_with_piston_coordinates_x[x+1]=list_with_piston_coordinates_x[x];
list_with_piston_coordinates_y[x+1]=list_with_piston_coordinates_y[x];
      if(x==0){
list_with_cylinder_coordinates_x[x]=cylinder_coordinate_x;
list_with_cylinder_coordinates_y[x]=cylinder_coordinate_y;
list_with_piston_coordinates_x[x]=piston_coordinate_x;
list_with_piston_coordinates_y[x]=piston_coordinate_y;
     }
}

number_of_coordinates_estimated++;

if(number_of_coordinates_estimated>20)
{
//-- The estimated coordinates for the cylinder and the piston
float final_cylinder_coordinate_x=GetMedian(list_with_cylinder_coordinates_x,20);
float final_cylinder_coordinate_y=GetMedian(list_with_cylinder_coordinates_y,20);
float final_piston_coordinate_x=GetMedian(list_with_piston_coordinates_x,20);
float final_piston_coordinate_y=GetMedian(list_with_piston_coordinates_y,20);

  //-- Publish cylinder coordinates
publishing_coordinates_cylinder.data.clear();
     for (int i = 0; i < 3; i++)
{
  if(i==0)
  {
publishing_coordinates_cylinder.data.push_back(final_cylinder_coordinate_x);
  }
  if(i==1)
  {
    publishing_coordinates_cylinder.data.push_back(final_cylinder_coordinate_y);
  }
  if(i==2)
  {
  publishing_coordinates_cylinder.data.push_back(0.17);
  }
}

 publisher_cylinder.publish(publishing_coordinates_cylinder);


  //-- Publish piston coordinates
 publishing_coordinates_piston.data.clear();
        for (int i = 0; i < 3; i++)
{
  if(i==0)
  {
publishing_coordinates_piston.data.push_back(final_piston_coordinate_x);
  }
  if(i==1)
  {
    publishing_coordinates_piston.data.push_back(final_piston_coordinate_y);
  }
  if(i==2)
  {
  publishing_coordinates_piston.data.push_back(0.10);
  }
}
 publisher_piston.publish(publishing_coordinates_piston);

}
  }
}
}

private:
  ros::NodeHandle n_;
  ros::Publisher publisher_cylinder;
  ros::Publisher publisher_piston;
  ros::Subscriber subscriber_point_cloud;


};

int main(int argc, char **argv)
{

  //Initiate ROS. The node name is "Pointcloud"
  ros::init(argc, argv, "Pointcloud");
  ros::NodeHandle nh;

    //-- Subscribe Transformation from camera to table
ros::Subscriber subnh = nh.subscribe<std_msgs::Float64MultiArray>("↩
    Transformation_from_camera_to_table", 1, callbacknh);

//Create an object of class SubscribeAndPublish that will take care of everything
  SubscribeAndPublish SAPObject;
  ros::spin();

  return 0;
}
```

```
//----------------------------------------------------------------------------
//-------------------- Node run for Region Growing Segmentation --------------
//----------------------------------------------------------------------------
//----------------------------------------------------------------------------
//--------------------Authors: Geir Ole Tysse and Martin Morland .------------
//-------------------------------NTNU 2015------------------------------------
//----------------------------------------------------------------------------

#include <ros/ros.h>
#include <pcl_ros/point_cloud.h>
#include <ros/ros.h>
#include <iostream>
#include <pcl/io/pcd_io.h>
#include <pcl/point_types.h>
#include <boost/foreach.hpp>
#include <sensor_msgs/PointCloud2.h>
#include <limits>
#include <fstream>
#include <vector>
#include <Eigen/Core>
#include <pcl/point_cloud.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/features/normal_3d.h>
#include <pcl/features/fpfh.h>
#include <pcl/registration/ia_ransac.h>
#include <armadillo>
#include <sstream>
#include "std_msgs/String.h"
#include "std_msgs/Float64MultiArray.h"
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <boost/thread/thread.hpp>
#include <pcl/common/common_headers.h>
#include <pcl/visualization/pcl_visualizer.h>
#include <pcl/console/parse.h>
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/Int32MultiArray.h"
#include <armadillo>
#include <pcl/filters/filter.h>
#include <pcl/filters/statistical_outlier_removal.h>
#include <pcl/search/search.h>
#include <pcl/search/kdtree.h>
#include <pcl/segmentation/region_growing_rgb.h>
#include <pcl/visualization/cloud_viewer.h>
#include <unistd.h>



using namespace std;
using namespace arma;

float piston_x_coordinate_list[20];
float piston_y_coordinate_list[20];
float cylinder_x_coordinate_list[20];
float cylinder_y_coordinate_list[20];
int number_of_coordinates_to_save=20;
int number_of_saved_coordinates=0;
```

```cpp
int subsribed_TF_camera_table=0;
   mat Transformation_table_camera(4,4);
   mat Transformation_camera_table(4,4);
  rowvec r;
  colvec temporary_point_with_respect_to_table;
  colvec temporary_point;
std_msgs::Float64MultiArray publishing_coordinates_cylinder;
std_msgs::Float64MultiArray publishing_coordinates_piston;


//-- Visualize effects
      void
printUsage (const char* progName)
{
  std::cout << "\n\nUsage: "<<progName<<" [options]\n\n"
            << "Options:\n"
            << "-------------------------------------------\n"
            << "-h           this help\n"
            << "-s           Simple visualisation example\n"
            << "-r           RGB colour visualisation example\n"
            << "-c           Custom colour visualisation example\n"
            << "-n           Normals visualisation example\n"
            << "-a           Shapes visualisation example\n"
            << "-v           Viewports example\n"
            << "-i           Interaction Customization example\n"
            << "\n\n";
}


// -- Visualize point cloud
boost::shared_ptr<pcl::visualization::PCLVisualizer> rgbVis (pcl::PointCloud<pcl::↩
    PointXYZRGB>::ConstPtr cloud)
{
  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer (new pcl::visualization::↩
      PCLVisualizer ("3D Viewer"));
  viewer->setBackgroundColor (0, 0, 0);
  pcl::visualization::PointCloudColorHandlerRGBField<pcl::PointXYZRGB> rgb(cloud);
  viewer->addPointCloud<pcl::PointXYZRGB> (cloud, rgb, "sample cloud");
  viewer->setPointCloudRenderingProperties (pcl::visualization::↩
      PCL_VISUALIZER_POINT_SIZE, 3, "sample cloud");
  viewer->addCoordinateSystem (5.0);
  viewer->initCameraParameters ();
  return (viewer);
}



// -- Visualize normals
boost::shared_ptr<pcl::visualization::PCLVisualizer> normalsVisRGB (
    pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr cloud, pcl::PointCloud<pcl::Normal>::↩
        ConstPtr normals)
{
  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer (new pcl::visualization::↩
      PCLVisualizer ("3D Viewer"));
  viewer->setBackgroundColor (0, 0, 0);
  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZRGB> rgb(cloud, 0, 255,↩
       0);
  viewer->addPointCloud<pcl::PointXYZRGB> (cloud,rgb, "sample cloud");
  viewer->setPointCloudRenderingProperties (pcl::visualization::↩
      PCL_VISUALIZER_POINT_SIZE, 3, "sample cloud");
  viewer->addPointCloudNormals<pcl::PointXYZRGB, pcl::Normal> (cloud, normals, 5, 0.02,↩
       "normals");
  viewer->initCameraParameters ();
  return (viewer);
}



// --Visualize viewport
boost::shared_ptr<pcl::visualization::PCLVisualizer> viewportsVis (
    pcl::PointCloud<pcl::PointXYZRGB>::ConstPtr cloud, pcl::PointCloud<pcl::Normal>::↩
        ConstPtr normals1, pcl::PointCloud<pcl::Normal>::ConstPtr normals2)
{
  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer (new pcl::visualization::↩
      PCLVisualizer ("3D Viewer"));
  viewer->initCameraParameters ();
  int v1(0);
  viewer->createViewPort(0.0, 0.0, 0.5, 1.0, v1);
  viewer->setBackgroundColor (0, 0, 0, v1);
  viewer->addText("Radius: 0.01", 10, 10, "v1 text", v1);
  pcl::visualization::PointCloudColorHandlerRGBField<pcl::PointXYZRGB> rgb(cloud);
  viewer->addPointCloud<pcl::PointXYZRGB> (cloud, rgb, "sample cloud1", v1);
  int v2(0);
  viewer->createViewPort(0.5, 0.0, 1.0, 1.0, v2);
  viewer->setBackgroundColor (0.3, 0.3, 0.3, v2);
  viewer->addText("Radius: 0.1", 10, 10, "v2 text", v2);
  pcl::visualization::PointCloudColorHandlerCustom<pcl::PointXYZRGB> single_color(cloud↩
      , 0, 255, 0);
  viewer->addPointCloud<pcl::PointXYZRGB> (cloud, single_color, "sample cloud2", v2);
  viewer->setPointCloudRenderingProperties (pcl::visualization::↩
      PCL_VISUALIZER_POINT_SIZE, 3, "sample cloud1");
  viewer->setPointCloudRenderingProperties (pcl::visualization::↩
      PCL_VISUALIZER_POINT_SIZE, 3, "sample cloud2");
  viewer->addCoordinateSystem (1.0);
  viewer->addPointCloudNormals<pcl::PointXYZRGB, pcl::Normal> (cloud, normals1, 10, ↩
      0.05, "normals1", v1);
  viewer->addPointCloudNormals<pcl::PointXYZRGB, pcl::Normal> (cloud, normals2, 10, ↩
      0.05, "normals2", v2);
  return (viewer);
}


//-- Effects in visualization window
unsigned int text_id = 0;
void keyboardEventOccurred (const pcl::visualization::KeyboardEvent &event,
                            void* viewer_void)
{
  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer = *static_cast<boost::↩
      shared_ptr<pcl::visualization::PCLVisualizer> *> (viewer_void);
  if (event.getKeySym () == "r" && event.keyDown ())
  {
    std::cout << "r was pressed => removing all text" << std::endl;

    char str[512];
    for (unsigned int i = 0; i < text_id; ++i)
    {
      sprintf (str, "text#%03d", i);
      viewer->removeShape (str);
    }
    text_id = 0;
  }
```

```
}

//-- Effects in visualization window
void mouseEventOccurred (const pcl::visualization::MouseEvent &event,
                         void* viewer_void)
{
  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer = *static_cast<boost::↩
      shared_ptr<pcl::visualization::PCLVisualizer> *> (viewer_void);
  if (event.getButton () == pcl::visualization::MouseEvent::LeftButton &&
      event.getType () == pcl::visualization::MouseEvent::MouseButtonRelease)
  {
    std::cout << "Left mouse button released at position (" << event.getX () << ", " <<↩
        event.getY () << ")" << std::endl;

    char str[512];
    sprintf (str, "text#%03d", text_id ++);
    viewer->addText ("clicked here", event.getX (), event.getY (), str);
  }
}

//-- Effects in visualization window
boost::shared_ptr<pcl::visualization::PCLVisualizer> interactionCustomizationVis ()
{
  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer (new pcl::visualization::↩
      PCLVisualizer ("3D Viewer"));
  viewer->setBackgroundColor (0, 0, 0);
  viewer->addCoordinateSystem (1.0);

  viewer->registerKeyboardCallback (keyboardEventOccurred, (void*)&viewer);
  viewer->registerMouseCallback (mouseEventOccurred, (void*)&viewer);

  return (viewer);
}


//-- Transformation from camera to table
    void callbacknh_Tf(const std_msgs::Float64MultiArray::ConstPtr& msg)
   {
     Transformation_camera_table << msg->data[0] << msg->data[4] << msg->data[8] << msg↩
         ->data[12] << arma::endr
     << msg->data[1] << msg->data[5] << msg->data[9] << msg->data[13] << arma::endr
     << msg->data[2] << msg->data[6] << msg->data[10] << msg->data[14] << arma::endr
     << msg->data[3] << msg->data[7] << msg->data[11] << msg->data[15] << arma::endr;
    subsribed_TF_camera_table =1;

   }

 // -- Filter out NAN points
void filterOutNans(const pcl::PointCloud<pcl::PointXYZRGB>& _inCloud, pcl::PointCloud<↩
    pcl::PointXYZRGB>& _outCloud) {
    // Filter out NaNs from data
    pcl::PassThrough<pcl::PointXYZRGB> NANRemover;
    NANRemover.setInputCloud(_inCloud.makeShared());
    NANRemover.setFilterFieldName("z");
    NANRemover.setFilterLimits(-10.0, 10.0); /// Get valid range in kinect point cloud ↩
        data
    NANRemover.filter(_outCloud);
}



```

```

// -- Median estimation
float GetMedian(float daArray[], int iSize) {
    // Allocate an array of the same size and sort it.
    double* dpSorted = new double[iSize];
    for (int i = 0; i < iSize; ++i) {
        dpSorted[i] = daArray[i];
    }
    for (int i = iSize - 1; i > 0; --i) {
        for (int j = 0; j < i; ++j) {
            if (dpSorted[j] > dpSorted[j+1]) {
                double dTemp = dpSorted[j];
                dpSorted[j] = dpSorted[j+1];
                dpSorted[j+1] = dTemp;
            }
        }
    }

    // Middle or average of middle values in the sorted array.
    double dMedian = 0.0;
    if ((iSize % 2) == 0) {
        dMedian = (dpSorted[iSize/2] + dpSorted[(iSize/2) - 1])/2.0;
    } else {
        dMedian = dpSorted[iSize/2];
    }
    delete [] dpSorted;
    return dMedian;
}


// -- The point clouds used
  pcl::PointCloud<pcl::PointXYZRGB> cloud_from_camera;
  pcl::PointCloud<pcl::PointXYZRGB> cloud_table_removed_NAN;
  pcl::PointCloud<pcl::PointXYZRGB> cloud_table;
  pcl::PointCloud<pcl::PointXYZRGB> cloud_not_on_table;
  pcl::PointCloud<pcl::PointXYZRGB> first_cloud_from_camera;
  pcl::PointCloud<pcl::PointXYZRGB> second_cloud_from_camera;
  pcl::PointCloud<pcl::PointXYZRGB> third_cloud_from_camera;
  pcl::PointCloud<pcl::PointXYZRGB> fourth_cloud_from_camera;
  pcl::PointCloud<pcl::PointXYZ> segment_piston;
  pcl::PointCloud<pcl::PointXYZ> segment_piston_pcd;

// -- The class
//-----------------
class SubscribeAndPublish
{
public:
  SubscribeAndPublish()
  {
   // The different topics beeing published to and subscribed to
   publisher_cylinder = n_.advertise<std_msgs::Float64MultiArray>("cylinder_coordinates↩
       ", 1);
   publisher_piston = n_.advertise<std_msgs::Float64MultiArray>("piston_coordinates", ↩
       1);
  subscribe_point_cloud = n_.subscribe<pcl::PointCloud<pcl::PointXYZRGB> >("/camera/↩
      depth_registered/points", 1, &SubscribeAndPublish::callbackcolor, this);

  }

// -- The main function
  void callbackcolor(const pcl::PointCloud<pcl::PointXYZRGB >::ConstPtr& msg)
  {
    //-- The live point cloud from camera
       pcl::PointCloud<pcl::PointXYZRGB> cloud_from_camera=*msg;

       // Visualize the live point cloud
  pcl::visualization::CloudViewer viewer_in_cloud ("Pointcloud from camera");
  viewer_in_cloud.showCloud(msg);
   while (!viewer_in_cloud.wasStopped ())
   {
    }

//-- The code starts run when the transformation from camera to table is subscribed
    if(subsribed_TF_camera_table==1)
    {
       temporary_point << 1 << 1 << 1 << 1;
       q_cloud << 1 << 1 << 1 << 1;;
       Transformation_table_camera=inv(Transformation_camera_table);

//--Point cloud given with respect to table
     for (size_t i = 0; i <  cloud_from_camera.points.size (); ++i)
  {
     //-- Transform every point in point cloud to be given with (X,Y,Z) coordinates ↩
         with respect to table
   temporary_point <<  cloud_from_camera.points[i].x << cloud_from_camera.points[i].y ↩
       <<  cloud_from_camera.points[i].z << 1;
   temporary_point_with_respect_to_table = Transformation_table_camera* temporary_point↩
       ;
   r = temporary_point_with_respect_to_table.t();

   cloud_from_camera.points[i].x = r(0);
    cloud_from_camera.points[i].y = r(1);
    cloud_from_camera.points[i].z = r(2);

  }

  //-- Visualize the point cloud given with respect to table
    pcl::PointCloud <pcl::PointXYZRGB >::Ptr show_cloud(&cloud_from_camera);
   pcl::visualization::CloudViewer viewer_out_cloud ("Pointcloud transformed");

   viewer_out_cloud.showCloud(show_cloud);
 while (!viewer_out_cloud.wasStopped ())
  {
  }


 //-- Number of points in cloud on the table
  int number_of_points_on_table =0;
  int number_of_points_not_on_table=0;
      for (size_t i = 0; i < cloud_from_camera.points.size (); ++i)
  {
    if( (cloud_from_camera.points[i].z>0.01)&&(cloud_from_camera.points[i].x>0)&&(↩
        cloud_from_camera.points[i].x<0.8)&&(cloud_from_camera.points[i].y>0)&&(↩
        cloud_from_camera.points[i].y<0.8))
    {
    number_of_points_on_table++;
    }
    else
    {
     number_of_points_not_on_table++;
    }

  }

//-- Save point cloud
  pcl::io::savePCDFileASCII ("/home/minions/ff/scene.pcd", cloud_from_camera);


// -- Segment the part of the point cloud on the table and the part not on the table
  cloud_table.clear();
    cloud_table.width = number_of_points_on_table;
    cloud_table.height = 1;
    cloud_table.is_dense = cloud_from_camera.is_dense;
  cloud_table.points.resize (cloud_table.width * cloud_table.height);

  cloud_not_on_table.clear();
    cloud_not_on_table.width = number_of_points_not_on_table;
    cloud_not_on_table.height = 1;
    cloud_not_on_table.is_dense = cloud_from_camera.is_dense;
  cloud_not_on_table.points.resize (cloud_not_on_table.width * cloud_not_on_table.↩
      height);

  int index_point_on_table=0;
  int index_point_not_on_table=0;
   for (size_t i = 0; i < cloud_from_camera.points.size (); ++i)
  {
    if( (cloud_from_camera.points[i].z>0.01)&&(cloud_from_camera.points[i].x>0)&&(↩
        cloud_from_camera.points[i].x<0.8)&&(cloud_from_camera.points[i].y>0)&&(↩
        cloud_from_camera.points[i].y<0.8))
    {
    cloud_table.points[index_point_on_table].x = cloud_from_camera.points[i].x;
    cloud_table.points[index_point_on_table].y = cloud_from_camera.points[i].y ;
    cloud_table.points[index_point_on_table].z = cloud_from_camera.points[i].z;

      cloud_table.points[index_point_on_table].r = cloud_from_camera.points[i].r;
    cloud_table.points[index_point_on_table].g = cloud_from_camera.points[i].g ;
    cloud_table.points[index_point_on_table].b = cloud_from_camera.points[i].b;
      index_point_on_table++;
    }
    else
    {
           cloud_not_on_table.points[index_point_not_on_table].x = cloud_from_camera.↩
               points[i].x;
    cloud_not_on_table.points[index_point_not_on_table].y = cloud_from_camera.points[i↩
        ].y ;
    cloud_not_on_table.points[index_point_not_on_table].z = cloud_from_camera.points[i↩
        ].z;
     cloud_not_on_table.points[index_point_not_on_table].r = cloud_from_camera.points[i↩
         ].r;
    cloud_not_on_table.points[index_point_not_on_table].g = cloud_from_camera.points[i↩
        ].g ;
    cloud_not_on_table.points[index_point_not_on_table].b = cloud_from_camera.points[i↩
        ].b;
      index_point_not_on_table++;
    }
  }





// -- Save those two segments
```

```
  pcl::io::savePCDFileASCII ("/home/minions/pcl/build/pcd/scene_outlier_not_table.pcd",↩
       cloud_not_on_table);
  pcl::io::savePCDFileASCII ("/home/minions/pcl/build/pcd/scene_inlier_table.pcd", ↩
      cloud_table);


//-- Remove remaining NAN-points
filterOutNans(cloud_table, cloud_table_removed_NAN);


//-- Statistical outliers removal
  pcl::StatisticalOutlierRemoval<pcl::PointXYZRGB> statistical_filter;
statistical_filter.setInputCloud(boost::make_shared<pcl::PointCloud<pcl::PointXYZRGB> ↩
    >(cloud_table_removed_NAN));
 statistical_filter.setMeanK(80);
  statistical_filter.setStddevMulThresh(2.9);
  statistical_filter.filter(cloud_table_removed_NAN);
    pcl::PCDWriter writer;
  writer.write<pcl::PointXYZRGB> ("/home/minions/pcl/build/pcd/scene_inliers.pcd", ↩
      cloud_table_removed_NAN, false);
 statistical_filter.setNegative (true);
  statistical_filter.filter (cloud_table_removed_NAN);
   writer.write<pcl::PointXYZRGB> ("/home/minions/pcl/build/pcd/scene_outliers.pcd", ↩
       cloud_table_removed_NAN, false);
 statistical_filter.setNegative (false);
 statistical_filter.filter (cloud_table_removed_NAN);
   pcl::PointCloud <pcl::PointXYZRGB >::Ptr cloud_ptr;
     cloud_ptr.reset (new pcl::PointCloud<pcl::PointXYZRGB> (cloud_table_removed_NAN));


//-- Normal estimation of the cloud on the table
  pcl::NormalEstimation<pcl::PointXYZRGB, pcl::Normal> ne;
  ne.setInputCloud (cloud_ptr);
  pcl::search::KdTree<pcl::PointXYZRGB >::Ptr tree (new pcl::search::KdTree<pcl::↩
      PointXYZRGB> ());
  ne.setSearchMethod (tree);
   ne.setViewPoint(0.4,1.1,0.3);
    pcl::PointCloud<pcl::Normal >::Ptr cloud_normals (new pcl::PointCloud<pcl::Normal>);
 ne.setRadiusSearch (0.04);
  ne.compute (*cloud_normals);


//-- Visualize the normals
   boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer;
viewer = normalsVisRGB(cloud_ptr, cloud_normals);
   while (!viewer->wasStopped ())
  {
    viewer->spinOnce (100);
    boost::this_thread::sleep (boost::posix_time::microseconds (100000));
  }


//-- Segment only those points that are listed in the indices (used for later)
    pcl::IndicesPtr indices (new std::vector <int>);
  pcl::PassThrough<pcl::PointXYZRGB> pass;
 pass.setInputCloud (cloud_ptr);
    pass.filter (*indices);


  //-- Region growing segmentations
  pcl::RegionGrowing<pcl::PointXYZRGB, pcl::Normal> reg;
```

```
    reg.setResidualThreshold(150);
  reg.setMinClusterSize (70);
   reg.setMaxClusterSize (100000);
   reg.setResidualTestFlag(true);
  reg.setSearchMethod (tree);
reg.setNumberOfNeighbours (20);
  reg.setInputCloud (cloud_ptr);
  reg.setIndices (indices);
  reg.setInputNormals (cloud_normals);
 reg.setSmoothnessThreshold (70.0 / 180.0 * M_PI);
  std::vector <pcl::PointIndices> clusters;
  reg.extract (clusters);
  pcl::PointCloud <pcl::PointXYZRGB >::Ptr colored_cloud = reg.getColoredCloud ();


  //-- Visualize the segments from region growing segmentations
pcl::visualization::CloudViewer viewer_cloud ("Cluster viewer");
viewer_cloud.showCloud(colored_cloud);
 while (!viewer_cloud.wasStopped ())
  {
  }


  //-- Segment cylinder and piston
   pcl::PointCloud <pcl::PointXYZRGB> segment_cylinder;
     pcl::PointCloud <pcl::PointXYZRGB> segment_piston;
  int j = 0;
      int size_piston_array=1;
int size_cylinder_array=1;


 //-- loops through the segments from region growing
  for (std::vector<pcl::PointIndices >::const_iterator it = clusters.begin (); it != ↩
      clusters.end (); ++it)
  {
    float distance_from_table=0;

    pcl::PointCloud<pcl::PointXYZRGB >::Ptr cloud_cluster (new pcl::PointCloud<pcl::↩
        PointXYZRGB >);
    for (std::vector<int >::const_iterator pit = it->indices.begin (); pit != it->↩
        indices.end (); ++pit)
      cloud_cluster->points.push_back (cloud_ptr->points[*pit]);
    cloud_cluster->width = cloud_cluster->points.size ();
    cloud_cluster->height = 1;
    cloud_cluster->is_dense = true;

    int piston_or_cylinder =0;

    //-- Checks the different segments for a match for either beeing the piston or ↩
        cylinder
    for(size_t i = 0; i < cloud_cluster->points.size (); ++i)
    {
       if(cloud_cluster->points[i].z >0.05)
       {
    piston_or_cylinder=1;
       }
    }
    if(piston_or_cylinder==1 &&  cloud_cluster->points.size () >50)
    {
      for (size_t i = 0; i < cloud_cluster->points.size (); ++i)
      {
```

```
  if(cloud_cluster->points[i].z >distance_from_table)
  {
   distance_from_table=cloud_cluster->points[i].z;
   }
   }
   if(distance_from_table >0.135)
   {
  //-- Segment cylinder
  segment_cylinder=*cloud_cluster;
  size_cylinder_array=cloud_cluster->points.size ();

   }
   else
   {
  //-- Segment piston
  segment_piston=*cloud_cluster;
  size_piston_array=cloud_cluster->points.size ();
   }
  j++;
  }
 }


 float array_cylinder_y_coordinates_in_segment[size_cylinder_array];
float array_cylinder_x_coordinates_in_segment[size_cylinder_array];
float array_piston_y_coordinates_in_segment[size_piston_array];
float array_piston_x_coordinates_in_segment[size_piston_array];


//-- Extract the points from the segments
    for (size_t i = 0; i < segment_piston.points.size () ; ++i)
   {
   array_piston_x_coordinates_in_segment[i]=segment_piston.points[i].x;
   array_piston_y_coordinates_in_segment[i]=segment_piston.points[i].y;
  }

    for (size_t i = 0; i < segment_cylinder.points.size () ; ++i)
   {
   array_cylinder_x_coordinates_in_segment[i]=segment_cylinder.points[i].x;
   array_cylinder_y_coordinates_in_segment[i]=segment_cylinder.points[i].y;
  }



 //-- Save the segments of the piston and the cylinder in pcd-files
pcl::io::savePCDFileASCII ("/home/minions/pcl/build/pcd/cylinder.pcd", segment_cylinder);
 pcl::io::savePCDFileASCII ("/home/minions/pcl/build/pcd/piston.pcd", segment_piston);




 //--- Coordinates are extracted from the segment. That is, the median of the coordinates in the segments
 float piston_x_coordinate = GetMedian(array_piston_x_coordinates_in_segment,size_piston_array);
 float piston_y_coordinate = GetMedian(array_piston_y_coordinates_in_segment,size_piston_array);
 float cylinder_x_coordinate = GetMedian(array_cylinder_x_coordinates_in_segment,size_cylinder_array);
 float cylinder_y_coordinate = GetMedian(array_cylinder_y_coordinates_in_segment,↩
    size_cylinder_array);



 //-- Write coordinates to txt-files
           ofstream cylinder_file;
 cylinder_file.open ("cylinder_coordinates.txt",std::ios_base::app);
 cylinder_file << cylinder_x_coordinate;
  cylinder_file << " , ";
  cylinder_file << cylinder_y_coordinate;
 cylinder_file << endl;
 cylinder_file.close();

    ofstream piston_file;
 piston_file.open ("Piston_coordinates.txt",std::ios_base::app);
  piston_file << piston_x_coordinate;
  piston_file << " , ";
  piston_file << piston_y_coordinate;
 piston_file << endl;
 piston_file.close();




//-- Save the last 20 coordinates
 for(int x = number_of_coordinates_to_save-1;x>=0;x--){
  piston_x_coordinate_list[x+1]=piston_x_coordinate_list[x];
  piston_y_coordinate_list[x+1]=piston_y_coordinate_list[x];
  cylinder_x_coordinate_list[x+1]=cylinder_x_coordinate_list[x];
  cylinder_y_coordinate_list[x+1]=cylinder_y_coordinate_list[x];
   if(x==0){
  piston_x_coordinate_list[x]=piston_x_coordinate;
  piston_y_coordinate_list[x]=piston_y_coordinate;
  cylinder_x_coordinate_list[x]=cylinder_x_coordinate;
  cylinder_y_coordinate_list[x]=cylinder_y_coordinate;
  number_of_saved_coordinates++;
 }
 }


 //-- Median of the 20 last saved coordinates. These are the accepted coordinates
  if(number_of_saved_coordinates>=number_of_coordinates_to_save)
  {
float final_piston_x_coordinate = GetMedian(piston_x_coordinate_list,↩
  number_of_coordinates_to_save);
float final_piston_y_coordinate = GetMedian(piston_y_coordinate_list,↩
  number_of_coordinates_to_save);
float final_cylinder_x_coordinate = GetMedian(cylinder_x_coordinate_list,↩
  number_of_coordinates_to_save);
float final_cylinder_y_coordinate = GetMedian(cylinder_y_coordinate_list,↩
  number_of_coordinates_to_save);


 //-- Print out the coordinates
  std::cout << "cylinder coordinates: " << final_cylinder_x_coordinate << " , " << ↩
      final_cylinder_y_coordinate << endl;
    std::cout << "piston coordinates: " << final_piston_x_coordinate << " , " << ↩
        final_piston_y_coordinate << endl;


//-- Publish cylinder coordinates
publishing_coordinates_cylinder.data.clear();
    for (int i = 0; i < 3; i++)
{
 if(i==0)
 {
publishing_coordinates_cylinder.data.push_back(final_cylinder_x_coordinate);
 }
 if(i==1)
 {
  publishing_coordinates_cylinder.data.push_back(final_cylinder_y_coordinate);
 }
 if(i==2)
 {
 publishing_coordinates_cylinder.data.push_back(0.17);
 }
}
 publisher_cylinder.publish(publishing_coordinates_cylinder);


 //-- Publish piston coordinates
 publishing_coordinates_piston.data.clear();
    for (int i = 0; i < 3; i++)
{
 if(i==0)
 {
publishing_coordinates_piston.data.push_back(final_piston_x_coordinate);
 }
 if(i==1)
 {
  publishing_coordinates_piston.data.push_back(final_piston_y_coordinate);
 }
 if(i==2)
 {
 publishing_coordinates_piston.data.push_back(0.10);
 }
}
publisher_piston.publish(publishing_coordinates_piston);
  }
 }
}



private:
 ros::NodeHandle n_;
 ros::Publisher publisher_cylinder;
 ros::Publisher publisher_piston;
 ros::Subscriber subscribe_point_cloud;

};

int main(int argc, char **argv)
{

 //Initiate ROS.  The node name is "Pointcloud"
 ros::init(argc, argv, "Pointcloud");
 ros::NodeHandle nh;

 //-- Subscribe Transformation from camera to table
```

```
ros::Subscriber subnh = nh.subscribe<std_msgs::Float64MultiArray>("↩
    Transformation_from_camera_to_table", 1, callbacknh_Tf);

//-- Create an object of class SubscribeAndPublish that will take care of everything
  SubscribeAndPublish SAPObject;
  ros::spin();

  return 0;
}
```

```
//-------------------------------------------------------------------------------
//-------------------- Node run for estimating orientation of cylinder --------
//-------------------------------------------------------------------------------
//-------------------------------------------------------------------------------
//--------------------Authors: Geir Ole Tysse and Martin Morland .--------------
//----------------------------------NTNU 2015------------------------------------
//-------------------------------------------------------------------------------

#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/core/core.hpp>
#include "opencv2/calib3d/calib3d.hpp"
#include <stdio.h>
#include <iostream>
#include <armadillo>
#include <cmath>
#include "std_msgs/Float64MultiArray.h"
using namespace cv;
using namespace std;
static const std::string OPENCV_WINDOW = "Image window";


float degree_estimated_for_0;
float degree_estimated_for_90;
float degree_estimated_for_180;
float degree_estimated_for_270;
arma::mat camera_matrix(3,3);
float true_orientation;
int number_of_times_approved_H_0=0;
int number_of_times_approved_H_90=0;
int number_of_times_approved_H_180=0;
int number_of_times_approved_H_270=0;
float estimated_orientation;
float list_with_last_degrees[7];
std_msgs::Float64MultiArray array_orientation_cylinder;

//-- Estimate the median
float GetMedian(float daArray[], int iSize) {
    double* dpSorted = new double[iSize];
    for (int i = 0; i < iSize; ++i) {
        dpSorted[i] = daArray[i];
    }
    for (int i = iSize - 1; i > 0; --i) {
        for (int j = 0; j < i; ++j) {
            if (dpSorted[j] > dpSorted[j+1]) {
                double dTemp = dpSorted[j];
                dpSorted[j] = dpSorted[j+1];
                dpSorted[j+1] = dTemp;
            }
        }
    }
    double dMedian = 0.0;
    if ((iSize % 2) == 0) {
        dMedian = (dpSorted[iSize/2] + dpSorted[(iSize/2) - 1])/2.0;
    } else {
        dMedian = dpSorted[iSize/2];
    }
    delete [] dpSorted;
    return dMedian;
}


class ImageConverter
{
  ros::NodeHandle nh_;
  image_transport::ImageTransport it_;
  image_transport::Subscriber image_sub_;
  image_transport::Publisher image_pub_;
  ros::Publisher publish_orientation_cylinder;
private:



public:


  ImageConverter()
    : it_(nh_)
  {
    //-- Subscrive to input video feed
    image_sub_ = it_.subscribe("/usb_cam/image_raw", 1,
      &ImageConverter::imageCb, this);
    cv::namedWindow(OPENCV_WINDOW,WINDOW_NORMAL);

    //-- Publish orientation of the cylinder
    publish_orientation_cylinder= nh_.advertise<std_msgs::Float64MultiArray>("↩
        cylinder_orientation", 1);


  }

  ~ImageConverter()
  {
    cv::destroyWindow(OPENCV_WINDOW);

  }

  void imageCb(const sensor_msgs::ImageConstPtr& msg)
  {

    cv_bridge::CvImagePtr cv_ptr;
    try
    {
      cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
      ROS_ERROR("cv_bridge exception: %s", e.what());
      return;
    }

//-- Get the live image stream and the training images
Mat img_show;
Mat img_object = imread("/home/jorge/catkin_ws/syl_0.jpg");
Mat img_object_90 = imread("/home/jorge/catkin_ws/syl_90.jpg");
Mat img_object_180 = imread("/home/jorge/catkin_ws/syl_180.jpg");
Mat img_object_270 = imread("/home/jorge/catkin_ws/syl_270.jpg");
Mat img_scene = cv_ptr->image;

//-- Get gray image from the image stream also, used for reading greyscale values
Mat gray_image;
 cvtColor( img_scene, gray_image, CV_BGR2GRAY );


if( !img_object.data || !img_scene.data )
  { std::cout<< " --(!) Error reading images " << std::endl; }

  //-- Step 1: Detect the keypoints using Sift Detector
  int minHessian = 200;
SiftFeatureDetector detector( minHessian );
std::vector<KeyPoint> keypoints_object, keypoints_scene,  keypoints_object_90,  ↩
    keypoints_object_180,  keypoints_object_270;

//-- Detection of feature in image stream. Searches within a defined mask
Mat mask = Mat::zeros(img_scene.size(), CV_8U);
Mat roi(mask, cv::Rect(200,50,220,280));
roi = Scalar(255, 255, 255);
detector.detect(img_scene, keypoints_scene, mask);


//-- Detection of feature in training images
detector.detect( img_object, keypoints_object );
detector.detect( img_object_90, keypoints_object_90);
detector.detect( img_object_180, keypoints_object_180 );
detector.detect( img_object_270, keypoints_object_270 );




  //-- Step 2: Calculate descriptors
  SiftDescriptorExtractor extractor;

  Mat descriptors_object, descriptors_scene, descriptors_object_90, ↩
      descriptors_object_180, descriptors_object_270;

  extractor.compute( img_object, keypoints_object, descriptors_object );
  extractor.compute( img_object_90, keypoints_object_90, descriptors_object_90);
  extractor.compute(  img_object_180, keypoints_object_180 , descriptors_object_180 );
  extractor.compute(img_object_270, keypoints_object_270, descriptors_object_270 );
  extractor.compute( img_scene, keypoints_scene, descriptors_scene );

  //-- Step 3: Matching descriptor vectors using FLANN matcher
 FlannBasedMatcher matcher;

  std::vector< DMatch > matches,matches_90 ,matches_180, matches_270;
  matcher.match( descriptors_object, descriptors_scene, matches );
  matcher.match( descriptors_object_90, descriptors_scene, matches_90 );
  matcher.match( descriptors_object_180, descriptors_scene, matches_180 );
  matcher.match( descriptors_object_270, descriptors_scene, matches_270 );


std::vector< DMatch > good_matches, good_matches_90, good_matches_180, good_matches_270↩
    ;

   //-- Matching descriptor vectors in image stream and training image 0 degree
  Mat  img_matches_0;
  drawMatches( img_object, keypoints_object, img_scene, keypoints_scene,
               good_matches,  img_matches_0, Scalar::all(-1), Scalar::all(-1),
               vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

  //-- Matching descriptor vectors in image stream and training image 90 degree
 Mat  img_matches_90;
  drawMatches( img_object_90, keypoints_object_90, img_scene, keypoints_scene,
               good_matches_90,  img_matches_90, Scalar::all(-1), Scalar::all(-1),
               vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

     //-- Matching descriptor vectors in image stream and training image 180 degree
    Mat  img_matches_180;
  drawMatches( img_object_180, keypoints_object_180, img_scene, keypoints_scene,
               good_matches_180,  img_matches_180, Scalar::all(-1), Scalar::all(-1),
               vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

   //-- Matching descriptor vectors in image stream and training image 270 degree
      Mat  img_matches_270;
  drawMatches( img_object_270, keypoints_object_270, img_scene, keypoints_scene,
               good_matches_270,  img_matches_270, Scalar::all(-1), Scalar::all(-1),
               vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );



   //-- Quick calculation of max and min distances between keypoints in 0 degrees ↩
       training image and image stream
    double max_dist = 0; double min_dist = 70;

  for( int i = 0; i < descriptors_object.rows; i++ )
  { double dist = matches[i].distance;
    if( dist < min_dist ) min_dist = dist;
    if( dist > max_dist ) max_dist = dist;
  }



  for( int i = 0; i < descriptors_object.rows; i++ )
  { if( matches[i].distance < 3*min_dist )
     { good_matches.push_back( matches[i]); }
  }


  //-- Quick calculation of max and min distances between keypoints in 90 degrees ↩
      training image and image stream
    max_dist = 0;
    min_dist = 70;

  for( int i = 0; i < descriptors_object_90.rows; i++ )
  { double dist = matches_90[i].distance;
    if( dist < min_dist ) min_dist = dist;
    if( dist > max_dist ) max_dist = dist;
  }
  for( int i = 0; i < descriptors_object_90.rows; i++ )
  { if( matches_90[i].distance < 3*min_dist )
     { good_matches_90.push_back( matches_90[i]); }
  }

  //-- Quick calculation of max and min distances between keypoints in 180 degrees ↩
      training image and image stream
    max_dist = 0;
    min_dist = 70;

  for( int i = 0; i < descriptors_object_180.rows; i++ )
  { double dist = matches_180[i].distance;
    if( dist < min_dist ) min_dist = dist;
    if( dist > max_dist ) max_dist = dist;
  }
  for( int i = 0; i < descriptors_object_180.rows; i++ )
  { if( matches_180[i].distance < 3*min_dist )
     { good_matches_180.push_back( matches_180[i]); }
  }

   //-- Quick calculation of max and min distances between keypoints in 270 degrees ↩
       training image and image stream
    max_dist = 0;
    min_dist = 70;

  for( int i = 0; i < descriptors_object_270.rows; i++ )
  { double dist = matches_270[i].distance;
    if( dist < min_dist ) min_dist = dist;
    if( dist > max_dist ) max_dist = dist;
  }
  for( int i = 0; i < descriptors_object_270.rows; i++ )
  { if( matches_270[i].distance < 3*min_dist )
     { good_matches_270.push_back( matches_270[i]); }
  }




  //-- Localize the object
  std::vector<Point2f> obj , obj_90, obj_180 ,obj_270;
  std::vector<Point2f> scene, scene_90 , scene_180 ,scene_270;


  //TRAINING IMAGE 0 DEGREES
  //---------------
  //---------------


   //-- Get the keypoints from the good matches 0 degree
  for( int i = 0; i < good_matches.size(); i++ )
  {

    obj.push_back( keypoints_object[ good_matches[i].queryIdx ].pt );
    scene.push_back( keypoints_scene[ good_matches[i].trainIdx ].pt );
  }

  std::vector<Point2f> obj_corners(4);
   if (good_matches.size() > 3)
 {
   // -- Estimate the homography between training image 0 degrees and image stream
  cv::Mat H_0 = findHomography( obj, scene, CV_RANSAC );

  //-- Get the corners from the training image 0 degrees ( the object to be "detected" ↩
      )
  obj_corners[0] = cvPoint(0,0); obj_corners[1] = cvPoint( img_object.cols, 0 );
  obj_corners[2] = cvPoint( img_object.cols, img_object.rows ); obj_corners[3] = ↩
      cvPoint( 0, img_object.rows );
  std::vector<Point2f> scene_corners(4);

  perspectiveTransform( obj_corners, scene_corners, H_0);

  //-- Draw lines between the corners (the mapped object in the scene - training image 0 degrees)
  line(  img_matches_0, scene_corners[0] + Point2f( img_object.cols, 0), scene_corners[1] + Point2f( img_object.cols, 0), Scalar(0, 255, 0), 4 );
 line(  img_matches_0, scene_corners[1] + Point2f( img_object.cols, 0), scene_corners[2] + Point2f( img_object.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_0, scene_corners[2] + Point2f( img_object.cols, 0), scene_corners[3] + Point2f( img_object.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_0, scene_corners[3] + Point2f( img_object.cols, 0), scene_corners[0] + Point2f( img_object.cols, 0), Scalar( 0, 255, 0), 4 );




//-- Check if it is an affine transformation (training image 0)
if( H_0.at<double>(2,0) < 0.0001 &&  H_0.at<double>(2,1) < 0.0001)
{
  // check if it is a stable homography (training image 0)
 number_of_times_approved_H_0++;

 // calculate the orientation of the object (training image 0)
float hypotenuse_0 = sqrt((scene_corners[2].x-scene_corners[1].x)*(scene_corners[2].x-scene_corners[1].x)+(scene_corners[2].y-scene_corners[1].y)*(scene_corners[2].y-scene_corners[1].y));
float  cathetus_0  = scene_corners[2].y-scene_corners[1].y;
degree_estimated_for_0 = acos( cathetus_0/hypotenuse_0)*180/(arma::datum::pi);


if((scene_corners[2].x-scene_corners[1].x) < 0)
{
  degree_estimated_for_0=360-degree_estimated_for_0;
}
}
else
{
  //If it is not an stable homography (training image 0)
  number_of_times_approved_H_0=0;
}
 }




   //TRAINING IMAGE 90 DEGREES
  //---------------
  //---------------


 if (good_matches_90.size() > 3)
 {
 //-- Get the keypoints from the good matches 90 degree
  for( int i = 0; i < good_matches_90.size(); i++ )
  {

    obj_90.push_back( keypoints_object_90[ good_matches_90[i].queryIdx ].pt );
    scene_90.push_back( keypoints_scene[ good_matches_90[i].trainIdx ].pt );
  }

  // -- Estimate the homography between training image 90 degrees and image stream
  cv::Mat H_90 = findHomography( obj_90, scene_90, CV_RANSAC );
```

```

    //-- Get the corners from the training image 90 degrees ( the object to be "detected" )
  std::vector<Point2f> obj_corners_90(4);
  obj_corners_90[0] = cvPoint(0,0); obj_corners_90[1] = cvPoint( img_object_90.cols, 0 );
  obj_corners_90[2] = cvPoint( img_object_90.cols, img_object_90.rows ); obj_corners_90[3] = cvPoint( 0, img_object_90.rows );
    std::vector<Point2f> scene_corners_90(4);

      //-- Draw lines between the corners (the mapped object in the scene - training image 90 degrees)
  perspectiveTransform( obj_corners_90, scene_corners_90, H_90);
  line(  img_matches_90, scene_corners_90[0] + Point2f( img_object_90.cols, 0), scene_corners_90[1] + Point2f( img_object_90.cols, 0), Scalar(0, 255, 0), 4 );
  line(  img_matches_90, scene_corners_90[1] + Point2f( img_object_90.cols, 0), scene_corners_90[2] + Point2f( img_object_90.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_90, scene_corners_90[2] + Point2f( img_object_90.cols, 0), scene_corners_90[3] + Point2f( img_object_90.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_90, scene_corners_90[3] + Point2f( img_object_90.cols, 0), scene_corners_90[0] + Point2f( img_object_90.cols, 0), Scalar( 0, 255, 0), 4 );

    //-- Check if it is an affine transformation (training image 90)
if( H_90.at<double>(2,0) < 0.0001 &&  H_90.at<double>(2,1) < 0.0001)
{
     // check if it is a stable homography (training image 90)
number_of_times_approved_H_90++;


     // calculate the orientation of the object (training image 0)
float hypotenuse_90 = sqrt((scene_corners_90[1].x-scene_corners_90[0].x)*(scene_corners_90[1].x-scene_corners_90[0].x)+(scene_corners_90[1].y-scene_corners_90[0].y)*(scene_corners_90[1].y-scene_corners_90[0].y));
float cathetus_90 = scene_corners_90[1].x-scene_corners_90[0].x;
degree_estimated_for_90 = acos(cathetus_90/hypotenuse_90)*180/(arma::datum::pi);


if(scene_corners_90[1].y > scene_corners_90[0].y)
{
 degree_estimated_for_90= 90 - degree_estimated_for_90;
}
else
{
  degree_estimated_for_90= 90 + degree_estimated_for_90;
}

 }
 else
 {
   //If it is not an stable homography (training image 90)
   int number_of_times_approved_H_90=0;
 }
 }

     //TRAINING IMAGE 180 DEGREES
  //---------------
  //---------------


 if (good_matches_180.size() > 3)
 {
```

```
//-- Get the keypoints from the good matches 180 degree
  for( int i = 0; i < good_matches_180.size(); i++ )
  {

    obj_180.push_back( keypoints_object_180[ good_matches_180[i].queryIdx ].pt );
    scene_180.push_back( keypoints_scene[ good_matches_180[i].trainIdx ].pt );
  }


// -- Estimate the homography between training image 180 degrees and image stream
  cv::Mat H_180 = findHomography( obj_180, scene_180, CV_RANSAC );

   //-- Get the corners from the training image 180 degrees ( the object to be "detected" )
  std::vector<Point2f> obj_corners_180(4);
  obj_corners_180[0] = cvPoint(0,0); obj_corners_180[1] = cvPoint( img_object_180.cols, 0 );
  obj_corners_180[2] = cvPoint( img_object_180.cols, img_object_180.rows ); obj_corners_180[3] = cvPoint( 0, img_object_180.rows );
  std::vector<Point2f> scene_corners_180(4);

  perspectiveTransform( obj_corners_180, scene_corners_180, H_180);
   //-- Draw lines between the corners (the mapped object in the scene - training image 180 degrees)
  line(  img_matches_180, scene_corners_180[0] + Point2f( img_object_180.cols, 0), scene_corners_180[1] + Point2f( img_object_180.cols, 0), Scalar(0, 255, 0), 4 );
  line(  img_matches_180, scene_corners_180[1] + Point2f( img_object_180.cols, 0), scene_corners_180[2] + Point2f( img_object_180.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_180, scene_corners_180[2] + Point2f( img_object_180.cols, 0), scene_corners_180[3] + Point2f( img_object_180.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_180, scene_corners_180[3] + Point2f( img_object_180.cols, 0), scene_corners_180[0] + Point2f( img_object_180.cols, 0), Scalar( 0, 255, 0), 4 );

    //-- Check if it is an affine transformation (training image 0)
if( H_180.at<double>(2,0) < 0.0001 &&  H_180.at<double>(2,1) < 0.0001)
{
      // check if it is a stable homography (training image 0)
  number_of_times_approved_H_180++;

   // calculate the orientation of the object (training image 0)
  float  hypotenuse_180 = sqrt((scene_corners_180[2].x-scene_corners_180[1].x)*(scene_corners_180[2].x-scene_corners_180[1].x)+(scene_corners_180[2].y-scene_corners_180[1].y)*(scene_corners_180[2].y-scene_corners_180[1].y));
  float cathetus_180 = scene_corners_180[2].y-scene_corners_180[1].y;
degree_estimated_for_180 = acos(cathetus_180/hypotenuse_180)*180/(arma::datum::pi);

if(scene_corners_180[2].x < scene_corners_180[1].x)
{
 degree_estimated_for_180 =180-degree_estimated_for_180;
}
else
{
  degree_estimated_for_180 = 180 +degree_estimated_for_180;
}
 }
 else
 {
      //If it is not an stable homography (training image 0)
  number_of_times_approved_H_180=0;
 }
 }
```

```

      //TRAINING IMAGE 270 DEGREES
   //----------------
   //----------------











 if (good_matches_270.size() > 3)
 {
        //-- Get the keypoints from the good matches 270 degree
  for( int i = 0; i < good_matches_270.size(); i++ )
  {

    obj_270.push_back( keypoints_object_270[ good_matches_270[i].queryIdx ].pt );
    scene_270.push_back( keypoints_scene[ good_matches_270[i].trainIdx ].pt );
  }

  // -- Estimate the homography between training image 270 degrees and image stream
  cv::Mat H_270 = findHomography( obj_270, scene_270, CV_RANSAC );

  //-- Get the corners from the training image 270 degrees ( the object to be "detected↩
      " )
  std::vector<Point2f> obj_corners_270(4);
  obj_corners_270[0] = cvPoint(0,0); obj_corners_270[1] = cvPoint( img_object_270.cols,↩
       0 );
  obj_corners_270[2] = cvPoint( img_object_270.cols, img_object_270.rows ); ↩
      obj_corners_270[3] = cvPoint( 0, img_object_270.rows );
  std::vector<Point2f> scene_corners_270(4);
  perspectiveTransform( obj_corners_270, scene_corners_270, H_270);

      //-- Draw lines between the corners (the mapped object in the scene - training ↩
          image 0 degrees)
  line(  img_matches_270, scene_corners_270[0] + Point2f( img_object_270.cols, 0), ↩
      scene_corners_270[1] + Point2f( img_object_270.cols, 0), Scalar(0, 255, 0), 4 );
  line(  img_matches_270, scene_corners_270[1] + Point2f( img_object_270.cols, 0), ↩
      scene_corners_270[2] + Point2f( img_object_270.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_270, scene_corners_270[2] + Point2f( img_object_270.cols, 0), ↩
      scene_corners_270[3] + Point2f( img_object_270.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_270, scene_corners_270[3] + Point2f( img_object_270.cols, 0), ↩
      scene_corners_270[0] + Point2f( img_object_270.cols, 0), Scalar( 0, 255, 0), 4 );



  //-- Check if it is an affine transformation (training image 270)
if( H_270.at<double>(2,0) < 0.0001 &&  H_270.at<double>(2,1) < 0.0001)
{
    // check if it is a stable homography (training image 270)
number_of_times_approved_H_270++;

   // calculate the orientation of the object (training image 270)
float hypotenuse_270 = sqrt((scene_corners_270[1].x-scene_corners_270[0].x)*(↩
    scene_corners_270[1].x-scene_corners_270[0].x)+(scene_corners_270[1].y-↩
    scene_corners_270[0].y)*(scene_corners_270[1].y-scene_corners_270[0].y));
float cathetus_270 = scene_corners_270[1].y-scene_corners_270[0].y;
degree_estimated_for_270 = acos(cathetus_270/hypotenuse_270)*180/(arma::datum::pi)+180;
```

```

 }
 else
 {
      //If it is not an stable homography (training image 270)
  number_of_times_approved_H_270=0;
 }

 }


 // CHECK WICH OF THE ESTIMATED "degree_estimated_for_xx" THAT VOTE FOR THE SAME VALUE/↩
     DEGREES
 // --- ------
 //---------


// Votes for training image 0 having the correct orientation
  int g=0;
  if( (degree_estimated_for_0 < 45 || degree_estimated_for_0 >315)&& ↩
      number_of_times_approved_H_0>4)
  {
    g++;
    if( (degree_estimated_for_90 < 45 || degree_estimated_for_90 >315) && ↩
        number_of_times_approved_H_90>4)
    {
      g++;
    }
    if( (degree_estimated_for_180 < 45 || degree_estimated_for_180 >315) && ↩
        number_of_times_approved_H_180>4)
    {
      g++;
    }
    if( (degree_estimated_for_270 < 45 || degree_estimated_for_270 >270) && ↩
        number_of_times_approved_H_270 >4)
    {
      g++;
    }
  }



  // Votes for training image 90 having the correct orientation
  int g_90=0;
  if( degree_estimated_for_90 < 135 && degree_estimated_for_90 > 45 && ↩
      number_of_times_approved_H_90>4)
  {
    g_90++;
    if( degree_estimated_for_0 < 135 && degree_estimated_for_0 > 45 && ↩
        number_of_times_approved_H_0>4)
    {
      g_90++;
    }
    if( degree_estimated_for_180 < 135 && degree_estimated_for_180 >45 && ↩
        number_of_times_approved_H_180>4)
    {
      g_90++;
    }
    if( degree_estimated_for_270 < 135 && degree_estimated_for_270 >45 && ↩
        number_of_times_approved_H_270 >4)
    {
      g_90++;
    }
  }


  // Votes for training image 180 having the correct orientation
  int g_180=0;
  if( degree_estimated_for_180 < 225 && degree_estimated_for_180 > 135 && ↩
      number_of_times_approved_H_180>4)
  {
    g_180++;
    if( degree_estimated_for_0 < 225 && degree_estimated_for_0 > 135 && ↩
        number_of_times_approved_H_0>4 )
    {
      g_180++;
    }
    if( degree_estimated_for_90 < 225 && degree_estimated_for_90 >135&& ↩
        number_of_times_approved_H_90>4)
    {
      g_180++;
    }
    if( degree_estimated_for_270 < 225 && degree_estimated_for_270 >135 && ↩
        number_of_times_approved_H_270 >4)
    {
      g_180++;
    }
  }


     // Votes for training image 270 having the correct orientation
  int g_270=0;
  if( degree_estimated_for_270 < 315 && degree_estimated_for_270 > 225 && ↩
      number_of_times_approved_H_270>4)
  {
    g_270++;
    if( degree_estimated_for_0 < 315 && degree_estimated_for_0 > 225&& ↩
        number_of_times_approved_H_0>4 )
    {
      g_270++;
    }
    if( degree_estimated_for_90 < 315 && degree_estimated_for_90 >225 && ↩
        number_of_times_approved_H_90>4)
    {
      g_270++;
    }
    if( degree_estimated_for_180 < 315 && degree_estimated_for_180 >225&& ↩
        number_of_times_approved_H_180>4)
    {
      g_270++;
    }
  }


  // CHECK WHICH OF THE "degree_estimated_for_xx" THAT HAS HIGHEST VOTES
  if(g > g_90 && g > g_180  && g > g_270 )
  {
   estimated_orientation= degree_estimated_for_0;
  }

  if(g_90 > g && g_90 > g_180  && g_90 > g_270 )
  {
   estimated_orientation= degree_estimated_for_90;
  }

  if(g_180 > g && g_180 > g_90  && g_180 > g_270 )
  {
   estimated_orientation= degree_estimated_for_180;
  }

  if(g_270 > g && g_270 > g_90  && g_270 > g_180 )
  {
   estimated_orientation= degree_estimated_for_270;
  }


// --CHECK PIXELREGIONS, IS THE ORIENTATION TRUE OR 180 DEGREES PHASE SHIFTED ?
// -------------
//--------------


std::vector<Point2f> center_point_image_stream(1);
center_point_image_stream[0]=cvPoint((cv_ptr->image.cols)/2,(cv_ptr->image.rows)/2);


// Delta degrees for localizing the pixel region
 std::vector<Point2f> delta_degree_for_pixels_along_cylinder_edge(5);
   delta_degree_for_pixels_along_cylinder_edge[0].x= cos((423-estimated_orientation)*(↩
       arma::datum::pi)/180)*100;
delta_degree_for_pixels_along_cylinder_edge[0].y= sin((423-estimated_orientation)*(arma↩
    ::datum::pi)/180)*100;
 delta_degree_for_pixels_along_cylinder_edge[1].x= cos((452-estimated_orientation)*(↩
     arma::datum::pi)/180)*100;
delta_degree_for_pixels_along_cylinder_edge[1].y= sin((452-estimated_orientation)*(arma↩
    ::datum::pi)/180)*100;
   delta_degree_for_pixels_along_cylinder_edge[2].x= cos((431-estimated_orientation)*(↩
       arma::datum::pi)/180)*100;
delta_degree_for_pixels_along_cylinder_edge[2].y= sin((431-estimated_orientation)*(arma↩
    ::datum::pi)/180)*100;
   delta_degree_for_pixels_along_cylinder_edge[3].x= cos((427-estimated_orientation)*(↩
       arma::datum::pi)/180)*100;
delta_degree_for_pixels_along_cylinder_edge[3].y= sin((427-estimated_orientation)*(arma↩
    ::datum::pi)/180)*100;


   uint16_t greyscale_values[3];
   float radius_cylinder =0.055;
   float distance_from_camera_to_cylinder=0.155;
   float delta_degree_for_pixelregion_around_cylinder [3] = {423,427,431};
   std::vector<Point2f> visualize_pixel_region_edge_object(3);

   arma::colvec norm_pix_1, norm_pix_2, norm_pix_3, pix_1, pix_2, pix_3;

   // Camera matrix
   arma::mat camera_matrix(3,3);
   camera_matrix << 540.927271 << 0 << 320 <<arma::endr
   << 0 << 540.264582 << 240 <<arma::endr
   << 0 << 0 << 1 <<arma::endr;



 for(int i=0;i<3;i++)
 {
```

```

 if(i==0)
 {
// Localise one pixel and its greyscale value
  norm_pix_1 << radius_cylinder*cos((delta_degree_for_pixelregion_around_cylinder[0]-↩
      estimated_orientation)*(arma::datum::pi)/180)/distance_from_camera_to_cylinder <<↩
       radius_cylinder*sin((delta_degree_for_pixelregion_around_cylinder[0]-↩
      estimated_orientation)*(arma::datum::pi)/180)/distance_from_camera_to_cylinder <<↩
       1;
   pix_1=camera_matrix*norm_pix_1;
  greyscale_values[i] = gray_image.at<uchar>(pix_1[1],pix_1[0]);

  visualize_pixel_region_edge_object[i].x=pix_1[0];
  visualize_pixel_region_edge_object[i].y=pix_1[1];

 }
 if(i==1)
 {
   // Localise one pixel and its greyscale value
    norm_pix_2 << radius_cylinder*cos((delta_degree_for_pixelregion_around_cylinder[1]-↩
        estimated_orientation)*(arma::datum::pi)/180)/distance_from_camera_to_cylinder ↩
        << radius_cylinder*sin((delta_degree_for_pixelregion_around_cylinder[1]-↩
        estimated_orientation)*(arma::datum::pi)/180)/distance_from_camera_to_cylinder ↩
        << 1;
   pix_2=camera_matrix*norm_pix_2;
  greyscale_values[i] = gray_image.at<uchar>(pix_2[1],pix_2[0]);

 visualize_pixel_region_edge_object[i].x=pix_2[0];
  visualize_pixel_region_edge_object[i].y=pix_2[1];

 }
if(i==2)
 {
   // Localise one pixel and its greyscale value
    norm_pix_3 << radius_cylinder*cos((delta_degree_for_pixelregion_around_cylinder[2]-↩
        estimated_orientation)*(arma::datum::pi)/180)/distance_from_camera_to_cylinder ↩
        << radius_cylinder*sin((delta_degree_for_pixelregion_around_cylinder[2]-↩
        estimated_orientation)*(arma::datum::pi)/180)/distance_from_camera_to_cylinder ↩
        << 1;
   pix_3=camera_matrix*norm_pix_3;
  greyscale_values[i] = gray_image.at<uchar>(pix_3[1],pix_3[0]);

   visualize_pixel_region_edge_object[i].x=pix_3[0];
  visualize_pixel_region_edge_object[i].y=pix_3[1];
 }
 }

 // Visualize the pixels (default:NOT SHOWING)
   cv::circle(cv_ptr->image, visualize_pixel_region_edge_object[0], 2, CV_RGB(0, 255, ↩
       0));
 cv::circle(cv_ptr->image, visualize_pixel_region_edge_object[1], 2, CV_RGB(0, 255, 0))↩
     ;
   cv::circle(cv_ptr->image, visualize_pixel_region_edge_object[2], 2, CV_RGB(0, 255, ↩
       0));


   // Check the greyscale values
if(abs(greyscale_values[0]-greyscale_values[1])>50 || abs(greyscale_values[0]-↩
    greyscale_values[2])>50 ||abs(greyscale_values[2]-greyscale_values[1])>50)
 {
```

```
delta_degree_for_pixels_along_cylinder_edge[1].x= cos((450-estimated_orientation-180)*(↩
    arma::datum::pi)/180)*100;
delta_degree_for_pixels_along_cylinder_edge[1].y= sin((450-estimated_orientation-180)*(↩
    arma::datum::pi)/180)*100;
estimated_orientation=estimated_orientation+180;
  }


// Check if the orientation is stable -- list the last orientations and check if those ↩
    are stable
 if(estimated_orientation>360)
 {
  estimated_orientation=estimated_orientation-360;
 }

   for(int x = 6-1;x>=0;x--){
    list_with_last_degrees[x+1]=list_with_last_degrees[x];

      if(x==0){
    list_with_last_degrees[x]=estimated_orientation;
      }
   }

   int i=0;
   for(int x=0 ;x<6;x++)
   {
     if(abs(list_with_last_degrees[x]-list_with_last_degrees[x+1])<2)
     {
       i++;
     }

   }
   if(i>=5)
   {
     // Median of the last orientations
     true_orientation = GetMedian(list_with_last_degrees,7);
sleep(1);

   }


   // THIS IS THE ESTIMATED ORIENTATION FOR THE OBJECT
   std::cout << "Rotation around Z-axis (degrees): " << true_orientation << std::endl;



// - Visualize the orientation with a green line
   std::vector<Point2f> visualize_orientation(3);
visualize_orientation[0].x=cos((450-true_orientation)*(arma::datum::pi)/180)*100;
visualize_orientation[0].y= sin((450-true_orientation)*(arma::datum::pi)/180)*100;
line( cv_ptr->image, center_point_image_stream[0] , center_point_image_stream[0]+↩
    visualize_orientation[0], Scalar( 0, 255, 0), 2 );





// - Offset values for where the printed degrees should show in the protractor
std::vector<Point2f> offset_text(12);
offset_text[0].x=-3; offset_text[0].y=12;
offset_text[1].x=-15; offset_text[1].y=15;
```

```
offset_text[2].x=-25; offset_text[2].y=12;
offset_text[3].x=-28; offset_text[3].y=6;
offset_text[4].x=-29; offset_text[4].y=0;
offset_text[5].x=-16; offset_text[5].y=-6;
offset_text[6].x=-14; offset_text[6].y=-4;
offset_text[7].x=5; offset_text[7].y=-2;
offset_text[8].x=3; offset_text[8].y=3;
offset_text[9].x=3; offset_text[9].y=4;
offset_text[10].x=8; offset_text[10].y=7;
offset_text[11].x=-6; offset_text[11].y=12;



// - Visulize the protractor
 for( int i = 0; i < 12; i++ )
  {
std::vector<Point2f> visualize_line_in_protractor(1);
visualize_line_in_protractor[0].y= sin((30*i+90)*(arma::datum::pi)/180)*89;
visualize_line_in_protractor[0].x= cos((30*i+90)*(arma::datum::pi)/180)*89;


   cv::circle(cv_ptr->image, center_point_image_stream[0], 89, CV_RGB(0,0,0));
 line( cv_ptr->image, center_point_image_stream[0] , center_point_image_stream[0] +↩
     visualize_line_in_protractor[0], Scalar( 255, 255, 255), 1 );

 int visualize_degrees_in_protractor = 360-30*i;

    if(i==0)
    {
      visualize_degrees_in_protractor = 0;
    }

string text = static_cast<ostringstream*>( &(ostringstream() << ↩
    visualize_degrees_in_protractor) )->str();
int fontFace = FONT_HERSHEY_PLAIN;
double fontScale = 0.9;
int thickness = 2.5;


int baseline=0;
Size textSize = getTextSize(text, fontFace,
                            fontScale, thickness, &baseline);
baseline += thickness;

// center the text
Point textOrg((cv_ptr->image.cols - textSize.width)/2,
              (cv_ptr->image.rows + textSize.height)/2);

putText(cv_ptr->image, text, center_point_image_stream[0] +visualize_line_in_protractor↩
    [0]+offset_text[i], fontFace, fontScale,
        Scalar::all(0), thickness, 8);
  }



// -- Visualize results
cv::imshow(OPENCV_WINDOW, cv_ptr->image);

//-- Publish orientation
     array_orientation_cylinder.data.clear();
for (int i = 0; i < 2; i++)
```

```
{
array_orientation_cylinder.data.push_back(true_orientation);
}
 publish_orientation_cylinder.publish(array_orientation_cylinder);

  }
};

int main(int argc, char** argv)
{

  ros::init(argc, argv, "Orientation_cylinder_node");
  ImageConverter ic;

  ros::spin();
  return 0;
}
```

```cpp
//--------------------------------------------------------------------------------
//------------------------ Node run for estimating orientation of piston -----------
//--------------------------------------------------------------------------------
//---------------------------------------------------------------------------------
//------------------------Authors: Geir Ole Tysse and Martin Morland .---------------
//------------------------------------NTNU 2015---------------------------------------
//---------------------------------------------------------------------------------

#include <ros/ros.h>
#include <image_transport/image_transport.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/image_encodings.h>
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/nonfree/features2d.hpp>
#include <opencv2/features2d/features2d.hpp>
#include <opencv2/core/core.hpp>
#include "opencv2/calib3d/calib3d.hpp"
#include <stdio.h>
#include <iostream>
#include <armadillo>
#include <cmath>
#include "std_msgs/Float64MultiArray.h"
using namespace cv;
using namespace std;
static const std::string OPENCV_WINDOW = "Image window";

float degree_estimated_for_0;
float degree_estimated_for_90;
float degree_estimated_for_180;
float degree_estimated_for_270;
arma::mat camera_matrix(3,3);
float true_orientation;
int number_of_times_approved_H_0=0;
int number_of_times_approved_H_90=0;
int number_of_times_approved_H_180=0;
int number_of_times_approved_H_270=0;
float estimated_orientation;
float list_with_last_degrees[7];
std_msgs::Float64MultiArray array_orientation_piston;

//-- Estimate the median
float GetMedian(float daArray[], int iSize) {
    double* dpSorted = new double[iSize];
    for (int i = 0; i < iSize; ++i) {
        dpSorted[i] = daArray[i];
    }
    for (int i = iSize - 1; i > 0; --i) {
        for (int j = 0; j < i; ++j) {
            if (dpSorted[j] > dpSorted[j+1]) {
                double dTemp = dpSorted[j];
                dpSorted[j] = dpSorted[j+1];
                dpSorted[j+1] = dTemp;
            }
        }
    }
    double dMedian = 0.0;
    if ((iSize % 2) == 0) {
        dMedian = (dpSorted[iSize/2] + dpSorted[(iSize/2) - 1])/2.0;
    } else {
        dMedian = dpSorted[iSize/2];
    }
    delete [] dpSorted;
    return dMedian;
}



class ImageConverter
{
  ros::NodeHandle nh_;
  image_transport::ImageTransport it_;
  image_transport::Subscriber image_sub_;
  image_transport::Publisher image_pub_;
  ros::Publisher publish_orientation_piston;


private:


public:


  ImageConverter()
    : it_(nh_)
  {
    //-- Subscrive to input video feed
    image_sub_ = it_.subscribe("/usb_cam/image_raw", 1,
      &ImageConverter::imageCb, this);
    cv::namedWindow(OPENCV_WINDOW,WINDOW_NORMAL);

    //-- Publish orientation of the piston
   publish_orientation_piston= nh_.advertise<std_msgs::Float64MultiArray>("↩
       piston_orientation", 1);



  }

  ~ImageConverter()
  {
    cv::destroyWindow(OPENCV_WINDOW);

  }

  void imageCb(const sensor_msgs::ImageConstPtr& msg)
  {

    cv_bridge::CvImagePtr cv_ptr;
    try
    {
      cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception& e)
    {
      ROS_ERROR("cv_bridge exception: %s", e.what());
      return;
    }

//-- Get the live image stream and the training images
Mat img_show;
Mat img_object = imread("/home/jorge/catkin_ws/piston_0.jpg");
Mat img_object_90 = imread("/home/jorge/catkin_ws/piston_90.jpg");
Mat img_object_180 = imread("/home/jorge/catkin_ws/piston_180.jpg");
Mat img_object_270 = imread("/home/jorge/catkin_ws/piston_270.jpg");
Mat img_scene = cv_ptr->image;

//-- Get gray image from the image stream also, used for reading greyscale values
Mat gray_image;
 cvtColor( img_scene, gray_image, CV_BGR2GRAY );


if( !img_object.data || !img_scene.data )
  { std::cout<< " --(!) Error reading images " << std::endl; }

  //-- Step 1: Detect the keypoints using Sift Detector
  int minHessian = 200;
 SiftFeatureDetector detector( minHessian );
std::vector<KeyPoint> keypoints_object, keypoints_scene,  keypoints_object_90,  ↩
    keypoints_object_180,  keypoints_object_270;

//-- Detection of feature in image stream. Searches within a defined mask
Mat mask = Mat::zeros(img_scene.size(), CV_8U);
Mat roi(mask, cv::Rect(260,50,120,250));
roi = Scalar(255, 255, 255);
detector.detect(img_scene, keypoints_scene, mask);


//-- Detection of feature in training images
detector.detect( img_object, keypoints_object );
detector.detect( img_object_90, keypoints_object_90);
detector.detect( img_object_180, keypoints_object_180 );
detector.detect( img_object_270, keypoints_object_270 );




  //-- Step 2: Calculate descriptors
  SiftDescriptorExtractor extractor;

  Mat descriptors_object, descriptors_scene, descriptors_object_90, ↩
      descriptors_object_180, descriptors_object_270;

  extractor.compute( img_object, keypoints_object, descriptors_object );
  extractor.compute( img_object_90, keypoints_object_90, descriptors_object_90);
  extractor.compute(  img_object_180, keypoints_object_180 , descriptors_object_180 );
  extractor.compute(img_object_270, keypoints_object_270, descriptors_object_270 );
  extractor.compute( img_scene, keypoints_scene, descriptors_scene );



  //-- Step 3: Matching descriptor vectors using FLANN matcher
 FlannBasedMatcher matcher;

  std::vector< DMatch > matches,matches_90 ,matches_180, matches_270;
  matcher.match( descriptors_object, descriptors_scene, matches );
  matcher.match( descriptors_object_90, descriptors_scene, matches_90 );
  matcher.match( descriptors_object_180, descriptors_scene, matches_180 );
  matcher.match( descriptors_object_270, descriptors_scene, matches_270 );
```

```
std::vector< DMatch > good_matches, good_matches_90, good_matches_180, good_matches_270↩
    ;

   //-- Matching descriptor vectors in image stream and training image 0 degree
  Mat  img_matches_0;
  drawMatches( img_object, keypoints_object, img_scene, keypoints_scene,
               good_matches,  img_matches_0, Scalar::all(-1), Scalar::all(-1),
               vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

  //-- Matching descriptor vectors in image stream and training image 90 degree
 Mat  img_matches_90;
  drawMatches( img_object_90, keypoints_object_90, img_scene, keypoints_scene,
               good_matches_90,  img_matches_90, Scalar::all(-1), Scalar::all(-1),
               vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

     //-- Matching descriptor vectors in image stream and training image 180 degree
    Mat  img_matches_180;
  drawMatches( img_object_180, keypoints_object_180, img_scene, keypoints_scene,
               good_matches_180,  img_matches_180, Scalar::all(-1), Scalar::all(-1),
               vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );

   //-- Matching descriptor vectors in image stream and training image 270 degree
      Mat  img_matches_270;
  drawMatches( img_object_270, keypoints_object_270, img_scene, keypoints_scene,
               good_matches_270,  img_matches_270, Scalar::all(-1), Scalar::all(-1),
               vector<char>(), DrawMatchesFlags::NOT_DRAW_SINGLE_POINTS );



   //-- Quick calculation of max and min distances between keypoints in 0 degrees ↩
       training image and image stream
    double max_dist = 0; double min_dist = 70;

  for( int i = 0; i < descriptors_object.rows; i++ )
  { double dist = matches[i].distance;
    if( dist < min_dist ) min_dist = dist;
    if( dist > max_dist ) max_dist = dist;
  }



  for( int i = 0; i < descriptors_object.rows; i++ )
  { if( matches[i].distance < 3*min_dist )
     { good_matches.push_back( matches[i]); }
  }


  //-- Quick calculation of max and min distances between keypoints in 90 degrees ↩
      training image and image stream
    max_dist = 0;
    min_dist = 70;

  for( int i = 0; i < descriptors_object_90.rows; i++ )
  { double dist = matches_90[i].distance;
    if( dist < min_dist ) min_dist = dist;
    if( dist > max_dist ) max_dist = dist;
  }
  for( int i = 0; i < descriptors_object_90.rows; i++ )
  { if( matches_90[i].distance < 3*min_dist )
     { good_matches_90.push_back( matches_90[i]); }
  }

  //-- Quick calculation of max and min distances between keypoints in 180 degrees ↩
      training image and image stream
    max_dist = 0;
    min_dist = 70;

  for( int i = 0; i < descriptors_object_180.rows; i++ )
  { double dist = matches_180[i].distance;
    if( dist < min_dist ) min_dist = dist;
    if( dist > max_dist ) max_dist = dist;
  }
  for( int i = 0; i < descriptors_object_180.rows; i++ )
  { if( matches_180[i].distance < 3*min_dist )
     { good_matches_180.push_back( matches_180[i]); }
  }

   //-- Quick calculation of max and min distances between keypoints in 270 degrees ↩
       training image and image stream
    max_dist = 0;
    min_dist = 70;

  for( int i = 0; i < descriptors_object_270.rows; i++ )
  { double dist = matches_270[i].distance;
    if( dist < min_dist ) min_dist = dist;
    if( dist > max_dist ) max_dist = dist;
  }
  for( int i = 0; i < descriptors_object_270.rows; i++ )
  { if( matches_270[i].distance < 3*min_dist )
     { good_matches_270.push_back( matches_270[i]); }
  }




  //-- Localize the object
  std::vector<Point2f> obj , obj_90, obj_180 ,obj_270;
  std::vector<Point2f> scene, scene_90 , scene_180 ,scene_270;


  //TRAINING IMAGE 0 DEGREES
  //------------------------------
  //------------------------------


   //-- Get the keypoints from the good matches 0 degree
  for( int i = 0; i < good_matches.size(); i++ )
  {

    obj.push_back( keypoints_object[ good_matches[i].queryIdx ].pt );
    scene.push_back( keypoints_scene[ good_matches[i].trainIdx ].pt );
  }

  std::vector<Point2f> obj_corners(4);
   if (good_matches.size() > 3)
 {
   // -- Estimate the homography between training image 0 degrees and image stream
  cv::Mat H_0 = findHomography( obj, scene, CV_RANSAC );

  //-- Get the corners from the training image 0 degrees ( the object to be "detected" ↩
      )
  obj_corners[0] = cvPoint(0,0); obj_corners[1] = cvPoint( img_object.cols, 0 );
  obj_corners[2] = cvPoint( img_object.cols, img_object.rows ); obj_corners[3] = ↩
      cvPoint( 0, img_object.rows );
  std::vector<Point2f> scene_corners(4);

  perspectiveTransform( obj_corners, scene_corners, H_0);

  //-- Draw lines between the corners (the mapped object in the scene - training image ↩
      0 degrees)
  line(  img_matches_0, scene_corners[0] + Point2f( img_object.cols, 0), scene_corners↩
      [1] + Point2f( img_object.cols, 0), Scalar(0, 255, 0), 4 );
 line(  img_matches_0, scene_corners[1] + Point2f( img_object.cols, 0), scene_corners↩
     [2] + Point2f( img_object.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_0, scene_corners[2] + Point2f( img_object.cols, 0), scene_corners↩
      [3] + Point2f( img_object.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_0, scene_corners[3] + Point2f( img_object.cols, 0), scene_corners↩
      [0] + Point2f( img_object.cols, 0), Scalar( 0, 255, 0), 4 );




//-- Check if it is an affine transformation (training image 0)
if( H_0.at<double>(2,0) < 0.0001 &&  H_0.at<double>(2,1) < 0.0001)
{
  // check if it is a stable homography (training image 0)
 number_of_times_approved_H_0++;

 // calculate the orientation of the object (training image 0)
float hypotenuse_0 = sqrt((scene_corners[2].x-scene_corners[1].x)*(scene_corners[2].x-↩
    scene_corners[1].x)+(scene_corners[2].y-scene_corners[1].y)*(scene_corners[2].y-↩
    scene_corners[1].y));
float  cathetus_0  = scene_corners[2].y-scene_corners[1].y;
degree_estimated_for_0 = acos( cathetus_0/hypotenuse_0)*180/(arma::datum::pi);


if((scene_corners[2].x-scene_corners[1].x) < 0)
{
  degree_estimated_for_0=360-degree_estimated_for_0;
}
}
else
{
  //If it is not an stable homography (training image 0)
  number_of_times_approved_H_0=0;
}
 }




   //TRAINING IMAGE 90 DEGREES
  //---------------
  //---------------


 if (good_matches_90.size() > 3)
 {
 //-- Get the keypoints from the good matches 90 degree
  for( int i = 0; i < good_matches_90.size(); i++ )
  {

    obj_90.push_back( keypoints_object_90[ good_matches_90[i].queryIdx ].pt );
    scene_90.push_back( keypoints_scene[ good_matches_90[i].trainIdx ].pt );
  }

  // -- Estimate the homography between training image 90 degrees and image stream
  cv::Mat H_90 = findHomography( obj_90, scene_90, CV_RANSAC );

    //-- Get the corners from the training image 90 degrees ( the object to be "↩
        detected" )
  std::vector<Point2f> obj_corners_90(4);
  obj_corners_90[0] = cvPoint(0,0); obj_corners_90[1] = cvPoint( img_object_90.cols, 0 ↩
      );
  obj_corners_90[2] = cvPoint( img_object_90.cols, img_object_90.rows ); obj_corners_90↩
      [3] = cvPoint( 0, img_object_90.rows );
    std::vector<Point2f> scene_corners_90(4);

      //-- Draw lines between the corners (the mapped object in the scene - training ↩
          image 90 degrees)
  perspectiveTransform( obj_corners_90, scene_corners_90, H_90);
  line(  img_matches_90, scene_corners_90[0] + Point2f( img_object_90.cols, 0), ↩
      scene_corners_90[1] + Point2f( img_object_90.cols, 0), Scalar(0, 255, 0), 4 );
  line(  img_matches_90, scene_corners_90[1] + Point2f( img_object_90.cols, 0), ↩
      scene_corners_90[2] + Point2f( img_object_90.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_90, scene_corners_90[2] + Point2f( img_object_90.cols, 0), ↩
      scene_corners_90[3] + Point2f( img_object_90.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_90, scene_corners_90[3] + Point2f( img_object_90.cols, 0), ↩
      scene_corners_90[0] + Point2f( img_object_90.cols, 0), Scalar( 0, 255, 0), 4 );

    //-- Check if it is an affine transformation (training image 90)
if( H_90.at<double>(2,0) < 0.0001 &&  H_90.at<double>(2,1) < 0.0001)
{
     // check if it is a stable homography (training image 90)
number_of_times_approved_H_90++;


     // calculate the orientation of the object (training image 0)
float hypotenuse_90 = sqrt((scene_corners_90[1].x-scene_corners_90[0].x)*(↩
    scene_corners_90[1].x-scene_corners_90[0].x)+(scene_corners_90[1].y-↩
    scene_corners_90[0].y)*(scene_corners_90[1].y-scene_corners_90[0].y));
float cathetus_90 = scene_corners_90[1].x-scene_corners_90[0].x;
degree_estimated_for_90 = acos(cathetus_90/hypotenuse_90)*180/(arma::datum::pi);


if(scene_corners_90[1].y > scene_corners_90[0].y)
{
 degree_estimated_for_90= 90 - degree_estimated_for_90;
}
else
{
  degree_estimated_for_90= 90 + degree_estimated_for_90;
}

 }
 else
 {
   //If it is not an stable homography (training image 90)
   int number_of_times_approved_H_90=0;
 }
 }

     //TRAINING IMAGE 180 DEGREES
  //-----------------
  //————————————


 if (good_matches_180.size() > 3)
 {
//−− Get the keypoints from the good matches 180 degree
  for( int i = 0; i < good_matches_180.size(); i++ )
  {

    obj_180.push_back( keypoints_object_180[ good_matches_180[i].queryIdx ].pt );
    scene_180.push_back( keypoints_scene[ good_matches_180[i].trainIdx ].pt );
  }


// −− Estimate the homography between training image 180 degrees and image stream
  cv::Mat H_180 = findHomography( obj_180, scene_180, CV_RANSAC );

   //−− Get the corners from the training image 180 degrees ( the object to be "↩
        detected" )
  std::vector<Point2f> obj_corners_180(4);
  obj_corners_180[0] = cvPoint(0,0); obj_corners_180[1] = cvPoint( img_object_180.cols,↩
       0 );
  obj_corners_180[2] = cvPoint( img_object_180.cols, img_object_180.rows ); ↩
      obj_corners_180[3] = cvPoint( 0, img_object_180.rows );
  std::vector<Point2f> scene_corners_180(4);

  perspectiveTransform( obj_corners_180, scene_corners_180, H_180);
   //−− Draw lines between the corners (the mapped object in the scene − training image↩
       180 degrees)
  line(  img_matches_180, scene_corners_180[0] + Point2f( img_object_180.cols, 0), ↩
      scene_corners_180[1] + Point2f( img_object_180.cols, 0), Scalar(0, 255, 0), 4 );
  line(  img_matches_180, scene_corners_180[1] + Point2f( img_object_180.cols, 0), ↩
      scene_corners_180[2] + Point2f( img_object_180.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_180, scene_corners_180[2] + Point2f( img_object_180.cols, 0), ↩
      scene_corners_180[3] + Point2f( img_object_180.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_180, scene_corners_180[3] + Point2f( img_object_180.cols, 0), ↩
      scene_corners_180[0] + Point2f( img_object_180.cols, 0), Scalar( 0, 255, 0), 4 );

    //−− Check if it is an affine transformation (training image 0)
if( H_180.at<double>(2,0) < 0.0001 &&  H_180.at<double>(2,1) < 0.0001)
{
      // check if it is a stable homography (training image 0)
  number_of_times_approved_H_180++;

   // calculate the orientation of the object (training image 0)
  float  hypotenuse_180 = sqrt((scene_corners_180[2].x−scene_corners_180[1].x)*(↩
      scene_corners_180[2].x−scene_corners_180[1].x)+(scene_corners_180[2].y−↩
      scene_corners_180[1].y)*(scene_corners_180[2].y−scene_corners_180[1].y));
  float cathetus_180 = scene_corners_180[2].y−scene_corners_180[1].y;
degree_estimated_for_180 = acos(cathetus_180/hypotenuse_180)*180/(arma::datum::pi);

if(scene_corners_180[2].x < scene_corners_180[1].x)
{
 degree_estimated_for_180 =180−degree_estimated_for_180;
}
else
{
  degree_estimated_for_180 = 180 +degree_estimated_for_180;
}
 }
 else
 {
       //If it is not an stable homography (training image 0)
  number_of_times_approved_H_180=0;
 }
 }

     //TRAINING IMAGE 270 DEGREES
  //————————————
  //————————————


 if (good_matches_270.size() > 3)
 {
       //-- Get the keypoints from the good matches 270 degree
  for( int i = 0; i < good_matches_270.size(); i++ )
  {

    obj_270.push_back( keypoints_object_270[ good_matches_270[i].queryIdx ].pt );
    scene_270.push_back( keypoints_scene[ good_matches_270[i].trainIdx ].pt );
  }

  // -- Estimate the homography between training image 270 degrees and image stream
  cv::Mat H_270 = findHomography( obj_270, scene_270, CV_RANSAC );

  //-- Get the corners from the training image 270 degrees ( the object to be "detected↩
      " )
  std::vector<Point2f> obj_corners_270(4);
  obj_corners_270[0] = cvPoint(0,0); obj_corners_270[1] = cvPoint( img_object_270.cols,↩
       0 );
  obj_corners_270[2] = cvPoint( img_object_270.cols, img_object_270.rows ); ↩
      obj_corners_270[3] = cvPoint( 0, img_object_270.rows );
  std::vector<Point2f> scene_corners_270(4);
  perspectiveTransform( obj_corners_270, scene_corners_270, H_270);

      //-- Draw lines between the corners (the mapped object in the scene - training ↩
          image 0 degrees)
  line(  img_matches_270, scene_corners_270[0] + Point2f( img_object_270.cols, 0), ↩
      scene_corners_270[1] + Point2f( img_object_270.cols, 0), Scalar(0, 255, 0), 4 );
  line(  img_matches_270, scene_corners_270[1] + Point2f( img_object_270.cols, 0), ↩
      scene_corners_270[2] + Point2f( img_object_270.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_270, scene_corners_270[2] + Point2f( img_object_270.cols, 0), ↩
      scene_corners_270[3] + Point2f( img_object_270.cols, 0), Scalar( 0, 255, 0), 4 );
  line(  img_matches_270, scene_corners_270[3] + Point2f( img_object_270.cols, 0), ↩
      scene_corners_270[0] + Point2f( img_object_270.cols, 0), Scalar( 0, 255, 0), 4 );


  //-- Check if it is an affine transformation (training image 270)
if( H_270.at<double>(2,0) < 0.0001 &&  H_270.at<double>(2,1) < 0.0001)
{
    // check if it is a stable homography (training image 270)
number_of_times_approved_H_270++;

   // calculate the orientation of the object (training image 270)
float hypotenuse_270 = sqrt((scene_corners_270[1].x-scene_corners_270[0].x)*(↩
    scene_corners_270[1].x-scene_corners_270[0].x)+(scene_corners_270[1].y-↩
    scene_corners_270[0].y)*(scene_corners_270[1].y-scene_corners_270[0].y));
float cathetus_270 = scene_corners_270[1].y-scene_corners_270[0].y;
degree_estimated_for_270 = acos(cathetus_270/hypotenuse_270)*180/(arma::datum::pi)+180;

 }
 else
 {
      //If it is not an stable homography (training image 270)
  number_of_times_approved_H_270=0;
 }

 }


 // CHECK WICH OF THE ESTIMATED "degree_estimated_for_xx" THAT VOTE FOR THE SAME VALUE/↩
     DEGREES
 // ––– –––––––
 //–––––––––––


// Votes for training image 0 having the correct orientation
  int g=0;
  if( (degree_estimated_for_0 < 45 || degree_estimated_for_0 >315)&& ↩
      number_of_times_approved_H_0>4)
  {
    g++;
    if( (degree_estimated_for_90 < 45 || degree_estimated_for_90 >315) && ↩
        number_of_times_approved_H_90>4)
    {
      g++;
    }
    if( (degree_estimated_for_180 < 45 || degree_estimated_for_180 >315) && ↩
        number_of_times_approved_H_180>4)
    {
      g++;
    }
    if( (degree_estimated_for_270 < 45 || degree_estimated_for_270 >270) && ↩
        number_of_times_approved_H_270 >4)
    {
      g++;
    }
  }



  // Votes for training image 90 having the correct orientation
  int g_90=0;
  if( degree_estimated_for_90 < 135 && degree_estimated_for_90 > 45 && ↩
      number_of_times_approved_H_90>4)
  {
    g_90++;
    if( degree_estimated_for_0 < 135 && degree_estimated_for_0 > 45 && ↩
        number_of_times_approved_H_0>4)
    {
      g_90++;
    }
    if( degree_estimated_for_180 < 135 && degree_estimated_for_180 >45 && ↩
        number_of_times_approved_H_180>4)
    {
      g_90++;
    }
    if( degree_estimated_for_270 < 135 && degree_estimated_for_270 >45 && ↩
        number_of_times_approved_H_270 >4)
    {
      g_90++;
    }
  }


  // Votes for training image 180 having the correct orientation
  int g_180=0;
  if( degree_estimated_for_180 < 225 && degree_estimated_for_180 > 135 && ↩
      number_of_times_approved_H_180>4)
  {
    g_180++;
    if( degree_estimated_for_0 < 225 && degree_estimated_for_0 > 135 && ↩
        number_of_times_approved_H_0>4 )
    {
      g_180++;
    }
    if( degree_estimated_for_90 < 225 && degree_estimated_for_90 >135&& ↩
        number_of_times_approved_H_90>4)
    {
      g_180++;
    }
    if( degree_estimated_for_270 < 225 && degree_estimated_for_270 >135 && ↩
        number_of_times_approved_H_270 >4)
    {
      g_180++;
    }
  }


     // Votes for training image 270 having the correct orientation
  int g_270=0;
  if( degree_estimated_for_270 < 315 && degree_estimated_for_270 > 225 && ↩
      number_of_times_approved_H_270>4)
  {
    g_270++;
    if( degree_estimated_for_0 < 315 && degree_estimated_for_0 > 225&& ↩
        number_of_times_approved_H_0>4 )
    {
      g_270++;
    }
    if( degree_estimated_for_90 < 315 && degree_estimated_for_90 >225 && ↩
        number_of_times_approved_H_90>4)
    {
      g_270++;
    }
    if( degree_estimated_for_180 < 315 && degree_estimated_for_180 >225&& ↩
        number_of_times_approved_H_180>4)
    {
      g_270++;
    }
  }


  // CHECK WHICH OF THE "degree_estimated_for_xx" THAT HAS HIGHEST VOTES
  if(g > g_90 && g > g_180  && g > g_270 )
  {
   estimated_orientation= degree_estimated_for_0;
  }

  if(g_90 > g && g_90 > g_180  && g_90 > g_270 )
  {
   estimated_orientation= degree_estimated_for_90;
  }

  if(g_180 > g && g_180 > g_90  && g_180 > g_270 )
  {
   estimated_orientation= degree_estimated_for_180;
  }

  if(g_270 > g && g_270 > g_90  && g_270 > g_180 )
  {
   estimated_orientation= degree_estimated_for_270;
  }


// --CHECK PIXELREGIONS, IS THE ORIENTATION TRUE OR 180 DEGREES PHASE SHIFTED ?
// -----------
//-----------


std::vector<Point2f> center_point_image_stream(1);
center_point_image_stream[0]=cvPoint((cv_ptr->image.cols)/2,(cv_ptr->image.rows)/2);


// Delta degrees for localizing the pixel region
 std::vector<Point2f> delta_degree_for_pixels_along_cylinder_edge(5);
   delta_degree_for_pixels_along_cylinder_edge[0].x= cos((450-estimated_orientation)*(↩
       arma::datum::pi)/180)*100;
delta_degree_for_pixels_along_cylinder_edge[0].y= sin((450-estimated_orientation)*(arma↩
    ::datum::pi)/180)*100;
 delta_degree_for_pixels_along_cylinder_edge[1].x= cos((270-estimated_orientation)*(↩
     arma::datum::pi)/180)*100;
delta_degree_for_pixels_along_cylinder_edge[1].y= sin((270-estimated_orientation)*(arma↩
    ::datum::pi)/180)*100;
   delta_degree_for_pixels_along_cylinder_edge[2].x= cos((431-estimated_orientation)*(↩
       arma::datum::pi)/180)*100;
delta_degree_for_pixels_along_cylinder_edge[2].y= sin((431-estimated_orientation)*(arma↩
    ::datum::pi)/180)*100;
   delta_degree_for_pixels_along_cylinder_edge[3].x= cos((427-estimated_orientation)*(↩
       arma::datum::pi)/180)*100;
delta_degree_for_pixels_along_cylinder_edge[3].y= sin((427-estimated_orientation)*(arma↩
    ::datum::pi)/180)*100;


   uint16_t greyscale_values[3];
   float radius_piston =0.115;
   float distance_from_camera_to_piston=0.205;
   float delta_degree_for_pixelregion_around_piston [3] = {423,427,431};
   std::vector<Point2f> visualize_pixel_region_edge_object(3);

   arma::colvec norm_pix_1, norm_pix_2, norm_pix_3, pix_1, pix_2, pix_3;

   // Camera matrix
   arma::mat camera_matrix(3,3);
   camera_matrix << 540.927271 << 0 << 320 <<arma::endr
   << 0 << 540.264582 << 240 <<arma::endr
   << 0 << 0 << 1 <<arma::endr;



 for(int i=0;i<3;i++)
 {

 if(i==0)
 {
```

```
// Localise one pixel and its greyscale value
  norm_pix_1 << radius_piston*cos((delta_degree_for_pixelregion_around_piston[0]-↩
      estimated_orientation)*(arma::datum::pi)/180)/distance_from_camera_to_piston << ↩
      radius_piston*sin((delta_degree_for_pixelregion_around_piston[0]-↩
      estimated_orientation)*(arma::datum::pi)/180)/distance_from_camera_to_piston << ↩
      1;
   pix_1=camera_matrix*norm_pix_1;
  greyscale_values[i] = gray_image.at<uchar>(pix_1[1],pix_1[0]);

  visualize_pixel_region_edge_object[i].x=pix_1[0];
  visualize_pixel_region_edge_object[i].y=pix_1[1];

 }
 if(i==1)
 {
   // Localise one pixel and its greyscale value
    norm_pix_2 << radius_piston*cos((delta_degree_for_pixelregion_around_piston[1]-↩
        estimated_orientation)*(arma::datum::pi)/180)/distance_from_camera_to_piston <<↩
         radius_piston*sin((delta_degree_for_pixelregion_around_piston[1]-↩
        estimated_orientation)*(arma::datum::pi)/180)/distance_from_camera_to_piston <<↩
         1;
   pix_2=camera_matrix*norm_pix_2;
  greyscale_values[i] = gray_image.at<uchar>(pix_2[1],pix_2[0]);

 visualize_pixel_region_edge_object[i].x=pix_2[0];
  visualize_pixel_region_edge_object[i].y=pix_2[1];

 }
if(i==2)
 {
   // Localise one pixel and its greyscale value
    norm_pix_3 << radius_piston*cos((delta_degree_for_pixelregion_around_piston[2]-↩
        estimated_orientation)*(arma::datum::pi)/180)/distance_from_camera_to_piston <<↩
         radius_piston*sin((delta_degree_for_pixelregion_around_piston[2]-↩
        estimated_orientation)*(arma::datum::pi)/180)/distance_from_camera_to_piston <<↩
         1;
   pix_3=camera_matrix*norm_pix_3;
  greyscale_values[i] = gray_image.at<uchar>(pix_3[1],pix_3[0]);

   visualize_pixel_region_edge_object[i].x=pix_3[0];
  visualize_pixel_region_edge_object[i].y=pix_3[1];
 }
 }

 // Visualize the pixels (default:NOT SHOWING)
   cv::circle(cv_ptr->image, visualize_pixel_region_edge_object[0], 2, CV_RGB(0, 255, ↩
       0));
 cv::circle(cv_ptr->image, visualize_pixel_region_edge_object[1], 2, CV_RGB(0, 255, 0))↩
     ;
   cv::circle(cv_ptr->image, visualize_pixel_region_edge_object[2], 2, CV_RGB(0, 255, ↩
       0));


   // Check the greyscale values
if(abs(greyscale_values[0]-greyscale_values[1])>50 || abs(greyscale_values[0]-↩
    greyscale_values[2])>50 ||abs(greyscale_values[2]-greyscale_values[1])>50)
 {
delta_degree_for_pixels_along_cylinder_edge[1].x= cos((450-estimated_orientation-180)*(↩
    arma::datum::pi)/180)*100;
delta_degree_for_pixels_along_cylinder_edge[1].y= sin((450-estimated_orientation-180)*(↩
    arma::datum::pi)/180)*100;
```

```
estimated_orientation=estimated_orientation+180;
  }


// Check if the orientation is stable -- list the last orientations and check if those ↩
    are stable
 if(estimated_orientation>360)
 {
  estimated_orientation=estimated_orientation-360;
 }

   for(int x = 6-1;x>=0;x--){
    list_with_last_degrees[x+1]=list_with_last_degrees[x];

      if(x==0){
    list_with_last_degrees[x]=estimated_orientation;
      }
   }

   int i=0;
   for(int x=0 ;x<6;x++)
   {
     if(abs(list_with_last_degrees[x]-list_with_last_degrees[x+1])<2)
     {
       i++;
     }

   }
   if(i>=5)
   {
     // Median of the last orientations
     true_orientation = GetMedian(list_with_last_degrees,7);
sleep(1);

   }


   // THIS IS THE ESTIMATED ORIENTATION FOR THE OBJECT
   std::cout << "Rotation around Z-axis (degrees): " << true_orientation << std::endl;



// - Visualize the orientation with a green line
   std::vector<Point2f> visualize_orientation(3);
visualize_orientation[0].x=cos((450-true_orientation)*(arma::datum::pi)/180)*100;
visualize_orientation[0].y= sin((450-true_orientation)*(arma::datum::pi)/180)*100;
line( cv_ptr->image, center_point_image_stream[0] , center_point_image_stream[0]+↩
    visualize_orientation[0], Scalar( 0, 255, 0), 2 );





// - Offset values for where the printed degrees should show in the protractor
std::vector<Point2f> offset_text(12);
offset_text[0].x=-3; offset_text[0].y=12;
offset_text[1].x=-15; offset_text[1].y=15;
offset_text[2].x=-25; offset_text[2].y=12;
offset_text[3].x=-28; offset_text[3].y=6;
offset_text[4].x=-29; offset_text[4].y=0;
offset_text[5].x=-16; offset_text[5].y=-6;
```

```
offset_text[6].x=-14; offset_text[6].y=-4;
offset_text[7].x=5; offset_text[7].y=-2;
offset_text[8].x=3; offset_text[8].y=3;
offset_text[9].x=3; offset_text[9].y=4;
offset_text[10].x=8; offset_text[10].y=7;
offset_text[11].x=-6; offset_text[11].y=12;



// - Visulize the protractor
 for( int i = 0; i < 12; i++ )
  {
std::vector<Point2f> visualize_line_in_protractor(1);
visualize_line_in_protractor[0].y= sin((30*i+90)*(arma::datum::pi)/180)*89;
visualize_line_in_protractor[0].x= cos((30*i+90)*(arma::datum::pi)/180)*89;


   cv::circle(cv_ptr->image, center_point_image_stream[0], 21, CV_RGB(0,0,0));
 line( cv_ptr->image, center_point_image_stream[0] , center_point_image_stream[0] +↩
     visualize_line_in_protractor[0], Scalar( 255, 255, 255), 1 );

 int visualize_degrees_in_protractor = 360-30*i;

    if(i==0)
    {
      visualize_degrees_in_protractor = 0;
    }

string text = static_cast<ostringstream*>( &(ostringstream() << ↩
    visualize_degrees_in_protractor) )->str();
int fontFace = FONT_HERSHEY_PLAIN;
double fontScale = 0.9;
int thickness = 2.5;


int baseline=0;
Size textSize = getTextSize(text, fontFace,
                            fontScale, thickness, &baseline);
baseline += thickness;

// center the text
Point textOrg((cv_ptr->image.cols - textSize.width)/2,
              (cv_ptr->image.rows + textSize.height)/2);

putText(cv_ptr->image, text, center_point_image_stream[0] +visualize_line_in_protractor↩
    [0]+offset_text[i], fontFace, fontScale,
        Scalar::all(0), thickness, 8);
  }



// -- Visualize results
cv::imshow(OPENCV_WINDOW, cv_ptr->image);


//-- Publish orientation
     array_orientation_piston.data.clear();
for (int i = 0; i < 2; i++)
{
//-- Transformation ready for publishing
array_orientation_piston.data.push_back(true_orientation);
}
 publish_orientation_piston.publish(array_orientation_piston);


  }
};

int main(int argc, char** argv)
{

  ros::init(argc, argv, "Orientation_piston_node");
  ImageConverter ic;

  ros::spin();
  return 0;
}
```

```
//------------------------------------------------------------------------------------------
//-------------------- Node run for estimating kinematics robots------------------
//------------------------------------------------------------------------------------------
//------------------------------------------------------------------------------------------
//--------------------Authors: Geir Ole Tysse and Martin Morland .------------------
//-----------------------------------NTNU 2015-----------------------------------
//------------------------------------------------------------------------------------------

#include <ros/ros.h>
#include <pcl_ros/point_cloud.h>
#include <ros/ros.h>
#include <iostream>
#include <limits>
#include <fstream>
#include <vector>
#include <armadillo>
#include <sstream>
#include "std_msgs/String.h"
#include "std_msgs/Float64MultiArray.h"
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <cmath>
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/Int32MultiArray.h"
#include <armadillo>

using namespace std;
using namespace arma;


float theta_cylinder=0;
float theta_piston=0;

colvec coordinates_piston;
colvec coordinates_cylinder;
  mat Transformation_table_cylinder(4,4);
  mat Transformation_table_piston(4,4);
mat Transformation_world_table(4,4);
mat Transformation_world_cylinder(4,4);
mat Transformation_world_piston(4,4);
mat Transformation_gripper_end_effector(4,4);
mat Transformation_parts_gripper(4,4);
mat Transformation_parts_end_effector(4,4);
mat Transformation_world_end_effector_piston(4,4);
mat Transformation_world_end_effector_cylinder(4,4);

mat Transformation_table_cylinder_none_rotation(4,4);
mat Transformation_table_piston_none_rotation(4,4);
mat Transformation_world_cylinder_none_rotation(4,4);
mat Transformation_world_piston_none_rotation(4,4);
mat  Transformation_table_webcam_piston(4,4);
mat Transformation_table_webcam_cylinder(4,4);

std_msgs::Float64MultiArray Transformation_cylinder;
std_msgs::Float64MultiArray Transformation_piston;
std_msgs::Float64MultiArray Transformation_cylinder_none_rotation;
std_msgs::Float64MultiArray Transformation_piston_none_rotation;
std_msgs::Float64MultiArray array_transformation_table_webcam_piston;
std_msgs::Float64MultiArray array_transformation_table_webcam_cylinder;
```

```




//-- The class
//--------------
class SubscribeAndPublish
{
public:
  SubscribeAndPublish()
  {
   //-- Topics beeing subscribed and published to
 sub_coord_cylinder = n.subscribe("cylinder_coordinates", 1, &SubscribeAndPublish::↩
     CallbackcoordCylinder, this);
 sub_coord_piston = n.subscribe("piston_coordinates", 1, &SubscribeAndPublish::↩
     CallbackcoordPiston, this);
 sub_orientation_piston = n.subscribe("cylinder_orientation", 1, &SubscribeAndPublish::↩
     CallbackorientationCylinder, this);
 sub_orientation_piston = n.subscribe("piston_orientation", 1, &SubscribeAndPublish::↩
     CallbackorientationPiston, this);
 subnh = n.subscribe<std_msgs::Float64MultiArray>("Transformation_from_camera_to_table"↩
     , 1, &SubscribeAndPublish::callbacknh,this);
 publisher_cylinder = n.advertise<std_msgs::Float64MultiArray>("↩
     cylinder_pose_with_respect_to_word", 1);
   publisher_piston = n.advertise<std_msgs::Float64MultiArray>("↩
       piston_pose_with_respect_to_world", 1);
    publisher_cylinder_none_rotation = n.advertise<std_msgs::Float64MultiArray>("↩
        cylinder_pose_none_rotation_with_respect_to_world", 1);
   publisher_piston_none_rotation = n.advertise<std_msgs::Float64MultiArray>("↩
       piston_pose_none_rotation_with_respect_to_world", 1);
  publisher_transformation_table_webcam_piston=n.advertise<std_msgs::Float64MultiArray↩
      >("Transformation_table_webcam_piston", 1);
  publisher_transformation_table_webcam_cylinder=n.advertise<std_msgs::↩
      Float64MultiArray>("Transformation_table_webcam_cylinder", 1);
  }


void CallbackcoordCylinder(const std_msgs::Float64MultiArray::ConstPtr& msg)
{
  coordinates_cylinder << msg->data[0] << msg->data[1] << msg->data[2];

}

void CallbackcoordPiston(const std_msgs::Float64MultiArray::ConstPtr& msg)
{
  coordinates_piston << msg->data[0] << msg->data[1] << msg->data[2];
}

void CallbackorientationCylinder(const std_msgs::Float64MultiArray::ConstPtr& msg)
{
theta_cylinder=msg->data[0];
}

void CallbackorientationPiston(const std_msgs::Float64MultiArray::ConstPtr& msg)
{
theta_piston=msg->data[0];
}

void callbacknh(const std_msgs::Float64MultiArray::ConstPtr& msg)
{
  //-- Tf from table to cylinder pose
```

```
   Transformation_table_cylinder << cos(theta_cylinder/180*arma::datum::pi) << -sin(↩
       theta_cylinder/180*arma::datum::pi) << 0 << coordinates_cylinder[0] << arma::endr
      << sin(theta_cylinder/180*arma::datum::pi) << cos(theta_cylinder/180*arma::datum::↩
          pi)<< 0 << coordinates_cylinder[1] << arma::endr
      << 0 << 0 << 1 << coordinates_cylinder[2] << arma::endr
      << 0 << 0 << 0 << 1 << arma::endr;

        //-- Tf from table to piston pose
       Transformation_table_piston << cos(theta_piston/180*arma::datum::pi) << -sin(↩
           theta_piston/180*arma::datum::pi) << 0 << coordinates_piston[0] << arma::endr
      << sin(theta_piston/180*arma::datum::pi) << cos(theta_piston/180*arma::datum::pi)↩
          << 0 << coordinates_piston[1] << arma::endr
      << 0 << 0 << 1 << coordinates_piston[2] << arma::endr
      << 0 << 0 << 0 << 1 << arma::endr;

       //-- Tf from table to cylinder pose (R=I)
       Transformation_table_cylinder_none_rotation << 1 << 0 << 0 << coordinates_cylinder↩
           [0] << arma::endr
      << 0<< 1 << 0 << coordinates_cylinder[1] << arma::endr
      << 0<< 0 << 1 << coordinates_cylinder[2] << arma::endr
      << 0 << 0 << 0 << 1 << arma::endr;

       //-- Tf from table to piston pose (R=I)
       Transformation_table_piston_none_rotation << 1 << 0 << 0 << coordinates_piston[0] ↩
           << arma::endr
      << 0<< 1 << 0 << coordinates_piston[1] << arma::endr
      << 0<< 0 << 1 << coordinates_piston[2] << arma::endr
      << 0 << 0 << 0 << 1 << arma::endr;

       //-- Tf from world to table
     Transformation_world_table << 0 << -1 << 0 << 0.67562 << arma::endr
      << 1 << 0 << 0 << -0.39624 << arma::endr
      << 0<< 0<< 1 << 0.938 << arma::endr
      << 0 << 0 << 0 << 1 << arma::endr;

      //-- Tf from table to webcam (orienation analysis cylinder)
  Transformation_table_webcam_cylinder << -1 << 0 << 0 << coordinates_cylinder[0] << ↩
      arma::endr
      << 0 << 1 << 0 << coordinates_cylinder[1] << arma::endr
      << 0<< 0 << -1 << coordinates_cylinder[2]+0.055 << arma::endr
      << 0 << 0 << 0 << 1 << arma::endr;

      //-- Tf from table to webcam (orienation analysis piston)
  Transformation_table_webcam_piston << -1 << 0 << 0 << coordinates_piston[0] << arma::↩
      endr
      << 0<< 1<< 0 << coordinates_piston[1] << arma::endr
      << 0<< 0<< -1 << coordinates_piston[2]+0.115 << arma::endr
      << 0 << 0 << 0 << 1 << arma::endr;

       //-- Tf from world to piston
 Transformation_world_piston = Transformation_world_table*Transformation_table_piston;
      //-- Tf from world to cylinder
 Transformation_world_cylinder = Transformation_world_table*↩
     Transformation_table_cylinder;
      //-- Tf from world to piston (none orienation of piston)
 Transformation_world_piston_none_rotation = Transformation_world_table*↩
     Transformation_table_piston_none_rotation;
      //-- Tf from world to piston (none orienation of cylinder)
 Transformation_world_cylinder_none_rotation = Transformation_world_table*↩
     Transformation_table_cylinder_none_rotation;

```

```
    //--Publish Tf world to cylinder
      Transformation_cylinder.data.clear();
for (int i = 0; i < 16; i++)
{
Transformation_cylinder.data.push_back(Transformation_world_cylinder[i]);
}
 publisher_cylinder.publish(Transformation_cylinder);

    //--Publish Tf world to piston
  Transformation_piston.data.clear();
for (int i = 0; i < 16; i++)
{
Transformation_piston.data.push_back(Transformation_world_piston[i]);
}

publisher_piston.publish(Transformation_piston);


    //--Publish Tf world to cylinder (none rotation)
      Transformation_cylinder_none_rotation.data.clear();
for (int i = 0; i < 16; i++)
{
Transformation_cylinder_none_rotation.data.push_back(↩
    Transformation_world_cylinder_none_rotation[i]);
}
 publisher_cylinder_none_rotation.publish(Transformation_cylinder_none_rotation);

    //--Publish Tf world to piston (none rotation)
  Transformation_piston_none_rotation.data.clear();
for (int i = 0; i < 16; i++)
{
Transformation_piston_none_rotation.data.push_back(↩
    Transformation_world_piston_none_rotation[i]);
}

publisher_piston_none_rotation.publish(Transformation_piston_none_rotation);


    //--Publish Tf world to webcam (cylinder)
      array_transformation_table_webcam_cylinder.data.clear();
for (int i = 0; i < 16; i++)
{
array_transformation_table_webcam_cylinder.data.push_back(↩
    Transformation_table_webcam_piston[i]);
}
 publisher_transformation_table_webcam_cylinder.publish(↩
     array_transformation_table_webcam_cylinder);

    //--Publish Tf world to webcam (piston)
  array_transformation_table_webcam_piston.data.clear();
for (int i = 0; i < 16; i++)
{
array_transformation_table_webcam_piston.data.push_back(↩
    Transformation_world_piston_none_rotation[i]);
}

publisher_transformation_table_webcam_piston.publish(↩
    array_transformation_table_webcam_piston);

}

```

```
private:
  ros::NodeHandle n;
  ros::Publisher publisher_cylinder;
  ros::Publisher publisher_piston;
  ros::Subscriber sub_coord_cylinder ;
ros::Subscriber sub_coord_piston;
 ros::Subscriber sub_orientation_piston;
 ros::Subscriber subnh ;
 ros::Publisher publisher_cylinder_none_rotation;
  ros::Publisher publisher_piston_none_rotation;
  ros::Publisher publisher_transformation_table_webcam_piston;
  ros::Publisher publisher_transformation_table_webcam_cylinder;


};

int main(int argc, char **argv)
{

  //Initiate ROS. The node name is "Publish_transformations"
  ros::init(argc, argv, "Publish_transformations");


//Create an object of class SubscribeAndPublish that will take care of everything
  SubscribeAndPublish SAPObject;
  ros::spin();

  return 0;
}
```

```
//------------------------------------------------------------------------------
//-------- Node run for publishing the end effector poses for both robots------
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
//--------------------Authors: Geir Ole Tysse and Martin Morland .--------------
//-------------------------------NTNU 2015--------------------------------------
//------------------------------------------------------------------------------

#include <ros/ros.h>
#include <pcl_ros/point_cloud.h>
#include <ros/ros.h>
#include <iostream>
#include <limits>
#include <fstream>
#include <vector>
#include <armadillo>
#include <sstream>
#include "std_msgs/String.h"
#include "std_msgs/Float64MultiArray.h"
#include <stdio.h>
#include <stdlib.h>
#include <string>
#include <cmath>
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/Int32MultiArray.h"
#include <armadillo>

using namespace std;
using namespace arma;

std_msgs::Float64MultiArray Transformation_cylinder;
std_msgs::Float64MultiArray Transformation_piston;
std_msgs::Float64MultiArray array_transformation_world_end_effector_webcam_cylinder;
std_msgs::Float64MultiArray array_transformation_world_end_effector_webcam_piston;
mat Transformation_gripper_end_effector(4,4);
mat Transformation_parts_gripper(4,4);
mat Transformation_end_effector_web_cam(4,4);
mat  Transformation_world_table(4,4);
mat Transformation_table_webcam_piston(4,4);
mat Transformation_table_webcam_cylinder(4,4);
mat Transformation_ag2_world_end_effector_cylinder(4,4);
mat Transformation_world_end_effector_piston(4,4);
mat Transformation_world_cylinder(4,4);
mat Transformation_ag2_world_end_effector_piston(4,4);
mat Transformation_ag1__world_end_effector_cylinder(4,4);
mat Transformation_world_piston(4,4);
mat Transformation_ag1_world_end_effector_piston(4,4);


//-- The class
//-------------
class SubscribeAndPublish
{
public:



  SubscribeAndPublish()
  {
    //-- Tf from gripper to end effector
```

```
Transformation_gripper_end_effector << 1 << 0 << 0 << 0 << arma::endr
     << 0 << 1 << 0 << 0<< arma::endr
     << 0<< 0<< 1 << -0.16<< arma::endr
     << 0 << 0 << 0 << 1 << arma::endr;

      //-- Tf from gripper to 2D camera
Transformation_end_effector_web_cam <<1<< 0<< 0 << 0 << arma::endr
     << 0 << 1 << 0 << -0.028 << arma::endr
     << 0<< 0 << 1 << 0.01<< arma::endr
     << 0 << 0 << 0 << 1 << arma::endr;

      //-- Tf from world to table
  Transformation_world_table << 0 << -1 << 0 << 0.67562 << arma::endr
     << 1 << 0 << 0 << -0.39624 << arma::endr
     << 0<< 0<< 1 << 0.938 << arma::endr
     << 0 << 0 << 0 << 1 << arma::endr;

  //--- Topics beeing subscribed and published to
  subscriber_transformation_table_webcam_piston=n.subscribe("↩
      Transformation_table_webcam_piston", 1,&SubscribeAndPublish::↩
      CallbackWebcam_piston, this);
  subscriber_transformation_table_webcam_cylinder=n.subscribe("↩
      Transformation_table_webcam_cylinder", 1,&SubscribeAndPublish::↩
      CallbackWebcam_cylinder, this);
 sub_coord_cylinder = n.subscribe("cylinder_pose_with_respect_to_world", 1, &↩
     SubscribeAndPublish::CallbackCylinder, this);
 sub_coord_piston = n.subscribe("piston_pose_with_respect_to_world", 1, &↩
     SubscribeAndPublish::CallbackPiston, this);
 publisher_gripper_cylinder = n.advertise<std_msgs::Float64MultiArray>("↩
     End_Effector_pose_cylinder_ag1", 1);
   publisher_gripper_piston = n.advertise<std_msgs::Float64MultiArray>("↩
       End_Effector_pose_piston_ag1", 1);
    publisher_cam_cylinder = n.advertise<std_msgs::Float64MultiArray>("↩
        End_Effector_pose_cylinder_ag2", 1);
   publisher_cam_piston = n.advertise<std_msgs::Float64MultiArray>("↩
       End_Effector_pose_piston_ag2", 1);
  }

 void CallbackWebcam_cylinder(const std_msgs::Float64MultiArray::ConstPtr& msg)
{
  //-- TF webcam to cylinder when doing orientation analysis
Transformation_table_webcam_cylinder << msg->data[0] << msg->data[4] << msg->data[8] <<↩
     msg->data[12] << arma::endr
     << msg->data[1] << msg->data[5] << msg->data[9] << msg->data[13] << arma::endr
     << msg->data[2] << msg->data[6] << msg->data[10] << msg->data[14] << arma::endr
     << msg->data[3] << msg->data[7] << msg->data[11] << msg->data[15] << arma::endr;

     //-- TF world to end-effector when doing orientation analysis
Transformation_ag2_world_end_effector_cylinder=Transformation_world_table*↩
    Transformation_table_webcam_cylinder*inv(Transformation_end_effector_web_cam);

//-- Publish TF world to end-effector when doing orientation analysis
    array_transformation_world_end_effector_webcam_cylinder.data.clear();
for (int i = 0; i < 16; i++)
{

array_transformation_world_end_effector_webcam_cylinder.data.push_back(↩
    Transformation_ag2_world_end_effector_cylinder[i]);
}
 publisher_cam_cylinder.publish(array_transformation_world_end_effector_webcam_cylinder↩
     );
```

```

}


 void CallbackWebcam_piston(const std_msgs::Float64MultiArray::ConstPtr& msg)
{
  //-- TF webcam to piston when doing orientation analysis
Transformation_table_webcam_piston << msg->data[0] << msg->data[4] << msg->data[8] << ↩
    msg->data[12] << arma::endr
     << msg->data[1] << msg->data[5] << msg->data[9] << msg->data[13] << arma::endr
     << msg->data[2] << msg->data[6] << msg->data[10] << msg->data[14] << arma::endr
     << msg->data[3] << msg->data[7] << msg->data[11] << msg->data[15] << arma::endr;

  //-- TF world to end-effector when doing orientation analysis
Transformation_ag2_world_end_effector_piston=Transformation_world_table*↩
    Transformation_table_webcam_piston*inv(Transformation_end_effector_web_cam);

//-- Publish TF world to end-effector when doing orientation analysis
    array_transformation_world_end_effector_webcam_piston.data.clear();
for (int i = 0; i < 16; i++)
{
array_transformation_world_end_effector_webcam_piston.data.push_back(↩
    Transformation_ag2_world_end_effector_piston[i]);
}
 publisher_cam_piston.publish(array_transformation_world_end_effector_webcam_piston);

}


void CallbackCylinder(const std_msgs::Float64MultiArray::ConstPtr& msg)
{
  //-- Tf world to cylinder
Transformation_world_cylinder << msg->data[0] << msg->data[4] << msg->data[8] << msg->↩
    data[12] << arma::endr
     << msg->data[1] << msg->data[5] << msg->data[9] << msg->data[13] << arma::endr
     << msg->data[2] << msg->data[6] << msg->data[10] << msg->data[14] << arma::endr
     << msg->data[3] << msg->data[7] << msg->data[11] << msg->data[15] << arma::endr;

   //-- Tf world to end-effector (cylinder)
Transformation_ag1__world_end_effector_cylinder=Transformation_world_cylinder*↩
    Transformation_parts_gripper*Transformation_gripper_end_effector;

 //-- Publish TF world to end-effector grasping piston
    Transformation_cylinder.data.clear();
for (int i = 0; i < 16; i++)
{
Transformation_cylinder.data.push_back(Transformation_ag1__world_end_effector_cylinder[↩
    i]);
}
 publisher_gripper_cylinder.publish(Transformation_cylinder);

}

void CallbackPiston(const std_msgs::Float64MultiArray::ConstPtr& msg)
{
    //-- Tf world to piston
 Transformation_world_piston<< msg->data[0] << msg->data[4] << msg->data[8] << msg->↩
     data[12] << arma::endr
     << msg->data[1] << msg->data[5] << msg->data[9] << msg->data[13] << arma::endr
     << msg->data[2] << msg->data[6] << msg->data[10] << msg->data[14] << arma::endr
     << msg->data[3] << msg->data[7] << msg->data[11] << msg->data[15] << arma::endr;
```

```
       //-- Tf world to end-effector (piston)
 Transformation_ag1_world_end_effector_piston =Transformation_world_piston*↩
     Transformation_parts_gripper*Transformation_gripper_end_effector;

 //-- Publish TF world to end-effector grasping cylinder
       Transformation_piston.data.clear();
for (int i = 0; i < 16; i++)
{
Transformation_piston.data.push_back(Transformation_ag1_world_end_effector_piston[i]);
}

publisher_gripper_piston.publish(Transformation_piston);

}









private:
  ros::NodeHandle n;
  ros::Publisher publisher_gripper_cylinder;
  ros::Publisher publisher_gripper_piston;
  ros::Subscriber sub_coord_cylinder ;
ros::Subscriber sub_coord_piston;
 ros::Subscriber sub_orientation_piston;
 ros::Subscriber subnh ;
 ros::Publisher publisher_cam_cylinder;
  ros::Publisher publisher_cam_piston;
   ros::Subscriber subscriber_cylinder_none_rotation;
 ros::Subscriber  subscriber_piston_none_rotation;
   ros::Subscriber subscriber_transformation_table_webcam_piston;
 ros::Subscriber  subscriber_transformation_table_webcam_cylinder;


};

int main(int argc, char **argv)
{

  //Initiate ROS. The node name is "Pointcloud"
  ros::init(argc, argv, "Publish_end_effector_poses");

//Create an object of class SubscribeAndPublish that will take care of everything
  SubscribeAndPublish SAPObject;
  ros::spin();

  return 0;
}
```

Bibliography

[1] Clemons, J. SIFT: Scale invariant feature transform by david lowe. Lecture, Opened 02.10.2014 .

[2] Corke, P. (2011). *Robotics, Vision and Control*. Springer-Verlag Berlin Heidelberg, 1st edition.

[3] Dorst, Leo, Fontijne, D. A. M. S. (2007). *Geometric Algebra for Computer Science: An Object-oriented Approach to Geometry*. Morgan Kaufmann Publishers.

[4] Hartley, R. and Zisserman, A. (2004). *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2nd edition.

[5] Hecht, J. (2011). Photonic frontiers: Gesture recognition: Lasers bring gesture recognition to the home. `http://www.laserfocusworld.com/articles/2011/01/lasers-bring-gesture-recognition-to-the-home.html`. [Online; accessed 14-April-2015].

[6] Holz, D., Holzer, S., Rusu, R. B., and Behnke, S. (2012). Real-time plane segmentation using rgb-d cameras. In *RoboCup 2011: Robot Soccer World Cup XV*, pages 306–317. Springer.

[7] Kim, J., Nguyen, H. H., Lee, Y., and Lee, S. (2013). Structured light camera base 3d visual perception and tracking application system with robot grasping task. In *Assembly and Manufacturing (ISAM), 2013 IEEE International Symposium on*, pages 187–192. IEEE.

[8] Ko, D.-I. and Agarwal, G. (2012). Gesture recognition: Enabling natural interactions with electronics.

[9] Kuka (2014). KR C4 compacts. `http://www.kuka-robotics.com/res/sps/94de4d6a-e810-4505-9f90-b2d3865077b6_Spez_KR_C4_compact_en.pdf`. [Online; accessed 12-March-2015].

[10] Lin, C.-Y. and Setiawan, E. (2009). Object orientation recognition based on sift and svm by using stereo camera. In *Robotics and Biomimetics, 2008. ROBIO 2008. IEEE International Conference on*, pages 1371–1376. IEEE.

[11] Lowe, D. (2004). Distinctive image features from scale-invariant keypoints. *International journal of computer vision 60, no. 2*.

[12] MoveIt! MoveIt! home page. `http://moveit.ros.org/`. [Online; accessed 22-May-2015].

[13] Omidalizarandi, M. and Saadatseresht, M. (2013). Segmentation and classification of point clouds from dense aerial image matching. *The International Journal of Multimedia & Its Applications*, 5(4):33.

[14] OpenCV (2014). Features2D + Homography to find a known object. http://docs.opencv.org/doc/tutorials/features2d/feature_homography/feature_homography.html. [Online; accessed 08-April-2015].

[15] OpenCV (2015). Open Source Computer Vision. http://opencv.org/. [Online; accessed 12-March-2015].

[16] Patnaik, S. and Zhong, B. (2014). *Soft Computing Techniques in Engineering Applications*, volume 543. Springer.

[17] Payen de La Garanderie, G. and Breckon, T. (2014). Improved depth recovery in consumer depth cameras via disparity space fusion within cross-spectral stereo. In *Proceedings of the British Machine Vision Conference*. BMVA Press.

[18] PCL (2013). Point Cloud Library. http://pointclouds.org/. [Online; accessed 19-May-2015].

[19] Rabbani, T., van den Heuvel, F., and Vosselmann, G. (2006). Segmentation of point clouds using smoothness constraint. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 36(5):248–253.

[20] Radke, R. J. (2013). *Computer vision for visual effects*. Cambridge, 1st edition.

[21] Robots, K. I. (2015). KR AGILUS sixx. http://www.kuka-robotics.com/res/sps/e6c77545-9030-49b1-93f5-4d17c92173aa_Spez_KR_AGILUS_sixx_en.pdf. [Online; accessed 15-May-2015].

[22] ROS. ROS home page. http://wiki.ros.org/. [Online; accessed 10-May-2015].

[23] Rusu, R. B. (2009). *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Technischen Universität München, München.

[24] Rusu, R. B., Blodow, N., and Beetz, M. (2009). Fast point feature histograms (fpfh) for 3d registration. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 3212–3217. IEEE.

[25] Rusu, R. B., Blodow, N., Marton, Z. C., and Beetz, M. (2008). Aligning point cloud views using persistent feature histograms. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 3384–3391. IEEE.

[26] Shao, L., Han, J., Kohli, P., and Zhang, Z. (2014). *Computer Vision and Machine Learning with RGB-D Sensors*. Springer.

[27] Stückler, J., Steffens, R., Holz, D., and Behnke, S. (2013). Efficient 3d object perception and grasp planning for mobile manipulation in domestic environments. *Robotics and Autonomous Systems*, 61(10):1106–1115.

[28] Tutorial, A. I., Perwass, C. B., and Hildenbrand, D. (2004). Aspects of geometric algebra in euclidean, projective and conformal space.

[29] Ugolotti, R., Micconi, G., Aleotti, J., and Cagnoni, S. (2014). Gpu-based point cloud recognition using evolutionary algorithms. In *Applications of Evolutionary Computation*, pages 489–500. Springer.

[30] Vince, J. (2009). *Geometric Algebra: An Algebraic System for Computer Games and Animation.* Springer, 1st edition.