Augmented Reality Chess Using OpenCV and OpenGL

Jay Skrobola
jskrobo@clemson.edu

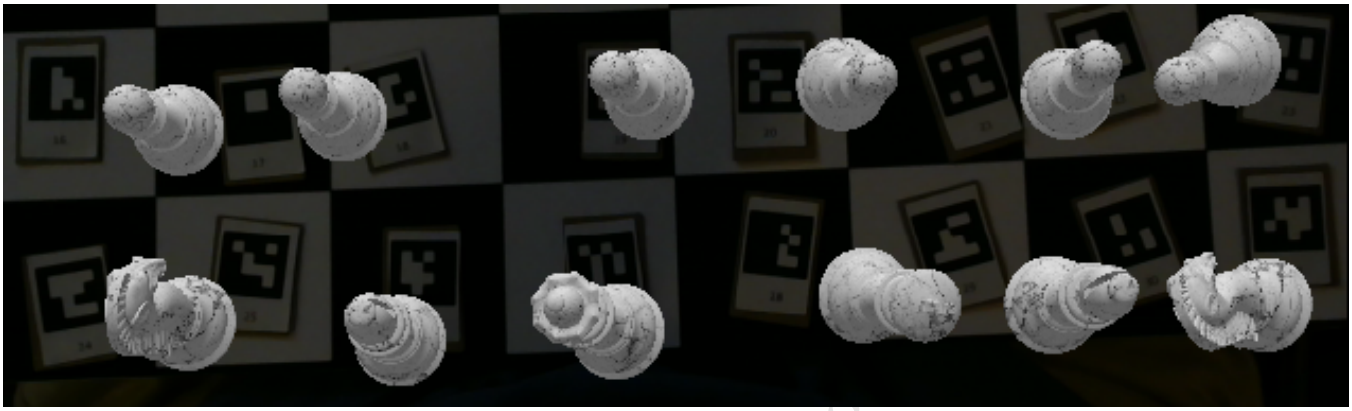

Figure 1: Chess pieces being superimposed onto a real-world chessboard

ABSTRACT

The goal of this project was to provide users with a unique version of chess presented in an augmented reality format. The potential of applications using augmented reality are nearly endless, and this project is meant to as an introduction to what I see to be a beyond exciting field. My lack of experience in OpenGL going into this project added a large amount of complexity to the project, however I feel that, besides learning a new skillset, I have a newfound appreciation for professionals in computer graphics.

ACM Reference Format:
Jay Skrobola. 2021. Augmented Reality Chess Using OpenCV and OpenGL. In *Proceedings of CPSC 4820: Final Project (CPSC 4820).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The entirety of this project was completed using OpenCV and OpenGL, all written in Python. In order to provide a medium for users to move pieces, as well as a way to track each pieces location and pose, 32 individual ArUco markers were utilized. Since ArUco markers provide IDs, it was simple to discern which marker belonged to which piece. OpenCV was used to track each ArUco marker and provide pose estimation. These rotation and transition matrices were then used by the OpenGL module to draw each model. OpenGL includes a steep learning curve, so while I was able to successfully draw the objects for the most part, I believe this portion of the project could be significantly improved with more experience.

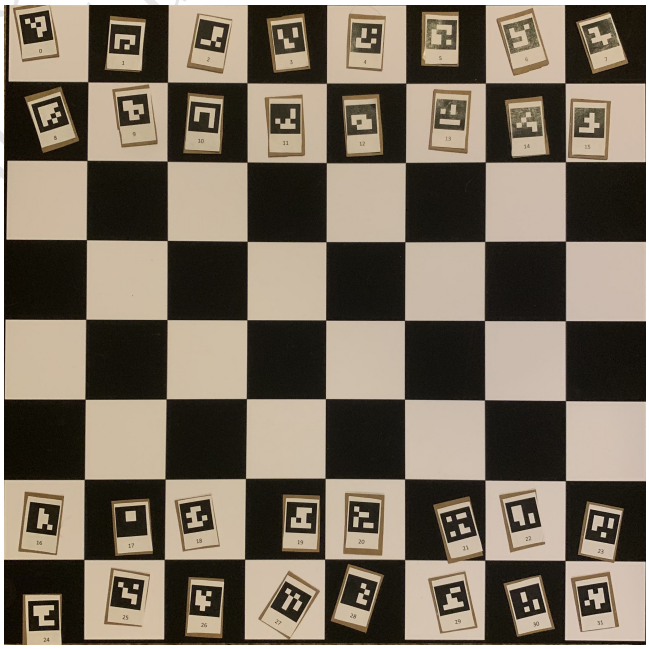

Figure 2: Overview of the chessboard and ArUco markers

2 ARUCO TRACKING

As noted in the introduction, OpenCV was the primary library used for tracking and pose estimation in this project. However, before the ArUco markers could be utilized, the program needed to receive the camera matrix and distortion coefficients in order

CPSC 4820, April 20, 2021, Clemson, SC

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

to account for variations with each user's camera. I chose to solve this by providing a separate program that must be run once before the user ever runs the main program. This program simply looks at the user's empty chessboard and is able to calibrate using the information it gathers (OpenCV provides very useful findChessboardCorners and getOptimalCameraMatrix function). Once the calibration values are created, they are stored locally in a yaml file to be read by the main program from that point forward. The detection of markers is extremely simple using OpenCV, as a single function call returns the location of all markers found in a frame, and a subsequent call provides the rotation and translation vectors needed in OpenGL. OpenCV also provides a simple way to draw frames around each detected marker, which was extremely useful for debugging purposes.

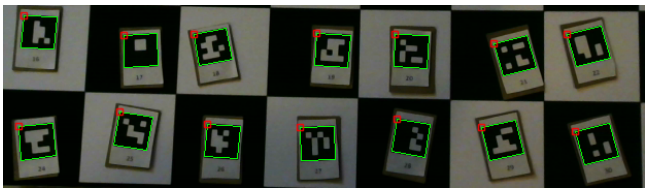

Figure 3: ArUco markers detected using OpenCV

3 LEARNING OPENGL

Coming into this project, I had no experience with OpenGL whatsoever. This provided a pretty extreme hurdle that I needed to overcome with this project. While documentation on using OpenGL in an augmented reality application was sparse, I was able to find some resources that used the library in a context similar to my own. A quick reading of the base OpenGL documentation also gave me a much better idea of how the library is used, and what the important concepts are. OpenGL is far different than any other library I have ever used, so getting used to using it was a challenge, but once I understood the base methodology, the rest was not too bad.

3.1 Pose Estimation in OpenGL

While learning the concepts of OpenGL and configuring all the settings correctly was challenging, the pose estimation portion of OpenGL was actually extremely easy, mostly because I barely had to use it. With the vectors returned from the ArUco markers, I was able to call OpenCV's Rodrigues function to create a matrix. This matrix was converted into a projection matrix that could simply be loaded into OpenGL before each model was drawn. Any other adjustments that I wanted to make could also be achieved with some simple matrix math.

4 3D MODELS

The next step was to implement the 3D models for each of the chess pieces into the program. While OpenGL provides a few 3D models, unfortunately none of them look like chess pieces, so I had to figure out how to bring my own models into my application. I was fortunate to find a script included in the PyGame library that was able to translate an Obj file from blender into a series of OpenGL calls necessary to draw each model. Some slight changes to this script made it extremely easy to import any 3D model into my program, and I was lucky enough to find plenty of free models for all of the chess pieces. The calls created by this object loading could be organized into OpenGL call lists so that each object could be drawn on command without having to repeat the entirety of the calls. Textures for these objects were also easy to implement. Using a single function call, either a white or black texture could be loaded into OpenGL to specify what the color of the next object should be. Both the color and piece type of each marker is determined by the marker's ID, and are arranged in a specific manner.

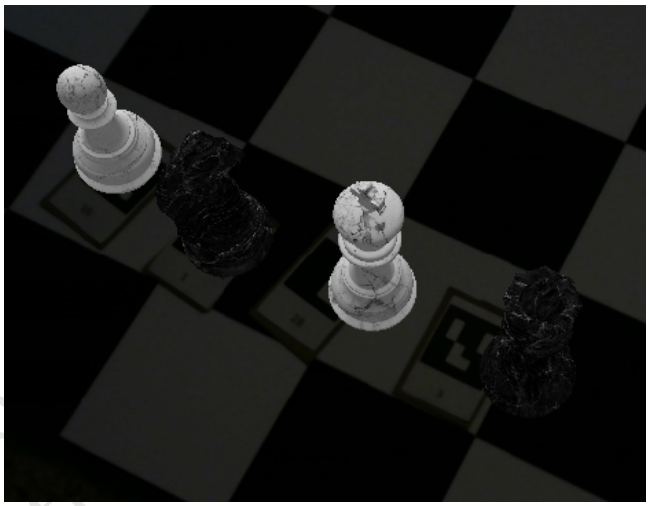

Figure 4: 3D Chess Models being drawn in real-time

5 FUTURE POSSIBILITIES

Unfortunately there were some features that I wanted to include in this application that I was not able to get to work correctly, so they were not included in the final program. I had previously wished to have the application track the location of each piece in the context of the game. In other words, I would have like the application to know when the game was in check or checkmate. I attempted to achieve this by using OpenCV's chessboard function and looping through each individual square to look for a marker. However, I could not get the function to work correctly on the entirety of the board, nor would it work if there were markers on the board. My next attempt was to use point projection to try and check each square in relation to a piece and where it could possibly go. Using this method, I was able to draw lines on the image showing each piece's range of motion, however this feature could not account for the rotation of the piece, the location other pieces, nor for the size of the board, so I chose to comment it out. I believe all of these features are feasible with enough time, but my lack of experience using these made it very difficult.

6 OTHER DOWNFALLS

The result of drawing the models left a lot to be desired from what I wanted out of my final product. For an unknown reason, the models like to shake around. They also like to stray from their ArUco

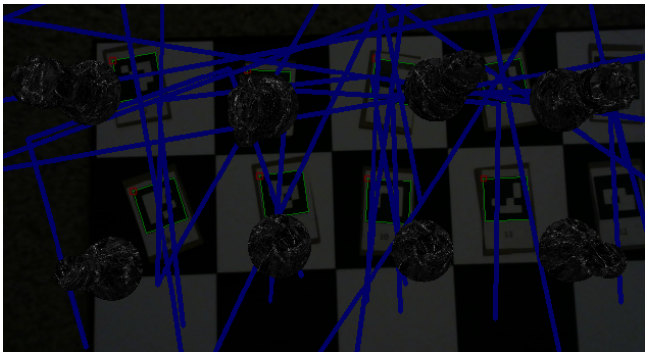

Figure 5: Jumbled Mess that is the range-of-motion feature

markers depending on where they are in the frame. I tried many methods to solve these issues, from re-calibrating the camera, to double checking the projection matrix process, but nothing seemed to resolve them. Once again I feel this is more than likely just a result of my lack of experience with OpenGL, but even searching on the internet for similar problem produced little to no results. Hopefully this is something I can fix in the future.

7 CONCLUSION

This program gave me a tremendous insight into artificial reality and computer graphics, two topics that I had great interest in, but very little experience with leading up to this application. I have a newfound respect for those who produce these applications on the consumer scale, as it can be very difficult to interface the real and virtual worlds through a camera. Overall, I thoroughly enjoyed working on this project, and I plan to continue working to become more familiar with OpenCV and OpenGL, so that hopefully I will be able to come back and complete this project to the point that I had initially envisioned.