HAW HAMBURG

BACHELOR THESIS
Justus Johann Biermann

Grid Reconstruction from Detected Corner Points of a Calibration Pattern

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE
WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Justus Johann Biermann

Grid Reconstruction from Detected Corner Points of a Calibration Pattern

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Angewandte Informatik*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Peer Stelldinger
Supervisor: Prof. Dr. Julia Padberg

Submitted on: 14.05.2024

Justus Johann Biermann

Title of Thesis

Grid Reconstruction from Detected Corner Points of a Calibration Pattern

Keywords

Camera, Calibration, De Brujin Torus, Pattern, MST, Kruskal, Union Find

Abstract

Autonomous vehicles depend on camera sensors to detect visual features, like fiducial markers and draw conclusions from these. However, the cameras first need to be calibrated, often realized using calibration patterns.
This thesis introduces an automatic grid construction algorithm for a new pattern which combines a checkerboard with a fiducial marker system. It is designed to correctly index detected corner points, particularly under challenging situations. By implementing an automatic corner detection method and developing a robust grid construction process based on Kruskal's algorithm, consistent corner point indexing and efficient runtimes are achieved. For partial occlusion and fisheye distortions, the indexing remains consistent. The algorithm allows for viewing angles up to 50° with future enhancements to it potentially extending the viewing angles.

Kurzzusammenfassung

Autonome Fahrzeuge sind auf Kamerasensoren angewiesen, um visuelle Merkmale wie Fiducial Marker zu erkennen und daraus Schlüsse zu ziehen. Allerdings müssen die Kameras zuerst kalibriert werden, was oft mithilfe von Kalibrierungsmustern geschieht. Diese Arbeit stellt einen automatischen Rasterkonstruktionsalgorithmus für ein neues Muster vor, das ein Schachbrettmuster mit einem Fiducial Marker System kombiniert. Es ist darauf ausgelegt, erkannte Eckpunkte korrekt zu indexieren, insbesondere in schwierigen Situationen. Durch die Implementierung einer automatischen Eckpunkt-Erkennungsmethode und die Entwicklung eines robusten Rasterkonstruktionsprozesses auf Basis von Kruskals Algorithmus werden konsistente Eckpunkt-Indizierungen und effiziente Laufzeiten erreicht. Bei teilweiser Verdeckung und Fischaugenverzerrungen bleibt die Indizierung konsistent. Der Algorithmus ermöglicht Betrachtungswinkel bis zu 50°, wobei zukünftige Verbesserungen diese Winkel noch erweitern könnten.

Contents

List of Figures

List of Tables

Abbreviations

DBT	De Brujin Torus.
MST	Minimum Spanning Tree.

1 Introduction

1.1 Motivation

Modern autonomous systems like mobile robots or self driving vehicles need to successfully master the tasks of navigation, 3-D scene reconstruction and pose estimation[4]. As cameras are often the sensors of choice, a growth of camera sensor usage is implicitly given. Camera pose estimation can be realized by using fiducial markers[16]. However, in many cases, cameras first need to be correctly calibrated to effectively work with fiducial markers as pose estimation[11].

The broadly used calibration frameworks, like OpenCV[6], rely on the full visibility of the board for correct calibration[2]. This makes them less user-friendly and complicates the calibration process, as no partial occlusion is allowed and the whole board has to be in the frame.

Combining the advantages of calibration patterns like a chessboard and fiducial marker systems like ArUco allows for calibrating a camera with partial occlusions. Furthermore, the junctions of the chessboard tiles will allow for high-precision detection down to subpixel accuracy.[2]

Prof. Dr. Stelldinger has developed a novel calibration pattern that also combines the chessboard pattern with a fiducial marker system. However, instead of encoding bits in the form of small squares, it encodes bits along the chessboard edges with a single circle for each edge. This is built on multiple *De Brujin Tori* to give each 3×3 cutout of tiles on this board a unique identification and thus a unique location on the whole board, when decoding the edges' bits.

This thesis develops a fast algorithm robust against partial occlusion, distortions and different viewing angles. It is done by first detecting the corner points robustly and fast based on *Liu et al.'s*[21] method for the calibration pattern, then constructing the grid of the chessboard and finally decoding the bits encoded on every edge.

1.2 Goal of this work

The goal of this work is to build a robust automatic grid construction algorithm for the novel calibration pattern. Based on an already robust and adapted corner detection introduced in reference [21], it should be able to correctly index the puzzleboard corners for challenging conditions being: partial occlusion, rotation of the board, distorted images and steep viewing angles.

1.3 Related Work

Recent scientific efforts have focused on developing automatic chessboard corner detection algorithms. Especially, for high-accuracy detection, convolutional neural networks are used in recent work. Notably, Chen et al.[7] introduced a robust, automatic, and precise corner detection method that takes advantage of a fully convolutional network architecture. In reference [17] the neural network not only detects an initial set of corner points, but also detects unreliable input images, discarding those, where no full calibration pattern is detected. Thus, the algorithm reaches higher precision in the overall task of corner detection on the calibration pattern, but is not suitable for calibration with a partly occluded calibration pattern.

Other recent work has been focused on algorithms that do not rely on machine learning, especially when efficient runtimes are needed. *Geiger et al.* [12] pioneered a method for corner detection that uses region growing, facilitating the identification and matching of several checkerboards simultaneously within a single capture from a stereo camera configuration. The work in reference [10] focuses on robustly detecting occluded chessboard patterns, relying on the graph detection used in the already robust corner detection method in reference [24]. A fast and robust corner point detection approach, though not handling occlusions, has been contributed by Liu et al. [21] enhancing the Hessian detector.
Based on the fast corner detection method provided in reference [21] this thesis' corner detection algorithm has been implemented.
Using fiducial marker systems for the task of camera calibration has been shown in reference [9] yielding good results for partial occlusions and narrow viewing angles. An automatic generation and detection of highly reliable fiducial markers under occlusion has been developed by *S. Garrido-Jurado et al.*[11]. However, this approach relies on the

fiducial markers being 4-point polygons to be detected reliably. They first segment the image, then extract contours and apply filtering for 4-vertex polygons. Finally, having a 4-vertex polygon, they can apply an image transform to robustly decode a marker, even under steep viewing angles.

1.4 Structure of this thesis

Chapter 2: Background covers an introduction into camera calibration, calibration patterns and fiducial markers. Finally, the puzzleboard calibration pattern and its characteristics are explained.

Chapter 3: Detection of Puzzleboard Corners presents the steps taken to detect a set of corner candidates based on related work, which are used for grid construction in the next chapter.

Chapter 4: Grid Construction from Corner Points explains the grid construction algorithm. First, related grid constructions are taken into consideration. Then, the data structures and algorithms used for the grid construction are explained in detail. Next, the process of merging two coordinate systems in the grid construction is presented, and finally, the grid extraction from the constructed minimum spanning trees is explained.

Chapter 5: Experiments covers experiments for the task of corner detection and edge weighting. Specifically, the factor for the Hessian matrix trace and the weighting for edges that do not fulfill a true neighbor relationship in the puzzleboard grid are experimentally determined.

Chapter 6: Evaluation discusses the success of the developed algorithm by testing its ability on correct corner indexing for different viewing angles, distorted images and partial occlusions. Finally, constraints for the usage of this algorithm are drawn from these findings.

Chapter 7: Conclusion and Outlook gives a conclusion of this thesis and provides an outlook for ideas of further possible improvements to the algorithm.

2 Background

This chapter explains the foundations of camera calibration and the according patterns used for the task. Finally, it introduces the novel puzzleboard pattern along with its distinctive features, for which the grid construction algorithm is later built.

2.1 Camera Calibration

For visual tasks, such as measuring objects and depth estimation (illustrated in Figure 2.1), a camera first needs to be calibrated[3].

Zhang et al.[30] have developed a state-of-the-art camera calibration method. It is used to calibrate the intrinsic and extrinsic parameters of a camera in a pinhole model using a calibration pattern.

The goal of camera calibration is to retrieve the correct mapping from 3-D real world coordinates into the image 2-D coordinates, as seen in Figure 2.2.

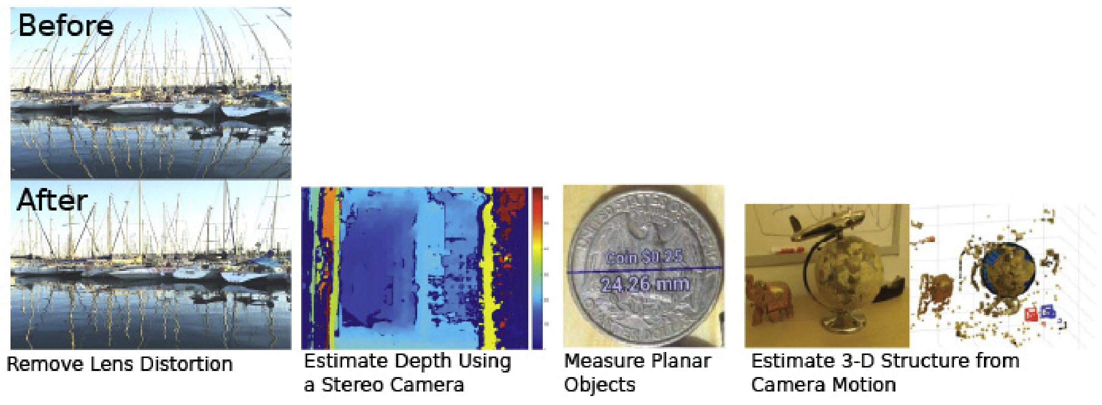

Before
After
Remove Lens Distortion
Estimate Depth Using a Stereo Camera
Coin $0.25
24.26 mm
Measure Planar Objects
Estimate 3-D Structure from Camera Motion

Figure 2.1: “Examples of what you can do after calibrating your camera”[3]

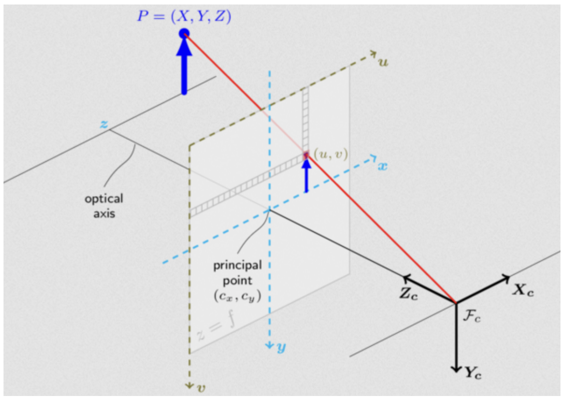


Figure 2.2: Pinhole Camera Model. Figure taken from reference [1]

This relationship between a 3-D point **M** and its 2-D image projection **m** is defined as:

$$s\mathbf{m} = A[R\,t]\mathbf{M} \quad [30]$$

Where A is the intrinsics matrix, which "represent[s] a projective transformation from the 3-D camera's coordinates into the 2-D image coordinates."[3] The extrinsics are represented by R and t, "[. . .] represent[ing] a rigid transformation from 3-D world coordinate systems to the 3-D camera's coordinate system."[3]

For an uncalibrated camera, the intrinsic and extrinsic parameters are unknown. Essentially, calibrating a camera is the process of solving for these parameters by recording a set of 3-D real world coordinates with known 2-D image point correspondences[3].

2.1.1 Intrinsic Parameters

The intrinsics matrix, seen in Equation 2.1, holds the intrinsic parameters for calibration.

$$A = \begin{bmatrix} \alpha & \gamma & u_0 \\ 0 & \beta & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.1}$$

u_0, v_0 are the coordinates of the camera's principal point, which essentially is the center of the image coordinate system. α and β are the scale factors in the image u and v axes[30]. γ describes the skew of the two image axes[30].

2.1.2 Extrinsic Parameters

The extrinsic parameters consist of the rotation matrix R and the translation t[30]:

$$[R\,t] = [r_1, r_2, t]$$

They describe the rotation and translation of the real world coordinates into the camera coordinate system[30].

2.1.3 Radial and Tangential Distortion

In reality, cameras are not just pinhole models but have a lens built in. Thereby, cameras can also have **radial** and **tangential** lens distortion, which must be represented in the camera model.[3]

The radial distortion (see Figure 2.3) is modeled by the *radial distortion coefficients*[3]:

$$x_{distorted} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)$$

$$y_{distorted} = y(1 + k_1 r^2 + k_2 r^4 + k3 r^6)$$

Where x, y are the undistorted pixel locations, k_1, k_2 and k_3 are the radial distortion coefficients and $r^2 = x^2 + y^2$.[3]

The tangential distortion (see Figure 2.4) is modeled by the *tangential distortion coefficients*[3]:

$$x_{distorted} = x + [2p_1 xy + p_2(r^2 + 2x^2)]$$

$$y_{distorted} = y + [p1(r^2 + 2y^2) + 2p_2 xy]$$

Again with x, y as the undistorted pixel locations, $p1$ and $p2$ as the tangential distortion coefficients and $r^2 = x^2 + y^2$.[3]

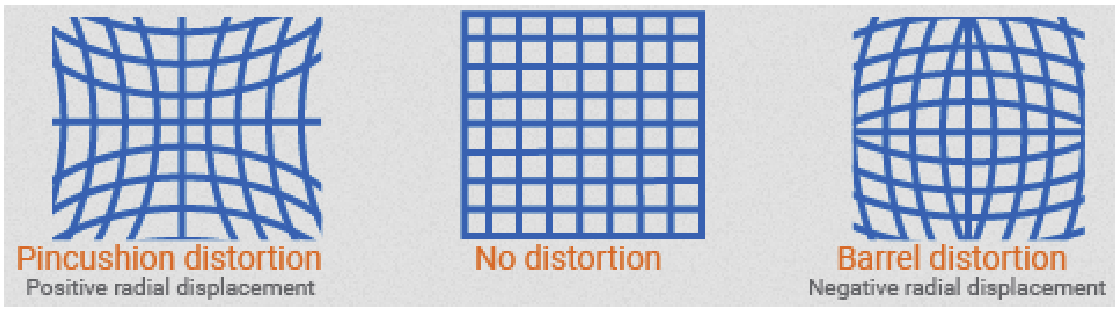


Figure 2.3: Radial Distortion Types. Taken from reference [3]

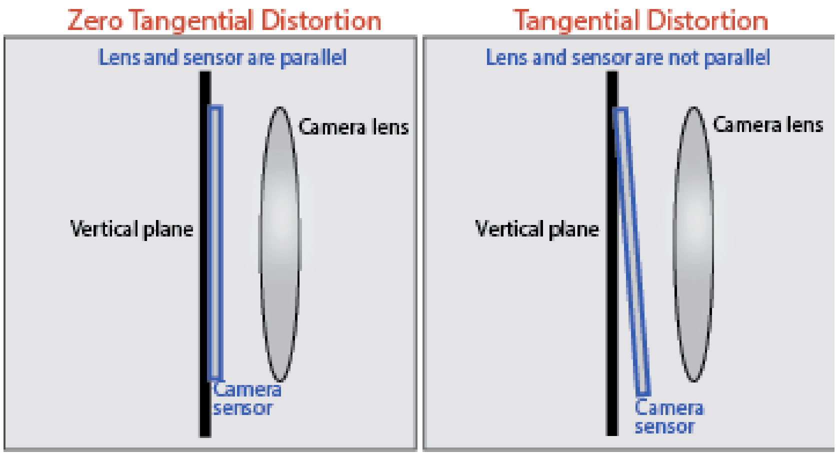


Figure 2.4: Tangential Distortion. Taken from reference [3]

2.2 Camera Calibration Patterns

In reference [30], a calibration pattern on a plane is needed to properly calibrate the camera. Calibration patterns aim at providing easy detectable markers, often realized by corner junctions in the conventional patterns (Figure 2.6). This section shortly outlines the two patterns broadly used for the task of camera calibration.

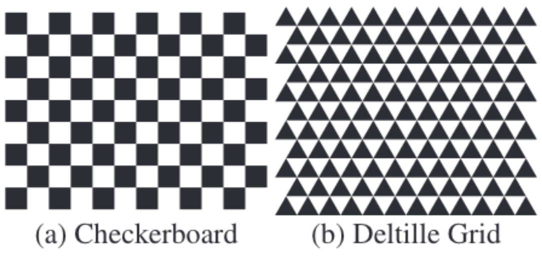


Figure 2.5: (a) Checkerboard Pattern, (b) Deltille Pattern. Figure adapted from [13]

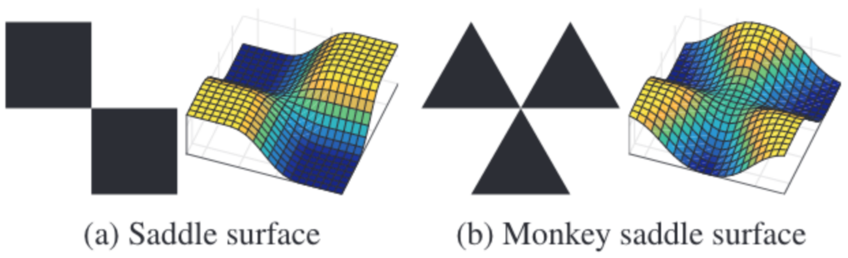


Figure 2.6: “Two types of checkerboard corners and corresponding local surface shape”[13]. Figure adapted and taken from reference [13].

2.2.1 The Checkerboard

The checkerboard pattern as seen in Figure 2.5(a) is frequently utilized for calibration purposes due to its simplicity in creation and the precise pattern it offers, characterized by known dimensions and measurements [10, 24]. The pattern’s x-junctions, formed where two black and two white squares meet, produce strong corner responses, making it ideal for calibration tasks.

2.2.2 Deltille Grid Pattern

The Deltille grid pattern (Figure 2.5(b)) introduced in reference [13] aims at providing more accurate calibration for high resolution cameras. It offers a denser tiling and is more robust due to its isotropic tiling, when viewed under perspective transformation[13]. Furthermore, each junction consists of three edges intersecting compared to the two edges in the checkerboard pattern (Figure 2.6), thus supplying a greater number of constraints for each corner point[13].

2.3 Fiducial Markers

Fiducial Markers are used in Augmented Reality systems to mark and track objects in the real world. They contain codes that determine the information to be displayed at the marker's position (Figure 2.7)[22].

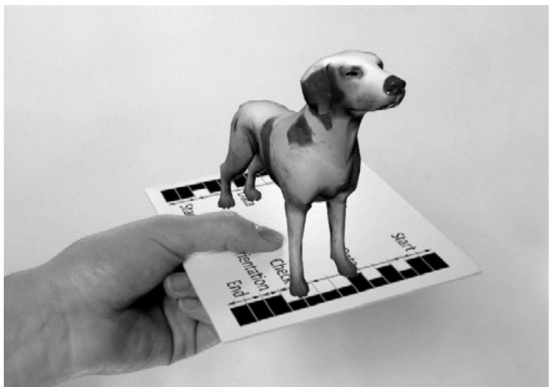


Figure 2.7: Fiducial Marker: "Split Marker" used to track a virtual 3d object. Figure adapted from [22].

For mobile robotics, fiducial markers often find usage for pose estimation.[16] *Garrido-Jurado et al.*[11] designed highly reliable fiducial markers, based on *ARTags*[8] (Figure 2.8).

Garrido-Jurado et al. also provide an algorithm to automatically detect fiducial markers even under partial occlusion. Furthermore, a dynamic dictionary for the encoded information in these fiducial markers is introduced. Thereby, designing dynamic and reliable fiducial markers.

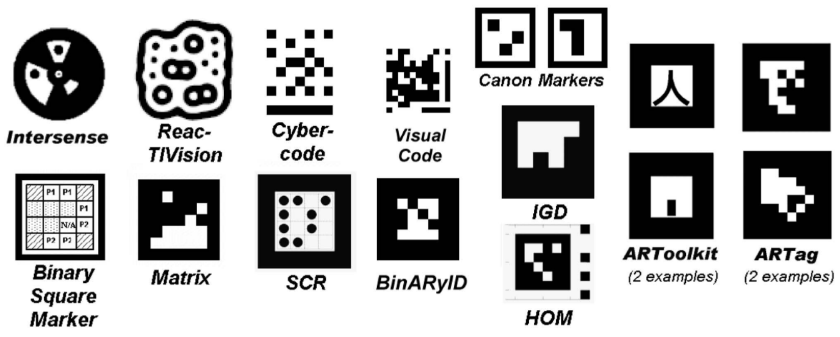


Figure 2.8: Different fiducial markers. Figure taken from reference [11].

In reference [9] it has been shown that fiducial markers may also be used for the task of camera calibration.

2.4 The Puzzleboard

The calibration pattern used in this thesis is a novel board created by *Prof. Dr. Stelldinger*. From now on, the board will be called “puzzleboard” and the black and white tiles will be named “puzzle tiles”. It still offers strong corner junctions, as the white and black tiles still exist and therefore can be detected by conventional checkerboard corner detectors. Furthermore, the puzzle tiles’ edges either have a black or a white circle encoding a 0 or a 1, respectively. This is based on the *De Brujin Sequence*. Similarly to [26], the puzzleboard represents multiple De Brujin Tori.

With the unique encoding of each 3×3 cutout, it additionally serves as a fiducial marker system. The pattern is robust against different resolutions. Even in very low resolutions or a dense formation, this pattern is still decodable.

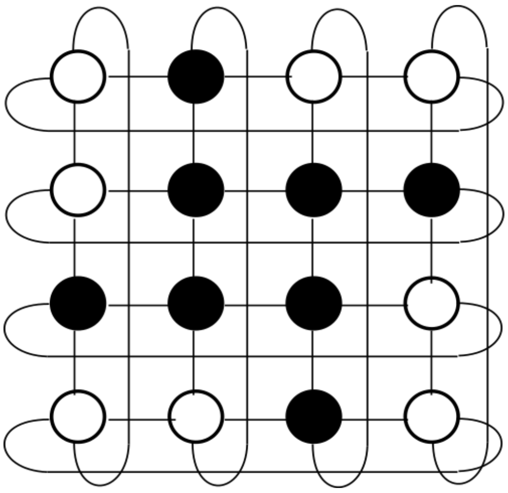
Finally, as already mentioned above and shown in reference [9], a robust fiducial marker system may also be used for automatic camera calibration. Therefore, enabling the puzzleboard to be used as a calibration pattern by tradtionial calibration algorithms, but also being able to be used for systems relying on fiducial markers like Augmented Reality or Robotics.

2.4.1 De Brujin Torus

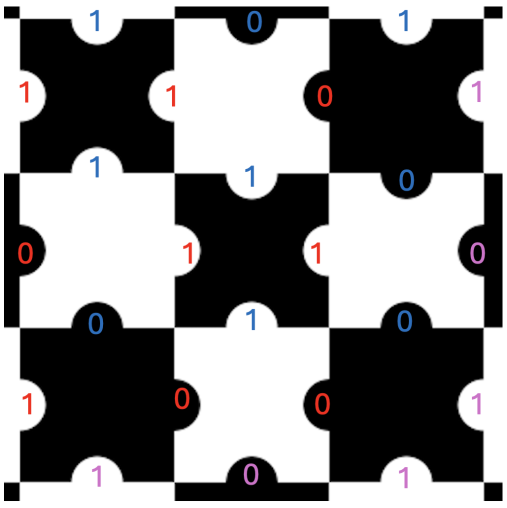
Taken from reference [14]: "A de Brujin array(or $(r, v; n, m)_d$-array) is an $r \times v$ d-ary array in which every window of size $n \times m$ appears exactly once. A De Brujin Torus (DBT) is a de Brujin array in which the last row is adjacent to the first row, and similarly the last column is adjacent to the first column." An example DBT is illustrated in Figure 2.9a.

For the puzzleboard, two DBTs with $DBT_v = (3, 167, 3, 3)_2$ and $DBT_h = (167, 3, 3, 3)_2$ are used for the vertical lines and horizontal lines, respectively. In many cases, more than three rows and columns are needed for the puzzleboard, thus DBT_v will be repeated on the vertical axis, after every three rows and DBT_h will be repeated on the horizontal axis, after every three columns.

Once a 3×3 window of edges is known (as seen in Figure 2.9b), the position of this window on the whole puzzleboard can be inferred. This is possible because its decoded 3×3 window of vertical edges is unique in every DBT_v and the 3×3 window of decoded horizontal edges is unique in every DBT_h.



(a) De Brujin Torus example taken from reference [5]. White equals 1 and black equals 0.



(b) Decoded edges of a cutout of the puzzleboard. In red the vertical edges and in blue the horizontal edges. Pink shows the repeated pattern of the DBT in that direction.

2.5 Corners of Interest

A chessboard consists of inner and outer corners, as seen in Figure 2.10.

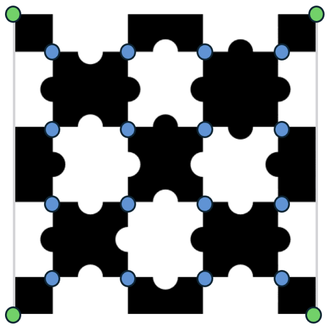


Figure 2.10: Puzzleboard with inner corners (blue) and outer corners (green)

In this thesis, the corners of relevance are the inner corners (blue corners in Figure 2.10) of the puzzleboard.

3 Detection of Puzzleboard Corners

In this chapter, the algorithm for detecting the initial corner point set is presented by iterating over the different steps of the detection algorithm.

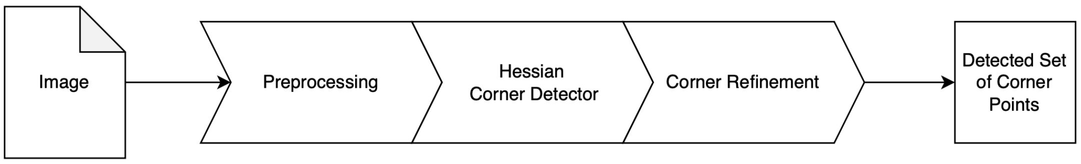


Figure 3.1: Corner Detection Pipeline. Consisting of three steps, it yields a detected corner point set of a given input image.

For the task of retrieving corners of an input image, the steps needed can roughly be divided in the three areas seen in Figure 3.1: *Preprocessing, Hessian Corner Detection, and Corner Refinement.*

3.1 Preprocessing

This section will discuss the required preprocessing steps for the image prior to initiating corner detection.

3.1.1 Normalization of the Input Image

By using OpenCV's method `cvtColor()` [6], the input image will be translated into a grayscale image. To further preprocess the image, it is normalized by dividing each pixel's value by the maximum pixel value in the whole image. Given the image as a function $f(x, y) = z$ with $z \in \mathbb{N}[0; 255]$. x and y denote a pixel's position in the image and z its corresponding grayscale value in between 0 and 255. Then the normalized image is retrieved by creating the new normalized image function $n(x, y)$:

$$n(x, y) = \frac{f(x, y)}{f_{max}}, \; with \; f_{max} = max_{x,y} f(x, y) \tag{3.1}$$

After the normalization, each pixel represents a value z, with $z \in \mathbb{R}[0; 1.0]$. 0 represents a black pixel and 1.0 represents a white pixel. All values in between 0 and 1.0 represent a gray-scale value.

3.1.2 Gaussian Blur

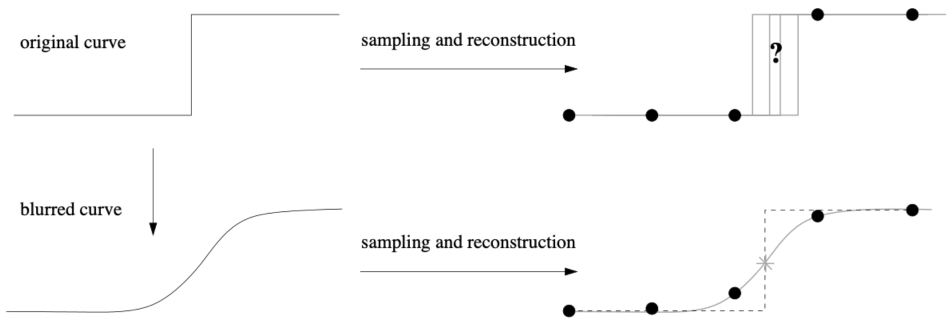
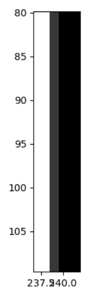


Figure 3.2: Blurring an image allows for a smooth transition between white and black pixels, thus enabling a more precise estimation of the saddle point. Figure adapted from reference [19]

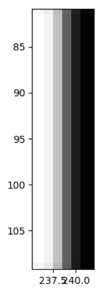
The Gaussian filter is applied to images for smoothing or blurring, effectively reducing noise. Importantly, this filter also establishes saddle points essential for further detailed image analysis[21].

The creation of the saddle point is illustrated in Figure 3.2. While the retrieval of an estimated inflection point in the top figure shown in Figure 3.2 might be unprecise (towards one or the other pixel), the applied gaussian blur will enable a better approximation of the inflection point as seen in the bottom figure of Figure 3.2.

The gaussian blur is realized by using OpenCV2's `cv2.GaussianBlur()` [6] function with a 5×5 kernel size and standard deviation set to $\sigma = 1$. The resulting edges can be seen in Figure 3.3b.



(a) Close-up of a synthetic edge without any blur



(b) Close-up of a synthetic edge with blur

Figure 3.3: Close-ups of synthetic edges. Without gaussian blur and with gaussian blur, respectively.

After the preprocessing step is done, the actual corner detection can take place. From here on the preprocessed input image will be named $\boldsymbol{I_p}$.

3.2 Hessian Corner Detector

In reference [21] an adapted version of the Hessian corner detector is introduced as a fast and robust corner detector. In this section, the in reference [21] introduced Hessian detection is explained, covering the second step of the pipeline shown in Figure 3.1.

3.2.1 Relevance of Hessian Matrix for Corner Detection

To begin with, the Hessian matrix holds all partial second-order derivatives of a function with n dimensions. Consequently, the Hessian matrix can be used as a tool to detect areas of high curvature in n-dimensional space. An image can be modeled as a two-dimensional function, and as already seen in Figure 2.6 from section 2.2, the puzzleboard corners are saddle points in this function. The Hessian matrix allows detecting saddle points in an image, therefore, being a suitable tool for finding puzzleboard corner candidates.

Let $f(x, y)$ be the image and $r(x, y)$ the image with gaussian blur applied, then the Hessian matrix is defined as (taken from references [15] and [21]):

$$H_{f(x,y)} = \begin{bmatrix} \frac{\partial^2 r}{\partial x^2} & \frac{\partial^2 r}{\partial x \partial y} \\ \frac{\partial^2 r}{\partial x \partial y} & \frac{\partial^2 r}{\partial y^2} \end{bmatrix} \tag{3.2}$$

Or simplified:

$$H = \begin{bmatrix} r_{xx} & r_{xy} \\ r_{xy} & r_{yy} \end{bmatrix} \tag{3.3}$$

To calculate r_{xx}, r_{yy} and r_{xy}, the discrete derivatives in x and y directions are calculated. Furthermore, an additional Hessian matrix on diagonal derivatives is calculated to enhance the algorithms' robustness. The kernels D_y, D_x, D_{rl} and D_{lr} are convoluted with I_p, yielding:

$$\begin{gathered} I_p * D_y = r_y \\ r_y * D_y = r_{yy} \\ r_y * D_x = r_{xy} \\ I_p * D_x = r_x \\ r_x * D_x = r_{xx} \\ I_p * D_{rl} = r_{rl} \\ r_{rl} * D_{rl} = r_{rl\_rl} \\ r_{rl} * D_{lr} = r_{rl\_lr} \\ I_p * D_{lr} = r_{lr} \\ r_{lr} * D_{lr} = r_{lr\_lr} \end{gathered} \tag{3.4}$$

with

$$D_y = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \quad D_x = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \tag{3.5}$$

$$D_{lr} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \quad D_{rl} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{3.6}$$

Where I_p is the preprocessed image. Convoluting I_p with the corresponding derivative filters will yield derivatives in x, y, diagonal right-left (rl) and diagonal left-right (lr) directions, respectively.

3.2.2 Finding Corner Candidates with the Hessian Detector

As the corner junctions of black and white squares in the puzzleboard are saddle points in I_p, the Hessian detector allows for the detection of the corners of I_p.[21]

in reference [21], the response of the corner detector used to find chessboard corners is named S and consists of the determinant of the Hessian matrix H such that S is defined as:

$$S = det(H) = r_{xx} \cdot r_{yy} - r_{xy}^2 \quad [21] \tag{3.7}$$

In this thesis, the equation is adapted by multiplying the trace of the Hessian matrix by a factor k yielding more robust detection (bold part in 3.8).[27]

$$\begin{aligned} S = det(H) &= r_{xx} \cdot r_{yy} - r_{xy}^2 + \boldsymbol{k \cdot trace(H)^2} \\ &= r_{xx} \cdot r_{yy} - r_{xy}^2 + \boldsymbol{k \cdot (r_{xx} + r_{yy})^2} \end{aligned} \tag{3.8}$$

The same as seen in Equation 3.8 is done for the diagonal derivatives. r_{xx} is replaced by r_{rl_rl}, r_{yy} is replaced by r_{lr_lr} and r_{xy} is replaced by r_{rl_lr}:

$$\begin{aligned} S_d = det(H_d) &= r_{lr\_lr} \cdot r_{rl\_rl} - r_{lr\_rl}^2 + \boldsymbol{k \cdot trace(H_d)^2} \\ &= r_{lr\_lr} \cdot r_{rl\_rl} - r_{lr\_rl}^2 + \boldsymbol{k \cdot (r_{lr\_lr} + r_{rl\_rl})^2} \end{aligned} \tag{3.9}$$

A corner now is retrieved by solving the following constraint in the response S and S_d, respectively:

$$\lambda_1 > \varepsilon \text{ and } \lambda_2 < -\varepsilon \quad [21] \tag{3.10}$$

With λ_1 and λ_2 being the eigenvalues of the Hessian matrix H. Furthermore, it is shown that $S = \lambda_1 \cdot \lambda_2$, thus allowing to search for each pixel in S that solves for $S < \varepsilon$, with ε being a threshold for the response value in S[21].

Or as it is done in this thesis, flipping the response S from minima to maxima, by multiplying each value by -1 and thus solving for $-S > \varepsilon$ instead of $S < \varepsilon$. This is done for S and S_d.
ε is set to 0.03, taken from the reference [21] and k is set to $k = 0.5$ to retrieve robuster response results, as shown in chapter 5.
Finally, S and S_d are element-wise multiplied yielding S_f The corresponding response

S_f can be seen in Figure 3.4 on the right side. Especially high responses are visible at corner junctions of the underlying Puzzleboard.

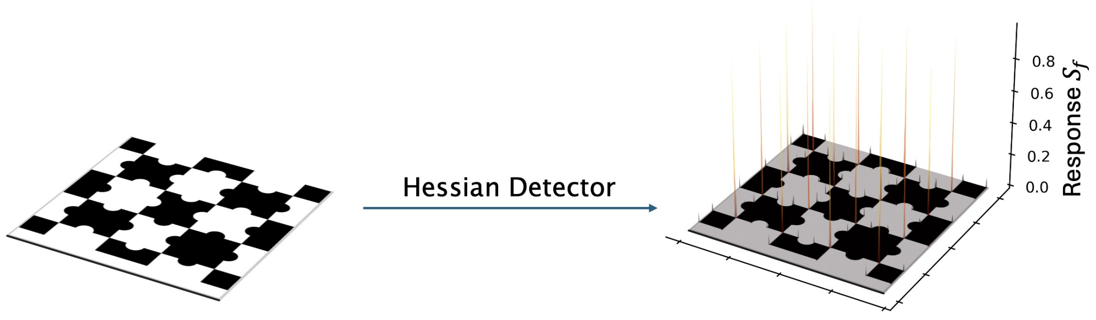


Figure 3.4: Response S_f of Hessian detector. The spikes show the maxima of S_f.

3.2.3 Finding Local Maxima of S_f

Around a detected corner point the response S_f is maximum. In some cases, there is no single maximum value, but rather multiple maximum values. In such situations, a decision must be made to keep only one of these maxima. Therefore, maximum suppression is realized[29] by utilizing the `peak_local_max()` function from the scikit-image[28] library. The `min_distance` parameter is set to 10.

The corner candidates resulting from the maximum suppression form the initial corner candidate set.

3.3 Corner Refinement

Finally, the last step of the pipeline seen in Figure 3.1 is covered in this section. The process of refining the beforehand detected corner points is explained. First, the detected corner set is further filtered by applying a corner mask. Secondly, the position of the detected corners is refined by reaching sub-pixel accuracy. Therefore, yielding a final corner candidate set with sub-pixel accuracy, which can be used for following operations in grid construction seen in chapter 4.

3.3.1 Applying a Corner Mask to Detected Points

Some of the former detected corners with the Hessian detector may not be true puzzleboard junctions, but rather other local maxima of the Hessian response S_f (as seen in Figure 3.5). In consequence, further filtering is needed to remove faulty corners. In this work, due to simplicity, the only filtering used is a so-called Corner Mask.

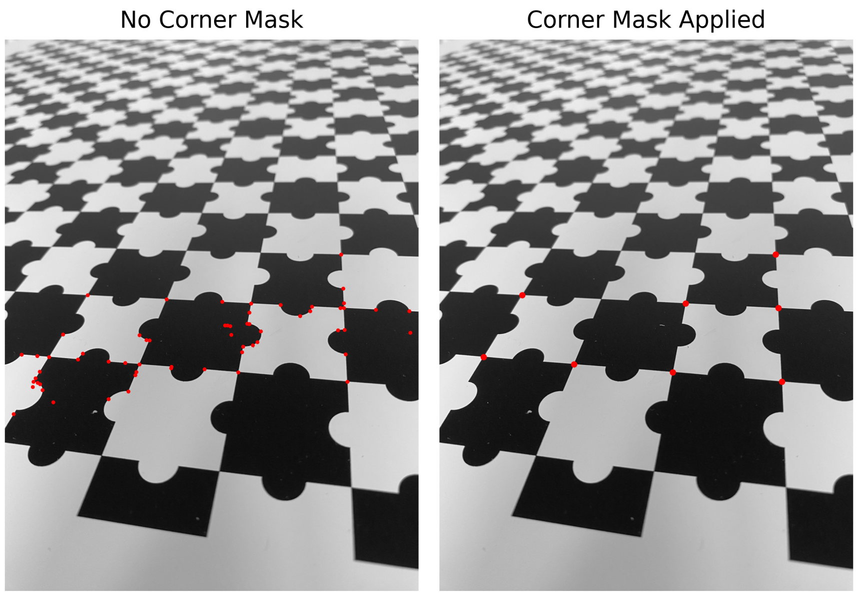


Figure 3.5: Left: Detected corners without corner mask applied. Right: Detected corners with corner mask applied

First, the corner mask uses a couple of kernels to check for potential corners:

$$\begin{bmatrix} -1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

Each of those corner kernels are separately convoluted with I_p and yield strong responses, if the window is on top of a puzzleboard junction. For each pixel in those convoluted results, the strongest response is taken for that position, creating the junction response J.

Secondly, several kernels to filter edges are used:

$$\begin{bmatrix}-1&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&1\end{bmatrix}\begin{bmatrix}0&-1&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&1&0\end{bmatrix}\begin{bmatrix}0&0&-1&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&1&0&0\end{bmatrix}\begin{bmatrix}0&0&0&-1&0\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\0&1&0&0&0\end{bmatrix}$$

$$\begin{bmatrix}0&0&0&0&-1\\0&0&0&0&0\\0&0&0&0&0\\0&0&0&0&0\\1&0&0&0&0\end{bmatrix}\begin{bmatrix}0&0&0&0&0\\-1&0&0&0&0\\0&0&0&0&0\\0&0&0&0&1\\0&0&0&0&0\end{bmatrix}\begin{bmatrix}0&0&0&0&0\\0&0&0&0&0\\-1&0&0&0&1\\0&0&0&0&0\\0&0&0&0&0\end{bmatrix}\begin{bmatrix}0&0&0&0&0\\0&0&0&0&1\\0&0&0&0&0\\-1&0&0&0&0\\0&0&0&0&0\end{bmatrix}$$

Again, each of these responses will be compared pixel-wise, and the maximum response will be taken into account to form the edge response E.

Then, the maximum response for each pixel will be taken along E and J. If, at a pixel's location, J compared to E yields a greater response, the value in the final corner mask response C is set to 1 and else to 0. Thus leading to a binary response C_b of corners in the image I_p.

Finally, the binary corner response C_b will be element-wise multiplied with the original corner response S_f, thus leading to the elimination of wrong corner responses in S_f, since they will be multiplied with 0, if C_b yields 0 at that pixels location.

3.3.2 sub-pixel Accuracy

In some cases, the detected corner, or more precisely the zero-crossing of the Hessian matrix, may not truly lie in the center of a pixel but rather towards one or two of the pixel's edges. To reach sub-pixel accuracy for corner detection, the squared greyscale centroid method is used[21].

$$x_0 = \frac{\sum_{(i,j)\in R} i \cdot I^2(i,j)}{\sum_{(i,j)\in R} I^2(i,j)}, y_0 = \frac{\sum_{(i,j)\in R} j \cdot I^2(i,j)}{\sum_{(i,j)\in R} I^2(i,j)} \quad [21] \tag{3.11}$$

Where $I(i, j)$ is the intensity value at corner c with the position (i, j)[21]. R is the pixel neighborhood of the corner c, chosen as a small circular window centered on c. In this thesis, R is chosen as the 8 pixels surrounding the corner c (a $3\,x\,3$ window). Finally, using Equation 3.11, the sub-pixel position of a corner c is determined to be (x_0, y_0).

This calculation is done for every detected corner, to reach sub-pixel accuracy for all corner candidates.

Altogether, in this chapter, the detection of the intial corner set by using the Hessian detector with the sub-pixel method introduced in reference [21] and a corner mask filtering has been shown. After reaching sub-pixel accuracy for the whole corner set, it can now be used for the next step of grid reconstruction of the puzzleboard in chapter 4.

4 Grid Construction from Corner Points

This chapter describes the construction of the coordinate system from the prefiltered, detected puzzleboard corners. It provides an in-depth overview of the calculation of edge weights, the detailed process of constructing the Minimum Spanning Tree (MST), and the subsequent grid construction. These elements are critical for understanding how the coordinate system is systematically built from the ground up, ensuring accurate indexing of the puzzleboard's inner corners.

4.1 Why the Grid Construction is Needed

The construction of the calibration pattern, essentially meaning the retrieval of the relationship of a detected corner to the other detected corners (*e.g. Is Corner1 with image coordinates [x: 200, y: 150] adjacent to Corner2 with image coordinates [x:210, y:-92]?*), is needed for several reasons.

First of all, to enable a camera calibration with the detected corner points, they have to be indexed correctly to draw relationships between the detected points.
Trivial in the case of a top-down view without any rotation, distortion or partial occlusions, the sole utilization of image coordinates of the detected corner candidates will yield correct indexing.
However, as soon as challenging situations as the aforementioned occur, the sole retrieval of the corners' image coordinates no longer yields a trivial relationship.

Secondly, the puzzleboard holds encoded information on its edges. To successfully decode the pattern, it is necessary to know where this edge lies compared to other edges, since a 3x3 window of correctly allocated edges is needed to find the correct position on the whole puzzleboard.

Finally, in case of partial occlusion or more than one puzzleboard being visible in the image, the grid construction can enable to create disjoint grids, therefore also solving the first mentioned challenge of correct indexing in more complex scenarios.

Taking the above-mentioned criteria into consideration, a grid with a relative coordinate system allowing for correct indexing of its nodes, is the needed structure and will be explained in the following sections.

4.2 Chosen Approach

Liu et al.[21] simply index the detected corner points from top left to bottom right, hence they do not provide any rotational robustness, robustness for steep viewing angles that lead to distortions or robustness for partial occlusion. Consequently, this approach is not considered any further in this work, as the goal is to create a robust grid construction towards partial occlusion, distortions and rotations.

Geiger et al.[12] use an energy function for region growing of the detected corner candidates. Although, being able to automatically detect more than one grid in an image and handling partial occlusion[12], the approach of searching through every possible corner candidate, minimizing the energy function, seems less efficient for larger chessboard patterns (seen in Figure 6.2).

Essentially, the construction of the grid is a construction of a graph. Therefore, a graph structure G is chosen in this thesis to recreate the calibration pattern with its vertices V and edges E.
With $v \in V := \{\text{detected corner points}\}$ and $e \in E := \{(v_i, v_j)|v_i, v_j \in V\}$

An approach to gathering the edges E would be the usage of edge detection in image processing. Nevertheless, in cases of strong lens distortions, this will lead to biased results[24]. Another approach used for edge detection is the Hugh Transform for parallel lines, but as stated by *Fürsattel et al.*[10] this also yields faulty results for distorted images.

Fürsattel et al.[10] use a refined corner detection with subgraph matching, which is not chosen for this thesis, since the generation of the center line image might lead to faulty graphs with the rounded edges of the introduced puzzleboard. Furthermore, it might not be suited for disjoint grids that should be independently detected.

Finally, potential edges are found by determining a Euclidean distance neighborhood of size n, with $n = 8$ for each corner candidate and simply assuming edges between every neighbor in the neighborhood and the corner candidate itself. An example for such a neighborhood on a top-down view can be seen in Figure 4.1b.

For these neighborhood edges, a simple breadth-first search or depth-first search could be used to build a graph structure. However, as seen in Figure 4.2, this might yield wrong edges added to the graph structure, when viewing angles or distortions are present.

Consequently, punishment for potentially wrong edges other than Euclidean distance is needed. Thus, a weighted graph is created, for which the edges are weighted based on Euclidean distance and faulty node pairs.

From the initial edges a construction of a Minimum Spanning Tree (MST) using Kruskal's Algorithm is used in this thesis to build the graph structure, allowing to penalize and exclude wrong edges as the one seen in Figure 4.2. The created MST will consist of nodes, with each node being capable of calculating its relative coordinates in the coordinate system of the MST.

The following sections will cover the used algorithms and data structures, the pre-filtering of the neighborhood edges, and the construction of the MST.

4.3 Algorithm Used for MST Construction

To construct the MST, Kruskal's MST Algorithm is used (see Algorithm 1).
Using Kruskal's Algorithm has two advantages.

Firstly, it allows for disjoint MST's to be created and therefore in the use-case scenario of grid construction, allowing for constructing more than one grid. Hence, detecting more than one puzzleboard in a single image.

Secondly, it can be implemented easily and efficiently, by using the union-find data structure[20], as it is done in this work. Consequently, supporting the goal of a fast algorithm to be employed.

Algorithm 1 Kruskal's Algorithm

```
1: Input: An undirected graph $G = (V, E, \gamma)$ with edge weights $c : E \to \mathbb{R}$
2: Output: The edge set $E_F$ of a minimum spanning forest
3: Sort the edges by their weight: $c(e_1) \leq \ldots \leq c(e_m)$
4: $E_F := \emptyset$
5: for all $v \in V$ do
6:     Make-Set($v$)                                  ▷ Create $n$ single-element sets
7: end for
8: for $i := 1, \ldots, m$ do
9:     Let $u$ and $v$ be the endpoints of $e_i$
10:    if Find-Set($u$) $\neq$ Find-Set($v$) then
11:        $E_F := E_F \cup \{e_i\}$
12:        Union($u, v$)
13:    end if
14: end for
15: return $E_F$
```

The algorithm seen in Algorithm 1 has been taken from the reference [20].

4.4 Data Structures

In the grid construction process, the selection of appropriate data structures is crucial. This section details the data structures selected, outlining their implementation within the grid construction algorithm.

4.4.1 Tree Structure

To efficiently construct and manipulate a MST, it is crucial to employ a tree data structure.

Each node in this tree data structure covers properties, used to later retrieve coordinates of that node in the whole tree. This is essential for generating the MST and subsequently indexing all nodes accurately. The properties maintained in each node include the following:

- **Predecessor:** Every node holds a reference to its predecessor. In case of it being the root node of the tree structure, it references itself.
- **Compressed Predecessor:** The compressed predecessor as created by the *Find(v)* operation, shown in subsection 4.4.2.
- **Absolute Image Position:** The position of the node on the input image. Used for direction decision, when being the first childnode appended to a node.
- **Vector to Predecessor:** This vector holds the distance in x and y values to its predecessor. Note, these x and y values are the relative x and y values of the coordinate system which is being constructed.
- **Subtree:** A set containing all children of a node in its subtree.
- **Edge Weights:** A list structure, holding every already added edge weight to this tree, allowing for average weight operations on the tree for outlier detection.
- **Rotation:** The rotation that applies for every child node's relative coordinates in the node's subtree. (E.g.: A child node has the relative coordinates $x = 1$, $y = 0$. A rotation by 90° would mean, its new location is $x = 0$, $y = 1$.)
- **Neighborhood:** A mapping from direction vectors as of $[1, 0]^T$ for positive x, $[-1, 0]^T$ for negative x, $[0, 1]^T$ for positive y and $[0, -1]^T$ for negative y directions to the according neighboring node as of the tree's relative coordinate system. Needed for later introduced collinearity checks.

4.4.2 Union-Find Data Structure

The union-find data structure is a suitable data structure for an efficient implementation of Kruskal's Minimum Spanning Tree (MST) algorithm[20]. It holds a collection of disjoint sets that dynamically change. Each of these sets contains a so-called representative for the set[20]. Such a data structure should offer three operations:

- *Make(v)*: Creates a new Set with v as the only element and thus representative of the set.
- *Find(v)*: Finds, in all the disjoint sets, the set which holds v and returns this set's representative.

- *Union(u, v)*: Merges the disjoint sets, where u is an element of one of the sets and v is the element of the other set. After merging, the two former sets will be destroyed.

Since the goal is the construction of a MST, instead of sets, tree structures are used. Additionally, using tree structures instead of sets, enables optimizations (*path compression shown below*), thus further optimizing the grid construction algorithm.

Firstly, *Make(v)* will build one-node trees. The representative of a tree is chosen as the root node.

Secondly, *Union(u, v)* will lead to either root node of the tree containing *u* becoming a child node of the tree containing *v* or the other way around, depending on the tree depths. As an optimization step, the shallower tree will become a subtree of the deeper tree [23].

Finally, *Find(v)* will find and return the root node of a tree, for which *v* is one of the nodes in that tree.
To optimize the traversal of nodes in the tree, path compression[23] is used. The implemented *Find-Set(v)* operation can be seen in Figure A.12.

4.4.3 Priority Queue for Kruskal

Kruskal's algorithm, used for the construction of MST, fundamentally requires the sorting of edges based on their weights as seen in Algorithm 1 in line 3. Consequently, a priority queue is employed to efficiently manage and sort all edges in ascending order of their weight, enabling the sequential construction of the MST by retrieving and attempting to add the lightest edges first. In this thesis, the implementation relies on the `PriorityQueue` class from Python's `queue` module[1], where the edge with the smallest weight is assigned the highest priority. This approach takes advantage of the priority queue's ability to efficiently manage and sort data as it changes, which helps streamline the process of building the MST.

[1]https://docs.python.org/3/library/queue.html

4.5 Edge Preparation for Kruskal's Algorithm

This section covers the preparation of the edges later used for the construction of the MST.

Initially, edges that connect a node to its diagonal neighbors are filtered out based on the comparison between the node's eigenvector orientation and that of the neighbor's eigenvector. This step helps in identifying and excluding edges that may connect wrong neighbors, thus breaking the consistent coordinate system.

Secondly, edges that are still wrong after filtering but could not be filtered out by the first step will receive a weight punishment. This penalty reduces their priority, ensuring that they are less likely to be included in the construction of the MST, thereby enhancing the robustness of the algorithm.

4.5.1 Filtering out Faulty Edges

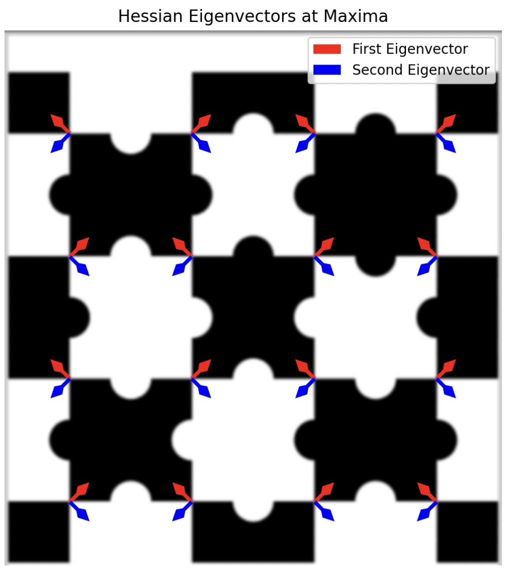
True neighbors are the ones that can be reached by traversing the puzzleboard piece's edge between two corner points. The edges to the correct neighbors are colored in orange in Figure 4.1b.

When simply searching for the eight closest neighbors as of Euclidean distance, diagonal neighbors can be taken into account as well. Therefore, a filtering method for only the true four neighbors is needed.

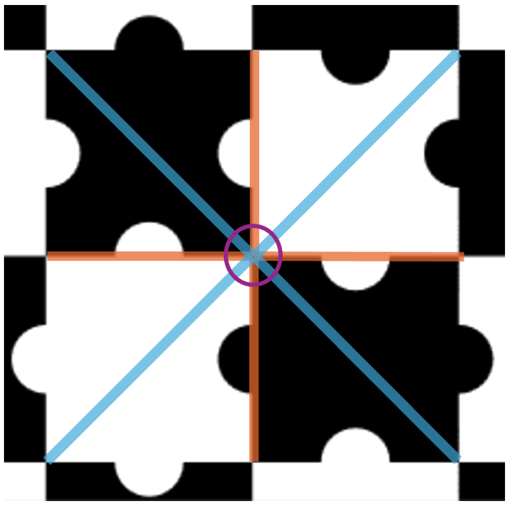
As seen in Figure 4.1a, the two eigenvectors of the Hessian matrix at each detected point are the angle bisectors for the black and white squares at the specific corner point. In Figure 4.1a, the red and blue arrows show the first and second eigenvectors at the detected corner points, which are the angle bisectors of the black and white puzzle tiles, respectively.

Since the Hessian matrix was initially used to identify corner points, using its features is beneficial in this situation. Consequently, from earlier introduced Equation 3.3 the eigenvectors $\vec{e_1}$ and $\vec{e_2}$ of the Hessian matrix can be retrieved as follows[27]:

$$\vec{e_1}, \vec{e_2} = \begin{pmatrix} r_{xx} - r_{yy} \pm \sqrt{(r_{xx} - r_{yy}) \cdot (r_{xx} - r_{yy}) + 4(r_{xy})^2} \\ 2r_{xy} \end{pmatrix} \tag{4.1}$$



(a) Hessian Eigenvectors on a Puzzleboard Cutout



(b) Corner (encircled in purple) with its edges to the eight neighbors, of which only the orange edges are edges of true neighbors

Figure 4.1: Eigenvectors and True Neighbors of a Corner Candidate

Whether the first or second eigenvector of a corner of interest is viewed, the true neighbors' corresponding eigenvector orientations are shifted by 90° (*can be seen in Figure 4.1a. The corner junctions horizontally and vertically adjacent have their Hessian eigenvectors rotated by 90°*).

This property allows for a pre-filtering by calculating the angles of potentially true neighbors eigenvector compared to the currently viewed corner's eigenvector.
Every neighbor yielding a Hessian eigenvector orientation difference below 45° is being filtered out, thus only true neighbors of a corner candidate are kept for further edge calculations[27].

Initially, for every detected corner point cp in the image, the orientation of the Hessian first eigenvector at this point is calculated:

Let $\vec{cp_1}$ be the first eigenvector of the Hessian matrix at the detected corner point cp of all detected corner points CP. The *orientation* for each of the corner points' Hessian eigenvectors can be calculated as follows:

$$ori(cp) := \{\, arctan2(\vec{cp}_{1y}, \vec{cp}_{1x}) \mid cp \in CP \wedge \vec{cp}_1 \text{ is first Hessian eigenvector of } cp\}$$

Once all the orientations have been calculated, an orientation comparison between the first Hessian eigenvector of a corner point and its neighbors first Hessian eigenvectors can be performed to gather the correct neighbors :

$$potentialNeighbor(x, y) := \begin{cases} 1 \text{ for } |\sin(ori(x) - ori(y))| > 0.707 \\ 0 \text{ for } |\sin(ori(x) - ori(y))| \leq 0.707 \end{cases}$$

$$CN := \{(cp_i, cp_j) \mid (cp_i, cp_j \in CP) \land (cp_j \in NN_{cp_i}) \land potentialNeighbor(cp_i, cp_j) = 1\}$$

With NN_{cp_i} indicating the detected corner points neighborhood of cp_i.

In conclusion, this step is utilized to eliminate edges to diagonal neighbors, as seen in Figure 4.1b colored in blue. The elimination of these edges supplements building a grid consisting only of the edges between two true neighbors.

For the following sections, all edges E are defined as the aforementioned CN, such that:

$$E := CN$$

4.5.2 Edge Weight Calculation

The construction of a MST per definition needs weighted edges to be created. Consequently, edge weights need to be calculated to enable the creation of a MST and determine the order in which edges should be added to the MST. Lowest weight edges will be prioritized.

The simplest approach to calculate the edges' weights in between two corner points is the sole usage of the Euclidean distance.
However, for certain viewing angles, this might lead to wrong neighbors edges being rated better than the correct neighbors edges (meaning, calculated weight is lower compared to true neighbors' edges). The potentially misleading Euclidean distance can be seen in Figure 4.2: The rose-colored edge diagonally spans over two different chess pieces, rendering it a faulty edge, although having a reasonable Euclidean distance.

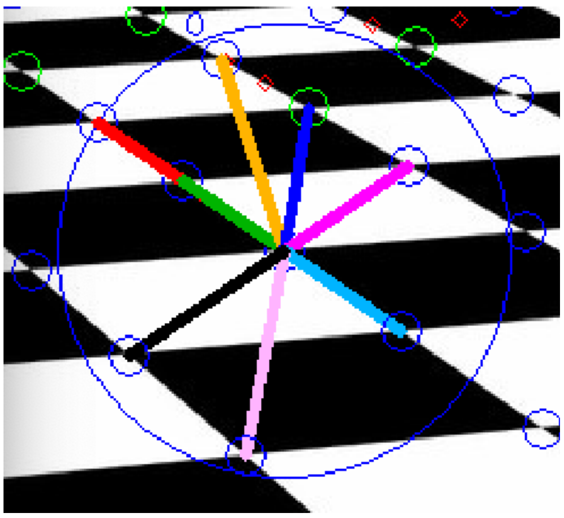


Figure 4.2: The eight closest neighbors in this angle. [25]

To solve this problem, a second heuristic is used for adding more weight to an edge. The first third and last third of the edge will be averaged in grayscale value. Additionally, an intensity difference threshold $\varDelta_I$ is chosen to compare the average intensity value of the first third of the edge compared to the last third of the edge.

The average grayscale value is chosen by sampling 15 evenly spaced points along the edge and calculating the average grayscale value out of these sampled grayscale values, thus μ is set dynamically for each edge:

$$\mu = avgGrayScale((n_1, n_2)), \text{ with } (n_1, n_2) \in E$$

Best results are reached by choosing $\varDelta_I = \mu * 0.5$ with μ being the average grayscale value for that specific edge.

If the difference threshold is exceeded in the first third compared to the last third, it can be assumed that the edge crosses two puzzleboard pieces and thus being a faulty edge. This applies to the rose-colored edge mentioned before (*seen in* Figure 4.2). Therefore, this edge will be punished by multiplying its initial weight determined by Euclidean distance by a factor p.

In Table 5.4.2 it can be seen, that $p = 2$ would be already suitable, but instead $p = 3$ is chosen, as it enables an easier threshold for the outlier detection covered in the following section.

4.6 Adding a Child Node to the MST

This section covers the actual merging of two disjoint tree structures, as described by the *Union(v, u)* operation of the union find data structure.

First, an outlier detection takes place to prohibit the merging of two trees that possibly are not actually connected puzzleboards in the image.
Next, the creation and merging process of two trees' coordinate systems is outlined.
Finally, the retrieval of the grid from the final tree is explained.

4.6.1 Outlier Detection

Given a node a in the tree T_1 and a node b in the tree T_2. The union find operation $Union(a, b)$ would try to append a to n, and thus merge T_1 and T_2.

However, in the algorithm implemented in this work, n would first check if a connection with a is allowed due to outlier constraints.

The first outlier constraint is defined as:

$$\forall n_i \in N_a : (\frac{w(a,b)}{2.9}) < w(n_i, a) < (w(a,b) \cdot 2.9) \tag{4.2}$$

with the neighborhood N_a of a defined as:

$$N_a := \{n | n \sim a\}$$

for the node a of T_1. And $w((a,b))$ yielding the weight for the edge (a,b).

The second outlier constraint is defined as:

$$(0.3 \cdot avgEdgeWeight(T_2)) < avgEdgeWeight(T_1) < (2.9 \cdot avgEdgeWeight(T_2)) \tag{4.3}$$

with $avgEdgeWeight()$ returning the average edge weight of all edges in that tree structure.

If either the first or second constraint fails, the edge (a,b) will not be added, thus T_1 and T_2 will not be merged. In every other case, they will be merged.

For various cases, using 0.3 and 2.9 as factors for the other trees and neighbors' average weight constraints (seen in Equation 4.2 and Equation 4.3), yielded best results.

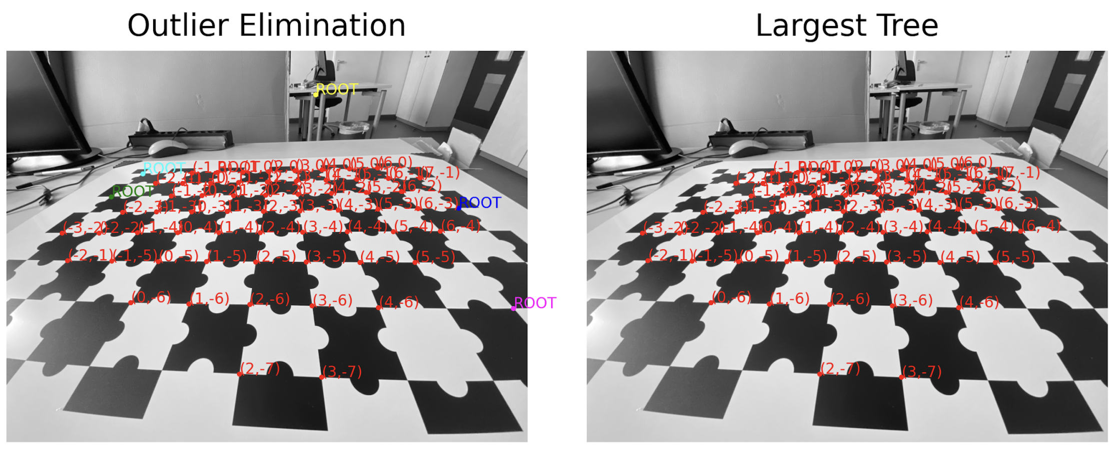


Figure 4.3: Outlier detection. Left: Showing due to outlier critera detected disjoint trees in different colors, while largest tree is colored red. Right: Just the largest tree left.

The successful outlier detection can be seen in Figure 4.3. While some of the inner puzzleboard corners are detected as outliers, the algorithm wins in robustness, by excluding corner points detected outside the puzzleboard. Notably, even some of the inner omitted corners are true outliers, since their only possible connection runs over two tiles, thus rendering them as a false edge for the MST.

4.6.2 Local Coordinate Systems

One of the challenges in giving the nodes in the tree relative x and y coordinates lies in defining in which direction relatively to its predecessor, a newly added node may lie. An example of given coordinates can be seen in Figure 4.3 of the last section. A node, relatively to its neighbor in the tree's coordinate system, has one of these four directions: $+x$, $-x$, $+y$ or $-y$. Consequently, for every node added to the MST, it is essential to determine its direction relatively to its predecessor from the four possible directions ($+x$, $-x$, $+y$,$-y$).

Simplest Approach

The simplest approach uses the absolute image coordinates of a newly added node to calculate x and y image coordinate distances and retrieve the direction with the highest difference. The retrieval of the different directions can be seen in Figure A.1.

This approach works fine for the different rotation angles of the image (*see* Figure A.4a *and* Figure A.4b), but as soon as perspective comes into play (*see* Figure A.4c), it loses its coordinate consistency.

The reason for this behavior can be seen in Figure 4.4:

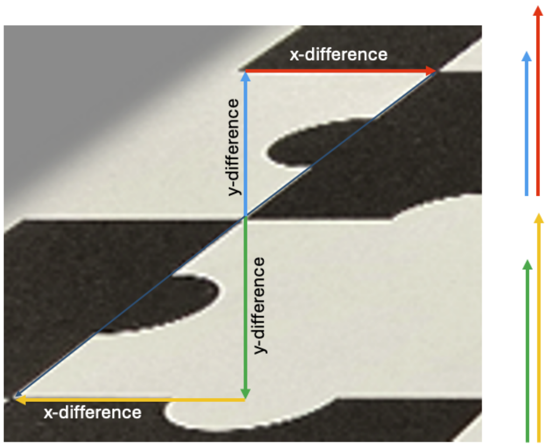


Figure 4.4: Flat viewing angle leads to greater x-distances in image coordinates for neighbors that should lie in y-direction. Red and yellow arrows show the difference in x-direction. Blue and green arrows show the difference in y-direction. Rotating the red and yellow arrows accordingly (seen on the right side of the figure) illustrates their greater length.

The x-differences are greater for the neighbors that should rather lie on the y-axis, thus yielding inconsistent coordinates, when using the simple approach to determine the appending node's position relatively to its predecessor. For this exact case, the simple algorithm first provided would yield only neighbors with x-directions even for the neighbors that lie in y-direction from a viewer's perspective (blue and green arrows in Figure 4.4).

Adapted Approach

In the adapted approach, the first added child of each node in the MST still gets its direction set with the simple approach explained above. For all the following children of a node, a new heuristic is used (*see* Figure A.3 *and* Figure A.2). This new heuristic will check for collinearity for any newly appended node to the already appended neighbors and rotate the added tree's coordinate system, if needed.

If the collinearity check passes, it can be inferred that the newly added node is on the same axis as the neighbor of the node it is appended to (*see* Figure A.2). This is especially relevant when the neighbor axes cannot be gathered from absolute image coordinates (in cases with distortion or viewing angle). The success of the adapted approach is illustrated in Figure A.5, compared to the error-prone version seen in Figure A.4c.

The two steps for the adapted approach (collinearity check and rotation of coordinate systems) will be viewed in detail in the following two subsections.

4.6.3 Collinearity Check of Nodes

For the approach mentioned in subsubsection 4.6.2, whenever a node gets appended to the tree and the tree consists of at least two nodes, it will first be checked if it is on a line *(collinear)* with any other neighbor of the tree's node and the predecessor of it (*see* Figure 4.5a). In case of distortions, this approach needs a tolerance for the decision on collinearity, since a straight line on the puzzleboard might become curved by the viewing camera's distortion.

Spanning a triangle between the three given points for the collinearity check is an easy approach of checking for collinearity of three points with a certain tolerance. The area spanned by the triangle consisting of corners A, B, C will be denoted as S_{ABC}, where every corner has image x and image y coordinates. It is calculated as follows:

$$S_{ABC} = 0.5 \cdot |A_x(B_y - C_y) + B_x(C_y - A_y) + C_x(A_y - B_y)|$$

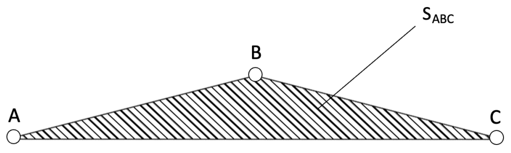
In case of the calculated area of the triangle equaling to 0, it can be assumed that the three points are collinear. Furthermore, a threshold area can be defined for offsets of the nodes, allowing for collinearity check passes, even if one of the nodes is not exactly on a line with the other two nodes. Since the S_{ABC} will always be calculated with absolute pixel coordinates, this approach works fine for images of similar resolution and distance to the puzzleboard. Consequently, higher resolution images or images of the same resolution but with different viewing distance to the puzzleboard yield much greater areas. This leads to a need for an adapted tolerance value independent on distance to board and image resolution.

To eliminate this dependency, a different approach is chosen and will be described in the following paragraph.

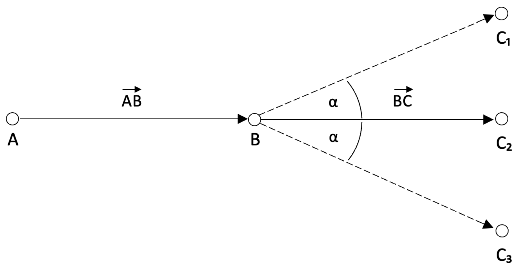
Reworked Approach for Collinearity Checks As seen in Figure 4.5b, α is the angle of interest, which will not be affected by resolution or distance to the board. It is calculated as follows:

$$\alpha = arctan2(\vec{AB}_y, \vec{AB}_x) - arctan2(\vec{BC}_y, \vec{BC}_x)$$

and as long as $|sin(\alpha)| \leq \tau$, with τ being a *threshold* set to 0.342 (equivalent to sin(20°)), the collinearity check will be considered successful. Consequently, a third node C, which is collinear with nodes A and B within this tolerance, will be aligned along the same axis that connects A to B. For instance, if A is positioned on the x-axis relative to B, and C satisfies the collinearity check, then C will also be placed on the x-axis and appended accordingly.



(a) Triangle Area Approach: S_{ABC} is the chosen tolerance.



(b) Angle Approach: α is the chosen tolerance.

Figure 4.5: Two Approaches for collinearity checks with tolerance, given the points A, B and C

4.6.4 Rotation of Coordinate Systems

Using the previously mentioned collinearity check, it is possible to determine whether the newly added node should be aligned on the same axis (either X or Y) as its neighboring

nodes. This assessment helps ensure that the placement of nodes maintains the intended coordinate relationships.
However, in certain situations, two trees in the merging process might not have consistent coordinate systems, as one of the coordinate systems might be rotated by either 90°, 180° or 270°. To solve this problem, the tree being added, needs to rotate its entire coordinate system to properly align with the new parent tree's coordinate system.

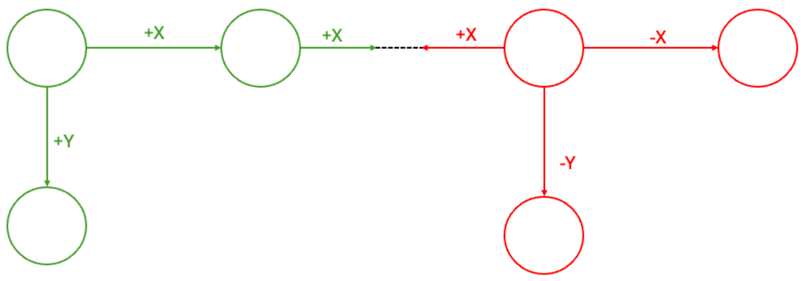


Figure 4.6: Inconsistent coordinate systems when merging. Right tree's coordinate system needs two rotations by 90° to become consistent with the left tree.

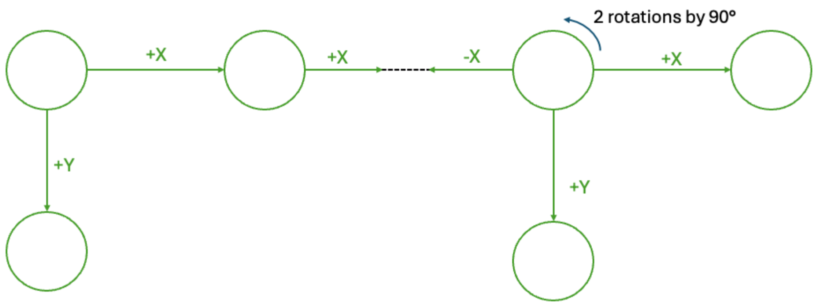


Figure 4.7: Consistent coordinate systems after correct number of rotations.

As illustrated in Figure 4.6, the tree on the right side operates under the assumption that the green node to which it connects is situated in the positive x-direction. Similarly, the tree on the left side also assumes that its connection points towards the positive x-direction. This behavior is incorrect, as for a consistent coordinate system to be maintained, the two trees should assume opposite directions for their connections.

In Figure 4.7 this conflict is resolved by rotating the tree's coordinate systems on the right side by 180°, resolving the coordinate inconsistency.

Once the desired orientation is achieved through 90-degree rotational steps, the node's rotation property will be set to reflect the total accumulated degrees (*which can be either 0°, 90°, 180°, or 270°*).

4.6.5 Coordinate Retrieval from a Node in the Tree

Each node in the tree has a vector pointing to its predecessor, indicating its position relative to its predecessor. Additionally, every node holds a rotation value in degrees. As already mentioned in subsection 4.6.4, this rotation value is calculated when one node gets appended to another one. More precisely, the root of the appended node maintains a rotation value for all its child nodes, eliminating the need to rotate the entire tree. Instead, the rotation happens when the coordinates of a node in the tree are retrieved.

To rotate a vector, a rotational matrix R_α is used. With R_α defined as:

$$\begin{bmatrix} \cos\alpha & \sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}$$ [18]

The algorithm to retrieve the relative position of a node in a tree is implemented as seen in Algorithm 2:

Algorithm 2 Get Vector to Root

```
1:  function GetVectorToRoot(node)
2:      current ← node
3:      vectorToRoot ← array[0, 0]
4:      while current ≠ current.predecessor do
5:          if current.rotation ≠ 0 then
6:              vectorToRoot ← rotatePoint(vectorToRoot, current.rotation)
7:          end if
8:          vectorToRoot ← vectorToRoot + current.vectorToPredecessor
9:          current ← current.predecessor
10:     end while
11:     return vectorToRoot
12: end function
```

For a tree structure as illustrated in Figure 4.8, retrieving the relative position of node c in the entire tree would therefore look like this:

$$c.getVectorToRoot() = \left(\left(\left(\begin{bmatrix}1\\0\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix}\right) \times R_{90}\right) + \begin{bmatrix}1\\0\end{bmatrix}\right) \times R_{270}$$

$$= \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right) \times R_{270} = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \times R_{270} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}$$

Thus, c's vector to its root in the tree is $x = 2$, $y = -1$. Therefore, to gather the position of c in the tree, the vector to its root has to be flipped, yielding $x = -2$ and $y = 1$. Consequently, when the grid has been constructed, moving two positions into $-x$ direction from root and one position to $+y$ direction, would locate c.

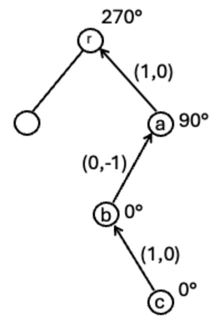


Figure 4.8: Simple tree structure with every node holding a rotation value and the vector to their predecessor in relative coordinates.

4.6.6 Merging Two Trees

For this section the tree being appended will be referred to as T_2 and the tree to which T_2 is appended will be referred to as T_1. Also, the roots of T_1 and T_2 are referred to as r_1 and r_2, respectively. After the needed coordinate system rotations by T_2 have been calculated as seen in subsection 4.6.4, r_2's rotation property will be set to the corresponding degrees.

As per definition of the union find data structure, $Union(a, b)$ will lead to the representatives of a and b to be merged. In the case of the tree structure, these are the roots of the nodes a and b. The whole merging process is exemplarily illustrated in Figure 4.9.

Initially, in Figure 4.9a, the edge chosen by Kruskal's algorithm is seen as a dashed line. The root of a is r_1 and the root of b is r_2, as seen in Figure 4.10a, implying that r_1 and r_2 are actually merged.

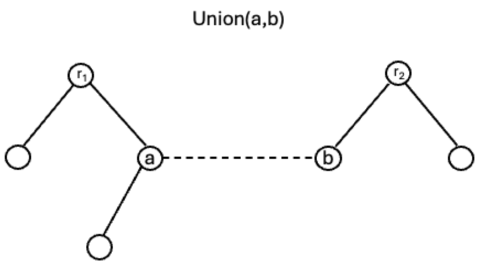
To keep the coordinate consistency after merging, the right predecessor vector must be chosen for $\overrightarrow{r_2r_1}$. Illustrated in Figure 4.9b - Figure 4.9d, this is done by:

1. Adding every *vector to predecessor* from a onwards to r_1, yielding $\vec{c}$. In the case of Figure 4.9, this consists of only one vector to predecessor.
2. Adding every *vector to predecessor* from b onwards to r_2 and rotating it by the earlier calculated needed rotations of the coordinate system, finally yielding $\vec{a}$. In the case of Figure 4.9, this also consists of only one vector to predecessor.
3. Determining in which direction relative to a, b must lie, according to the algorithm provided in subsubsection 4.6.2, thus yielding $\vec{b}$

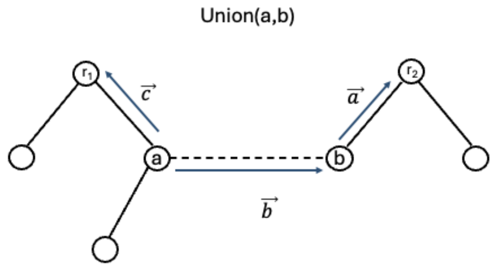
Additionally, the r_2 predecessor will be set to a reference of r_1.

Furthermore, for every node in T_2, the compressed predecessor will be eventually set to r_1's root when the $Find()$ operation is called. Thereby, keeping the union find tree structure flattened and efficient.

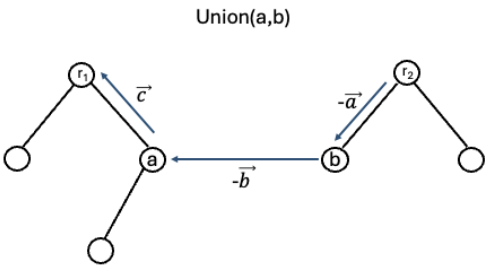
Finally, for a's neighborhood, b will be set with the earlier wanted direction $\overrightarrow{b}$. The same happens the other way around for b's neighborhood with a and $\overrightarrow{-b}$.



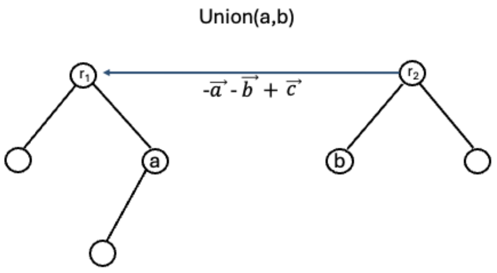
(a) Initial selected edge (a, b) chosen by Kruskal's Algorithm.



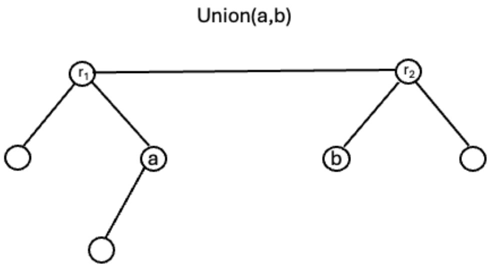
(b) The vectors $\vec{c}, \vec{b}$ and $\vec{a}$ are the vectors from a to r_1, a to b and b to r_2, respectively.



(c) At first $\vec{a}$ is rotated by the determined coordinate system rotations needed. Next, the vectors $\vec{b}$ and $\vec{a}$ are flipped to build the correct *vector to predecessor* from r_2 to r_1 as these two nodes are actually merged.

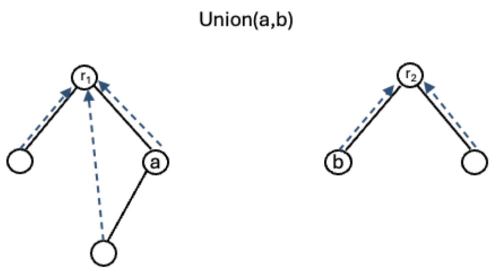


(d) Vector created by adding the flipped vectors $\vec{b}, \vec{a}$ and $\vec{c}$.

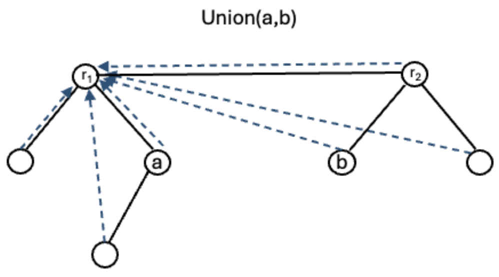


(e) Final merged tree, with r_2 holding its vector to predecessor to r_1

Figure 4.9: Merging Process of two Disjoint Trees.



(a) compressed predecessor references of every node in the trees.



(b) Eventual compressed predecessor references in the tree after successful merging.

Figure 4.10: Compressed predecessor references before and after merging

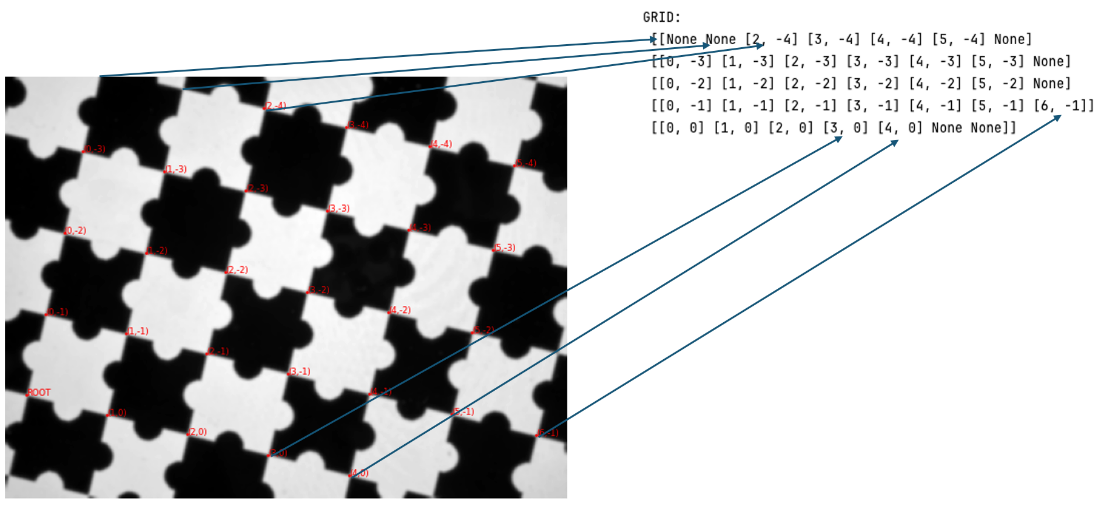
4.7 Grid Reconstruction and Edge Decoding

Once the MST has been constructed, the grid of the chessboard can be reconstructed. As the nodes of the MST hold their relative positions, neighbors now can be easily inferred, even if the nodes are not adjacent in the tree.

A 2-D array is constructed by taking the differences between the minimum x value and maximum x value in the coordinate system as well as the differences between the minimum y value and maximum y value. These are used to create the 2-D shape of the grid. The y difference will be the number of rows, and the x difference will be the number of columns of the 2-D grid.

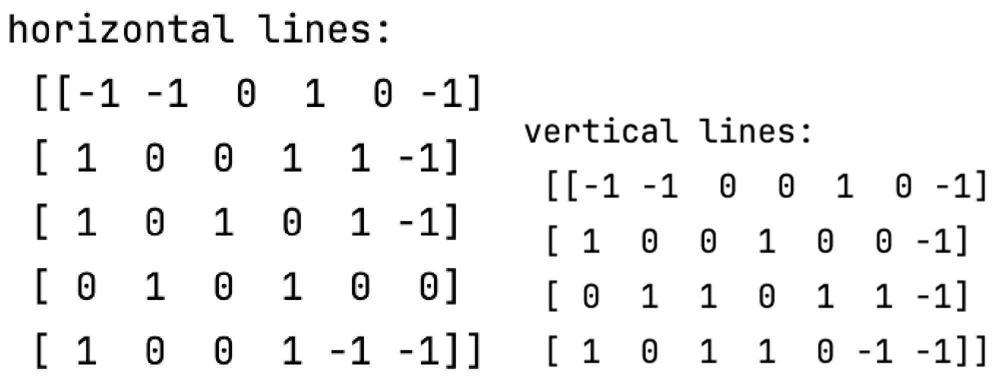
Finally, every child of the tree will get its position in the tree shifted by the minimum x and y values for its corresponding grid position. Thus, the minimum node as of y and x coordinates, will be located at (0,0) in the grid. This is exemplary illustrated in Figure 4.11.

Given the grid structure, horizontal and vertical edges can be decoded by checking if the pixel's value in the middle of two neighboring nodes is either black (*grayscale value* ≤ 0.5) or white (*grayscale value* > 0.5). For every node in the 2-D grid, its edges to its $+y$ and $+x$ neighbors are decoded, yielding all horizontal edges for all the $+x$ neighbors and all vertical edges for the $+y$ as seen in Figure 4.12 based on the detected grid in Figure 4.11.



```
GRID:
 [[None None [2, -4] [3, -4] [4, -4] [5, -4] None]
 [[0, -3] [1, -3] [2, -3] [3, -3] [4, -3] [5, -3] None]
 [[0, -2] [1, -2] [2, -2] [3, -2] [4, -2] [5, -2] None]
 [[0, -1] [1, -1] [2, -1] [3, -1] [4, -1] [5, -1] [6, -1]]
 [[0, 0] [1, 0] [2, 0] [3, 0] [4, 0] None None]]
```

Figure 4.11: Grid mapping, from tree to 2-D array. Position (0,0) for example, is *None* in the grid, because this coordinate has not been detected.



```
horizontal lines:
 [[-1 -1  0  1  0 -1]
 [ 1  0  0  1  1 -1]
 [ 1  0  1  0  1 -1]
 [ 0  1  0  1  0  0]
 [ 1  0  0  1 -1 -1]]
```

(a)

```
vertical lines:
 [[-1 -1  0  0  1  0 -1]
 [ 1  0  0  1  0  0 -1]
 [ 0  1  1  0  1  1 -1]
 [ 1  0  1  1  0 -1 -1]]
```

(b)

Figure 4.12: Decoded Horizontal and Vertical Edges. Black Circles are Decoded as 0 and White Circles as 1. Undetectable Edges are Decoded as -1.

5 Experiments

In this chapter, the experiments carried out to determine edge weights for grid reconstruction and hessian trace weighting for initial corner detection are presented.

5.1 Hardware used for Experiments

The experiments are run on a *MacBook Pro 14"* with a *M1 Pro* Processor, 16GB of RAM and a 512GB SSD.

5.2 Images used for Experiments

The images used for the experiments were shot on an iPhone 11 with a resolution of 12 Megapixels (4032x3024 pixels). They have been taken at different distances and angles to the board. The puzzleboards used in the images come in three different inner corner densities:

- **Small** puzzleboard with $11 \cdot 8 =$ **88** inner corners
- **Medium** puzzleboard with $23 \cdot 16 =$ **368** inner corners
- **Large** puzzleboard with $72 \cdot 52 =$ **3,744** inner corners

5.3 Different Factors for Trace of Hessian Matrix

The corner detector used in this thesis, multiplies the trace of the Hessian matrix with the factor k. In this experiment, different values for k are chosen, to find out which parameter k should be used for the most robust corner detection on different sample data.

5.3.1 Setup

The experiment is run with $k \in \{-2, -1, 0, 0.5, 1, 2, 3, 4\}$. For every run, a plot with the found corner points as well as the number of correct detected corner points is generated from which the best factor k for that specific image is inferred.

The dataset for this task consists of four different views (*seen in* Figure A.6).
The used images for this task contain four different settings. A simple top-down puzzleboard view, a partially occluded top-down puzzleboard view, a view with the puzzleboard and its surroundings and a view taken at a steeper angle, again mainly the puzzleboard in view.

5.3.2 Results

Image	**k**	**No. detected corners**	**% correct corners**
Top Down	$k = 0.5$, $k = 1$	368	100%
Partially Occluded	$k = 0.5$, $k = 1$	320	100%
Angled	$k = 0.5$	97	100%
With Environment	$k = 2$	379	97.01%

Table 5.1: Best Factor Candidates for Different Image Scenarios

For the top-down, angled and partially occluded views, highest numbers of correctly detected corners are achieved when setting $k = 0.5$, as can be seen in Figure A.8, Figure A.9 and Figure A.10. Opposingly, the view with environment (Figure A.11) still has a lot of outliers, when $k = 0.5$ is chosen and performs best, when $k = 2$ is set.

Since, most of the views perform best with $k = 0.5$ and the view with environment still detects all puzzleboard corners with $k = 0.5$, $k = 0.5$ should be chosen for the Hessian trace factor to achieve good results in all scenarios.

5.4 Edge Weight Punishment for Incorrect Edges

This experiment aims to determine the appropriate factor for penalizing incorrect edges in edge weighting. The goal is to adjust the weights in such a way that, despite the inclusion of these penalized edges, the algorithm still yields a correct coordinate system.

5.4.1 Setup

For this experiment three different images have been tested for coordinate consistency after constructing the MST with different weight punishments for edges that yield different gray scale values for their first third compared to their last third. When an edge is determined to be an incorrect edge, the penalty is the multiplication of its weight (Euclidean distance) by a factor p, chosen as $p \in \{1, 2, 3, 4\}$. Note, that $p = 1$ is the same as no punishment for the incorrect edge, since its weight will be multiplied by 1. The used images can be seen in Figure A.7.

5.4.2 Results

Image	**Reached Coordinate Consistency?**			
	$p = 1$	$p = 2$	$p = 3$	$p = 4$
Fisheye Image	No	Yes	Yes	Yes
Angled View	No	Yes	Yes	Yes
Angled View 2	No	Yes	Yes	Yes

Table 5.2: Different factors p for an incorrect edge's penalty and the corresponding grid construction algorithm's success in reaching coordinate consistency

As seen in Table 5.4.2, as little as a punishment of doubling the edge weight is suitable to eliminate inconsistent coordinate systems.

6 Evaluation

In this chapter, the built grid construction algorithm is evaluated based on the goals of this work.

Firstly, it is tested on its robustness of the viewing angle to the puzzleboard. Secondly, its performance on a set of distorted fisheye images is evaluated. Thirdly, its performance on partially occluded images is evaluated and finally its runtime compared to the algorithm provided in reference [12] with an opensource *C++* reimplementation is evaluated.

6.1 Performance on Viewing Angles

For the performance of the algorithm on viewing angles, a top down image is used and synthetically tilted, using Photoshop[1]. The top-down image is recorded in an angle close to 90° from the camera to the puzzleboard. It is tilted in 10° steps, yielding viewing angles for 90°, 80°, 70°, 60°, 50°, 40°, 30° and 20° respectively.

For these viewing angles, the algorithm is executed and its coordinate consistency is evaluated.

First, it is checked if all the coordinates are consistent and the corners are indexed correctly, in relation to each other.

Second, if not all coordinates are consistent, it is manually counted, how many coordinates do not fit into the constructed grid.

[1]https://www.adobe.com/products/photoshop.html

6.1.1 Results

As seen in Table 6.1.1, the total coordinate consistency is last reached at 50°. For 40°, however, there are only 6.82% wrongly indexed coordinates. For 30° and 20°, high rates of wrongly indexed coordinates are reached. The detected grids with their coordinates can be seen in Figure A.16.

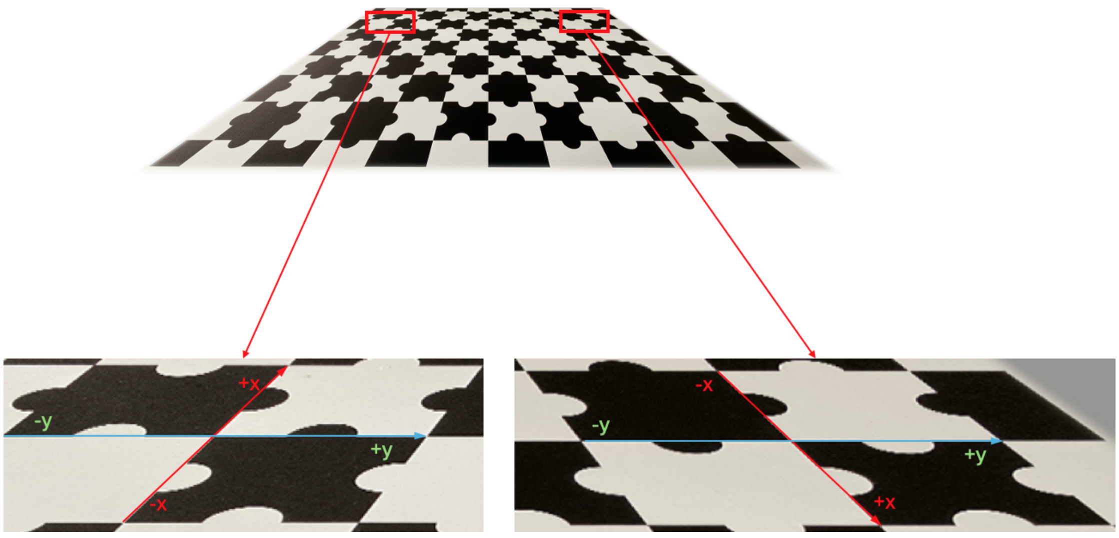


Figure 6.1: Suspected Creation of Mirrored Coordinate Systems for Narrow Viewing Angles. (40° in This Example)

One suspected reason for losing the coordinate consistency at 40° is the choice of direction for the first appended neighbor to a node: Using the absolute x and y image positions for guessing a node's direction relatively to its predecessor it is appended to, can lead to mirroring coordinate systems being constructed. This suspected behavior is illustrated in Figure 6.1: The left and right coordinate systems are consistent for the y-axis but mirrored for the x-axis due to both of its *north* facing neighbors being falsely chosen as x-direction neighbors. The false choice happens because the y distances compared to x distances in image coordinates are lower for corner points in the distance.

Since the algorithm only handles rotation but not mirroring of two coordinate systems in the merging process, this will continue as an erroneous coordinate branch of the entire coordinate system.

Viewing Angle	No. Indexed Corners	Wrong Coords
90°	88 of 88	0%
80°	88 of 88	0%
70°	88 of 88	0%
60°	88 of 88	0%
50°	88 of 88	0%
40°	88 of 88	6.82%
30°	68 of 88	52.94%
20°	12 of 88	41.67%

Table 6.1: Viewing Angle Comparison

Recalling Figure 4.4, if the *north* neighbor is connected first, it will align with the positive x direction. However, when compared to the right side of the board, a similar behavior would occur, but it would result in a negative x-direction alignment. This ultimately leads to the creation of mirrored coordinate systems, where one system requires the mirroring of x-values to maintain coherence with the other system.

Consequently, this algorithm is fully suitable for viewing angles between 90° and 50°. It is expected, that this ratio can be further improved by providing a handling for mirrored coordinate systems.

6.2 Performance for Highly Distorted Images

For this section, four highly distorted randomly picked images, taken from the deltille detector dataset[2] found on GitHub, are used to run the algorithm on. Again, it is checked if coordinate consistency is reached and how many coordinates have been detected.

6.2.1 Results

The resulting coordinate systems are illustrated in Figure A.13. In Table 6.2.1 the number of indexed corners and the percentage of these holding faulty coordinates is evaluated. Resulting from Table 6.2.1, the algorithm works well for many distorted images. In image (b) (Figure A.13b), the outlier detection leads to disjoint coordinate systems, where they rather should be merged, eventually yielding a smaller largest tree compared to the other images evaluated.

[2]https://github.com/deltille/dataset/tree/master

Image	No. Indexed Corners	Wrong Coords
(a)	80 of 88	0%
(b)	29 of 88	0%
(c)	77 of 88	0%
(d)	74 of 88	0%

Table 6.2: Evaluation of Distorted Image Performance

However, as none of the detected coordinate systems is faulty in its coordinate consistency, the algorithm is suitable for highly distorted images, as those taken by a fisheye lens.

6.3 Performance on Partially Occluded Images

One of the goals was to develop an algorithm suitable for partially occluded images. This section tests the algorithm on various partially occluded puzzleboard patterns. It is checked if all visible corner candidates are indexed consistently.

6.3.1 Setup

The images used for this evaluation can be found in Figure A.14. For image Figure A.14 (d), extra black circles have been put on the image to further occlude parts of the board.

6.3.2 Results

The resulting coordinates can be seen in Figure A.15.

Image	Percentage Correctly Indexed Visible Coordinates
(a)	100%
(b)	100%
(c)	100%
(d)	100%
(e)	100%

Table 6.3: Percentage correctly indexed coordinates of those visible

In Table 6.3.2, it can be seen that the algorithm performs well under partial occlusion. Even in a case where multiple areas are occluded (seen in image (d)), it is able to correctly index all corners.

6.4 Runtime Performance compared to Geiger et al. Algorithm

In this section, the runtime performance of the algorithm employed in this work compared to an open source *C++* implementation of the algorithm introduced in reference [12] is evaluated.

6.4.1 Setup

Notably, these runtime results have to be handled with care, since this is not the original implementation supplied by *Geiger et al.*[12]. The differences in implementation details could potentially influence the performance metrics and the generalizability of the test outcomes. An implementation of the algorithm employed in reference [12] found on github[3], is used in this thesis to execute runtime tests.

The before mentioned *C++* implementation prints debug information on runtime for finding the corners and finding the boards from these given corners. This debug information is used as the measurement for that algorithm.

For the algorithm in this work (*written in python*), python's `time.time()`[4] call is used before the grid construction commences, and after the grid construction is done. The difference of end time to start time is calculated and used as the runtime measurement.

Both algorithms are compared on the simple top-down views with different numbers of inner corners, as described in chapter 5.

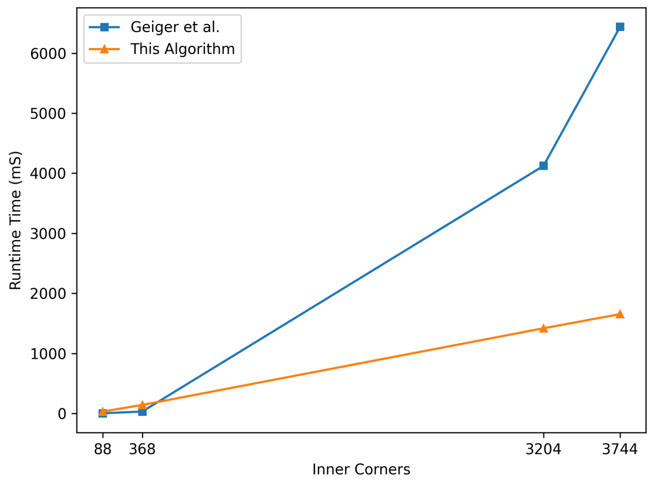


Figure 6.2: Runtime Comparison

Inner Corners Visible	Runtime Geiger et al.	This Algorithm
88	3.275 ms	32.423 ms
368	32.638 ms	142.322 ms
3204	4123.496 ms	1419.17 ms
3744	6435.432 ms	1654.045 ms

Table 6.4: Runtime Comparison

6.4.2 Results

Seen in Table 6.4.2 the runtime of the "libcbdetect" open source implementation is faster for patterns with fewer corners.

However, it can be seen in Figure 6.2, that the runtime grows much faster for the open source implementation compared to the algorithm in this thesis, when the number of corners grows.
Note that the algorithm in this work could even further benefit from a conversion into *C++* code.

All in all, it is shown that the algorithm in this work is a fast implementation, even for patterns with many corner candidates compared to the open source implementation of *Geiger et al.*[12] Algorithm[5].

[3]https://github.com/ftdlyc/libcbdetect
[4]https://docs.python.org/3/library/time.html
[5]https://github.com/ftdlyc/libcbdetect

7 Conclusion

In this thesis, a robust automatic grid construction from the puzzleboard calibration pattern has been developed. The initial corner detection is an adapted version of the Hessian detector introduced by *Liu et al.*[21]. Next, a grid construction from the detected set of corner points was developed, with evaluated robustness against distortion, different viewing angles and partial occlusion.

The corner detection employed in reference [21] has been adapted by adding the Hessian trace multiplied by the factor k to its response. The most suitable factor has been chosen to be $k = 0.5$, by running the corner algorithm for a hand-picked set of images and comparing the number of correctly detected puzzleboard corners versus corners not part of a calibration pattern.

For the grid construction algorithm based on kruskal's minimum spanning tree algorithm, a simple penalizing system for faulty edges between two corner candidates has been developed. It is based on dynamically choosing a threshold for the first third and last third of an edge candidate. This is done by sampling 15 evenly spaced points on the edge and comparing the first 5 and last 5 based on their average greyscale value compared to the average of the 15 sampled points.

Experiments on runtime performance comparison have shown the runtime of the grid construction yielding better results for higher numbers of detected corner candidates compared to the open source *C++* implementation of *Geiger et al.* algorithm on github[1].

Finally, it has been evaluated that the developed algorithm is suitable for a variety of cases. Viewing angles up to 50°, distorted fisheye images as found in the deltille dataset[2], and partially occluded images are suitable candidates for this algorithm to work on. Notably, it is assumed that an additional mirroring of coordinate systems to the already

[1]https://github.com/ftdlyc/libcbdetect
[2]https://github.com/deltille/dataset/tree/master

used rotation in the process of merging two disjointed trees will allow the algorithm to work for even steeper angles.

7.1 Outlook

The algorithm developed in this thesis is already suitable for a lot of challenging scenarios and yields robust results. Still, there are areas for further improvements. This section will provide ideas for improvements to the algorithm.

7.1.1 Runtime

As already seen in chapter 6, the algorithm yields good results for the task of grid construction. Nevertheless, the programming language of choice for this thesis was python. It was ideal for plotting purposes and object-oriented programming for an easier readability. However, this code has not been fully optimized. Translating this algorithm into a language like *Rust, Go, C or C++* will most likely be beneficial for even lower runtimes.

7.1.2 Viewing Angle Robustness

In chapter 6, it has been shown that the minimum viewing angle for a correct grid construction lies at 50 degrees. This could be further improved by adapting the algorithm's merging process and adding mirroring to the coordinate systems instead of only using rotation.

7.1.3 Outlier Detection

The algorithm developed in this thesis uses a simple outlier detection. For some cases, this still leads to faulty behavior and wrongly merged trees. In cases of overlapping calibration patterns or calibration patterns in proximity, the sole usage of average weights will not lead to disjoint trees.
Consequently, improving the outlier detection by taking more than average edge weights into account could enable for even robuster results in constructing the grid.

7.1.4 Edge Penalties

The penalty system for faulty edges is straightforward in this thesis. Even though yielding good results with the determined constraints for many cases, it still leads to faulty behavior in some cases. It has to be further analyzed to truly find the best case constraints for every situation. Additionally, further improvements could be tested: For the task of edge decoding, only $+x$ and $+y$ neighbors are relevant. Therefore, a system penalizing all other edges, which do not lie in $+x$ or $+y$ direction, to ultimately force the creation of the minimum spanning tree, with mainly $+x$ and $+y$ edges, could be tested.

7.1.5 Recovering the Position on the Puzzleboard

In chapter 4, the horizontal as well as vertical edges have been decoded. Taking any 3×3 window of vertical and horizontal edges could be used to gather the exact position of it on the whole puzzleboard pattern. An example approach for it would be the convolution of the vertical edges with the vertical DBT and the convolution of the horizontal edges with the horizontal DBT. The position yielding the highest response from these convolutions could be used to infer the position on the puzzleboard[25].

Bibliography

[1] : *Camera Calibration and 3D Reconstruction — OpenCV 2.4.13.7 documentation.* – URL https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html. – Zugriffsdatum: 2024-05-01

[2] : *OpenCV: Detection of ChArUco Boards.* – URL https://docs.opencv.org/4.9.0/df/d4a/tutorial_charuco_detection.html. – Zugriffsdatum: 2024-05-06

[3] : *What Is Camera Calibration? - MATLAB & Simulink.* – URL https://www.mathworks.com/help/vision/ug/camera-calibration.html. – Zugriffsdatum: 2024-05-04

[4] : *What Is Camera Calibration? - MATLAB & Simulink.* – URL https://www.mathworks.com/help/vision/ug/camera-calibration.html. – Zugriffsdatum: 2024-03-22

[5] *De-Bruijn-Folge.* März 2024. – URL https://de.wikipedia.org/w/index.php?title=De-Bruijn-Folge&oldid=243414257. – Zugriffsdatum: 2024-03-25. – Page Version ID: 243414257

[6] Bradski, G.: The OpenCV Library. In: *Dr. Dobb's Journal of Software Tools* (2000)

[7] Chen, Ben ; Xiong, Caihua ; Zhang, Qi: CCDN: Checkerboard Corner Detection Network for Robust Camera Calibration. In: Chen, Zhiyong (Hrsg.) ; Mendes, Alexandre (Hrsg.) ; Yan, Yamin (Hrsg.) ; Chen, Shifeng (Hrsg.): *Intelligent Robotics and Applications* Bd. 10985. Cham : Springer International Publishing, 2018, S. 324–334. – URL http://link.springer.com/10.1007/978-3-319-97589-4_27. – Zugriffsdatum: 2024-02-12. – Series Title: Lecture Notes in Computer Science. – ISBN 978-3-319-97588-7 978-3-319-97589-4

[8] FIALA, M.: ARTag, a fiducial marker system using digital techniques. In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)* Bd. 2, URL https://ieeexplore.ieee.org/document/1467495. – Zugriffsdatum: 2024-03-25, Juni 2005, S. 590–596 vol. 2. – ISSN: 1063-6919

[9] FIALA, Mark ; SHU, Chang: Self-identifying patterns for plane-based camera calibration. In: *Machine Vision and Applications* 19 (2008), Juli, Nr. 4, S. 209–216. – URL https://doi.org/10.1007/s00138-007-0093-z. – Zugriffsdatum: 2024-03-25. – ISSN 1432-1769

[10] FUERSATTEL, Peter ; DOTENCO, Sergiu ; PLACHT, Simon ; BALDA, Michael ; MAIER, Andreas ; RIESS, Christian: OCPAD — Occluded checkerboard pattern detector. In: *2016 IEEE Winter Conference on Applications of Computer Vision (WACV)*. Lake Placid, NY, USA : IEEE, März 2016, S. 1–9. – URL http://ieeexplore.ieee.org/document/7477565/. – Zugriffsdatum: 2024-02-12. – ISBN 978-1-5090-0641-0

[11] GARRIDO-JURADO, S. ; MUÑOZ-SALINAS, R. ; MADRID-CUEVAS, F. J. ; MARÍN-JIMÉNEZ, M. J.: Automatic generation and detection of highly reliable fiducial markers under occlusion. In: *Pattern Recognition* 47 (2014), Juni, Nr. 6, S. 2280–2292. – URL https://www.sciencedirect.com/science/article/pii/S0031320314000235. – Zugriffsdatum: 2024-01-19. – ISSN 0031-3203

[12] GEIGER, Andreas ; MOOSMANN, Frank ; CAR, Ömer ; SCHUSTER, Bernhard: Automatic camera and range sensor calibration using a single shot. In: *2012 IEEE International Conference on Robotics and Automation*, IEEE, Mai 2012, S. 3936–3943. – URL https://ieeexplore.ieee.org/document/6224570. – Zugriffsdatum: 2024-03-21. – ISSN: 1050-4729

[13] HA, Hyowon ; PERDOCH, Michal ; ALISMAIL, Hatem ; KWEON, In S. ; SHEIKH, Yaser: Deltille Grids for Geometric Camera Calibration. In: *2017 IEEE International Conference on Computer Vision (ICCV)*, IEEE, Oktober 2017, S. 5354–5362. – URL https://ieeexplore.ieee.org/document/8237833. – Zugriffsdatum: 2024-02-13. – ISSN: 2380-7504

[14] HORAN, Victoria ; STEVENS, Brett: Locating patterns in the de Bruijn torus. In: *Discrete Mathematics* 339 (2016), April, Nr. 4, S. 1274–

1282. – URL https://www.sciencedirect.com/science/article/pii/S0012365X15004240. – Zugriffsdatum: 2024-05-05. – ISSN 0012-365X

[15] Jähne, Bernd: Kanten, Linien und Ecken. In: Jähne, Bernd (Hrsg.): *Digitale Bildverarbeitung: und Bildgewinnung.* Berlin, Heidelberg : Springer, 2012, S. 367–394. – URL https://doi.org/10.1007/978-3-642-04952-1_12. – Zugriffsdatum: 2024-01-07. – ISBN 978-3-642-04952-1

[16] Kalaitzakis, Michail ; Cain, Brennan ; Carroll, Sabrina ; Ambrosi, Anand ; Whitehead, Camden ; Vitzilaios, Nikolaos: Fiducial Markers for Pose Estimation. In: *Journal of Intelligent & Robotic Systems* 101 (2021), März, Nr. 4, S. 71. – URL https://doi.org/10.1007/s10846-020-01307-9. – Zugriffsdatum: 2024-05-06. – ISSN 1573-0409

[17] Kang, Jiwoo ; Yoon, Hyunse ; Lee, Seongmin ; Lee, Sanghoon: Checkerboard Corner Localization Accelerated with Deep False Detection for Multi-camera Calibration. (2021)

[18] Karpfinger, Christian ; Stachel, Hellmuth: Euklidische und unitäre Vektorräume – orthogonales Diagonalisieren. In: Karpfinger, Christian (Hrsg.) ; Stachel, Hellmuth (Hrsg.): *Lineare Algebra.* Berlin, Heidelberg : Springer, 2020, S. 305–362. – URL https://doi.org/10.1007/978-3-662-61340-5_9. – Zugriffsdatum: 2024-04-23. – ISBN 978-3-662-61340-5

[19] Köthe, Ullrich: *Reliable low-level image analysis*, Dissertation, 2007

[20] Krumke, Sven O. ; Noltemeier, Hartmut: *Graphentheoretische Konzepte und Algorithmen ; mit 9 Tabellen und 90 Aufgaben.* 2., aktualisierte Aufl. Wiesbaden : Vieweg + Teubner, 2009 (Leitfäden der Informatik). – 102–106 S. – ISBN 978-3-8348-0629-1

[21] Liu, Yu: Automatic chessboard corner detection method. In: *IET Image Processing* (2015)

[22] Mehler-Bicher, Anett ; Steiger, Lothar: Augmented Reality: Theorie und Praxis. In: *Augmented Reality.* De Gruyter Oldenbourg, Juni 2014. – URL https://www.degruyter.com/document/doi/10.1524/9783110353853/html. – Zugriffsdatum: 2024-03-25. – ISBN 978-3-11-035385-3

[23] Ottmann, Thomas ; Widmayer, Peter: Manipulation von Mengen. In: Ottmann, Thomas (Hrsg.) ; Widmayer, Peter (Hrsg.): *Algorithmen und Datenstrukturen.* Heidelberg : Spektrum Akademischer Verlag, 2012, S. 403–444. – URL https://doi.org/10.1007/978-3-8274-2804-2_6. – Zugriffsdatum: 2024-04-22. – ISBN 978-3-8274-2804-2

[24] Placht, Simon ; Fürsattel, Peter ; Mengue, Etienne A. ; Hofmann, Hannes ; Schaller, Christian ; Balda, Michael ; Angelopoulou, Elli: ROCHADE: Robust Checkerboard Advanced Detection for Camera Calibration. In: Fleet, David (Hrsg.) ; Pajdla, Tomas (Hrsg.) ; Schiele, Bernt (Hrsg.) ; Tuytelaars, Tinne (Hrsg.): *Computer Vision – ECCV 2014.* Cham : Springer International Publishing, 2014 (Lecture Notes in Computer Science), S. 766–779. – ISBN 978-3-319-10593-2

[25] Schönherr, Nils: *Personal Communication.* Januar 2024

[26] Schüsselbauer, Dennis ; Schmid, Andreas ; Wimmer, Raphael: Dothraki: Tracking Tangibles Atop Tabletops Through De-Bruijn Tori. In: *Proceedings of the Fifteenth International Conference on Tangible, Embedded, and Embodied Interaction.* Salzburg Austria : ACM, Februar 2021, S. 1–10. – ISBN 978-1-4503-8213-7

[27] Stelldinger, Peer: *Personal Communication.* März 2024

[28] Walt, Stéfan van der ; Schönberger, Johannes L. ; Nunez-Iglesias, Juan ; Boulogne, François ; Warner, Joshua D. ; Yager, Neil ; Gouillart, Emmanuelle ; Yu, Tony ; contributors the scikit-image: scikit-image: image processing in Python. In: *PeerJ* 2 (2014), jun, S. e453. – URL https://doi.org/10.7717/peerj.453. – ISSN 2167-8359

[29] Wang, Shaoan ; Zhu, Mingzhu ; Hu, Yaoqing ; Li, Dongyue ; Yuan, Fusong ; Yu, Junzhi: Accurate Detection and Localization of Curved Checkerboard-Like Marker Based on Quadratic Form. In: *IEEE Transactions on Instrumentation and Measurement* 71 (2022), S. 1–11. – URL https://ieeexplore.ieee.org/document/9845472/. – Zugriffsdatum: 2024-01-05. – ISSN 0018-9456, 1557-9662

[30] Zhang, Z.: A flexible new technique for camera calibration. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22 (2000), November, Nr. 11, S. 1330–1334. – URL https://ieeexplore.ieee.org/document/888718. – Zugriffsdatum: 2024-03-22. – Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence. – ISSN 1939-3539

A Appendix

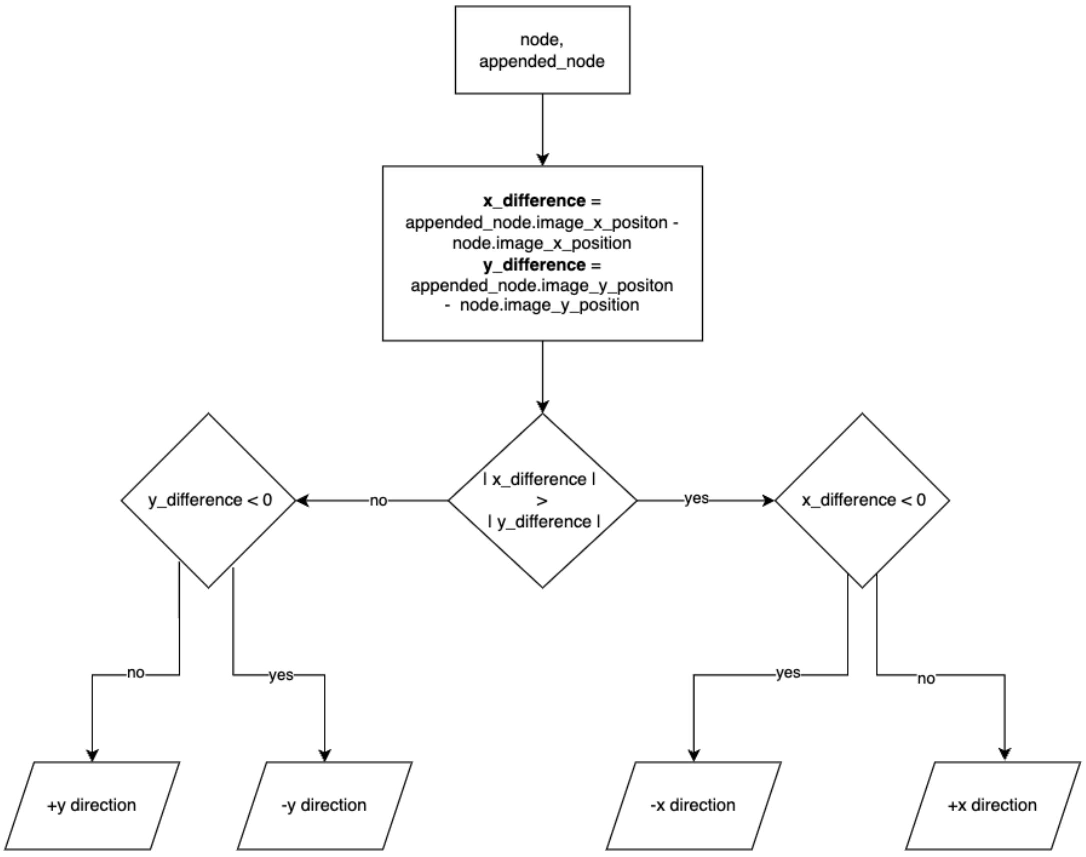


Figure A.1: Flowchart showing the simple approach, to determine the direction of a newly added node *appended_node* to *node*.

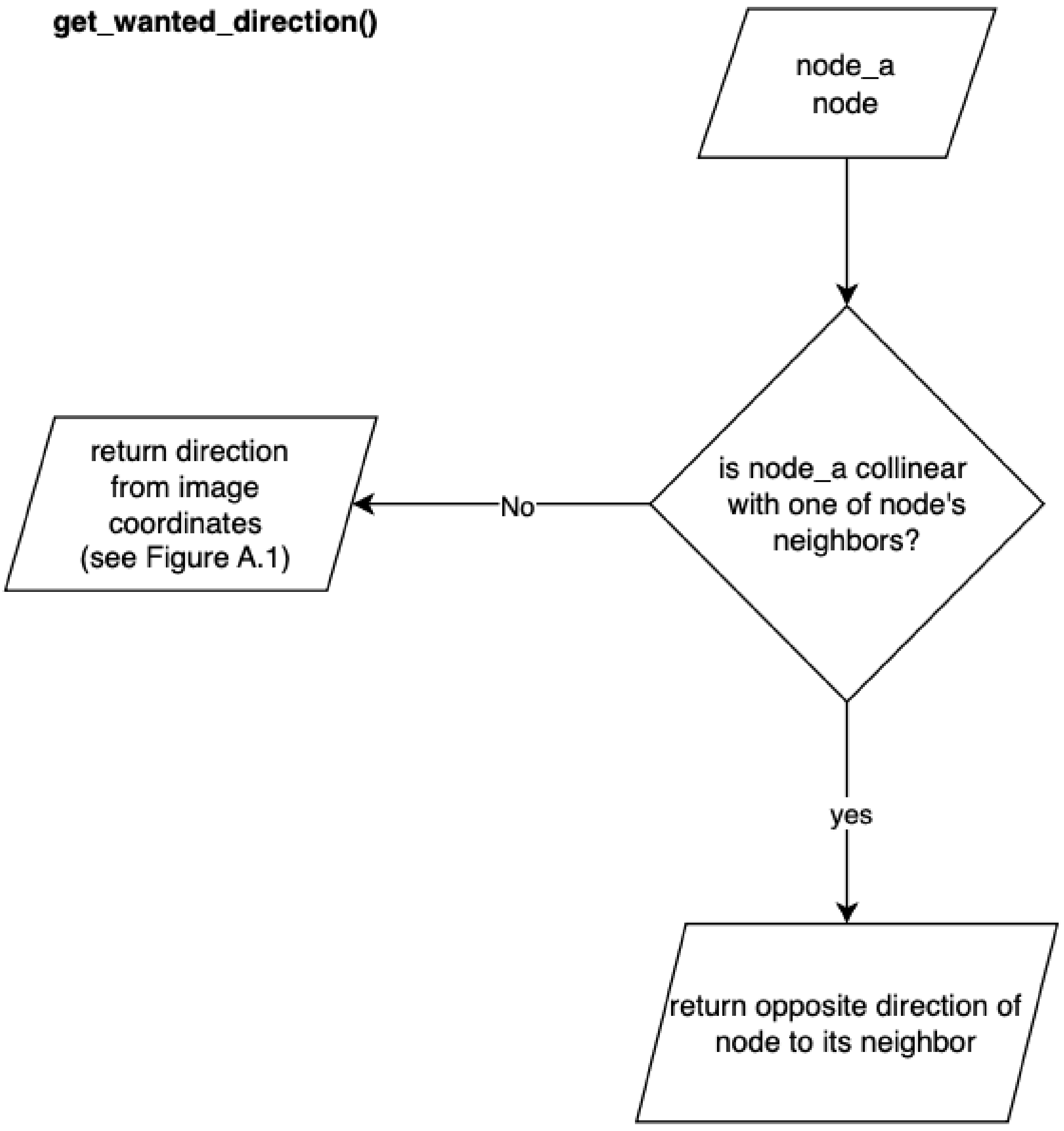

get_wanted_direction()
node_a
node
is node_a collinear with one of node's neighbors?
No
return direction from image coordinates (see Figure A.1)
yes
return opposite direction of node to its neighbor

Figure A.2: Flowchart of get_wanted_direction method.

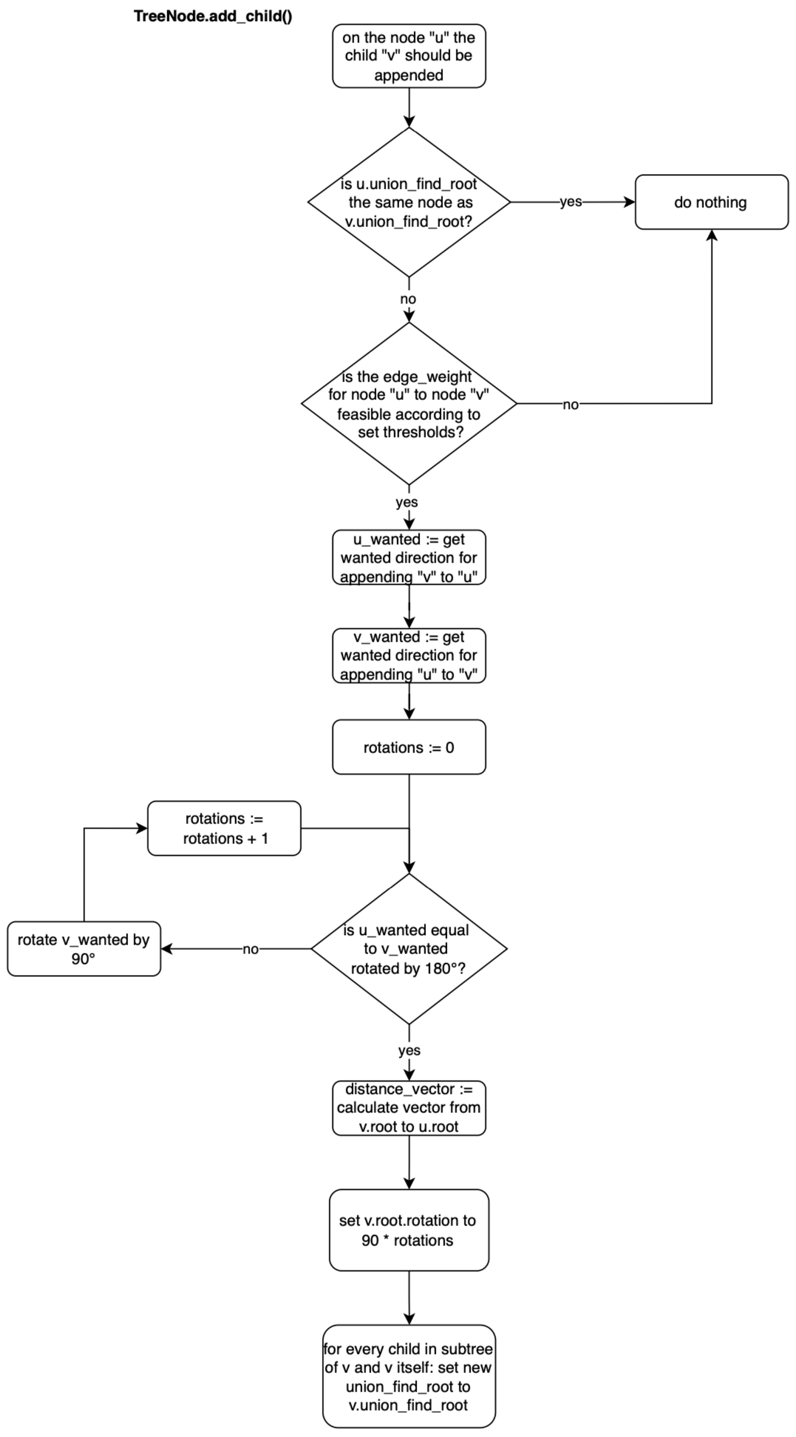
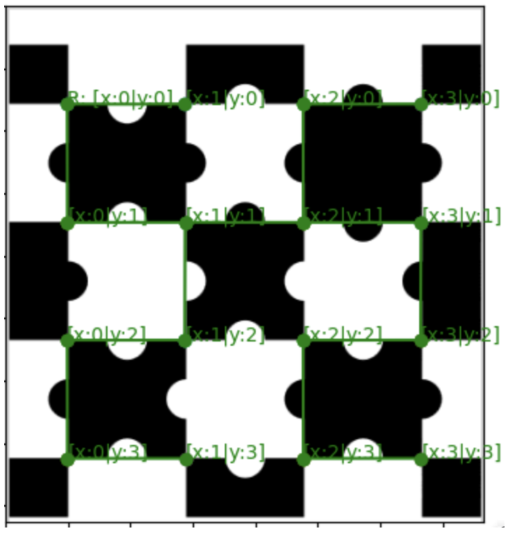
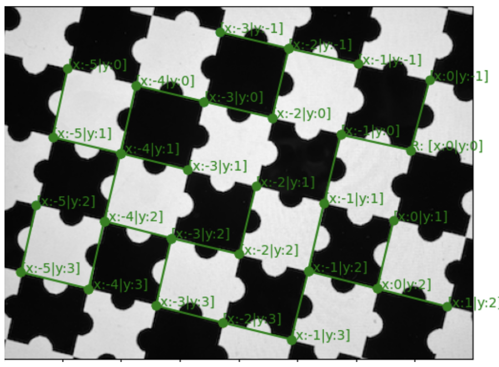

TreeNode.add_child()
on the node "u" the child "v" should be appended
is u.union_find_root the same node as v.union_find_root?
yes
do nothing
no
is the edge_weight for node "u" to node "v" feasible according to set thresholds?
no
yes
u_wanted := get wanted direction for appending "v" to "u"
v_wanted := get wanted direction for appending "u" to "v"
rotations := 0
rotations := rotations + 1
is u_wanted equal to v_wanted rotated by 180°?
no
rotate v_wanted by 90°
yes
distance_vector := calculate vector from v.root to u.root
set v.root.rotation to 90 * rotations
for every child in subtree of v and v itself: set new union_find_root to v.union_find_root

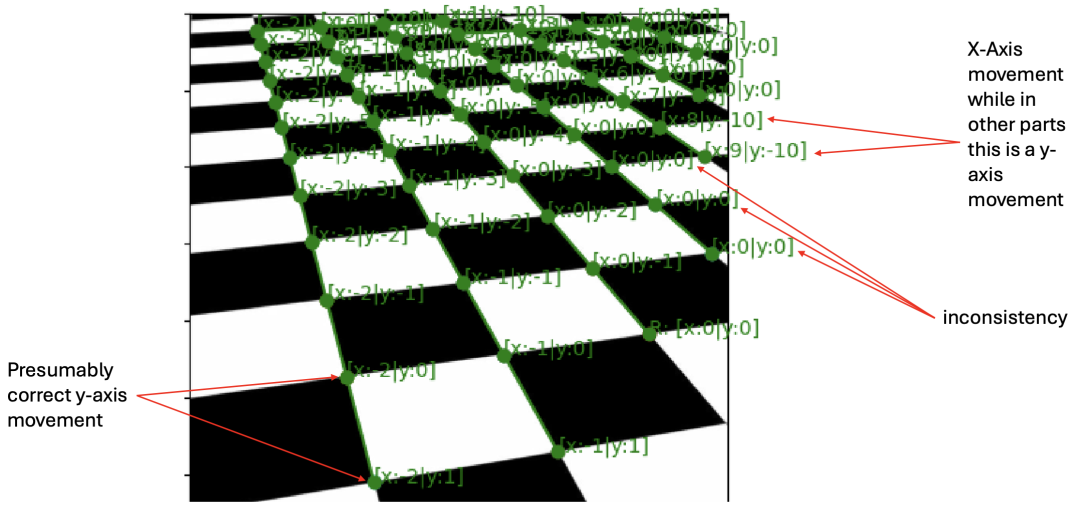
Figure A.3: Flowchart of adapted `add_child()` method



(a) Consistent coordinates throughout simple synthetic image without rotation or perspective



(b) Consistent coordinates throughout simple synthetic image with rotation but without perspective


X-Axis movement while in other parts this is a y-axis movement
inconsistency
Presumably correct y-axis movement
[x:8|y:-10]
[x:9|y:-10]
R: [x:0|y:0]
[x:-2|y:0]
[x:-2|y:1]
[x:-1|y:0]
[x:-1|y:1]

(c) Puzzleboard with perspective. Inconsistencies and errors with simple approach of coordinates.

Figure A.4: Coordinate allocation with simple approach

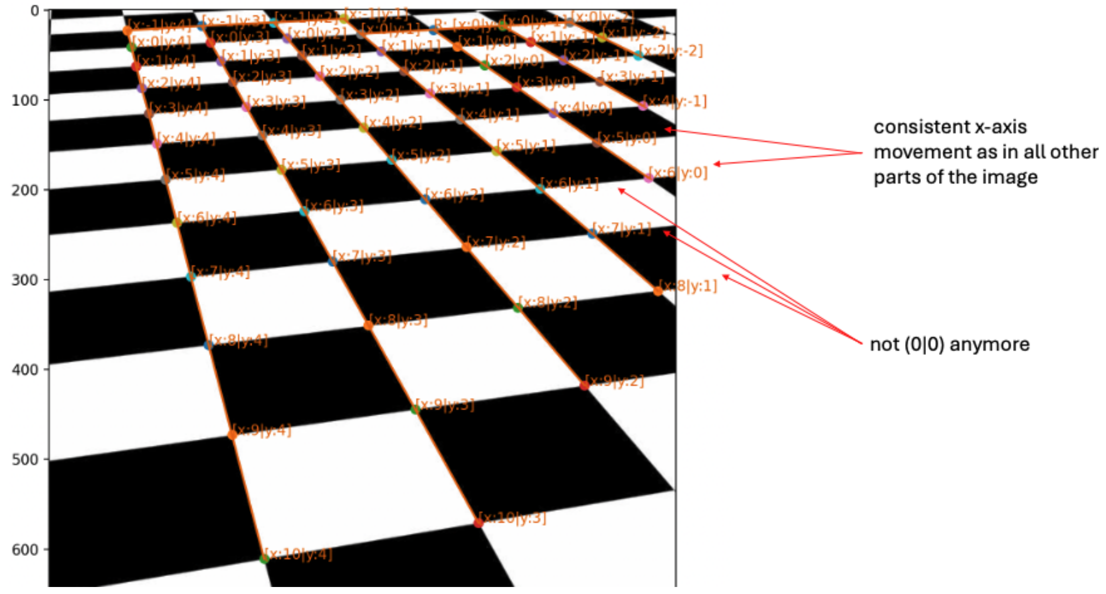
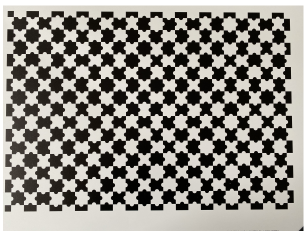
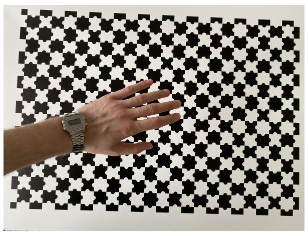

consistent x-axis movement as in all other parts of the image
not (0|0) anymore

Figure A.5: Adapted Merging Process. Puzzleboard with perspective no longer contains inconsistencies or errors in coordinates.

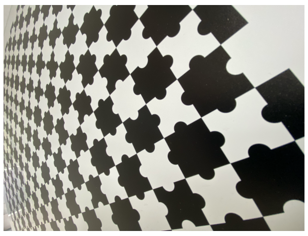


(a) Top-Down view of medium-sized puzzleboard

(b) View with environment

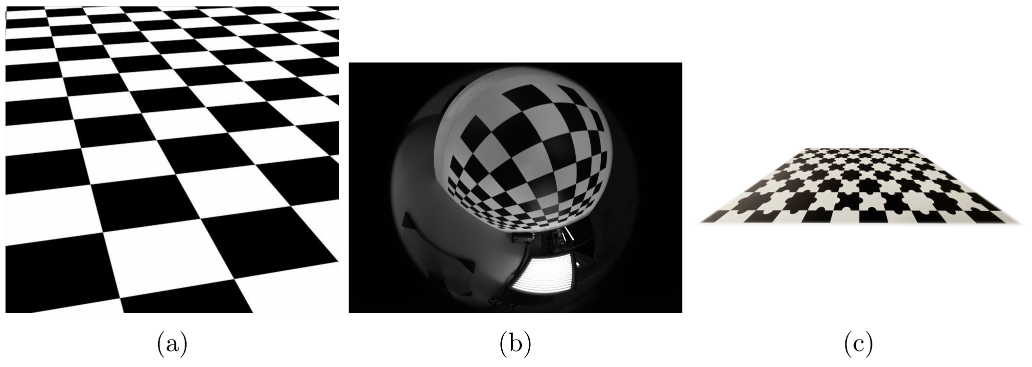


(c) View with partial occlusion



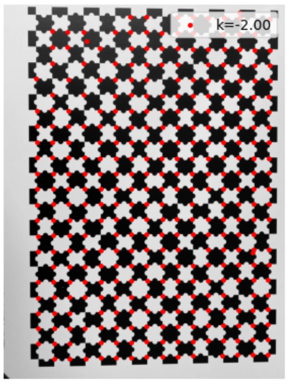
(d) Angled view

Figure A.6: The four different images used for the trace multiplication

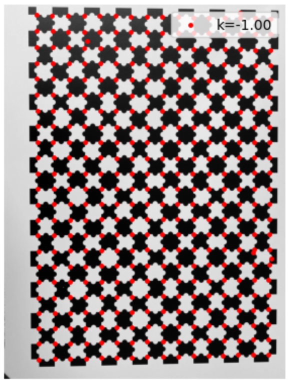


(a) (b) (c)

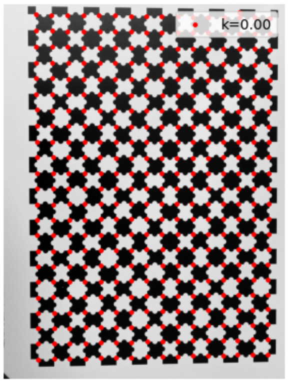
Figure A.7: The three different images used for the edge weighting.



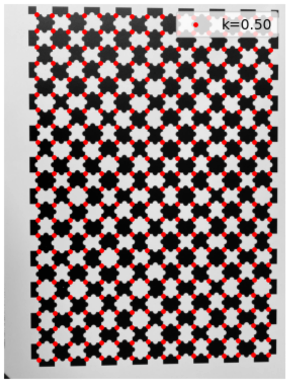
(a) k = -2; 333 Corners Detected



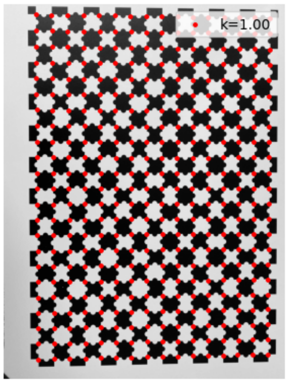
(b) k = -1; 372 Corners Detected



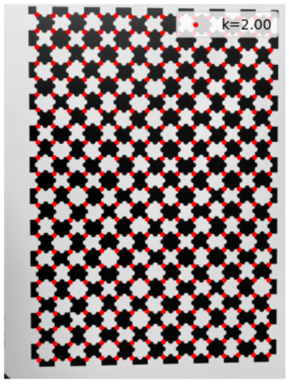
(c) k = 0; 369 Corners Detected



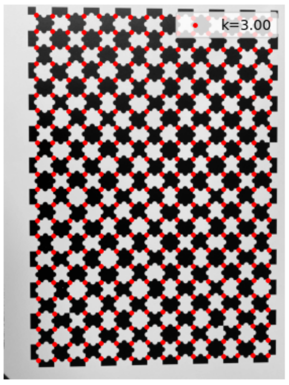
(d) k = 0.5; 368 Corners Detected



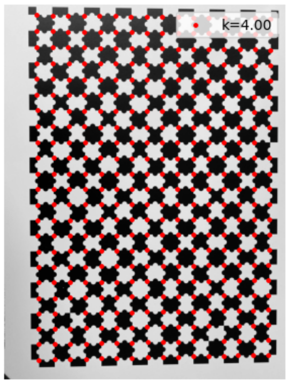
(e) k = 1; 368 Corners Detected



(f) k = 2; 367 Corners Detected

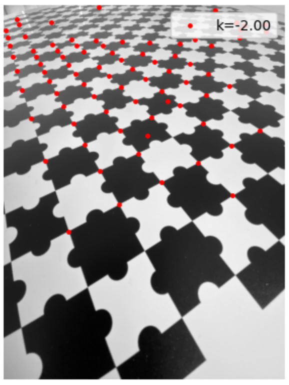


(g) k = 3; 365 Corners Detected

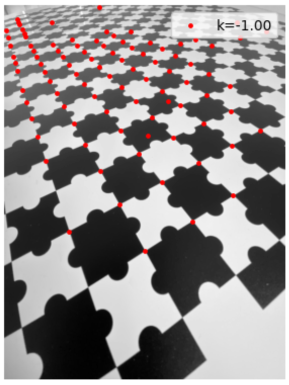


(h) k = 4; 363 Corners Detected

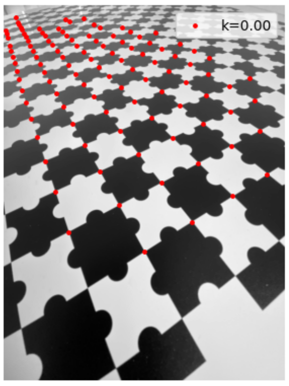
Figure A.8: Top-Down View: Plots of detected corners for different trace multiplication values k



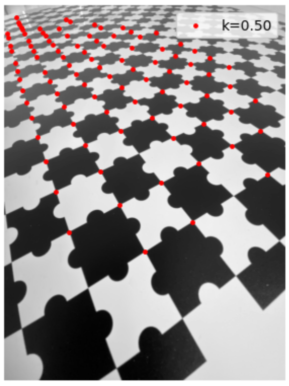
(a) k = -2, 80 Corners Detected



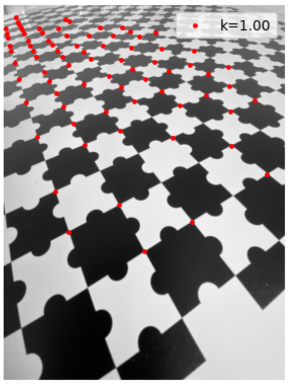
(b) k = -1, 93 Corners Detected



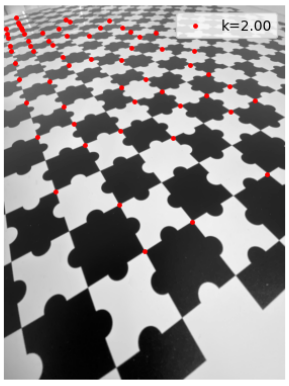
(c) k = 0, 110 Corners Detected



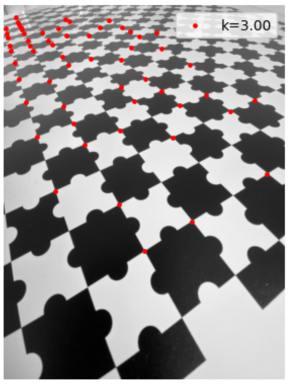
(d) k = 0.5; 97 Corners Detected



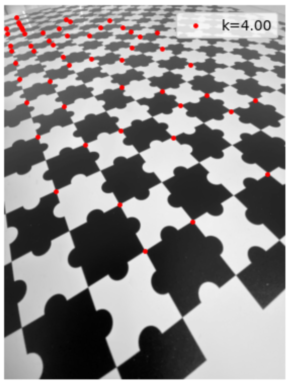
(e) k = 1; 78 Corners Detected



(f) k = 2; 61 Corners Detected

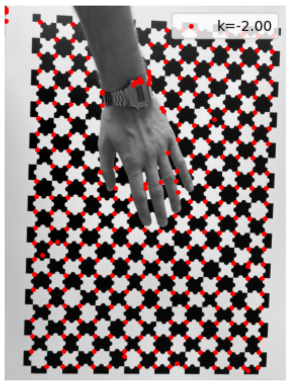


(g) k = 3; 58 Corners Detected

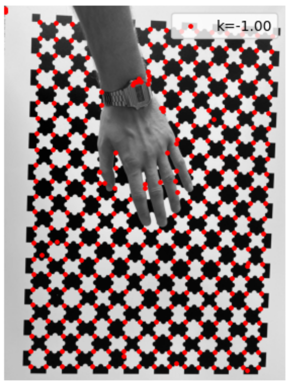


(h) k = 4; 53 Corners Detected

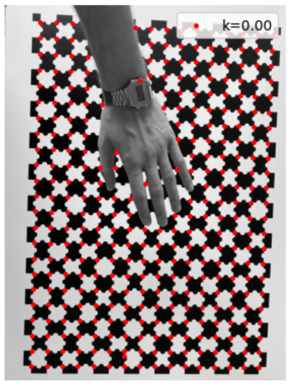
Figure A.9: Plots of detected corners for different trace multiplication values k



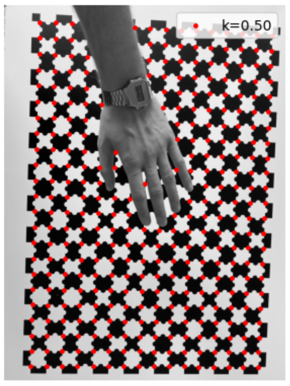
(a) k = -2; 306 Corners Detected



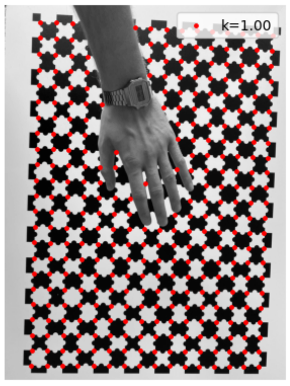
(b) k = -1; 351 Corners Detected



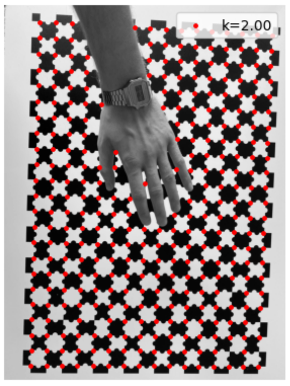
(c) k = 0; 322 Corners Detected



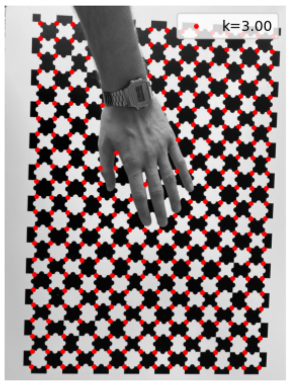
(d) k = 0.5; 320 Corners Detected

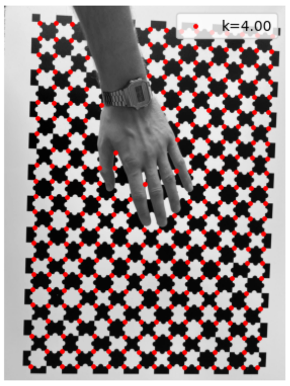


(e) k = 1; 320 Corners Detected



(f) k = 2; 314 Corners Detected

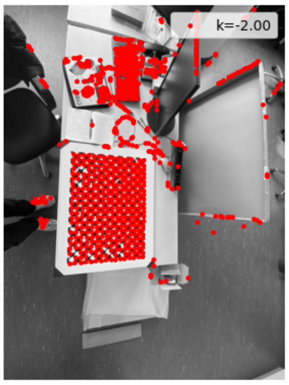




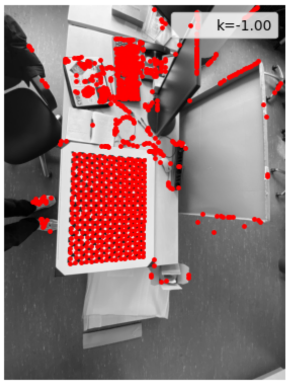
(g) k = 3; 307 Corners Detected

(h) k = 4; 298 Corners Detected

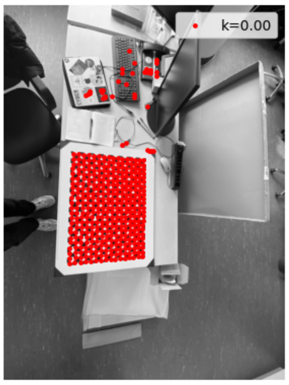
Figure A.10: Plots of detected corners for different trace multiplication values k



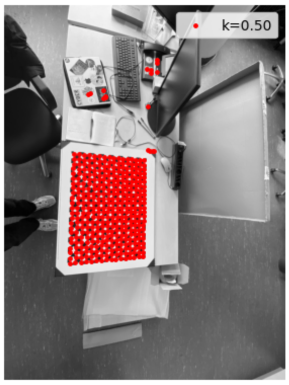
(a) k = -2; 901 Corners Detected



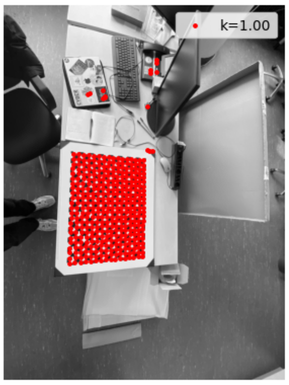
(b) k = -1; 946 Corners Detected



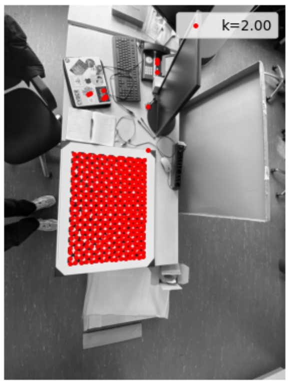
(c) k = 0; 406 Corners Detected



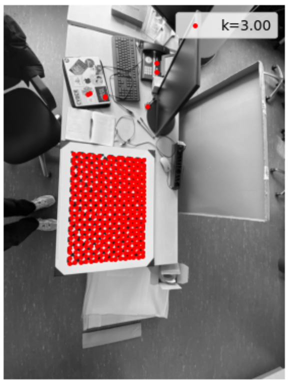
(d) k = 0.5; 385 Corners Detected



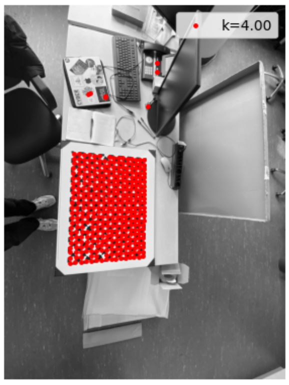
(e) k = 1; 383 Corners Detected



(f) k = 2; 379 Corners Detected



(g) k = 3; 374 Corners Detected

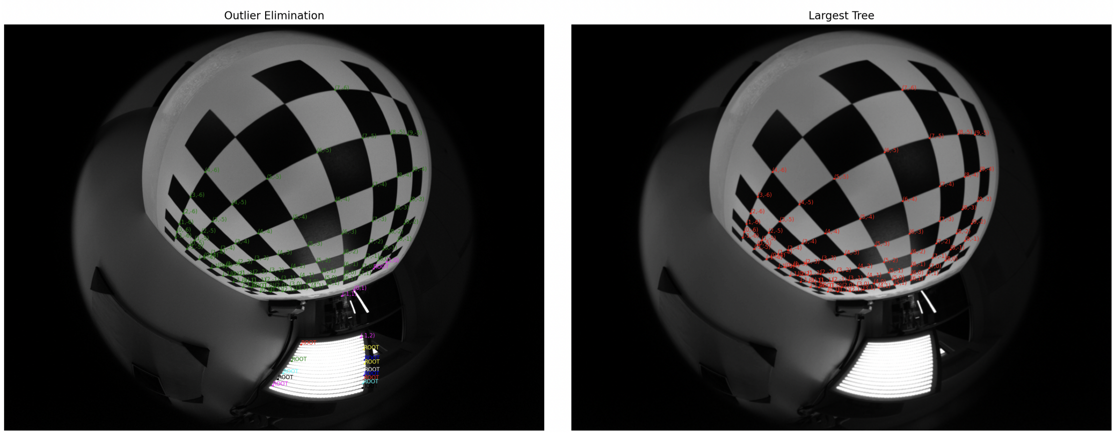


(h) k = 4; 371 Corners Detected

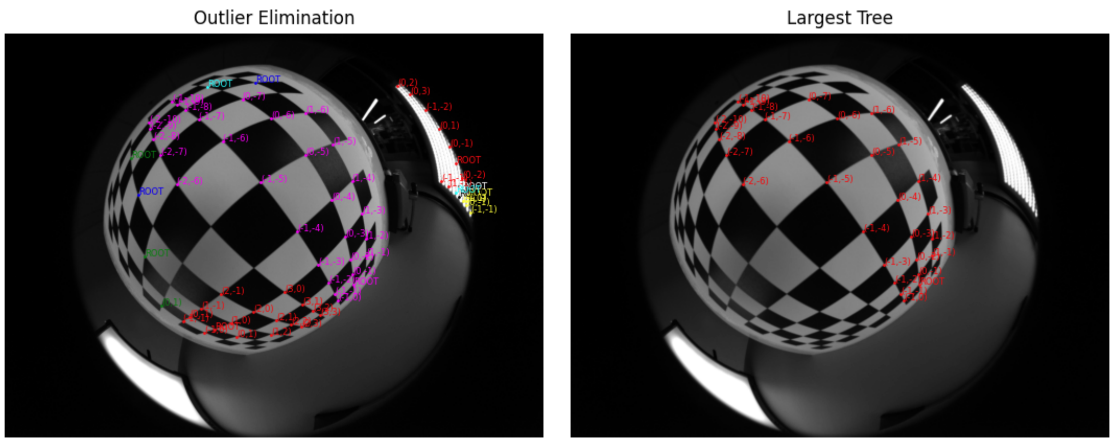
Figure A.11: Plots of detected corners for different trace multiplication values k

```
var $p$: array [$element$] of $element$;

procedure $Make\text{-}set$ ($x$ : $element$);
begin $p[x] := x$ end

procedure $Union$ ($e, f$ : $element$);
begin $p[f] := e$ end

function $Find$ ($x$ : $element$) : $element$;
var $y$ : $element$;
begin $y := x$;
   while $p[y] \neq y$ do $y := p[y]$;
   $Find := y$
end
```

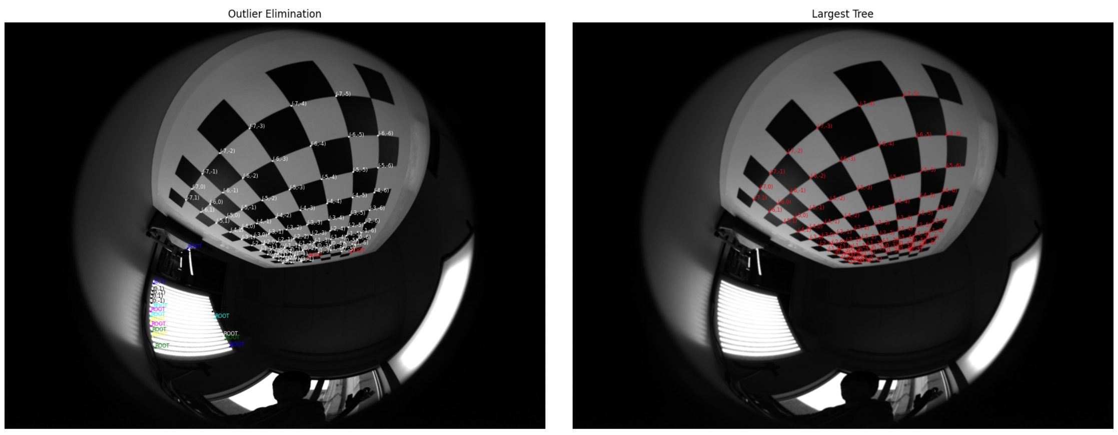
Figure A.12: Path compression algorithm for find operation in union find data structure. (Taken from [23])


Outlier Elimination
Largest Tree

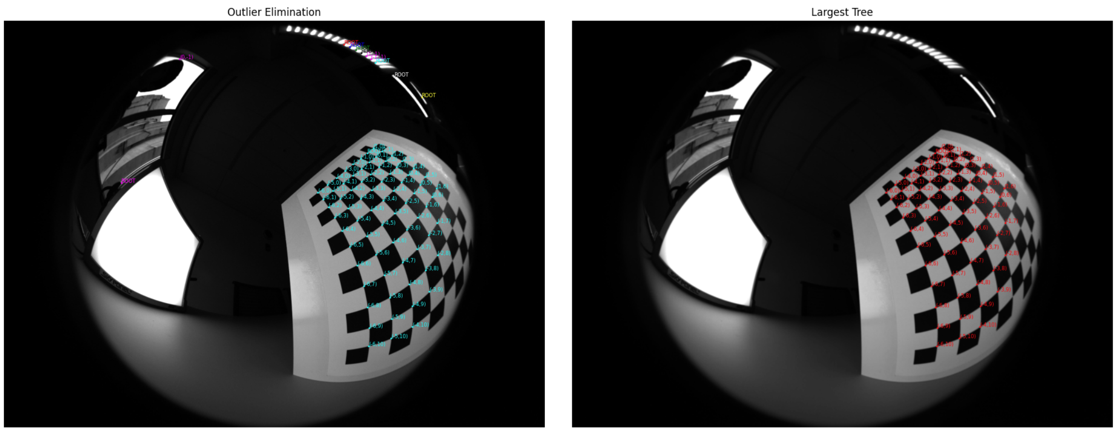
(a)


Outlier Elimination
Largest Tree

(b)

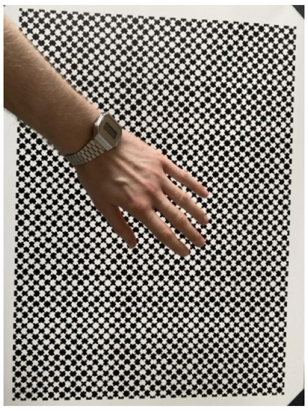


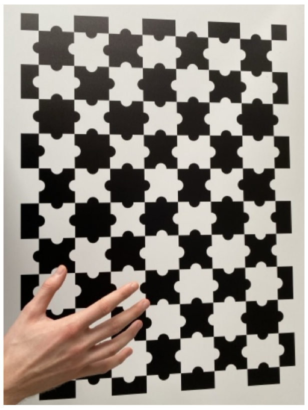
(c)

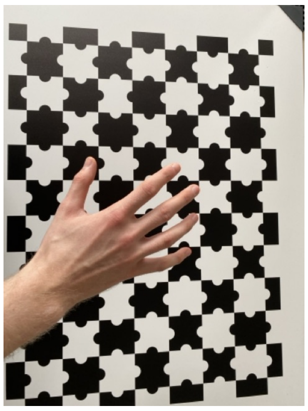


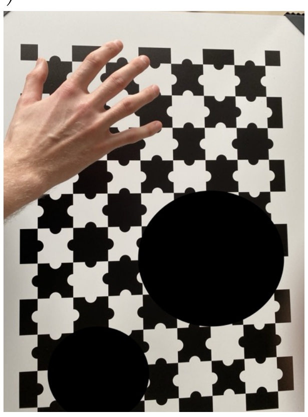
(d)

Figure A.13: Evaluation Results for Fisheye Images. Left: All Detected Trees. Right: Largest Tree


(a)


(b)


(c)


(d)

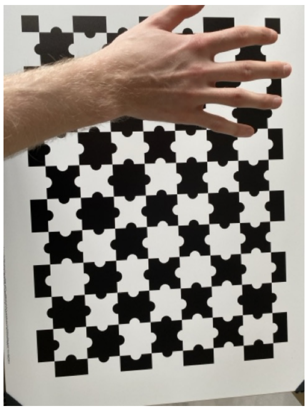

(e)

Figure A.14: Partially Occluded Boards Used for Evaluation

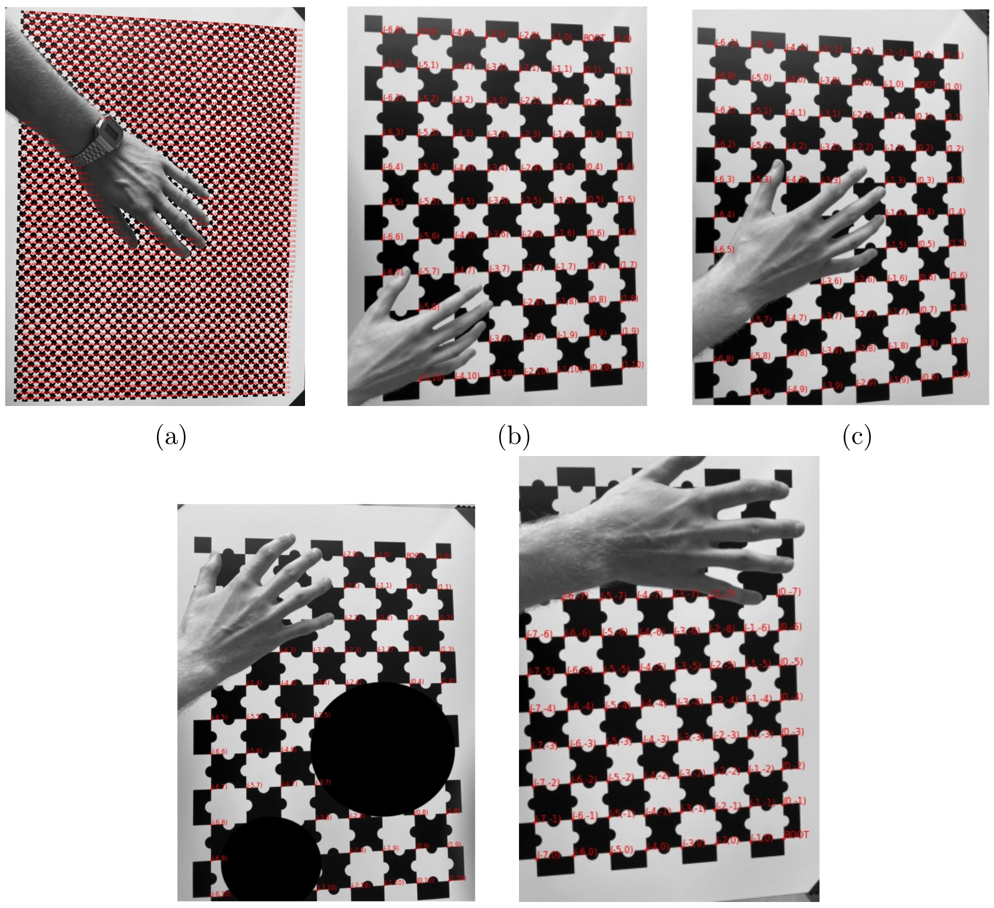
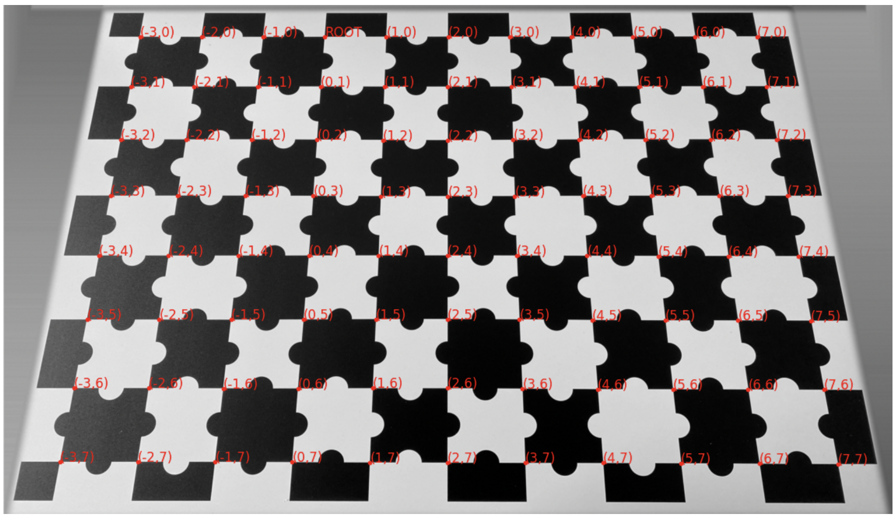
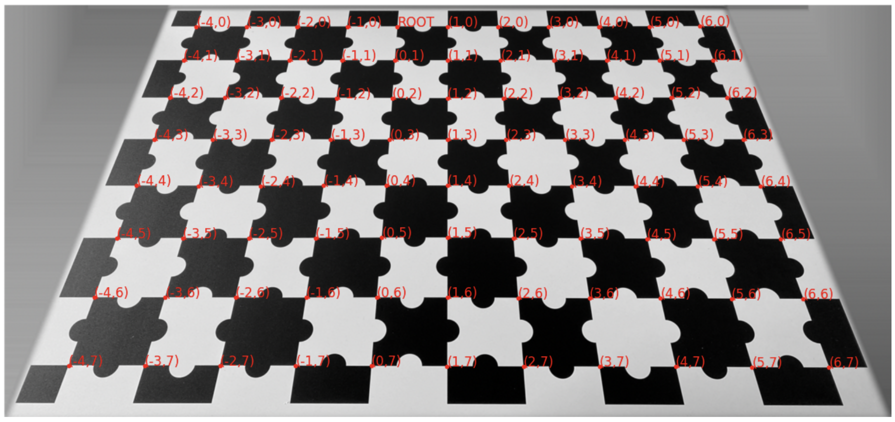


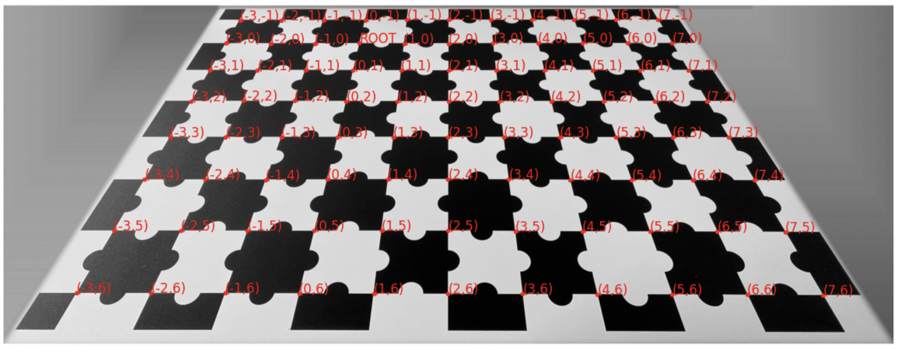
Figure A.15: Results of Partially Occluded Board Evaluation



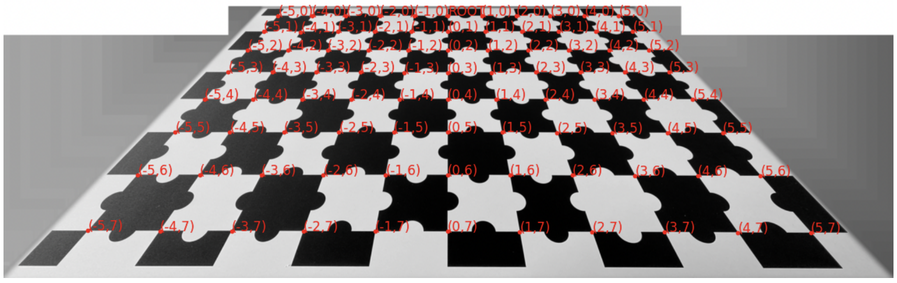
(a) 80 Degrees Viewing Angle



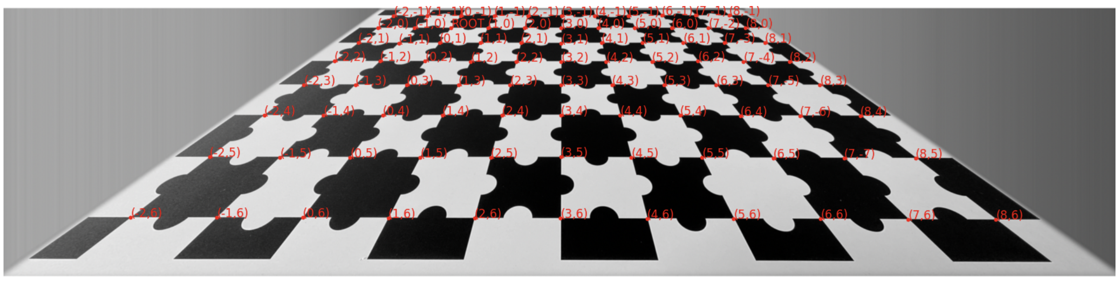
(b) 70 Degrees Viewing Angle



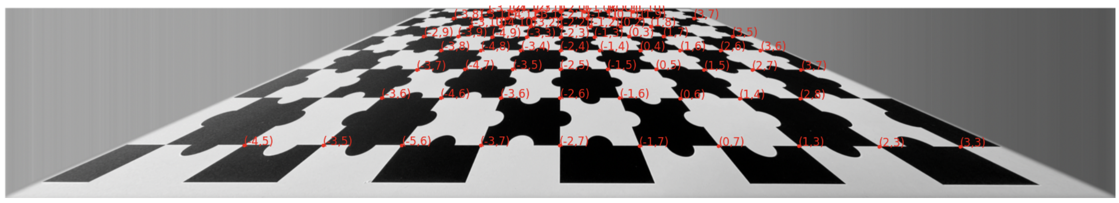
(c) 60 Degrees Viewing Angle



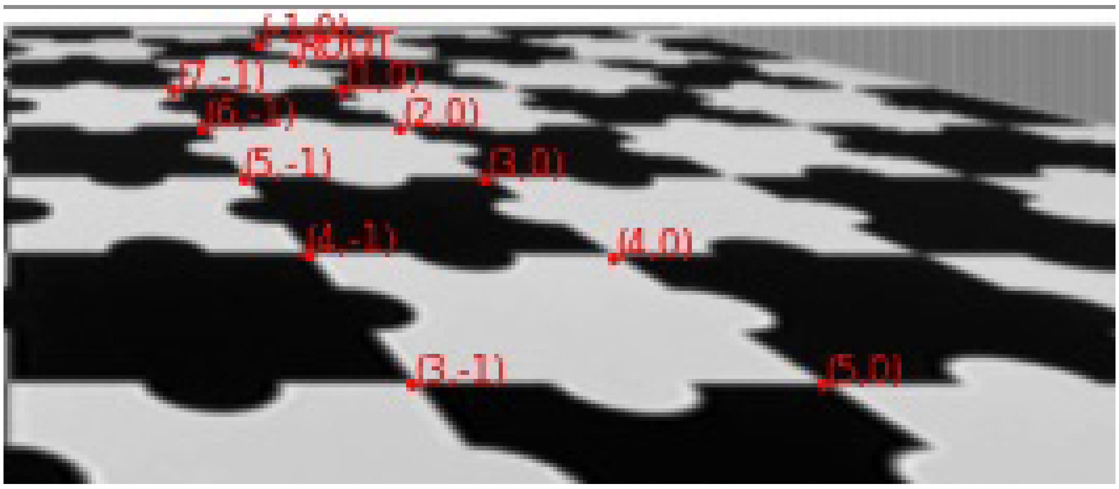
(d) 50 Degrees Viewing Angle



(e) 40 Degrees Viewing Angle. 6 inconsistent Coordinates in the second last Column from the left



(f) 30 Degrees Viewing Angle



(g) 20 Degrees Viewing Angle. Only a few corners have been indexed.

Figure A.16: Different Viewing Angles and the Corresponding Indexed Corners

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original