*CS5330: Pattern Recognition and Computer Vision*

Project 4: Calibration and Augmented Reality

Hardik Devrangadi [devrangadi.h@northeastern.edu](mailto:devrangadi.h@northeastern.edu)

Thejaswini Goriparthi [goriparthi.t@northeastern.edu](mailto:goriparthi.t@northeastern.edu)

**Camera Calibration and Augmented Reality**

This project aims to perform Camera Calibration and use the intrinsic parameters to process the image sequence output from a webcam, to isolate a known pattern “the chessboard” placed in view. The points are projected as 3D points so that they can be used to place a 3D object or wireframe that tracks with movement (translation and rotation). Robust features are detected (Harris Corners) and analysis of the detected features is done.

**Tasks**

**Task 1: Detect and Extract Chessboard Corners**

This program extracts and displays the corners of a chessboard pattern in a certain video frame. The frame, the size of the pattern, and a vector to store the extracted corners are sent to the extractChessboardCorners function. It extracts the corners using the findChessboardCorners function and then uses the cornerSubPix method to adjust the placements. It returns true if the corners are correctly extracted and false otherwise.

Using the current frame and the corners vector, the extractChessboardCorners method is called in the main body of the code. If the method returns true, a duplicate of the frame has the corners painted using the drawChessboardCorners function (in this case, frame2 is the clone of the original frame).

**Task 2: Select Calibration Images**

This code defines a function generateChessboard3DPoints that takes a cv::Size object patternSize as input and returns a vector of 3D points that represent the corners of a chessboard. The function iterates over the rows and columns of the chessboard and creates a cv::Vec3f object for each corner, with the first and second elements representing the x and y coordinates, and the third element set to zero.

The initial width and height of the patternSize object are 8 and 6, respectively, to match the dimensions of an 8x6 square chessboard.

To create the 3D points for the chessboard, the generateChessboard3DPoints function is used in the main code. The corners vector and points3D vector are recorded to the imagePoints and objectPoints vectors, respectively, along with the relevant frame in calibration images, if the extractChessboardCorners method locates the chessboard corners in the current frame. If the chessboard corners cannot be located, the console prints a notice.

****

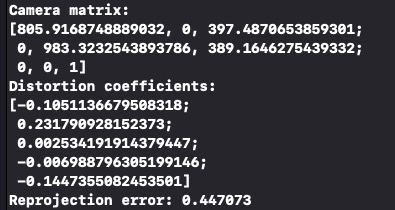
**Figure 1**: A calibration image highlighting the found chessboard corners

**Task 3: Calibrate the Camera**

This function calibrateFrame takes a single frame, a set of object points, and a set of image points, and outputs the camera matrix and distortion coefficients. It uses the calibrateCamera function provided by OpenCV to perform the calibration.

The camera matrix and distortion coefficients are initialized to default values, and then calibrateCamera is called with the provided inputs. rvecs and tvecs are output parameters, but they are not used in this function.

If the number of image points is greater than 5, the function calibrateFrame is called with the inputs frame2, objectPoints, imagePoints, cameraMatrix, and distCoeffs. Finally, it saves the camera matrix and distortion coefficients to a file using cv::FileStorage, and also prints them to the console. This function is used in a loop that processes a series of frames to perform camera calibration.

****

**Figure 2**: A screenshot of the console showing the camera intrinsics

**Task 4: Calculate Current Position of the Camera**

The camera matrix and distortion coefficients are loaded from a file named "intrinsics.xml". The camera matrix describes the relationship between the 3D world coordinates and the 2D image coordinates, while the distortion coefficients describe lens distortion effects.

Then, the solvePnP function is used to estimate the pose of the camera. The solvePnP function takes in a set of 3D object points and their corresponding 2D image points, along with the camera matrix and distortion coefficients. It then estimates the rotation vector (rvec) and translation vector (tvec) that describe the pose of the camera with respect to the object coordinate system.

Finally, the rotation and translation vectors are printed out to the console.

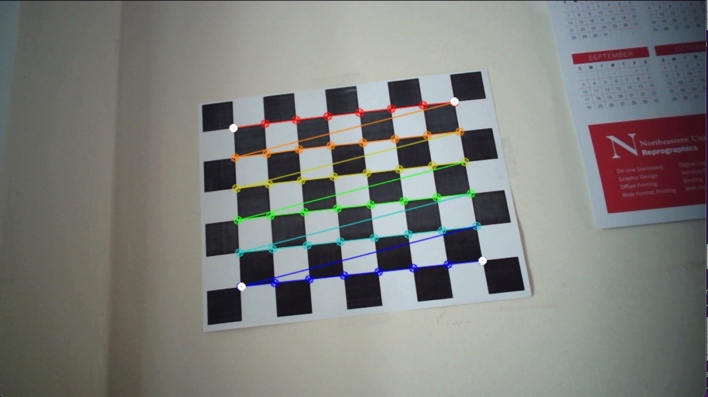


**Figure 3**: A screenshot of the console showing Rotation and Translation Vectors

**Task 5: Project Outside Corners or 3D Axes**

The code starts with a loop that iterates over the four outside points of the chessboard and projects them onto the image plane using camera calibration parameters (rotation vector, translation vector, camera matrix, and distortion coefficients) and passing them into the cv::projectPoints() function. The projected points are then drawn as white circles on the image using the cv::circle() function.

Next, six lines are drawn to connect the projected points and form a quadrilateral using the cv::line() function.

****

**Figure 4**: Projection of 3D points of the four corners of the chessboard on the image plane (shown by white dots)

**Task 6: Create a Virtual Object**

The function octahedronPlot is to plot an octahedron on the frame image using the octaPoints and objectPoints vectors, which contain the 3D coordinates of the octahedron's vertices. The function uses the solvePnP function from OpenCV to determine the rotation and translation vectors needed to project the 3D octahedron onto the 2D image plane. It then uses the projectPoints function to project the 3D points onto the 2D image plane and draws lines connecting the projected points to form the octahedron shape. The function draws circles at each of the projected points for indication.

In the main function, if the size of imagePoints is greater than 5, the current frame is calibrated and and the octahedronPlot function is executed for the obtained camera matrix, distCoeffs.

****

**Figure 5**: An Octahedron placed as a virtual object on top of the chessboard

**Task 7: Detect Robust Features**

The basic idea behind KNN is to find the K nearest neighbors in the training set that are closest to a given image under camera and assign the majority class of these neighbors to the current image. K is a hyperparameter that can be tuned to improve classifier performance. The value of K is taken as a user input and considered for classification of images. The value of K depends on the number of times a specific object is trained.

For KNN classifier with K>1, we train the object multiple times in different orientations making it more efficient. The function findMostFrequent is used to find the most common label among a vector of strings. It uses a map to count the frequency of each string and returns the string with the highest frequency. This function is used to compute the most common label among the K nearest neighbors. The classifyFeatures() function takes a feature vector and K and D (distance metrics) values to classify objects in frames. It uses a list of training features stored in a CSV file to find the K nearest neighbors of the input feature vector and returns the most frequent object label among those neighbors as the detected class.

**A board game on a table

Description automatically generated with low confidence**

**Figure 6**: Harris Corners being detected on a QR Code pattern

**Extensions**

Three extensions have been added to this project. More than 10 items have been added to the database, the classifier is able to classify 15 different objects. The classifier can be run by computing either of the two distance metric implementations, Manhattan distance metric, or Euclidean distance metric. The classifier can also classify multiple images at a time in a single frame.

**Extension 1: Comparing Calibrations of Different Cameras**

The feature vectors of more than the required 10 classes have been saved in the database file. The classifier can classify objects across 15 classes, namely tumbler, wallet, clip, highlighter, speaker, straw, headphones, trimmer, keychain, spoon, harddisk, sunglasses, cardholder, case, and mug.

**Extension 2: Multiple Target Detection**

There are two distance metric implementations, Manhattan distance metric and Euclidean distance metric. When the program is first run, it requests the user to enter the desired distance metric that must be used for classification. The user can enter ‘e’ for Euclidean distance metric to be used or ‘m’ for Manhattan distance metric.

**Extension 3: Target Modification**

First we calculate the minimum and maximum x and y coordinates of the chessboard corners and use them to define the destination (dstpoints) and source (srcpoints) points for perspective transformation.

The findHomography function calculates the homography matrix (h) to map the source points to the destination points. Then, the warpPerspective function warps the dst image using the homography matrix h and the size of the destination image (dstpic.size()), and stores the result in dstpic.

Finally, the code checks the value of the flag variable and, if it is true, replaces the non-black pixels of the input image (frame2) with the corresponding pixels in dstpic, effectively blending the input and warped images.

**Project Learnings and Insights**

Object recognition is a popular area of computer vision where the goal is to identify objects in an image or video. There are various techniques for object recognition, including methods based on deep learning and traditional methods based on image processing. In this project, the goal was to recognize 2D objects invariably in displacement, scaling, and rotation from the camera looking down. The method includes thresholding of the input video, cleaning the binary image, segmenting the image into regions, computing features for each key region, collecting training data, and classifying new images.

Additionally, we gained hands-on experience in collecting and labeling data for use in training and classification models. Overall, this project provided a valuable learning experience in key highlights of computer vision.

**Acknowledgements and Resources**

We would like to acknowledge Professor Bruce Maxwell and all the Teaching Assistants for their valuable insights and support throughout the project.

* OpenCV Tutorials: <https://docs.opencv.org/4.5.1/index.html>
* Morphological Operations: <https://medium.com/@rajilini/morphological-operations-in-image-processing-using-opencv-and-c-8580de272606>
* Segmentation Algorithm: <https://towardsdatascience.com/implementing-a-connected-component-labeling-algorithm-from-scratch-94e1636554f>
* Shape Matching using Hu moments: <https://learnopencv.com/shape-matching-using-hu-moments-c-python/>
* Li, Tt., Jiang, B., Tu, Zz., Luo, B., Tang, J. (2015). Image Matching Using Mutual k-Nearest Neighbor Graph. In:, et al. Intelligent Computation in Big Data Era. ICYCSEE 2015. Communications in Computer and Information Science, vol 503. Springer, Berlin, Heidelberg. <https://doi.org/10.1007/978-3-662-46248-5_34>