

CS 5330

Pattern Recognition and Computer Vision

Project 4: Calibration and Augmented Reality

Team Members:

Member 1: Jyothi Vishnu Vardhan Kolla
NUID: 002752854

Member 2: Vidya Ganesh
NUID: 002766414

Overview of the overall project:

The rendering of a virtual object onto a target is the project's ultimate objective. The first step is learning how to calibrate. We've set a threshold of 5 photographs, and after those 5 images are gathered, it prints a frame that is calibration-ready. We calibrate the camera matrix and the distortion coefficients using the gathered collection of image coordinates and related geographic coordinates. We then design a virtual item in 3D and sketch it out in 2D. We then explore different techniques for embedding virtual images, detect corners, calibrated using different devices and experimented using aruco markers.

As part of extensions:

1. We first integrated the **aruco module** of **OpenCV** into our system
2. We then got the system working with multiple targets in the scene.
3. Not only did we add a virtual object, but also did something to the target to make it not look like a target anymore.
4. We performed calibration with two different devices. One from iPhone 13ProMax and the other from Mac book Pro.
5. We also used the **OpenCV arucoMarker** as targets and overlayed virtual images on it on a live video sequence.

Required Image 1: Detect and Extract Chessboard Corners

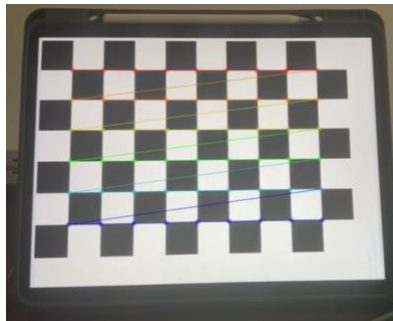


Fig Set 1: From the output it can be observed that targets are detected, and target corners are extracted from the chess board.

When it locates a chessboard in the live stream, it draws the corners of the board. Moreover, it prints the number of corners found on the chessboard as well as the top left corner's location.

Required image 2: Select Calibration Images: *Include a calibration image with chessboard corners highlighted in your project report.*

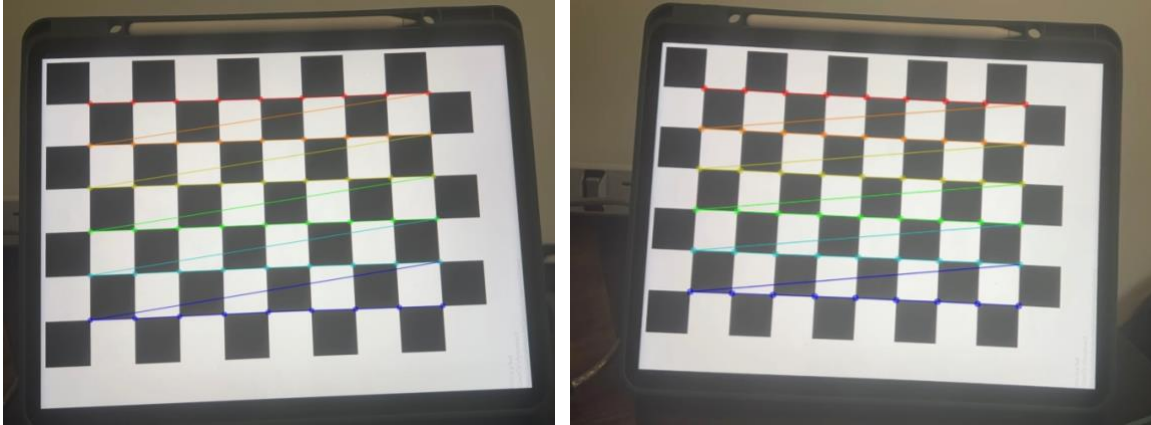


Fig Set 2 : *Example include a calibration image with chessboard corners highlighted.*

The user types 's' to specify that a particular image should be used for the calibration and saves the corner locations and the corresponding 3D world points. We save the most recent 5 images and chessboard corners in variables. Whenever we save a list of found corners, we create a point_set that specifies the 3D position of the corners in world coordinates. The point_set of 3D positions will always be the same, regardless of chessboard orientation.

Required Image 3: Calibrate the Camera: *Include the error estimate in your report.*

```
Reprojection errors
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
```

Fig Set 3.1: Error estimate

```

1  <?xml version="1.0"?>
2  <opencv_storage>
3  <camera_matrix type_id="opencv-matrix">
4    <rows>3</rows>
5    <cols>3</cols>
6    <dt>d</dt>
7    <data>
8      1.3677263207280855e+03 0. 9.6402508858855265e+02 0.
9      1.3677263207280855e+03 5.4589613592475212e+02 0. 0. 1.</data></camera_matrix>
10 <dist_coeffs type_id="opencv-matrix">
11   <rows>5</rows>
12   <cols>1</cols>
13   <dt>d</dt>
14   <data>
15     2.3210564510047477e-01 -1.1901493990867453e+00
16     -1.7785150365064151e-04 -1.3579616113848195e-03
17     1.8435563779380304e+00</data></dist_coeffs>
18 </opencv_storage>

```

Fig Set 3.2: Intrinsic Parameters

In this step we enable the user to write out the intrinsic parameters to a file: both the **camera_matrix** and the **distortion_coefficients**. We use the **cv::calibrateCamera** function to generate the calibration. The parameters to the function are the **point_list**, **corner_list** (definitions above), the size of the calibration images, the **camera_matrix**, the **distortion_coefficients**, the rotations, and the translations.

Required Image 4: Calculate Current Position of the Camera: *Rotation and translation outputs.*

```

Camera Matrix[1367.726320728086, 0, 964.0250885885526,? 0, 1367.726320728086, 545.8961359247521,? 0, 0, 1]
Distortion coefficients[0.2321056451004748,? -1.190149399086745,? -0.0001778515036506415,? -0.001357961611384819,? 1.843556377938030]

```

Fig Set 4: Rotation and translation outputs

The program reads the camera calibration parameters from a file and then starts a video loop. For each frame, it tries to detect a chessboard. If found, it grabs the locations of the corners, and then uses **solvePNP** to get the board's pose (rotation and translation).

Task 5: Project Outside Corners or 3D Axes: Include at least one image from this step in your report.

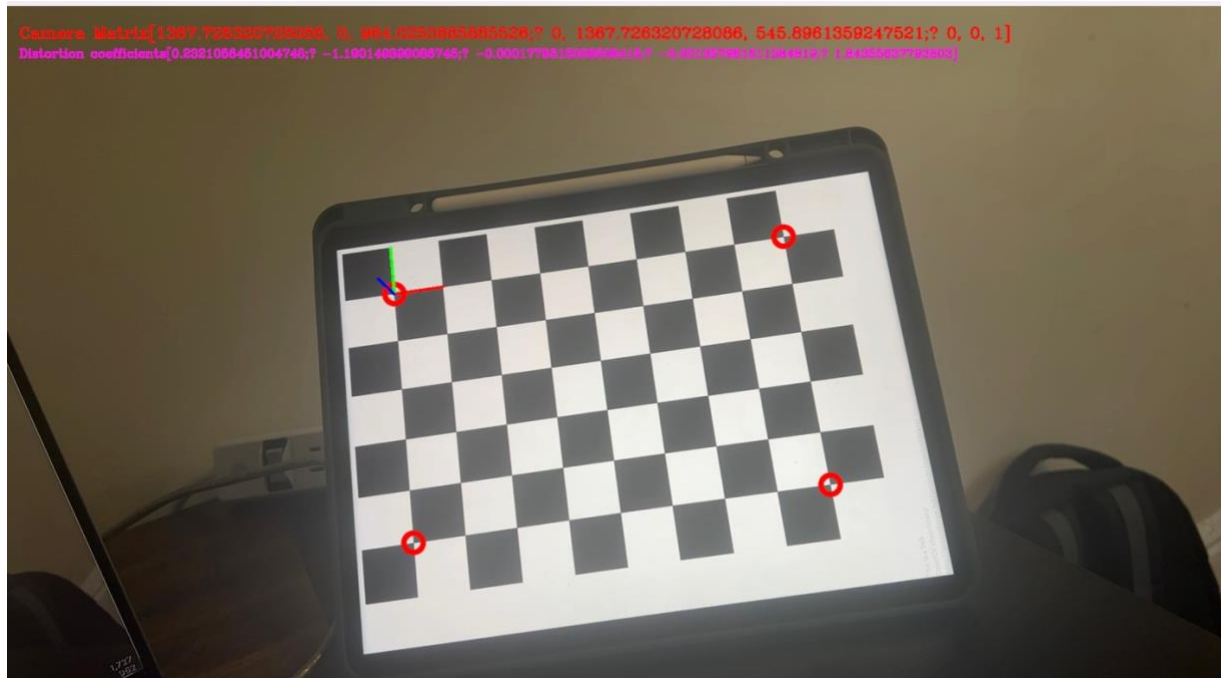


Fig Set 5: Detected 4 outside corners

We use the **projectPoints** function to project the 3D points corresponding to the four outside corners of the chessboard onto the image plane in real time as the chessboard or camera moves around.

Required Image 6: Create a Virtual Object: Take some screen shots and/or videos of your system in action for your project report.

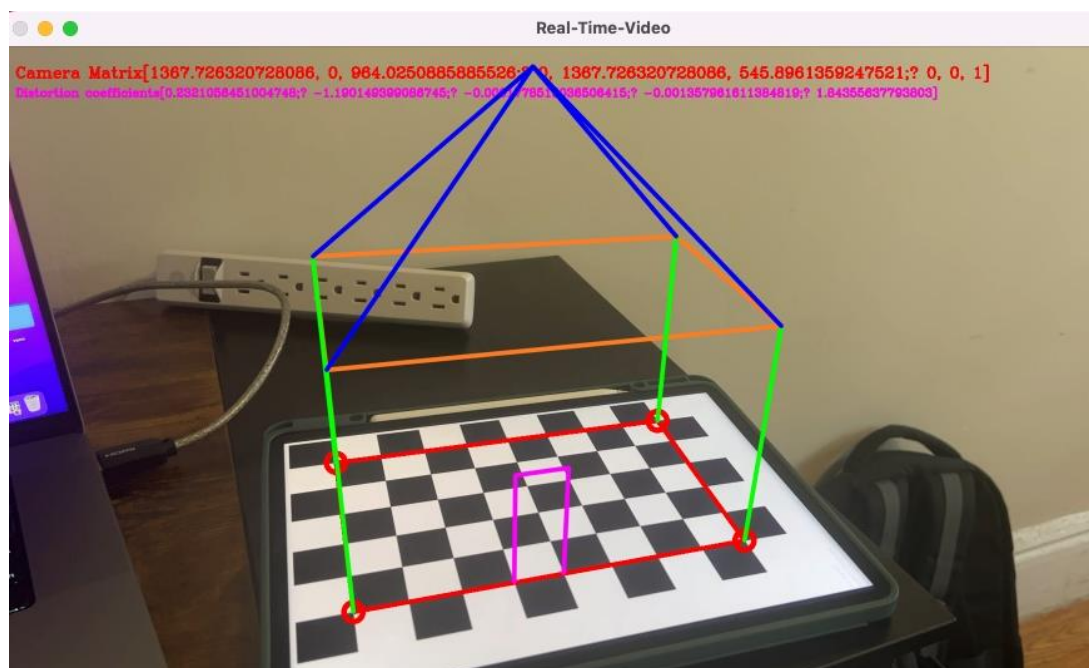


Fig Set 5: Demonstration of creating a virtual object on a video sequence.

We construct a virtual object in 3D world space made from lines. We then project that virtual object to the image and draw lines in the image.

Task 7: Detect Robust Features: *Harris corner detection output*

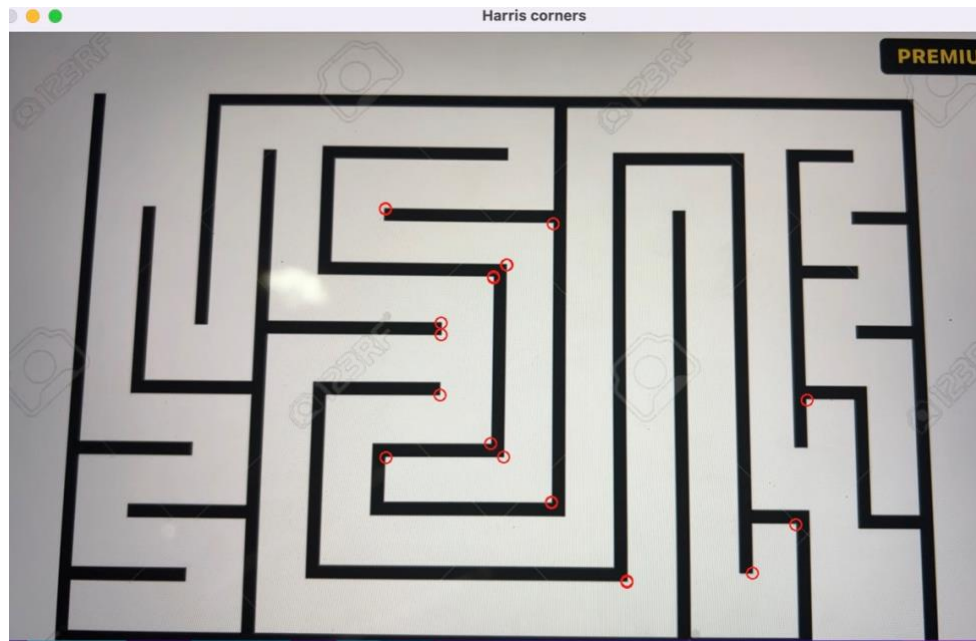


Fig Set 7.1: Harris Corner detection on a maze image.

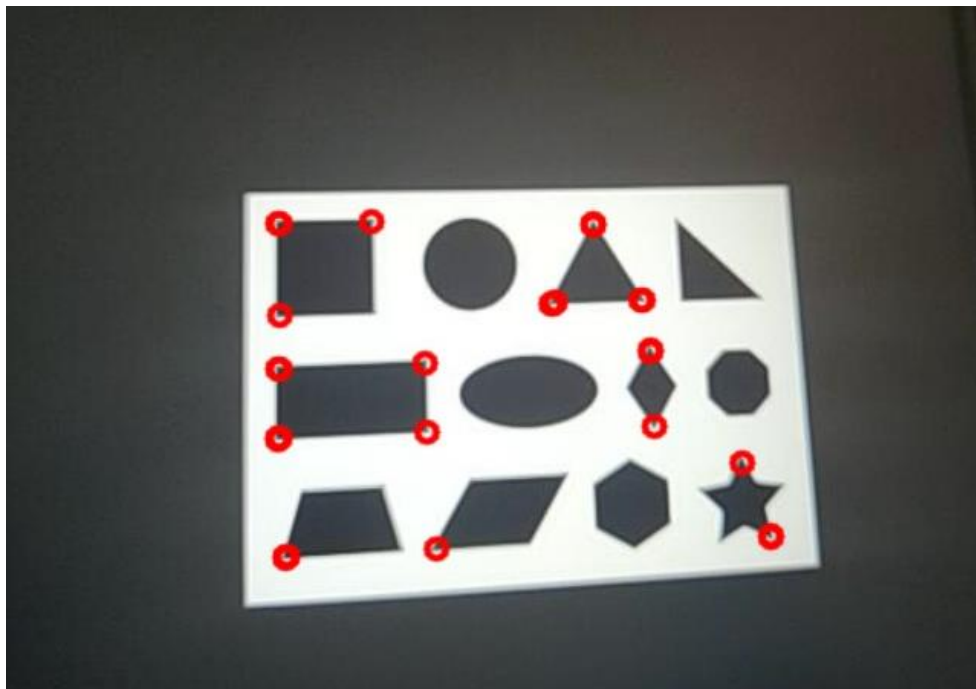


Fig Set 7.2: Harris Corner detection on an image with shapes.

We picked the corners as a feature and wrote a separate program that shows where the Harris Corner features are in the image in a live video stream. We made a pattern of shapes, and a maze were the features show up.

Like the chess game it's understood that two corners form a rectangular region, providing substantial information when located. The corner pattern can be utilized to create a database mapping corner patterns to various objects. With this information, we can identify the contents of an image using Harris corners. Subsequently, we can opt to project specific digital objects onto their corresponding real-life objects in the picture.

Extensions

Extension 1: We integrated the **aruco module** of **OpenCV** with our system

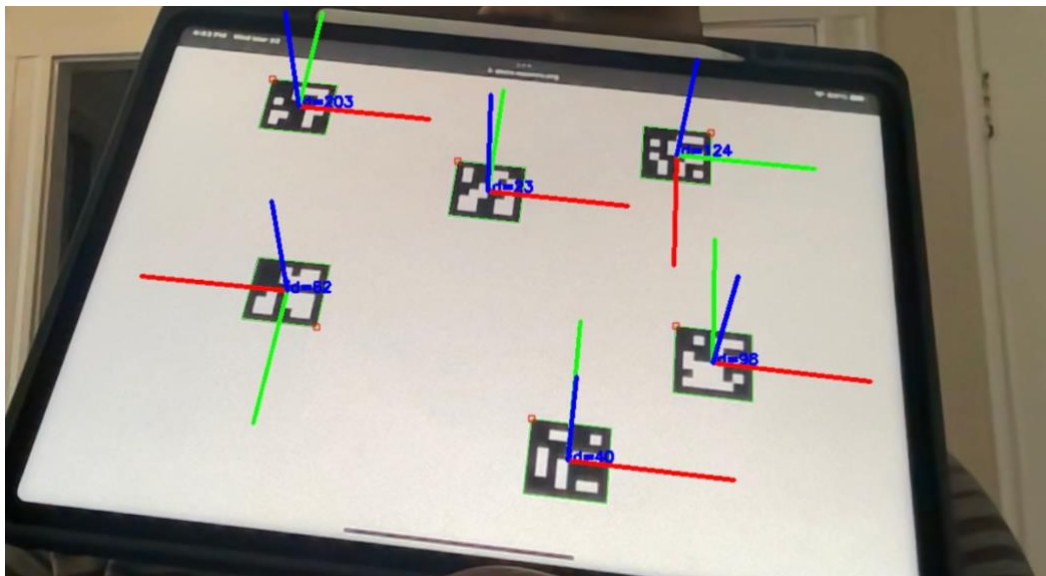


Fig Set Ext 1: The image shows the system detecting the aruco Markers and displays the 3-Dimensional coordinates for each of the Markers

We integrated the **arucoModule** with our system and identified the markers and got the bounding boxes associated with each marker. Finally, we display the 3D coordinated at each of the markers.

Extension 2: From the previous setup we got the system working with multiple targets in the scene.

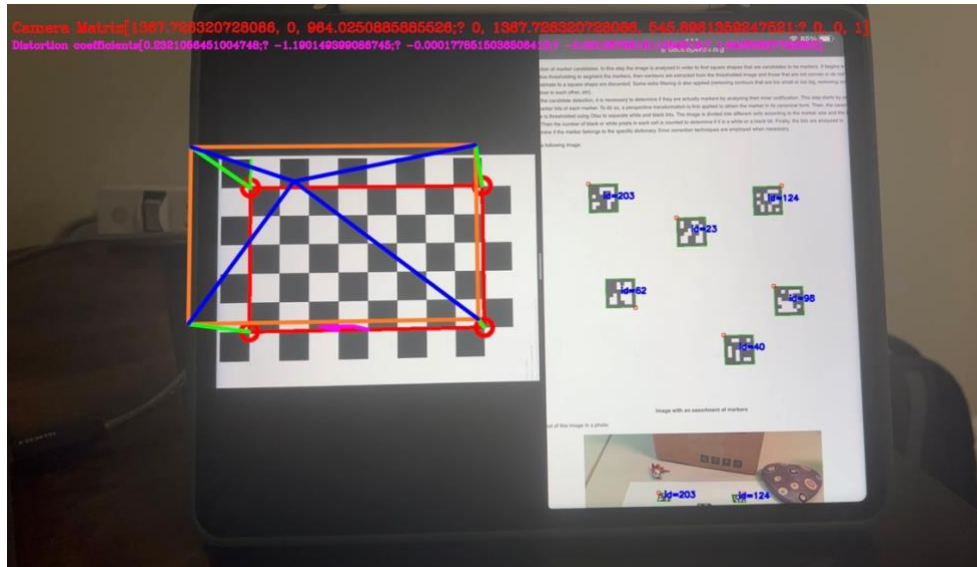


Fig Set Ext 2: *The image shows that the system detects the chess board and projects a virtual image on top of it. Simultaneously we also detect the aruco markers and produce a bounding box around each of the markers.*

The extension detects the chessboard and projects a digital image onto it. Additionally, it concurrently recognizes the ArUco markers and draws a rectangular bounding box around them.

Extension 3: Not only did we add a virtual object, but also did something to the target to make it not look like a target anymore.

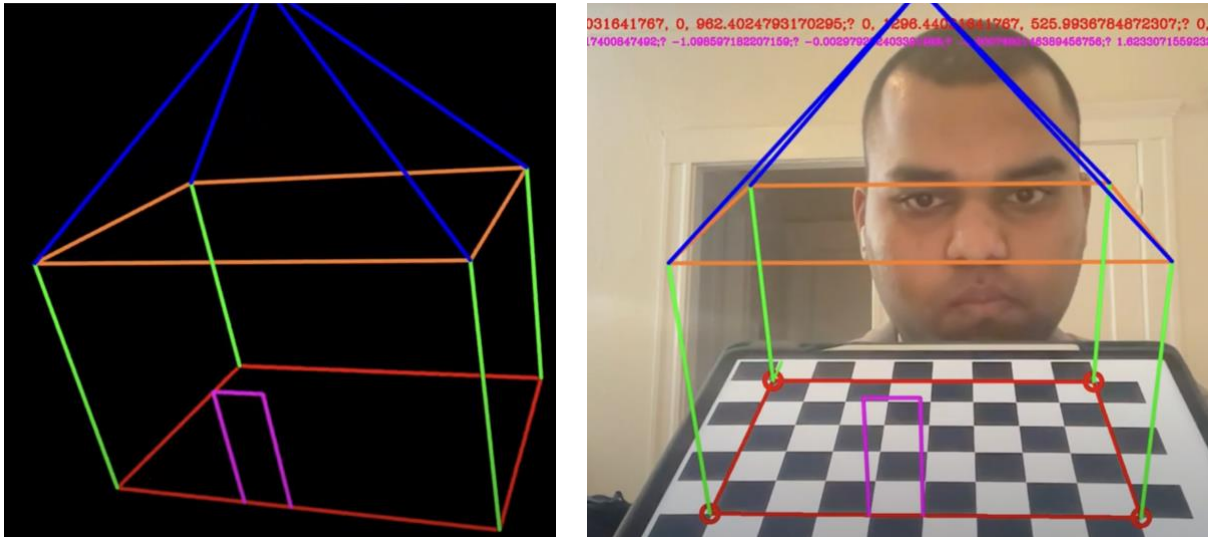


Fig Set Ext 3: *The image demonstrates the working of extension 3.*

On pressing the button 'r' you'll observe that the system turns all the pixels in the frame to black except the virtual object.

Video Link: https://drive.google.com/file/d/1Eho-cn3SE0oc_cz02fhy-aB39SMrVdOD/view?usp=sharing

Extension 4: We performed calibration with two different devices. One from iPhone 13ProMax and the other from Mac book Pro.

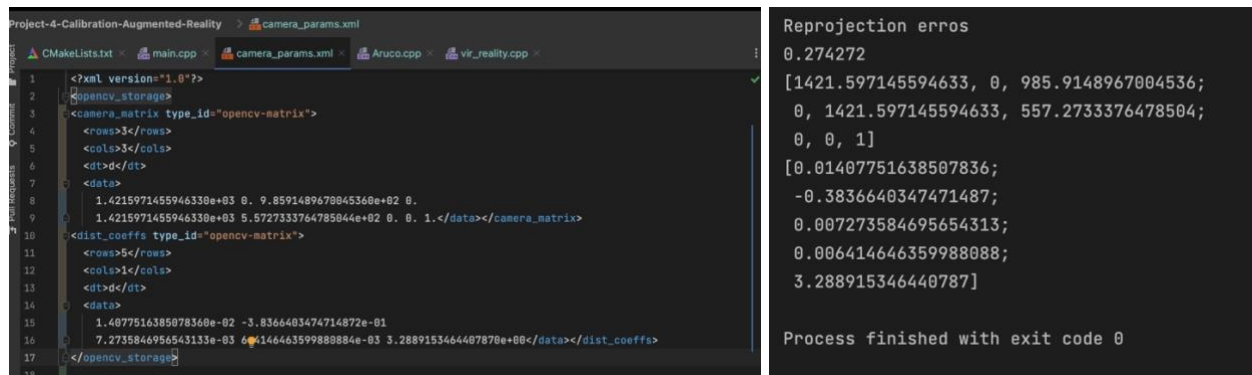
1. Mac Book Camera

Distortion matric and cameras matrix obtained after performing calibration with Mac Book Web Cam. And we obtained a reprojection error of 0.2742.

2. iPhone Camera

Distortion matric and cameras matrix obtained after performing calibration with iPhone Cam. And we obtained a reprojection error of 0.218.

Mac Book Camera



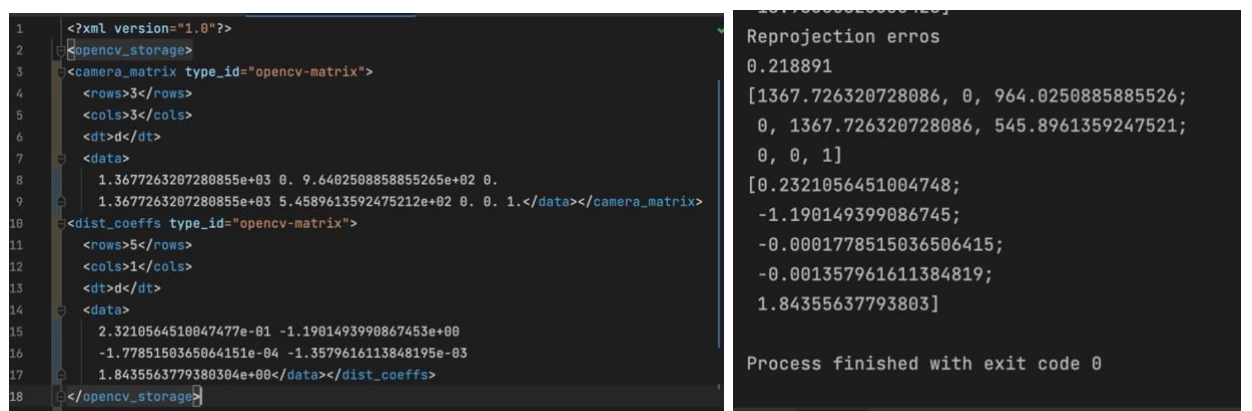
```
<?xml version="1.0"?>
<opencv_storage>
  <camera_matrix type_id="opencv-matrix">
    <rows>3</rows>
    <cols>3</cols>
    <dt>d</dt>
    <data>
      1.4215971455946330e+03 0. 9.8591489678045360e+02 0.
      1.4215971455946330e+03 5.5727333764785844e+02 0. 0. 1.</data></camera_matrix>
  <dist_coeffs type_id="opencv-matrix">
    <rows>5</rows>
    <cols>1</cols>
    <dt>d</dt>
    <data>
      1.4077516385078360e-02 -3.8366403474714872e-01
      7.2735846956543133e-03 6.414646359988884e-03 3.2889153464407870e+00</data></dist_coeffs>
</opencv_storage>
```

```
Reprojection erros
0.274272
[1421.597145594633, 0, 985.9148967004536;
 0, 1421.597145594633, 557.2733376478504;
 0, 0, 1]
[0.01407751638507836;
 -0.3836640347471487;
 0.007273584695654313;
 0.006414646359988088;
 3.288915346440787]
```

Process finished with exit code 0

Fig Set Ext 4.1: Intrinsic Parameters and Error estimate from iPhone Camera

iPhone Camera



```
<?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

Fig Set Ext 4.2: Intrinsic Parameters and Error estimate from iPhone Camera

Extension 5: Used the **OpenCV arucoMarker** as targets and overlaid virtual images on it on a live video sequence.

Video Link:

<https://drive.google.com/file/d/1PYm8uKytTqb3aaOXf6DTIgMAF56eoddn/view?usp=sharing>



Fig Set Ext 5.1: Image we used to overlay on the ArucoMarkers

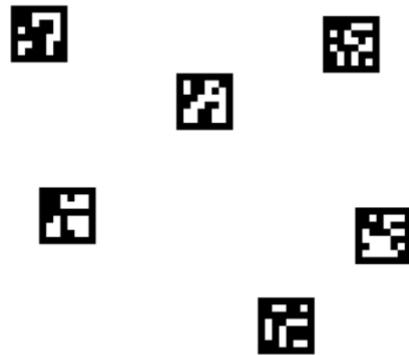
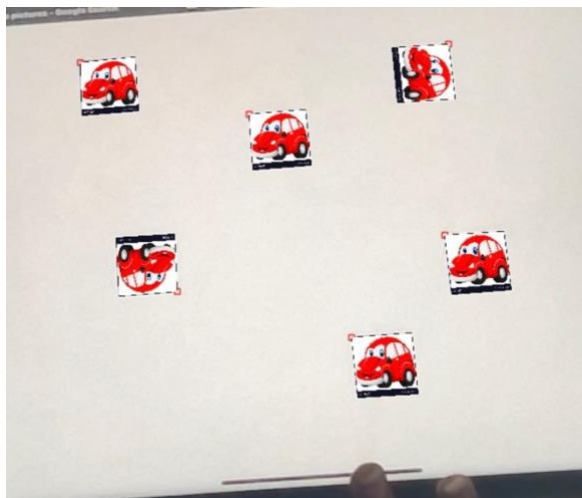


Fig Set Ext 5.2: Above image displays the image being overlaid on the arucoMarkers on a white background.

Reflections:

This project has provided us with a clear understanding of calibrated cameras and their functioning. A chessboard image is commonly employed to calibrate cameras, with OpenCV functions being utilized to identify chessboard features. By extracting the pattern, we were able to project 3D objects onto the board, creating an engaging and immersive experience. We now comprehend the underlying principles of this technology and recognize the potential of computer vision. This experience has also led us to ponder recent concepts such as AR and VR.

Acknowledgements:

Professor Bruce, and his lectures were helpful to understand concepts of calibration, corner detection and augmented reality.

Piazza discussions also helped us better understand certain details necessary for this project

Online Resource References:

1. OpenCV official documentation: <https://opencv.org>
2. https://www.cs.cmu.edu/~16385/s17/Slides/6.2_Harris_Corner_Detector.pdf
3. <https://anothertechs.com/programming/cpp/opencv-corner-harris-cpp/>
4. https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html