REPORT
PROJECT 04

(Calibration and Augmented Reality)

By – Veditha Gudapati
&
Tejasri Kasturi

Introduction:

In this project, our aim is to develop a system capable of analyzing and enhancing video streams in real time. To achieve this, we employ a range of sophisticated techniques, including chessboard corner extraction and Harris corner recognition. These methods enable us to pinpoint crucial locations within the video frame accurately. Moreover, our approach involves a crucial step known as camera calibration. This process ensures that the conversion from pixel coordinates to real-world coordinates is precise, laying a solid foundation for subsequent tasks.

Once calibrated, our focus shifts towards augmented reality (AR). Leveraging the calibration data obtained from the camera, we can determine the precise position of the target relative to the camera. This enables us to project virtual objects, created using a 3D coordinate system, onto the 2D video feed accurately. Even as the target or camera moves, these virtual objects maintain their correct orientation and position, enhancing the overall AR experience.

To further enhance the adaptability and reliability of our system, we explore feature detection techniques. By identifying distinctive features within the scene, our system can detect and superimpose virtual objects with greater accuracy, even in challenging conditions such as varying lighting or partial target occlusion.

TASK – 1: Detect and Extract Target Corners

We chose a chessboard pattern as our target since chessboard patterns have well-defined corners, making them suitable for corner detection algorithms commonly used in computer vision tasks. The function **"CornersExtract"** takes the input image (src) and attempts to locate corners using the **"cv::findChessboardCorners"** function, which searches for internal corners within the grid pattern. If corners are found, the function refines their positions with sub-pixel accuracy using **"cv::cornerSubPix"**. Optionally, if the **"drawCorners"** flag is set, the detected corners are drawn on the output image (dst) using **"cv::drawChessboardCorners"**. The detected corner pixel coordinates are stored in the corners vector, facilitating further processing such as camera calibration and augmented reality applications.

Implementation:

1. **Corner Detection:** To detect the corners of the checkerboard pattern, we utilized the **"findChessboardCorners"** function provided by the OpenCV library. This function takes the current frame (cv::Mat) as input and searches for the corners of a specified checkerboard grid size (e.g., 9x6). The function returns a Boolean value indicating whether corners are found and populates a vector (std::vector<cv::Point2f> corners) with the pixel coordinates of the detected corners.

2. **Corner Refinement:** To improve the accuracy of corner localization, we applied corner refinement using the **"cornerSubPix**" function. This function refines the corner locations based on grayscale image gradients, resulting in more precise corner coordinates.

3. **Draw Detected Corners:** We visually represented the detected corners on the frame using the **"drawCorners"** function. This step involved drawing circles or markers at the detected corner locations to provide visual feedback on the corner detection process. Drawing the detected corners on the frame facilitated real-time monitoring and validation of the corner detection algorithm.

This included the total number of corners detected and the coordinates of the first corner in the vector. Printing this information helped in monitoring the performance of corner detection and provided insights into the efficiency of corner detection in different scenarios.

Challenges:

1. Ensuring accurate corner detection under varying target orientations presented a challenge.
2. Achieving real-time performance while processing video frames for corner detection posed a considerable challenge.

TASK – 2: Select Calibration Images

The **"selectCalibrationImg()"** function in Task 2 of the project helps the functionality by accepting the detected corners, corner lists, points, and point lists as input. The parameters involved in the execution of this function are:

i. **corners:** A reference to a vector containing the pixel coordinates of detected corners in the current image frame.
ii. **corners_list:** A reference to a vector of vectors to store multiple sets of corner coordinates. Each set of corner coordinates corresponds to a single calibration image.
iii. **points:** A reference to a vector to store the world coordinates of the checkerboard target. These coordinates represent the physical locations of corners on the calibration target.
iv. **points_list:** A reference to a vector of vectors to store multiple sets of world coordinates. Each set of world coordinates corresponds to a single calibration image.

Implementation:

1. **World Coordinates Calculation:** For each detected corner in the current image frame, the function calculates its corresponding world coordinates based on the checkerboard pattern.

2. **Storage of Coordinates:** The pixel coordinates of detected corners (corners) and their corresponding world coordinates (points) are stored in the respective vectors.

3. **Adding Data to Lists:** The current set of corner coordinates (corners) and world coordinates (points) are added to the list's corners_list and points_list, respectively. Each set of coordinates represents one calibration image.

After corner detection, the program saves the current frame as a calibration image. Each saved image is named according to a specific format **"(calibration-frame-{frame_number}.jpg)"**, allowing for easy identification and retrieval during the calibration process. By doing this, it is made sure that the calibration process is started using a variety of photos for accuracy and resilience.

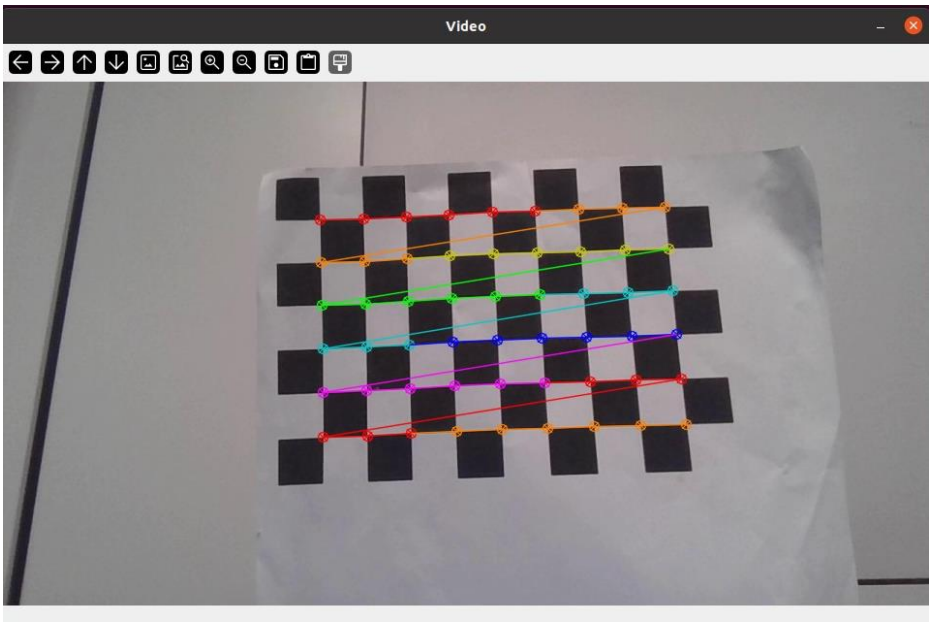

Figure 1: Calibration Image with Chessboard Corners

The purpose of this function is to prepare data for camera calibration. By collecting pixel coordinates and corresponding world coordinates from detected corners in calibration images, this function facilitates the estimation of the intrinsic parameters of the camera, such as focal length and principal point.

TASK – 3: Calibrate the Camera

In Task 3, the calibration target's corners in each image are detected using the **"extractCorners"** function, employing techniques like corner detection and sub-pixel refinement for precise corner localization. Making use of OpenCV's **"calibrateCamera()"** function simplifies the estimation of intrinsic camera characteristics and distortion coefficients, crucial for geometric image correction in the camera system.

To accurately map 3D world points to 2D image coordinates and correct lens distortions, this procedure is essential. When the user selects five or more calibration images, the function is used, prompting for inputs such as corner coordinates found in the calibration images and their associated 3D world points. These inputs, along with parameters like picture size and calibration flags, are organized into vectors and passed to the calibration function as **"point_list"** and **"corner_list"**.

During calibration, the function iteratively optimizes parameters including focal lengths, optical centers, and distortion coefficients to minimize the reprojection error, representing the difference between detected and projected image points. Lower reprojection errors indicate greater calibration accuracy.

To evaluate calibration effectiveness, monitoring the camera matrix and distortion coefficients before and after the operation is important. Intrinsic parameters are then permanently in a CSV file, aiding improved calibration results and enabling the reuse of calibrated camera models without recalibration in future tasks.

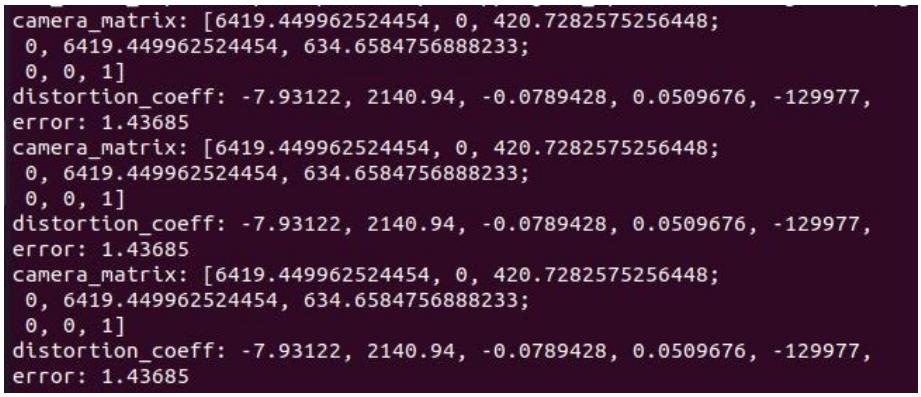

Figure 2: Error Estimate

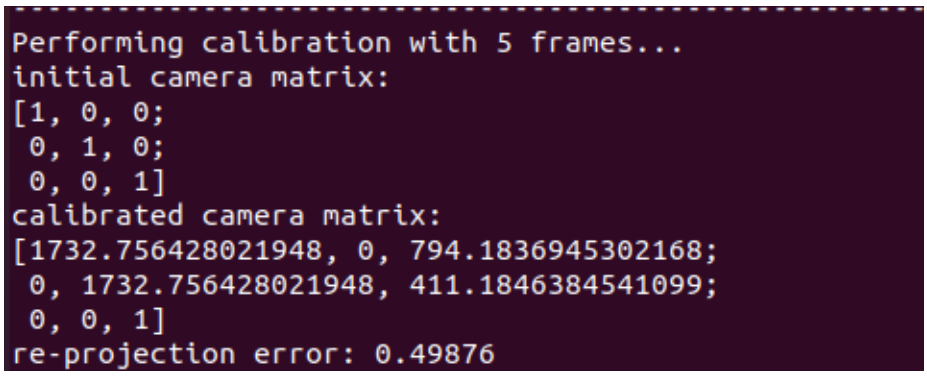

Figure 3: Calibrated Camera Matrix & Re-projection Error

TASK – 4: Compute Features for Each Major Region

In this task, after detecting feature points in the scene, they are matched with corresponding points in a reference frame or a known 3D model to establish correspondences between the 2D image coordinates and the 3D world coordinates of the feature points.

The **cameraCalcPosition** function is then invoked with parameters **points**, **corners**, **camera_matrix**, **dist_coeff**, **rot**, and **trans**. This function is responsible for estimating the rotation (rot) and translation (trans) matrices of the camera.

1. **Position Estimation:** This step involves estimating the camera's position in relation to the image after the corners have been found and the camera has been calibrated. Finding the rotation and translation matrices that characterize the shift from the camera's coordinate system to the world coordinate system is essential for position estimation. These matrices are estimated by the **cameraCalcPosition** function in the program.

2. **Rotation and Translation Matrices:** The translation matrix (trans) indicates the camera's location with respect to the world origin, while the rotation matrix (rot) depicts the camera's orientation in three dimensions. These matrices are computed using the detected corners, camera calibration settings, and potentially other data to establish correlation between 3D locations and their 2D projections.

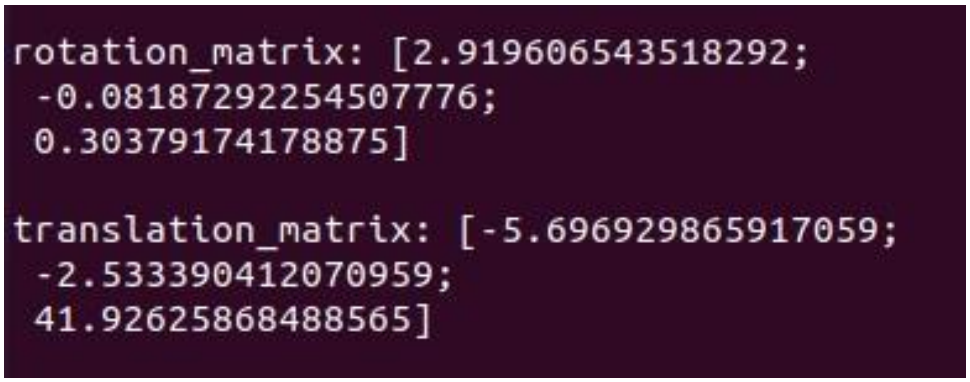

Figure 4: Rotation and Translation Matrices (1)

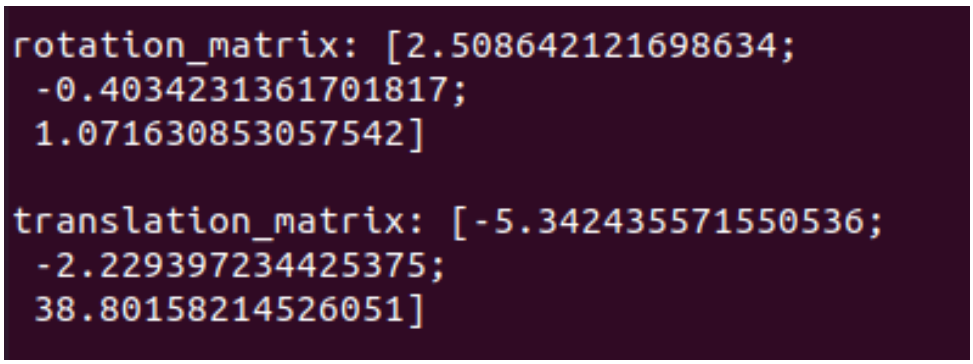

Figure 5: Rotation and Translation Matrices (2)

- **Rotation Matrices:**

  Rotation Matrix 1: [2.919606543518292; -0.08187292254507776; 0.30379174178875]

  Rotation Matrix 2: [2.508642121698634; -0.4034231361701817; 1.071630853057542]

  By comparing the two matrices, we can observe changes in the camera's orientation between the two frames. With respect to the obtained values:

  i. The rotation around the **x-axis** decreases from approximately 2.92 to 2.51.

  ii. The rotation around the **y-axis** changes from -0.08 to -0.40.

  iii. The rotation around the **z-axis** increases from 0.30 to 1.07.

  These variations show that the camera's tilt, rotation, or angle has changed between the two frames.

- **Translation Matrices:**

  Translation Matrix 1: [-5.696929865917059; -2.533390412070959; 41.92625868488565]

  Translation Matrix 2: [-5.342435571550536; -2.229397234425375; 38.80158214526051]

  These matrices represent the movement of the camera in each frame, typically in terms of x, y, and z coordinates. By comparing the two matrices, we can observe changes in the camera's position between the two frames. With respect to the obtained values:

  i. The camera moves closer to the scene along the **x-axis** (from approximately -5.70 to -5.34).

  ii. It also moves closer along the **y-axis** (from approximately -2.53 to -2.23).

  iii. The camera moves upward along the **z-axis** (from approximately 41.93 to 38.80).

  These changes indicate shifts in the camera's position in terms of (x, y) or (z).

TASK – 5: Project Outside Corners or 3D Axes

In Task 5, the objective is to project 3D axes onto the image frame based on the current estimated position of the camera. This task is crucial for visualizing the orientation and position of the camera relative to the target object. By projecting 3D axes onto the image frame, we provide a reference frame that aids in understanding the spatial relationship between the camera and the scene.

Implementation:

1. **Input Parameters:**
    - **src:** Input image frame where the 3D axes will be drawn.
    - **camera_matrix:** The calibrated camera matrix obtained from camera calibration.
    - **dist_coeff:** Distortion coefficients obtained from camera calibration.
    - **rot:** Rotation matrix representing the orientation of the camera in 3D space.
    - **trans:** Translation vector representing the position of the camera in 3D space.
2. **Coordinate System:** We use a right-handed coordinate system where the X-axis points to the right, the Y-axis points downwards, and the Z-axis points out of the camera lens (towards the scene).
3. **Generating 3D Axes Points:** We define 3D points representing the origin and endpoints of the X, Y, and Z axes. For example, (0, 0, 0) represents the origin, (2, 0, 0) represents the endpoint of the X-axis, and so on.
4. **Projecting 3D Points to 2D Image Coordinates:** Using the **"cv::projectPoints"** function provided by OpenCV, we project the 3D axes points onto the 2D image plane. This function takes the 3D points, camera rotation matrix, translation vector, camera matrix, and distortion coefficients as inputs and outputs the corresponding 2D image coordinates of the points.
5. **Drawing Axes Lines:** Once we obtain the 2D image coordinates of the axes points, we draw lines connecting these points on the image frame. Each axis is drawn using a distinct color (typically red for X-axis, green for Y-axis, and blue for Z-axis) to aid in visualization.
6. **Visualizing Camera Orientation:** By visualizing the 3D axes on the image frame, we provide a clear indication of the camera's orientation relative to the scene. This information is valuable for understanding how the camera is positioned and oriented with respect to the objects in the scene.

The output is the input image frame (src) with the 3D axes drawn on it. This modified image provides a visual representation of the camera's orientation and position in the scene.

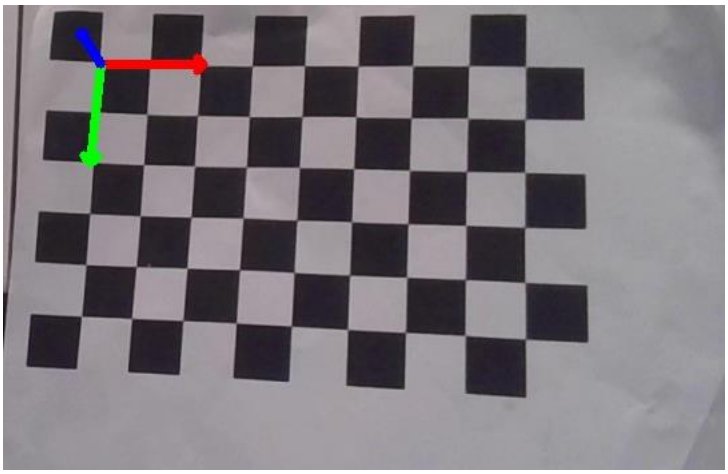

Figure 6: Reprojected Points

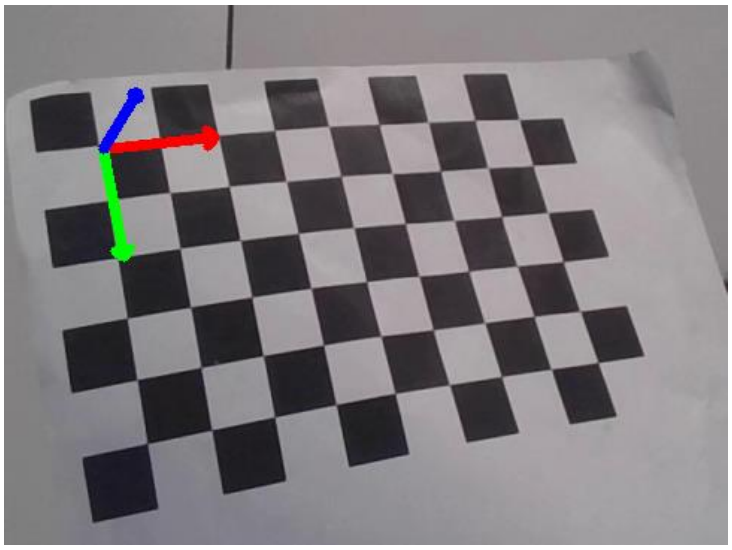

Figure 7: Reprojected Points

TASK – 6: Create a Virtual Object

In Task 6 of the computer vision project, we focus on the implementation of virtual object projection onto the target scene. The primary objective is to simulate the presence of three-dimensional (3D) objects within the captured image frames, enhancing the visual representation and potentially aiding in augmented reality applications.

Implementation:

1. **Projection Parameters:** Before projecting virtual objects, we need to establish the necessary projection parameters. These parameters include:
    - **Calibrated Camera Matrix:** This matrix encapsulates the intrinsic parameters of the camera, such as focal length, principal point, and skew factor, obtained through camera calibration. The function **"calibrateCamera()"** is used to calibrate the camera, obtaining the calibrated camera matrix **(camera_matrix)** and distortion coefficients **(dist_coeff)**.
    - **Distortion Coefficients:** These coefficients correct lens distortion effects present in the camera images.
    - **Camera Pose:** The current position and orientation of the camera relative to the scene, typically represented by rotation and translation matrices. These are estimated using techniques like solvePnP based on known reference points or features in the scene. The function **"calcCameraPosition()"** estimates the current position and orientation of the camera relative to the scene, providing rotation (rot) and translation (trans) matrices.

2. **Definition of Virtual Objects:** Each virtual object is defined in 3D space, using a set of vertices that describe its shape. The shapes implemented in the program are:
    - **Cylinders:** The function **"draw3dObject()"** defines a virtual cylinder by specifying parameters such as radius and height and approximates its vertices around the circumference.
    - **Pyramids:** The function **"draw3dObject()"** defines a virtual pyramid by specifying the vertices of its base and apex.
    - **Cubes:** The function **"draw3dObject()"** defines a virtual cube by specifying the vertices of its eight corners.

3. **Projection Process:**
    - **Transform to Camera Coordinate System:** The function **"projectPoints()"**, is used to transform the vertices of the virtual objects from the world coordinate system to the camera coordinate system.
    - **Projection to Image Plane:** The transformed 3D vertices are projected onto the 2D image plane using the function **"projectPoints()"**, considering the calibrated camera matrix and distortion coefficients.
    - **Drawing on Image Frame:** The function **"cv::line()"** is utilized to draw lines connecting the projected vertices onto the captured image frame.

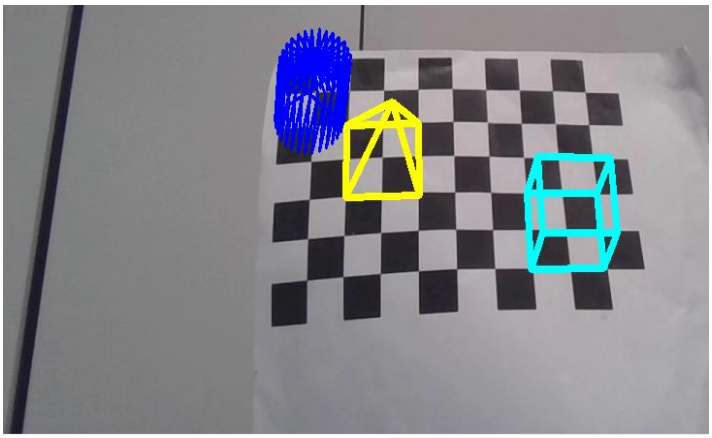

Figure 8: 3D Objects Represented

The list of keypresses that are implemented in the program until Task 6 are as follows:

1. *'**q**' to quit the program.*
2. *'**s**' to save the current calibration frame and perform calibration if frames >= 5.*
3. *'**c**' to save the current calibration in a CSV file to be read later.*
4. *'**x**' to display 3D axes at the origin of world coordinates.*
5. *'**d**' to display 3D objects.*

TASK – 7: Detect Robust Features

In Task 7, we focus on implementing robust feature detection using the Harris corner detection method. The objective is to identify significant image features that are robust to changes in lighting conditions, viewpoint, and other variations, which are crucial for various computer vision tasks such as object recognition, image stitching, and motion tracking.

The quality level parameter in the Shi-Tomasi corner detection algorithm (also known as the **"goodFeaturesToTrack"** function in OpenCV) determines the minimum accepted quality of corners to be detected.

The keypress "h" prints the number of Harris Corners detected onto the terminal.

Implementation:

1. **Corner Detection:** Utilized the **"cornerHarris"** function provided by OpenCV to compute the Harris corner response function for each pixel in the input image. The Harris corner detection algorithm is applied to the preprocessed image. This algorithm computes a corner response function for each pixel in the image, identifying regions with significant intensity variations in multiple directions. Each frame is converted to grayscale using **cvtColor()** function. The Harris

   corner detection algorithm is then applied using **cornerHarris()** function to detect corners in the grayscale image. Parameters such as the maximum number of corners to detect (500), quality level (0.01/0.1), and minimum Euclidean distance between detected corners (10) are provided to fine-tune the detection process.
2. **Normalizing:** The detected corners are normalized and converted to 8-bit unsigned integer format. Good features to track are extracted using the Shi-Tomasi corner detection algorithm via the **goodFeaturesToTrack()** function. Circles are drawn around the detected corners using the circle() function.
3. **Visualization:** Finally, the detected corner points are visualized on the original image to provide a visual representation of the identified robust features. This visualization of the detected corners are represented as circles.

Case 1 : Quality Level = 0.01

- **Pattern 1:**

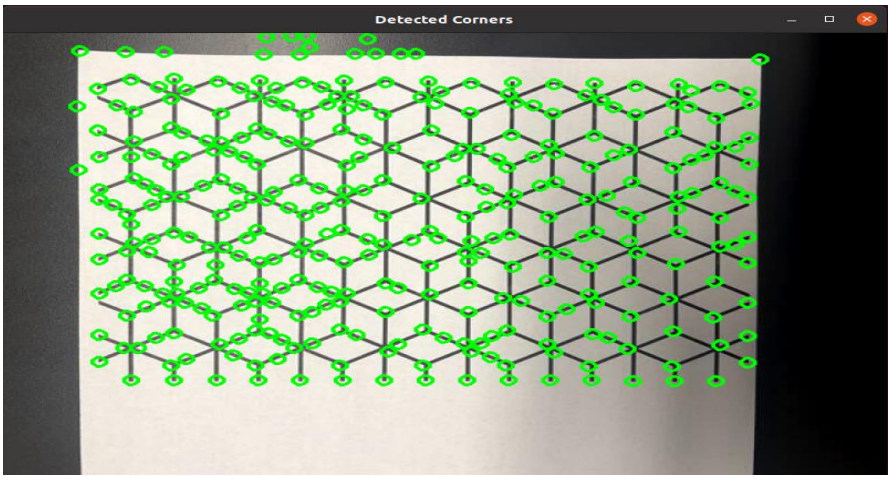

Figure 9 : Harris Corners

Figure 10: Number of Harris Corners Detected

- **Pattern 2:**

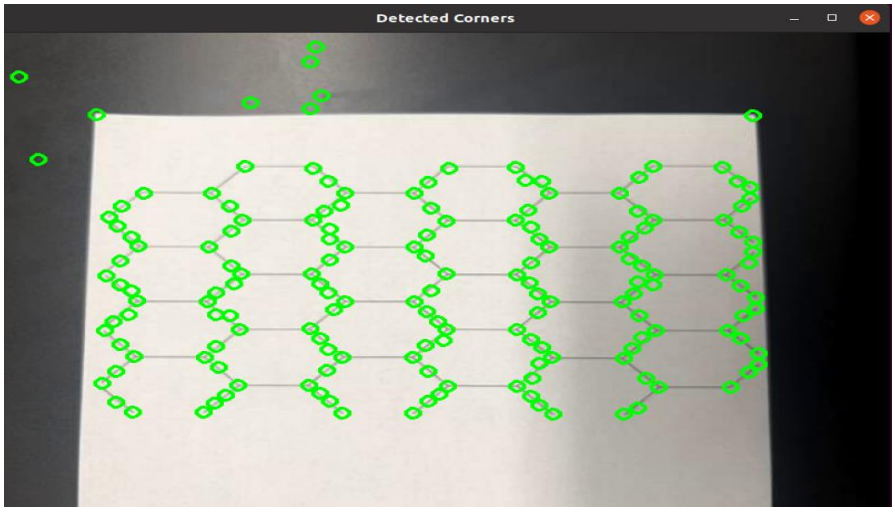

Figure 11:Harris Corners

Figure 12 : Number of Harris Corners Detected

- **Pattern 3:**

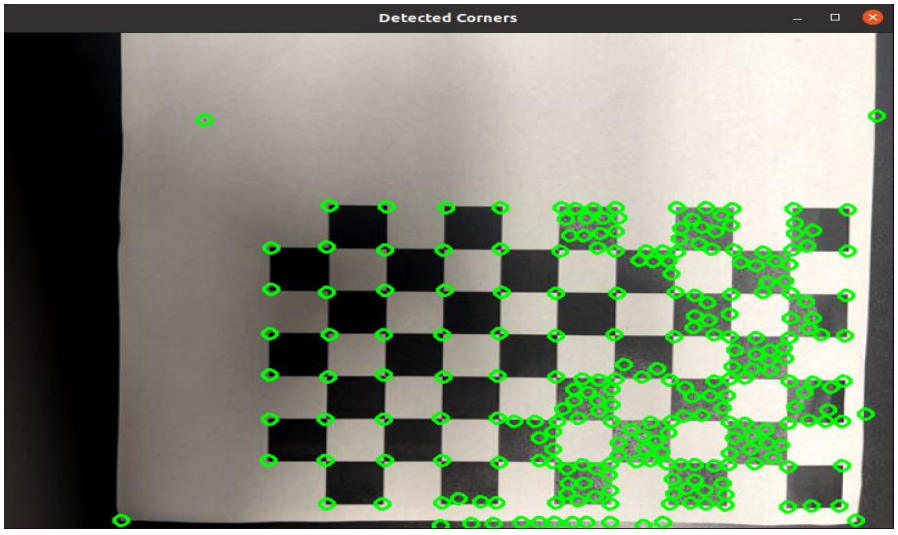

Figure 13: Harris Corners

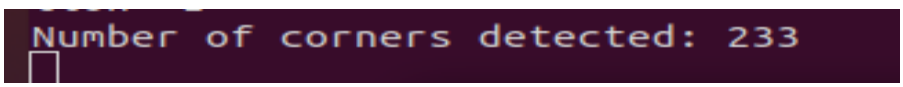

Figure 14: Number of Harris Corners Detected

- **Pattern 4:**

Figure 15: Harris Corners

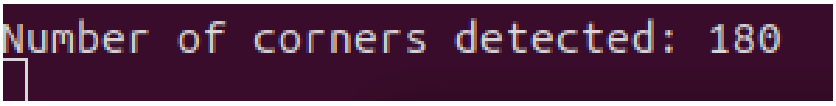

Figure 16 : Number of Harris Corners Detected

Case 2 : Quality Level = 0.1

- **Pattern 1:**

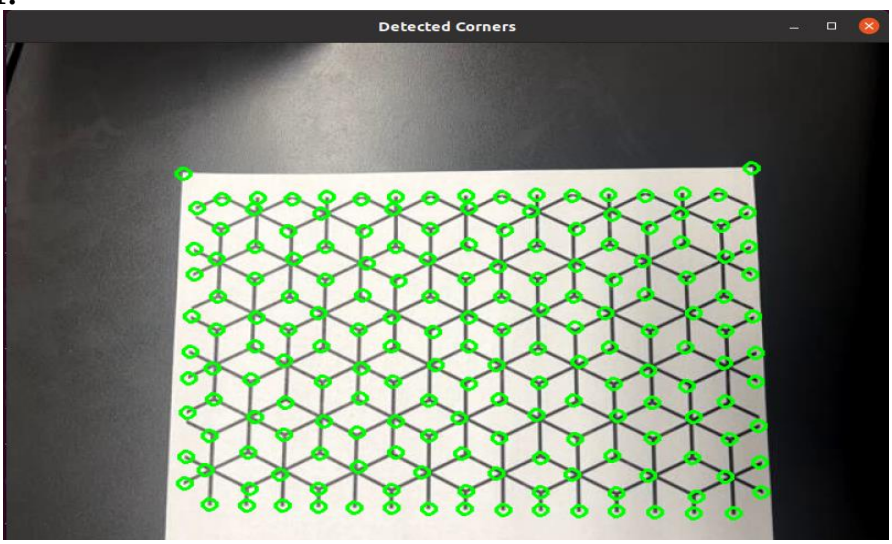

Figure 177 : Harris Corners

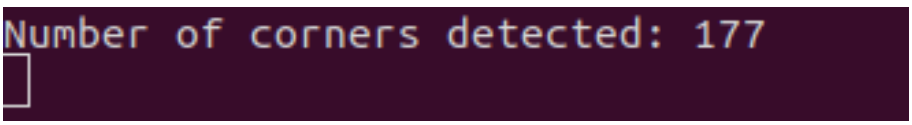

Figure 18: Number of Harris Corners Detected

- **Pattern 2:**

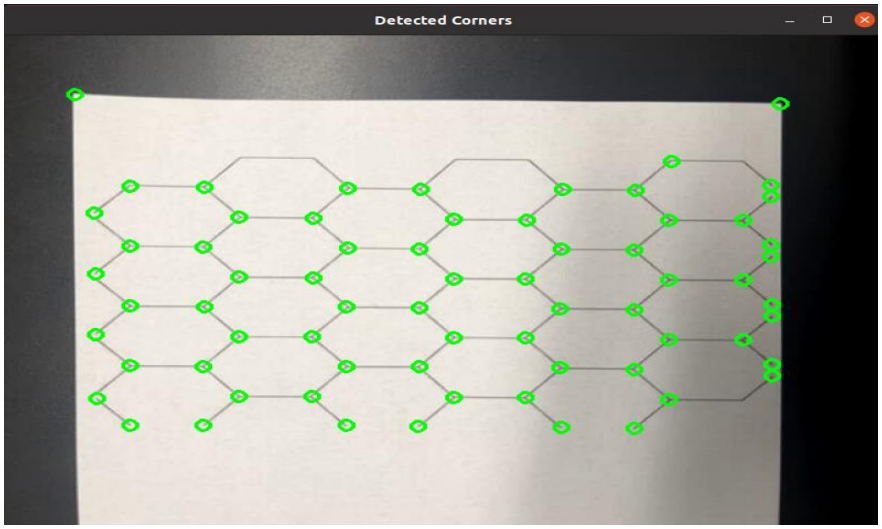

Figure 19:Harris Corners

Figure 20 : Number of Harris Corners Detected

- **Pattern 3:**

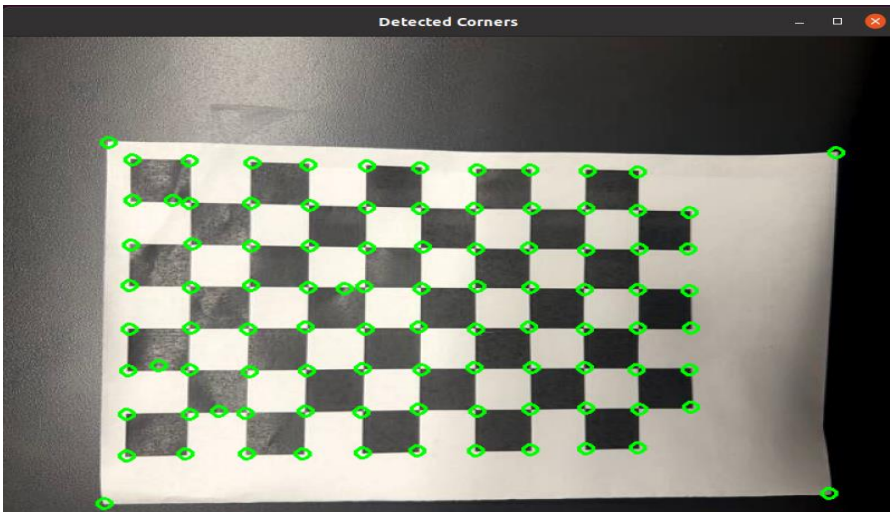

Figure 21: Harris Corners

Figure 22: Number of Harris Corners Detected

- **Pattern 4:**

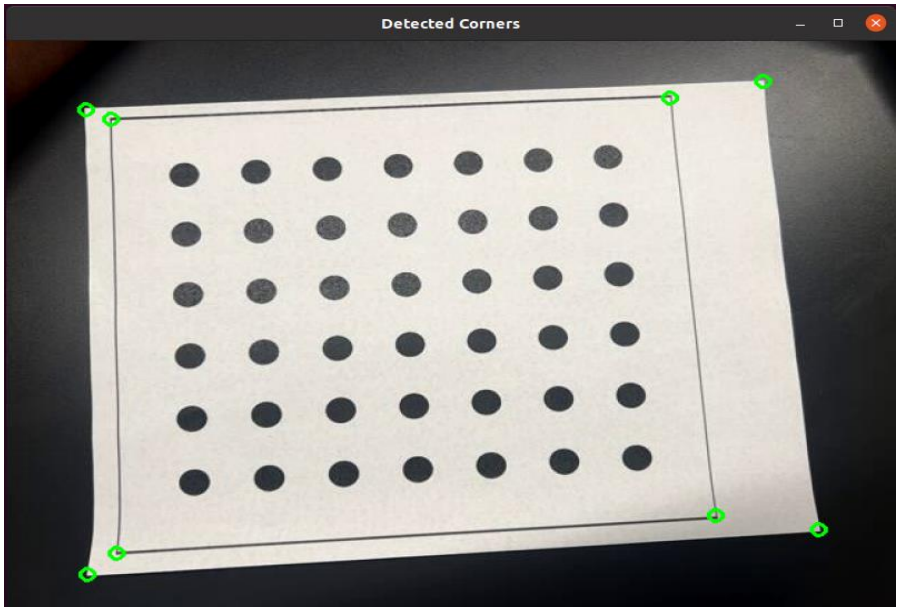

Figure 23: Harris Corners

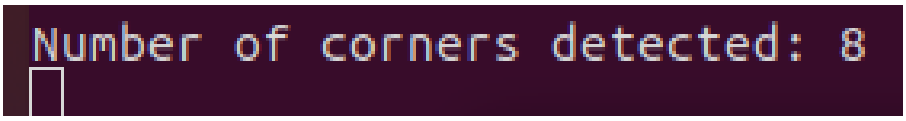

Figure 24 : Number of Harris Corners Detected

- **Quality Level = 0.01:**
  With a lower quality level, more corners will be detected, including those with lower quality. This may result in detecting a larger number of corners, some of which may not be as meaningful or reliable. However, it ensures a broader coverage of potential features in the image.
- **Quality Level = 0.1:**
  With a higher quality level, only corners with higher quality are considered. This might lead to detecting fewer corners, but they are likely to be more reliable and correspond to more salient features in the image. However, there's a risk of missing some less prominent features.

Using The Feature Points as The Basis for Putting Augmented Reality into The Image:
Augmented reality (AR) integrates digital content with the real world, and feature points play a critical role in this technology. Detected using algorithms like Harris corner detection, these points mark significant changes in image intensity, providing stable references for positioning virtual objects within a real scene. To introduce AR elements, these feature points are matched with corresponding points on a 3D model, allowing for the calculation of the camera's pose relative to the scene. By continuously updating this process in real-time, virtual objects can be depicted seamlessly onto the video stream, responding naturally to camera movements and changes in the environment. This blend of virtual and real elements creates immersive AR experiences, enhancing interactivity and realism through accurate positioning, rendering, and possibly user interaction with the augmented content.

EXTENSION 1:

Input Parameters:

- **cv::Mat &src:** This is a reference to the input image frame where circle centers need to be detected.
- **cv::Mat &dst:** A reference to the output image frame where detected circle centers may be drawn.
- **std::vector<cv::Point2f> &centers:** A vector to hold the image pixel coordinates of detected circle centers.
- **bool drawCenters:** A Boolean flag indicating whether to draw the detected circle centers on the output frame or not.

Circle Extraction and Grid Detection:

The **circleExtractCenters** function detects the centers of circles present in a circle grid within the input image frame. It utilizes OpenCV's **findCirclesGrid** function to detect the grid pattern based on circles. Detected centers are stored in the provided vector, and optionally, they can be drawn on the output image frame. This function enables the identification of specific points in the input image, which is crucial for camera calibration.

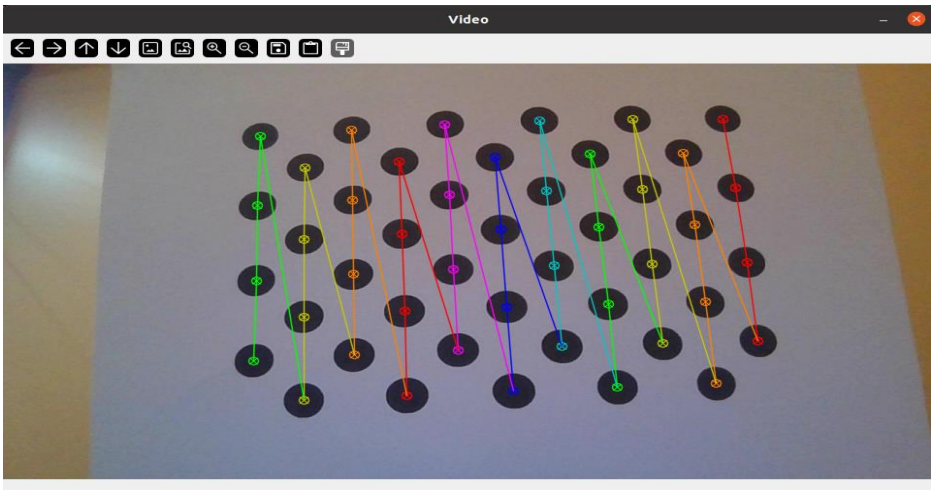

Figure 25: Circles Detected

```
rotation matrix: [2.251700497222867;
 -0.03687998117483337;
 0.1406633395153955]

translation matrix: [-5.450129051384487;
 12.22119658701738;
 78.62930484725658]
```

Figure 26 : Rotation & Translation Matrices

EXTENSION 2:

Input Parameters:

- **src:** The input image frame where the artwork will be drawn.
- **dst:** The output image frame with the artwork drawn on the target.
- **camera_matrix** and **dist_coeff**: Calibration parameters representing the intrinsic camera matrix and distortion coefficients.
- **rot and trans**: Rotation and translation matrices representing the position and orientation of the camera relative to the target.
- **img_filename**: Filename of the artwork image to be drawn on the target.

Perspective Transformation:

The function first reads the artwork image from the provided filename. It defines four source points **(inputQuad)** representing the corners of the artwork image. Four destination points **(outputQuad)** are determined by projecting 3D world coordinates of the corners of the target onto the 2D image plane using the provided rotation (rot) and translation (trans) matrices, along with the camera matrix and distortion coefficients. A perspective transformation matrix (lambda) is computed using **getPerspectiveTransform()** function from OpenCV, mapping the source points to the destination points.

Drawing the Artwork:

The function then warps the artwork image onto the target area in the output image frame using **warpPerspective()** function from OpenCV. This applies the perspective transformation calculated earlier to the artwork image. Before drawing the artwork, it fills the target area in the input image frame with a black color to create a masking effect. This ensures that only the target area is modified when overlaying the artwork. Finally, the input image frame is copied onto the output image frame, ensuring that the rest of the input image remains unchanged while the artwork is drawn on the target.

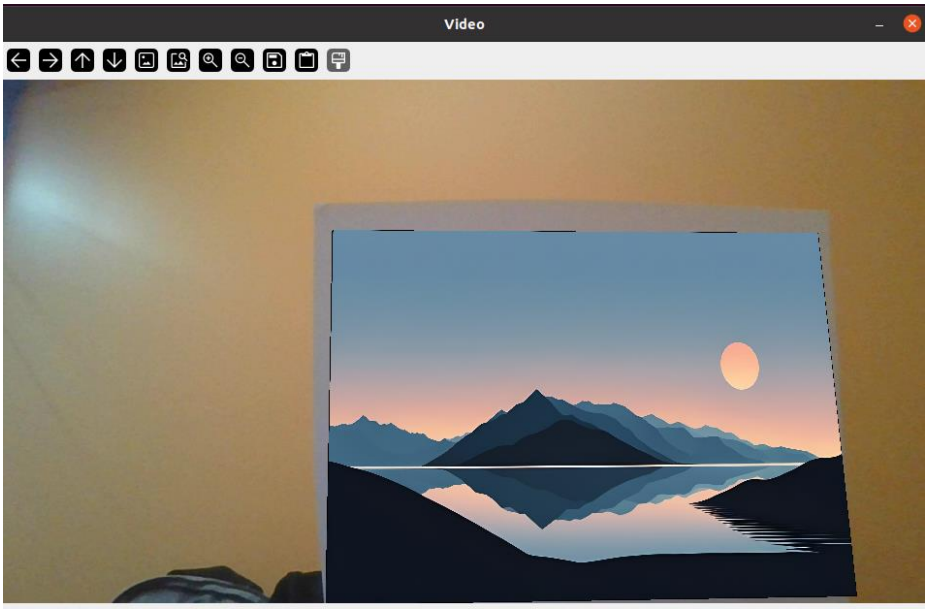

Figure 27: Artwork on Target

EXTENSION 3:

Upon testing out several different cameras and comparing the calibrations and quality of the results.
The reprojection error serves as a metric for evaluating the accuracy of the camera calibration. It measures the average discrepancy between the projected 2D image points and their corresponding actual 2D image points. The comparative study revealed significant differences in reprojection errors among the three cameras:

1. The laptop camera recorded a higher reprojection error of approximately 0.9, indicating less precise calibration and higher inaccuracies in spatial measurements.
2. In contrast, the iPhone 13 Pro exhibited a lower reprojection error of 0.4, indicating better calibration quality and higher spatial measurement accuracy.
3. The Logitech webcam performed even better, with a reprojection error of around 0.3, indicating superior calibration precision and spatial measurement accuracy compared to both the laptop camera and iPhone 13 Pro.

Cameras with advanced optics and sensors, such as the iPhone 13 Pro and high-end Logitech webcams, offer superior precision in calibration and spatial mapping.
In contrast, the basic hardware of typical laptop cameras results in less accurate measurements, impacting the reliability of data for applications demanding precise spatial mapping, such as augmented reality and 3D modeling.

REFLECTION ON WHAT WE LEARNT:

1. Camera Calibration:

   We gained insights into the process of camera calibration, which involves estimating intrinsic parameters like focal length and distortion coefficients. By implementing calibration techniques, we learned how to correct lens distortions in images, which is crucial for accurate computer vision applications like object tracking and augmented reality.

   We also observed that the corners detected are high in accuracy in the case of the checkerboard than that of the circle grid.

2. 3D Object Projection:

   Through this project, we learned about the process of projecting virtual 3D objects onto real-world scenes captured by a camera. We gained knowledge about transformation matrices and how they are used to position and orient virtual objects relative to the camera's viewpoint.

3. Augmented Reality Integration:

   We gained knowledge about matching feature points detected in the scene with corresponding points on a 3D model. Calculating the camera's pose relative to the scene for seamless rendering of virtual objects. Practical applications of augmented reality in creating immersive experiences.

ACKNOWLEDGEMENTS:

I am grateful to the OpenCV documentation, which has been a vital resource during this project. Understanding and putting different computer vision algorithms and techniques into practice has been made possible by the comprehensive explanations given in the OpenCV documentation.

I would also like to thank my peers for their help and feedback during this project. Throughout this process, exchanging knowledge, working together to troubleshoot challenges, and sharing ideas have all been extremely beneficial .