Camera calibration using OpenCV

Wilbert Pumacay, *Catholic University San Pablo*, wilbert.pumacay@ucsp.edu.pe
Gerson Vizcarra, *Catholic University San Pablo*, gerson.vizcarra@ucsp.edu.pe

***Abstract*—Camera calibration is an important step in several applications, like augmented reality. To do calibration properly using current calibration methods we need to get features we can related in both 2D camera space and 3D world space to estimate the camera parameters that give this mapping. In this context, the use of grid patterns help by giving us the features we need, being the components in the pattern which we must detect in every frame in video.**

In this paper we use Zhang [2] camera calibration technique implemented by OpenCV, and compare results using feature extraction of chessboard corners, symmetric/asymmetric circle grids, and concentric circle grid patterns, the first two, using funnctions implemented in OpenCV and the third one using our own algorithm by implementing a pipeline that gets these features by combining various techniques from classical image processing.

***Index Terms*—Camera calibration, calibration pattern, circle grid, image processing.**

I. Introduction

THE camera calibration problem consists of finding 11 parameters that describe the mapping between 2D camera space and 3D world space. Six parameters, called extrinsic, come from an homogeneous transform, giving 6 parameters (rotation and translation around the axes). The other 5 parameters, called intrinsic, define some internal properties of the camera. This can be expressed in the following transformation equation:

$$\begin{bmatrix}\mu\\ \nu\\ 1\end{bmatrix} = \begin{bmatrix}\alpha & \gamma & \mu_0\\ 0 & \beta & \nu_0\\ 0 & 0 & 1\end{bmatrix} \begin{bmatrix}r_{x_1} & r_{y_1} & r_{z_1} & t_x\\ r_{x_2} & r_{y_2} & r_{z_2} & t_y\\ r_{x_3} & r_{y_3} & r_{z_3} & t_z\end{bmatrix} \begin{bmatrix}x\\ y\\ z\\ 1\end{bmatrix} \quad (1)$$

In this equation we can detail each of the variables: x, y, z are original coordinates of an object, r_{ij} and t_{ij} means the rotation and translation values respectively in the model matrix, α and β represents the focal length in x and y axis respectively, μ_0 and ν_0 are the coordinates x and y at the optical center.

To find these parameters, camera calibration methods make use of correspondences between 2D and 3D spaces in order to fit the parameters that best describe this mapping. We achieve this by minimizing the following function:

$$\sum_{i}^{m}\sum_{j}^{n} \|TP_{3D}^{ij} - P_{2D}^{ij}\|^2 \quad (2)$$

Where we are trying to minimize the difference between the expected projection and the actual projection over some set of features. The key idea is that supplying sufficient features that have a correct mapping, we can get the 11 parameters needed that minimize this function. So, a key part is the detection of these features.

In equation 2 we are looping through a set of features n that are found in each frame of a video of m frames, so, we basically need to detect some feature points in each frame of video, which is what we focus in this paper.

II. About the method

To compare results of feature extraction, we implemented the following pipeline.

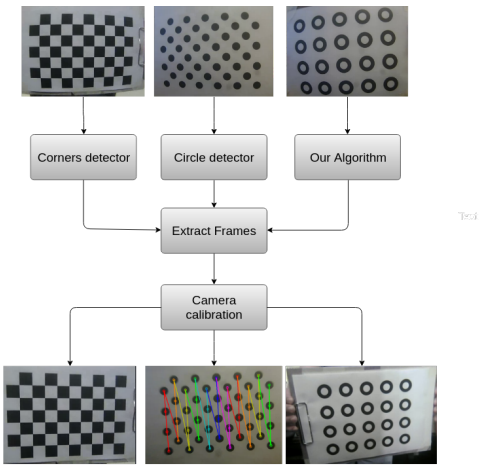

Fig. 1. Comparison pipeline.

A. Pattern detection

We used three principal patterns to calibrate the camera with OpenCV calibration: Chessboard pattern, Circle symmetric/asymmetric pattern and circle concentric pattern. In the first two patterns we applied OpenCV implemented functions, in chessboard pattern we have to recognize corners and save their position, in circle pattern we have to detect circles and save the position of their centers.

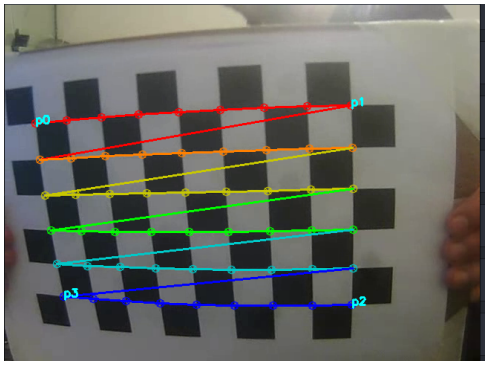

Fig. 2. Chessboard pattern detection.

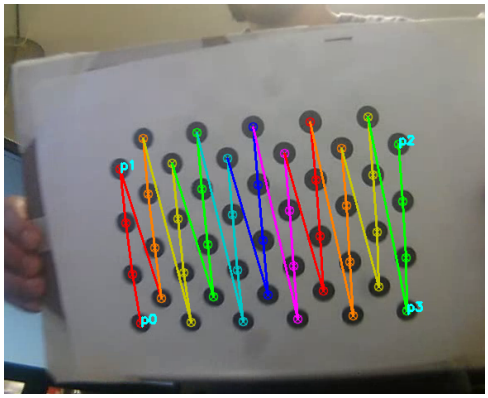

Fig. 3. Circle asymmetric grid pattern detection.

B. *Circle concentric pattern detection*

The pipeline of our pattern detection and tracking algorithm works as follows:

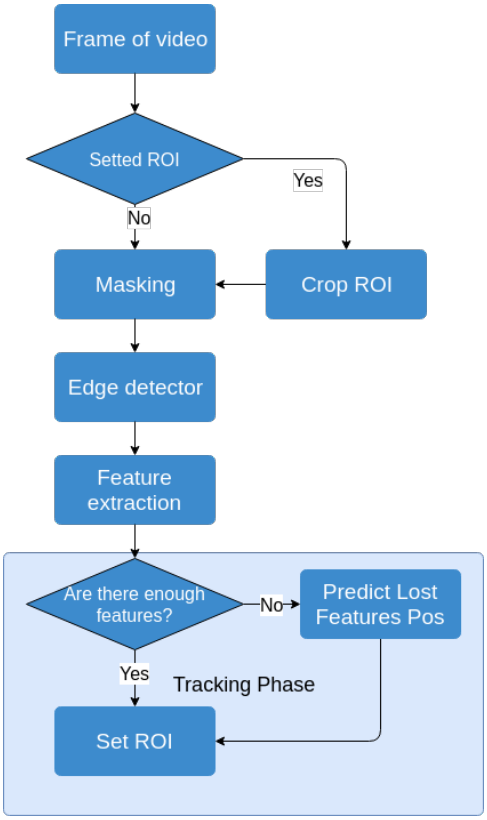

Fig. 4. Pipeline of previous work.

1) Use of ROI : We setted a ROI (Region Of Interest) based on the firsts iterations, after we extract enough features on *Feature extraction* stage, we used a ROI based on coordinates in features and a margin for predicting next frame features in order to reduce processing and avoid noise produced by ambient, we will apply next masking, edge detection, and feature extraction using ROI.

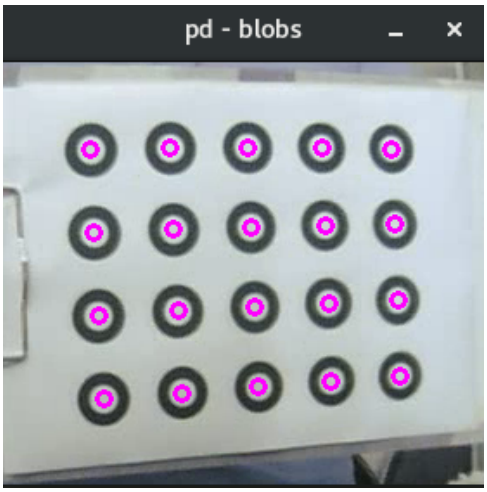

Fig. 5. ROI crop.

2) Masking : This step is in charge of isolating the pattern by using threhsolding operations.

The approach consists on creating a mask from the grayscale transformed image applying **Adaptive Thresholding algorithm**, this algorithm relies on Integral Image technique especified in [3].

Algorithm 1 Masking

1: *Set up thresholding parameters*
2: $mask =$ *rgb2gray*$(inputImage)$
3: $mask =$ *AdaptiveThreshold*$(mask, blockSize)$
4: **return** $masked$

Fig. 6. Image mask applied to ROI.

3) Edge detection: In this stage of the pipeline we extract edges from the result of the previous stage. In order to do this, we applied **Scharr operators** on *x* and *y* axis (as described in algorithm 2); Scharr is the result from Sobel algorithm minimizing weighted mean squared angular error in Fourier domain.

Algorithm 2 Edge detection

1: $axis_x =$ *Scharr*$(masked, 1, 0)$
2: $axis_x =$ *Abs*$(axis_x)$
3: $axis_y =$ *Scharr*$(masked, 0, 1)$
4: $axis_y =$ *Abs*$(axis_y)$
5: $edgesImage =$ *Add*$(axis_x, axis_y)$
6: **return** $edgesImage$

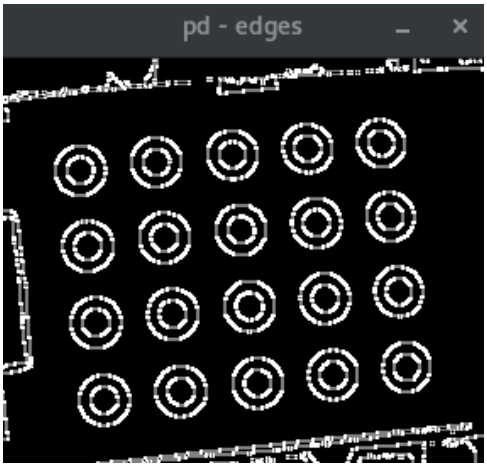

Fig. 7. Edge detection to mask.

4) Feature extraction: The last stage consist of extracting the features needed for the calibration from the edges detected in the previous stage. We used OpenCV's **SimpleBlobDetector** that apply an extra thresholding on image, applies the findContours algorithm calculating their centers, groups centers of several images by their coordinates in blobs, finally, estimates the final centers of blobs. For detecting only pattern blobs, we had to apply similar heuristics to avobe (color blobs, area, aspect ratio, and convexity of points) specified in *Algorithm 3*.

Also, in this stage we recognized the order of features detected. We started calculating the center of pattern given by the average coordinates of all features, then we computed features which are located in the corners of the pattern given by the farthest features from the center point. Next, we detected the vertical border points, selecting the nearest point to the line that pass through two corners. Similarly, we used border points to recognize the horizontal strips of points.

Finally, we used this information to give an ID number to every key point in pattern and store their location to track in next phase.

It should be noted that we only do recognition of features once, in next frames we only will extract features using *SimpleBlobDetector* and track key points result in next step.

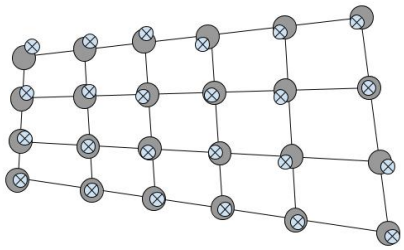

Fig. 8. Edge detection to mask.

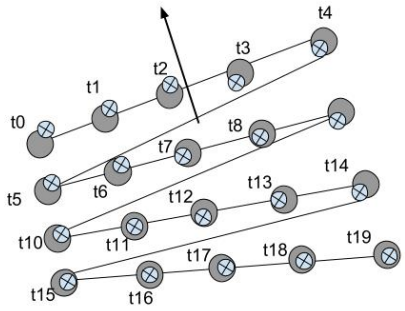

Fig. 9. Edge detection to mask.

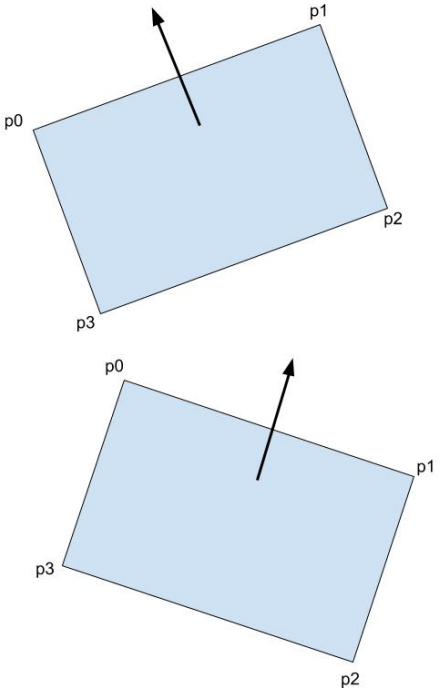

Fig. 10. Edge detection to mask.

Algorithm 3 Feature extraction

Set up detector parameters
SimpleBlobDetector$(params)$
$keypoints =$ *detect*$(mask)$
$midPt =$ *avg*$(keypoints)$
$newKeypts[corners] = farthest(keypoints, midPt)$
$newKeypts[bord] = near(newKeypts[corners], keypoints)$
$newKeypts[rest] = near(newKeypts[bord], keypoints)$
return $newKeypts$

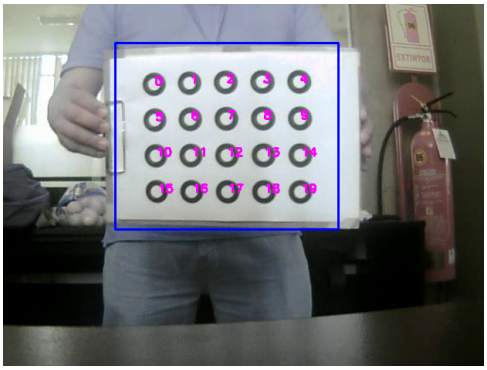

Fig. 11. Result after feature detection.

III. Results

IV. Conclusions and Future improvements

References

[1] Bradski, G.
OpenCV library. - 2000

[2] Zhengyou Zhang
A Flexible New Technique for Camera Calibration. - 2000

[3] Derek Bradley, Gerhard Roth
Adaptive Thresholding Using the Integral Image. - 2011