PROJEKTSEMINAR ROBOTIK UND COMPUTATIONAL INTELLIGENCE, VOL.17, SS 2017

Dynamic Distortion Calibration

Ahmed Ashraf, Nils Hamacher, Linghan Qian, Vivica Wirth

***Zusammenfassung*—Kameras als Sensoren werden in immer mehr Applikationem im alltäglichen Leben wie zum Beispiel in Smartphones, oder um unser Haus mit Hilfe von Smart-Home-Systemen zu überwachen, verwendet. In Fabriken hingegen werden noch viel mehr Kameras zur Überwachung von verschiedensten Prozessen wegen ihrer universellen Anwendung, niedrigen Kosten und dem Potenzial der erhaltenen Daten verwendet. Außerdem wächst ihre Rolle im Automobilbereich. Jedoch ist jede Kameralinse aufgrund von nicht exakt gleichmäßigen Fertigungstechniken mehr oder weniger gekrümmt. Die meisten Kameras zeigen eine radiale Verzerrung, da die Linsen in der Regel eine konvexe Krümmung aufweisen. Diese ist nicht immer gleichmäßig konvex oder hat Einschlüsse, die die Verzerrung zu einem sehr nichtlinearen Verhalten verändert. Das macht die Kalibrierung des Kameraobjektivs zu einem wesentlichen Aspekt der Computer Vision, da eine hohe Genauigkeit ihrer Bilder erforderlich ist. Die meisten Kamerakalibrierungen erfordern menschliche Interaktionen wie die Kalibrierung mit einem Schachbrettmuster. In dieser Ausarbeitung wird eine Lösung für die Kamera-Kalibrierung vorgestellt, die auf einem planaren Display wie gängigen Computermonitoren arbeitet und nicht auf menschliche Interaktion stützt. Dabei wird eine Karte zwischen den Bildschirmpixeln und den projezierten Karmerabildpixeln bestimmt. Wir präsentieren verschiedene Wege, die wir mit ihren Vor- und Nachteilen in Genauigkeit und Laufzeit ausprobiert haben.**

***Abstract*—Cameras as a sensor application are widely used in our daily life like in smartphones or to surveille our home via smart home installation. But even more cameras monitor many processes in factories because of their universal application, low cost and the potential of the obtained data. Also it influence in the automotive branche is rising. However, every camera lens is more or less distorted due to imperfect manufacturing techniques. Most cameras show radial distortion due to the fact that most lenses have convex curvature. These aren't always perfectly convex or got inclusions which do alter the distortion to a very non-linear behavior. That makes calibration of the camera lens an essential aspect of computer vision, since a high accuracy of their images is required. Most camera calibrations require human interactions like the calibration with a checkerboard. In this paper a solution to camera calibration is shown that works on a 2D planar display like common used computer monitors and doesn't rely on human interaction. Therefore a mapping between the screen pixel coordinate system and the image pixel coordinate system is determined. We present different ways we tried with their pros and cons in accuracy and runtime.**

I. Introduction

CAMERAS are used in more and more situations. Application areas are monitoring technologies, toys, smartphones and increasingly in vehicles for example. However all camera lenses are somewhat different from each other. The curvature of a lens is never perfect, and inclusions may occur which also alter the curvature of the incident light. This leads to distortion that has to be calibrated. State of the art approaches like the checkerboard calibration only base their models on a few interest points. This method for calibration is to position a checkerboard in different poses in front of the camera and detect the intersections of the black and white squares. The more pictures are done the higher is accuracy of the result what increases the human interaction and the time invested. We present this and other methods in Chapter I-A. Our aim was to develop a program, which automatically creates a dense model of the lens distortion. This dense model is based on information of the correspondencies of pixels on a screen and the pixels in a captured image. Once the mapping between the screen and the image corresponding pixels is done, we can undistort the image simply by moving the pixels of the actual image we take. In order to reach the desired accuracy we need to set some pre-conditions for the environment in which this takes place. These conditions are darkness, in order to avoid reflections of the camera on the screen, and the camera being perpendicular to the screen. We will present the fundamental principles of several different ideas, which all result in a mapping. To compare them we give a short insight to their runtimes. This article is structured as follows: First follows the related work in I-A then in Chapter II the theoretical foundations are outlined. Chapter III includes the idea and realization of the project. In Chapter IV is the conclusion.

This paper was supported by M. Sc. Raul Acuña .

A. Related Work

Several approaches has been proposed in order to calibrate a camera, they can roughly be classified into two categories [1]: photogrammetric calibration and self-calibration. Photogrammetric calibration is based on a spatial object with its 3D shape known which is placed in the view of camera. The commonly used object is planar object point array, so called checkerboard pattern [1], or a 3D cube with determined pattern on it. This method is restricted because errors will occur while printing and manually measuring the pattern or also if the pattern isn't always available.
In self-calibration [3] a reference object is not required, instead it requires the correspondence between points of the scene and the projection. Duane C B. [5] proposed a nonmetric method, based on the fact that the projection of a straight line from the world space should also be a straight line on the image. Swaminathan R and Nayar S K. [6] extended this method by reducing noise and using polycamera. However, this method still relies on calculating parameters, which is not accurate because the model is simplified. Sagawa R proposed a way using structured light to determine the correspondence and

creates directly a dense map so that the error generated by parameter fitting is reduced. The system proposed is based on [4] since they are both based on detecting the correspondence of lines and get the map non-parametrically. [1]

II. Mathematical Backround

III. Mapping of correspondencing pixels

Our aim was to create a map of corresponding pixels of the screen to the camera image. For that we followed different approaches which we evaluate by accuracy and especially by runtime in Chapter III-C.

A. *Pixelwise approaches*

1) *every Pixel:*

2) *spiral:*

3) *by lines:*

B. *simple linedetection for x and y matching*

We set the backround of the screen to black. Then we draw a white rectangle with height of the screenheight and an increasing width. At the border of the white rectangle to the black backround the *Canny*-operator detects the points the border as a line. We match to every image pixel that belongs to that line the x coordinate to which it should belong. In figure 1 we show how the rectangle is getting wide.

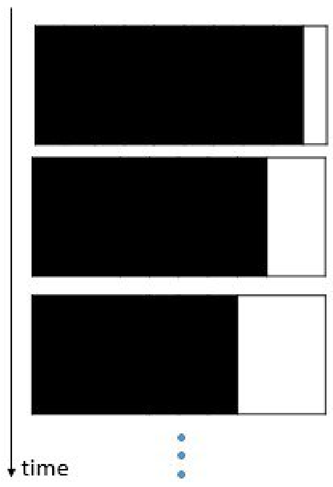

Bild 1. Rectangle over time

C. *runtimes of the approaches*

1) *pixelwise:* Each frame makes one call to the draw function. Given that we need 7 calls of the draw function to fully draw, capture, and process one image, we can, with a frame rate of 21 frames per second (FPS), achieve turnover rate of 3 images per second. This allows us to make an estimation of the runtime of each approach we tested out or considered for the lens calibration. For the following estimates we're assuming a screen of the dimensions $1290x1080$ pixels and a field of view (FOV) of roughly $670x395$ pixels for the camera. The FOV has been experimentally determined for a distance of 16 cm from the screen. From the results the aperture angle can be calculated, it yields roughly $68.45°$, the camera's specifications state an aperture angle of $68.50°$. So these results should be a good estimate of the true FOV. Furthermore the assumption is made that each pixel can be detected on its own, instead of needing larger patches to be detectable by the camera. The following variables will be use: h_{FOV} as the height of the FOV, w_{FOV} as the width of the FOV, w_s as the width of the screen, and h_s as the height of the screen. 1. Singular pixel detection With a naive approach each patch of minimal detectable size on the screen has to be turned on separately. This means screen width times screen height necessary images. In our set up $921,600$ images would be needed, yielding a runtime of $307,200$ seconds, or 3 days 13 hours 20 minutes. It can be sped up however. By lighting up single pixels in a spiral pattern around an estimation of the middle point, the amount of images can be reduced to the amount of pixels within the camera's FOV runtime to roughly $88,572$ seconds, or 1 day 1 hour 30 minutes. The center point estimation has been left out, as it is comparably negligible.

2) *Line-based detection:* The first idea for a line-based detection was to first detect the borders and store how the border pixels of the image map to the screen. The idea was to first detect a pixel on the border via binary search and then move this pixel to new positions to check whether these are seen. We need to check at least four surrounding pixels for visibility. This means $4*(2*h_{FOV}+2*w_{FOV})$, in our setup that leads to $8,520$ images. To process these images, we need $2,840$ seconds, or 47 minutes 20 seconds. After discovering the border only two images need to be created and processed. However, since the discovery of the border would take still too long, we rejected this method as well. The next method is a segmentation of the screen. By segmenting the screen sequentially in a black and a white part, we can detect a single edge in the image. The white part starts out as the entire screen and gradually decreases in width, pixel by pixel. After that is done, the white part is again stretched out over the entire screen and then decreases in height. At each image we know exactly at which position the screen shows the edge. From this we can calculate where the edge pixels from the image should have appeared. When doing this naively and simply going over the entire screen width and height, this method needs to process $2,370$ images. It would take 790 seconds, or 13 minutes 10 seconds. This is already immensely faster. It can still be sped up, however. Instead of going over the entire screen, we first estimate the center point again. This time we additionally store which was the outermost pixel, which was visible during the estimation, for each border. From there we add the spacing, that was used, plus a small margin to be on the safe side. The margin was chosen to be 5 pixels wide. In this method the runtime for the center point estimation is relevant. While the runtime for the center point estimation increases with smaller spacing, the runtime for the line-based

detection decreases. This means a runtime-optimal spacing can be calculated. Let s be the spacing between the positions of the pixels to be lit, and m the margin. Then the runtime of the center estimation, as a function of spacing s, is:

$$rt_c(s) = \frac{w_s \cdot h_s}{s^2} = 1393200/s^2. \tag{1}$$

For the runtime of the line-based detection after that, follows:

$$rt_d(s) = 4s + 4 \cdot m + w_{FOV} + h_{FOV} = 4s + 1085 \tag{2}$$

From there the overall runtime can be calculated as:

$$rt(s) = \frac{1393200}{s^2} + 4s + 1085. \tag{3}$$

This functions minimum is at roughly s = 89. For this spacing we need then $1,617$ images, leading to a runtime of 539 seconds, or nearly 9 minutes. Of these roughly 59 seconds are center point estimation.

IV. Results

A. *accuracy*

B. *comparation and valuation*

Quellen

[1] O. Faugeras, Q.T. Luong and S. Maybank, *Camera self-calibration: Theory and experiments, pages: 321–334 , Computer Vision—ECCV'92*, 1992

Autor A Biographie Autor A.

Autor B Biographie Autor B.

Autor C Biographie Autor C.

Autor D Biographie Autor D.