INDIAN INSTITUTE OF TECHNOLOGY, DELHI

REPORT

Assignment 3: Camera Calibration and Augmented Reality

ARNAV TULI | ENTRY NO. 2019CS10424

Course - COL780 | Prof. Anurag Mittal

Due April 7, 2023

Contents

Chapter 1

Camera Calibration and Augmented Reality

1.1 Introduction

Camera calibration involves finding the projection matrix P (3×4) that takes a homogeneous coordinate in the world space (X_{world}) and outputs a homogeneous coordinate in the screen/image space (x). In short, for all $X_{world} \in \mathbb{P}^3$:

$$x = PX_{world}$$

where $x \in \mathbb{P}^2$. For finite cameras, the projection matrix P can be decomposed into product of a 3×3 upper-triangular matrix and a 3×4 matrix, such that:

$$P = K[R|t]$$

where K is the matrix comprising of camera's intrinsic parameters and R and t are the camera's rotational (3×3) and translational (3×1) parameters w.r.t to the world coordinate system:

$$\tilde{X}_{camera} = R\tilde{X}_{world} + t$$

where $\tilde{X} \in \mathbb{R}^3$ denotes the actual point represented by the homogeneous coordinate $X \in \mathbb{P}^3$. Thus, the matrix $[R|t]$ comprises of camera's extrinsic parameters and change from one view-point to another. Hence, for AR applications, we will have to systematically update these extrinsic parameters as the camera's position in the world changes.

To find P, I will find matrices K and $[R|t]$. Since, K does not change from one view-point to another, I will first find K and store it. I will use this K later to find $[R|t]$ for object augmentation in the scene. The camera calibration algorithm is described below:

1. Find pairs of vanishing points (v_1 and v_2) in image space corresponding to orthogonal ideal points in the world space ($\tilde{d}_1$ and $\tilde{d}_2$). Using these, solve the following equation using DLT to get $W = K^{-T}K^{-1}$:

   $$v_2{}^T W v_1 = 0$$

   Since, K has 5 degrees of freedom, so does W. Hence, minimum 5 pairs of such vanishing points are needed. To reduce noise in the estimate, I have used 20 such pairs of vanishing points. The details of acquiring these vanishing points is mentioned in *section 1.2*.
2. Do *Cholesky* decomposition of W to get K^{-T}. Then, take inverse transpose of it to get K.

3. In the scene (in which the object is to be augmented), using K, mark known set of world points (checkerboard) in the image, and solve the following equation using DLT: (x: image point, X_{world}: known world point on the checkerboard)

$$K^{-1}x = [R|t]X_{world}$$

Since, all world points lie on a plane (checkerboard, $Z = 0$), the equation reduces to:

$$K^{-1}x = [r_1|r_2|t] \begin{bmatrix} (X_{world})_x \\ (X_{world})_y \\ 1 \end{bmatrix}$$

where r_1 and r_2 are the first two columns of the rotation matrix R. Since, $[R|t]$ has 6 degrees of freedom, a minimum of 6 such markings are needed to solve the equation. To reduce noise in the estimate, I have used 25 such markings of known world points in the image. The calibration markings and images are given below for reference:

(a) (b) (c) (d)

Figure 1.1: Images for augmenting object (original)

(a) (b) (c) (d)

Figure 1.2: Images for augmenting object (points marked for calibration)

The matrix $[r_1|r_2|t]$ is normalized so that r_1 and r_2 vectors have unit norm. Since, R is a rotation matrix, r_3 is found by taking the cross product of r_1 and r_2. The sign of the cross-product is determined by the fact that $det(R) = -1$, as we are transitioning from a right-handed coordinate system (world space) to a left-handed coordinate system (camera space). Thus, for each scene, I am able to find the matrix $[R|t]$. Finally, multiplying K with $[R|t]$ gives the projection matrix, P.

1.2 Finding orthogonal vanishing points

To find 20 pairs of orthogonal vanishing points, I took 20 pictures of checkerboard from different distances and angles. In each picture, I marked 12 points: 6 points corresponding to the horizontal line in real world (or row in checkerboard), and remaining 6 points corresponding to the vertical line in real world (or column in checkerboard). In each line, the marked points were equidistant in the real world, with distance equal to the side length of a square on the checkerboard = 3.175 (cm). Then, using these 6 points on a line, I found the 2×2 homography matrix from world points on the line to image points on the the line (using DLT):

$$\begin{bmatrix} (x_i)_x \\ 1 \end{bmatrix} = H \begin{bmatrix} 3.175 * i \\ 1 \end{bmatrix}$$

where $0 \leq i \leq 5$. Since H is a 2×2 homography matrix, it only has 3 degrees of freedom. Hence, 6 points are more than sufficient to determine precise value of H.
Once H is determined, I plug the ideal point in the homography equation to get the vanishing point in the image:

$$\begin{bmatrix} (x)_x \\ (x)_h \end{bmatrix} = H \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This implies that the vanishing point is at a distance of $\frac{(x)_x}{(x)_h}$ from the origin point in image. Since, both the origin point (marked in image) and the equation of line in image (can be determined from any two marked points in the image) are known, the vanishing point can be uniquely determined. Some reference calibration images are given below:

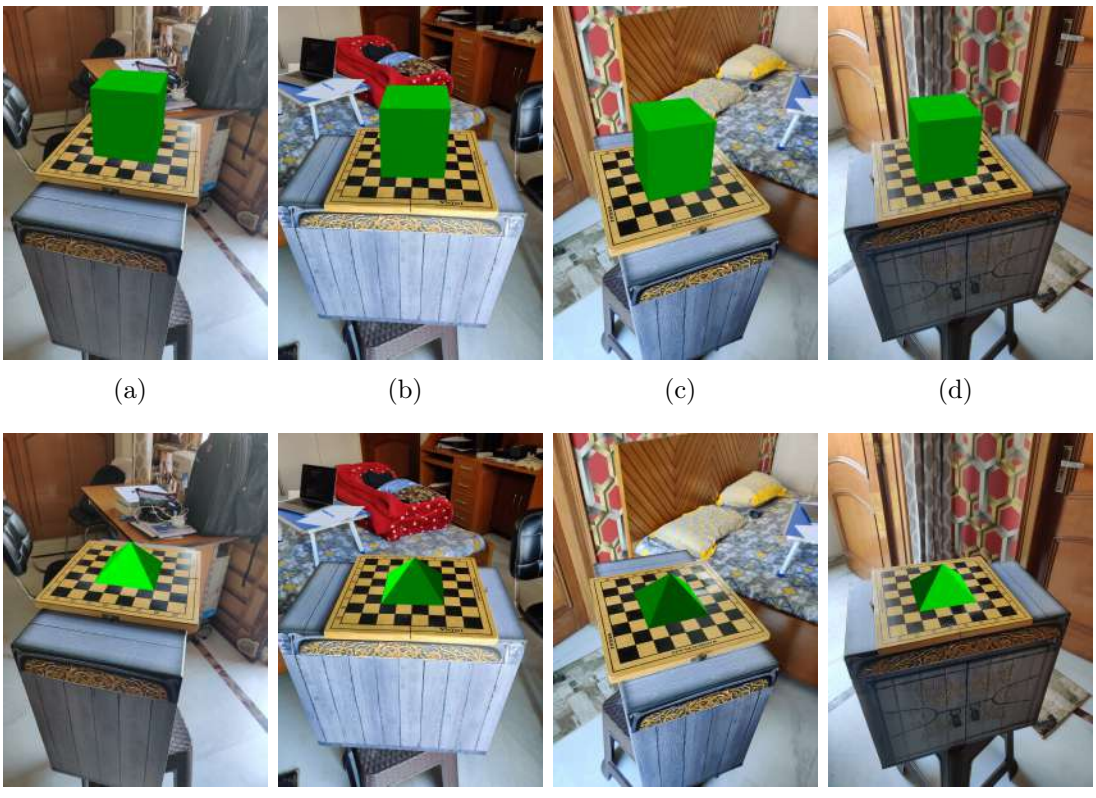
(a) (b) (c) (d) (e)

Figure 1.3: Images for finding K (points marked along row and colum)

1.3 Object augmentation

To augment an object into the world scene (image), I have generated meshes (list of vertices, triangles and normals) of two simple types: *cube* and *pyramid*. The mesh is generated in the world space, and needs to be transformed via camera's projection matrix, P, to get its position in the image. Once in the image space, the object is rasterized onto the image buffer to complete the augmentation. I have also added light sources in the 3D world to get a shading effect on the added object, and make it look realistic. I have implemented two different methods to rasterize the object onto the image:

- The general rasterization pipeline as implemented in the GPU (with depth buffer). This allows correct rasterization of all types of objects, even the complicated ones. However, this is implemented on the CPU, and hence, is extremely slow (even for simple objects).
- Visibility-based rasterization. This is a heuristic-based rasterization technique that works for simple objects (like cube and pyramid). In this method, I determine which faces of the mesh are visible from the camera view-point given the camera's position and face normals, and only rasterize those faces that are visible. For simple meshes, these faces will be non-overlapping and can be rasterized in any order to produce a correct image. OpenCV's *fillPoly()* method is used to perform rasterization on GPU. Consequently, it is much faster than the first method, and is used to rasterize simple meshes.



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 1.4: Mixed/AR pictures corresponding to *figure 1.1*: (a)-(d) Cube; (e)-(h) Pyramid

Chapter 2

Appendix: Code Overview

The link to the dataset used for calibration, learnt camera models and mixed/AR pictures is here. There are five code files `main.py`, `globals.py`, `mesh.py`, `rasterizer.py` and `calibration.py`. The code assumes the following directory structure:

```
directory
├── Dataset/
│   ├── Chessboard/
│   ├── TableTop/
│   ├── Output/
│   └── Calibration/
│       ├── Chessboard/
│       └── TableTop/
├── CameraModels/
└── main.py globals.py mesh.py rasterizer.py calibration.py
```

`main.py` has support for the following command line arguments:

- `--load_K`: Load intrinsic parameter matrix flag
- `--load_RT`: Load extrinsic parameter matrix flag
- `--augment`: Object augmentation flag
- `--pyramid`: Pyramid object flag
- `--color`: Object color name ("red", "green" or "blue")
- `--heuristic`: Visibility-based rasterization flag

Run command: `python main.py` ⟨ `args` ⟩
Output: Final mixed/AR images are saved in the Dataset/Output/ `directory`.