Camera Calibration using OpenCV

Thomas W. C. Carlson

2-3-2021

Abstract

Before stereovision can be used to reliably extract information about the world from the camera image planes, the behavior of the camera must first be known. The process of understanding how a world point projects onto a camera image plane is called calibration. The following is a description of how a pinhole camera can be modeled and calibrated.

1 The Pinhole Camera Model

The pinhole camera model is a simple model containing an image capture plane, an aperture or focal point, and a virtual image plane. It captures reflected light which travels through the aperture on its image plane. As a result of the aperture being a hole and light traveling in vectors, the image plane image is actually inverted. The virtual image plane is an image of the same size as the image plane, but with the same orientation of points as the world.

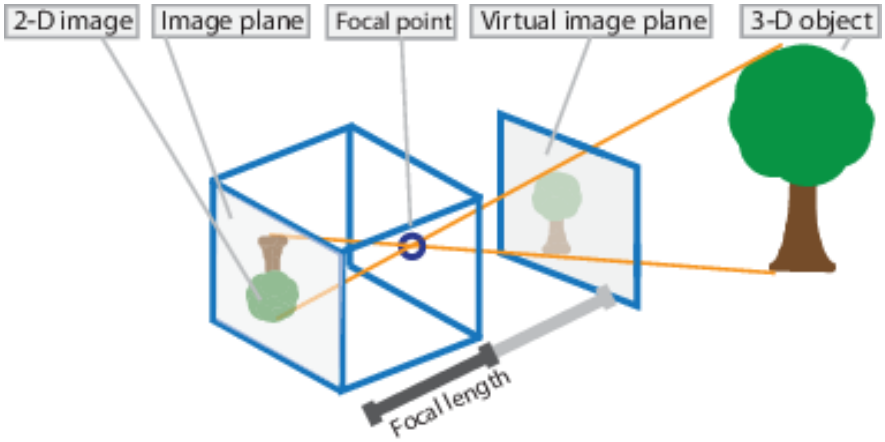

Figure 1: Pinhole Camera Model.

Put another way, the aperture can be modeled as the origin of a 3D graph, through which all lines of interest pass. The lines of interest begin at an object in the world and intersect the image plane located a distance equal to the focal length away from the origin.

With this representation a relationship between world and image plane coordinates can be established.

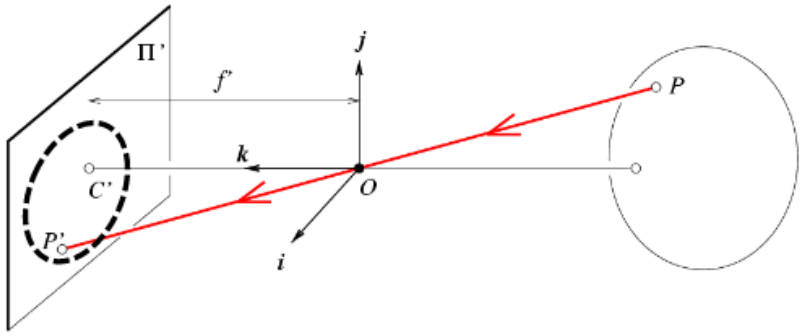

Figure 2: Pinhole Camera Model.

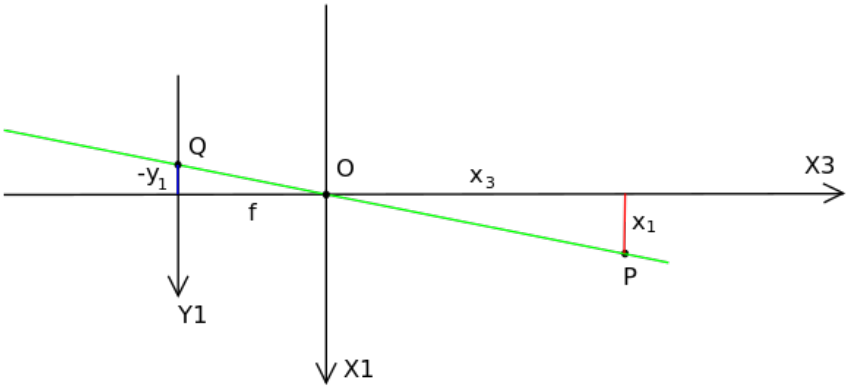

Figure 3: Pinhole Camera model as seen from above.

$$\frac{-y_1}{f} = \frac{x_1}{x_3} \rightarrow y_1 = \frac{-fx_1}{x_3} \tag{1}$$

$$\frac{-y_2}{f} = \frac{x_2}{x_3} \rightarrow y_2 = \frac{-fx_2}{x_3} \tag{2}$$

Which can be expressed as a matrix transformation

$$P' = \begin{bmatrix} P'_{x1} \\ P'_{x2} \end{bmatrix} = \frac{-f}{x_3} \begin{bmatrix} P_{x1} \\ P_{x2} \end{bmatrix} \tag{3}$$

Using homogeneous coordinates, we find the matrix for the intrinsic parameters of the camera. This contains the focal length, aspect ratio, and its optical center position. For simplicity, we assume the aspect ratio to be 1, and the optical center of the camera to be at (0,0,0). We call this matrix the intrinsic camera matrix, and its values the intrinsic parameters of the camera.

$$P' = \begin{bmatrix} fx \\ fy \\ z \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = K \begin{bmatrix} I & 0 \end{bmatrix} P \tag{4}$$

To include the effect of skew s, a non-origin optical center located at (u_0, v_0), non-square pixels (yielding different focal lengths in x and y), the intrinsic matrix would become

$$K = \begin{bmatrix} f_x & s & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

Extrinsic parameters of a camera include its rotation R and translation t in real world coordinates by postmultiplying the intrinsic matrix K by the sequence of transformations

$$P' = K \begin{bmatrix} R & \bar{t} \end{bmatrix} P = \begin{bmatrix} f_x & s & u_0 \\ 0 & f_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_x \\ r_{31} & r_{32} & r_{33} & t_x \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6}$$

So, to find the location of the projection of a point (x, y, z) on the image plane of the camera, it is necessary to define both the intrinsic and extrinsic parameters. This is the job of calibration.

2 Calibration

In essence, calibration is the process of taking a number of photos (data) and determining what parameters can give rise to the specific set of images. This is most easily done with a known pattern which can be easily found by the evaluating algorithm. The high-contrast squares of a chessboard offer ideal conditions for this procedure, and as such are very commonly used.

OpenCV's libraries contain functions which make this very easy to do. The procedure will involve

1. Capturing images of a chessboard with specific dimensions
2. Algorithmically locating the interior corners of the chessboard in each image
3. Refining the corner subpixel locations to be more precise
4. Storing the corner locations in memory
5. Calibrating the camera using the stored corner locations

Two scripts are provided. `CogCalib.py` uses images captured using two Cognex cameras with much lower resolution than the Pentax K3 ii. As such it runs much faster than the `Calib.py` script. Removing images from the imageset can reduce runtime, but the majority of the runtime is spent in the calibration step. It is suggested to, if edits are to be made, edit `CogCalib.py` to understand the effects of the change before using `Calib.py`.

2.1 Stepping Through the Calibration Code

To locate the corners of a chessboard in an image, we pass each image into the `cv.findChessboardCornersSB()` function. Per the OpenCV documentation, this function accepts an image, the chessboard pattern size, and two optional parameters. The first is an option for flags to alter the performance of the algorithm, and the second allows for the attachment of metadata or the importing of corners which have already been found. Neither will be used.

First, define the dimensions of the chessboard. Remember that the algorithm looks only for the interior corners when defining this value. Then initialize the object point lists to contain the location of each chessboard point in a grid (i.e. (0,0,0) (1,0,0) (2,0,0) ... (5,7,0))

```
# python -m pip install opencv-python
import cv2 as cv
# python -m pip install numpy==1.19.3
import numpy as np
import glob
# python -m pip install tqdm
from tqdm import tqdm

# Calibration
chessboard_size = (4,7)

# Array definitions
# Left image arrays
obj_pointsL = []        # Real world space 3D points
img_pointsL = []        # Image plane 3D points
objpL = np.zeros((np.prod(chessboard_size),3),dtype=np.float32)
objpL[:,:2] = np.mgrid[0:chessboard_size[0],0:chessboard_size[1]].T.reshape(-1,2)

# Right image arrays
obj_pointsR = []        # Real world space 3D points
img_pointsR = []        # Image plane 3D points
objpR = np.zeros((np.prod(chessboard_size),3),dtype=np.float32)
objpR[:,:2] = np.mgrid[0:chessboard_size[0],0:chessboard_size[1]].T.reshape(-1,2)
```

Begin by reading in the images. The code uses `glob` to grab the entire list of files in the target folder and store it in a vector. `tqdm` provides a progress bar representing the progress of the `for` loop. Within the loop, the image is read into memory using `cv.imread()`, then converted to grayscale to improve results using `cv.cvtColor()`. `cv.Gaussianblur()` and `cv.addWeighted()` are image processing techniques which are employed to sharpen the image and enhance the border contrast of the chessboard pattern. The kernel size and weighting of the images are up to user discretion and are not numerically optimizable. More information can be found in the OpenCV documentation.

```
# Read calibration image set
calibration_pathsL = glob.glob('../IMAGES/CognexSet3/LeftImages/*.bmp')
# calibration_pathsL = glob.glob('../LearnOpenCv/CameraCalibration/images/*.jpg')
calibration_pathsR = glob.glob('../IMAGES/CognexSet3/RightImages/*.bmp')
FoundCornerCountL = 0
```

```
FoundCornerCountR = 0

for image_path in tqdm(calibration_pathsL):
        # Load the image
        print(image_path)
        imageL = cv.imread(image_path)
        # Convert to grayscale
        imageLg = cv.cvtColor(imageL, cv.COLOR_BGR2GRAY)
        # Process the image
        imageLg = cv.GaussianBlur(imageLg, (5,5), -2)
        imageLg = cv.addWeighted(imageLg, 1.5, imageLg, -0.5, 0, None)
        # Find the chessboard
        retL, cornersL = cv.findChessboardCornersSB(imageLg, chessboard_size,
                None, None)
```

If the value of `retL` is `True`, then count the chessboard as found. If not, the image will be rejected from the calibration step by virtue of not having produced a list of points. Also note that, per OpenCV documentation, `findChessboardCornersSB` has demonstrated "that the returned sub-pixel positions are more accurate than the one returned by `cornerSubPix`..." so that step is unnecessary. It is performed in the code for example purposes.

Because the image set contains photos from two cameras, this procedure is repeated for the other camera, but it is identical.

Now the `cv.calibrateCamera()` function may be employed to find the intrinsic camera matrix, distortion coefficients (the functions assume radial distortion is present), rotation vectors, and translation vectors. For the purposes of this project, the distortion coefficients and intrinsic camera matrix are of interest.

```
# Calibrate camera for each lens
print("Calibrating...")
retL, CmatrixL, distCoeffsL, rvecsL, tvecsL = cv.calibrateCamera(obj_pointsL,
        img_pointsL, imageLg.shape[::-1], None, None)
print("Left calibration complete.")
retR, CmatrixR, distCoeffsR, rvecsR, tvecsR = cv.calibrateCamera(obj_pointsR,
        img_pointsR, imageRg.shape[::-1], None, None)
print("Right calibration complete.")
```

Because the camera calibration is a time consuming process that only needs to be done once, the data is saved using `Numpy.save` to be retrieved in other programs relevant to the project. If the folder `calibParams` does not exist, it will be created.

```
# Save the outputs
# Left
np.save('./calibParams/retL', retL)
np.save('./calibParams/CmatrixL', CmatrixL)
np.save('./calibParams/distCoeffsL', distCoeffsL)
np.save('./calibParams/rvecsL', rvecsL)
np.save('./calibParams/tvecsL', tvecsL)
# Right
```

```
np.save('./calibParams/retR', retR)
np.save('./calibParams/CmatrixR', CmatrixR)
np.save('./calibParams/distCoeffsR', distCoeffsR)
np.save('./calibParams/rvecsR', rvecsR)
np.save('./calibParams/tvecsR', tvecsR)
```

The remainder of the program deals with image rectification through undistortion. Some amount of distortion is present in every image, but it is ultimately up to the user to decide whether undistorting an image is worth the time. OpenCV offers a qualitative method for calculating reprojection error—that is, the error in distance between a point of interest's true location and its location in a particular image. A value less than 1 indicates an error of less than one pixel in distance, and is thus "good". This can be helpful in diagnosing whether an image's camera conditions are significantly problematic i.e. the reprojection error for this image exceeds unity. Such images may disturb a model in undesired ways despite averaging of many samples. This will be noticeable upon undistorting them, as the process will produce an image which is very much non-representative of the world. Such images should likely be removed from the calibration set, and discarded.

To show that the reprojection error for an image set is very low, import an image as before with `cv.imread()`, and acquire its dimensions using the `.shape` property. The OpenCV function `cv.getOptimalNewCameraMatrix()` provides an altered camera matrix which omits black pixels. These pixels are "virtual" pixels which exist outside of the originally captured image. The fewer there are in the resulting output, the less distorted the original image was.

```
# Undistort an image
NCMatrixL, ROIL = cv.getOptimalNewCameraMatrix(CmatrixL, distCoeffsL,
        (RXL,RYL), 1 ,(RXL,RYL))
RMSLUnDist = cv.undistort(RMSImageL, CmatrixL, distCoeffsL, None, NCMatrixL)

# Display the undistorted image
RMSImageL = cv.resize(RMSImageL, (1000,1000))
RMSLUnDist = cv.resize(RMSLUnDist, (1000,1000))
cv.imshow('pre', RMSImageL)
cv.waitKey(0)
cv.imshow('post', RMSLUnDist)
cv.waitKey(0)

# Calculate reprojection error
mean_errorL = 0
for i in range(len(obj_pointsL)):
img_pointsL2, _ = cv.projectPoints(obj_pointsL[i], rvecsL[i], tvecsL[i],
        CmatrixL, distCoeffsL)
errorL = cv.norm(img_pointsL[i], img_pointsL2, cv.NORM_L2)/len(img_pointsL2)
mean_errorL += errorL
```

With calibration completed, the resulting camera matrices can be used so long as the reprojection error is within specifications and the output makes sense to the user.