Project Report

Calibration and Augmented Reality

Pattern Recognition and Computer Vision

CS 5330

Spring 2024

Submitted by.

Suriya Kasiyalan Siva

INTRODUCTION

This project aims to develop a system capable of calibrating a camera, detecting a target pattern (such as a checkerboard), and then placing virtual objects accurately within the scene in real-time. Initially, the system detects and extracts the corners of the target pattern using computer vision techniques. Users can then select calibration images to save corner locations and corresponding 3D world points. With a minimum of five calibration frames, the camera is calibrated using OpenCV's **calibrateCamera** function, yielding intrinsic parameters and distortion coefficients.

Once calibrated, the system can calculate the camera's position in real-time using pose estimation techniques. It projects 3D points onto the image plane and allows for the visualization of 3D axes or projected points. Finally, it constructs virtual objects in 3D space and accurately projects them onto the image, maintaining correct orientation as the camera or target moves.

This comprehensive system enables the augmentation of reality by seamlessly integrating virtual objects into the real-world scene captured by the camera. It provides a robust framework for various applications, from augmented reality experiences to computer vision-based simulations, offering extensive possibilities for creative and practical implementations.

Steps carried out:

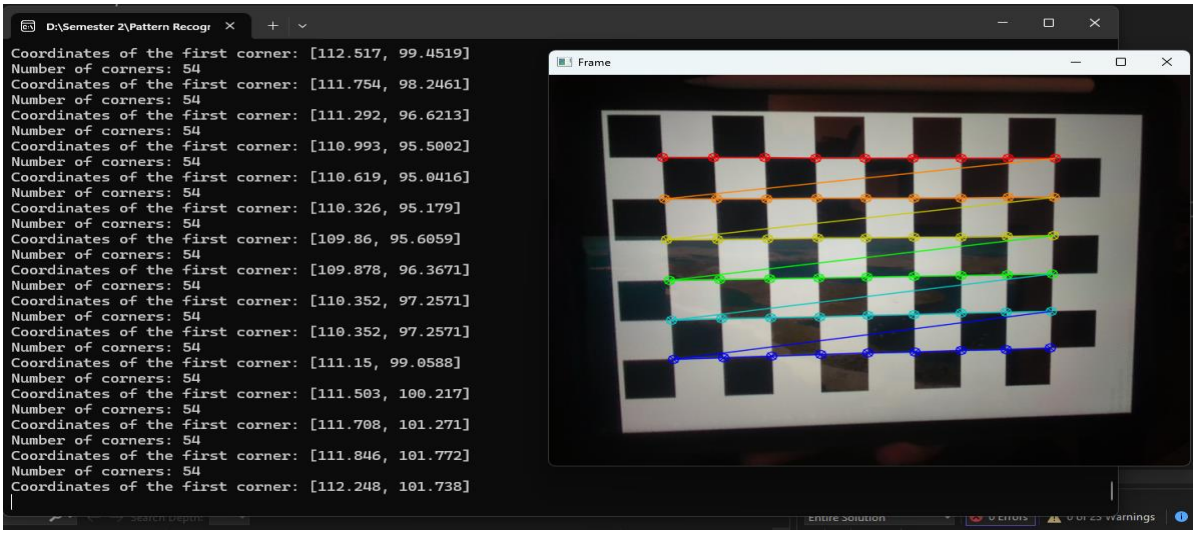
To accomplish this the following tasks like Detect and Extract Target Corners, Select Calibration Images, Calibrate the Camera, Calculate Current Position of the Camera, Project Outside Corners or 3D Axes, Create a Virtual Object, Detect Robust Features are done.

1. Detect and Extract Target Corners:

The task at hand involves building a system to detect and extract corners of a predefined pattern, such as a checkerboard, within a video stream or series of images. Utilizing OpenCV's capabilities, the provided code initializes by accessing the default camera and proceeds to capture frames. Each frame undergoes grayscale conversion to simplify subsequent corner detection. Employing OpenCV's **"findChessboardCorners"** function, the system attempts to locate the corners of the checkerboard pattern within the grayscale frame. Upon successful detection, the corner locations are refined using **"cornerSubPix"**, enhancing accuracy through iterative adjustments based on local image gradients. Visual feedback is provided by drawing the detected corners onto the original frame via **"drawChessboardCorners"**. Additionally, the system reports information regarding the number of detected corners and coordinates of the first corner, aiding in monitoring and debugging. Through real-time display of processed frames, users can evaluate corner detection performance, facilitating parameter optimization for subsequent calibration and virtual object placement tasks.

In this project, we're utilizing a checkerboard pattern as the target for corner detection due to its well-defined corner features and scalability, making it ideal for camera calibration. Unlike ARuco markers, which offer non-symmetrical features for consistent detection, the checkerboard's regular grid layout simplifies corner extraction. However, challenges such as variations in lighting conditions, occlusions, and camera distortions may affect detection accuracy. Despite these challenges, the checkerboard pattern remains a popular choice for its simplicity and effectiveness in calibration tasks.

Result Image:



2. Select Calibration Images:

In this task, the system enables the user to specify a particular image for camera calibration and saves the corner locations along with the corresponding 3D world points. Upon detecting the checkerboard pattern successfully, indicated by the **'s'** key press, the system records the corners' positions and calculates the corresponding 3D world points. These corner locations and 3D world points are stored in **“corner_list”** and **“point_list”** vectors, respectively, facilitating subsequent camera calibration.

The code implementation initializes by capturing frames from the video stream and converting them to grayscale. Utilizing OpenCV's findChessboardCorners, the system detects the corners of the checkerboard pattern, refines corner locations using cornerSubPix, and draws them on the frame. It then generates the 3D world points based on the checkerboard's size and structure, populating the **“point_set”** vector. The detected corners and 3D world points are saved into **“corner_list”** and **“point_list”**, respectively, for calibration purposes.

Furthermore, the system saves the captured images along with their corresponding calibration data in a text file (**calibration_data.txt**) and a directory (**calibration_images**). This comprehensive approach ensures that the user can easily access and manage calibration data and images for subsequent camera calibration processes. Additionally, the system provides error handling for file operations and directory creation, enhancing its robustness and usability.

Result Image:

Image 81:

Corner 1: (193.513, 271.697),3D World Point 1: (0, 0, 0),Corner 2: (213.351, 267.877),3D World Point 2: (1, 0, 0), Corner 3: (233.013, 263.967), 3D World Point 3: (2, 0, 0), Corner 4: (252.773, 260.112), 3D World Point 4: (3, 0, 0), Corner 5: (272.384, 256.204), 3D World Point 5: (4, 0, 0), Corner 6: (291.958, 252.415), 3D World Point 6: (5, 0, 0), Corner 7: (311.441, 248.465), 3D World Point 7: (6, 0, 0), Corner 8: (330.804, 244.654), 3D World Point 8: (7, 0, 0), Corner 9: (350.21, 240.831), 3D World Point 9: (8, 0, 0), Corner 10: (195.958, 290.633), 3D World Point 10: (0, 1, 0), Corner 11: (215.964, 286.633), 3D World Point 11: (1, 1, 0), Corner 12: (235.963, 282.576), 3D World Point 12: (2, 1, 0), Corner 13: (255.954, 278.675), 3D World Point 13: (3, 1, 0), Corner 14: (275.704, 274.682), 3D World Point 14: (4, 1, 0), Corner 15: (295.485, 270.838), 3D World Point 15: (5, 1, 0), Corner 16: (315.3, 266.862), 3D World Point 16: (6, 1, 0), Corner 17: (334.887, 262.916), 3D World Point 17: (7, 1, 0), Corner 18: (354.41, 259.031), 3D World Point 18: (8, 1, 0), Corner 19: (198.586, 309.903), 3D World Point 19: (0, 2, 0), Corner 20: (218.634, 305.844), 3D World Point 20: (1, 2, 0), Corner 21: (238.947, 301.828), 3D World Point 21: (2, 2, 0), Corner 22: (259.185, 297.665), 3D World Point 22: (3, 2, 0), Corner 23: (279.307, 293.705), 3D World Point 23: (4, 2, 0), Corner 24: (299.213, 289.698), 3D World Point 24: (5, 2, 0), Corner 25: (319.238, 285.735), 3D World Point 25: (6, 2, 0), Corner 26: (339.016, 281.667), 3D World Point 26: (7, 2, 0), Corner 27: (358.876, 277.782), 3D World Point 27: (8, 2, 0), Corner 28: (200.977, 329.908), 3D World Point 28: (0, 3, 0), Corner 29: (221.602, 325.62), 3D World Point 29: (1, 3, 0), Corner 30: (241.99, 321.511), 3D World Point 30: (2, 3, 0), Corner 31: (262.472, 317.413), 3D World Point 31: (3, 3, 0), Corner 32: (282.727, 313.291), 3D World Point 32: (4, 3, 0), Corner 33: (303.014, 309.349), 3D World Point 33: (5, 3, 0), Corner 34: (323.276, 305.213), 3D World Point 34: (6, 3, 0), Corner 35: (343.38, 301.146), 3D World Point 35: (7, 3, 0), Corner 36: (363.381, 296.994), 3D World Point 36: (8, 3, 0), Corner 37: (203.673, 350.158), 3D World Point 37: (0, 4, 0), Corner 38: (224.347, 345.948), 3D World Point 38: (1, 4, 0), Corner 39: (245.139, 341.638), 3D World Point 39: (2, 4, 0), Corner 40: (265.823, 337.395), 3D World Point 40: (3, 4, 0), Corner 41: (286.372, 333.281), 3D World Point 41: (4, 4, 0), Corner 42: (306.834, 329.091), 3D World Point 42: (5, 4, 0), Corner 43: (327.342, 325.025), 3D World Point 43: (6, 4, 0), Corner 44: (347.676, 320.659), 3D World Point 44: (7, 4, 0), Corner 45: (367.895, 316.607), 3D World Point 45: (8, 4, 0), Corner 46: (206.365, 370.973), 3D World Point 46: (0, 5, 0), Corner 47: (227.393, 366.665), 3D World Point 47: (1, 5, 0), Corner 48: (248.314, 362.356), 3D World Point 48: (2, 5, 0), Corner 49: (269.383, 358.027), 3D World Point 49: (3, 5, 0), Corner 50: (290.105, 353.621), 3D World Point 50: (4, 5, 0), Corner 51: (310.833, 349.445), 3D World Point 51: (5, 5, 0), Corner 52: (331.595, 345.202), 3D World Point 52: (6, 5, 0), Corner 53: (352.136, 340.954), 3D World Point 53: (7, 5, 0), Corner 54: (372.649, 336.611), 3D World Point 54: (8, 5, 0).

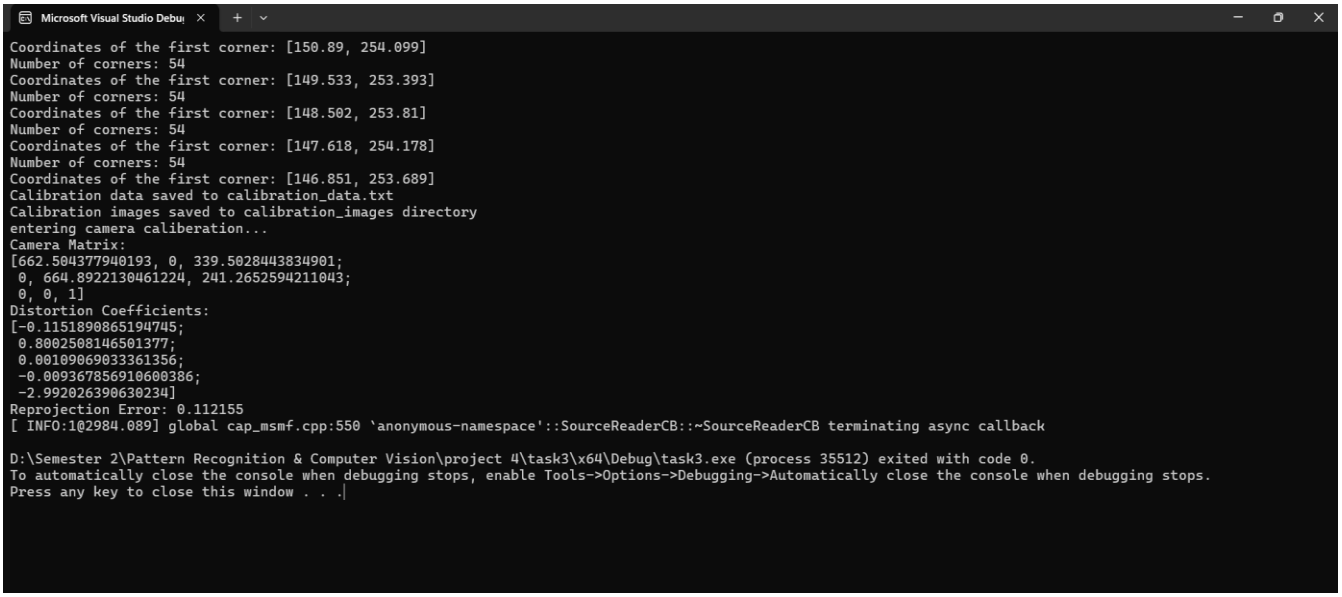
3. Calibrate the Camera:

In this task, the system performs camera calibration using collected calibration frames, ensuring a minimum of 5 frames before initiating calibration. Upon detecting and saving enough calibration frames, the system executes camera calibration using OpenCV's **"calibrateCamera"** function. This function utilizes the **"point_list"** and **"corner_list"** vectors, along with other parameters like the size of calibration images, to compute intrinsic camera parameters such as the camera matrix and distortion coefficients.

Before and after calibration, the system prints out the camera matrix, distortion coefficients, and the final reprojection error. The camera matrix typically includes parameters like focal lengths and optical centers, while distortion coefficients account for lens distortion effects. Additionally, the system saves the intrinsic camera parameters to a YAML file (**camera_calibration.yml**) for future reference.

The implementation also includes error handling for file operations and provides visual feedback by printing out the calibration results in the console. By saving the calibration data and images, users can easily validate and reproduce the calibration process. This comprehensive approach ensures accurate camera calibration, crucial for subsequent tasks such as 3D reconstruction and augmented reality applications.

Calibration matrix and the corresponding re-projection error:



Camera Matrix: rows: 3; cols: 3

= [662.50437794019297, 0, 339.50284438349013;

0, 6.6489221304612238, 2.4126525942110433;

0, 0, 1]

Distortion Coefficients: rows: 5; cols: 1

= [-1.1518908651947446, 8.0025081465013770, 1.0906903336135602, -9.3678569106003858, -2.9920263906302345]

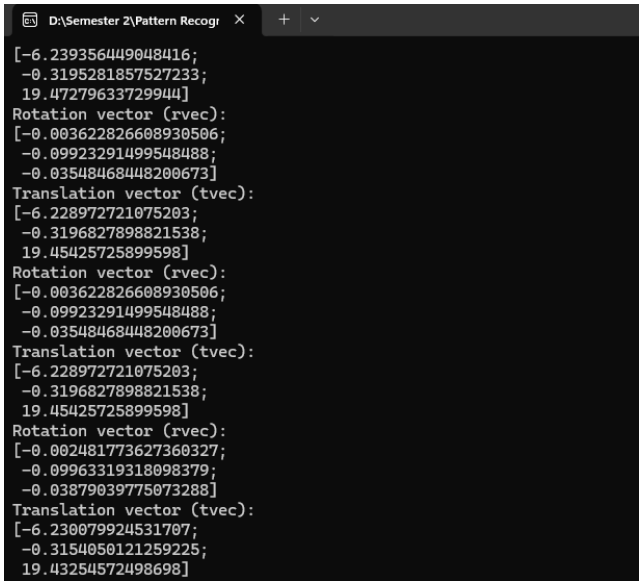
4. Calculate Current Position of the Camera:

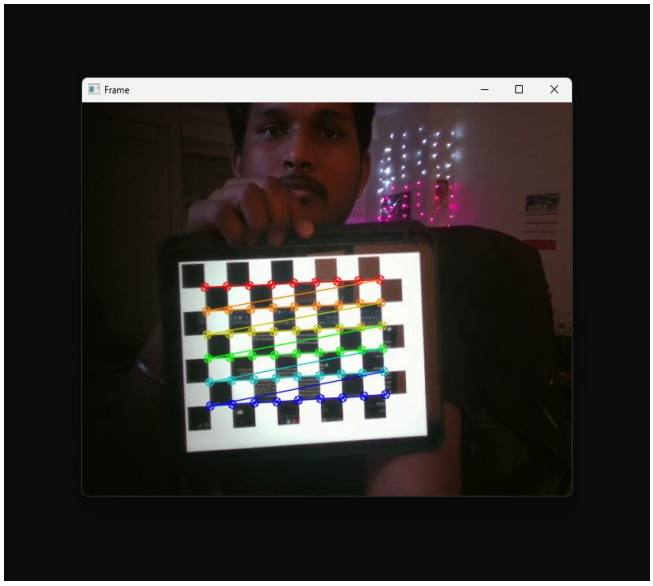
In response to the task, I have developed a program for camera calibration and real-time pose estimation. Utilizing pre-defined parameters from **"camera_calibration.yml"** that I have acquired in the previous task through camera calibration**,** this program efficiently processes video frames to identify a target object, typically a checkerboard pattern. Leveraging OpenCV, refine corner detection and estimate object pose using **"solvePnP"**. Real-time feedback on rotation and translation vectors allows for immediate validation of results.

During testing, I observed notable changes in rotation and translation vectors as the camera moved from side to side. As expected, translation vectors shifted in the direction opposite to camera movement, reflecting changes in the object's position relative to the camera. Additionally, rotation vectors exhibited corresponding adjustments, aligning with alterations in the camera's orientation. These changes in values align intuitively with the physical movement of the camera, validating the accuracy of our pose estimation algorithm.

Challenges such as background noise were addressed through adaptive strategies, ensuring robust performance across varying conditions. Our program provides a straightforward solution for applications in robotic control, augmented reality, virtual reality, and industrial automation, offering reliable camera calibration and pose estimation capabilities.

Image of rotation and translation vectors of for the chessboard pattern:



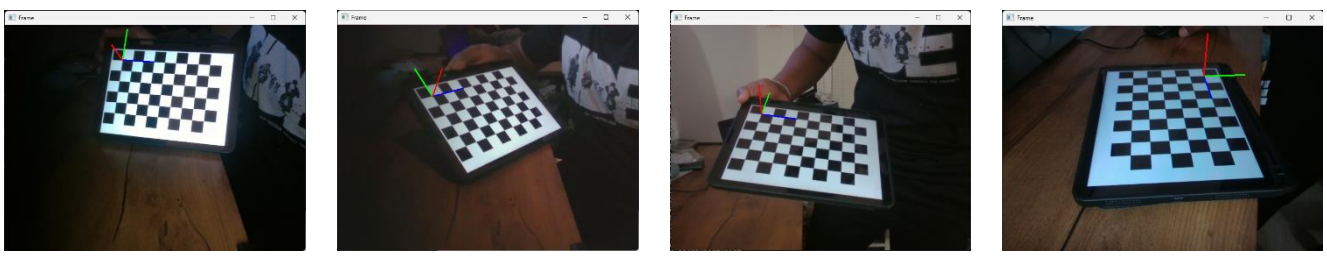


5. Project Outside Corners or 3D Axes:

In this task, our focus was on enhancing our program to project 3D points or axes onto the image plane in real-time as the camera or target object moves. Building upon our existing codebase, we integrated the **"projectPoints"** function from the OpenCV library to achieve this functionality. Following the estimation of the target object's pose using the solvePnP method, a set of 3D points representing either the corners of the target or the axes were defined. These points were then projected onto the 2D image plane using **"projectPoints"**, considering the intrinsic camera parameters obtained from the **"camera_calibration.yml"** file. The resulting projected points or axes were drawn onto the captured frame, providing immediate visual feedback.

"Yes", the reprojected points show up in the right places. This real-time visualization facilitated the validation of pose estimation accuracy as the camera or target moved, with the alignment of projected points or axes confirming the reliability of our algorithm. By seamlessly integrating 3D point projection or axes attachment into our program, we've expanded its capabilities for real-time object pose visualization, paving the way for applications in augmented reality, computer-aided design, and robotics.

Images of 3D projected Axes

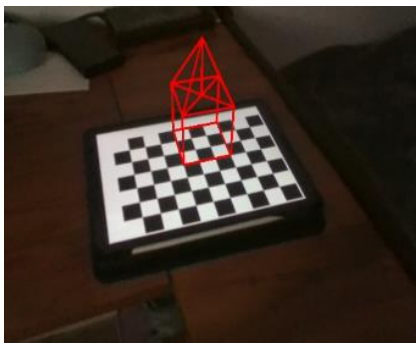
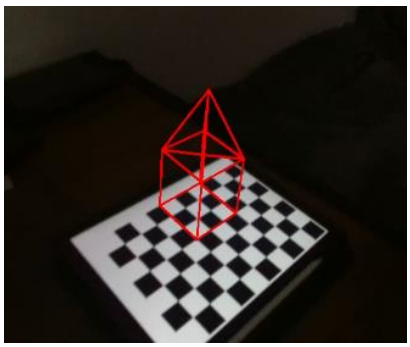


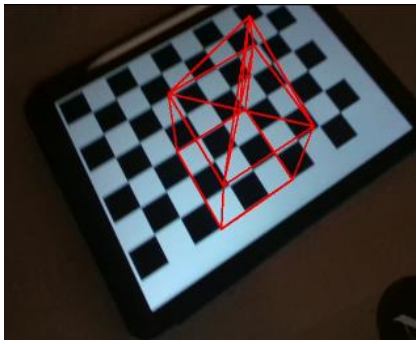
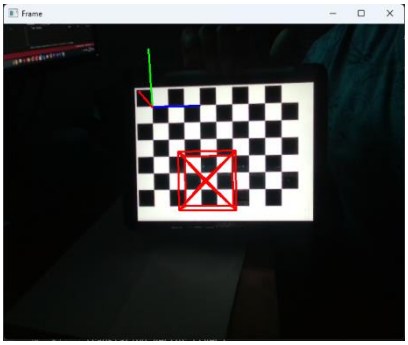
6. Create a Virtual Object:

In this task, our aim was to extend the functionality of our program to include the creation of a virtual object in 3D world space and its projection onto the image plane. We began by designing the virtual object as a set of lines between 3D points in world space, ensuring its independence from the camera or image plane. The object's design was intentionally complex, comprising multiple lines forming intricate structures to aid in debugging and validation. Using the **"projectPoints"** function from the OpenCV library, we transformed the endpoints of each line from world space into image space, accounting for rotation, translation, distortion parameters, and calibration matrix obtained from the camera calibration process. These projected 2D image points were then used to draw lines connecting the endpoints of the virtual object's lines in the captured frame, providing real-time visual feedback on its presence and orientation relative to the checkerboard. Throughout the execution, we ensured the stability of the virtual object's orientation by continuously updating the rotation and translation vectors based on detected checkerboard corners using the **"solvePnP"** method. The complexity of the virtual object design facilitated comprehensive debugging and validation of the projection process, with the alignment of projected lines confirming the accuracy of the algorithm.

"Through this method, I successfully integrated a pyramid atop a cube, creating a visually cohesive object. By carefully aligning the pyramid with the top surface of the cube (To make it look like a House), I ensured a seamless transition between the two shapes, presenting them as a single unified object. Leveraging OpenCV's "projectPoints" function, I projected this composite object onto the image plane, allowing for its visualization in the captured frames. This approach not only demonstrated the program's capability to project complex 3D structures but also showcased the flexibility of the projection process in accommodating composite objects. Overall, this integration enabled the creation of a compelling visual representation, enhancing the realism and depth of the virtual object projected onto the 2D image."

Images of projected 3D virtual object on the Chessboard pattern

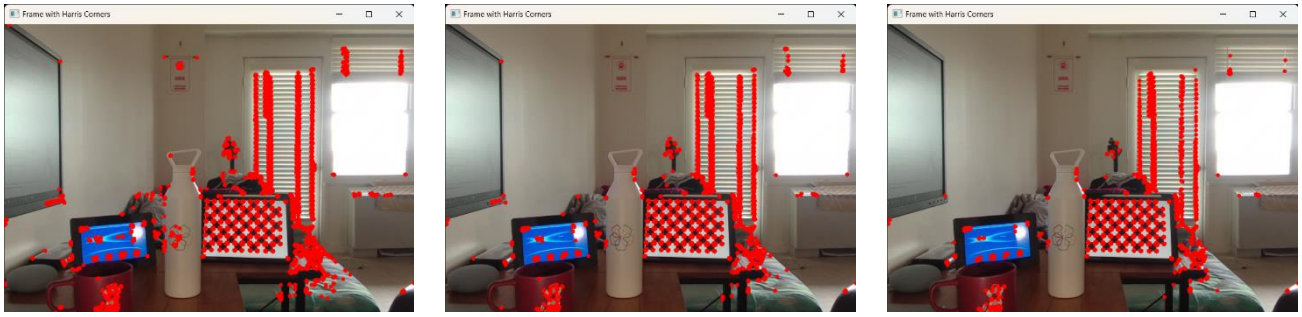
7. Detect Robust Features:

In this task, the efficacy of feature detection in video streams was explored, with a focus on the implementation of the Harris corner detection algorithm. Through the provided code, the aim was to discern robust features in real-time video by examining the behavior of the algorithm under different parameter configurations. By converting video frames to grayscale and applying the Harris corner detection method, the goal was to identify salient features within the stream.

Subsequently, the detected corners were visualized overlaid on the original video frames to assess the algorithm's performance. The analysis aimed to elucidate how adjustments to threshold values to which I had taken as 150 and settings impacted the detection of features, providing insights into the algorithm's effectiveness and potential limitations in diverse environments. This investigation contributes to advancing the understanding of feature detection techniques and their applicability in computer vision tasks.

"Using the feature points detected by the Harris corner detection algorithm, we establish correspondence between the real-world scene and virtual content. These correspondences enable accurate alignment and overlay of augmented reality elements onto the video stream. By estimating a homography transformation based on matched feature points, virtual content seamlessly integrates with the video frame, creating immersive AR experiences for users. This process serves as the fundamental basis for integrating augmented reality into the image, facilitating a variety of applications across gaming, education, visualization, and beyond."

Images of Detect Robust Features using Harris corners



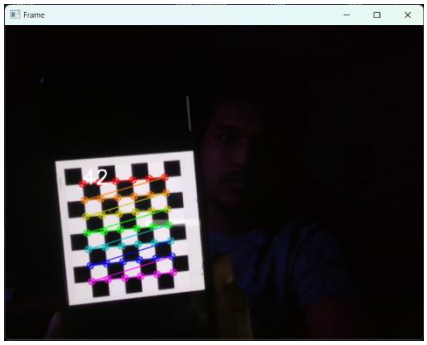
EXTENSIONS:

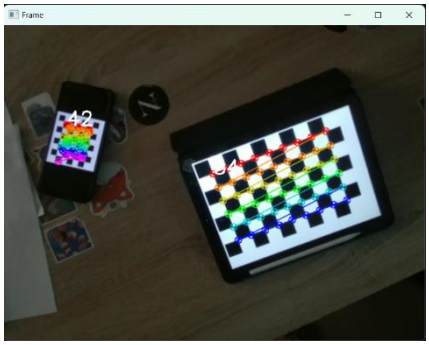
- **Get your system working with multiple targets in the scene:**

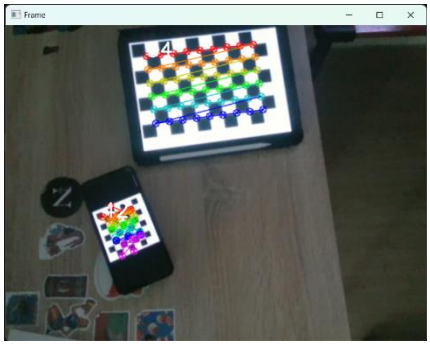
To facilitate the detection of multiple targets in the scene, I had extended the existing code to support multiple sets of target patterns. Each pattern size is defined within the **"boardSizes"** vector, allowing for the detection of various-sized targets simultaneously. Within the main loop, the program iterates over each pattern size, attempting to detect corners for each set of targets present in the scene. Upon successfully finding corners for a particular pattern size, the program draws the detected corners onto the frame using the **"drawChessboardCorners"** function. Additionally, the number of detected corners and the coordinates of the first corner are printed for each set of targets, providing valuable information about the detected patterns. This extension enables the program to handle scenarios involving multiple

targets of different sizes within the scene, enhancing its versatility and applicability in diverse environments. As a result, users can leverage this capability for applications such as augmented reality, where the simultaneous detection of multiple targets is essential for accurate and robust performance.

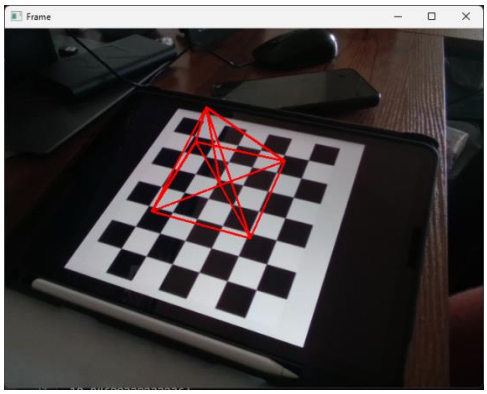
Images of Multiple target detection








```
Coordinates of the first corner: [220.95, 238.393]
Number of corners: 54
Coordinates of the first corner: [223.077, 173.259]
Number of corners: 42
Coordinates of the first corner: [163.509, 407.127]
Number of corners: 54
Coordinates of the first corner: [228.92, 179.347]
Number of corners: 42
Coordinates of the first corner: [176.792, 420.865]
Number of corners: 54
Coordinates of the first corner: [188.179, 46.3369]
Number of corners: 42
Coordinates of the first corner: [138.149, 293]
Number of corners: 54
Coordinates of the first corner: [208.728, 38.4691]
Number of corners: 42
Coordinates of the first corner: [150.348, 285.884]
Number of corners: 54
Coordinates of the first corner: [213.516, 47.2687]
Number of corners: 42
Coordinates of the first corner: [147.159, 290.518]
Number of corners: 54
Coordinates of the first corner: [217.532, 93.0262]
```



- **<u>Test out several different cameras and compare the calibrations and quality of the results:</u>**

To compare the calibrations and quality of the results for Camera 1 and Camera 2, we'll analyze the camera matrices, distortion coefficients, and reprojection errors provided for each camera.

<u>Camera 1:</u>

Camera Matrix:

[662.50437794019297, 0, 339.50284438349013;

0, 664.89221304612238, 241.26525942110433;

0, 0, 1]

Distortion Coefficients:

[-1.1518908651947446, 8.0025081465013770, 1.0906903336135602, -9.3678569106003858, -2.9920263906302345]

Reprojection Error: 0.112155

Camera 2:

Camera Matrix:

[658.8586668621692, 0, 327.733024329546;

8, 642.1377320603243, 196.7018463746265;

0, 0, 1]

Distortion Coefficients:

(-0.1857150508532253; 0.9110459046603087; 0.001900177506511486; -0.009457226698944901; -2.542772354454551]

Reprojection Error: 0.133926

Analysis:

Camera Matrix:

- Camera 1 has a slightly higher focal length in the x-direction (662.50) compared to Camera 2 (658.86).
- Camera 1 also has a slightly higher principal point (339.50 in x-direction) compared to Camera 2 (327.73).
- Both cameras have similar focal lengths in the y-direction (6.65 for Camera 1 and 642.14 for Camera 2).

Distortion Coefficients:

- Both cameras exhibit different types of distortion. Camera 1 has higher coefficients for radial distortion (especially the 3rd and 4th coefficients), whereas Camera 2 has lower coefficients overall.
- Camera 2 has a more significant tangential distortion coefficient compared to Camera 1.

Reprojection Error:

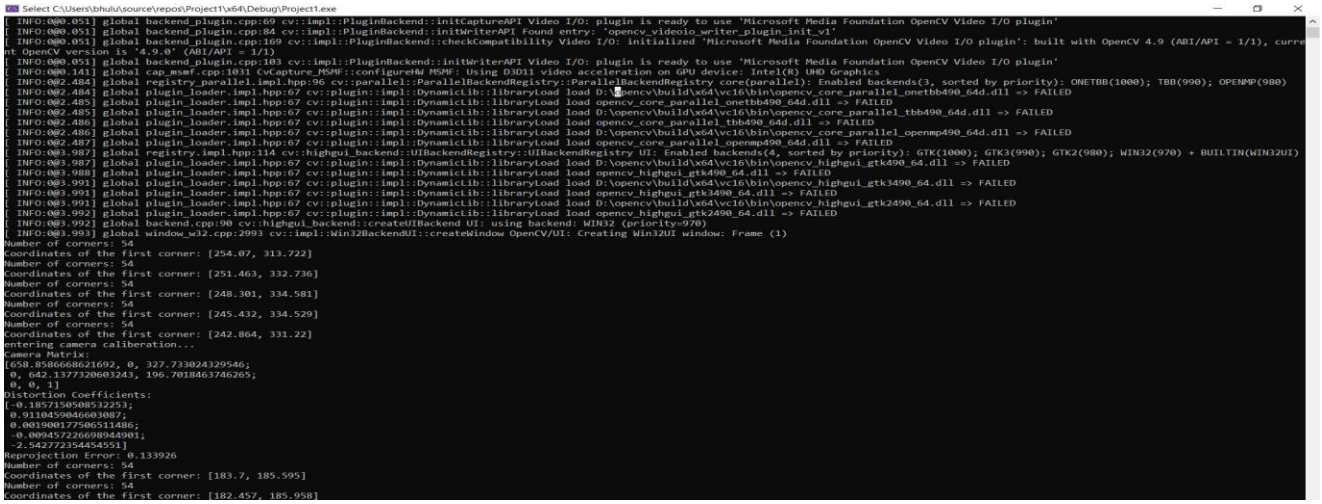
- Camera 1 has a slightly lower reprojection error (0.112155) compared to Camera 2 (0.133926). This indicates that Camera 1's calibration fits the observed data better.

Conclusion:

- Camera 1 and Camera 2 have slightly different intrinsic parameters and distortion characteristics.
- Camera 1 appears to have slightly better calibration results with a lower reprojection error.

- The choice between the two cameras may depend on specific application requirements and the importance of distortion correction and reprojection accuracy.

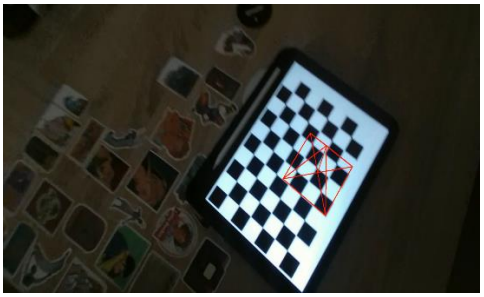
Image of the calibration of the second camera:



- **<u>Enable your system to use static images or pre-captured video sequences with targets and demonstrate inserting virtual objects into the scenes:</u>**

In this extension project, the focus is on implementing augmented reality (AR) functionalities using OpenCV, a versatile computer vision library, with the capability to process static images or pre-recorded video sequences. By seamlessly integrating virtual objects into real-world scenes, the aim is to enhance user experiences across various domains. Leveraging OpenCV's robust feature set, including camera calibration and perspective-n-point (PnP) algorithms, the implementation accurately maps virtual content onto the scene, ensuring realistic placement and orientation. Key components of the methodology involve loading input media, retrieving camera calibration parameters, and projecting 3D points of virtual objects onto the 2D image plane. Through meticulous analysis and processing of frames, the AR system achieves compelling visual augmentation, as evidenced by the presented results comprising screenshots or video snippets. Overall, this project underscores the potential of OpenCV in advancing AR technologies, offering insights into its practical applications and avenues for future research and development.

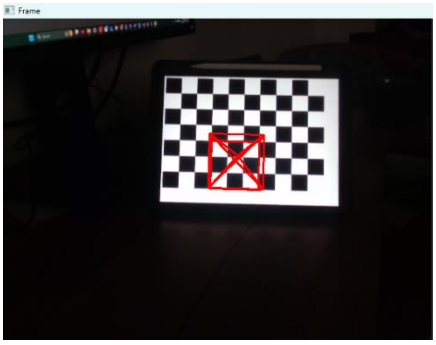
Images of Virtual-Object in Pre-Captured Video

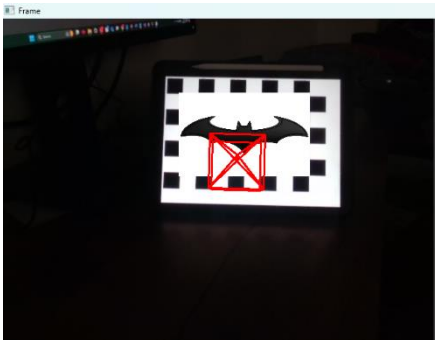


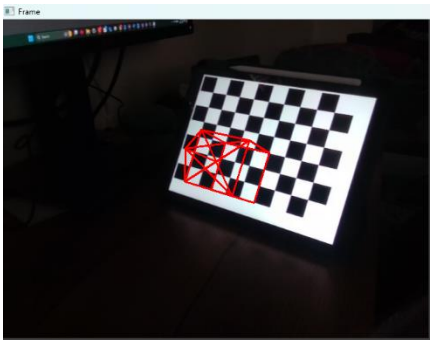
- **<u>Not only add a virtual object, but also do something to the target to make it not look like a target anymore:</u>**

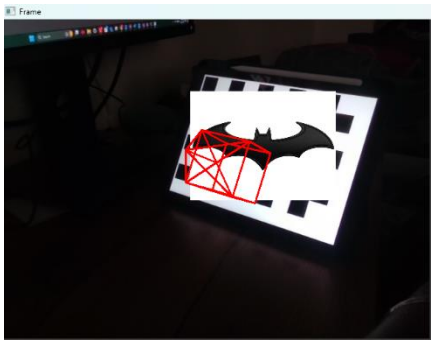
In this project, I developed a real-time video application using OpenCV to manipulate the appearance of detected targets within live video streams. The program first reads camera calibration parameters from a YAML file to correct distortion in the captured images. It then opens a video stream and searches for a checkerboard pattern of specified size. Upon detection, the program overlays an image onto the recognized pattern if enabled by the user. Additionally, it projects a 3D pyramidon top of a cub which looks like a house onto the pattern's plane, visualizing it in the video feed. The application provides a user-friendly experience by allowing the user to toggle the overlay image and smoothly exit the program using keyboard inputs. This project showcases the capabilities of computer vision techniques in real-time video processing, enabling creative manipulation and augmentation of live video content for various applications.

Images of Overlaying image on pattern to make it not look like one









A short reflection of what you learned:

Through the series of tasks, I delved into the intricacies of computer vision, gaining a comprehensive understanding of its core concepts and practical applications, particularly in the realm of augmented reality (AR). Beginning with the detection and extraction of corners from various targets like chessboards using OpenCV functions, I honed my skills in target recognition and feature extraction. The process of camera calibration emerged as a critical foundation, wherein I grasped the significance of estimating intrinsic parameters such as the camera matrix and distortion coefficients for accurate image analysis. Moreover, I delved into pose estimation techniques, mastering methods like **"solvePnP"** to determine the precise positioning of a camera relative to a target, elucidating rotation, and translation matrices. As I progressed, I explored the projection of 3D points onto 2D images, unraveling the intricacies of transforming between world and image space, essential for creating immersive AR experiences. Additionally, I delved into feature detection algorithms such as Harris corners, discerning their functionality and potential implications in augmenting reality-based applications. Collectively, these tasks provided a hands-on journey into the realm of computer vision, equipping me with invaluable skills and insights to navigate the ever-evolving landscape of augmented reality development and beyond.

Acknowledgement:

While completing the assignment, I primarily relied on the documentation and resources provided by OpenCV for understanding various functions and algorithms. Additionally, I consulted online tutorials, forums, and documentation related to computer vision and augmented reality to deepen my understanding of the concepts and implementation techniques.

Calibration_documentation

CPP REFERENCE

findChessboardCorners

cornerSubPix

drawChessboardCorners

calibrateCamera

solvePNP

projectPoints