DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Frame Marker Evaluation for mobile Applications with architectural Purpose

**Markus Hamberger**

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

# Frame Marker Evaluation for mobile Applications with architectural Purpose

# Frame Marker Evaluierung für mobile Architektur Anwendung

| | |
|---|---|
| Author: | Markus Hamberger |
| Supervisor: | Prof. Gudrun Klinker |
| Advisor: | Linda Rudolph |
| Submission Date: | 16.09.2019 |

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 16.09.2019 Markus Hamberger

# Acknowledgments

I would like to thank Linda Rudolph for her commitment and her excellent support while working on this bachelor's thesis. Especially I would like to thank her for her advise on the implementation concept and for her help with the measurements.

# Abstract

In order to rehabilitate, repair or remodel buildings it is necessary to know the position of the installations in the wall, such as pipes, cables ,and structural elements. Currently, this is still determined by blueprints or special sensors. With AR it is possible to project the installations on the wall via smartphone.

For the implementation, markers are needed as orientation points. The idea is now to design a so-called frame marker that has a cutout in the middle, so you can attach it to light switches and sockets.

The aim of this thesis is to create an application on a smartphone, which is able to detect a frame marker, which can ,later on, be used as a reference point in the 3D world space in order to display pipes, cables ,etc going through the wall. In addition, the quality of detection, the maximum detection distance, and the viewing angle are measured and evaluated in this work.

# Kurzfassung

Um Gebäude zu sanieren, zu reparieren oder umzubauen, ist es erforderlich die Position der Installationen, wie Rohre, Kabel und Tragelemente, in der Wand zu kennen. Derzeit wird dies noch mit der Hilfe von Bauplänen oder speziellen Sensoren bestimmt. Mit AR ist es möglich, die Installationen per Smartphone an die Wand zu projizieren.

Für die Implementierung werden allerdings Marker als Orientierungspunkte benötigt. Die Idee ist nun einen sogenannten Rahmenmarker zu entwerfen welcher in der Mitte eine Aussparung hat, damit man ihn an Lichtschaltern und Steckdosen anbringen kann.

Ziel dieser Arbeit ist es, eine Anwendung auf einem Smartphone zu entwerfen, welche einen Rahmenmarker erkennen kann, welcher später als Referenzpunkt im 3D-Raum verwendet werden kann, um Rohre, Kabel usw. anzuzeigen, die durch die Wand verlaufen. Zudem wird in dieser Arbeit auch die Erfassungsqualität, der maximale Erfassungsabstand und der Betrachtungswinkel gemessen und ausgewertet.

# Contents

**Acknowledgments** iii

**Abstract** iv

**Kurzfassung** v

**1. Introduction** 1

**2. Basics of Computer Vision** 3
2.1. Color Spaces . . . 3
2.1.1. RGB Color Space . . . 3
2.1.2. RGBA format . . . 4
2.1.3. YUV color model . . . 4
2.1.4. Grayscale conversion . . . 5
2.2. Canny Edge Detection . . . 5
2.3. Douglas-Peucker algorithm . . . 7
2.4. Pinhole Camera Model . . . 7
2.5. Extrinsic Camera Parameters . . . 9
2.6. Perspective Transformation . . . 9
2.7. Camera Distortion . . . 10
2.7.1. Radial Distortions . . . 10
2.7.2. Tangential Distortion . . . 11
2.8. Camera Calibration . . . 11

**3. Related Work** 14
3.1. ARBlocks . . . 14
3.2. Studierstube Tracker . . . 14
3.3. Differences in Implementation . . . 15

**4. Concept** 16

**5. Implementation** 17
5.1. Marker Type . . . 17
5.2. Different approaches . . . 17
5.2.1. Outside Edge approach . . . 17
5.2.2. Inside Edge approach . . . 17
5.3. Loading the Image . . . 18

5.4. Marker Detection . . . . . 18
5.5. Marker Identification . . . . . 21
5.5.1. Bits extraction . . . . . 21
5.5.2. Error correction . . . . . 22
5.5.3. Orientation . . . . . 22
5.6. Marker Pose Estimation . . . . . 22
5.7. Visualization . . . . . 23
5.7.1. Contour, corners, 2D marker and code . . . . . 23
5.7.2. Axis . . . . . 25

**6. Evaluation** **26**
6.1. Using the paper marker on a socket frame . . . . . 26
6.2. Measurements . . . . . 27
6.2.1. Measurement Setup . . . . . 27
6.2.2. Distance Measurement . . . . . 27
6.2.3. Viewing Angle Stability . . . . . 29
6.2.4. Translation Vector Error . . . . . 29

**7. Conclusion** **31**

**8. Future Work** **32**

**A. Software** **33**

**B. Smartphone Specifications** **34**

**C. Translation Vector Measurements** **35**

**List of Figures** **39**

**List of Tables** **40**

**Bibliography** **41**

# 1. Introduction

The term augmented reality (AR) refers to a computer-assisted perception or representation that expands the real world with virtual aspects. Unlike virtual reality, where the user moves through a completely artificial world, in AR virtual objects are inserted in the correct position in the real environment.

AR is finding more and more applications and approval in the course of Industry 4.0. Although the beginnings of AR date back to 1960s, where Sutherland already used a head-mounted display to show 3D graphics [1], the application of AR has just increased strongly in recent years. The augmented reality and virtual reality market projects a feasible market valuation of USD 767.67 billion by the year of 2025. A report by Market Research Future predicts even a 73.3% compound annual growth rate for the next 6 years. [2] This rapid growth rate can be explained by the numerous potential application areas such as entertainment, gaming, military, education and architecture.

For instance with the 'smart reality' app from JBKnowledge it is possible to display 3D buildings or other constructions in AR. 3D BIM models, which architectures are using to design and document buildings and infrastructure designs, can be directly displayed on e.g. a smartphone or a tablet by using the blueprint as a reference point. [3] This has the advantage that you can walk around your 3D model and discuss certain design option with a direct representation. The augmented model also helps stakeholders to understand what the building will look like before it is built. But that is only one already existing example of where AR can be used.

In order to rehabilitate, repair or remodel buildings it is necessary to know the position of the installations in the wall, such as pipes, cables, and structural elements. Currently, this is still determined by blueprints or special sensors. With AR it is possible to project the installations on the wall via smartphone. However, for this sort of application, a marker for tracking is needed. Ideally, the marker should stay on the wall permanently so it will stay in exact the same spot, which is needed when you want to display the installations in the place where they actually are.

The idea is now to design a so-called frame marker that has a cutout in the middle, so you can attach it to light switches and sockets. The advantages of that, compared to already existing markers, are that the frame marker on a socket or light switch is much more unobtrusive and does not need to be extra marked in the blueprint since it has the same position in the construction plan as the socket or light switch. So when designing the 3D models of the installations, you exactly know where the marker will be and since it can stay permanently on the wall it will not change place. In this thesis a frame marker, which can be put around socket frames, and an application for a smartphone, which is able to detect

the marker will be designed. Furthermore, the marker is tested in terms of the quality of detection, the maximum detection distance, and the viewing angle.

# 2. Basics of Computer Vision

Nearly all marker tracking or object detection applications have a similar procedure. In the beginning there is always data collection via sensors, depending on the environment and use case. Sensors can be e.g. radar, lidar or a camera. Second, these pictures are processed in order to minimize disturbances. In the next step, certain features are searched in the processed picture. The features can be e.g edges, corners, colors, brightness. Then these features are extracted and matched with features from known objects. In the final step, the position of the recognized object or marker in the picture is outputted. [4]

In the following are some basic techniques described, which are commonly used in marker tracking. In particular, we will discuss the principles behind the techniques used in the implementation part (see chapter 5).

## 2.1. Color Spaces

The human eye is sensitive to light which has a wavelength of about 400 to 700 nanometers, which is referred to as the visible light'. The wavelength determines the perceived hue, whereas the amplitude defines the intensity of the color. The human eye has two kinds of photoreceptors in the retina, rods ,and cones. The main difference between them is that cones function very well in bright light, whereas rod cell function in dim light. There are also three different types of cones and rods, which each are sensitive to different wavelengths. In combination, they are responsible for color discrimination. In dependence on the stimulation of the three types of receptors, a certain color sensation is produced. Although the sensitivities of the receptors overlap, they are especially sensitive to red, green or blue color (see figure 2.1). That is why nearly any color can be described by a combination of red, green and blue wavelengths, which mimics the stimulus this color would arise in the cones and rods. [5]

### 2.1.1. RGB Color Space

Based on the model that nearly any color can be represented by a combination of red green and blue, the RGB color space is defined. The RGB color space is defined by an orthogonal threespace whose axes are the red green and blue primary colors. Therefore color can be represented as a point in this cubic space. The r (red),g (green),b (blue) component each range from 0 to 1 or from 0 to 100%. [5]

Since computers work with discrete values, the continuous range from 0 to 1 has to be quantized. The value range for each color axis is commonly between 0 and 255, where 0 represents the minimum value and 255 the maximum. So every color is described by an 8 bit long value for each red green and blue. Therefore it is possible to display $256^3$ different colors.

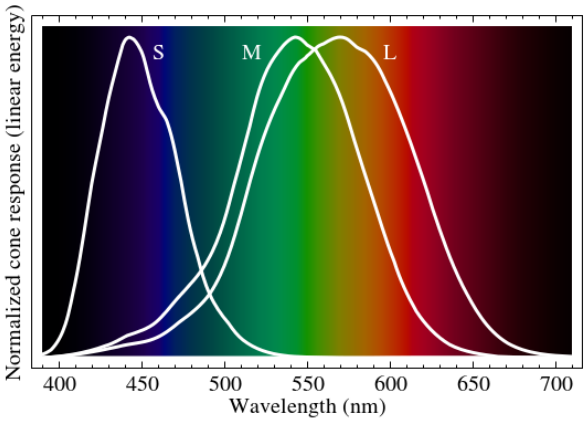



Figure 2.1.: Sensitivity Spectrum of Human Cone Cells [6]

This RGB format is often referred to as 'true color'. Other color resolutions with 16, 24 or even 48-bit per color channel are possible and result in much more precise color representation, but are not used as much, because the 8-bit resolution is sufficient in most use cases. [7]

### 2.1.2. RGBA format

The RGBA format is an extension to the RGB format. It contains an additional 'alpha' channel which holds transparency information for each pixel. An alpha value of zero represents full transparency whereas a value of 255 (assumed having a true-color picture) represents a fully opaque pixel. However, the alpha value of a pixel does not affect the color itself. The alpha channel allows images to be combined over others to create the appearance of transparency. An example of where the RGBA format is being used is the PNG file format, which is often utilized in image editing software. [8]

### 2.1.3. YUV color model

The perception of the human eye is better described by other color systems, which correspond more to the natural human sensation of color. /Instead of using values for red, green and blue, the YUV color model uses a luminance value $Y$ and two chroma components $U$ and $V$, which encode different color differences. Historically, the YUV model was developed to provide compatibility between color and black/white analog television systems. YUV is the basis for color encoding for the north American NTSC as well as for the European PAL-System. The luminance,as well as the chrominance components, can be deduced from the RGB values with :

$$Y = 0.299 * R + 0.587 * G + 0.114 * B \tag{2.1}$$

$$U = 0.492(B - Y) \tag{2.2}$$

$$V = 0.877(R - Y) \tag{2.3}$$

[9]

### 2.1.4. Grayscale conversion

In order to convert an RGB image into a grayscale image an equivalent gray value $Y$ has to be calculated for each pixel. A simple approach is to take the average value of all color components :

$$Y = \frac{R + G + B}{3} \tag{2.4}$$

However, since the human eye has a higher subjective perception of brightness of red and green color, the result is that the areas with a high proportion of red or green are too dark and areas with a high proportion of blue are too bright. That is why a weighted sum of the RGB components is used for the calculation. For analog TV signals, the gray value equals the $Y$ value in the YUV model (2.1). Also other weights are common, e.g. the weights for digital color encoding recommended by the International Telecommunication Union (ITU-BT.079) (2.5).

$$Y = 0.2125 * R + 0.7154 * G + 0.072 * B \tag{2.5}$$

[9]

## 2.2. Canny Edge Detection

Edges and contours play a considerable role in human perception and provide structural information about object boundaries. Therefore complete objects can be recognized solely by having by analyzing their edges, which results in drastically reducing the amount of data to be analyzed. An edge can be defined as a spot in a picture, where the intensity of the pixels suddenly changes in a certain direction. The intensity value equals the gray value in a grayscale image. Therefore edge detectors like the canny edge detector only properly work with grayscale images.

The strength of an intensity change is expressed by the first derivative. As we have a discrete number of pixels, we need to estimate the first derivative of the intensity function. For better understanding, we first look at a one-dimensional intensity function in the $x$ respectively $u$ direction. The derivative of the intensity at a point $u$ can be estimated by the slope of the straight line, which goes through the neighboring points $u + 1$ and $u - 1$ (2.6).

$$\frac{dI}{du}(u) \approx \frac{I(u+1) - I(u-1)}{2} \tag{2.6}$$

The derivative of the multidimensional intensity function which is defined along both axis is called a gradient, which expresses the slope in both $u$ and $v$ direction at a point ($u$,$v$).

$$\nabla I(u,v) = \begin{pmatrix} \frac{\partial I}{\partial u}(u,v) \\ \frac{\partial I}{\partial v}(u,v) \end{pmatrix} \tag{2.7}$$

The estimation of the first derivative can also be estimated by a linear filter. The commonly used Canny edge detector uses the filter of the Sobel operator (2.8). The derivative of the picture in the $x$ (or $U$) direction is calculated by the convolution of the Sobel filter $S_\mathrm{x}$ with the picture $I$. The same applies to the $y$ (or $v$) direction with the Sobel filter $S_\mathrm{y}$ (2.9). As we see, the filter extends over three columns and three rows, which results in noise reduction compared to the approach in (2.6). Furthermore, the Sobel Operator the central column and row are weighted more. The result of the 2 convolutions are now 2 new images with the values of the partial derivatives in the $x$ and $y$ direction.

$$S_\mathrm{x} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 1 & 2 \\ -1 & 0 & 1 \end{bmatrix} \qquad S_\mathrm{y} = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ -1 & 2 & 1 \end{bmatrix} \tag{2.8}$$

Subsequently, both pictures are combined into one by calculating the total intensity for each pixel. The total intensity is defined as the square root of the added square values of the filtered picture (2.10).

$$D_\mathrm{x}(u,v) = S_\mathrm{x} * I \qquad D_\mathrm{y}(u,v) = S_\mathrm{y} * I \tag{2.9}$$

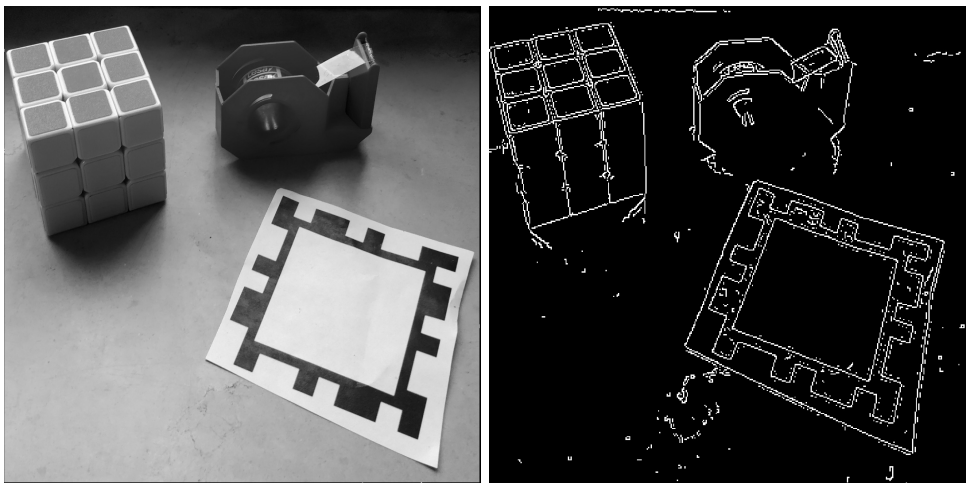
$$E(x,y) = \sqrt{(D_\mathrm{x}(u,v))^2 + (D_\mathrm{y}(u,v))^2} \tag{2.10}$$

The next step is called 'nonmaximum suppression'. For that, we move along the local edge normal direction, which can be calculated by (2.2), and compare the current pixel with the pixel next to it. If the pixel is a local maximum, then it stays untouched. If not, then the pixel is suppressed, which means its value is set to 0. This procedure results in having a thin edge which only is one pixel wide.

$$\Phi(u,v) = \tan^{-1} \frac{D_\mathrm{x}(u,v)}{(D_\mathrm{y}(u,v)} \tag{2.11}$$

The final step is hysteresis thresholding, which tries to combine edge points to a complete contour. For this we need two thresholds $T_1$ and $T_2$, where $T_1 < T_2$. First, total intensity picture is scanned for a value higher than $T_2$. These pixels are considered to be reliable edges. No we move along those edges and analyze adjacent pixels which have a likelihood being also an edge point, because of their proximity to other edge points. Every adjacent pixel, which has a higher value than $T_1$ are now marked as edge points. A rule of thumb is that the lower threshold should be about half the upper threshold. [9][10]

An example of the Canny edge detector can be seen in figure 2.2.



(a) Gray scale input image (b) Canny edge detector output image

Figure 2.2.: Applying the Canny edge detector on a gray scale image

## 2.3. Douglas-Peucker algorithm

Nearly all contours are stored with more points than it would be necessary in order to represent them. The Douglas-Peucker algorithm is a curve smoothing algorithm, which takes a curve or contour, which consists of several line segments described by a sequence of points and represents a similar curve with fewer points. Given a sequence of points ($P_1$ to $P_n$), the algorithm first draws a line between the first and last point of the sequence. If one of the other points in between is closer to the line than the distance $\epsilon$, then the point is discarded. If there are points, which are further away from the line than $\epsilon$, then the furthest point from the line is added to the new sequence of points. So now we have a curve with 3 points and 2 line segments. This process is repeated until no point is further away than $\epsilon$ from a line segment described by the kept points. The new sequence of points now represents an approximation of the original curve with fewer points. Note that the first and last point of the input sequence is always kept.[11]

## 2.4. Pinhole Camera Model

The pinhole camera model describes the relationship between the 3D world coordinates and its projection onto the image plane. The pinhole camera consists of a closed box with a tiny hole in the center at the front. Light rays coming from an object are going through the whole and projects an image onto the back of the box, which is also called the image or projective plane. The size of the image relative to the distance of the viewed object is given by the focal length $f$, which is defined as the distance between the projective plane and the hole. This relation can be seen in figure 2.3, where $Z$ is the distance to the object, $X$ the height of the object and $x$ the object's height on the projective plane. The height x of the object in the image

plane can now be calculated by :

$$-x = f\frac{X}{Z} \tag{2.12}$$

Since the projected image is upside down, the x value is negative. By putting the image plane in front of the pinhole plane at a distance $f$, we can obtain the same sized image, but now the object is right side up, which results in having a positive x value. Note that this is only a mathematical abstraction, which can not be realized with real cameras.

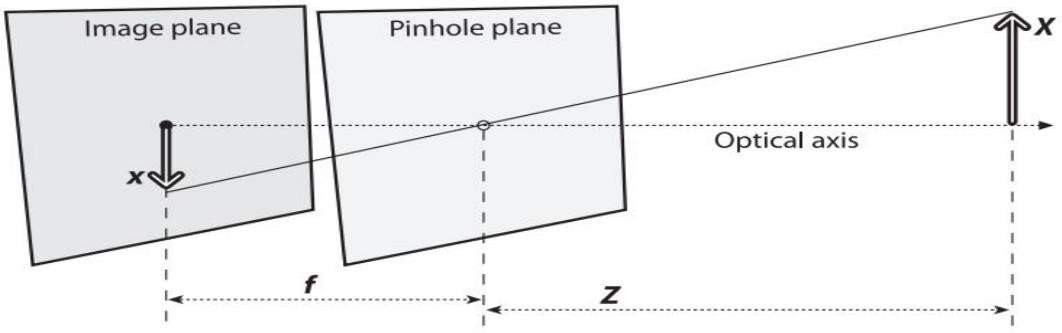



Figure 2.3.: Pinhole Camera Model
[12]

Ideally, the image plane center is equivalent to the point where the optical axis hit the image plane, which would mean that the image plane is perfectly aligned with the pinhole plane. Usually that is not the case. Therefore, we introduce two new parameters $c_x$ and $c_y$, which described this deviation in $X$ and $Y$ direction. The coordinates of the image plane are now calculated by :

$$x = f_x\left(\frac{X}{Z}\right) + c_x \qquad\qquad y = f_y\left(\frac{Y}{Z}\right)) + c_y \tag{2.13}$$

The image plane corresponds to the screen in a digital camera or a smartphone. Since the pixels on a screen are not exactly square shaped, we have to introduce two different focal lengths $f_x$ and $f_y$. The parameter $f_x$, $f_y$, $c_x$ and $c_y$ are called the camera intrinsics. By applying a perspective transformation (see section 2.6) with the camera intrinsics matrix $M$ (2.14) we can project points in the physical world into the pixel coordinate system. The camera intrinsics can be obtained by camera calibration (see section 2.8).

$$M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.14}$$

## 2.5. Extrinsic Camera Parameters

The transformation from the world coordinate space to the camera coordinate space can be realized by rotation and translation in the three-dimensional space. The position and orientation of the camera coordinate space in the world coordinate space can be expressed by a 3x1 translation vector $T$ and a 3x3 rotation matrix $R$. The rotation matrix is the product of 3 single rotations around the $X$, $Y$ and $Z$ axis (2.15). The rotation around the $X$, $Y$ and $Z$ axes with the rotation angles $\psi$, $\varphi$ and $\theta$ are defined by the three matrices $R_x(\psi)$, $R_y(\varphi)$ and $R_z(\theta)$ (2.16).

$$R = R_x(\psi)R_y(\varphi)R_z(\theta) \tag{2.15}$$

$$R_x(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos\psi & sin\psi \\ 0 & -sin\psi & cos\psi \end{bmatrix} \quad R_y(\varphi) = \begin{bmatrix} cos\varphi & 0 & -sin\varphi \\ 0 & 1 & 0 \\ 0 & sin\varphi & cos\varphi \end{bmatrix} \quad R_z(\theta) = \begin{bmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.16}$$

The translation vector $T$ describes the shift from the world coordinate system to the camera coordinate system. Therefore it can be calculated by the abstraction of the origin of the camera coordinate system from the world coordinate system (2.17). The camera coordinates $P_c$ of a point $P_w$ given in world coordinates can be calculated by (2.18). The rotation matrix and the transformation vector are referred to as the extrinsic camera parameters. [12]

$$T = origin_{world} - origin_{camera} \tag{2.17}$$

$$P_c = R(P_w - T) \tag{2.18}$$

## 2.6. Perspective Transformation

The perspective transformation, also known as projective mapping, is a "projection from one plane through a point onto another plane"[13]. This means that the perspective transformation maps the points of one plane to another plane, which, when e.g., applied to images, creates the effect of looking at the original image from another perspective. Since projective geometry deals with the projective plane, which is a superset of the real 3D plane, we are using homogeneous coordinates. So a point on a plane is now described by $(x, y, w)^T$. The homogeneous representation of a normal $(x, y, w)^T$ point is then $(xw, yw, w)^T$. So for a simple presentation of the homogeneous point, $w$ is set to 1. The projective transformation can be expressed by multiplying a point on a plane with a three by three matrix. Note that the

resulting vector is scaled, which means in order to get the point coordinates on the other plane the $x$ and $y$ values have to be divided by $w$.

$$\begin{bmatrix} x' \\ y' \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{2.19}$$

Hence there are only 8 degrees of freedom in a 2-D projective transformation, it can be assumed that $i = 1$. The other 8 unknowns of the transformation matrix can be determined from 4 points $(x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)$ with their 4 corresponding points $(u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3)$ on the other plane. The eight equations resulting from (2.19) can be written as :

$$\begin{bmatrix} u_o & v_0 & 1 & 0 & 0 & 0 & -u_0x_0 & -v_0x_0 \\ u_1 & v_1 & 1 & 0 & 0 & 0 & -u_1x_1 & -v_1x_1 \\ u_2 & v_2 & 1 & 0 & 0 & 0 & -u_2x_2 & -v_2x_2 \\ u_3 & v_3 & 1 & 0 & 0 & 0 & -u_3x_3 & -v_3x_3 \\ 0 & 0 & 0 & u_o & v_0 & 1 & -u_0y_0 & -v_0y_0 \\ 0 & 0 & 0 & u_1 & v_1 & 1 & -u_1y_1 & -v_1y_1 \\ 0 & 0 & 0 & u_2 & v_2 & 1 & -u_2y_2 & -v_2y_2 \\ 0 & 0 & 0 & u_3 & v_3 & 1 & -u_3y_3 & -v_3y_3 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} \tag{2.20}$$

Solving this equation system for the missing 8 unknowns can be done by using Gaussian elimination. After obtaining the values for the transformation matrix, every point on one plane can be transformed to a point on the other plane using the equation 2.19. [14]

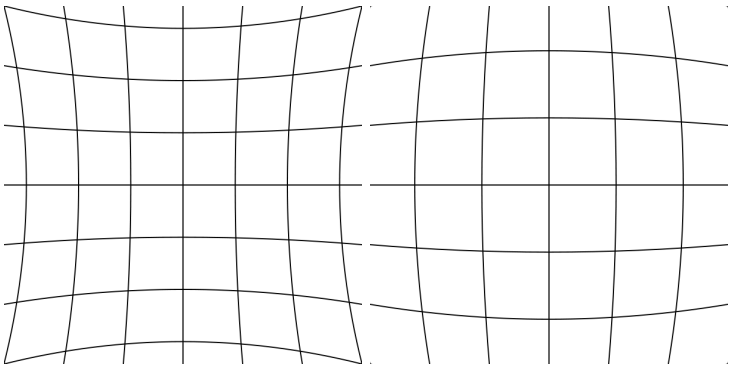
## 2.7. Camera Distortion

Cameras are typically using lenses in front of their image sensor. A lens allows the camera to gather a lot of light over a wide area and focuses them on creating an image. This is what makes a difference between today's cameras and the simple pinhole camera model. With the introduction of lenses, we also introduce distortions. There are two main lens distortions.

### 2.7.1. Radial Distortions

Radial distortions are caused by the shape of the lens. The lens causes that the magnification is not equal to every image point. Image points that are further away from the center are magnified more, resulting in bent lines that are at the margin of the image. This effect is called pincushion distortion and can be seen in figure 2.4a. The opposite effect can be seen in figure 2.4b, which shows the barrel distortion, caused by a decreasing magnification towards the margin of the image. This effect occurs particularly strong e.g., with wide-angle lenses.

The distortion can be characterized by a Taylor Polynomial with the coefficients $k_1$, $k_2$ and $k_3$. Since the degree of magnification depends on the distance of the picture point to the



(a) Pincushion Distortion [15] (b) Barrel Distortion [16]

Figure 2.4.: Radial Distortion

center of the image, the correction is calculated by using the radius $r$ defined in (2.21). The distortion can be corrected by recalculating the location of each picture point by using the equation (2.22). The coefficients $k_1$, $k_2$, $k_3$ can be obtained by camera calibration (see section 2.8). [12]

$$r = \sqrt{x^2 + y^2} \tag{2.21}$$

$$\begin{aligned} x_{corrected} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6), \\ y_{corrected} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{aligned} \tag{2.22}$$

### 2.7.2. Tangential Distortion

Tangential distortion occurs when the lens is not perfectly parallel to the image sensor. This type of distortion can be corrected by (2.23):

$$\begin{aligned} x_{corrected} &= x + [2p_1 xy + p_2(r^2 + 2x^2)], \\ y_{corrected} &= y + [p_1(r^2 + 2y^2) + 2p_2 xy] \end{aligned} \tag{2.23}$$

The two additional parameters $p_1$ and $p_2$ can also be obtained by camera calibration (see section 2.8). [12]

## 2.8. Camera Calibration

The goal of the camera calibration process is the determination of the intrinsic camera parameters as well as the distortion coefficients. For this process, a calibration object with known feature points is needed. The calibration object used in OpenCV is a flat chessboard

with alternating black and white squares. The grid corners are then used as reference points. In general, when the lens distortion is not considered, transforming a point given in world coordinates into pixel coordinates is given by multiplying the camera intrinsics matrix and the rotation/translation vectors with the point's coordinates :

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} \tag{2.24}$$

Now we take multiple images of the chessboard from different directions and use these images to compute both the individual translation and rotation vectors for each view and the camera intrinsics, which are the same for all views. So we have 6 unknowns extrinsic parameters, 3 angles for the 3 rotations and 3 values for the translation. When mapping the object plane into the image plane, we get 8 equations, since for this transformation we are using 4 points, which have each a value for x and y. So we got more equations than extrinsic unknowns , and therefore we are able to compute any number of intrinsic unknowns if we are taking enough images. In order to get precise results, a total number of minimum 10 images of a 7-by-8 chessboard is needed to have enough points and equations.

As we are using a flat chessboard, we can define the $Z$ value of the object plane to be 0. The new homography $H$ matrix which maps the planar object point onto the image plane is described by :

$$H = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{11} & r_{12} & t_y \\ r_{11} & r_{12} & t_z \end{bmatrix} \tag{2.25}$$

H can be written as three column vectors $H = [h_1\ h_2\ h_3]$. Since $r_1$ and $r_2$ are orthonormal it implies that $r_1^T r_2 = 0$. We can rewrite this equation with :

$$h_1^T M^{-T} M^{-1} h_2 = 0 \tag{2.26}$$

The magnitude of the rotation vectors are also equal ($r_1^T = r_2^T r_2$) which leads to the second constraint:

$$h_1^T M^{-T} M^{-1} h_1 = h_2^T M^{-T} M^{-1} h_2 \tag{2.27}$$

A matrix B is now defined as:

$$B = M{-}TM{-}1 = \begin{bmatrix} \frac{1}{f_x^2} & 0 & \frac{-c_x}{f_x^2} \\ 0 & \frac{1}{f_y^2} & \frac{-c_x}{f_y^2} \\ \frac{-c_x}{f_x^2} & \frac{-c_y}{f_y^2} & \frac{c_x^2}{f_x^2} + \frac{c_y^2}{f_y^2} + 1 \end{bmatrix} \tag{2.28}$$

The two constraints can now be rewritten as :

$$h_i^T B h_j = v_{ij}^T b = \begin{bmatrix} h_{i1}h_{j1} \\ h_{i1}h_{j2} + h_{i2}h_{j1} \\ h_{i2}h_{j2} \\ h_{i3}h_{j1} + h_{i1}h_{j3} \\ h_{i3}h_{j2} + h_{i2}h_{j3} \\ h_{i3}h_{i3} \end{bmatrix}^T \begin{bmatrix} B_{11} \\ B_{12} \\ B_{22} \\ B_{13} \\ B_{23} \\ B_{33} \end{bmatrix}^T \tag{2.29}$$

The first constraint is then written as :

$$\begin{bmatrix} v_{12}^T \\ (v_{11} - v_{22})^2 \end{bmatrix} b = 0 \tag{2.30}$$

By collecting $\geq$ images, this equation system can be solved with a unique solution. The intrinsic camera parameters can be obtained by the values of $B$:

$$f_x = \sqrt{\lambda / B_{11}} \tag{2.31}$$

$$f_y = \sqrt{\lambda B_{11} / (B_{11}B_{22} - B_{12}^2)} \tag{2.32}$$

$$c_x = -B_{13} f_x^2 / \lambda \tag{2.33}$$

$$c_y = (B_{12}B_{13} - B_{11}B_{23}) / (B_{11}B_{22} - B_{12}^2) \tag{2.34}$$

$$\lambda = B_{33} - (B_{13}^2 + c_y(B_{12}B_{13} - B_{11}B_{23})) / B_{11} \tag{2.35}$$

The rotation and translation vectors can then be calculated by solving the equation system (2.29) with the second constraint (2.27). The vectors are obtained by :

$$r_1 = \lambda M^{-}1 h_1 \tag{2.36}$$

$$r_2 = \lambda M^{-1} h_2 \tag{2.37}$$

$$r_3 = r_1 \times r_2 \tag{2.38}$$

$$t = \lambda M^{-1} h_3 \tag{2.39}$$

$$\lambda = 1 / \| M^{-1} h_1 \| \tag{2.40}$$

So far, we have not considered any distortion. The coordinates of an undistorted point ($x_u$,$y_u$) are calculated by combining the radial distortion correction and the tangential distortion correction from section 2.7 :

$$\begin{bmatrix} x_u \\ y_u \end{bmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{bmatrix} x_d \\ y_d \end{bmatrix} + \begin{bmatrix} 2p_1 x_d y_d + p_2(r^2 + 2x_d^2) \\ p_1(r^2 + 2y_d^2) + 2p_2 x_d y_d \end{bmatrix} \tag{2.41}$$

After estimating the intrinsic parameter without considering distortion, these estimations are kept. With multiple equations given by different points in different images, we can estimate the distortion coefficients. After that, the intrinsic and extrinsic parameters can be reestimated. The procedure of estimating the distortion coefficients via the intrinsic/extrinsic parameters and vice versa is done until convergence.[17] [12]

# 3. Related Work

## 3.1. ARBlocks

The idea of using a frame marker for tracking is not new. The Virtual Reality and Multimedia Research Group of the Federal University of Pernambuco in Recife, Brazil used frame markers for their application called ARBlocks in 2011. ARBlocks is a concept of a dynamic blocks platform for educational activities. It is based on projective augmented reality and tangible user interfaces. Their application setup was that there are a projector and a camera one meter above the table, pointing towards its surface. The camera tracks the position of the markers, which are located on the top of the tangible blocks. Now the projector projects a picture directly onto the top of the block, depending on the code embedded on the marker. Their aim was to use this platform as a support tool on early childhood literacy and therefore created a game where letters a projected onto the tangible blocks. One of their design questions was, which type of marker to use. They finally decided to use frame markers, since it is the only marker type next to split markers, which allows for tracking a big number of multiple markers simultaneously and still having a blank spot to display information. However, it does not get clear why the idea of using split markers is discarded since they also have a free spot in the middle for displaying pictures .[18]

## 3.2. Studierstube Tracker

Wagner, Langlotz ,and Schmalstieg from the Graz University of Technology developed their own marker tracking library called 'Studierstube Tracker' for applications on mobile phones. Their aim was to develop different marker types, that occupy significantly less space and therefore are much more unobtrusive. So, in 2008, they introduced 3 new marker types for their library including dot markers, split markers ,and also frame markers. The markers were tested against each other in tracking time on a Windows mobile smartphone (Asus M530W). They came to the conclusion that the shape detection of split markers is slightly slower than for frame markers due to their more complex shape, but on the other hand, split markers take more time for marker detection since each marker consists of two parts. This leads to the result that split markers take about 6.5ms to be tracked and frame marker about 4.8ms. Dot markers need even more time (9.0ms), because they require much time in filtering out noncircular structures. That is why the frame marker the clear winner in terms of tracking time. Unfortunately, there is no information given about the difference in the performance of the 3 marker types concerning different light conditions, distances, and angles. [19]

## 3.3. Differences in Implementation

For the implementation of the frame marker, the ARBlocks team used a sequence of 10 black and white squares, each square representing 1 bit. This code is the same on every edge for adding redundancy, leading to a more robust identification. The first and the last bit on every edge differ, in order to detect the correct orientation of the marker. Therefore, it is possible to distinguish between 256 different markers. For the actual detection process, the camera image is converted to grayscale. The image is then handed over to the Canny edge detector which detects the edges in the picture. After that, all squares with a reasonable size are sorted out. In the end the code is read and compared with their predefined markers.[18] The developer of the Studierstube Tracker used a 9-bit long code, also encoded in black and white squares, where each square represents 1-bit. The difference is here that they use 27-bit redundancy leading to a 36-bit code which is printed on the marker in clockwise order. The code is chosen in a way, that only one corner of the marker makes sense as a starting point of code and the others result in correction errors. With this technique, they save the place for 4 additional bit placed in the corners for orientation like in the ARBlocks example. Unfortunately, neither the concrete error correction algorithm nor the further detection process is described. [19] What these two frame marker implementations have in common is that on the one hand they have their code encoded in black and white squares, use some redundancy for error correction and also have a continuous thin line wrapped around the marker. This line helps to recognize the marker as a square after the Canny edge detector was used.

# 4. Concept

The goal of this thesis is to create an application on a smartphone, which is able to detect a frame marker. The design of the frame marker should be chosen so that it fits onto the frame of an electrical socket or respectively onto a light switch frame. The encoded code on the marker itself should be big enough so that you can differentiate between 256 different markers, which should be enough for this architectural use case. Another requirement for the design is that the marker should be still detectable even if one of the 4 edges is hidden or obscured by other things.

The general approach of the implementation is first to take the RGB picture taken by the smartphone's camera and convert it into a grayscale picture. After that, we analyze the picture for edges and sort those contours out, which describe a quadrangle. These contours are marked as possible frame marker candidates. Now for each candidate, we analyzed the encoding and performed some error correction on it. If the code matches the code we are looking for, that means that we have found a frame marker. After detecting the marker, we perform pose estimation in order to get the position of the marker in the room. In the last step, the pose is visualized via a drawn axis, and the extracted code is also displayed. Figure 4.1 gives a schematic overview of the detection procedure. At first, a frame marker printed out on paper is used as a testing object in the implementation phase. After that, the marker is glued onto a real socket frame and is being tested regarding the deviation of the calculated pose, maximum detection distance, and maximum viewing angle.

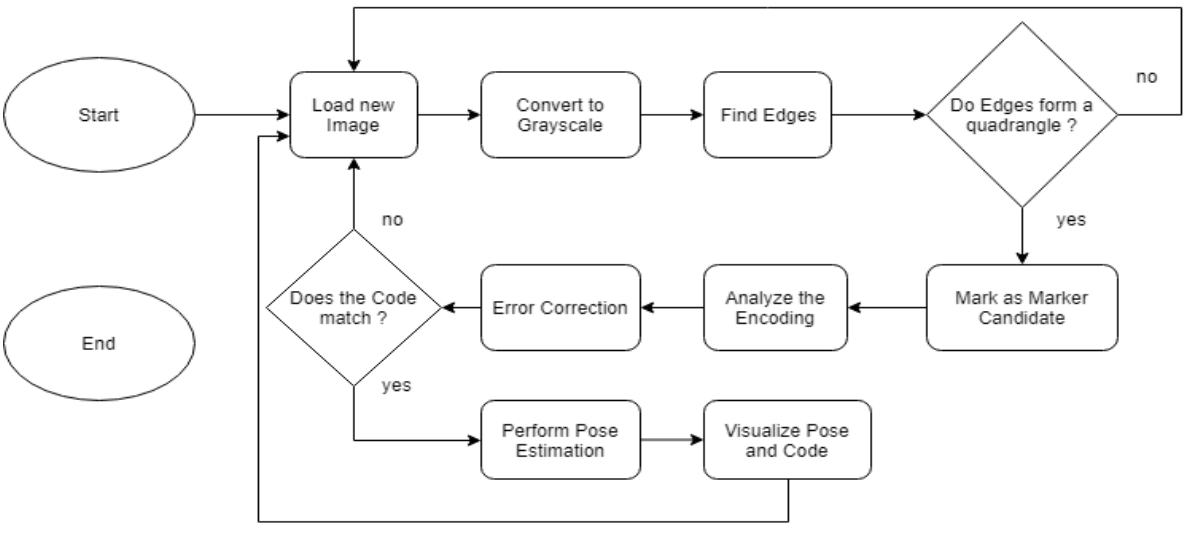



Figure 4.1.: Flow chart of the general detection procedure

# 5. Implementation

The computer vision library OpenCV is used for this implementation. The code is written in Java and tested on a OnePlus 5t smartphone, which is running on Android 9.0. For further information about the used software and hardware, see appendix.

## 5.1. Marker Type

For this project, a so called frame marker was used. A frame marker is a square marker, which has a cutout in the middle so that the code is only on the edges of the marker. Since the inside of the marker has no information, this part of the marker can be ignored and therefore, it can be easily put around objects. This makes a frame perfect for putting onto electrical sockets an light switches. The marker can be placed around them ,and the light switches and sockets are still fully usable. Furthermore, the marker is much less obtrusive and is rather perceived as a sort of decoration than a marker. On each edge of the marker is the same code, in the form of 8 individual black or white squares encoding 1 bit each. Additionally, there are 3 corners marked with a black square and 1 corner marked with a white square, which will be used for determination of the orientation.

## 5.2. Different approaches

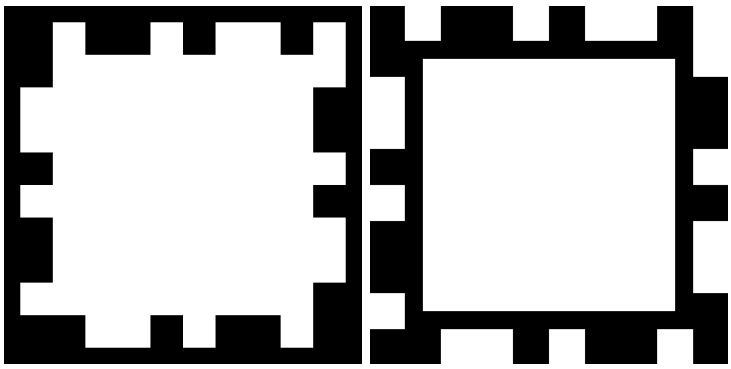
### 5.2.1. Outside Edge approach

The first approach was to design the marker as described in the previous section, with a thin black line going around the edges of the marker. This thin line can then be recognized as a square after the edge detection. So first we are looking for the square and then analyze the inside lying code of the marker. As we will see in the evaluation chapter 6 this approach caused some problems.

### 5.2.2. Inside Edge approach

The second approach differs from the first approach,such that the thin black line is going around the inner edge of the marker. So we have to look at the outer of the square to analyze the code. This approach gave much better results in terms of detection quality when this marker is attached to a socket frame, which is further described in chapter 6. Figure 5.1 shows the two different marker designs.

For now on, only the implementation of the second approach will be described.



(a) Outside Square Approach (b) Inside Square Approach

Figure 5.1.: Different Marker Designs

## 5.3. Loading the Image

For loading the image, OpenCV provides the class `CameraBridgeViewBase`. It implements the interaction with the smartphone camera and the library. Its main purpose is to control when the camera can be enabled, process the frame, call (in our case) the listener `CvcameraViewListener2` and draw the frame to the screen. The main method of `CvcameraViewListener2` we are working on is `onCameraFrame()`. Everytime a new frame is needed, this function is called. Therefore it represents our main loop. At the first the image is loaded in RGBA format (see section 2.1.2) and stored as a 'Mat'. Mat is an n-dimensional dense numerical single or multi-channel array which can be used to store e.g. color or grayscale images, matrices and vectors[20]. The specified data type for storing the pixels of the image is 'CV_8UC4' which means that we use unsigned char types with a length of 8 bit. The digit a the end defines the number of channels that are used. In our case there are 4, the red, green, blue and alpha channel. Therefore it is possible to display $(2^8)^3$ different colors.

OpenCV offers a wide variety of color conversion methods. With `cvtColor (src, dst, COLOR_RGBA2GRAY)` the image is subsequently converted into a grayscale image. Since the grayscale image has only one Y channel for the luminance (see section 2.1.3) which is calculated by a weighted sum of the R,G and B channel (5.1), which is exactly the same as the $Y$ component in the YUV color model (2.1).

$$RGB[A]toGray : Y = 0.299 * R + 0.587 * G + 0.114 * B \tag{5.1}$$

## 5.4. Marker Detection

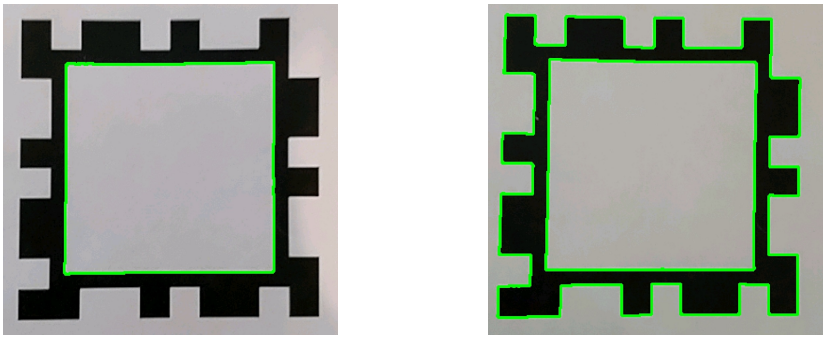
The very first step after obtaining a grayscale image is to find marker candidates. Since the frame marker is a square, the image is analyzed in order to find rectangle shapes. To do so, we first need to detect all edges. Therefore the canny edge detector is applied. In OpenCV, the

implementation already exists with `Canny(src, dst, thresh1, thresh2)`, whereas thresh1 and thresh2 are the max and min value for the Hysteresis thresholding (see section 2.2).The values 100 for thresh1 and 80 for thresh2 deliver good results indoors under daylight conditions.

Afterwards, the image, which now contains only edges, is scanned for contours. A contour is a curve joining all the points having the same intensity, which makes it great for shape analysis. That means that all edges which are connected are now described by a single curve. In order to find these contours the method `findContours(src Canny, contours, hierarchy, RETR_TREE, CHAIN_APPROX_SIMPLE, new Point(0, 0))` which uses the algorithm from S. Suzuki [20], is applied. The method stores the detected contours as a vector of 2D points. Depending on the contour approximation method, different points of the curve are stored. In this case, the simple approximation is used which compresses horizontal, vertical, and diagonal segments and leaves only their endpoints, so that a contour in the shape of a rectangle is encoded by only storing the corner points. Another possibility would be to store all the points of the contour (non approximation), which would be completely unnecessary, because you have to store many more points per rectangle, and since we only want to detect rectangular shapes the simple approximation is sufficient. As there are shapes and contours which are inside of other contours, therefore it exists a certain hierarchy, where the inner contour is the child and the outer contour the parent. Depending on the contour retrieval mode flag, the contours are store differently. With `RETR_TREE` they are stored in a tree-like structure, whereas e.g `RETR_EXTERNAL` only the outermost contours are stored. Since we do not need this extra bit of information for the marker detection, we can ignore it. The last parameter describes an optional offset by which every contour point is shifted. Since we also do not need that, the offset is set to 0,0.

For even better results, each found contour is approximated by another curve with fewer vertices. That means that e.g., the contour of a square which has a slight kink on one of the sides which is normally described by a 5 point contour, will be approximated by a 4 point contour. The OpenCV method `approxPolyDP(curve, approxCurve, epsilon, closed)` uses the Douglas-Peucker algorithm (see section 2.3) for this approximation. After that, all closed contours with 4 points/corners are sorted out so that we only receive quadrangles. The challenge here is to find a fitting value for $\epsilon$, which describes the maximum distance between the original curve and the approximation. A good value for $\epsilon$ was found by testing which contour was approximated by only 4 points. As you can see in figure 5.2a a very small epsilon of 1% deviation of the original curve length lead to the desired result, that only the square in the middle of the marker is accepted. Whereas as you can see in figure 5.2b only a slightly higher deviation of 2% leads to the consequence that also the contour of the whole code is approximated by a 4 point curve. So in general to obtain the inner square of the marker, we can only apply a very accurate approximation with a very small $\epsilon$. Finally the 4 corner points are sorted in clockwise order, starting from the top right.

After the marker candidates are found, we need to analyze whether they are real markers or just some random squares in the picture. But in order to properly analyze the marker the perspective view has to be removed so that the marker is then displayed in its 2D canonical form.



(a) Contour with max. 1% difference in length from the approximation

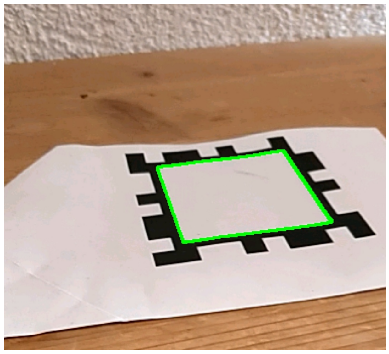
(b) Contour with max. 2% difference in length from the approximation

Figure 5.2.: Effects of different parameter Values on the approximation accuracy

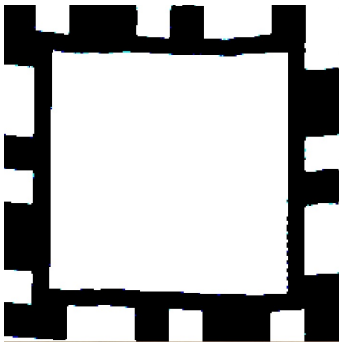
The method `removePerspectiveView (inputImg, outputImg,points,offset)` takes the 4 corner points,calculates a $3x3$ transformation matrix $M$ applies it on the image and scales it to a size of $250x250$ pixel image, which only contains the transformed marker. The matrix $M$ is calculated by the OpenCV method `getPerspectiveTransform(points,corresPoints)` which takes 4 points from the original image and 4 corresponding points from the output image for the calculation (see section 2.6). In our case the corresponding points of the output image are: (0,0), (250,0), (250,250) and (0,250) since these are the corner points of the new '2D' picture. `removePerspectiveView()` also makes use of the OpenCV method `warpPerspective(img,M,size)` which applies the previously calculated matrix $M$ on the picture and scales it to a certain size. The problem now is that corner points of the inner rectangle of the marker are taken for this perspective transformation, therefore these corner points are the new corner points in the '2D' image, which results in cutting off the code. The first idea was to scale the corner points of the image before applying the transformation, so that the transformation is done for a bigger square. But since the marker is not always perpendicular to the camera, the square would not perfectly fit around the code. So we first calculate the $M$ matrix for the inner square. But the before applying the transformation on the image we take the corresponding points in the '2D' image, scale them by an offset which resembles the thickness of the code rim and transform them back into the original image by applying the transformation with the inverse matrix of $M$. So now we can do the perspective transformation like previously but with right scaled 4 points, which exactly represent the outer corner points of the marker. Therefore the '2D' image also contains the marker code.

After that the '2D' image of the marker is first converted into a gray scale image like in section 5.3 and then converted into a black and white image. The conversion is done by a binary thresholding operation. If the pixel value of the gray image is lower than a certain threshold, then it is set to 0 which represents black, otherwise it is set to a maximum value which in our case is 255 which represents white. For the threshold 127 is chosen, because it is in the middle of 0 and 255. But also threshold between 100 and 150 achieved good results, as the marker is rich in contrast.An example of this whole transformation and thresholding

process can be seen in figure 5.3, where figure 5.3a represent the image which was taken by the camera and figure 5.3b which represents the image obtained after the perspective transformation and subsequent black and white conversion.



(a) Marker in perspective



(b) 2D marker image after perspective transformation

Figure 5.3.: Applying perspective transformation on the image

## 5.5. Marker Identification

### 5.5.1. Bits extraction

The marker code itself can now be finally extracted, but for that an extra marker object is generated. The marker object is constructed by taking the '2D' black and white marker image, the code length of the marker code and an additional offset. Each edge of the marker is then analyzed separately. Each edge is divided into different cells, where each cell is the size of one bit of the code on the picture. The cell size is determined by the code length (note that the corner bits also count towards the code length). It is expected that every single bit is square-shaped. Therefore the cell size is calculated by *image width/code length*. Now in every cell, the number of black and white pixel is counted. The majority of black/white pixels then determines whether the bit is black or white. Since we know the number of pixels for each bit, only the non-zero (white) pixels are counted for faster execution. But that also makes it possible to construct a marker object with a black and white image which is encoded with 1 (for white) and 0 (for black). Every cell on one edge is analyzed and then stored as 1 (for white) and 0 (for black) in a 1-dimensional integer array. This is done for each of the 4 edges which are stored as `topCode, bottomCode, leftCode and rightCode`. In addition, the 'raw' numbers of white pixels on each cell are also kept. The offset parameter defines the width (in pixels) of the image rim which is cut off and ignored for the bit extraction. This parameter is necessary for the first approach (see section 5.2.1), in order to compensate the width of the black outside square, so that these pixels are not counted towards the pixels of each bit. Since the final implementation way has no black square border on the outside, this offset is set to 0.

### 5.5.2. Error correction

Since not every bit is always detected correctly, due to different lighting conditions, no perfect image transformation, different thresholds, offsets ,etc., error correction has to be applied to the code. Because of the fact that the code is the same on all 4 edges, the 4 edges are compared bitwise and the majority of 1 or 0 finally determines the code. Therefore even if the code on one edge is detected completely wrong or is obscured, the code can be still extracted correctly (assuming all the other 3 edges are detected correctly). The error correction is also applied on the corner bit, which is only used for the orientation. As 3 of the 4 corner points are black, the first and last bit of the code is stored as 0. This part can be ignored since the orientation bit is looked at separately. Finally, the code is stored as `finalCode` in a one-dimensional integer array.

### 5.5.3. Orientation

The orientation, as already described, is only encoded in the corner bits. The only white corner is set as the top right corner in the orientation. So we have to only search for the white corner and return its position. The problem with that is that this only works if all the other corners are all detected as black. Therefore only if all other corners are black, the position is returned, otherwise, the 'most' white looking corner is returned. As we also stored the 'raw' numbers of white pixels on each cell (see section 5.5.1) the position of the corner cell with the most white pixels is returned. The position of the white corner bit is encoded as seen in the table 5.1.

| | Top Right | Bottom Right | Bottom Left | Top Left |
|---|---|---|---|---|
| Value | 0 | 1 | 2 | 3 |

Table 5.1.: White corner orientation encoding

## 5.6. Marker Pose Estimation

In order to estimate the pose of the marker, we need to once again calculate a transformation matrix, since the world coordinates of the marker are different from the camera coordinates. But unlike in section 5.4 also the intrinsic and extrinsic parameters of the camera, as well as the radial and tangential distortion is taken into account, for getting a more precise result. The radial distortion can be corrected by applying the formula 2.22 on each pixels x and y coordinates. For the tangential distortion, the formula 2.23 is used. The five distortion parameters are represented by the $Distortion\ Coefficients$ matrix 5.2. For removing the additional distortion due to the specific lens on the used camera, we need to know the intrinsic parameters of the camera. The intrinsics parameters include the focal length $(f_x, f_y)$ and the optical center $(c_x, c_y)$ and are stored in a 3x3 matrix called *Camera Matrix* 5.3. Since the *Camera matrix* is always the same for a specific camera it can be calculated in advance. The same applies to the

*Distortion Coefficients*, as long as the resolution has not changed. The *Camera Matrix* as well as the *Distortion Coefficients* can be obtained by calibration.[20] The calibration is done externally with the app 'Vizario' which uses a black and white chessboard for the calibration process (see section 2.8) [21]. The concrete values for the used smartphone camera can be seen in 5.4 and 5.5. The OpenCV method `solvePnP(ojbPoints, imagePoints, cameraMatrix, distCoeffs, rvec, tvec)` now estimates the object pose given a set of object points, their corresponding image projections, as well as the camera matrix and the distortion coefficients. The estimated pose is now stored in contrast to section 5.4 in separate rotation (`rvec`) and translation (`tvec`) vectors. The object points are chosen so that the center of the marker represents the origin of the coordinate system and the corner points are at the world coordinates: (-m,-m,0),(m,-m,0),(m,m,0) and (-m,m,0) where m represents half of the marker size in meters. The Z-Coordinate is set to 0, since the origin of the coordinate system is defined as the center of the marker. The object points are sorted according to the marker orientation and returned by `getOrientedObjectPoints(objPointsMat, markerOrientation)`.

$$Distortion\ Coefficients = (k_1\ k_2\ p_1\ p_2\ k_3) \tag{5.2}$$

$$Camera\ Matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5.3}$$

$$Camera\ Matrix\ of\ used\ camera = \begin{bmatrix} 516 & 0 & 319 \\ 0 & 515 & 238 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.4}$$

$$Distortion\ Coefficients\ of\ used\ camera = (0.3495\ \ -2.2549\ 0.0\ 0.0\ 4.9959) \tag{5.5}$$

## 5.7. Visualization

### 5.7.1. Contour, corners, 2D marker and code

As in section 5.5 the contour of a marker candidate is drawn in green color by the OpenCV method `drawContours(image, contours, contourIdx, color, thickness)`, which takes the image, all found contours, color scalar in RGB format and thickness given in pixels as input values and draws the contour outlines directly onto the image. The contour index parameter describes which contour of all found contours is drawn. Since we are iterating over all found contours are drawn, which can be e.g useful for finding the right $\epsilon$ parameter (see section 5.5), unless we check if this contour belongs to a marker beforehand.

The corner points are visualized with the OpenCV method `putText(image, text, point, font, fontScale, color, thickness)` which again takes a text (in this case

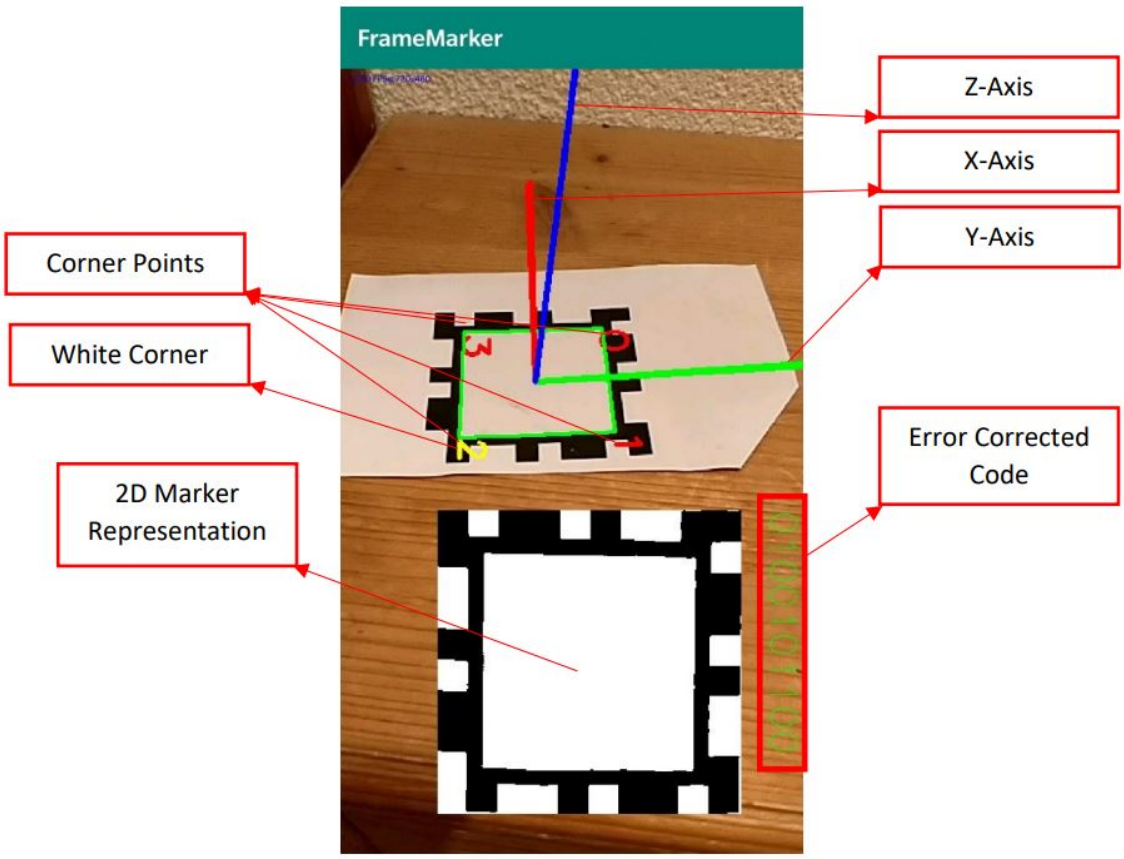



Figure 5.4.: Visualization of marker information

the numbers 1 to 4), a font, a font scale, a color scalar in RGB format and thickness given in pixels and a point in the image where the text is written, which is in our case the four corner points. The corner points are labeled in red clockwise starting from the top right, as the corner points are sorted this way (see section 5.4). One exception is the white corner point, which is labeled in yellow since it is important for the orientation detection.

For better debugging and evaluation, the '2D' marker image from section 5.5 is also copied into the final output image. The provided method `copyTo(image, mask)` is used for that, which takes the image matrix and copies the values into another image matrix, where mask defines an operation mask, which indicates what elements are being copied.

Next to the '2D' representation of the marker, the error corrected code of the marker is displayed in green. Note that the corner bits are also displayed.

### 5.7.2. Axis

For visualizing the X, Y ,and Z-Axis as it can be seen in figure 5.4, the endpoints of the axes are transformed from the world coordinates into pixel coordinates by using the calculated pose from section 5.6 via the OpenCV function `projectPoints(objPoints, rvec, tvec, cameraMatrix, distCoeffs, imagePoints)` which performs the following equation 2.24, where $r_{**}$ and $t_*$ represent the rotation and translation vectors, $X_w$,$Y_w$,$Z_w$ the world coordinates $u$,$v$ the corresponding x and y pixel coordinates.[20]

After projecting the axis endpoints into the pixel domain, a line from each endpoint is drawn to the coordinate origin for visualizing the X (in red), Y (in green) and Z-axis (in blue) (see figure 5.4).

# 6. Evaluation

## 6.1. Using the paper marker on a socket frame

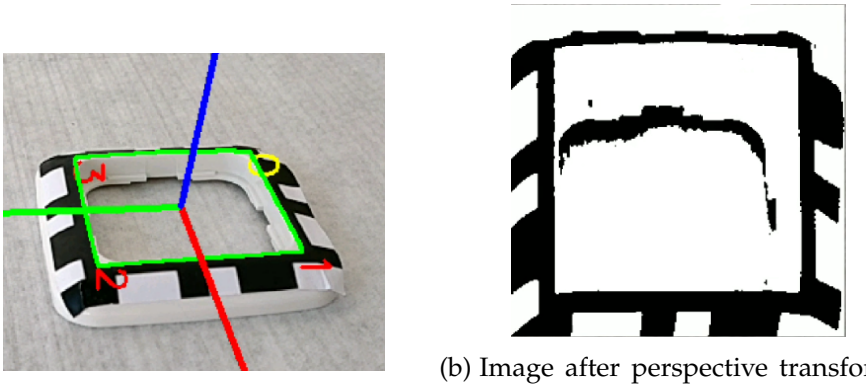
First, we have a look at the first implementation approach from section 5.2. For testing the marker in the implementation process, it was first printed out on white paper. The paper version works just fine, and the marker can be detected correctly with the right code. Then the marker was cut out from the paper and glued onto the socket frame. The problem now was that the marker was no longer recognized correctly and even sometimes not considered as a marker candidate. The socket marker is not perfectly flat. It has rounded edges and corners. When gluing the paper marker onto the three-dimensional frame, the shape of the marker changes. So the black line going around the marker was no longer perfectly squared, with its rounded corners. But the biggest problem is, when you are looking at a slight angle onto the socket frame, one or more black edges disappear behind the curvature of the frame edges. Therefore the marker is no longer considered as a candidate since one or more edges that built up the square for which we are looking for, after the edge detection process, are missing. So the first approach failed on the socket marker, although it worked on paper.

The second approach solved this issue by moving the black edge of the marker inwards. This has the advantage that the black edge no longer disappears behind the rounded edge and therefore can be recognized as a square.

However, both approaches work equally good on a flat surface. The issue with the disappearing edge could also be solved by using another type of socket frame, which is completely flat. But these types are not as common in Europe, as the rounded ones. Extracting the code form the inside edge approach is a bit more difficult. As explained in the Marker Detection section 5.4, there has to be an additional step, where the outside corner points of the marker are calculated, before doing the perspective transformation.

By transferring the paper marker onto the socket frame another problem occurs, while performing the perspective removal. Since the edges are rounded, the edges of the single encoded bits are also rounded. When we perform the perspective removal, the normally horizontal edges are now tilted, which can be seen in figure 6.1b. Therefore the virtual grid we are lying onto the '2D' image for counting the pixels, is no longer aligned with the encoded bits. This may result in extracting the wrong code if the aligned is way too off. The steeper the viewing angle, the higher is the error of the extracted code. A possible compensation for that is described in chapter 8.

During the testing, the application had about an average frame rate of 15 frames per second (FPS).



(a) Socket Marker in perspective

(b) Image after perspective transformation

Figure 6.1.: Applying perspective transformation on socket frame image

## 6.2. Measurements

### 6.2.1. Measurement Setup

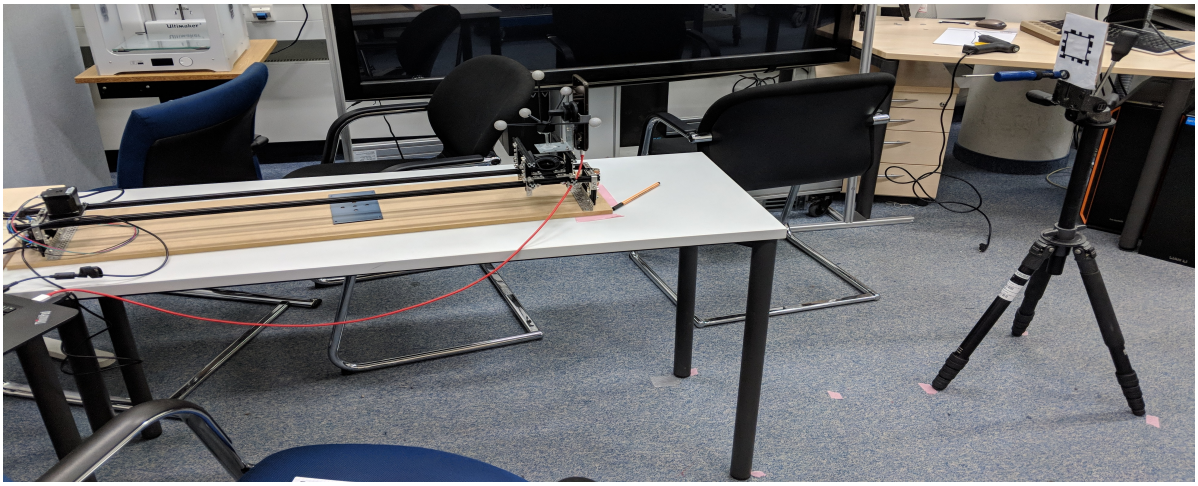
For measuring the distance and angles, at which the marker can still be detected, we use a slider rail, where the smartphone sits fixed in a mount. The smartphone can be moved on the rail via a stepper motor, which allows us to move it precisely to any location on the rail. On the opposite side, there is the marker mounted on a camera tripod. The center of the marker and the smartphone's camera lie on the same line and are at the same height so that the optical axis is parallel to the ground. All measurements were taken indoors under daylight conditions. Figure 6.2 shows the measuring setup for better understanding.

### 6.2.2. Distance Measurement

For each measurement, the tripod with the marker stays at a fixed location, and only the smartphone is being moved by the stepper motor. First, we measure the maximum distance at which the paper marker is still detected correctly. Initially, the marker is placed at a distance of 95 cm away from the rail. At this distance, the detection and code extraction worked really well. So by continuously increasing the distance via the stepper motor, the maximum distance at which the marker was still detectable, was at about 10000 steps of the stepper motor away from the initial point, which corresponds to a distance of about 120 cm. At a distance of 121 cm (500 additional steps) there was bare to no detection and above that, the marker clearly can not be detected anymore.

The same measurements were performed with the socket marker. The socket marker was not even detected at the initial 95 cm away from the smartphone, therefore the tripod was

(a) Slider rail with stepper motor and mounted smartphone at the front



(b) Slider rail (left) with marker mounted on a camera tripod (right)

Figure 6.2.: Overview of measuring setup

moved closer at a distance of 72 cm. The maximum distance at which the socket marker was still detectable was at a distance of about 81 cm, which is significantly worse than the paper marker. The reason for that is the rounding edges of the socket frame, which makes it much harder to detect the edges of the marker and extract the code correctly.

### 6.2.3. Viewing Angle Stability

Additionally, the viewing angle stability is measured, by using the same setup as previously, but this time the smartphone stays at a fixed location on the rail and the head of the tripod with the marker is rotated by a certain degree. At a distance of 95 cm, the paper marker version could be detected to about a viewing angle up to 45°, whereas a stable detection was possible up to 30°. By reducing the distance to 50 cm, the marker was still detectable even up to 60°.

The socket frame marker version performed significantly worse as the paper version, which once again is because of the rounding of the edges. Although the inside edge approach is used, the edge still disappears behind the curvature of the frame, because the inner edge is still not perfectly flat. That is why at a distance of 72 cm the maximum possible viewing is at 30°. Even reducing the distance to 50 cm could only increase the maximum viewing angle to 45°. Which means that even at a close distance, the maximum viewing angle is 15° less than the paper marker version.

| | Paper Marker | Socket Marker |
|---|---|---|
| Maximum Detection Distance | 120 cm | 81 cm |
| Maximum Viewing Angle (at 50cm distance) | 45° | 60° |

Table 6.1.: Maximum detection distance and maximum viewing angle

### 6.2.4. Translation Vector Error

During the distance measurements, the translation vector of the pose estimation was also recorded. Five measurements were made for each distance. The maximum differences of the translation vector components were about 0.02, which corresponds to 2 cm. For each distance, the mean translation vector was calculated. Since the camera and the marker are aligned, the rotation matrix is the identity matrix. That means that the translation vector represents the camera coordinates in world coordinates. By calculating the norm of the translation vector, we obtain the distance between camera and marker. The calculated distance for the first paper marker measurements is 85,8 cm, which means that the calculation has about a 9,2 cm deviation from the real distance. However, the euclidean distance between the mean translation vectors at a real distance of 95 cm and 107 cm is about 11,6 cm, which means that the distance error is only about half a centimeter if the initial deviation of 9,2 cm is

subtracted. When looking at the euclidean distance between the mean translation vectors at a real distance of 95 cm and 120 cm, we see an increase of the distance error which is about 2,8 cm. So the further away you go, the greater will be the distance error. For more accurate statements, more measured values are needed. The translation vectors of the socket frame marker gave similar results.

# 7. Conclusion

In this thesis, a frame marker detection system was designed and implemented on a smartphone. For the detection, the image coming from the smartphone's camera is first converted into a grayscale image. On the grayscale image, the Canny edge detector is applied. The contours found with it, which form a quadrangle are marked as marker candidates. After analyzing and correcting the code of each marker candidate, pose estimation is performed, if the analyzed code matches. In the last step, the contour, the axis of the marker, as well as the marker code, are displayed on the smartphone's display.

During the implementation and testing phase, it became clear that the inside edge design of the marker worked much better on a socket frame since it was stable when looking on it from an angle. However printed out on paper, both design approaches performed equally good.

The paper marker, as well as, the socket marker version was tested. For measuring the maximum detection distance and maximum viewing angle, the smartphone was mounted on a slider rail. Via a stepper motor, it was possible to move the smartphone precisely and to change the distance. The marker was mounted on a tripod with a rotating head on the opposite side. The position and height of the tripod was chosen so that the marker lied on the optical axis of the camera. Several measurements were taken. The results were that paper marker was still recognizable at about a distance of 120 cm, whereas the maximum detection distance of the socket frame marker was about 80 cm. The maximum possible viewing angle of the socket frame marker was also significantly worse (about 45°) compared to the paper marker (about 60°) at about a distance of 50 cm. The reason for this performance difference is the rounding of the edges of the socket frame.

The calculated translation vector of the pose estimation has about a 0,5 cm deviation in length at close range, and an increasing deviation at greater distances.

Overall the system has a reasonable performance with an average frame rate of about 15 FPS and the goal of designing and detecting a frame marker, which could be put on electrical sockets or light switches, has been reached.

# 8. Future Work

Both in terms of concept and implementation, improvements of the system are conceivable. Instead of scanning the whole picture for markers, a region of interest in the middle of the screen could be defined, where the user is told to place the marker. Thereby only a part of the picture has to be analyzed, and so processing power can be saved and with that higher frame rates can be achieved. Although 15 FPS are good usable, higher frame rates will definitely enhance the user experience. The problem described in 6.1, where the '2D' image of the socket frame marker does not align with the grid for the bits extraction could be compensated by instead of counting all the black and white pixel in each cell of the grid, counting only the pixels lying in the middle of each cell. Therefore when pixels from one encoded bit are overlapping, they are not counted anymore, unless the overlapping is too big.

The frame marker could only be recognized at a distance less than a meter. Therefore it is not possible to have a view of the whole wall, where the marker is placed, because the marker will be out of reach. A solution for that would be combining the current detection system with a marker-less tracking technique like e.g., ARCore provides it. ARCore is an augmented reality framework by Google and uses, amongst other things, the digital compass, velocity meter and accelerometer of the smartphone to perform motion tracking. So the frame marker in the system could be used as an anchor point in ARCore.[22] Now when the marker is no longer visible, we could still track the position of the smartphone/camera via motion tracking and still display the correct augmented image on the screen. In the architectural use case, that would mean that the user points their phone to the frame marker on the electrical socket and approach the marker until the system detects it correctly. From now on the markerless tracking technique will take over and the user can back off the wall so that he can see the whole augmented image of the wall at once.

# A. Software

OpenCV is an open-source computer vision and machine learning software library for both academic research and commercial usage. The library contains over more than 2500 optimized algorithms for computer vision as well as for machine learning. OpenCV can be used for image processing, face recognition, camera movement tracking, marker tracking, etc. Originally written in C++, it also has Python, Java and Matlab interfaces and supports Windows, Linux, Android, and Mac OS.[20] For the frame marker implementation, OpenCV version 3.4.3 was used.

Since the program should run on a mobile device, Android was chosen as the operating system. The program was tested under the newest Android version 9.0. For the development environment, Android Studio version 3.4 was chosen. The whole code of the program is written in Java, as it is supported by Android Studio and OpenCV.

# B. Smartphone Specifications

| OnePlus 5T General Specifications | |
|---|---|
| Operating System | OxygenOS based on Android 7.1.1 Nougat |
| Screen size | 6.01 inches |
| Resolution | 1080 x 2160 |
| Processor | Qualcomm Snapdragon 835 (Octa-Core, 10 NM, up to 2,45 GHz) |
| GPU | Adreno 540 |
| RAM | 6GB |

Table B.1.: OnePlus 5T General Specifications [23]

| OnePlus 5T Camera Specifications | |
|---|---|
| Rear Camera | Dual, 16MP Primary Sensor, 20MP Secondary Sensor |
| Rear Camera Primary Sensor | Sony IMX 398 |
| Pixel Size for Primary Sensor | 1.12µm |
| Aperture and Focal Length for Primary Sensor | f/1.7, 27.22mm |
| Video Recording (Rear Camera) | 4K videos at 30fps, 1080p@60fps/30fps, 720p@30fps, Slow Motion: 720p@120fps, Time Lapse |

Table B.2.: OnePlus 5T Camera Specifications [23]

# C. Translation Vector Measurements

| Paper Marker Translation Vector Measurement | | | |
|---|---|---|---|
| Measurement | Steppermotor Steps | Distance | Translation Vector |
| 1 | 0 | 95 cm | $\begin{bmatrix} 0.0554 \\ 0.0065 \\ 0.8553 \end{bmatrix}$ |
| 2 | 0 | 95 cm | $\begin{bmatrix} 0.0554 \\ 0.0069 \\ 0.8508 \end{bmatrix}$ |
| 3 | 0 | 95 cm | $\begin{bmatrix} 0.0518 \\ 0.0150 \\ 0.8652 \end{bmatrix}$ |
| 4 | 0 | 95 cm | $\begin{bmatrix} 0.0518 \\ 0.0134 \\ 0.8651 \end{bmatrix}$ |
| 5 | 0 | 95 cm | $\begin{bmatrix} 0.0518 \\ 0.0069 \\ 0.8487 \end{bmatrix}$ |
| Mean Translation Vector | | | $\begin{bmatrix} 0.0533 \\ 0.0097 \\ 0.8570 \end{bmatrix}$ |
| Norm | | | 0.8587 |
| 1 | 5000 | 107 cm | $\begin{bmatrix} 0.0601 \\ -0.0287 \\ 0.9707 \end{bmatrix}$ |
| 2 | 5000 | 107 cm | $\begin{bmatrix} 0.0594 \\ -0.0287 \\ 0.9590 \end{bmatrix}$ |
| 3 | 5000 | 107 cm | $\begin{bmatrix} 0.0594 \\ -0.0287 \\ 0.9590 \end{bmatrix}$ |
| 4 | 5000 | 107 cm | $\begin{bmatrix} 0.0607 \\ -0.0287 \\ 0.9737 \end{bmatrix}$ |

| | | | |
|---|---|---|---|
| 5 | 5000 | 107 cm | $\begin{bmatrix} 0.0607 \\ -0.0287 \\ 0.9737 \end{bmatrix}$ |
| Mean Translation Vector | | | $\begin{bmatrix} 0.0601 \\ -0,0287 \\ 0.9672 \end{bmatrix}$ |
| Norm | | | 0.9695 |
| 1 | 10000 | 120 cm | $\begin{bmatrix} 0.0694 \\ -0.0483 \\ 1.0721 \end{bmatrix}$ |
| 2 | 10000 | 120 cm | $\begin{bmatrix} 0.0694 \\ -0.0483 \\ 1.0721 \end{bmatrix}$ |
| 3 | 10000 | 120 cm | $\begin{bmatrix} 0.0688 \\ -0.0487 \\ 1.0696 \end{bmatrix}$ |
| 4 | 10000 | 120 cm | $\begin{bmatrix} 0.0664 \\ -0.0471 \\ 1.0686 \end{bmatrix}$ |
| 5 | 10000 | 120 cm | $\begin{bmatrix} 0.0673 \\ -0.0476 \\ 1.0716 \end{bmatrix}$ |
| Mean Translation Vector | | | $\begin{bmatrix} 0.0683 \\ -0.0480 \\ 1.0708 \end{bmatrix}$ |
| Norm | | | 1.0740 |

| Socket Frame Marker Translation Vector Measurement | | | |
|---|---|---|---|
| Measurement | Steppermotor Steps | Distance | Translation Vector |
| 1 | 0 | 72 cm | $\begin{bmatrix} -0.0138 \\ -0.0151 \\ 0.6459 \end{bmatrix}$ |
| 2 | 0 | 72 cm | $\begin{bmatrix} -0.0135 \\ -0.0135 \\ 0.6454 \end{bmatrix}$ |
| 3 | 0 | 72 cm | $\begin{bmatrix} -0.0139 \\ -0.0126 \\ 0.6397 \end{bmatrix}$ |

| | | | |
|---|---|---|---|
| 4 | 0 | 72 cm | $\begin{bmatrix} -0.0141 \\ -0.0142 \\ 0.6447 \end{bmatrix}$ |
| 5 | 0 | 72 cm | $\begin{bmatrix} -0.0137 \\ -0.0140 \\ 0.6408 \end{bmatrix}$ |
| Mean Translation Vector | | | $\begin{bmatrix} -0.0138 \\ -0.0139 \\ 0.6433 \end{bmatrix}$ |
| Norm | | | 0.6426 |
| 1 | 2500 | 78 cm | $\begin{bmatrix} -0.0035 \\ 0.0024 \\ 0.6933 \end{bmatrix}$ |
| 2 | 2500 | 78 cm | $\begin{bmatrix} -0.0034 \\ 0.0020 \\ 0.7036 \end{bmatrix}$ |
| 3 | 2500 | 78 cm | $\begin{bmatrix} -0.0035 \\ 0.0019 \\ 0.7022 \end{bmatrix}$ |
| 4 | 2500 | 78 cm | $\begin{bmatrix} -0.0034 \\ 0.0034 \\ 0.7036 \end{bmatrix}$ |
| 5 | 2500 | 78 cm | $\begin{bmatrix} -0.0046 \\ 0.0011 \\ 0.6850 \end{bmatrix}$ |
| Mean Translation Vector | | | $\begin{bmatrix} -0.0037 \\ -0.0022 \\ 0.6975 \end{bmatrix}$ |
| Norm | | | 0.6976 |
| 1 | 3500 | 81 cm | $\begin{bmatrix} 0.0192 \\ 0.0221 \\ 0.7373 \end{bmatrix}$ |
| 2 | 3500 | 81 cm | $\begin{bmatrix} 0.0185 \\ 0.0220 \\ 0.7278 \end{bmatrix}$ |
| 3 | 3500 | 81 cm | $\begin{bmatrix} 0.0185 \\ 0.0199 \\ 0.7366 \end{bmatrix}$ |

| | | | |
|---|---|---|---|
| 4 | 3500 | 81 cm | $\begin{bmatrix} 0.0185 \\ 0.0185 \\ 0.7365 \end{bmatrix}$ |
| 5 | 3500 | 81 cm | $\begin{bmatrix} 0.0171 \\ 0.0185 \\ 0.7365 \end{bmatrix}$ |
| Mean Translation Vector | | | $\begin{bmatrix} 0.0184 \\ 0.0202 \\ 0.7349 \end{bmatrix}$ |
| Norm | | | 0.7354 |

# List of Figures

2.1. Sensitivity Spectrum of Human Cone Cells [6] . . . . . 4
2.2. Applying the Canny edge detector on a gray scale image . . . . . 7
2.3. Pinhole Camera Model . . . . . 8
2.4. Radial Distortion . . . . . 11

4.1. Flow chart of the general detection procedure . . . . . 16

5.1. Different Marker Designs . . . . . 18
5.2. Effects of different parameter Values on the approximation accuracy . . . . . 20
5.3. Applying perspective transformation on the image . . . . . 21
5.4. Visualization of marker information . . . . . 24

6.1. Applying perspective transformation on socket frame image . . . . . 27
6.2. Overview of measuring setup . . . . . 28

# List of Tables

5.1. White corner orientation encoding . . . . . 22

6.1. Maximum detection distance and maximum viewing angle . . . . . 29

B.1. OnePlus 5T General Specifications [23] . . . . . 34
B.2. OnePlus 5T Camera Specifications [23] . . . . . 34

# Bibliography


[1] I. E. Sutherland. "A head-mounted three dimensional display". In: *Proceedings of the December 9-11, 1968, fall joint computer conference, part I on - AFIPS 68 (Fall, part I)* (1968).

[2] *Market Watch Article*. https://www.marketwatch.com/press-release/augmented-reality-and-virtual-reality-market-2019-global-trends-size-segments-and-growth-by-forecast-to-2025-2019-03-20; last accessed 15.09.2019.

[3] *Smart Reality Website*. https://smartreality.co; last accessed 15.09.2019.

[4] M. L. C. Lordemann. *Objecterkennung in Bilddaten*. Universität Würzburg, 2003.

[5] G. H. Joblove and D. Greenberg. "Color Spaces for Computer Graphics". In: *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '78. New York, NY, USA: ACM, 1978, pp. 20–25.

[6] *Human Cone Cells Image*. https://commons.wikimedia.org/wiki/File:Cone-fundamentals-with-srgb-spectrum.svg; last accessed 15.09.2019.

[7] N. Efford. *Digital Image Processing: A Practical Introduction Using Java (with CD-ROM)*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN: 0201596237.

[8] T. Boutell. *PNG (Portable Network Graphics) Specification Version 1.0*. RFC 2083. RFC Editor, Mar. 1997.

[9] W. Burger and M. Burge. *Digitale Bildverarbeitung: Eine Einführung Mit Java und ImageJ*. Springer, 2005. ISBN: 9783540214656.

[10] E. R. Davies. *Computer Vision - Principles, Algorithms, Applications, Learning*. 5. Amsterdam, Boston: Academic Press, 2017. ISBN: 978-0-128-09575-1.

[11] D. H. Douglas and T. K. Peucker. "Algorithms For The Reduction Of The Number Of Points Required To Represent A Digitized Line Or Its Caricature". In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 10.2 (1973), pp. 112–122.

[12] G. Bradski. *Learning OpenCV, [Computer Vision with OpenCV Library ; software that sees]*. 1. ed. O'Reilly Media, 2008. ISBN: 0-596-51613-4.

[13] E. A. Maxwell. *The methods of plane projective geometry based on the use of general homogeneous coordinates* -. London: The University press, 1946.

[14] P. Heckbert. "Projective Mappings for Image Warping". In: (1999).

[15] *Pincushion Distortion Image*. https://commons.wikimedia.org/wiki/File:Pincushion_distortion.svg; last accessed 15.09.2019.

[16] *Barrel Distortion Image*. https://commons.wikimedia.org/wiki/File:Barrel_distortion.svg; last accessed 15.09.2019.

[17] Z. Zhang. "A Flexible New Technique for Camera Calibration". In: *IEEE Trans. Pattern Anal. Mach. Intell.* 22.11 (Nov. 2000), pp. 1330–1334.

[18] R. Roberto, D. Freitas, J. P. Lima, V. Teichrieb, and J. Kelner. "ARBlocks: A Concept for a Dynamic Blocks Platform for Educational Activities". In: *2011 XIII Symposium on Virtual Reality* (2011).

[19] D. Wagner, T. Langlotz, and D. Schmalstieg. "Robust and unobtrusive marker tracking on mobile phones". In: *2008 7th IEEE/ACM International Symposium on Mixed and Augmented Reality* (2008).

[20] *OpenCV (Open Source Computer Vision Library)*. https://opencv.org/; last accessed 15.09.2019.

[21] *Vizario Website*. https://www.vizar.io/vizariocam/; last accessed 15.09.2019.

[22] *ARCore Website*. https://developers.google.com/ar/; last accessed 15.09.2019.

[23] *OnePlus 5T Website*. https://www.oneplus.com/de/support/spec/oneplus-5t; last accessed 15.09.2019.