Augmented Reality with Aruco Marker Tracking in OpenCV and OpenGL

BENJAMIN BUCK, Clemson University, USA

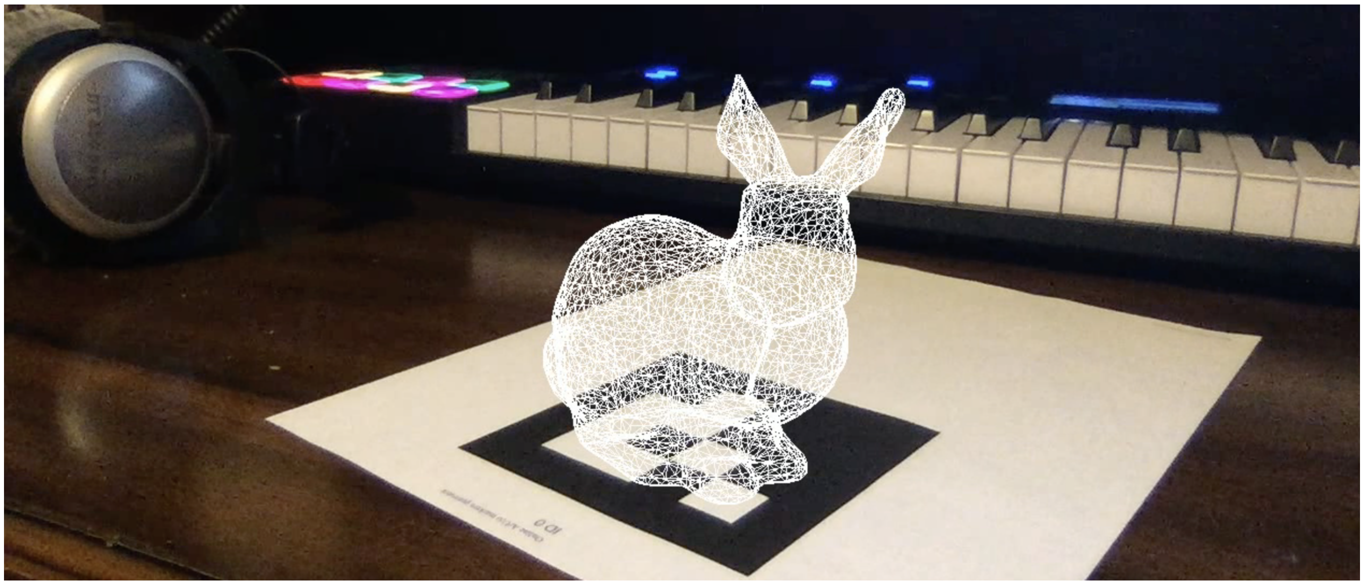

Fig. 1. The Stanford bunny tracked to an Aruco marker using OpenCV and OpenGL.

1 INTRODUCTION

For my project, I aim to use OpenCV and OpenGL to create an augmented reality application. The main goal is to provide the math that allows transformation of the camera intrisics and extrinsics as provided by OpenCV to the View and Projection matrices used for rendering in OpenGL. As a demonstration that the math is correct, I have tracked the Stanford bunny, as loaded from a .obj file, onto an Aruco marker. In this project, I will be using the OpenGL Compatibility profile for simplicity.

2 OPENCV CAMERA CALIBRATION

Firstly, I needed to calibrate my camera in OpenCV. I used a chessboard pattern and captured multiple images of the chessboard in different positions and orientations on the screen. I was careful to include pictures with the chessboard in each corner of the frame, as I found this improved the accuracy of the distortion coefficients near the edges of the screen. OpenCV camera calibration returns a camera intrisic matrix in the form and a list of distortion coefficients, in the following forms.

$$\text{camera intrsic matrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

$$\text{distortion coefficients} = \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix} \tag{2}$$

Camera calibration was performed beforehand in a separate program. These matrices were saved in a numpy .npz file to be loaded into the AR program later.

3 OPENCV ARUCO MARKER DETECTION

In the AR program, I used OpenCV to detect Aruco markers in the frame and to calculate the camera extrinsic properties with respect to the markers. OpenCV outputs the camera extrinsic properties as a translation vector and a rotation vector.

$$\text{tvec} = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} \tag{3}$$

$$\text{rvec} = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} \tag{4}$$

4 OPENGL COORDINATE SPACES

OpenGL uses a series of matrix multiplications to transform vertex locations through a number of coordinate spaces before being rendered to the screen. These matrices are known as the Model, View, and Projection matrices. The Model matrix is used to transform vertices from local coordinate space to global coordinate space. An

Author's address: Benjamin Buck, bbuck@clemson.edu, Clemson University, Clemson, South Carolina, USA, 29631.

XXXX-XXXX/2022/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

example of this type of transform is importing a model into your scene and translating, rotating, or scaling it to be at the correct location in your scene. The View matrix is used to transform the vertices from global coordinate space to camera coordinate space. This orients the scene around the camera, using translation and rotations. The Projection matrix transforms from camera space to screen space. This application uses a perspective projection that imitates the original camera based on the calibration coefficients. Screen space is clipped to a cube from $(-1,-1,-1)$ to $(1,1,1)$ before rendering. Note that the Model and View matrices are often combined into one ModelView matrix in OpenGL.

5 LOADING OBJECTS

I was able to easily load models from .obj files using the pywavefront library. I also calculated the center of the model's vertices and the overall scale of the model to help compute the translation and scaling appropriate for the Model matrix. In this demonstration, I used the classic Stanford bunny model. I translated it so that the center of the bunny's base was located at $(0,0,0)$ in world coordinate space, and scaled it so that it appears to fit on the Aruco Marker.

6 CALCULATING THE PROJECTION MATRIX

The most difficult bit of math to figure out was how to calculate the Projection matrix to imitate the original camera's projection. I found some equations describing how to leverage OpenGL's glFrustum method to reproduce the projection matrix from the OpenCV parameters. [1] However, these equations contain some order of operations errors which I had to correct. The following equations define how I used the OpenCV camera instrinsic parameters to calculate the top, bottom, left, and right bounds for the frustum to mimic the original camera.

$$\text{left} = near \cdot \frac{-c_x}{f_x} \tag{5}$$

$$\text{right} = near \cdot \frac{w - c_x}{f_x} \tag{6}$$

$$\text{top} = near \cdot \frac{-c_y}{f_y} \tag{7}$$

$$\text{bottom} = near \cdot \frac{h - c_y}{f_y} \tag{8}$$

Note that w and h here are the width and height of the camera frame in pixels. For the frustum, near and far clipping planes must be defined. These may vary based on your scene, but I got good results with $near = 0.001$ and $far = 1$. Using these values in glFrustum produces a fairly accurate reproduction of the camera's projection.

7 CALCULATING THE VIEW MATRIX

The view matrix can be easily calculated from the camera extrinsics, as provided in rvec and tvec by OpenCV. The first step is to load a $4x4$ identity matrix into for the ModelView matrix. Then, to convert between the coordinate space, you must negate the z component of both rvec and tvec.

$$\text{tvec}' = \begin{bmatrix} t_x \\ t_y \\ -t_z \end{bmatrix} \tag{9}$$

$$\text{rvec}' = \begin{bmatrix} r_x \\ r_y \\ -r_z \end{bmatrix} \tag{10}$$

Next, glTranslate can be used to directly apply the translation of tvec to the view. Note that these operations are not commutative, and the translation must be done first. Then, cv2.Rodrigues is used to convert the rvec into a Rodrigues rotation matrix, which then needs to be extended into a $4x4$ matrix, as follows.

$$\begin{bmatrix} R_{00} & R_{01} & R_{02} & 0 \\ R_{10} & R_{11} & R_{12} & 0 \\ R_{20} & R_{21} & R_{22} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{11}$$

This extended rotation matrix can then be multiplied into the existing ModelView matrix using glMultMatrix.

8 MODEL TRANSFORMATIONS

Model transformations can be made to the ModelView matrix after the View matrix has been created using glTranslate, glRotate, and glScale. Especially in the case where multiple objects must be rendered to the screen, you can use glMatrixPush and glMatrixPop to keep these transformations separate, and you can "undo" them.

9 RENDERING OVER THE CAMERA FRAME

The easiest way to render the camera frame to the screen with OpenGL is to temporarily set the Projection and ModelView matrices to the identity, and then render a plane with 2D coordinates $(-1,-1), (1,-1), (1,1), (-1,1)$. These points must be given their respective texture coordinates $(0,0), (1,0), (1,1), (0,1)$. Flipping the frame image vertically and binding it to a texture on this plane will render it correctly behind all of the 3D objects in the scene.

10 FURTHER WORK

There are many good ways to further extend this work into a more fleshed out application. Firstly, this application does not take the distortion coefficients into account. This did not pose a problem for me, but two solutions would be to undistort the camera frame in OpenCV before rendering or actually implementing distortions on the 3D models with a more complex vertex shader, using the OpenGL Core profile. Another avenue of further work would be to add animated models which could be tracked to the Aruco markers.

REFERENCES

[1] Kyle Simek. 2013. Calibrated cameras in opengl without glFrustum. http://ksimek.github.io/2013/06/03/calibrated_cameras_in_opengl/