INTERNATIONAL GRADUATE STUDIES IN MECHATRONICS

UNIVERSITY OF SIEGEN

PROJECT WORK REPORT

A Rapid Prototyping Framework for Human-Robot Interaction

Author:	Henry Ugochukwu Odoemelem, 1463255
Supervisor:	Prof. Dr. Kristof Van Laerhoven
Advisor:	Prof. Dr. Kristof Van Laerhoven
Submission Date:	July 1, 2020

I confirm that this project work report is my own work and I have documented all sources and material used.

Siegen, July 1, 2020 Henry Ugochukwu Odoemelem, 1463255

Abstract

Many current human-robot interactive systems tend to use accurate and fast, but also costly, actuators and tracking systems to establish working prototypes that are safe to use and deploy for user studies. This work presents an embedded framework to build a desktop space for human-robot interaction, using an open-design robot arm, as well as two RGB cameras connected to a Raspberry Pi-based controller that allow a fast yet low-cost object tracking and manipulation in 3D. Results shows that this facilitates prototyping a number of systems in which user and robot arm can commonly interact with physical objects.

Contents

1. Introduction

This project presents an embedded framework for human-robot interaction to allow robotics enthusiasts and researchers to develop low-cost and rapid prototypes of robot arms, to test out project ideas for educational purposes or before a full scale implementation. This report will show that the robot arm can autonomously and quickly locate any object(s) at any position within its workspace, and can perform a specific (different) task depending on the id of the fiducial marker identified. Also, early stage development of the embedded framework is introduced and explained, experimental case studies and performance results are presented.

2. Methodology

This chapter will describe the system components and working principles in detail.

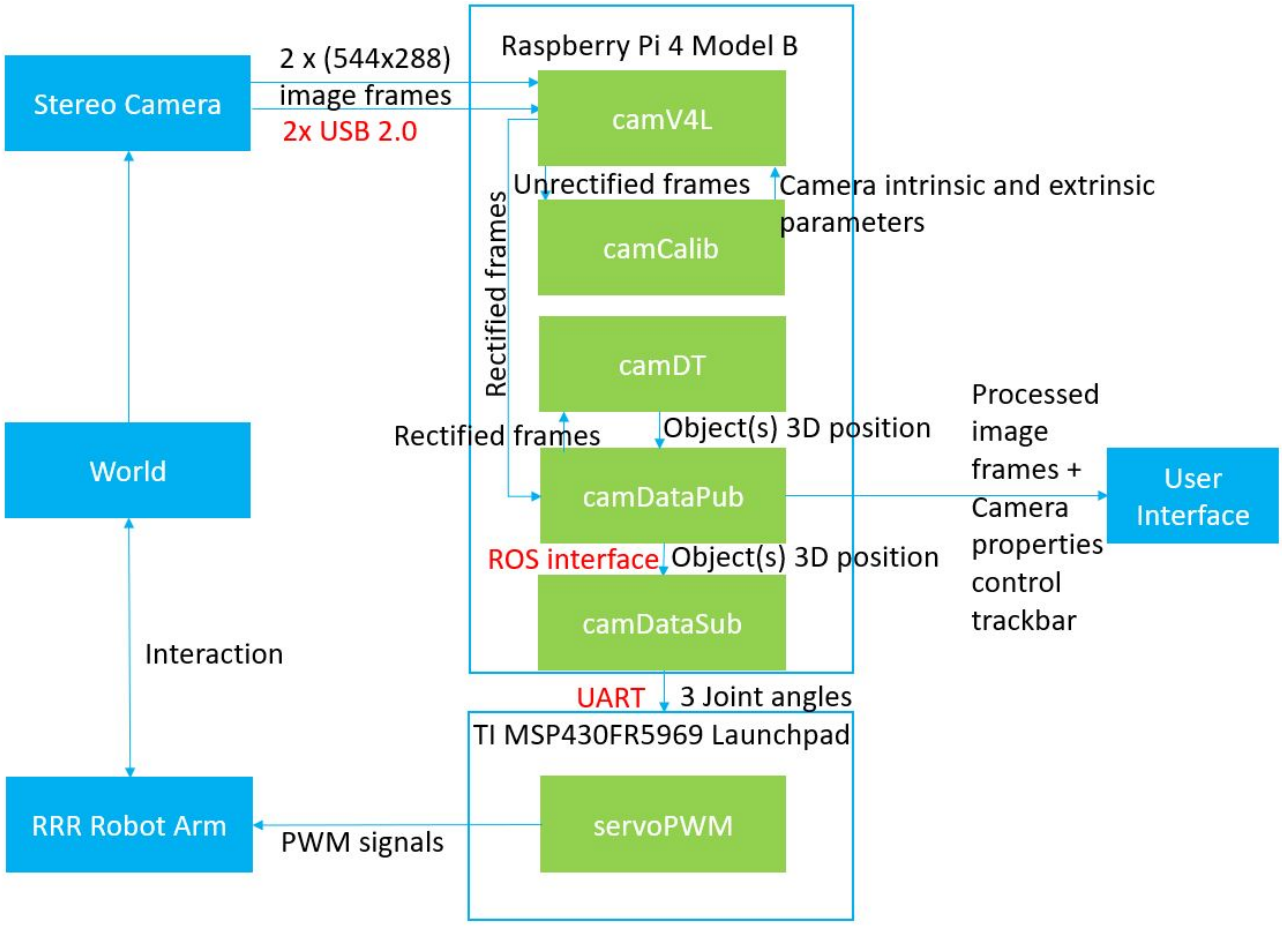

Figure 2.1.: Overall system architecture

The system architecture and methodology in a nutshell is shown in figure 2.1. The image frames from the stereo camera are read by the Raspberry Pi. Using the system source code (in green) written in Python3, the Raspberry Pi processes these images to obtain tracking data (3D position of fiducial marker(s)). The processed image frames and camera properties control trackbar are displayed on the user interface. A ROS publisher node is then used to publish the tracking data, while a ROS subscriber node receives this data, which is then

converted into joint angles using inverse kinematics. The joint angles are then sent to the TI MSP430FR5969 micro-controller through UART. servoPWM is the Embedded C source code that runs on the TI MSP430FR5969 micro-controller, which uses the joint angles, a lookup table and Hardware timers to generate corresponding PWM signals. This PWM signals are then sent to the robot arm servo motors to command joint positions, so that the end effector reaches the 3D position of the fiducial marker(s) identified. This process, ensures that the robot arm is able to track the 3D position of object(s) with fiducial marker(s) within its workspace. To understand the fundamentals of stereo camera, the photogrammetry course by Cyrill Stachniss on [1], is a great learning resource.

The following subsections, provides a detailed explanation of the system components and methodology.

2.1. Hardware Description

The hardware components of the system is described here in detail.

2.1.1. User Interface

For the user interface, the user can SSH/VNC to connect to Raspberry Pi from a PC or use the Raspberry Pi Touch Display. The user interface used for all testings in this project is the FUJITSU Display B24-8 TE Pro [2], which is connected to the Raspberry Pi through a HDMI cable. It is possible to observe lower or higher fps, depending on the frequency of the Display used.

2.1.2. Raspberry Pi 4 Model B

The Raspberry Pi 4 Model B on figure 2.2 [[3]] has 4 Gigabyte LPDDR4 RAM, and is capable of H.265 (HEVC) hardware decode (up to 4Kp60). It also has 4 USB port which are sufficient for the two USB cameras used in stereo normal case, the USB connected TI MSP430FR5969 micro-controller, and an extra space for the Bluetooth enabled keyboard and mouse used. More details on Raspberry Pi 4 Model B specifications can be found here [4] .

[1] https://www.youtube.com/watch?v=_mOG_lpPnpYlist=PLgnQpQtFTOGRsi5vzy9PiQpNWHjq-bKN1
[2] https://sp.ts.fujitsu.com/dmsp/Publications/public/ds-display-b24-8-te-pro.pdf
[3] https://www.raspberrypi.org/products/raspberry-pi-4-model-b/
[4] https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711/rpi_DATA_2711_1p0_preliminary.pdf

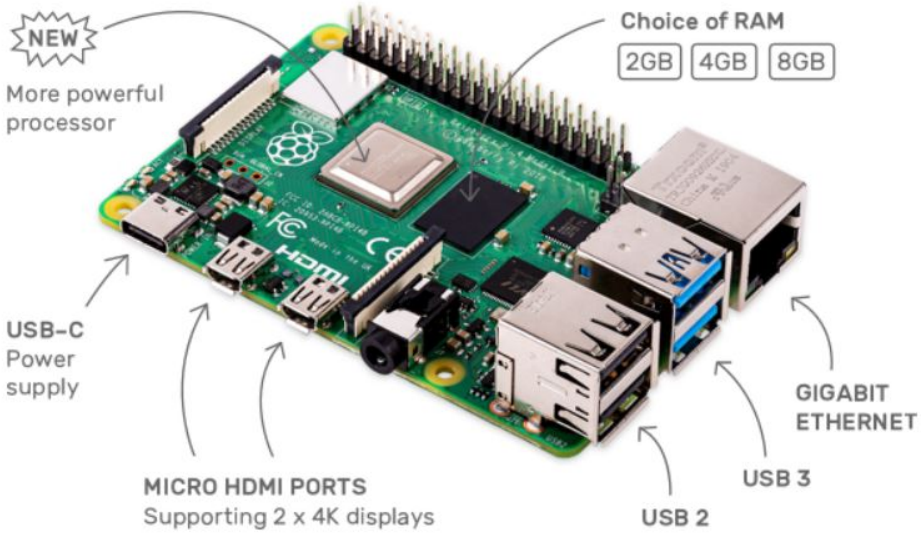

Figure 2.2.: Raspberry Pi 4 Model B.

2.1.3. TI MSP430FR5969 LaunchPad

TI MSP430FR5969 on figure 2.3 [5] is an efficient micro-controller. The User's guide [6] and datasheet [7], provide more information on specifications and how to program the micro-controller.

2.1.4. Stereo camera

To achieve object triangulation by stereo imaging, two low-cost Logitech c525 webcams with 69° Field of view (FoV) are used to implement a cost-effective stereo camera in the normal case configuration as shown in figure 2.4 .To reduce errors in the configuration, stereo calibration and rectification were done, this will be discussed in more details in section 2.2 . The intrinsic and extrinsic properties of the stereo camera from calibration and rectification is on figure 2.9.

The basic concept of stereo triangulation is illustrated in figure 2.5. The depth Z of an object can be found if we know the disparity $d = \left(x^l - x^r\right)$, where x^l and x^r are the horizontal position of the object point P in left and in right image planes respectively, focal length f and the horizontal translation T between the two cameras as show in equation (2.1).

$$\frac{T - \left(x^l - x^r\right)}{Z - f} = \frac{T}{Z} \Rightarrow Z = \frac{fT}{x^l - x^r} \tag{2.1}$$

[5]https://de.farnell.com/ti-msp430fr5969-launchpad-development-kit
[6]https://www.ti.com/lit/ug/slau367p/slau367p.pdf
[7]https://www.ti.com/lit/ds/symlink/msp430fr5969.pdf

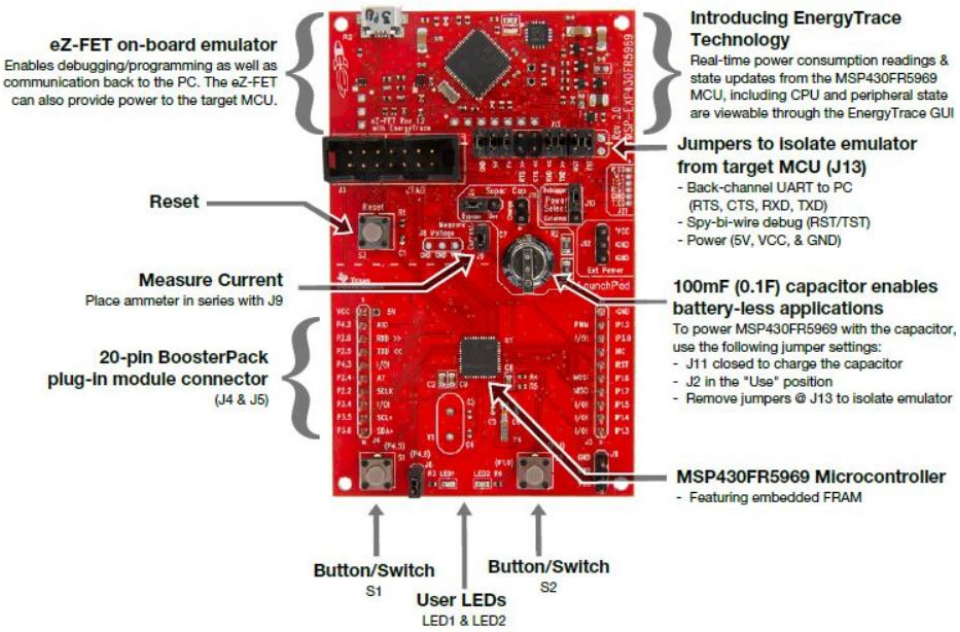

Figure 2.3.: TI MSP430FR5969 LaunchPad.

Figure 2.4.: Stereo camera assembly using two Logitech c525 webcams.

2.1.5. RRR Robot Arm and Inverse Kinematics

The 3 joints of the RRR robot arm are fitted with MG996R servo motors [8]. The 3D printed model of the robot arm links and base used in this project is an open-design made available by [9] .

[8]https://pdf1.alldatasheet.com/datasheet-pdf/view/1131873/ETC2/MG996R.html
[9]https://howtomechatronics.com/tutorials/arduino/diy-arduino-robot-arm-with-smartphone-control/

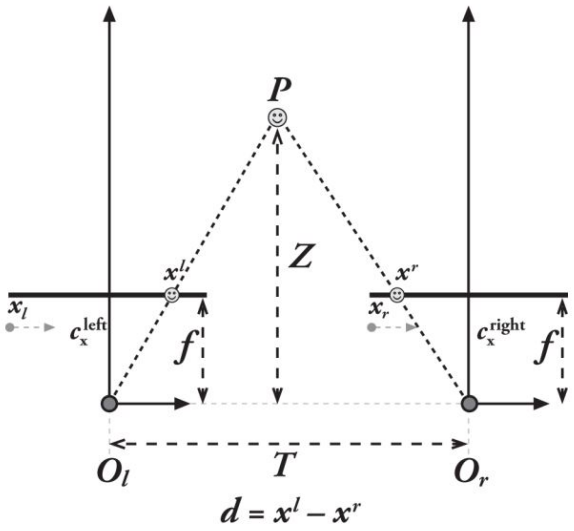

Figure 2.5.: Perfectly undistorted stereocamera. [1]

The coordinate frames in figure 2.6 is assumed in the formulation of the inverse kinematics of the 3 DOF robot arm.

The frame of reference *i*, with coordinate (x*i* ,y*i* ,z*i*), where for i = 1,2,3 is the reference frame for links L1,L2,L3 respectively. For i = 0 , we have reference frame 0, which is the inertial reference frame. The links rotates about the z-axis with angles θ_1,θ_2,θ_3, respectively. The transformation matrix from reference frame *i* to reference frame *i-1* is A_i^{i-1} and is defined by the homogeneous matrix as seen in equation (2.2). Where $R(\theta)$ is a link rotation about a joint and T is a link translation.

$$A_i^{i-1} = \begin{bmatrix} R(\theta) & T \\ 0 & 1 \end{bmatrix}_i^{i-1} \tag{2.2}$$

For our particular case, the transformation matrix A_{EF}^0 from the end effector EF to the inertial frame of refeerence $(x0, y0, z0)$ is computed as in equation (2.3)

$$A_{EF}^0 = \begin{bmatrix} R_z(\theta_1) & 0 \\ 0 & 1 \end{bmatrix}_1^0 \begin{bmatrix} R_z(0) & T_{L1} \\ 0 & 1 \end{bmatrix}_2^1 \left[\begin{bmatrix} R_x(90) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_z(\theta_2) & T_{L2} \\ 0 & 1 \end{bmatrix} \right]_3^2 \begin{bmatrix} R_z(\theta_3) & T_{L3} \\ 0 & 1 \end{bmatrix}_{EF}^3 \tag{2.3}$$

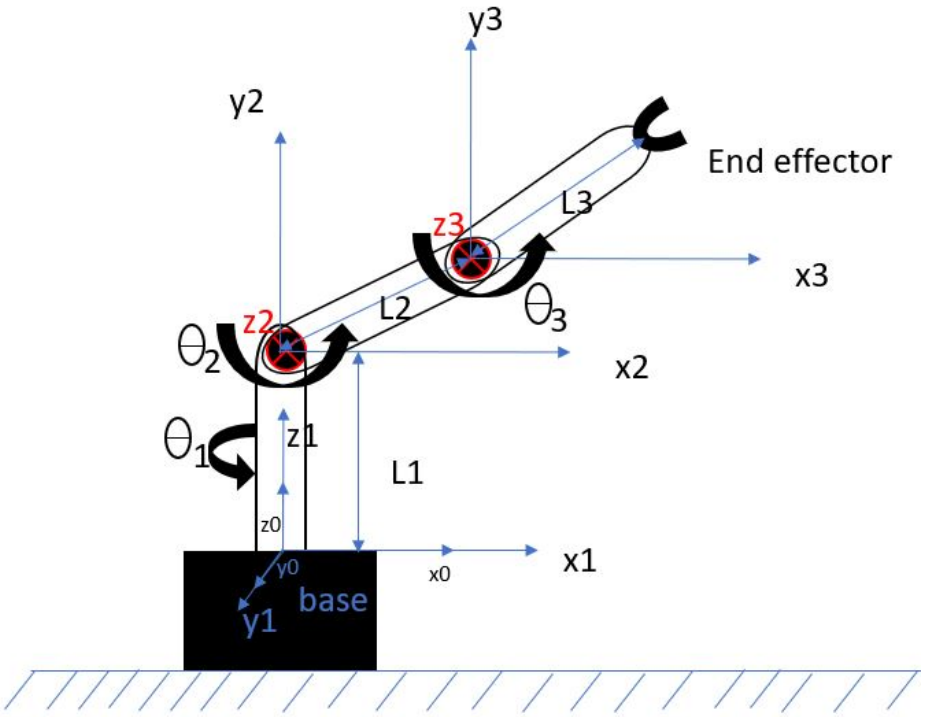

Figure 2.6.: 3 DOF RRR Robot arm reference frames

where;

$$\begin{bmatrix} R_z(\theta_1) & 0 \\ 0 & 1 \end{bmatrix}_1^0 = \begin{bmatrix} c\theta_1 & -s\theta_1 & 0 & 0 \\ s\theta_1 & c\theta_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} R_z(0) & T_{L1} \\ 0 & 1 \end{bmatrix}_2^1 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & L1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\left[\begin{bmatrix} R_x(90) & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} R_z(\theta_2) & T_{L2} \\ 0 & 1 \end{bmatrix} \right]_3^2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & c90 & -s90 & 0 \\ 0 & s90 & c90 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} c\theta_2 & -s\theta_2 & 0 & L2c\theta_2 \\ s\theta_2 & c\theta_2 & 0 & L2s\theta_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} R_z(\theta_3) & T_{L3} \\ 0 & 1 \end{bmatrix}^3_{EF} = \begin{bmatrix} c\theta_3 & -s\theta_3 & 0 & L3c\theta_3 \\ s\theta_3 & c\theta_3 & 0 & L3s\theta_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The above computation results in equation (2.4)

$$A^0_{EF} = \begin{bmatrix} c\theta_1 c\psi & -\cos s\varphi & s\theta_1 & L3c\theta_1 c\psi + L2c\theta_2 c\theta_1 \\ s\theta_1 s\psi & -s\theta_1 s\psi & -c\theta_1 & L3s\theta_1 c\psi + L2c\theta_2 s\theta_1 \\ s\psi & c\psi & 0 & L3s\psi + L2s\theta_2 + L1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.4}$$

where c is cos and s is sin, ψ is the angle of the end effector $\psi = \theta_2 + \theta_3$, and the position (X, Y, Z) of the end effector with respect to the inertial frame of reference is

$$X = L3c\theta_1 c\psi + L2c\theta_2 c\theta_1, Y = L3s\theta_1 c\psi + L2c\theta_2 s\theta_1, Z = L3s\psi + L2s\theta_2 + L1$$

Hence, given the position and orientation (X, Y, Z, ψ) of the end effector, the joint angles $(\theta_1, \theta_2, \theta_3)$ can be calculated as follows;

$$Z = L3s\psi + L2s\theta_2 + L1 => s\theta_2 = \frac{Z - LI - L3s\psi}{L2}$$

$$c\theta_2 = \pm\sqrt{1 - (s\theta_2)^2}$$

Angle θ_2 of second joint is as shown in equation (2.5)

$$\theta_2 = atan2\left(\frac{s\theta_2}{c\theta_2}\right) \tag{2.5}$$

similarly,

$$s\theta_1 = \frac{Y}{L3c\psi + L2c\theta_2}$$

and Angle θ_1 of first joint is as shown in equation (2.6)

$$\theta_1 = atan2\left(\frac{s\theta_1}{c\theta_1}\right) \tag{2.6}$$

Angle θ_3 of the third joint is as shown in equation (2.7)

$$\theta_3 = \psi - \theta_2 \tag{2.7}$$

2.1.6. Hardware cost

The total hardware cost of this project is under 300€.

Table 2.1.: Cost of hardware components

Component	Quantity	Cost
Raspberry Pi 4 Model B 4GB Plus Desktop kit	1	∼ 100€
TI MSP430FR5969 LaunchPad	1	∼ 19 €
Logitech webcam c525	2	∼ 135 €
MG996R servo motor	3	∼ 20 €
Total cost		∼ 274 €

2.2. Software/Algorithm Description

The software algorithm of the system is described here in detail. A link to the source code repository can be found in the appendix section.

2.2.1. Software Dependencies

To successfully run the source-code described in this section, the following software installations are necessary. You will need to install Raspberry Pi OS (previously called Raspbian), Python3 and openCV 4, links [10] and [11] are helpful to accomplish these installations on Raspberry Pi 4 Model B. [12] is a great resource for the openCV modules used in this project. ROS node is used in this project for publishing and subscribing object 3D position. To install ROS Melodic on Raspberry Pi 4 Model B, this link [13] will walk you through the installation. The image processing source code for this system was written in Python3 using the Visual Studio code editor on Raspberry Pi 4 Model B, go to [14], for the installation guide. Finally, the Embedded C source code that runs on the TI MSP430FR5969 to command joint angles was written on Windows 10 OS, using the Code Composer studio. Here is the Code Composer studio product page [15] and direct download link [16].

2.2.2. camV4L: Camera Video for Linux

This module defines the class camImage() which consist of the following methods.

[10] https://www.pyimagesearch.com/2019/09/16/install-opencv-4-on-raspberry-pi-4-and-raspbian-buster/
[11] https://qengineering.eu/install-opencv-4.1-on-raspberry-pi-4.html
[12] https://docs.opencv.org/master/
[13] https://www.instructables.com/id/ROS-Melodic-on-Raspberry-Pi-4-RPLIDAR/
[14] https://www.youtube.com/watch?v=izOhCUdLv2s
[15] http://www.ti.com/tool/ccstudio
[16] http://processors.wiki.ti.com/index.php/Download_CCS

1. __init__(self, source = 0, resolution = (544, 288))
   This is the constructor of the class, it takes two arguments, *source* , which is the index of the first camera and *resolution*, which is the desired image resolution. The default image resolution, which was used in calibrating the camera is $(544, 288)$. Also using v4l2-ctl, default camera properties are set here. The camera intrinsic and extrinsic parameters from Camera Calibration are defined here as class attributes, and are used to get rectification maps and projection matrices.
2. read(self)
   This method simply reads the new camera frames if both cameras are available. It also uses the rectification maps to rectify the images using openCV cv2.remap() method, and then returns the rectified images to the calling function. Note, when using this method to read images for camera calibration, cv2.remap() is commented out.
3. getPl(self)
   This method returns the projection matrix for left camera.
4. getPr(self)
   This method returns the projection matrix for right camera.
5. StopCam(self)
   Use this method to release both cameras before exiting the program.
6. setExposure(self, exposure)
   Call this method to set the camera exposure dynamically.
7. setContrast(self, contrast)
   Call this method to set the camera contrast dynamically.

2.2.3. camCalib: Camera Calibration

Camera calibration is based on [1] .

1. snapShot():
   For camera calibration, at least 10 images of a 7-by-8 or larger chessboard is required according to [1]. In this project, at least 50 chessboard images (per camera) in different orientations and positions were used for stereo calibration and rectification. This function is used to take chessboard images (7-by-9 chessboard is used in this project as shown in figure 2.7) and save them into a file, to be used for camera calibration.
2. The camera calibration process begins with finding the chessboard corners on the chessboard images saved as shown in figure 2.8, using openCV cv2.findChessboardCorners() function, and then appending the corresponding image points (chessboard corners in pixels coordinate found) and object points (physical coordinates of each chessboard corner) into a list.

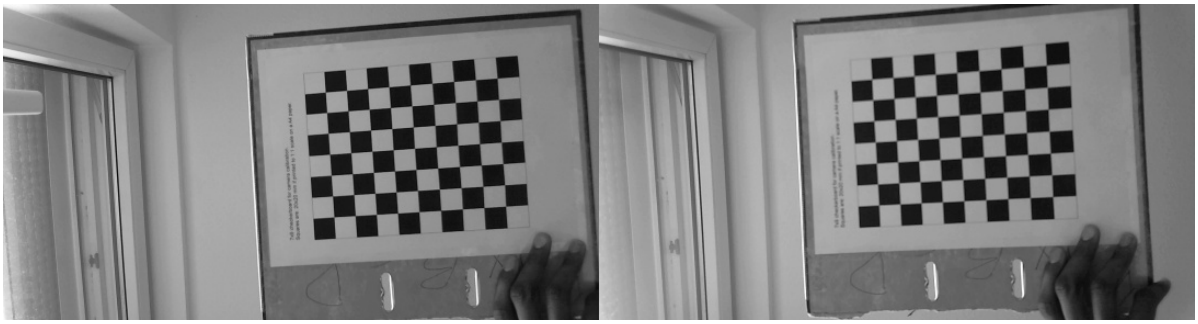

Figure 2.7.: Left and Right chessboard images.

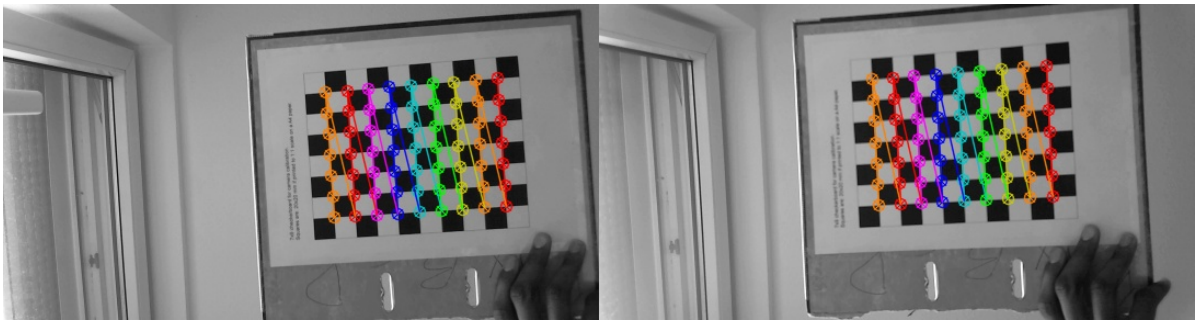

Figure 2.8.: Left and Right chessboard image corners found.

The image and object points are then passed to cv2.calibrateCamera(), to obtain the camera intrinsic and extrinsic parameters, by finding the solution to equation 2.8.

$$q = sMWQ \tag{2.8}$$

where

$$q = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$sMWQ = s \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

q is the image pixels coordinate system , s is a non-zero scale factor, M is the intrinsic camera parameters (the focal length f_x,f_y and position displacements c_x,c_y away from the optic axis). $W = \begin{bmatrix} R_z(\theta_z)R_y(\theta_y)R_x(\theta_x) \mid \mathbf{t} \end{bmatrix}$ is the extrinsic camera parameters, which is the rotation and translation of the object relative to the camera coordinate system. Q is the world/scene coordinate system. WQ gives the camera coordinate

system. cv2.calibrateCamera() also return the radial and tangential distortions of the cameras which are necessary parameters for undistorted rectification.

3. The stereo camera projection P (contains the rectified camera matrices and the rotation(R) and translation(S) between left and right cameras) and reprojection matrices A, are then calculated using cv2.stereoRectify(). Given a 3D point in homogeneous coordinate, a 2D point in homogeneous coordinate can be calculated as shown in equation (2.9), where the image coordinate is $(x/w, y/w)$. Similarly, given 2D homogeneous point and its associated disparity d, we get the 3D homogeneous point as shown in equation (2.10), where the 3D point is $(X/W, Y/W, Z/W)$.

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = P \begin{bmatrix} x \\ y \\ Z \\ 1 \end{bmatrix} \tag{2.9}$$

$$\begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = A \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} \tag{2.10}$$

4. The result of the camera calibration and stereo rectification done is shown in figure 2.9.

2.2.4. camDT: Camera Detect and Track

This module has class camDetectandTrack(threading.Thread), with the following methods;

1. __init__(self,arucoID,tractID = 6,newFrame = False,frameL = None,frameR = None,Pl = None, Pr = None)
   This class constructor takes as argument *arcuoID*, which is the ID of the fiducial marker (ArUCo), a tracker id *tractID* with a default value of ,6, which corresponds here to "MOOSE" tracker. The default parameter *newFrame* which indicates if the camera has a new frame or not. $frameL$ and $frameR$, are the initial left and right camera frames respectively. *Pl* and *Pr* are the left and right projection matrices respectively.

2. exitcamDetectandTrack(self)
   This method should be called just before existing your main program to inform the user that the program is exiting.

3. Run(self,newFrame,frameL,frameR)
   Run this method to first detect an object with a fiducial marker before tracking can start.

4. startTracking(self,newFrame,frameL,frameR)
   In this method, a tracker is implemented using the user selected openCV tracker.

```
Right camera matrix(Mr):
[[439.59731982   0.         269.80055053]
 [  0.         439.22300607 143.95737863]
 [  0.           0.           1.        ]]

Left camera matrix (Ml):
[[424.37377986   0.         273.05342749]
 [  0.         425.06630163 140.10688074]
 [  0.           0.           1.        ]]

Right camera distortion (dr):
[[ 0.07498266]
 [-0.18503787]
 [ 0.00111275]
 [ 0.00726558]
 [ 0.39932496]]

Left camera distortion(dl):
[[ 0.05257172]
 [-0.29232969]
 [-0.00098365]
 [ 0.00617034]
 [ 0.07534596]]

Rotation(T) between cameras:
[[ 0.99995187 -0.00837098  0.00511781]
 [ 0.00853753  0.99940436 -0.03343718]
 [-0.00483486  0.03347926  0.99942772]]

Translation(S) between cameras:
[[-47.19772823]
 [ -0.9012592 ]
 [  9.28331359]]

Essential Matrix E = R*S(Before stereo rectification):
[[ -0.07489912  -9.30795753  -0.59033563]
 [  9.05467255   1.50243462  47.21822801]
 [  0.49826377 -47.17715959   1.58277126]]

Fundamental Matrix F = (Mr^-1)^T*E*(Ml^-1) (Before stereo rectification):
[[-2.16097126e-07 -2.68113424e-05  3.09265679e-03]
 [ 2.61465868e-05  4.33141408e-06  5.01166622e-02]
 [-3.07373526e-03 -5.31278671e-02  1.00000000e+00]]

Right camera Projection matrix Pr (after stereo rectification):
[[ 4.32144654e+02  0.00000000e+00  3.96685831e+02 -2.07906842e+04]
 [ 0.00000000e+00  4.32144654e+02  1.40836176e+02  0.00000000e+00]
 [ 0.00000000e+00  0.00000000e+00  1.00000000e+00  0.00000000e+00]]

Left camera Projection matrix Pl (after  stereo rectification):
[[432.14465385   0.         396.68583107   0.        ]
 [  0.         432.14465385 140.83617592   0.        ]
 [  0.           0.           1.           0.        ]]

Reprojection matrix A (after stereo rectification):
[[ 1.00000000e+00  0.00000000e+00  0.00000000e+00 -3.96685831e+02]
 [ 0.00000000e+00  1.00000000e+00  0.00000000e+00 -1.40836176e+02]
 [ 0.00000000e+00  0.00000000e+00  0.00000000e+00  4.32144654e+02]
 [ 0.00000000e+00  0.00000000e+00  2.07854946e-02 -0.00000000e+00]]
```

Figure 2.9.: Result of camera calibration and stereo rectification

In other to improve accuracy of the tracker, especially with more than one fiducial marker on the scene, and also to increase the processing and tracking speed, the main frames $frameL$,$frameR$ received are first copied to a temporary variables $self.frameLtemp$,$self.frameRtemp$. The main frames are then cropped and used in the tracking process. The following terminologies will be adopted to explain the process;
BBMF = BBTF + TFMF
BBMF = Bounding Box coordinate with respect to the Main Frame
BBTF = Bounding Box coordinate with respect to the Tracking Frame
TFMF = Tracking Frame coordinate with respect to the Main Frame
The tracking Frame is obtained by adding a small margin $(self.expand)$ to Bounding Box frame , thus, tracking frame is larger than the bounding box frame, and they are only equal if the margin (self.expand) is set to 0. An example of the main frame, tracking frame and bounding box frame can be found in figure 2.10,2.11 and 2.12 respectively.

Figure 2.10.: Main image frame.

Figure 2.11.: Tracking image frame.

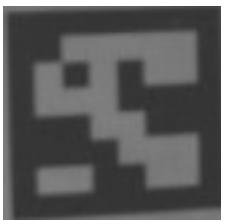

Figure 2.12.: Bounding box image frame.

First, copies of the main frames are created using $self.frameLtemp, self.frameRtemp$. Then the main frames are cropped using TFMF coordinates to get the tracking frame. If no object (fiducial marker) is found in the last iteration because of tracker failure, $self.detectFrame()$ is called and uses copies of $self.frameLtemp, self.frameRtemp$ to detect the current position of the object (fiducial marker) before tracking starts/continues. Else, the tracking frame is passed to the tracker which returns the new BBTF for the left and right images. Then, the new TFMF is calculated. Some trackers can erroneous return values after the object (fiducial marker) has been removed from the scene. To avoid this error, the bounding box returned by the tracker is periodically examined. This is done by checking the number of contours and the total area of the contours within the returned bounding box, after removing the largest contour, which is usually the contour of the frame's boundary. A bounding box with few contours (typically < 5) and small area (typically $<< 100$) from testings (change the limits according to your test condition), indicate that the object (fiducial marker) has been removed from the scene and hence, the tracker is reset using $self.resetTracker()$. After that, the new BBMF for left and right image is calculated, and is used to calculate the midpoint of the object (fiducial marker) on the left and right images in pixel coordinates. The midpoints in pixel coordinate, together with the projection matrices (Pl and Pr) are passed to openCV cv2.triangulatePoints() method. This method returns a homogeneous coordinate that represents the world position coordinate (X, Y, Z) of the object (fiducial marker) midpoint. Next, the bounding box,midpoint position etc. are marked on copies of $self.frameLtemp, self.frameRtemp$ (that is the main frame), for the purpose of display on the user interface. Finally, $startTracking(self, newFrame, frameL, frameR)$ returns the marked frames, bounding boxes and the object (fiducial marker) midpoint in world position coordinate (in cm).

5. getBox(self)
   A simple method that returns the BBMF for left and right images.

6. getPosition(self)
   A simple method that returns the position $(self.x, self.y, self.z)$ of object (fiducial marker) midpoint.

7. drawLine(self,image,line):
   This method is used by drawEpipolarLine(self,inframeL,inframeR,inBBMFL,inBBMFR)

within the class to draw lines on images.

8. drawEpipolarLine(self,inframeL,inframeR,inBBMFL,inBBMFR)
   This method uses the image frames and BBMF to get midpoint (ql and qr) of the fiducial marker to draw Epipolar lines $lr = F * ql$ and $ll = F^T * qr$ on right and left frames respectively.
   where, F is the fundamental matrix of a stereo camera, and $F = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix}$, for a rectified stereo camera in normal case, ql is an image point in pixels coordinate on the left image frame, while qr is the corresponding image point on the right image frame. Drawing epipolar lines can be used to verify that the stereo image is correctly rectified and undistorted, as a truly rectified stereo image will have horizontal Epipolar lines aligned on both left and right images as shown in figure 2.13.

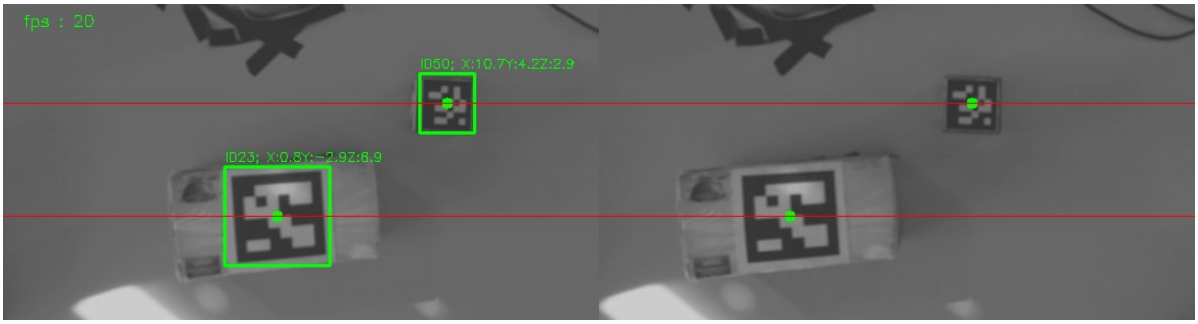

Figure 2.13.: Left and Right rectified images showing Epipolar line.

9. resetTracker(self)
   In the event of a tracking failure, this method is called to reset the tracker.

10. detectFrame(self)
    When detecting object (fiducial marker), the main frames are used and passed to cv2.aruco.detectMarkers(), which returns all available marker ids and their 4 corners. If the returned ids contain our object marker id, we get BBMF for both frames using cv2.boundingRect(). Next, the user selected tracker is created. TFMF and BBTF are then calculated. TFMF is then used to get the new tracking frames. The tracking frames, together with BBTF for both frames are used to initialize the trackers using $trackerL.init()$, $trackerR.init()$.

2.2.5. camDataPub: Camera Data Publisher

In this module, an object of $camImage()$ class is created to receive rectified image frames and its methods are used in cv2.createTrackbar() to enable users control the camera properties like contrasts, exposure etc dynamically. Objects of the class $camDetectandTrack()$ are created for

each fiducial marker that need to be identified. The $Run()$ method of $camDetectandTrack()$ class is used to detect object(s) (fiducial marker) before the tracking begins.

1. talker()
   In this function, $startTracking()$ method of $camDetectandTrack()$ class is used to get processed image frames, bounding box, and midpoint position (in cm) of object(s) (fiducial marker) in world coordinates. The fps and processed images are displayed on the user interface. Then using ROS node "*talker*", the position data is published on the topic "*chatter*".

2.2.6. camDataSub : Camera Data Subscriber

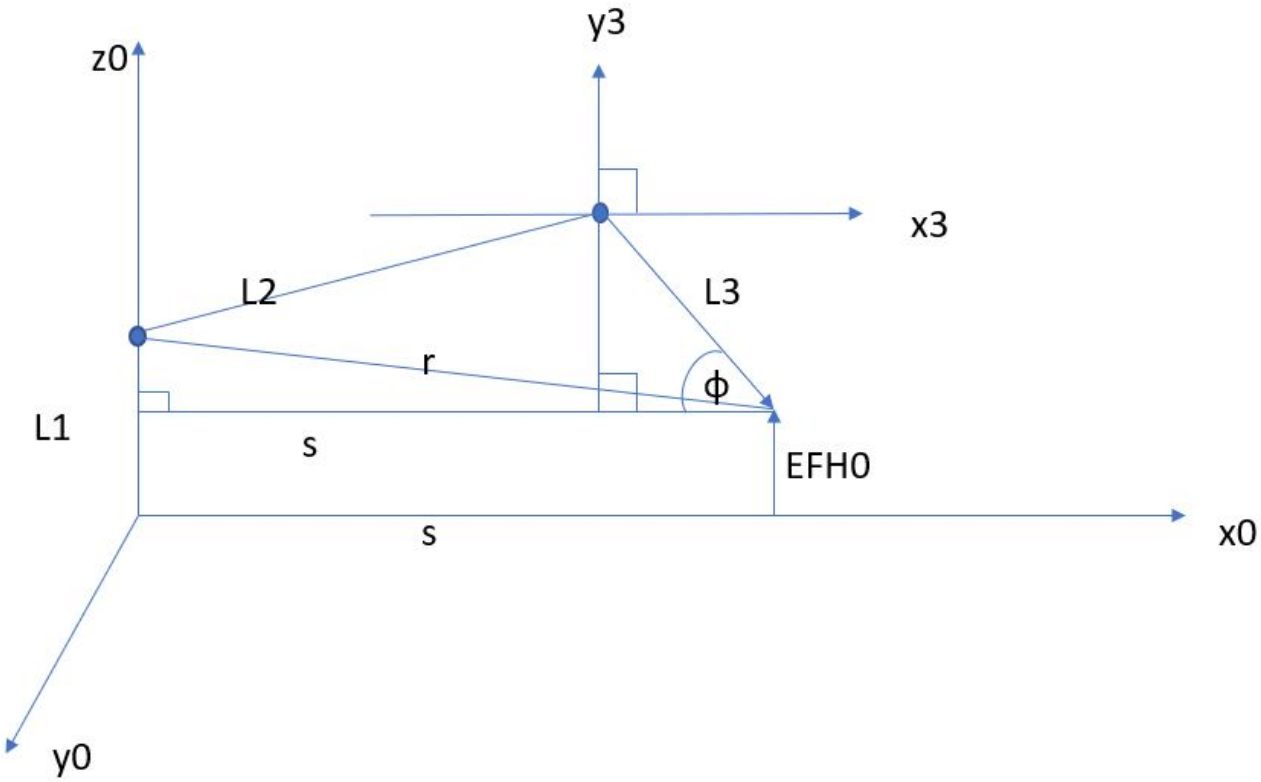

Figure 2.14.: Getting the angle of end effector

1. inverse_Kinematics(ix,iy,EfH)
   Given the end effector position (ix, iy, EfH) as parameters we get $(x, y, EFH0)$, where $EFH0$ is the end effector height after subtracting the height of the robot arm base. The

orientation of the end effector ψ is calculated and implemented using the end effector position, as illustrated in figure 2.14 and equations (2.11),(2.12),(2.13) and (2.14). Hence, the joint angles as shown in equations (2.6),(2.5) and (2.7) are implemented.

$$s = \sqrt{x^2 + y^2} \tag{2.11}$$

$$r = \sqrt{x^2 + y^2 + EFH0^2} \tag{2.12}$$

$$\phi = acos\left(\frac{r^2 + L3^2 - L2^2}{2rL3}\right) + atan2\left(\frac{L1 - EFH0}{s}\right) \tag{2.13}$$

$$\psi = 360^\circ - \phi \tag{2.14}$$

2. listener()
   This function has ROS node "*listener*", which subscribes to the topic "*chatter*", hence, object (fiducial marker) position published by the "*talker*" node on topic "*chatter*" is received here. The received position data are then passed to $inverse\_Kinematics(ix, iy, EfH)$ function, which returns the joint angles $(\theta_1, \theta_2, \theta_3)$. The joint angles (in bytes), are then sent to TI MSP430FR5969 through UART. The byte preambles 251,252 and 253 are used to indicate that a joint angle data belongs to joint 1 (for link L1), joint 2 and joint 3 respectively. Note that the servo motor has a possible range of 0 to 180°.

2.2.7. servoPWM

servoPWM is the Embedded C program that runs on TI MSP430FR5969 to send PWM signals corresponding to commanded joint angles to the robot arm servo motors. It receives the joint angles through UART from *camDataSub*. The PWM period is 20ms or 50 Hz. The joint angle (range is 0 to 180°) is then used as index for a look up table. The look up table contains corresponding values (0(900 counts) to 180°(2700 counts), with an increment of 10 counts for each integer degree), which is the number of timer counts for Timer A1 and B0 to send PWM signals to P1.3 (joint 1 for link L1),P1.4 (joint 2) and P1.5(joint 3). Note that an appropriate external power supply is used to power the servo motors. More information on programming TI MSP430 can be found here [17] .

[17]https://www.youtube.com/watch?v=KfFBEBN5UHUlist=PL643xA3Ie_EuHoNV7AgvJXq-z1hrE8vsm

3. Results and Discussion

3.0.1. Object tracking processing speed

The typical system setup is shown in figure 3.1. As seen in this figure, the robot arm has located the object tagged with ArUco id 23. For 2 objects in the scene simultaneously, the image processing time is between 50ms to 33ms, which is 20-30 fps. The processing time increases as the number of objects in the scene to be identified increases.

Figure 3.1.: Typical system setup

3.0.2. Case study: The cups-and-ball game

This section presents the use of the system in a simple task of cups-and-ball game as a demonstration of the object tracking and how well the robot arm locates the tracked object. A link to the video can be found here [1].

[1] https://youtu.be/wiNIN8MH4uc

4. Conclusion and Recommendations

The previous chapters of this project has shown that the object tracking is fast and the robot arm can autonomously locate any tagged object within its workspace. And with different tags on the object(s), the robot arm can perform a different task depending on the tag id identified. Recommendations for future works to improve the system includes: implementing a filter to reduce tracker noise, precise closed loop torque, velocity and position control, robust motion planning, improved low-cost 3D printable robot arm design with suitable end effectors, robust automatic illumination monitoring/control and improved low-cost stereocamera design.

A. Appendix

The source code for this embedded framework is on this GitHub link [1]

[1] https://github.com/hodoemelem/A-Rapid-Prototyping-Framework-for-Human-Robot-Interaction

List of Figures

List of Tables

Bibliography

[1] A. Kaehler and G. Bradski. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. 1st. O'Reilly Media, Inc., 2016. ISBN: 1491937998.