Project 4 Report

Name: Xujia Qin

Date: March 13th, 2025

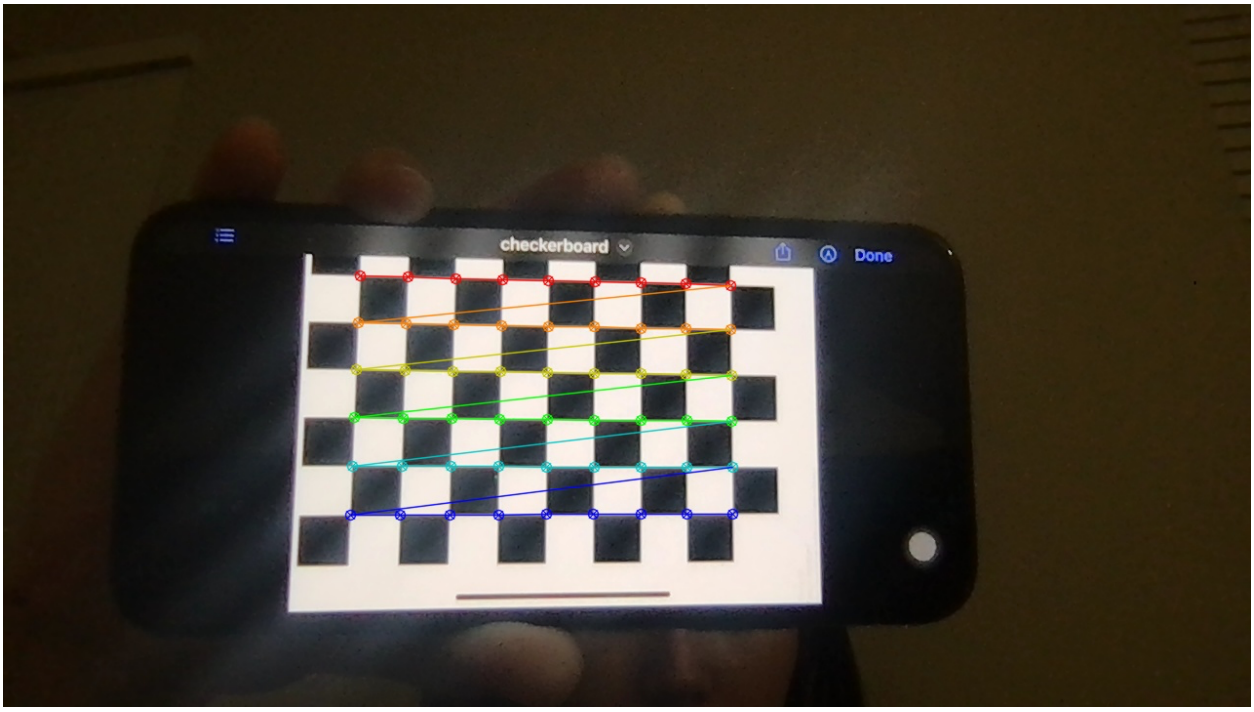
Description:

The objective of this project is to accurately render a virtual object onto a physical target. The initial phase involves camera calibration, where I establish a threshold of five images. Once these images are captured, a calibration-ready frame is generated. Using the collected image coordinates in conjunction with their corresponding geographic coordinates, I calibrate the camera matrix and distortion coefficients. Subsequently, I design and model a 3D virtual object and generate its 2D projection. Then explore various techniques for integrating virtual images, including corner detection, calibration with different devices, and experimentation with aruco markers for enhanced accuracy.

Extensions:

1. Especially creative virtual objects and scenes are a good extension. It is possible to integrate OpenCV with OpenGL to render shaded objects (big extension).
2. Test out several different cameras and compare the calibrations and quality of the results. Calibration was performed using two distinct devices: an iPhone X and a MacBook Pro.
3. Enable your system to use static images or pre-captured video sequences with targets and demonstrate inserting virtual objects into the scenes.

Task 1: Detect and Extract Target Corners



Console output:

```
Number of corners detected: 54
Coordinates of the first corner (upper-left): (364, 277)
```

The system uses a chessboard pattern with opencv as the target for corner detection. The chessboard is a grid of alternating black and white squares, with a known number of inner corners (e.g., 9x6 for a 10x7 grid of squares).

Limitation: The detection of the chessboard corners depends on both lighting conditions, distance, and orientation; poor lighting, shadows, or reflections, as well as extreme angles or rotations, can significantly reduce accuracy. Ensuring the chessboard is well-lit and properly aligned in the frame enhances the coherence of detection results.

Task 2: Select Calibration Images

When the user presses ‘s’, the current image is marked for calibration, and the detected chessboard corners along with their corresponding 3D world coordinates are saved.

The system retains the recent images and their detected corners. Each time a set of corners is stored, a point_set is generated to define their 3D positions in world coordinates. These 3D positions remain consistent regardless of the chessboard’s orientation.

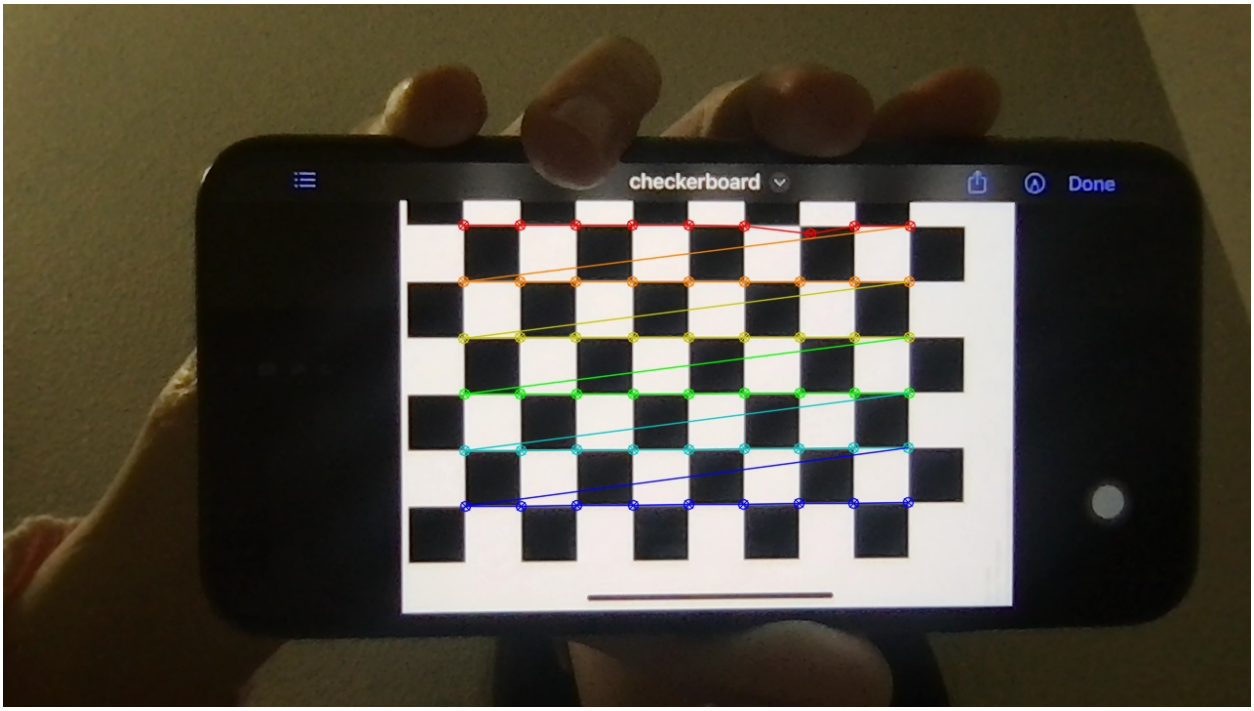
Console output:

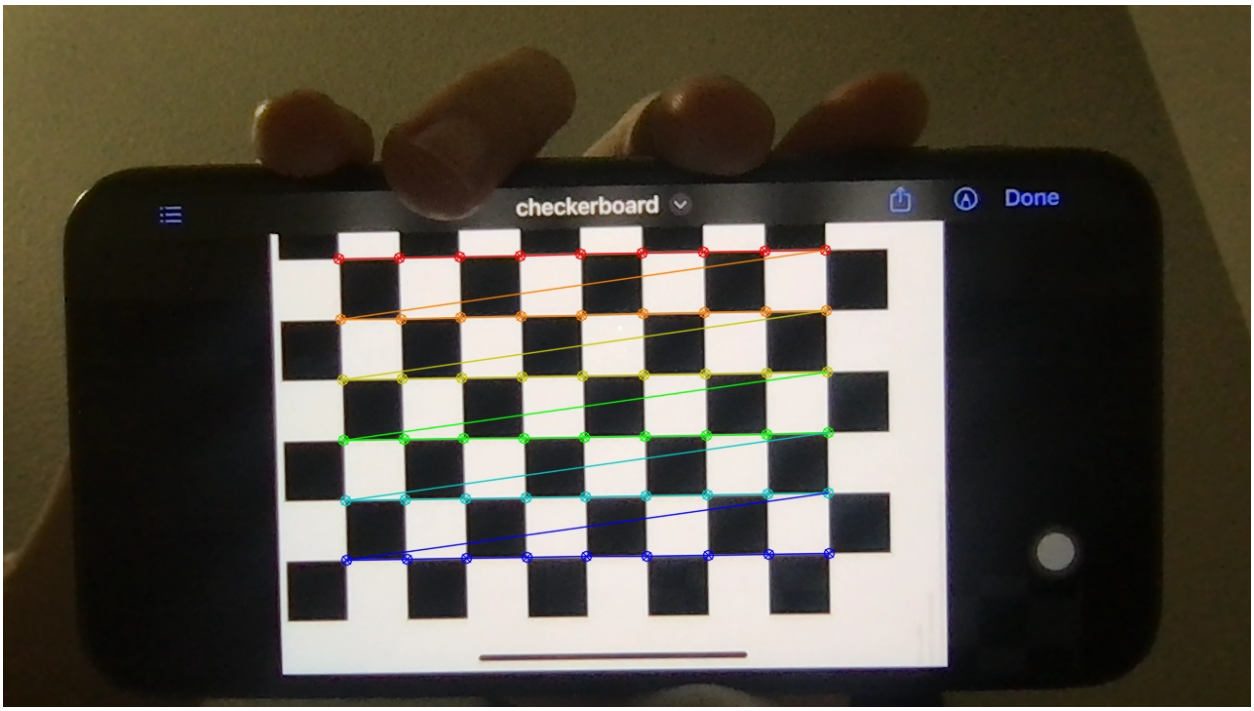
```
Image and corners saved!

=== Stored Corner Points (2D) ===
Image 0:
(472.121, 226.303) (529.683, 226.285) (587.356, 226.227) (645.451, 226.324) (702.635, 226.42) (759.75, 226.754) (828, 235) (873.273, 227.43) (929.519, 227.345)
(472.253, 284.033) (529.734, 283.915) (587.418, 283.845) (645.49, 283.597) (702.779, 283.793) (759.878, 283.882) (816.856, 284.168) (872.969, 284.07) (929.349,
284.234) (472.385, 341.716) (529.894, 341.433) (587.223, 341.003) (645.555, 341.256) (702.829, 341.069) (759.949, 341.106) (816.718, 340.854) (872.889, 340.869)
 (928.837, 340.732) (472.671, 399.478) (530.134, 399.48) (587.651, 399.308) (645.583, 399.023) (702.922, 398.808) (759.783, 398.38) (816.647, 398.248) (872.565,
 397.918) (928.764, 397.724) (473.15, 456.862) (530.603, 456.557) (587.922, 456.416) (645.804, 456.09) (702.678, 455.664) (759.722, 455.37) (816.363, 454.695) (
872.418, 454.403) (928.435, 453.901) (473.617, 513.922) (530.861, 513.609) (588.329, 513.328) (645.729, 512.863) (702.716, 512.363) (759.449, 511.629) (816.19,
511.299) (872.173, 510.571) (928.426, 510.283)
Image 1:
(344.365, 260.386) (406.284, 259.229) (467.751, 257.996) (529.994, 256.548) (592.247, 255.207) (654.854, 254.057) (718.089, 252.786) (780.633, 251.778) (843.479
, 250.62) (346.13, 322.492) (407.694, 321.517) (469.12, 320.213) (531.282, 319.04) (593.326, 317.817) (655.948, 316.486) (719.221, 315.416) (781.613, 314.345) (
844.115, 313.454) (347.814, 384.287) (409.236, 383.2) (470.424, 382.165) (532.498, 381.191) (594.389, 380.165) (657.122, 379.111) (719.959, 377.85) (782.591, 37
6.767) (844.914, 375.684) (349.338, 446.222) (410.683, 445.358) (471.999, 444.245) (533.983, 443.591) (595.797, 442.658) (658.088, 441.652) (721.048, 440.68) (7
83.231, 439.5) (845.646, 438.391) (350.688, 507.733) (412.188, 506.542) (473.556, 505.701) (535.549, 504.875) (597.059, 504.087) (659.293, 503.351) (721.753, 50
2.222) (783.956, 501.332) (846.245, 500.157) (352.202, 569.055) (413.632, 568.058) (474.888, 567.028) (536.717, 566.162) (598.455, 565.429) (660.155, 564.606) (
722.72, 563.702) (784.546, 562.727) (847.12, 562.016)
```

```
=== Stored World Points (3D) ===
Image 0:
(0, 0, 0) (1, 0, 0) (2, 0, 0) (3, 0, 0) (4, 0, 0) (5, 0, 0) (6, 0, 0) (7, 0, 0) (8, 0, 0) (0, -1, 0) (1, -1, 0) (2, -1, 0) (3, -1, 0) (4, -1, 0) (5, -1, 0) (6,
-1, 0) (7, -1, 0) (8, -1, 0) (0, -2, 0) (1, -2, 0) (2, -2, 0) (3, -2, 0) (4, -2, 0) (5, -2, 0) (6, -2, 0) (7, -2, 0) (8, -2, 0) (0, -3, 0) (1, -3, 0) (2, -3, 0)
 (3, -3, 0) (4, -3, 0) (5, -3, 0) (6, -3, 0) (7, -3, 0) (8, -3, 0) (0, -4, 0) (1, -4, 0) (2, -4, 0) (3, -4, 0) (4, -4, 0) (5, -4, 0) (6, -4, 0) (7, -4, 0) (8, -
4, 0) (0, -5, 0) (1, -5, 0) (2, -5, 0) (3, -5, 0) (4, -5, 0) (5, -5, 0) (6, -5, 0) (7, -5, 0) (8, -5, 0)
Image 1:
(0, 0, 0) (1, 0, 0) (2, 0, 0) (3, 0, 0) (4, 0, 0) (5, 0, 0) (6, 0, 0) (7, 0, 0) (8, 0, 0) (0, -1, 0) (1, -1, 0) (2, -1, 0) (3, -1, 0) (4, -1, 0) (5, -1, 0) (6,
-1, 0) (7, -1, 0) (8, -1, 0) (0, -2, 0) (1, -2, 0) (2, -2, 0) (3, -2, 0) (4, -2, 0) (5, -2, 0) (6, -2, 0) (7, -2, 0) (8, -2, 0) (0, -3, 0) (1, -3, 0) (2, -3, 0)
 (3, -3, 0) (4, -3, 0) (5, -3, 0) (6, -3, 0) (7, -3, 0) (8, -3, 0) (0, -4, 0) (1, -4, 0) (2, -4, 0) (3, -4, 0) (4, -4, 0) (5, -4, 0) (6, -4, 0) (7, -4, 0) (8, -
4, 0) (0, -5, 0) (1, -5, 0) (2, -5, 0) (3, -5, 0) (4, -5, 0) (5, -5, 0) (6, -5, 0) (7, -5, 0) (8, -5, 0)
```

Images:





Task 3: Calibrate the Camera

```
Camera Matrix:
[1035.7421345318 38, 0, 642.9637907399352;
 0, 1034.014099433831, 348.8566476234124;
 0, 0, 1]
Distortion Coefficients:
[-0.2474726929681517, 2.621036420666849, 0.001238210741144203, 0.0008071103150801668, -8.088367212950519]
Reprojection Error: 0.160969
Calibration data saved!
```

In this step, I save the intrinsic parameters, including the camera_matrix and distortion_coefficients, to a camera_parameters.xml file.

The calibration is performed using the cv::calibrateCamera function, which takes several parameters: the point_list and corner_list (as defined earlier), the dimensions of the calibration images, the camera_matrix, the distortion_coefficients, as well as the rotation and translation vectors.

camera_parameters.xml

```xml
<?xml version="1.0"?>
<opencv_storage>
<CameraMatrix type_id="opencv-matrix">
  <rows>3</rows>
  <cols>3</cols>
  <dt>d</dt>
  <data>
    1232.5417747772462 0. 640.07503188750024 0. 1227.7510093357487
    399.49948419228656 0. 0. 1.</data></CameraMatrix>
<DistortionCoefficients type_id="opencv-matrix">
  <rows>1</rows>
  <cols>5</cols>
  <dt>d</dt>
  <data>
    -0.41153769146022628 6.5726366651720154 0.0048000214920516458
    0.00099694993332508216 -25.40123365675008</data></DistortionCoefficients>
</opencv_storage>
```

Task 4: Calculate Current Position of the Camera

I moved the chessboard image closer to the camera, meanwhile rotates and shifts in this experiment.

In this set of results (the following screenshot), the rotation vectors show that the chessboard is significant rotational changed as it is moved closer to the camera.

The rotation values along the x-axis (first component) fluctuate between positive and negative values, suggesting that the chessboard is being tilted or rotated around the camera.

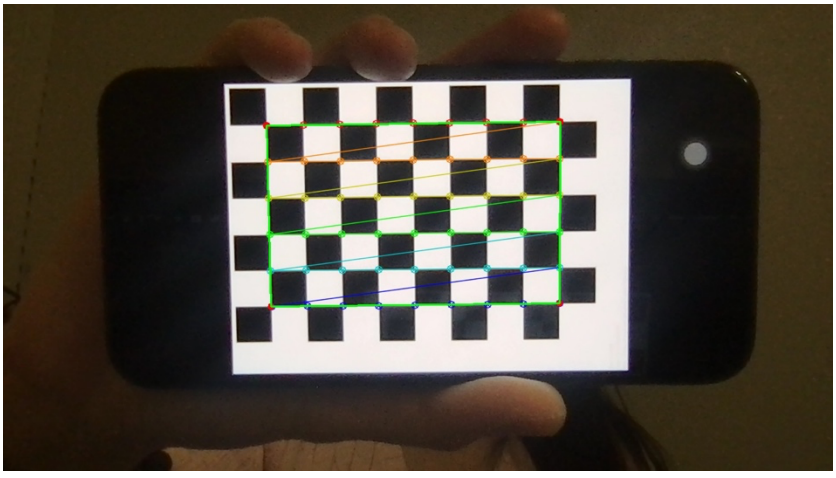
The translation vectors show that as the chessboard moves closer, the z-component (depth) decreases, which is typical as objects approach the camera.

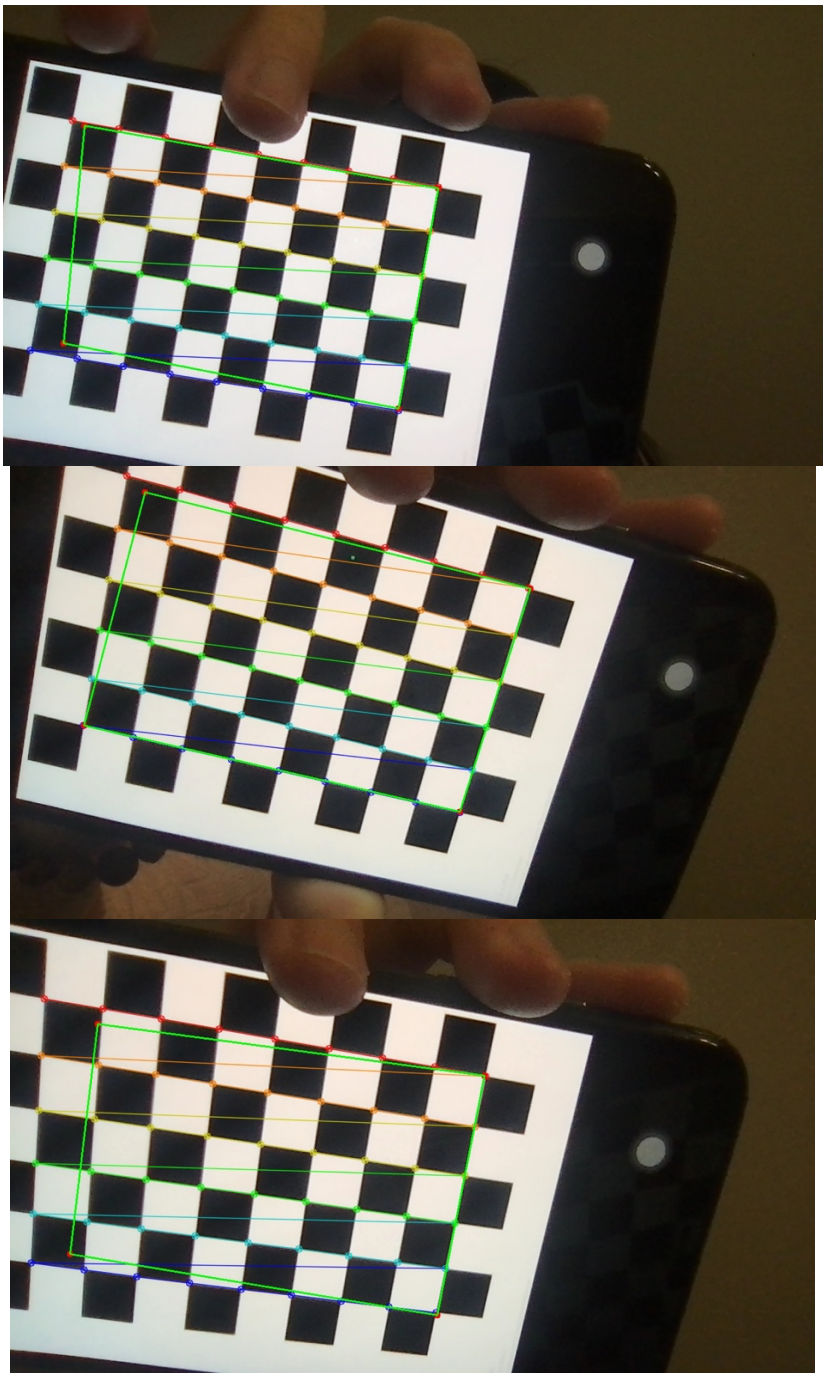
The x and y components of the translation vector also change, indicating that the chessboard is being shifted in position along those axes while maintaining its distance from the camera.

The changes in both rotation and translation values make sense when the chessboard is both rotated and moved closer to the camera.

```
Rotation Vector:
[2.963818794848384;
 -0.2970271560045592;
 0.16546674912367961
Translation Vector:
[-4.881515727917455;
 -3.7320235237336471
 22.016477115718851
Rotation Vector:
[2.975723554591699;
 -0.2931187301318144;
 0.17515460363249621
Translation Vector:
[-4.802195668257967;
 -4.202507056638536;
 23.822496647437641
Rotation Vector:
[3.0816332998055481
 -0.016574138864572131
 -0.099206227683154711
Translation Vector:
[-5.292907344493273;
 -3.26063087877346;
 16.280808929678191
Rotation Vector:
[3.070712495390166;
 -0.018052341827513571
 -0.135566551420660211
Translation Vector:
[-5.345805219928607;
 -3.195093696298259;
 16.366216457622981
Rotation Vector:
[3.077617292443025;
 -0.032241117725997841
 -0.129104207922053]
Translation Vector:
[-5.302733238487901;
 -2.988651415280822;
 16.374004896730551
Rotation Vector:
[3.102145306680254;
 -0.041823677863505491
 -0.116266327047063911
Translation Vector:
[-5.390061212346271;
 -3.213795470422485;
 16.365095684632971
Rotation Vector:
[3.115933648083592;
 -0.045811795180984771
 -0.113381648390978]
Translation Vector:
[-5.271209222982503;
 -3.212352885327983;
 16.611509239276941
```

Task 5: Project Outside Corners or 3D Axes

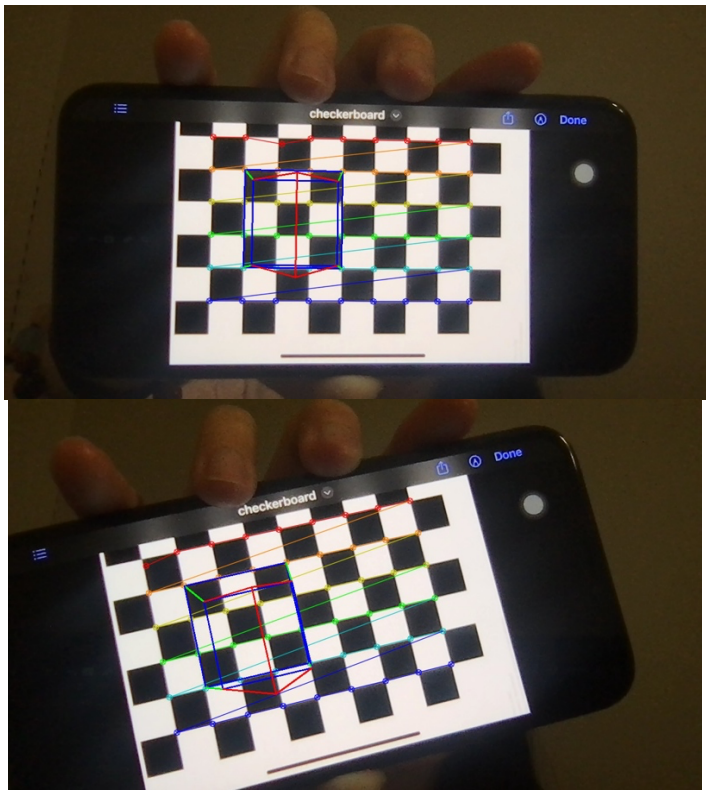




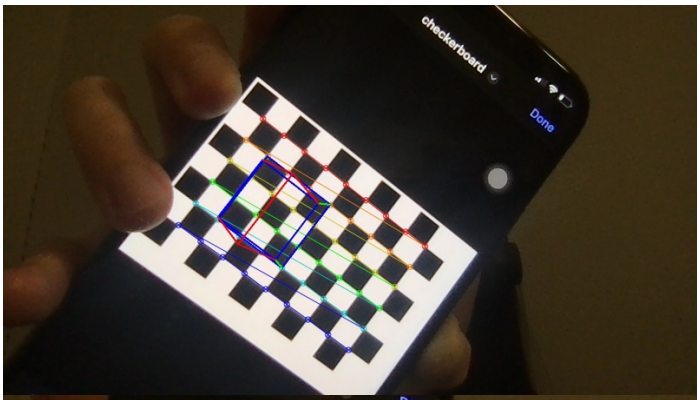
I utilize the projectPoints function to dynamically project the 3D coordinates of the four outer corners of the chessboard onto the image plane in real time as the chessboard or camera moves/ shifts/ rotates.

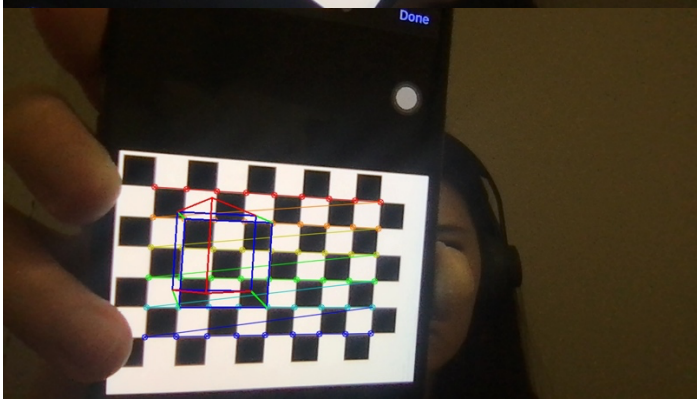
Task 6: Create a Virtual Object

I create a virtual object like a house in 3D space using lines, then project this object onto the image and draw the corresponding lines.

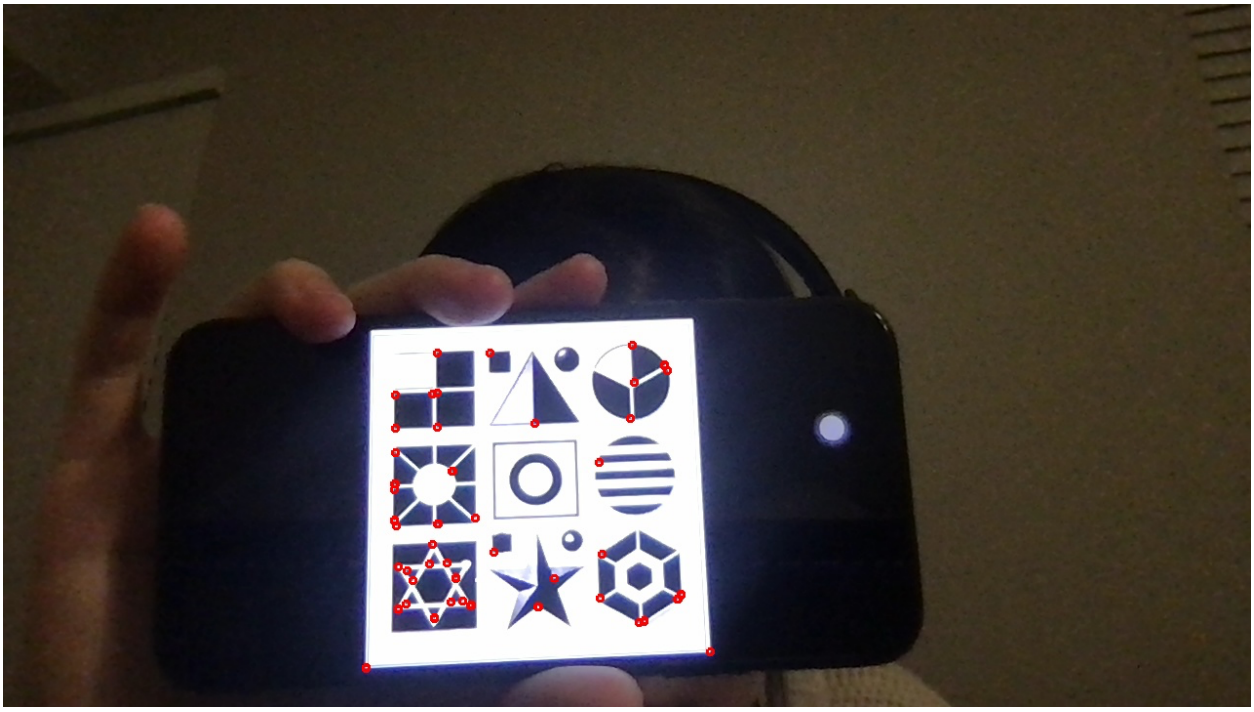

checkerboard
Done

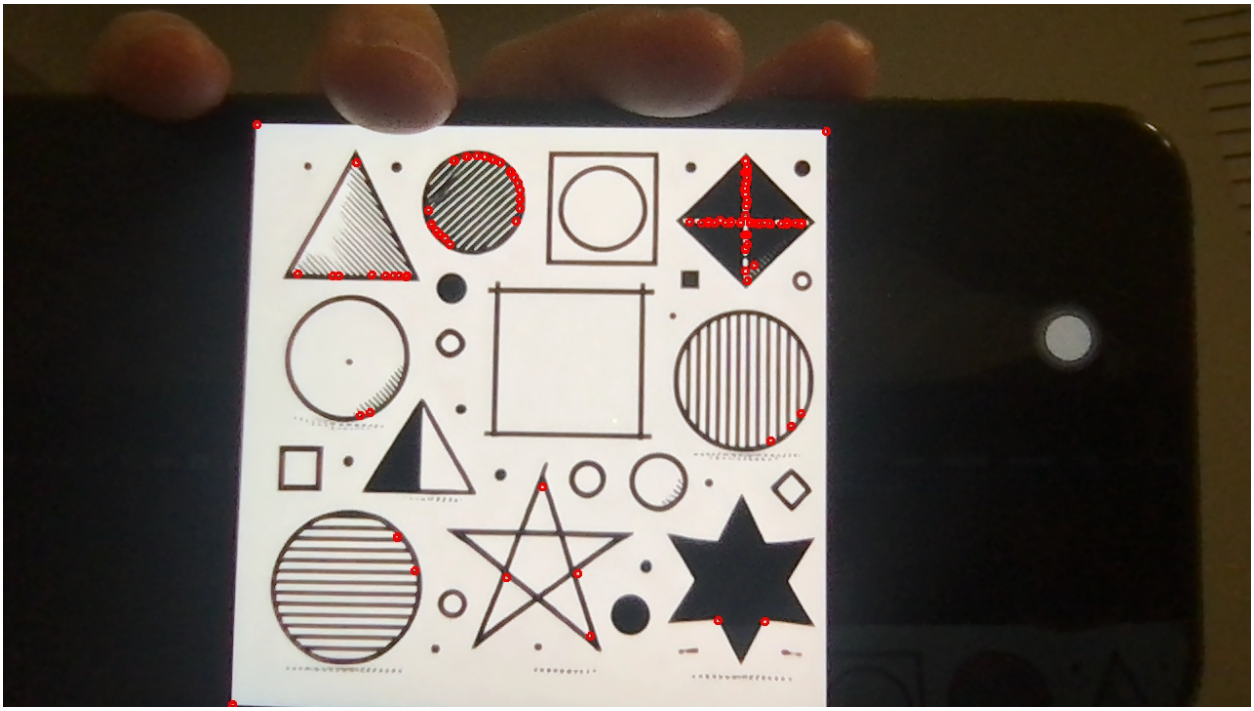
checkerboard
Done


checkerboard
Done


Done

Task 7: Detect Robust Features (Harris corners)





I selected corners as a key feature and developed a separate program to highlight Harris Corner features in a live video stream. To test this, we designed a pattern of shapes and a maze where these features appear.

Like a chessboard, two corners define a rectangular region, offering valuable spatial information when identified. By mapping corner patterns to different objects, we can build a database that enables object recognition using Harris corners. This information allows us to analyze image content and, if desired, overlay digital objects onto their corresponding real-world counterparts in the image.

Extensions:

1. Especially creative virtual objects and scenes are a good extension. It is possible to integrate OpenCV with OpenGL to render shaded objects (big extension).

   Demo video:
   https://drive.google.com/file/d/1O1Imv73y3Sq59AcuuVki8G3ydiXq1uBy/view?usp=sharing

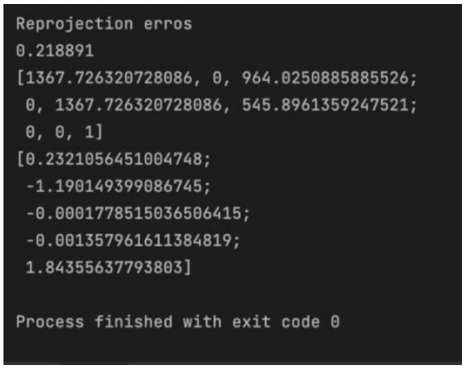
   I integrated OpenGL rendering using GLEW, GLFW, and OpenGL to display a shaded 3D house, consisting of a base and a triangular roof. OpenCV is used to detect a chessboard pattern, leveraging `solvePnP` to obtain the rotation (`rvec`) and translation (`tvec`) vectors. The projection is handled with OpenGL, where `gluLookAt()` positions the virtual house on the detected chessboard. Finally, `drawHouse()` is called to render the colored 3D house.

2. Test out several different cameras and compare the calibrations and quality of the results.

   The previous I used to be iPhone X camara, so this step I conducted calibration using two different devices: an iPhone 13 Pro Max and a MacBook Pro.

   - iPhone Camera: Calibration with the iPhone 13 Pro Max camera yielded the distortion matrix and camera matrix, resulting in a reprojection error of 0.218.

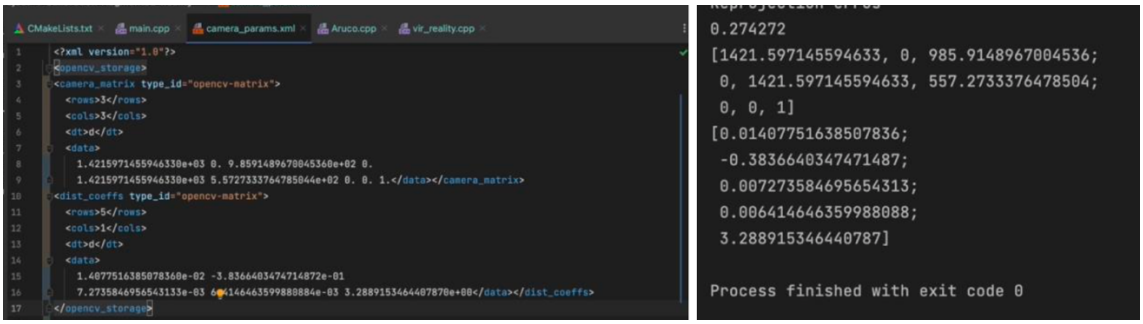
```
<?xml version="1.0"?>
<opencv_storage>
<camera_matrix type_id="opencv-matrix">
  <rows>3</rows>
  <cols>3</cols>
  <dt>d</dt>
  <data>
    1.3677263207280855e+03 0. 9.6402508858855265e+02 0.
    1.3677263207280855e+03 5.4589613592475212e+02 0. 0. 1.</data></camera_matrix>
<dist_coeffs type_id="opencv-matrix">
  <rows>5</rows>
  <cols>1</cols>
  <dt>d</dt>
  <data>
    2.3210564510047477e-01 -1.1901493990867453e+00
    -1.7785150365064151e-04 -1.3579616113848195e-03
    1.8435563779380304e+00</data></dist_coeffs>
</opencv_storage>
```



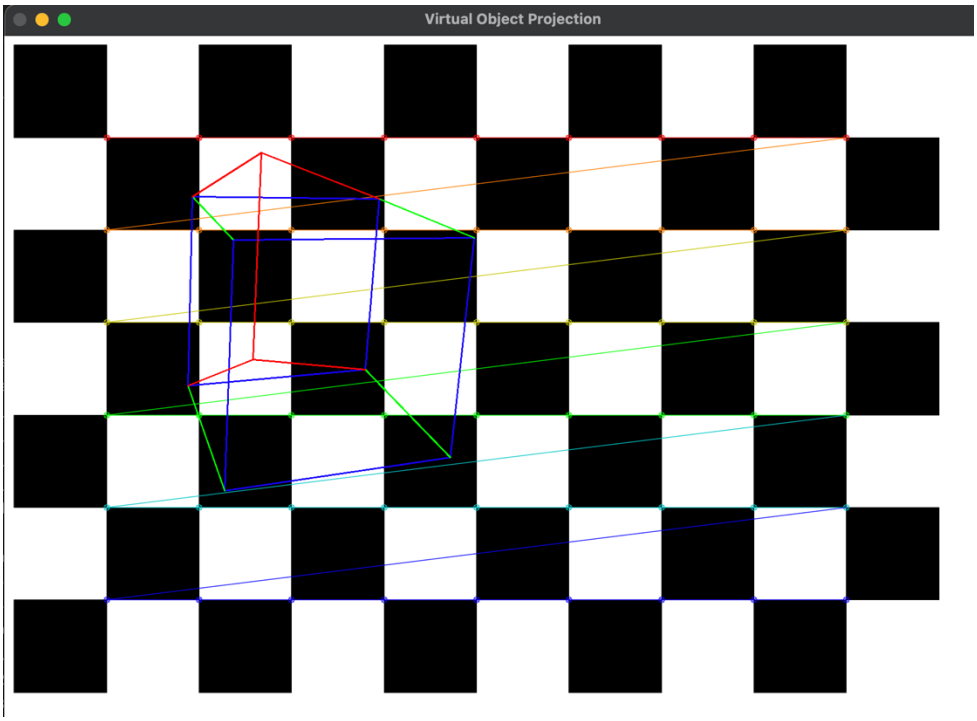
```
Reprojection erros
0.218891
[1367.726320728086, 0, 964.0250885885526;
 0, 1367.726320728086, 545.8961359247521;
 0, 0, 1]
[0.2321056451004748;
 -1.190149399086745;
 -0.0001778515036506415;
 -0.001357961611384819;
 1.84355637793803]

Process finished with exit code 0
```

   - MacBook Camera: After calibrating with the MacBook's webcam, I obtained the distortion matrix and camera matrix, with a reprojection error of 0.2742.



3. Enable your system to use static images or pre-captured video sequences with targets and demonstrate inserting virtual objects into the scenes.



This program instead use a static image as input, and then detects a chessboard in an image using OpenCV, estimates its pose with solvePnP, and projects a 3D house model onto it using camera parameters. The house, consisting of a base, walls, and a roof, is rendered as 2D projections on the image to simulate its placement in the real-world scene.

Reflections

This project has given me valuable insights into camera calibration and its functionality. A chessboard image is commonly used for this purpose, with OpenCV functions assisting in detecting its features. By extracting the pattern, I successfully projected 3D objects onto

the board, resulting in an interactive and immersive experience. Through this process, I have gained a deeper understanding of the underlying principles of camera calibration and the broader potential of computer vision. Additionally, this experience has sparked my curiosity about emerging concepts like augmented reality (AR) and virtual reality (VR).

References

Tutorial for OpenCV camera calibration:
https://docs.opencv.org/4.x/d4/d94/tutorial_camera_calibration.html

Relevant functions for detecting chessboard corners:

https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html#ga93efa9b0aa890de240ca32b11253dd4a

Tutorial for Aruco detection:

https://docs.opencv.org/4.x/d2/d1a/classcv_1_1aruco_1_1ArucoDetector.html#a0c1d14251bf1cbb06277f49cfe1c9b61

https://docs.opencv.org/4.x/de/d67/group__objdetect__aruco.html#ga2ad34b0f277edebb6a132d3069ed2909