CS 5330: Pattern Recognition and Computer Vision (Spring 2024)

Report

Project 4: Calibration and Augmented Reality

Submitted by: Haard Shah
Hrigved Suryawanshi

Description:

In this project, we will calibrate a camera and use the calibration results to generate virtual objects in a scene, creating an augmented reality (AR) experience. We will use a checkerboard pattern, such as the provided "checkerboard.png," as a target, which can be displayed on a screen or printed out. The project involves capturing multiple images of the checkerboard to estimate the camera's intrinsic parameters and lens distortion. Once calibrated, we will use the checkerboard to estimate the target's pose and place virtual objects relative to it. As the camera or target moves, the program will continuously update the virtual object's position and orientation. This project provides valuable insights into camera calibration, computer vision, and AR technologies with potential applications in various fields.

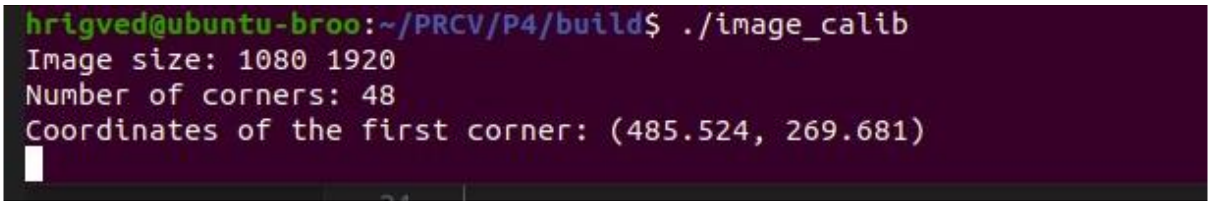
Tasks:

1. Detect and Extract Target Corners

In this we have chosen to use the target as checkerboard pattern on an iPad. The target detection and corner extraction process involved using OpenCV functions namely "findChessboardCornersLinks" & "cornerSubPixLinks". These functions allow for accurate detection and extraction of the marker corners in various orientations and conditions.

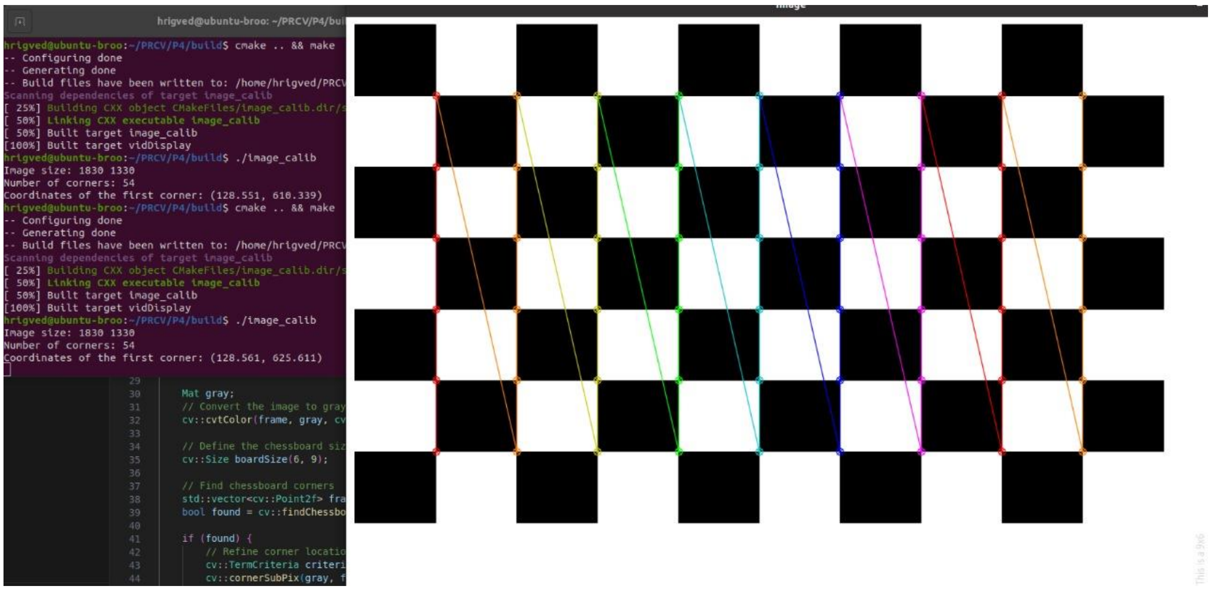
To provide visual feedback and ensure the correct operation of the corner detection system, I have implemented features such as drawing the detected target corners on the video and printing the number of corners found along with the coordinates of the first corner. This feedback is essential for debugging and verifying the accuracy of the corner detection system.

However, there are certain limitations and challenges faced by the system in locating the target. For instance, poor lighting conditions, partial occlusion of the marker, or the presence of similar patterns in the scene can negatively impact the detection process.



```
hrigved@ubuntu-broo:~/PRCV/P4/build$ ./image_calib
Image size: 1080 1920
Number of corners: 48
Coordinates of the first corner: (485.524, 269.681)
```

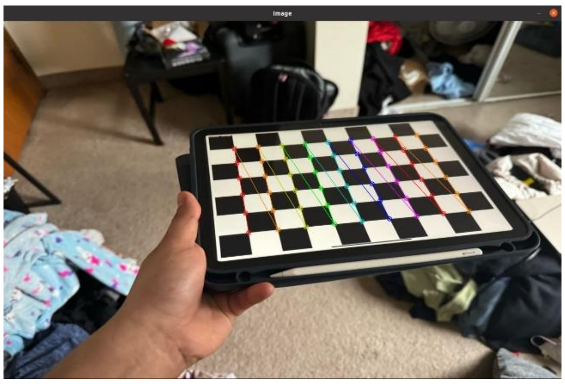
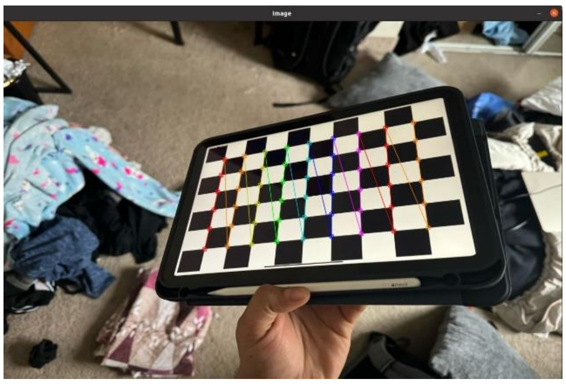
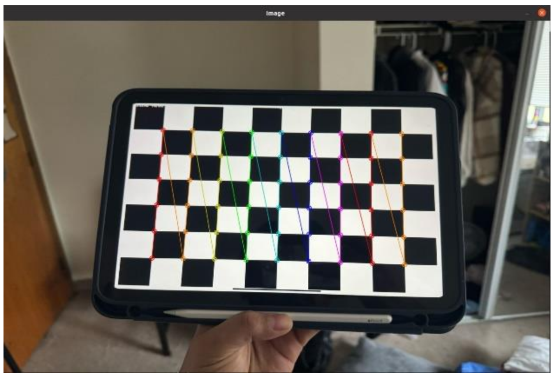
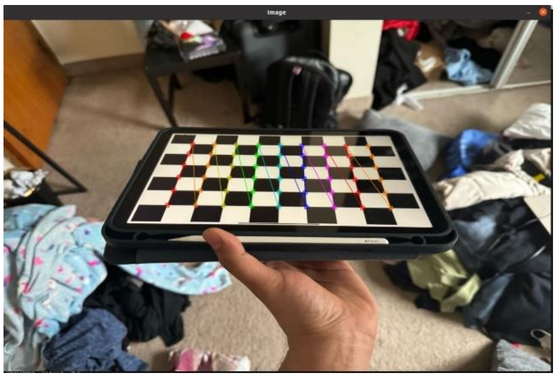
Figure 1: Visual Feedback in terms of Number of corners and Coordinates



```
hrigved@ubuntu-broo: ~/PRCV/P4/bui
hrigved@ubuntu-broo:~/PRCV/P4/build$ cmake .. && make
-- Configuring done
-- Generating done
-- Build files have been written to: /home/hrigved/PRCV
Scanning dependencies of target image_calib
[ 25%] Building CXX object CMakeFiles/image_calib.dir/s
[ 50%] Linking CXX executable image_calib
[ 50%] Built target image_calib
[100%] Built target vidDisplay
hrigved@ubuntu-broo:~/PRCV/P4/build$ ./image_calib
Image size: 1830 1330
Number of corners: 54
Coordinates of the first corner: (128.551, 610.339)
hrigved@ubuntu-broo:~/PRCV/P4/build$ cmake .. && make
-- Configuring done
-- Generating done
-- Build files have been written to: /home/hrigved/PRCV
Scanning dependencies of target image_calib
[ 25%] Building CXX object CMakeFiles/image_calib.dir/s
[ 50%] Linking CXX executable image_calib
[ 50%] Built target image_calib
[100%] Built target vidDisplay
hrigved@ubuntu-broo:~/PRCV/P4/build$ ./image_calib
Image size: 1830 1330
Number of corners: 54
Coordinates of the first corner: (128.561, 625.611)
```

```cpp

    Mat gray;
    // Convert the image to gray
    cv::cvtColor(frame, gray, cv

    // Define the chessboard siz
    cv::Size boardSize(6, 9);

    // Find chessboard corners
    std::vector<cv::Point2f> fra
    bool found = cv::findChessbo

    if (found) {
        // Refine corner locatio
        cv::TermCriteria criteri
        cv::cornerSubPix(gray, f
```

Figure 2: Checkerboard Image and corner detection

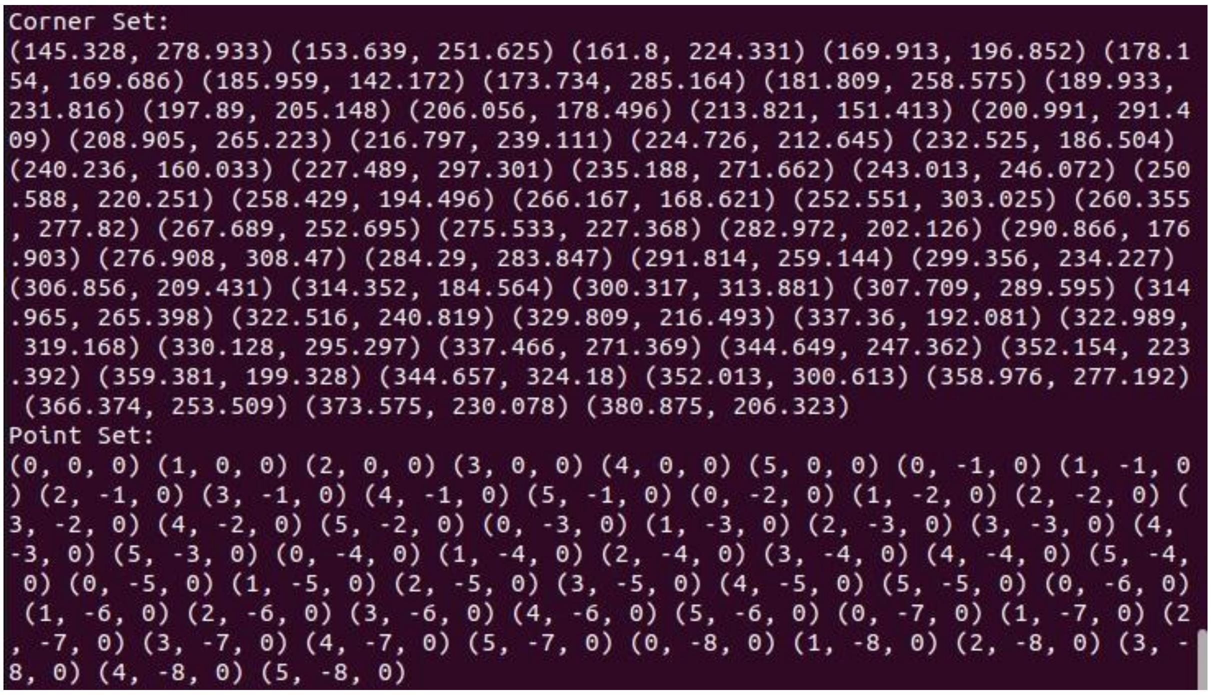
	
	

The above images showcase the successful detection of the checkerboard patterns and the extraction of their corner points.

2. **<u>Select Calibration Images</u>**

In this we have described the process of collecting and storing corner locations and their corresponding 3D world points after user selects a particular image for camera calibration. We gave the user the ability to specify that a particular image should be used for calibration by typing a specific key, such as 's' and the system stores the vector of corners found by the most recent successful target detection in a “corner_list”. To handle cases where the target is not found in the image when the user presses 's', we save the most recent image and target corners in variables. We also created a std::vector called point_set to store the 3D positions of the corners in world coordinates, which remains constant regardless of target orientation.

To build the 3D world point set, we measured the world in units of target squares and assigned coordinates to the chessboard target corners accordingly. Ensuring the consistency and accuracy of the data is crucial for successful camera calibration, so we made sure the number of points in the 3D world point_set matches the number of corners in the corner_set, and there are as many point_sets in the point_list as there is corner_sets in the corner_list. One common mistake when working with the chessboard target is swapping the number of rows and columns in the point_set, which can lead to incorrect calibration results. I stored the images used for calibration and included a calibration image with highlighted chessboard corners in my project report to provide visual evidence of the target detection and corner extraction process, demonstrating the successful identification of the chessboard pattern and the accurate localization of its corner points.



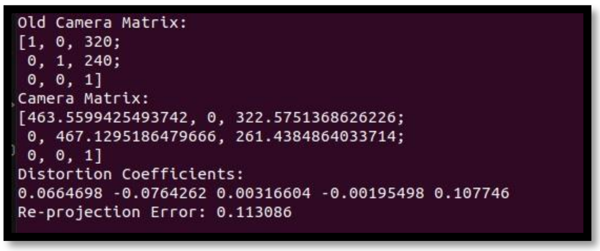
```
Corner Set:
(145.328, 278.933) (153.639, 251.625) (161.8, 224.331) (169.913, 196.852) (178.1
54, 169.686) (185.959, 142.172) (173.734, 285.164) (181.809, 258.575) (189.933,
231.816) (197.89, 205.148) (206.056, 178.496) (213.821, 151.413) (200.991, 291.4
09) (208.905, 265.223) (216.797, 239.111) (224.726, 212.645) (232.525, 186.504)
(240.236, 160.033) (227.489, 297.301) (235.188, 271.662) (243.013, 246.072) (250
.588, 220.251) (258.429, 194.496) (266.167, 168.621) (252.551, 303.025) (260.355
, 277.82) (267.689, 252.695) (275.533, 227.368) (282.972, 202.126) (290.866, 176
.903) (276.908, 308.47) (284.29, 283.847) (291.814, 259.144) (299.356, 234.227)
(306.856, 209.431) (314.352, 184.564) (300.317, 313.881) (307.709, 289.595) (314
.965, 265.398) (322.516, 240.819) (329.809, 216.493) (337.36, 192.081) (322.989,
 319.168) (330.128, 295.297) (337.466, 271.369) (344.649, 247.362) (352.154, 223
.392) (359.381, 199.328) (344.657, 324.18) (352.013, 300.613) (358.976, 277.192)
 (366.374, 253.509) (373.575, 230.078) (380.875, 206.323)
Point Set:
(0, 0, 0) (1, 0, 0) (2, 0, 0) (3, 0, 0) (4, 0, 0) (5, 0, 0) (0, -1, 0) (1, -1, 0
) (2, -1, 0) (3, -1, 0) (4, -1, 0) (5, -1, 0) (0, -2, 0) (1, -2, 0) (2, -2, 0) (
3, -2, 0) (4, -2, 0) (5, -2, 0) (0, -3, 0) (1, -3, 0) (2, -3, 0) (3, -3, 0) (4,
-3, 0) (5, -3, 0) (0, -4, 0) (1, -4, 0) (2, -4, 0) (3, -4, 0) (4, -4, 0) (5, -4,
 0) (0, -5, 0) (1, -5, 0) (2, -5, 0) (3, -5, 0) (4, -5, 0) (5, -5, 0) (0, -6, 0)
 (1, -6, 0) (2, -6, 0) (3, -6, 0) (4, -6, 0) (5, -6, 0) (0, -7, 0) (1, -7, 0) (2
, -7, 0) (3, -7, 0) (4, -7, 0) (5, -7, 0) (0, -8, 0) (1, -8, 0) (2, -8, 0) (3, -
8, 0) (4, -8, 0) (5, -8, 0)
```

Figure 3: List of corners and point sets

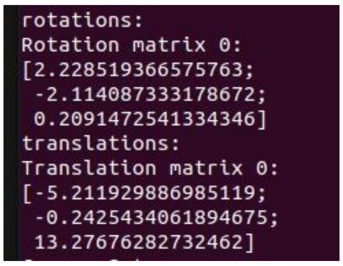
3. **<u>Calibrate the Camera</u>:**

In this task, I will discuss the camera calibration process using the cv::calibrateCamera function with a continuous calibration approach. Users are required to select at least five frames for calibration, although the more frames selected, the better the calibration results. The function takes several parameters, including point_list, corner_list, image size, camera_matrix, distortion_coefficients, rotations, and translations. I used the CV_CALIB_FIX_ASPECT_RATIO flag and initially turned off radial distortion by setting the distortion coefficients to a std::vector<double> of zero length.

After performing the calibration, I printed the camera matrix and distortion coefficients before and after the process, along with the final re-projection error. A successful calibration is indicated by a low per-pixel error, which is essential for accurate augmented reality applications. The continuous calibration approach allows users to keep pressing 'S' to add more frames for calibration, resulting in better calibration accuracy. However, after 5-6 images, there is only a slight improvement in calibration accuracy as it has already reached a good level of calibration.

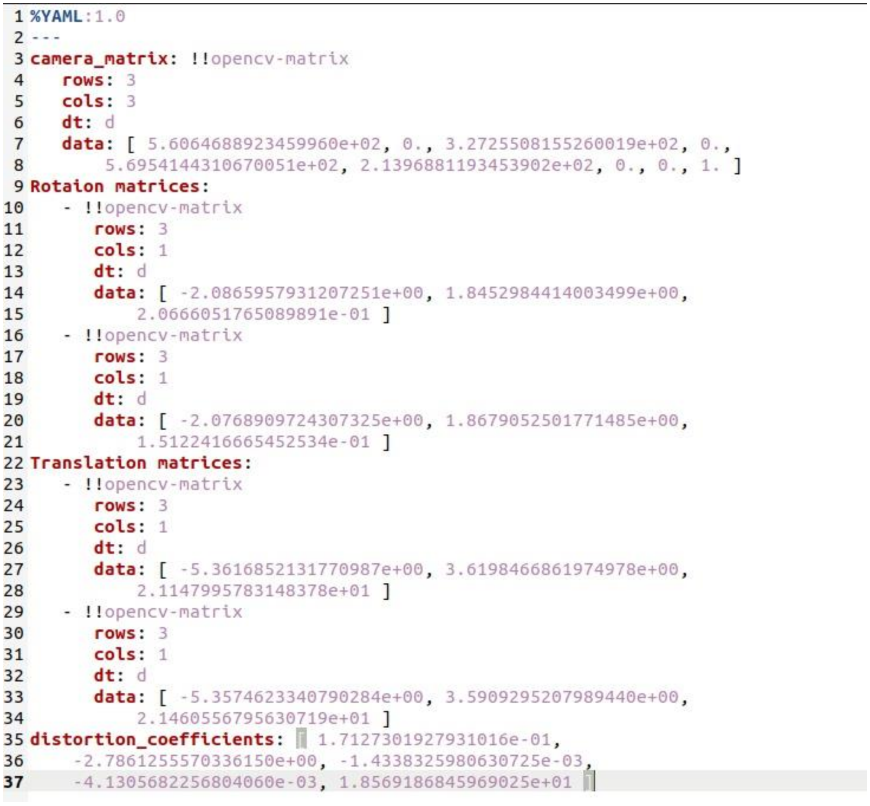


```
Old Camera Matrix:
[1, 0, 320;
 0, 1, 240;
 0, 0, 1]
Camera Matrix:
[463.5599425493742, 0, 322.5751368626226;
 0, 467.1295186479666, 261.4384864033714;
 0, 0, 1]
Distortion Coefficients:
0.0664698 -0.0764262 0.00316604 -0.00195498 0.107746
Re-projection Error: 0.113086
```



```
rotations:
Rotation matrix 0:
[2.228519366575763;
 -2.114087333178672;
 0.2091472541334346]
translations:
Translation matrix 0:
[-5.211929886985119;
 -0.2425434061894675;
 13.27676282732462]
```

Figure 4: Matrices with presented parameters and Reprojection Error value



```
%YAML:1.0
---
camera_matrix: !!opencv-matrix
   rows: 3
   cols: 3
   dt: d
   data: [ 5.6064688923459960e+02, 0., 3.2725508155260019e+02, 0.,
       5.6954144310670051e+02, 2.1396881193453902e+02, 0., 0., 1. ]
Rotaion matrices:
   - !!opencv-matrix
      rows: 3
      cols: 1
      dt: d
      data: [ -2.0865957931207251e+00, 1.8452984414003499e+00,
          2.0666051765089891e-01 ]
   - !!opencv-matrix
      rows: 3
      cols: 1
      dt: d
      data: [ -2.0768909724307325e+00, 1.8679052501771485e+00,
          1.5122416665452534e-01 ]
Translation matrices:
   - !!opencv-matrix
      rows: 3
      cols: 1
      dt: d
      data: [ -5.3616852131770987e+00, 3.6198466861974978e+00,
          2.1147995783148378e+01 ]
   - !!opencv-matrix
      rows: 3
      cols: 1
      dt: d
      data: [ -5.3574623340790284e+00, 3.5909295207989440e+00,
          2.1460556795630719e+01 ]
distortion_coefficients: [ 1.7127301927931016e-01,
    -2.7861255570336150e+00, -1.4338325980630725e-03,
    -4.1305682256804060e-03, 1.8569186845969025e+01 ]
```

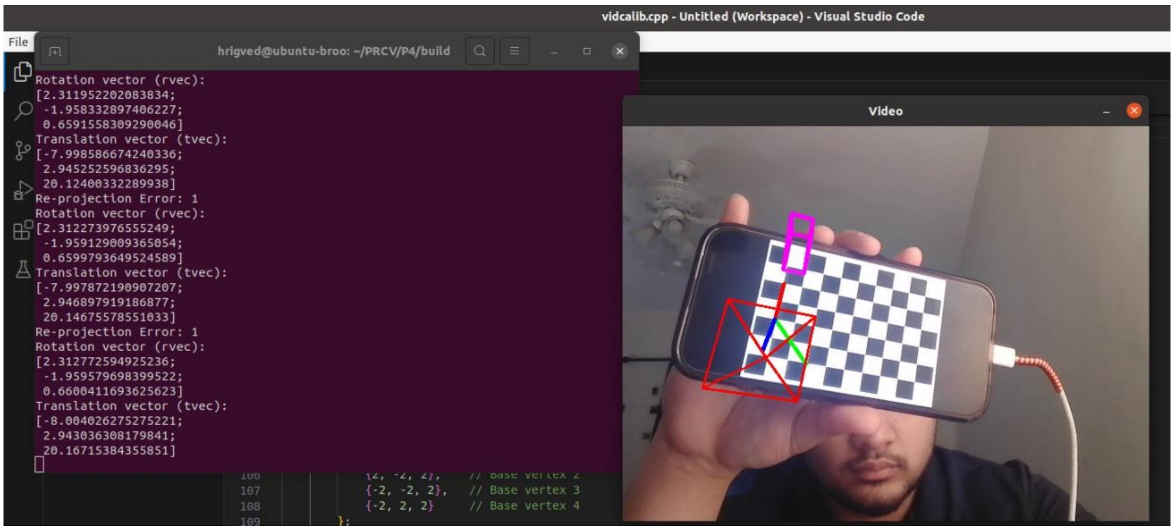
Figure 5: Matrix values

4. Calculate Current Position of the Camera:

In this task, we developed a program that performs real-time target detection and camera pose estimation using camera calibration parameters. The program reads the calibration parameters from a file and processes video frames to detect a target. Upon successful detection, the program extracts the corner locations and employs the "solvePnP" function to compute the board's pose, which includes rotation and translation data.

To evaluate the accuracy of the pose estimation, the program continuously prints the rotation and translation data as the camera moves horizontally. As the camera moves side to side, significant changes should be observed in the translation values along the X-axis. Similarly, the rotation values around the Y-axis should also change, reflecting the camera's new orientation.

By analyzing these changes in real-time, we can assess the accuracy of the rotation and translation data based on the camera's position and orientation. This evaluation is crucial for ensuring accurate tracking of the camera's pose in augmented reality applications. Our program provides an effective solution for real-time target detection and camera pose estimation, enabling reliable and precise augmented reality experiences.


vidcalib.cpp - Untitled (Workspace) - Visual Studio Code
File
hrigved@ubuntu-broo: ~/PRCV/P4/build
Video

```
Rotation vector (rvec):
[2.311952202083834;
 -1.958332897406227;
 0.6591558309290046]
Translation vector (tvec):
[-7.998586674240336;
 2.945252596836295;
 20.12400332289938]
Re-projection Error: 1
Rotation vector (rvec):
[2.312273976555249;
 -1.959129009365054;
 0.6599793649524589]
Translation vector (tvec):
[-7.997872190907207;
 2.946897919186877;
 20.14675578551033]
Re-projection Error: 1
Rotation vector (rvec):
[2.312772594925236;
 -1.959579698399522;
 0.6600411693625623]
Translation vector (tvec):
[-8.004026275275221;
 2.943036308179841;
 20.16715384355851]
```

```
    {-2, -2, 2},   // Base vertex 3
    {-2, 2, 2}     // Base vertex 4
};
```

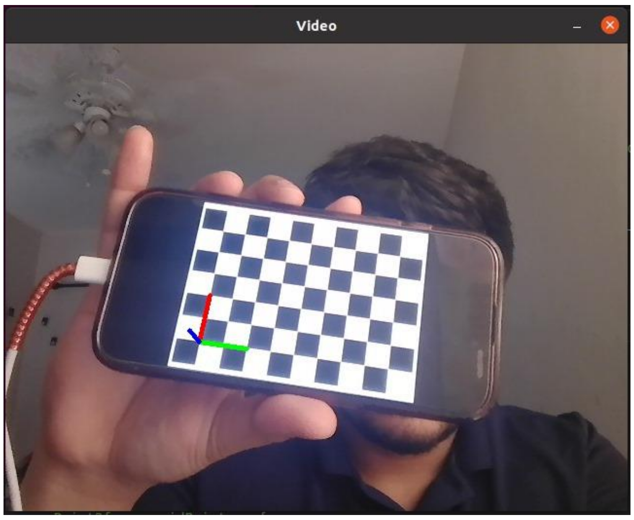
Figure 6: Translation and Rotation on live feed

5. Project Outside Corners or 3D Axes

In this program feature, we will discuss the real-time projection of 3D points corresponding to at least four corners of the target onto the image plane, as the target or camera moves. This feature enables us to verify the accuracy of the estimated pose and enhance our understanding of the target's orientation in the scene.

To accomplish this, we utilize the "projectPoints" function to project the 3D points onto the image plane. The function considers the rotation matrix, translation matrix, distortion parameters, and calibration matrix to accurately project the points. Alternatively, the program can also display 3D axes attached to the target's origin, providing a visual representation of the target's pose.

We use the “cv.line” function to draw the axis lines between two points in the image plane. By displaying the reprojected points or 3D axes, we can visually confirm if they appear in the correct positions as the target or camera moves. This visual confirmation ensures accurate projection and enhances our understanding of the target's pose in the scene.



6. Create a Virtual Object

In this task, we present a virtual object created in 3D world space, floating above the target board and composed of interconnected lines between 3D points. We start with 2-3 lines and gradually increase complexity, designing an asymmetrical object to aid in debugging. This object is more intricate than a simple cube.

To maintain the correct orientation of the virtual object as the camera moves, we utilize the "projectPoints" function. This function accurately transforms the endpoints of each line from world space to image space by using the rotation matrix, translation matrix, distortion parameters, and calibration matrix. Once the endpoints are in image space, we draw a line between them to create the virtual object.

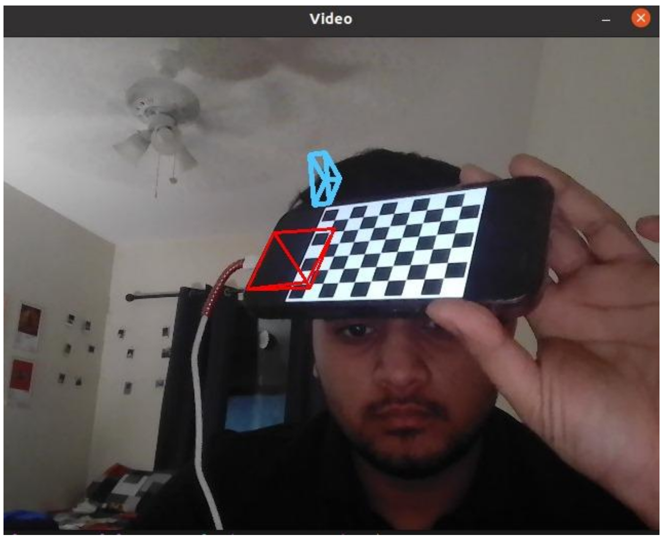


Figure7: Visual demonstration showcasing the stability and accurate orientation of the symmetric object as the camera moves around it

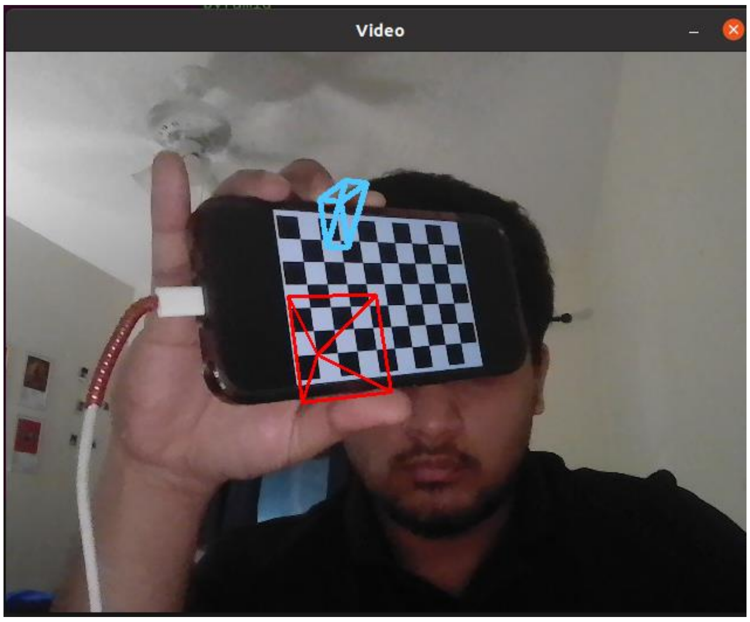
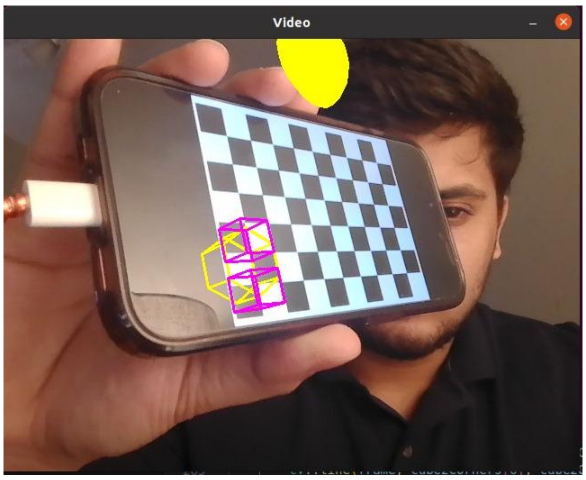
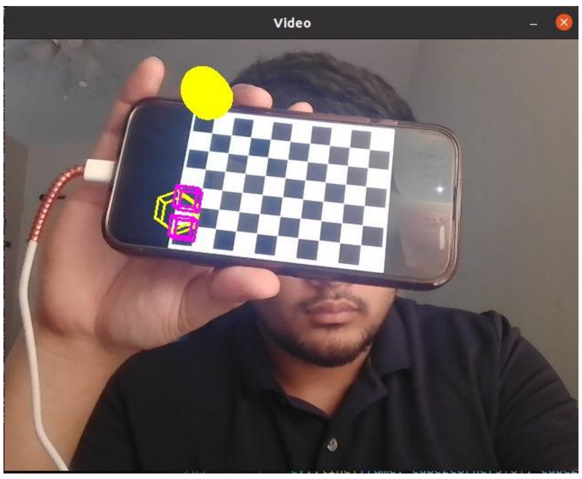


Figure 8: Symmetric





7. Detect Robust Features

In this task, we will develop a separate program to detect and display a specific feature Harris corner in a video stream. We will create a unique pattern and showcase where these features appear on the pattern.

To gain a better understanding of how the features work, we will experiment with different thresholds and settings. This process will help us identify the optimal parameters for accurate feature detection and tracking.

We will explain how these feature points can be used as the basis for incorporating augmented reality into the image. By tracking the movement and orientation of these feature points, we can accurately overlay virtual objects onto the real-world scene, creating a seamless and immersive augmented reality experience.

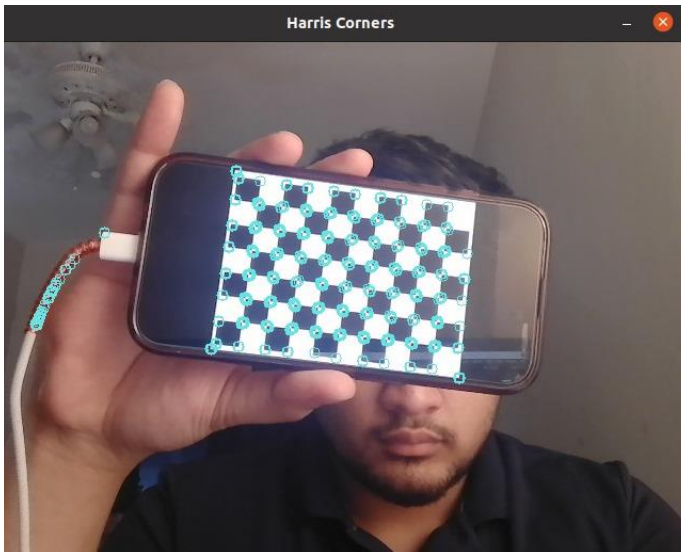
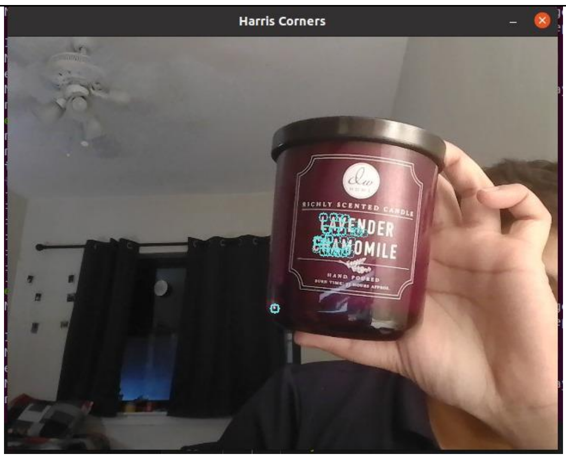
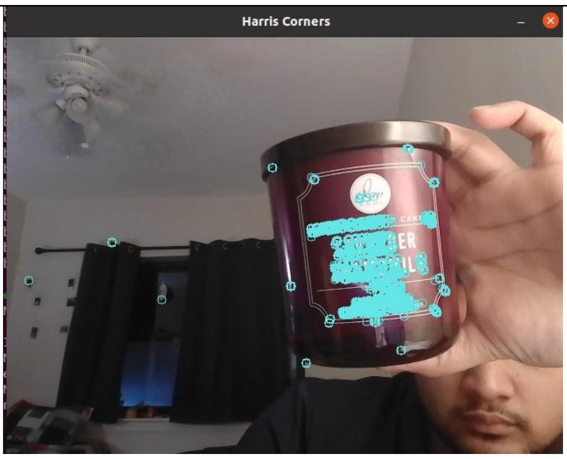


Figure 9: Performance on Checkerboard

	
K= 0.1	K = 0.4
Figure: Performance on Irregular Circular box	

From the images above, we can observe two notable differences as the threshold value increases. With a higher threshold, the algorithm extracts more features, providing a more detailed representation of the object. However, the choice of threshold value should be based on the specific features we aim to extract.

Learning Outcomes:

This project provided a comprehensive exploration of camera calibration and its role in Augmented Reality (AR). We successfully built a system capable of detecting, extracting, and projecting virtual objects onto image planes and video streams. This system achieved invariance to translation, scale, and rotation, ensuring accurate object placement regardless of camera position or object size.

Camera Calibration: We gained proficiency in using OpenCV functions to calibrate the camera, a crucial step for accurate AR applications.

Target Interaction: We mastered techniques for detecting and extracting target corners, paving the way for robust object tracking in future AR development.

Camera Pose Estimation: We implemented camera pose estimation to understand the camera's position and orientation relative to the target, enabling real-time object placement.

3D Object Projection: We developed skills in creating and projecting virtual 3D objects onto the image plane, ensuring their orientation adapts to camera movement, creating a realistic AR experience.

Acknowledgement:

We extend our heartfelt gratitude to the following platforms for their invaluable support and resources throughout the Calibration and Augmented Reality project:

- Springer
- ResearchGate
- GitHub
- Stack Overflow
- ChatGPT
- Gemini
- Medium

The insights and references obtained from these platforms have significantly enriched our understanding and greatly contributed to the successful development and implementation of various aspects of the project. Their assistance has been indispensable in ensuring a comprehensive and well-informed approach to the tasks at hand. We are truly thankful for the wealth of information and guidance provided by these platforms, which have been essential in our journey towards excellence in Calibration and Augmented Reality project.