REGELUNGSMETHODEN UND ROBOTIK

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. C.B. Duane proposed in [9] 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. R. Swaminathan and S.K. Nayar extended in [7] 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. In [5] a way is proposed 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 [5] since they are both based on detecting the correspondence of lines and get the map non-parametrically.

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 in Chapter IV and by runtime in Chapter III-C.
Before we begin with the mapping of the correspondencies we will discuss the setup of the device. The camera should be in a distance to the screen that it only detects the screen and nothing else like the borders or so. We should be able to manipulate the entire visible flat surface to excite every pixel on the camera image. The background color of the monitor should be black so that drawn white pixels can be better recognized. The camera and the display should be in a locked dark area where no light is coming so the monitor is the only light source. This leads to no unwanted reflections that sophisticate our result. If the camera is adjusted perpendicular to the display the program could run.

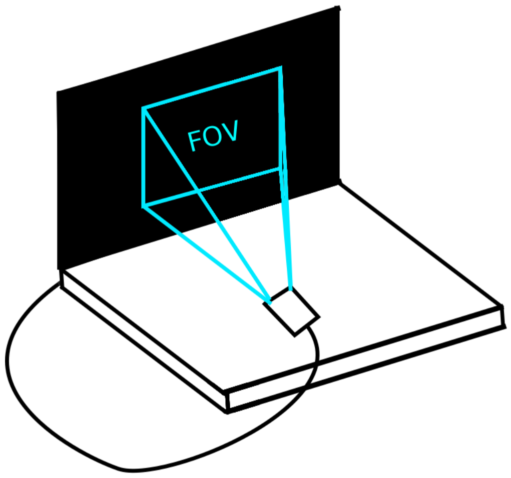

Bild 1. Setup of Camera and Monitor

First we have to make sure, that the captured image contains at least one position at which the screen is going to display a rectangle. To do this we draw a set amount of rectangles evenly spaced on the screen. Initially these rectangles have a *width* and *height* of 1. While the camera doesn't detects a pixel we increase *width* and *height* of the drawn rectangle by 1 until it does recognise a drawn pixel. This is shown in algorithm 1. In order to ensure, that the detection was not a accidental chance of luck, the same size has to be confirmed by further detections.

After the pixel size that is seen by the camera is detected we draw a pixel with that size on the screen that is moving with a spacing. That means that the pixel always gets a new coordinate that with a calculated distance in vertical and/or horizontal direction. Therefore we're able to calculate an

```
Data: cameraImage
Result: returns pixelSize
initialization: int pixelSize = 1;
while (notDetected) do
    if (maxBrightnestPixel != 0) then
        notDetected = false;
        return pixelSize;
    else
        notDetected=true;
        pixelSize++;
    end
end
```

Algorithm 1: pixel size detection

optimal spacing for a specific method of us which is described at the end of Chapter III-C.3. While the pixel is moving on the screen we accumulatively add the x_s and y_s positions of the seen rectangles on the screen. Simultanously we enumerate the number of seen rectangles. The algorithm is shown in 2.

```
Data: pixelPosition
Result: returns centerPoint
initialization: int counter = 0;
while (pixelPosition < display) do
    if (pixelSeen) then
        centerPoint += pixelPosition;
        counter++;
    else
        centerPoint /= counter;
    end
end
```

Algorithm 2: calculation of center point of FOV

As you will see in the following chapters, these two methods are fundamental to most of our approaches. Where the pixel-Size is not necessary for the algorithm itself, we still use it to generate a test image. To this test image the found distortion maps are applied. Therefore they are necessary in any case. In the next three Subchapter we'll present two principle different possibilities we obtained while this work how it could be possible to get a distortion calibration with a higher accuracy than the state of the art.

A. *Pixelwise approaches*

Our first idea was a matching from every screen pixel to his corresponding pixel in the image. We developed three different methods. These were faster in their implementation in this order, with the same accuracy. The duration of the individual methods is given in Chapter III-C with an exact minimum value for our example.

1) every Pixel: The first and easiest idea was to light a rectangle with minimum pixel size after another and row for row. If we see a drawn rectangle save the correspondencies of the image and the screen coordinate. Later we could have checked if an imaginary straight line of rectangles on the screen will be to a straight line of pixels on the image. If not we knew we have to move the image pixels so that the line that is seen on the image is also straight.

*2) For the sake of the **SPIRAL**:* However it's not very efficient if we light every pixel of the screen with the seen pixel size. As the seen area on the screen the field of view (FOV) could be significant smaller than the screen we developed another method to ensure a faster result. For this purpose we produce generate the idea to let the rectangle move on a spirallike path on the screen. The algorithm 2 was designed to find a optimal starting position to that spiral. After finding the center point of the FOV the rectangle moved slowly to the outside of the FOV. While the pixel wasn't seen for a hole circulation we know we should still be in vision of the camera. So everytime a rectangle is seen it's position on the screen and the camera image is saved for later for the distortion calibration. Exposing the view area in spiral-shape, as described in Chapter III-C, is not exactly fast. In addition, the method is very inefficient for fisheye cameras or any kind of wide angle cameras.

3) by lines: The pixelwise distortion calibration was our last concept in which we tried to detect a solution by lighting several rectangles with the minimum pixel size. However this method is already a hybrid since we introduced real line detection methods here to solve the problem of distortion calibration. In the further discussed methods we used the pixel correspondencies and calculated the lines in both images, the drawn and the detected one. In this method we drew lines in the seen area and detect their position in the camera image and wanted to correct them then. However we need to detect the borders of the seen area first. For that we are forced to find at least one border of the seen area. Therfore we developed a binary search algorithm that finds a border. The initial point to that algorithm is the center point we already detected with algorithm 2. From there we assumed that a point at any border of the screen is not seen by the camera. Then we check if a point between this two points is seen or not. If not we check between this and center point and else we check between that and the point of the border. This is described in algorithm 3.

```
Data: pixelPosition
Result: returns borderPoint
initialization: ;
point lastSeen = centerPoint;
point lastNotseen = screenBorder;
point nextPos;
while (abs(lastSeen - lastNotSeen)>1) do
    nextPos = (lastSeen+lastNotSeen)/2;
    if (pixelSeen) then
        lastSeen = pixelPosition;
    else
        lastNotSeen = pixelPosition;
    end
end
pixelPosition = nextPos;
return borderPoint = (lastSeen+lastNotSeen)/2;
```
Algorithm 3: binary search algorithm for border detection

In the process of this work the binary search showed that absolute darkness around the setup is necessary because otherwise we're losing vision on pixels in the outer area of the camera image.
After a border is found our purpose was to find all corresponding screen pixels that belongs to the frame of the camera image. If these are found our idea was to draw lines with *width* for vertical lines and the *height* for horizontal lines of the minimum pixel size on the screen that are straight. Then we should have been able to calibrate the distortion with the next two shown images. However it would have needed still a long time to detect the borders. Since around a pixel 8 possible positions to detect the next one that belongs to the frame exist. Even if you can exclude some of these possible positions if you know at which border you are and in which directions you want to move around the frame, at least 4 possible positions remain. In Chapter III-C we calculated the minimum time required for an exemplary FOV. However we dismissed this method too because we found a way faster solution with a line detection by a *openCV*–function called *Canny*.

B. *simple linedetection for x and y matching*

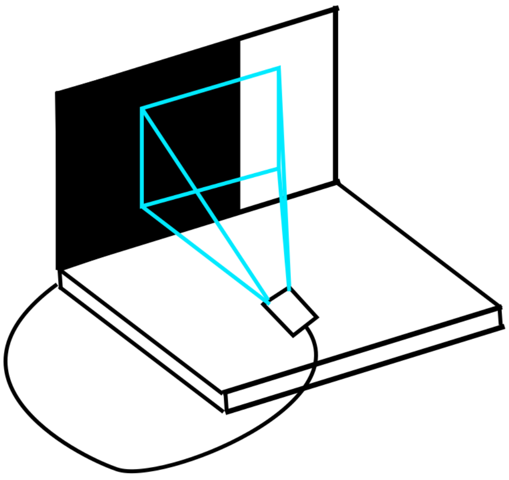

Bild 2. rectangle over the screen

We set the backround of the screen to black. Then we draw a white rectangle with height of the screen height and an increasing width. In figure 3 we show how the rectangle is getting wide. At the border of the white rectangle to the black backround the *Canny*-operator detects the points of the border as a line. The return of that operation is a point cloud of image coordinates that belong to that line. To every image pixel that in that point cloud we match the x–coordinate to which it should belong. We enlarge the width of the rectangle until it disappears from the FOV. After that we do the same with another rectangle that is as width as the screen width and height of one minimum pixel size *height*. We'll get again every pixel that belongs to that line in a point cloud by the *Canny*–operator. Now we can match the y–coordinates to the points which are obtained, since they always belong to the actual drawn line. At the end of Chapter III-C we calculated the runtime of this method.

It is significantly faster than any of the other methods we have presented so far. However, it can be accelerated even more, if not always a pixel size is increased, but several. Therefore we wrote a interpolation function that interpolates linearily between matched pixels. In future it could be tested if another interpolation method besides linear interpolation give better results. In case of radial distortion a radius to the center point dependent interpolation could return better results.

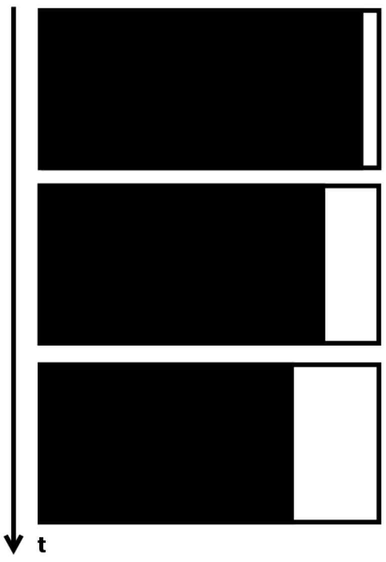

Bild 3. Rectangle over time

C. runtimes and seen area

1) Field of View: 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, 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 based on the amount of necessary images. The runtime rt of the program could be calculated as shown in the following Chapter.

$$rt(noImages) = \frac{1}{3} \cdot noImages \tag{1}$$

To determine the amount of images for some methods, we need an estimate of the field of view. In order to estimate the FOV we used a simple method with a measuring tape. The only further information necessary, is the amount of pixels per inch (ppi) of the used screen. By placing the measurement tape flat on the screen, we can measure the width and height visible in the captured image. The measurement has then to be converted to inches in order to calculate the amount of pixels. The conversion constant from centimeters to inches is roughly 0.3937. Multiplying this with the ppi yields the pixels seen. Let l be the length of the visible measurement tape in centimeters, then the amount of pixels in the FOV can be calculated as:

$$FOV(l) = ppi \cdot 0.3937 \frac{inch}{cm} \cdot l. \tag{2}$$

From this we can also calculate the diagonal aperture angle of the camera. By comparing the calculated angle with the angle from the specifications of the camera, we have an estimation about the accuracy of the calculated FOV. If the diagonal aperture angles are differ largely, our estimation is inaccurate. If they are similar, it is acceptable.
To calculate the aperture angle α for the width or height $(\alpha_{width||height})$, we also need the distance d of the camera to the screen. The distance and FOV dimensions need to be used with the same unit. From there the aperture angle is simple trigonometry, see image ... .

$$\frac{1}{2} \cdot \alpha_{width||height} = \arctan\left(\frac{FOV_{width||height}}{2 \cdot d}\right), \tag{3}$$

$$\Rightarrow \alpha_{width||height} = 2 \cdot \arctan\left(\frac{FOV_{width||height}}{2 \cdot d}\right), \tag{4}$$

$$\alpha_{diag} = \sqrt{\alpha_{width}^2 + \alpha_{height}^2)}. \tag{5}$$

In our setup we used an *ASUS–VX239* screen of dimensions 1290x1080 pixels and 95.78 ppi and a *Microsoft LifeCam HD–3000*. We weren't able to place the camera exactly perpendicular to the screen. This warps the FOV somewhat. Therefore we used an average of the largest and smallest measurement of the FOV width and height. On average we measured a width of 17.75 cm, and a height of 10.5 cm. This yields us FOV dimensions of

$$FOV_{width}(17.75\ cm) \approx 669.327\ pixels$$

$$FOV_{height}(10.45\ cm) \approx 394.050\ pixels$$

Since we would rather overstate the runtime than understate it, we used 670 pixels for the width and 395 for the height. The distance between camera and screen was roughly 16.0 cm. This yields aperture angles of:

$$\begin{aligned}\alpha_{width}(17.75\ cm) &= 2 \cdot \arctan\left(\frac{17.78\ cm}{2 \cdot 16\ cm}\right), \\ &= 58.12^\circ\end{aligned} \tag{6}$$

$$\begin{aligned}\alpha_{height}(10.45\ cm) &= 2 \cdot \arctan\left(\frac{10.45\ cm}{2 \cdot 16\ cm}\right), \\ &= 36.17^\circ\end{aligned} \tag{7}$$

$$\begin{aligned}\alpha_{diag} &= \sqrt{58.12^2 + 36.17^2} \\ &\approx 68.46^\circ.\end{aligned} \tag{8}$$

The specifications for the camera state a diagonal aperture angle of 68.50°. Judging from the difference of roughly 0.04°, this should be close enough to the true FOV dimensions to use it for runtime estimations.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.

2) Singular pixel detection: With a naive approach each patch of minimal detectable size on the screen has to be turned on separately.

$$rt(w_s, h_s) = \frac{1}{3} \cdot w_s \cdot h_s \tag{9}$$

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 plus one pixel in each dimension. The additional pixel is necessary, because it allows us to know, when the spiral does not need to actually be done, but can jump to a next position. This is due to the pixel outside no longer being detected.

$$rt(w_{FOV}, h_{FOV}) = \frac{1}{3}(w_{FOV}+1)\cdot(h_{FOV}+1) \tag{10}$$

In our setup, we need 265,716 images, leading to a runtime of 88,572 seconds, or 1 day 36 hours 12 seconds. The center point estimation has been left out, as it is comparably negligible.

3) 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. For the runtime of only the border detection, once the binary search is done, this means

$$rt(w_{FOV}, h_{VOF}) = \frac{4}{3}(2\cdot h_{FOV} + 2\cdot w_{FOV}) \tag{11}$$

In our setup, we need $8,520$ images, leading to a runtime of $2,840$ 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.

A naïve and simple approach is to go over the entire screen width and height. Each row and each column is once the edge row or column. This leads to a runtime of

$$rt(w_s, h_s) = \frac{1}{3}(w_s + h_s) \tag{12}$$

In our setup, we need $2,370$ images, leading to a runtime of 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 safe. 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:

$$rt_c(s) = \frac{1}{3}\frac{w_s\cdot h_s}{s^2} = \frac{1393200}{3s^2}. \tag{13}$$

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

$$\begin{align} rt_d(s) &= \frac{1}{3}\left(4s + 4\cdot m + w_{FOV} + h_{FOV}\right) \tag{14}\\ &= 4s + 1085 \tag{15}\end{align}$$

From there the overall runtime can be calculated as:

$$\begin{align} rt(s) &= \frac{1}{3}\left(\frac{w_s * h_s}{s^2} + 4s + 4m + w_{FOV} + h_{FOV}\right) \tag{16}\\ &= \frac{\frac{1{,}393{,}200}{s^2} + 4s + 1085}{3}. \tag{17}\end{align}$$

For calculating the minimum of this function we take the first derivative w.r.t. spacing s and set it to zero:

$$\begin{align} rt'(s) &= \frac{1}{3}\cdot\left(\frac{-w_s\cdot h_s}{s^3} + 4\right) = \frac{\frac{-1{,}393{,}200}{s^3} + 4}{3}, \tag{18}\\ \Rightarrow s &= \sqrt[3]{\frac{w_s * h_s}{4}} = \sqrt[3]{\frac{1393200}{4}}. \tag{19}\end{align}$$

The functions minimum is at roughly $s = 70$. 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 taken by center point estimation.

If this is deemed too slow, but the restriction of true pixelwise mapping is lifted, it can be sped up. We skip a set amount of pixels, when moving the white rectangle over the screen. We can introduce the jump width as variable j, i.e. every j–th pixel will be part of an edge. This changes our runtime formula for the detection part to, when assuming the same margin m:

$$\begin{align} rt_d(s;j) &= \frac{1}{3}\cdot\frac{4*s + 4*m + w_{FOV} + h_{FOV}}{j} \tag{20}\\ &= \frac{4s+1085}{3j}, \tag{21}\end{align}$$

and thus the overall formula and its derivative to:

$$\begin{align} rt(s;j) &= \frac{1}{3}\left(\frac{w_s\cdot h_s}{3} + (4s + 4m + w_{FOV} + h_{FOV})\frac{1}{j}\right)\\ &= \frac{1}{3}\left(\frac{1,393,200}{s^2} + \frac{4s+1085}{j}\right), \tag{22}\end{align}$$

$$\begin{aligned} rt'(s;j) &= \frac{1}{3}\left(\frac{-w_s * h_s}{s^3} + \frac{4}{j}\right) \\ &= \frac{-1,393,200}{3s^3} + \frac{4}{3j} \end{aligned} \tag{23}$$

The new optimal spacing is now calculated as:

$$\begin{aligned} s &= \sqrt[3]{\frac{w_s \cdot h_s \cdot j}{4}} \\ &= \sqrt[3]{\frac{1,393,200 * j}{4}}. \end{aligned} \tag{24}$$

For a skip of $j = 3$ we get in our setup an optimal spacing of roughly $s = 101$. This spacing and skipping, we need 633 images, leading to a runtime of 211 seconds, or 3 minutes 31 seconds. Of these roughly 46 seconds are taken by center point estimation.

IV. Results

A. accuracy

B. comparation and valuation

V. Problems

Several problems appear during the process of this project. The first is the use of new programming tools. Openframeworks is a toolkit with powerful embedded functions (setup(), update() and draw()) which has an advantage that draws parallel to processing, however, it brings timing issues in our project, since the update function only runs once before drawing. Getting rid of it costs the first several weeks and slowed the process down. Ultimately we divided each app into several states in draw(), and use a state machine to control the process, which turns out to be a good solution.

The next problem is, several classes with their own functions has been written to achieve a modularly structured program. A master–class was created to bring these module–classes and their functionality together. So each module–class brings a functionality and together they solve th our problem to undistort a image. This splitting of the code made it necessary to retain information gained in each part. This, however, was a problem. Any class that is supposed to draw on the screen, needs to inherit from openframeworks class ofApp. Any class that inherits from class ofApp, has to be started with the openframeworks function ofRunApp.

While the runtime of each App, the only way to access data is to work with multi-threading and trying to access variables from another thread. Since it have to be ensured, that there are no read/write access violations and that the threads are properly synchronized, this method is problematic.

There is also a problem to access information after the runtime of the apps, since after the runtime of the app all variables used within are deleted. To overcome this disadvantage, a pointer of the app is introduced to a variable outside of it. This ensured that the app wrote into that variable. Since the variable wasn't within the scope of the app, the information in it was retained even after the app was closed and deleted.

The modular program structure was important for us, as we wanted to keep an overview of the code itself. It also allows functionalities easily replaceable as long as the defined interfaces are kept the same.

We also set ofSetBackgroundAuto(false) to avoid flickering of the screen, which brings has problematics since for Windows 10 systems because of incompatibility.

Quellen

[1] Zhang and Zhengyou. *A flexible new technique for camera calibration*, p.965–980, vol.14 num.10, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2010

[2] P.F. Sturm and S.J. Maybank. *On plane-based camera calibration: A general algorithm, singularities, applications*, Computer Vision and Pattern Recognition, IEEE Computer Society Conference on. IEEE, p.432–437, 1999

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

[4] Y. Ma. *An invitation to 3-D vision: From images to geometric models, Springer , 978-1-4419-1846-8*

[5] R. Sagawa and M. Takatsuji and T. Echigo and Y. Yagi . *Calibration of lens distortion by structured-light scanning* IEEE/RSJ International Conference on Intelligent Robots and Systems, p.832–837 , 2005

[6] J. Salvi and J. Pagès and J. Batlle. *Pattern codification strategies in structured light systems*, vol.37 num.4, Pattern Recognition, p. 827–849, 2004

[7] R. Swaminathan and S.K. Nayar. *Nonmetric calibration of wide-angle lenses and polycameras*, p.1172–1178, vol.22 num.10, IEEE Transactions on Pattern Analysis and Machine Intelligence

[8] J. Weng and P. Cohen and M. Herniou. *Camera calibration with distortion models and accuracy evaluation*, p.965–980, vol.14 num.10, IEEE Transactions on Pattern Analysis and Machine Intelligence, 1992

[9] C.B. Duane *Close-range camera calibration*. Photogramm. Eng, vol.37 num.8, p.855-866. 1971

Ahmed Ashraf is from egypt. He likes sport and Matlab. · · ·

Nils Hamacher Biographie Autor B.

Linhan Quan Biographie Autor C.

Vivica Wirth Biographie Autor D.