Project 4: Calibration and Augmented Reality

By Bennet Brain and Chandler Smith

1. *A short description of the overall project in your own words. (200 words or less)*

At a high level, this project exposes us to the various techniques and building blocks required for engaging with virtual and physical objects in a dynamic, 3D space. It involves learning how to calibrate a camera, analyze physical objects within the video frame, and project virtual objects into the scene.

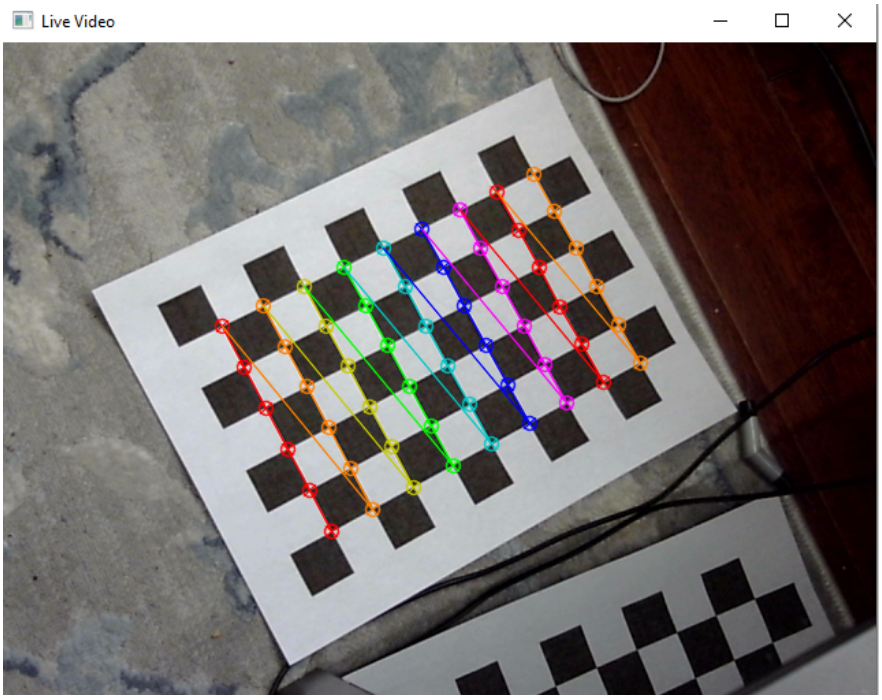
The project uses a checkerboard pattern as the base pattern for detecting and extracting information from the object, which is then used to calibrate the camera. This calibration uses the known position of the object to figure out where the camera is, and thus what the parameters for the transformation from the world space to the image plane are. The end result is a program that can detect a target and then place a virtual object in the scene relative to the target that moves and orients itself correctly, given the motion of the camera or target. Finally, we are briefly exposed to how this model can be used for feature identification, the foundation for augmented reality.

2. *Any required images along with a short description of the meaning of the image.*

Note: We have uploaded a video of the entire system in motion- finding the corners, capturing calibration images, creating the calibration matrix, projecting axes, projecting a 3-d object, displaying the SURF features, and the first extension (hiding the object) at the following link: https://drive.google.com/file/d/1RA1zzGkQ__GC4fhpxtL2Bd2bblIKJBD9/view?usp=sharing

Calibration image with chessboard corners highlighted Chessboard Corners:

Image:



The chessboard corners here are highlighted by the openCV function “drawChessboardCorners”. The first corner here is actually the bottom-left on Bennett’s setup, but on Chandler’s setup the program finds them in the top-right. One simple flip is the only difference between the two though, so the projections work similarly.

Include camera calibration error estimate in your report

Reprojection error estimate:

Bennett (15 calibration images used): Reprojection error of .227 pixels, as shown below:

```
Camera Mat pre-calib:[1, 0, 320;
 0, 1, 240;
 0, 0, 1]
Distortion Coeffs pre-calib:[0, 0, 0, 0, 0]
Camera Mat post-calib:[1045.270703081545, 0, 404.2844216922479;
 0, 1045.270703081545, 223.9898434557082;
 0, 0, 1]
Distortion Coeffs post-calib:[0.008770864724456793, 1.703361939743208, -0.004508033067499486, 0.01471263196615255, -8.52
1877296780209]
Reprojection error:0.227205
Enter your name: bennett
```

Chandler (15 calibration images used): Reprojection error of 1.310 pixels, as shown below:

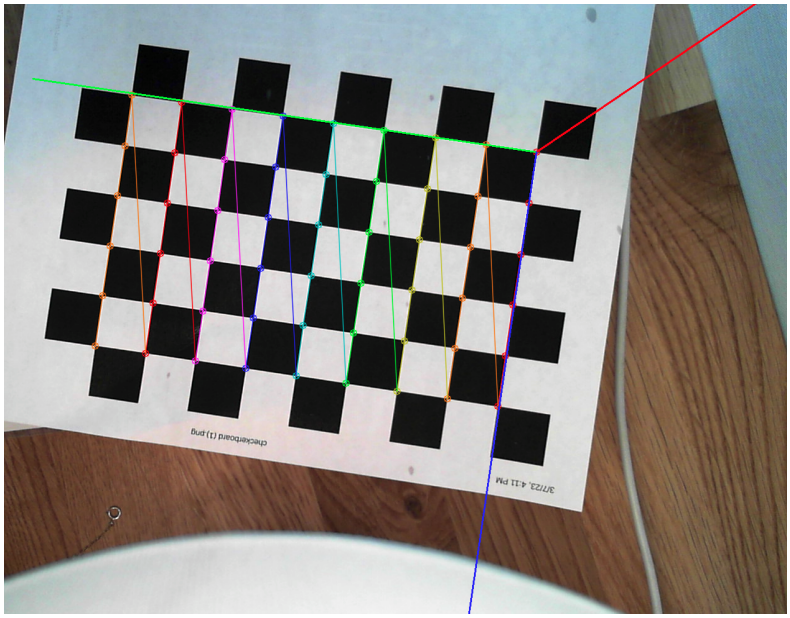
```
Camera Mat post-calib:[1486.171352235093, 0, 891.9347625720326;
 0, 1486.171352235093, 364.7038570161337;
 0, 0, 1]
Distortion Coeffs post-calib:[0.4190438269953864;
 -1.308987546345417;
 -0.08173503041975992;
 -0.04753095969732768;
 2.388783036856319]
Reprojection error:1.3097
Enter your name: Chandler
```

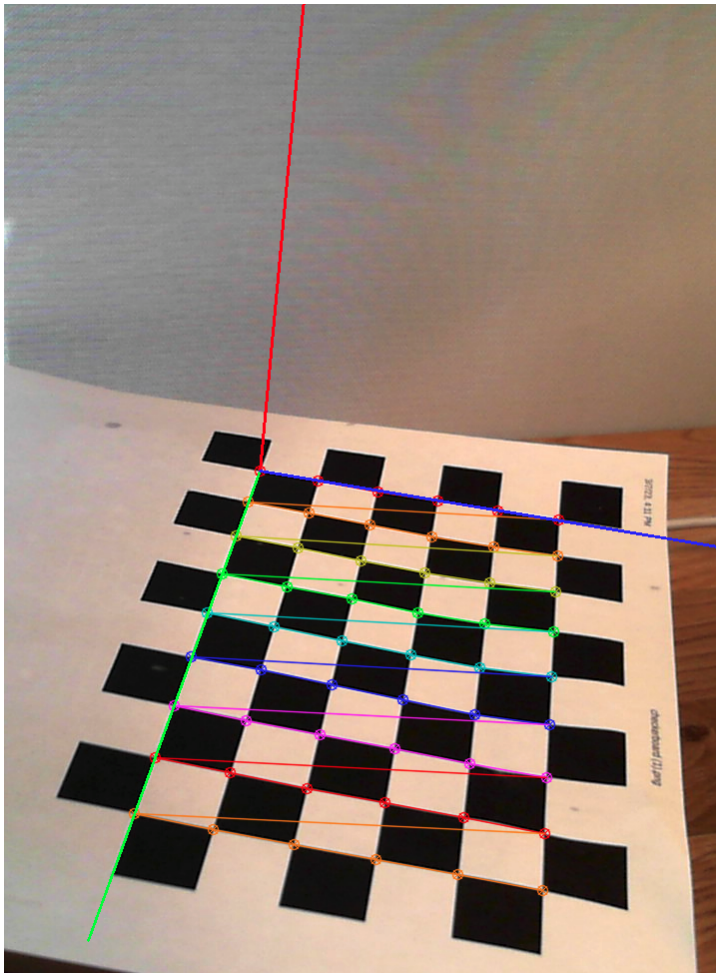
These error estimates were generated by the openCV function “calibrateCamera”, as were the post-calibration distortion and camera matrix entries. That function generates them by solving the transformation matrix from the real world plane to the image plane with the lowest reprojection error. The pre-calibration values were 0 for all the distortion vectors, and the camera matrix’s right-column values were the frame’s width/2, the frame’s height/2, and 1 for each camera.

A reprojection error of over 1 pixel is not ideal for Chandler’s camera, but with a less-powerful camera it’s understandable. Bennett has a stronger camera, which explains the low reprojection error of 0.227 pixels.

Project 3d Axes onto the image:

Image:



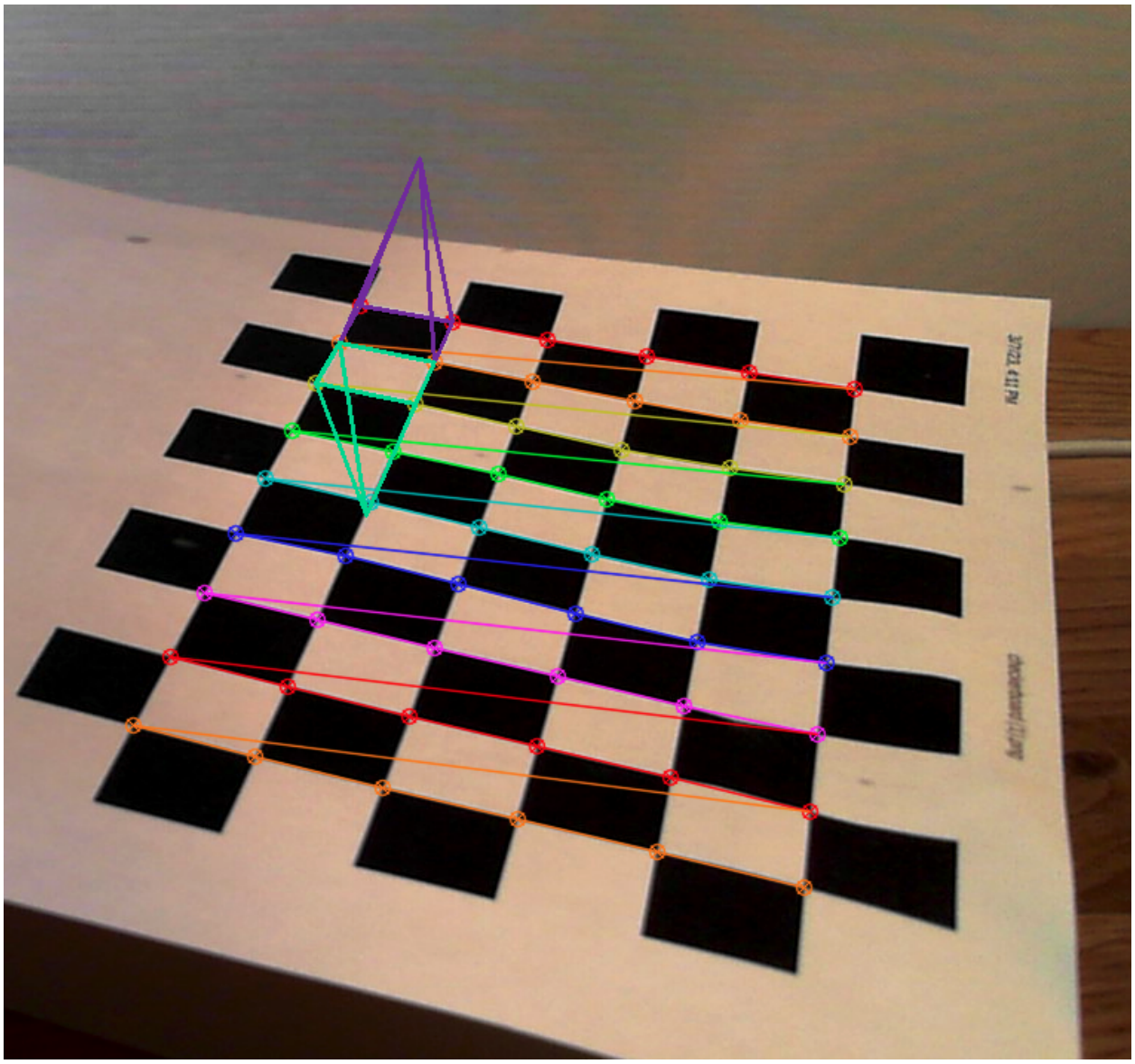


The 3-d axis that forms the baseline for the real-world plane is projected onto the image here. The X axis is the blue line, the Y axis is the green line, and the Z axis is the upwards red line.

They were each created by simply drawing a 10-length line onto the world starting at (0,0,0) and going to each of (10,0,0), (0,10,0), and (0,0,10), and reprojecting those lines onto the image plane from the 3-d plane using the matrices calculated earlier.

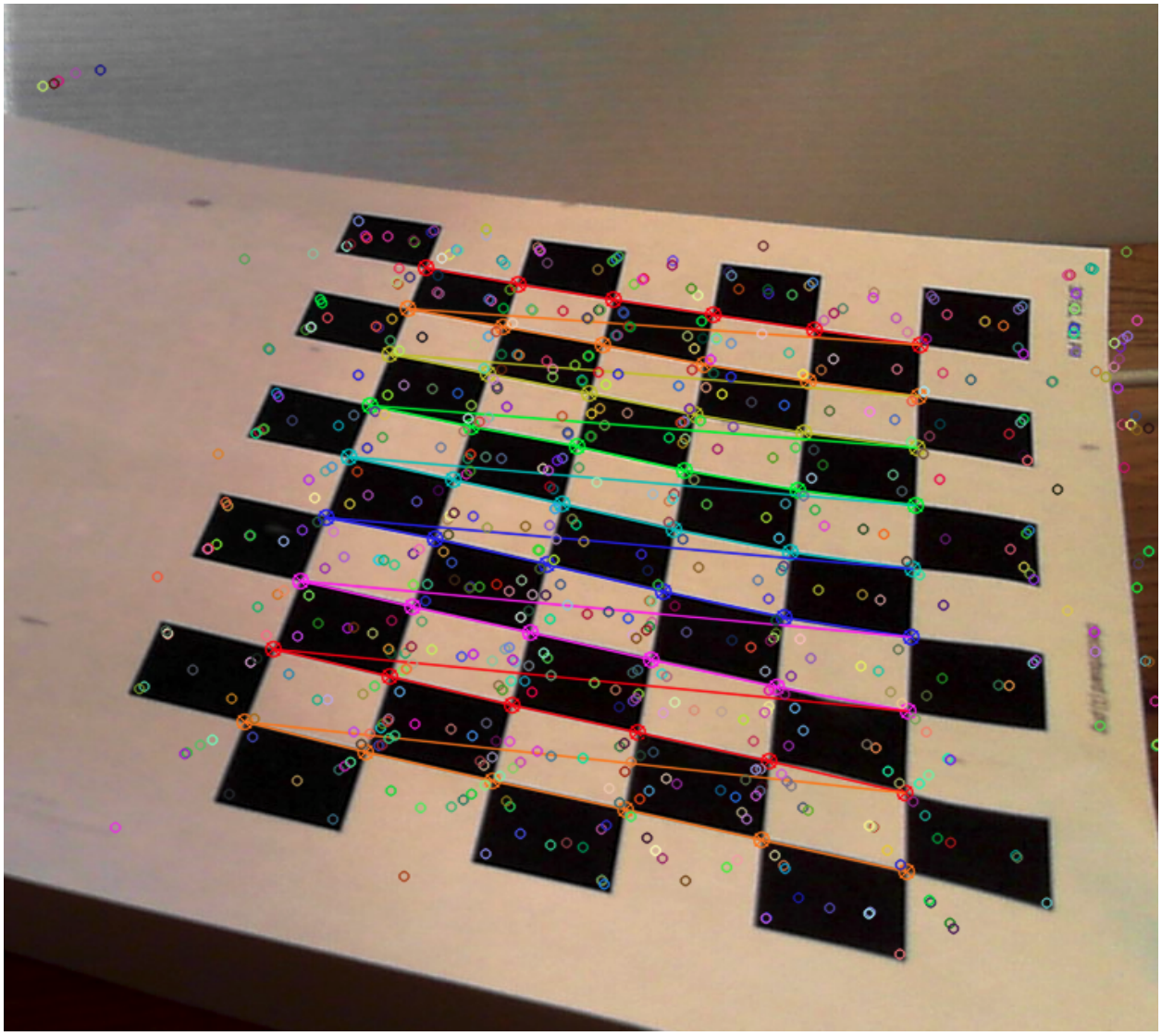
Include virtual object screenshots and/or video:

Image:



The two objects are a prism going up into the z-axis on the first checkerboard square, and another prism going down in the Z axis on its adjacent square. As shown in the video linked at the start of the report, these objects keep their proper position no matter how you move the camera. They were created by identifying the proper 3-d points to define the object, reprojecting those points into the 2-d image plane, and drawing lines between those points.

Include SURF features videos and/or screenshots:



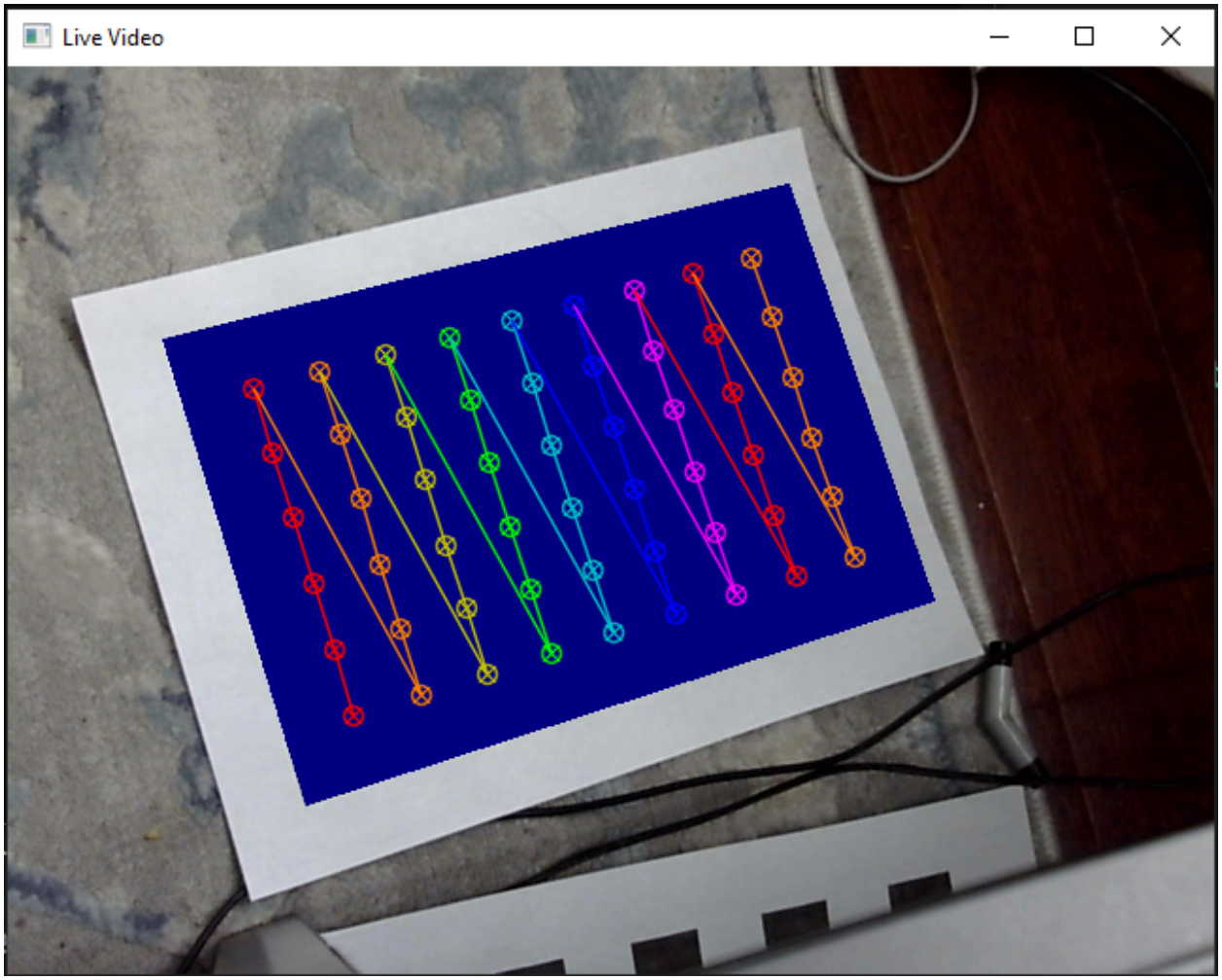
With these SURF features, and by capturing multiple images of the environment, we could use feature matching to identify where the points are in each of a few test images. Then, using the ideas behind stereo depth perception, we could calculate the depth of each of the matched features, giving us real-world positions of those features. Then, using those real world positions of the features and the in-frame positions for the features in current image matched against a stored database of images, we could use solvePNP to get the current rotation and translation of the camera, thus getting the matrix needed to project AR objects in the real world plane into the image plane.

In the current implementation, the surf features realistically do not provide a lot of value for AR. In the future, further honing in the criteria for surf features will be necessary to be able to do more advanced analysis.

Video of Working program:
https://drive.google.com/file/d/1RA1zzGkQ__GC4fhpxtL2Bd2bblIKJBD9/view?usp=sharing

3. A description and example images of any extensions.

Extension 1: Hiding checkerboard object



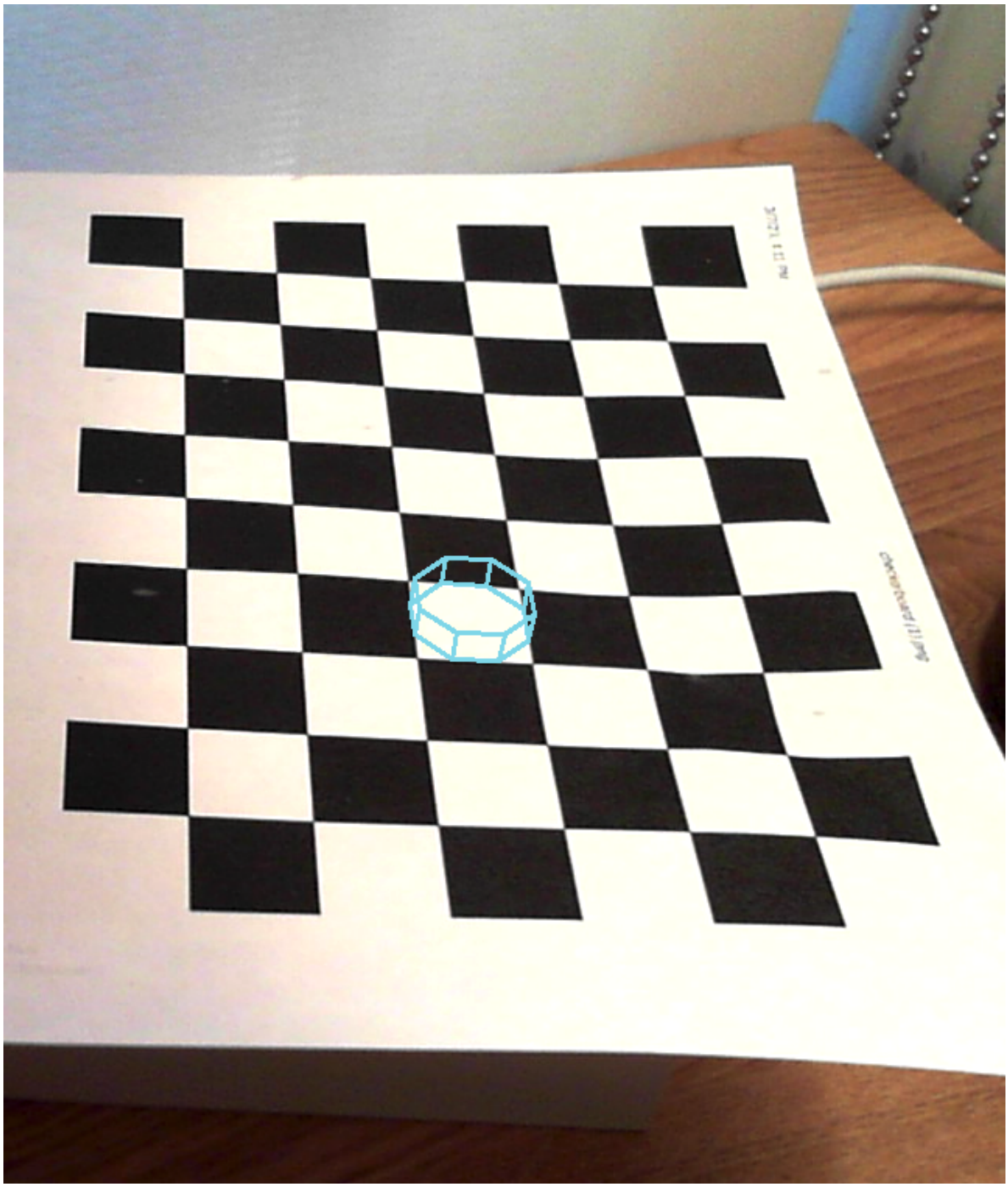
By reprojecting the outside points of the image; (-1,-1,0) (-1, width, 0), (height, -1, 0), and (height, width, 0) into the image plane, we can define the corners of a rectangle which exactly encapsulate the checkerboard. By filling that with a color of our choice using cv::fillPoly, we were able to hide the original target and make it no longer look like the target. We kept the identified corners on top to show where the target used to be.

Extension 2: Multiple Camera Calibration

As shown in the initial camera calibration part of the report, we compared calibrations on Bennett and Chandler's cameras. The stronger camera (Bennett's) had a lower reprojection

error, even though both used 15 calibration images. Additionally, generally higher distortion values can be seen with Chandler's camera, even though a few entries are higher on Bennett's. Regardless, both work decently well for this project. The linked video was taken with Chandler's camera.

Extension 3: Complex Virtual Object with user-defined position



Video link for extension 3:
https://drive.google.com/file/d/1DWe1iSnHYf_xnbaOKI_QbamBScOFtyqQ/view?usp=sharing
For this, we figured out the relative coordinates of a hexagonal checker piece and allowed the user to input where on the board it should be (this was with x = 2, y = 5 as the bottom-left corner relative to the axes, or the top-left corner with the specific camera rotation of this image. Indexing starts at 0 so it's in the 3rd square up and 6th square rightward). The coordinates to build the checker piece are defined by the following lines of code (we used a default height of 0.3):

```
// Bottom Hexagon Points
 imagePoints3D.push_back(cv::Point3d(xstart,ystart + 0.3,0));
 imagePoints3D.push_back(cv::Point3d(xstart,ystart + 0.71,0));
 imagePoints3D.push_back(cv::Point3d(xstart + 0.3, ystart + 1.0, 0));
 imagePoints3D.push_back(cv::Point3d(xstart + 0.71, ystart + 1.0, 0));
 imagePoints3D.push_back(cv::Point3d(xstart +1.0,ystart + 0.71,0));
 imagePoints3D.push_back(cv::Point3d(xstart +1.0,ystart + 0.3,0));
 imagePoints3D.push_back(cv::Point3d(xstart +0.7,ystart + 0.0,0));
 imagePoints3D.push_back(cv::Point3d(xstart +0.3,ystart + 0,0));
// Top Hexagon Points
float height = -0.3;
imagePoints3D.push_back(cv::Point3d(xstart + 0,ystart + 0.3,height));
imagePoints3D.push_back(cv::Point3d(xstart + 0,ystart + 0.71,height));
imagePoints3D.push_back(cv::Point3d(xstart + 0.3,ystart + 1.0,height));
imagePoints3D.push_back(cv::Point3d(xstart + 0.71,ystart + 1.0,height));
imagePoints3D.push_back(cv::Point3d(xstart + 1.0,ystart + 0.71,height));
imagePoints3D.push_back(cv::Point3d(xstart + 1.0,ystart + 0.3,height));
imagePoints3D.push_back(cv::Point3d(xstart + 0.7,ystart + 0.0,height));
imagePoints3D.push_back(cv::Point3d(xstart + 0.3,ystart + 0,height));
```

Then, to actually draw the piece, we projected the 3d points into the 2d image plane and drew appropriate lines between them (between adjacent vertices of the bottom, between adjacent vertices of the top, and between corresponding top/bottom vertices).

On the user-end, hitting "c" will ask for new coordinates each time, allowing you to place a new piece on the board (and deleting the old piece in the process). You could even place pieces off-board if you want to see where they'd end up in the image. The system in action is shown by the video link above.

4. *A short reflection of what you learned*.

Throughout this project, we've gotten a lot of hands-on experience with understanding the necessary components for both camera calibration and AR projection. In essence, enough data about both real-world points to use as references and where they are in multiple calibration images is both necessary and sufficient to project whatever fake object you want into the image. With a checkerboard, the real-world points are easy to understand- they are just the coordinates of the checkerboard in units of checkerboard length, and all are 0 in the Z axis. With more

complicated or even unknown environments, different real-world points will be needed, and the last part of the project showcased briefly how such points can be identified using SURF features.

Also, it was interesting to see the differences between cameras and how that bears out when performing calibration. Using the exact same code and number of calibration images, a pretty wide gulf in reprojection error shows up just because of varying camera quality.

Furthermore, it was fascinating to do the various visual projections. Executing code like the 3D axis really forced us to process and understand the 3D orientation of objects in a frame. It was also amazing to project a variety of virtual objects by manipulating 3D points and, in a very fun sense, we are close to building a game of checkers!

Lastly, this project really showed how quickly these computations can grow in size. Identifying the corners, solving for the current position, and drawing a new object on the image each frame quickly makes the program very laggy. This is fine for a 2-week project to learn about AR, but it really highlights how impressive the optimizations must be for real-time AR systems that function at or above 30 FPS.

5. *Acknowledgement of any materials or people you consulted for the assignment.*

- TA's: Gopal, Mrudula, Ravina, Santosh
- OpenCV Documentation
- Notes from Professor Bruce Maxwell's lectures

Thank you once again for all of your incredible help! Could not have done it without the TAs!