Hands-on Perception

Lab #1 - ArUcoMarkers

Delivered by:

Rihab Laroussi, u1100330

Davina Sanghera, u1100402

Supervisor:

Josep Forest

Date of Submission:

25/03/2025

Contents

1 Introduction

This report presents a series of lab assignments that focus on the practical applications of ArUco markers. These labs progressively build upon each other, beginning with generating ArUco markers, followed by detection, camera calibration and finally with the implementation of an augmented reality program.
ArUco markers are square markers that are encoded with bidimensional binary code that allows each marker to be uniquely identified. The applications of AruCo markers that we explore through these labs include pose estimation, camera calibration, object tracking and augmented reality. To implement these functionalities we make use of OpenCV in C++, with a particular focus on its ArUco library, which provides a set of tools for generating, detecting and working with the markers in both static and dynamic environments. Each lab builds upon the previous, incrementally using the more complex functionalities to make a more sophisticated program.

2 Methodology & Results

2.1 Generation of ArUco markers

The basis for this series of labs begins with the generation of ArUco markers using the OpenCV library in C++. We implemented two C++ programs for this purpose, using OpenCV's ArUco library: `generate_marker.cpp` for generating a single marker and `generate_board.cpp` for generating a board of ArUco markers.
The `generate_maker.cpp` program takes four command-line arguments: dictionary ID, marker ID, marker size and output filename. It uses OpenCV's `aruco::drawMarker()` function to draw the specified. Furthermore, we added a white border around the marker so that the white versus black contrast makes the marker's borders easier to detect. The resulting image is then saved as a .png file using the given filename. An example output using dictionary 16 marker ID 11 is displayed in Figure 1.

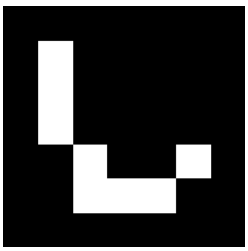

Figure 1: ArUco Marker

The second program, `generate_board.cpp`, takes six command-line arguments: number of rows, number of columns, dictionary ID, marker size, marker separation and output filename. It uses OpenCV's `aruco::GridBoard::create()` function to create a board that consists of the first M markers of the specified dictionary, where M is the number of rows multiplied by the number of columns. To better detect the individual markers, they are arranged in a grid with uniform white spacing, as well as a surrounding white border. An example board output for dictionary 16 is shown in Figure 2.

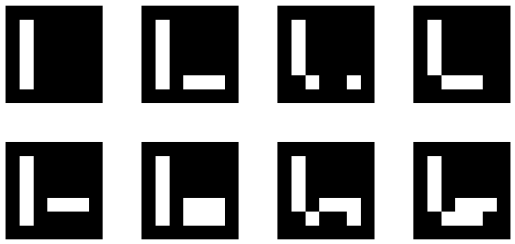

Figure 2: ArUco Board

2.2 Detection of ArUco markers

The next step was to detect ArUco markers in real time using the device's webcam. We implemented the `detect_marker.cpp` program, which overlays key visual indicators onto each detected marker, including a green frame outlining its border, a red square marking its top-left corner and the marker's ID. These overlaid details help us to verify that the markers are successfully detected and identified.

This program only takes the dictionary ID as input. It makes use of the `cv::VideoCapture` class to access the live video stream generated by the default camera. Each video frame is processed using the `aruco::detectMarkers()` function, which identifies all of the markers present in the frame that belong to the specified dictionary. Once the markers are detected, their outlines, corners and ids are drawn using the `aruco::drawDetectedMarkers()` function.

The detection updates in real time and remains stable as the markers move across the camera, verifying the program's successful implementation. An example of the ArUco markers detected in a video frame is shown in Figure 3.

Figure 3: Detected ArUco Markers

2.3 Camera Calibration Using an ArUco Board

Building upon the ArUco marker detection program implemented in Lab 3, this lab focuses on camera calibration using ArUco boards and OpenCV. Camera calibration is a computer vision process that establishes the mathematical relationship between 3D world coordinates and their 2D image projections. The calibration process is implemented in a C++ program, determines both intrinsic camera parameters (including focal length, principal point, and lens distortion coefficients) and extrinsic parameters (defining the camera's position and orientation relative to the calibration board). Initially, the program

defines command-line arguments for specifying the board configuration (number of markers in X×Y arrangement), marker dimensions (length and separation), dictionary specification, camera selection, detector parameters file, minimum frame requirement, and output filename. During operation, the system processes live video frames, employing OpenCV's detection algorithms to identify ArUco markers which are visually highlighted with bounding boxes and IDs.The system continuously monitors for the 'c' key press—upon which it verifies that all expected board markers are visible -to ensure good calibration- and then adds valid frames to the calibration set. The process terminates either when the ESC key is pressed or when sufficient valid frames have been collected.

Once enough valid frames are accumulated, the program concatenates marker corners and IDs from all frames while preserving the frame-to-marker mapping. Calibration execution then begins: using OpenCV's ArUco-specific calibration function, the program estimates the camera's intrinsic parameters (focal lengths and principal point), calculates the lens distortion coefficients, and computes the pose (rotation and translation) for each calibration frame. Finally, the average reprojection error (i.e., the average pixel deviation) is measured, and the final calibration parameters are saved to a YAML file containing the camera matrix (3×3), distortion coefficients (5×1), image dimensions, reprojection error, and a timestamp.

The calibration parameters obtained during the process have been saved in the file output$_c alibration4.yml$. These parameters include the following essential elements:

$$\textbf{Camera Matrix} = \begin{bmatrix} 9.40479 \times 10^2 & 0 & 6.51890 \times 10^2 \\ 0 & 9.44698 \times 10^2 & 3.95309 \times 10^2 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\textbf{Distortion Coefficients} = \begin{bmatrix} 5.1949 \times 10^{-2} \\ -1.7201 \times 10^{-1} \\ -1.3759 \times 10^{-4} \\ 6.4265 \times 10^{-3} \\ 2.1161 \times 10^{-1} \end{bmatrix}$$

$$\textbf{Average Reprojection Error} = 1.4972$$

2.4 Augmented Reality Using ArUco Markers

After we obtained the camera calibration parameters in Lab 4, this lab implements augmented reality applications using ArUco markers and OpenCV, performing marker detection and pose estimation in a video stream captured from a camera. The implementation addresses two fundamental challenges of augmented reality: precise pose estimation of physical markers and accurate projection of 3D virtual objects.

The program begins by loading the camera calibration parameters from a YAML file, which includes the intrinsic camera matrix and distortion coefficients. It then continuously captures frames from the video stream, detects the specified ArUco marker, and estimates its pose using OpenCV's `aruco::estimatePoseSingleMarkers` function. To visually represent the marker's pose, the program

annotates the detected marker with a green bounding frame around the marker, a three-axis coordinate system (X-red, Y-green, Z-blue), and the marker's ID. The coordinate system dynamically updates in real-time to reflect the marker's position and orientation in 3D space. Additionally, the program displays the X, Y, and Z coordinates of the marker's center on the screen, providing numerical feedback about its pose. The estimated pose is represented by a Cartesian coordinate system centered at the marker's origin. This system consists of three orthogonal axes: a red axis (X-axis) aligned with one side of the marker, a green axis (Y-axis) perpendicular to the X-axis within the marker's plane, and a blue axis (Z-axis) normal to the marker's plane. These axes provide an intuitive visualization of the marker's orientation and position in 3D space, updating seamlessly as the marker moves or rotates in the video stream.

Below is the representative screenshot of the program output: We run the command line: `./pose_estimation -d=16 -id=23 -l=0.05 -calib=output_calibration4.yml`

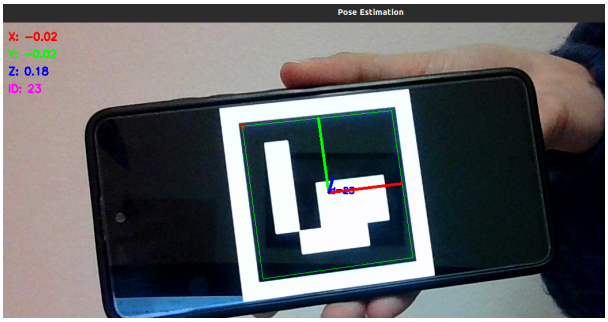

Figure 4: Pose Estimation

Shows the detected marker with a green bounding frame, red corner indicator, marker ID, and a three-axis coordinate system dynamically updating its position and orientation. Next, we use the selected ArUco marker to 'draw' a simple cube object on top of them, maintaining proper perspective as the camera moves. The program begins by parsing command-line arguments to specify the ArUco dictionary and the actual marker length in meters. It then opens the video stream and loads the camera calibration parameters (camera matrix and distortion coefficients) from a YAML file. The ArUco marker used is generated from the first lab program, with the addition of a white contour around it to ensure a proper detection of the marker. The program detects the ArUco marker in each frame, estimates its pose using OpenCV's `aruco::estimatePoseSingleMarkers` function, and overlays a 3D wireframe cube on top of the marker. The cube is drawn using OpenCV's line() function to render its edges, with its vertices positioned relative to the marker's pose in 3D space. This ensures that the cube appears affixed to the marker and maintains correct alignment and perspective as the marker moves or rotates within the camera's field of view. The annotated frames are written to a video file `(draw_cube.avi)`. This process exemplifies an augmented reality (AR) application, where a virtual 3D object (the cube) is imposed onto a physical object (the ArUco marker). The output demonstrates dynamic updates in real-time, accurately reflecting the marker's position and orientation in the scene.

The result is shown in the figure below.

We run the command: `./draw_cube -d=16 -l=0.05`

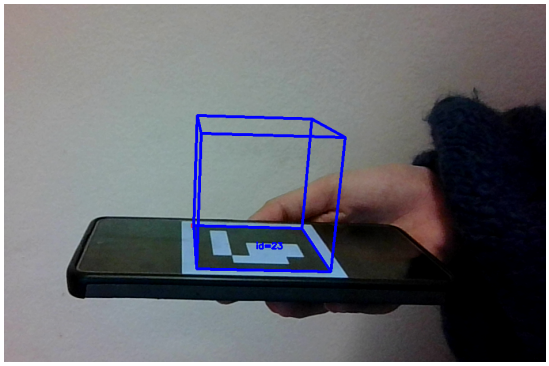

Figure 5: Draw Cube Result1.

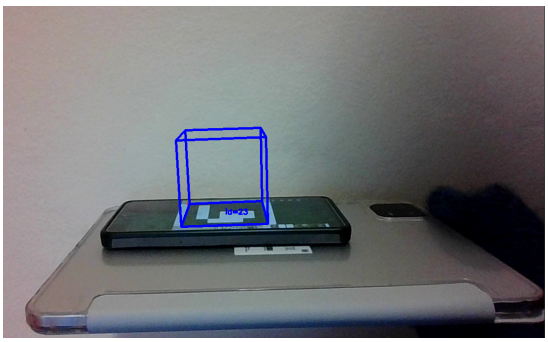

Figure 6: Draw Cube Result2.

3 Conclusion

To conclude, this series of labs demonstrated the diverse applications of computer vision techniques, using OpenCV tools. The primary focus was on understanding ArUco markers and their practical uses, beginning with generating individual markers and boards of markers. This was followed by learning how to detect the ArUco markers, a process that enabled the capability to analyze real-time video streams, identify objects, and estimate their poses. Subsequently, we explored the use of the ArUco board for camera calibration, an essential step in robotics to achieve precise spatial measurements. Such calibration plays a vital role in tasks like 3D reconstruction and robotic navigation, ensuring accuracy and reliability in real-world applications. The labs culminated in an introduction to augmented reality, where the techniques developed earlier were employed to merge the real and virtual worlds in a credible manner. This was accomplished by solving the challenges of detecting the pose of scene objects and projecting 3D virtual objects into an image scene using fiducial markers. It was a crucial exercise to learn how augmented reality leverages computer vision to transform various fields by creating an immersive mix of reality and virtuality. In essence, these implementations present the vast potential of computer vision in revolutionizing industries, ranging from healthcare to entertainment. By addressing complex problems and enhancing user experiences, computer vision technologies illustrate their transformative impact in shaping the future of innovation.

4 References

- OpenCV Documentation: https://docs.opencv.org/4.x/d2/d64/tutorial_table_of_content_objdetect.html