A Low-Cost Prototyping Framework for Human-Robot Desk Interaction

Henry Ugochukwu Odoemelem
henry.odoemelem@student.uni-siegen.de
Ubiquitous Computing Lab, University of Siegen
Siegen, Germany

Kristof Van Laerhoven
kvl@eti.uni-siegen.de
Ubiquitous Computing Lab, University of Siegen
Siegen, Germany

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 paper presents an embedded framework to build a desktop space for human-robot interaction, using an open-source 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. We show in our evaluations that this facilitates prototyping a number of systems in which user and robot arm can commonly interact with physical objects.

CCS CONCEPTS

• **Computer systems organization** → **Robotics**; *Embedded systems.*

KEYWORDS

human-robot interaction; tangible computing; object tracking

ACM Reference Format:
Henry Ugochukwu Odoemelem and Kristof Van Laerhoven. 2020. A Low-Cost Prototyping Framework for Human-Robot Desk Interaction. In *Adjunct Proceedings of the 2020 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2020 ACM International Symposium on Wearable Computers (UbiComp/ISWC '20 Adjunct), September 12–16, 2020, Virtual Event, Mexico.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3410530.3414323

1 INTRODUCTION

Human-centered robotics and human-robot interaction as research fields have emerged in the past decades in which human users and robots coexist in the same space, both able to manipulate the same physical objects that reside in it. This paper presents an embedded framework for desk-based human-robot interaction to allow robotics enthusiasts and researchers to quickly develop low-cost and rapid prototypes of functioning robot arms, to test out project ideas, for instance for educational purposes or before a full scale implementation. Our framework consists out of hardware components that are widely available, low-cost, easy to replace, and driven by an embedded system that holds all software to track and actuate nearby objects. Software-wise, we mainly use the existing frameworks for computer vision tasks such as detecting objects and tracking them in 3D, interfacing and driving the robot arm.

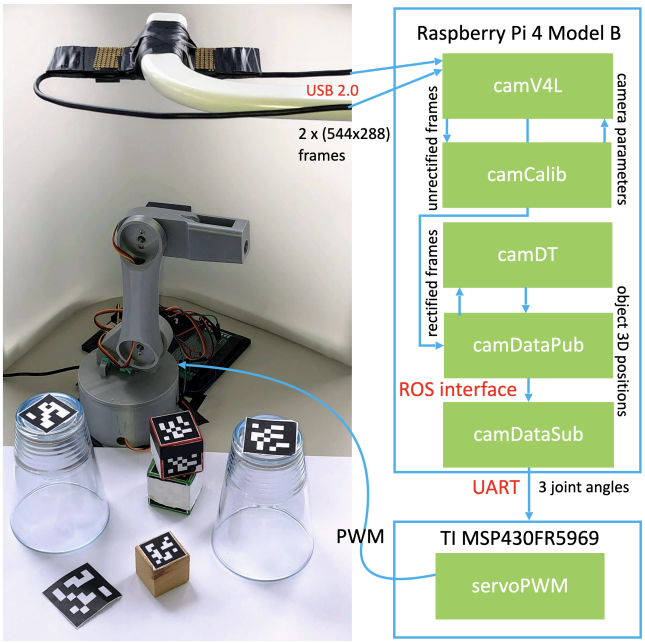

Figure 1: Overview of the tracking and actuation parts of our prototyping framework: Two miniature web cameras provide their frames via USB to the Raspberry Pi 4B, which via OpenCV libraries uses these to perform 3D object tracking. Tracked objects are then merged in the robot's coordinate systems via ROS and can be manipulated by changing the robot arm's joints, which are sent to a microcontroller that maps the joint angles to pulse width modulation (PWM).

Prototyping frameworks have been proposed for several interaction paradigms. For human-robot interactive systems, however, we argue in this paper that especially a low-barrier and low-cost prototyping framework that allows development of responsive interactive systems is still missing. We focus on the design space where tracking of objects is achieved through fiducial markers and a setup with two simple USB cameras, and where actuation is achieved through an open-source robot arm with its joints commanded locally via inverse kinematics. By abstracting from the underlying functionality of openCV and ROS in particular, and by introducing methods that speed up tracking and inverse kinematics on a system with limited resources, we thus contribute in this paper a framework to rapidly build low-cost but fast systems that allow

UbiComp/ISWC '20 Adjunct, September 12–16, 2020, Virtual Event, Mexico

ACM ISBN 978-1-4503-8076-8/20/09.
https://doi.org/10.1145/3410530.3414323

a robot arm and users to interact with objects on a desktop-based area. In the following sections, we situate this work among related research in this area, before we present our framework.

2 RELATED WORK

Early work in this research area included prototypes such as [5], which focused on guiding and teaching robot tasks through an extended digital desk where separated robot and human were made aware of each other through an augmented reality projector display and tracking system. Safety of the human is also an important aspect in such scenarios, as the larger the robot arms get, the higher the chance that a collision with the robot can hurt the user. Approaches to achieve this with industrial robot arms include estimation of external forces and saturation of joint control torques to keep the effective external forces under safety level [6].

Of the many challenges in such a setting, one is how users can program and control robot arms intuitively, with approaches ranging from touch-based programming [1] to using on-robot sensors. Close to our approach are works such as Synthé [3], allowing designers to bodystorm, or act out demonstrations of a robot-human interaction, automatically capturing and translating these into prototypes using program synthesis. Such systems tend to be fast and can deal with complex (e.g., humanoid) robots, but require expensive robots and sensor equipment. At the same time, powerful software libraries and frameworks such as openCV and ROS do exist, but are harder to use in rapid prototyping [4].

3 ARCHITECTURE

Our framework essentially focuses on a set of python classes, but in this section we first describe the hardware to illustrate the use of limited and affordable hardware components. The hardware and software architecture and methodology in a nutshell are shown in Figure 1.

3.1 Hardware components

The hardware cost of our prototype including the stereo-camera setup is under $250, the components include:

An open-source 3D-printed Robot arm. For the robot arm, we use an open-source design 3-DOF RRR robot arm, which can be 3D-printed using STL files available on [1]. We used 3 MG996R servo motors capable of 180 deg rotation and command the motors using the TI MSP430FR5969 microcontroller through pulse width modulation. This design results in an extensible, re-programmable, and low-cost robot arm that costs about $70.

Raspberry Pi-based main module. The Raspberry Pi 4 Model B which costs about $35 has 4 Gigabyte LPDDR4 RAM, and is capable of H.265 (HEVC) hardware decoding (up to 4Kp60). It has 4 USB port which are in our framework required for the two USB cameras used for stereo vision, as well as the USB-connected TI MSP430FR5969 microcontroller. More technical details on Raspberry Pi 4 Model B specifications can be found here [2].

Stereo-camera. To achieve object triangulation by stereo imaging, two Logitech c525 webcams (costs about $60 each) with 69° Field of view (FoV) are used to implement a cost-effective stereo camera in normal case. To reduce errors in the configuration, stereo calibration and rectification were done, this will be discussed in more details in the following section describing the software components. The intrinsic and extrinsic properties of the stereo camera from calibration and rectification were obtained, after which stereo triangulation [2] is performed. 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 are as show in equation (1). To understand the fundamentals of stereo vision, the photogrammetry course by Cyrill Stachniss on [3], is a recommended learning resource.

$$\frac{T - \left(x^l - x^r\right)}{Z - f} = \frac{T}{Z} \Rightarrow Z = \frac{fT}{x^l - x^r} \tag{1}$$

MSP430FR5969 microcontroller. The last components required for our system is a Texas Instruments MSP430 FR5969 LaunchPad (costs about $16), which is used as a lightweight UART to PWM connection with an efficient microcontroller to control the robot arm in software from the Raspberry Pi module.

3.2 Software components

For the tracking of nearby objects, the image frames from the stereo camera are read by the Raspberry Pi; Both cameras are attached via the USB ports. Using the system source code (depicted in green in Figure 1) written in Python3, the Raspberry Pi processes these images to obtain tracking data (3D position of any fiducial markers present in both cameras' frames). The processed image frames and camera properties control trackbar are displayed on the user interface of our framework, which can be accessed by a connected (touch) display, keyboard and mouse, or via a graphical secure shell (SSH) connection. 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 for the robot arm using inverse kinematics. The joint angles are then sent to the TI MSP430FR5969 micro-controller that is attached to the Raspberry Pi unit through UART. On this microcontroller, servoPWM is our Embedded C programmed software that uses the communicated joint angles, a lookup table and local hardware timers to generate corresponding pulse width modulation (PWM) signals. These PWM signals are then sent to the robot arm servo motors to command joint positions, so that the end effector can reach the 3D position of any of the fiducial markers identified. This process ensures that the robot arm is able to track the 3D position of objects with fiducial markers within its workspace.

camV4L: Camera Video for Linux. This module defines the camera image and contains methods to acquire the camera images, specifying their respective resolutions, and to set the camera properties (exposure, contrast, etc.). In our setup, the camera resolutions were both set to $(544, 288)$. The camera intrinsic and extrinsic parameters from calibration are defined here as class attributes, and are used to get rectification maps and projection matrices. A further method allows reading the new camera frames if both cameras are

[1] https://howtomechatronics.com/download/arduino-robot-arm-stl-files/
[2] https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2711/rpi_DATA_2711_1p0_preliminary.pdf
[3] https://www.youtube.com/watch?v=_mOG_lpPnpY

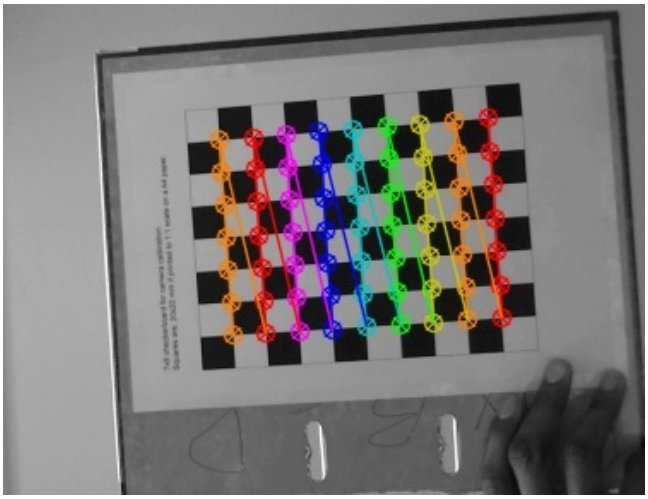

Figure 2: The calibration process is done with a chessboard image, here is the output of the process where the corners in one of the two images are found.

available. It uses rectification maps to rectify the images using the openCV cv2.remap() method, and then returns the rectified images to the calling function.

camCalib: Camera Calibration. Camera calibration is based on the methods described in [2]. For camera calibration, at least 10 images of a 7-by-8 or larger chessboard are required according to [2]. In our hardware setting, at least 50 chessboard images per camera in different orientations and positions were needed for stereo calibration and rectification: A snapShot() function is used to take chessboard images (a 7-by-9 chessboard image was used here) and save them into a file, to be later used for camera calibration. The camera calibration process begins with finding the chessboard corners on the save chessboard images, using the 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. Image and object points are then passed to cv2.calibrateCamera(), to obtain the set of camera intrinsic and extrinsic parameters, by finding the solution to:

$$q = sMWQ \tag{2}$$

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 holds the intrinsic camera parameters (the focal length f_x,f_y and position displacements c_x,c_y away from the optic axis). $W = \begin{bmatrix} R_z(\theta_z)R_y(\theta_y)R_x(\theta_x) & | & \mathbf{t} \end{bmatrix}$ holds the extrinsic camera parameters: the rotation and translation of the object relative to the camera coordinate system. Q is the world/scene coordinate system, and WQ holds the camera coordinate system. cv2.calibrateCamera() also return the radial and tangential distortions of the cameras which are necessary parameters for undistorted rectification.

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 (3), 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 the second equation (3), where the 3D point is $(X/W, Y/W, Z/W)$.

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = P \begin{bmatrix} x \\ y \\ Z \\ 1 \end{bmatrix}, \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} = A \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} \tag{3}$$

The result of the camera calibration and stereo rectification then allows detection and tracking tasks.

camDT: Camera Detect and Track. In this module, we provide a function to simultaneously detect and track fiducial markers, using $arcuoID$ as the ID of the fiducial marker (ArUCo), a tracker id $tractID$, taking flags to signify whether the cameras have new frames available, and the left and right projection matrices. The tracking itself is implemented using the user-selected openCV tracker. In order 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 frames are first copied to a temporary variables and these are then cropped before they are used in the tracking process. These bounding boxes with respect to the main frame then lead to a tracking frame, which is obtained by adding a small margin to the bounding box frame to allow for objects' displacement in the next frame.

If no object (through its fiducial marker) is found in the last iteration because of tracker failure, the original frames are used to detect the current position. Otherwise, the tracking frame is passed to the tracker which then returns the bounding box and tracking frames. Some trackers can erroneously return values after the fiducial marker has been removed from the scene. To avoid this error, the bounding box returned by the tracker is periodically examined 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. We found that depending on your test conditions a bounding box with few contours (typically < 5) and small area (typically << 100) allows for a quick test to see whether the fiducial marker has been removed from the scene. The detected objects' midpoints in pixel coordinates, together with the projection matrices, are then passed to the openCV cv2.triangulatePoints() method, which returns a homogeneous coordinate that represents the world position coordinate (X, Y, Z) of the fiducial marker's midpoint. Next, the bounding box and midpoint position are marked so that they can be displayed on the user interface. The marked frames, bounding boxes and the objects' midpoints in world position coordinate (in cm) are returned.

camDataPub: Camera Data Publisher In this module, a camera image is created to receive the rectified image frames and combined with cv2.createTrackbar() to enable users to control the camera properties such as contrasts, exposure, etc. dynamically. This module also keeps track of all objects that are detected through their fiducial markers that need to be identified before the tracking is triggered. Users can here get access to processed image frames, the bounding boxes, and the midpoint positions (in cm) of objects in world coordinates. The frames per second (fps) and processed images are displayed on the user interface, and then the ROS node "*talker*" is used to publish the position data on the topic "*chatter*".

camDataSub : Camera Data Subscriber This module is responsible for the inverse kinematics, which allows all the robot's joint angles to be calculated, given the position in 3D space of the robot's end effector. The end effector position (3D position of the fiducial marker (ix, iy, EfH)), we obtain $(x, y, EFH0)$, 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 (equations (4) - (7)) using the end effector position. Then by inverse kinematics, the joint angles are calculated and implemented using the orientation and position of the end effector.

$$s = \sqrt{x^2 + y^2} \tag{4}$$

$$r = \sqrt{x^2 + y^2 + EFH0^2} \tag{5}$$

$$\phi = \text{acos}\left(\frac{r^2 + L3^2 - L2^2}{2rL3}\right) + \text{atan2}\left(\frac{L1 - EFH0}{s}\right) \tag{6}$$

$$\psi = 360^\circ - \phi \tag{7}$$

A second function that is provided here is the ROS node "*listener*", which subscribes to the topic "*chatter*", so that objects' positions that are published by the "*talker*" node on topic "*chatter*" are received here. The received positions are then passed to the above-described inverse kinematics function, which returns the joint angles $(\theta_1, \theta_2, \theta_3)$. These joint angles are then sent to TI MSP430FR5969 through the UART as bytes; 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.

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 timer counts) to 180°(2700 timer 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.

4 CASE STUDY

By using the architecture as described above, it is now possible to quickly create interactive behaviours by interfacing with the five modules (implemented as python classes) present in the Raspberry Pi module, abstracting from off the underlying openCV and ROS functionality. The following case study was prototyped in a short amount of time and shows how our framework could be used, and reports on how fast and accurate such built systems can be. The aim of this example is to detect several stages in the typical ball-and-cups game, which requires to track a hidden block as it is covered by a cup and switched with another cup by the user. The goal of the game is for the robot arm to identify the cup that covers the block, by pointing at it, and then track the cup reliably, independent of its orientation until the user stops moving. The attached video on [4], shows the main components and the robot arm in action. For tracking in this scenario, which assumes that 2 different objects need to be tracked in the same scene simultaneously, the image processing time takes typically between 50ms to 33ms (allowing a processing speed of 20 to 30 frames per second). Note that the processing time will increase as higher numbers of objects are introduced in the scene, but that the reach of the robot arm and the fact that the cameras were installed about 50 cm above the desk resulted in a shared work space of about 40 by 50 cm.

5 CONCLUSIONS

Creating interactive systems where both robots and users can manipulate objects on a common desktop space is currently not trivial. Common hurdles include the high costs of the high-fidelity tracking and precise control of off-the-shelf robot arms, as well as a lack of a unifying software framework that ties this all together. This paper has presented our open-source prototyping framework (full system description and source code on [5]), which provides a combination of a simplified interface to a set of open-source software tools, with efficient implementations that allow tracking and inverse kinematics on a Raspberry Pi system with limited resources. This allows our prototyping framework to be run on low-cost hardware components, leading to a less precise yet fast, responsive, and affordable way of prototyping. Recommendations to improve this open-source framework includes robust; kinematics feedback control, motion planning, low-cost 3D printable robot-arm/end effectors, tracker filter, and low-cost stereocamera design.

REFERENCES

[1] G. Grunwald, G. Schreiber, A. Albu-Schaffer, and G. Hirzinger. 2003. Programming by touch: the different way of human-robot interaction. *IEEE Transactions on Industrial Electronics* 50, 4 (2003), 659–666.
[2] Adrian Kaehler and Gary Bradski. 2016. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library* (1st ed.). O'Reilly Media, Inc.
[3] David Porfirio, Evan Fisher, Allison Sauppé, Aws Albarghouthi, and Bilge Mutlu. 2019. Bodystorming Human-Robot Interactions. In *Proceedings of ACM UIST'19* (New Orleans, LA, USA). 479–491.
[4] Philipp M. Scholl, Brahim El Majoub, Silvia Santini, and Kristof Van Laerhoven. 2013. Connecting Wireless Sensor Networks to the Robot Operating System. *Procedia Computer Science* 19 (2013), 1121–1128.
[5] M. Terashima and S. Sakane. 1999. A human-robot interface using an extended digital desk. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, Vol. 4. 2874–2880 vol.4.
[6] A. Vick, D. Surdilovic, and J. Krüger. 2013. Safe physical human-robot interaction with industrial dual-arm robots. In *Robot Motion and Control*. 264–269.

[4] https://youtu.be/dv1DAupmCto
[5] https://github.com/hodoemelem/A-Rapid-Prototyping-Framework-for-Human-Robot-Interaction