

FRIEDRICH-ALEXANDER UNIVERSITÄT
ERLANGEN-NÜRNBERG

BACHELOR THESIS IN COMPUTER SCIENCE

Implementation of an Android App for
the Automated Recording of Go Games
by Tracking Their State

Author:

Tilman ADLER
born 16.12.1989 in Nürnberg

Supervisor:

Vincent CHRISTLEIN

Started:

01.01.2015

Written at:

Chair for Pattern Recognition (CS 5)

Finished:

01.06.2015

Department of Computer Science

FAU Erlangen-Nürnberg



FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Richtlinien des Lehrstuhls für Studien- und Diplomarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

Erlangen, den 28. Mai 2015

Übersicht

In dieser Arbeit untersuchen wir mögliche Arten, ein Go-Brett und den aktuellen Spielstand auf einem Android Mobiltelefon zu detektieren. Wir evaluieren verschiedene Algorithmen, nämlich *Hough-Transformation* und *LSD*, um die Linien eines solchen Bretts zu erkennen. Wir betrachten *Hough-Transformation* und strukturelle Analyse um die Steine, die darauf liegen, zu finden. Wir verwenden dann die Schnittpunkte der Linien und die Lage der Steine um das gesamte Brett zu extrapolieren und seine Orientierung herauszufinden. Abschließend analysieren wir das Bild an den hochgerechneten Koordinaten um zu detektieren, ob und welche Steine dort liegen.

Wir beschreiben weiterhin, wie wir Parameter für verwendete Algorithmen wählten und unser Programm mit einem Satz dazu erstellter Bilder testeten und vergleichen die verschiedenen Algorithmen.

Abstract

In this work we examine possible ways to detect a Go board and the current game state on an Android powered mobile device. We evaluate different algorithms, namely *Hough Transformation* and *LSD*, for detecting the lines of such a board. We look at *Hough Transformation* and structural analysis for finding pieces that lay on it. We then use the intersections of the lines and the location of the pieces to extrapolate the whole board and find its orientation. Lastly we analyze the image at the extrapolated coordinates to detect if a piece is present there and of which color it is.

We further describe how we chose parameters for the used algorithms and how we tested our program using a set of images that we created for that purpose. Lastly we compare the different algorithms.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Related work	2
1.3	Related patents	4
2	Detecting the board	5
2.1	Preprocessing	6
2.2	Detecting visible intersections	7
2.2.1	By intersecting lines	7
2.2.2	Using corner detectors	9
2.3	Detecting occluded intersections	9
2.3.1	Preprocessing	9
2.3.2	Using Hough circles	10
2.3.3	Using contours	10
2.4	Extrapolating the board	11
2.5	Refining intersection locations	13
2.6	Classifying intersections	14
2.7	Postprocessing	14
3	Creating the Android app	17
3.1	Tethering in the framework	17
3.1.1	Adding the source dependencies	17
3.1.2	Gathering data	18
3.2	Using the detector	19
3.2.1	Setup of the components	19
3.2.2	In- and output to the detector	19

3.2.3 Using the detection result	21
3.3 User interaction design	21
4 Evaluation and optimization	23
4.1 Dataset	23
4.2 Visible intersections	25
4.2.1 Optimizations on the training set	25
4.2.2 Performance on the test set	28
4.3 Occluded intersections	30
4.3.1 Optimizations on the training set	30
4.3.2 Performance on the testing set	32
4.4 Pre- and postprocessing	32
4.4.1 Rectifying perspective distortion	32
4.4.2 Removing specular highlights	34
4.4.3 Removing large outliers during detection	34
4.5 Speed on an Android device	35
4.5.1 Getting a baseline	35
4.5.2 Comparing the runtime of different approaches	35
4.6 Overall performance and live experiences	36
5 Conclusion and outlook	41
5.1 Conclusion	41
5.2 Outlook	42
A Simple mount for Nexus 4	43
List of Figures	45
List of Tables	47
Bibliography	49

Chapter 1

Introduction

1.1 Motivation

The game of Go (Japanese 囲碁) is ancient and its “origin [...] is lost in obscurity. [...] The traditional account that has won most credence ascribes the invention of Go to the [Chinese] Emperors Yau and Shun” [Mih39] (~2300 BCE). But even though its rules are simpler than the rules of chess it is — outside of Asia — not nearly as wide-spread as the latter. When learning Go one comes to a certain point where one single error often decides the whole game. Often the winning player will be able to remove many of his or her opponent’s pieces from the board, resulting in a devastating defeat. This can be quite frustrating. Other mistakes do not manifest until some moves after they have been made, which makes it hard to learn from them.

The solution to this is obvious: Record a game and analyze it later, maybe resume it at the questionable move and see if it would have turned out otherwise. While there are advanced players who can remember an entire game after playing it especially new players have to rely on a notation with pen and paper, pocket computers or mobile apps. As this process drains on concentration many players tend to not note games.

There are ways to save board games in an automated fashion, not requiring any interaction by the players. For chess there are specialized boards (DGT chess boards) starting at around 20€, which usually have special (magnetic) pieces [Bul01] or small holes with light sensors. They record a game and send

the data to a computer, mobile device or other hardware for display and analysis. However for Go there is — to our knowledge — no such device on the consumer market. Also, the need for specialized hardware makes a solution like this very unappealing.

What makes the situation worse is that while there is a notation called Kifu (Japanese 棋譜) for the recording of games it is — again outside of Asia — not very widespread. No numerical system as in chess (where the columns and rows are determined by a number and a letter) has been generally accepted. Ideally a recording system would save the moves in an independent fashion that makes it easy to transfer into the existing systems' formats. Another option would be to choose one system and save it in its data format. Once such a system exists there are other applications that can be thought of. For example playing against a computer or via the internet another human and using real pieces in the process. Anyhow it might lead to more players successfully learning how to play Go (well) and thus contribute to its spread in the western world, if just a little.

The ubiquity of mobile devices with built-in cameras lets one solution appear very attractive: Have a mobile application record the game, analyze it via a computer vision application and save the result when it detects a new move. This work intends to provide such a solution and show the efforts taken to create it.

After presenting related work and possibly related patents we will first describe the part of the application that does the actual detection and go on to the integration of it in an Android app. Thereafter we will discuss the impact of different algorithms that have been evaluated regarding detection quality and speed. We also provide an assessment to what extent the goal of an automated recording device for Go games has been reached. We will conclude with an outlook on possibilities for future work improving upon the solution at hand and a summary of this work.

1.2 Related work

The detection and augmenting of board games in general is illustrated by Eray Molla and Vincent Lepetit [Mol10], who demonstrate the successful tracking of colored pawns on a Monopoly™ board.

More closely related is Steven Scher's, Ryan Crabb's and Jamie Davis's work [Sch08] who collect samples of a Go-board using a fixed camera in regular intervals. They detect the grid in the first image using *Hough transformation* and assume it to be fixed. Afterwards they classify intersections (as "lack", "white" or "empty") by template images and use a Markov model to remove errors in the single images. To this, they model the detected state of every image and transitions in between according to the game rules.

This approach yields high accuracy when capturing whole games and performing the detection asynchronously. On a mobile device this is not ideal as it drains the battery at a time the user does not expect it to. Also movement of the camera or the board is not supported at all as detection of the grid is only performed on the first image. However smart phones as a multi purpose device should be relocatable at least within a certain range. The user, for example, might have to take a call and then replace the phone. Thereafter the grid might not be detectable as easy any longer.

Similarly Teemu Hirsimäki [Hir05] uses the *Hough space* to find parallel lines near the center of the image. He will then use this information in conjunction with the original image (filtered for edges) to build the board grid by solving a minimization problem on them. Intersections are classified by relating the median brightness to the median value in a certain window.

The author assumes that the approach can be used on consecutive frames, but one problem remains when using only line detection: during later phases of the game many pieces will be placed especially in the center of the image. The lines are occluded by tokens and there is no way left to detect the board.

Alexander K. Seewald [See10] solved this by detecting the intersections using the *SIFT algorithm*. His approach specifically targeted endgame situations. The keypoints are then classified (as before but additionally as "empty inside the grid", "empty on the edge" and "empty on the corners") by an SVM. The results are then further refined by estimating the board position using the corners and classifying the previously missed tokens by average brightness.

While he solved the problem of detecting end game states this came at the price of performance. With five seconds runtime on a PC for an 8 by 8 grid, this solution seems unpractical for slower mobile processors.

1.3 Related patents

There are several patents which protect apparatuses for the detection of Go games mostly from Japanese inventors. Most of them rely on some hardware device to detect where a piece has been placed. There are numerous approaches in this direction, for example using magnetic pieces [HIR99] or pressure sensitive mats [MAM02].

There are also some patents for work including a camera to record the game. A device from 2004, for example, uses a CMOS camera connected to a PC [Tsu04]. The camera is mounted on a pole next to the board which is permanently connected with the board, i.e. the camera angle is fixed. Another apparatus also uses a permanently fixed camera connected to a table [AI92]. Here, the player can even play against a computer controlled robot arm. All those approaches use proprietary hardware and fixed setups, though.

Chapter 2

Detecting the board

The lines of a Go board catch one's eye first and foremost when looking at it. However, relying solely on lines for the detection of the grid will probably fail in situations when too many tokens occlude them. Therefore, our algorithm uses them only to find intersections and tries to fill in the gaps by adding information about pieces. The reason behind this is that what is actually interesting are not the lines, but their intersections and a piece will always lie reasonably close to one. The only user interaction happens in the beginning of a recording session. The user has to place the camera so that the center of the board aligns with the center of the camera.

The implementation¹ is based on the OpenCV framework (version 2.4.10). If not noted otherwise all detections have been performed on an x86_64 architecture. As described in chapter 3 this does not give different results than the execution on mobile devices (i.e. ARM architecture), for example because of differences in floating point calculations.

In short we do the following steps:

1. roughly pre-segment the board by analyzing connected components around the center of the thresholded input image
2. detect horizontal and vertical lines and intersect them
3. detect pieces on the board and consider their centers to be intersections, too

¹The source code is GPLv3 licensed and can be freely accessed at <https://github.com/t-animal/GoBoardReader>

4. remove duplicates
5. select a few intersections around the center of the image
6. build a submodel of the board by estimating where each selected point lies on the grid
7. calculate their position in space using RANSAC and apply the resulting transformation matrix to a complete model of the board
8. refine the locations by shifting extrapolated intersections to measured ones and redo the calculations
9. extract piece color information by the amount of black pixels in a window around each intersection in a thresholded version of the input image

2.1 Preprocessing

Mobile devices do not have the same computing power as desktop hardware. Therefore our first step is to reduce the image size without losing relevant information. To do so, we threshold the grayscale input image using a mean adaptive threshold with a low constant value C and a window size of what we expected to be roughly the width of one square on the board (approximately 45px, as measured in one of our sample images).

Assuming that at a least part of the board is in the center of the image we can segment it from the background with high confidence by a simple connected-component analysis with a black rectangle in the image center. On our test case this fails only in one situation in which the board lies in grass in the evening. Shadows connect the board with the background and the whole image is segmented. On a flat surface this is not a problem and we found no image where we crop off too much and thus make the board undetectable.

This step does not just improve speed but also detection performance because interfering background information like patterned wood table tops can be cut off.

2.2 Detecting visible intersections

2.2.1 By intersecting lines

Having cropped the image to the board we use line detection to find the visible grid lines. Our goal is to detect the intersections and the most obvious approach to find them is to detect the visible line segments, continue them into infinity and intersect the results. This way all visible intersections should be detected plus some occluded intersections where enough of a line is visible, for example, because only one piece lies on it. When intersecting those lines that are split into several line segments by a piece, duplicates arise. We filter those in the intersection step rather than trying to find line segments that belong together beforehand.

We evaluated two different well known line detection methods.

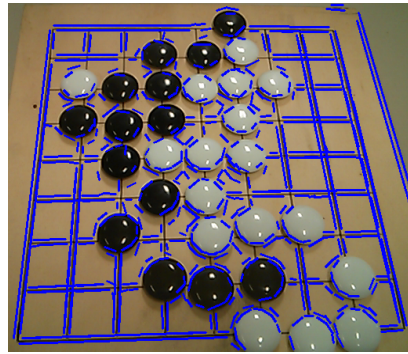
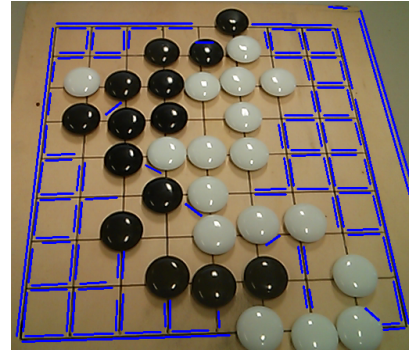
Using Hough lines detection

Our first approach was to use *Probabilistic Hough Line Transform* after applying a *Canny edge detector*. We furthermore found, that blurring the image with a light *Gaussian* improves detection quality and speed.

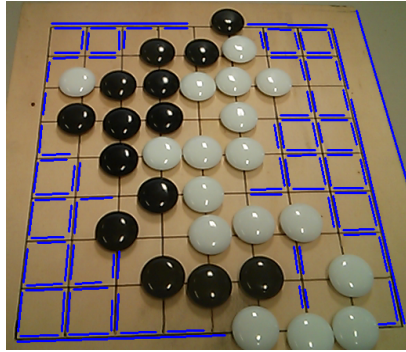
For each line the absolute pitch to the x-axis and the y-axis of the image is calculated and compared to a specified threshold. Initially, we set this pretty high to eliminate as many false lines as possible from the image. Later it turned out that the image segmentation in the first step allows us to lower the value to one. That means each line is either classified as horizontal or vertical and only perfect diagonal lines are discarded.

Using LSD

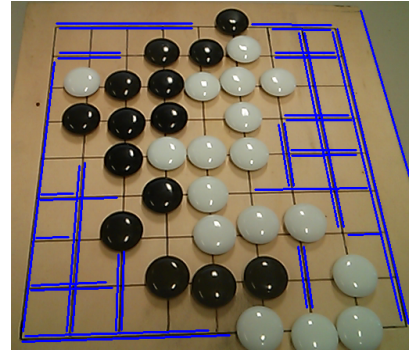
Next, we tried to apply von Gioi's Line Segment Detector [VG12] on the output of the *Canny edge detector*. We had to backport this from OpenCV 3 as it is not available in the aforementioned version. As was to be expected judging from the examples in von Gioi's paper this detector requires significant post-processing. For every square in the grid the *LSD* finds four line segments. Consequently two adjacent squares produce two parallel lines. This method is also prone to noisy backgrounds and produces many short, false positive lines. That is due

(a) Unfiltered *LSD* output

(b) ... filtered for line length,



(c) ... filtered for at least one parallel,



(d) and stitched and filtered once more for length

Figure 2.1: The postprocessing steps necessary for the *LSD*

to the nature of the detector of approximating circles with lines which comes into effect especially around black pieces, as can be seen in Figure 2.1a.

Therefore, our first postprocessing step is to eliminate short line segments (see Figure 2.1b). Afterwards we count the number of close-by parallels of each line and keep only those with a count of at least one. Having reduced the number of false positives significantly we classify the lines the same way as described above into horizontal and vertical lines. If we intersected these lines now, we would still have many false positives. That is, because one line which spans the whole length of the grid on the board might be represented by several detected line segments. As we prolong them into infinity before intersecting them, small directional variances sum up and result in intersections at the wrong place. Thus, as last postprocessing step, we merge nearby line segments into one if they are close and parallel to each other and filter once again for line length, this time with a slightly higher threshold.

2.2.2 Using corner detectors

Another option to detect the intersections directly without using line detection might be to use a corner detector. We tested the *FAST* [Ros06] detector on a slightly blurred image. However it provides too many false positives to be useful for this task. Especially around black pieces it returns keypoints which do not belong to the grid. The *ORB* detector which uses *FAST* internally does not increase quality either.

2.3 Detecting occluded intersections

Intersection positions which could previously not be detected are typically occluded by tokens. As those have to be placed on intersections, we can in turn use their location to determine the underlying intersections. Both methods (*Hough Circle Transformation* and a proprietary one based on structural analysis) which we tested need some preprocessing, though.

2.3.1 Preprocessing

To remove the lines and have only the pieces themselves remaining we tried to threshold the image. This works quite well for the black pieces. On the other hand the white pieces cannot be reliably detected this way. The reason for this is that the contrast between them and the board is too low. Therefore we turned to color images. We convert them from OpenCV's native BGR into HSV color space and threshold the value channel for black pieces. In this, we obtain good results.

White pieces cannot be detected in the value channel because their contrast is too low. In the other channels they remain hard to detect, too, as the white balance of the camera input is constantly being adjusted by the operating system. This leads to color aberrations when a player puts down a token and his or her hand takes up a large portion of the image: the image will change its color temperature and white tokens become slightly blue. Often the aberration will be corrected after the hand leaves the image, but depending on the background this does not happen every time. We tried fixing the white balance upon starting to record, but it seems the camera needs to adjust its white balance from time to time. Otherwise the white pieces stand out even less against the background.

We also tried to fix white balance settings by forcing camera parameters, but we could find none which worked under all tested illuminations.

Luckily the white tokens are well detectable under normal circumstances in the saturation channel (saturation of the pieces is low and the saturation of the board is high, hue is undefined) and when the color is shifted in the hue channel (saturation of the pieces and the board is low, hue of the pieces is low).

Therefore, we threshold the saturation and the hue channel separately, count the non-zero responses and use the individual results only if at most 20% of the image is below the threshold.

This yields pretty good results, however there is still a lot of noise in the thresholded image. Consequently we apply a Gaussian blur to the source image before thresholding and remove speckles afterwards by eroding and dilating the image slightly.

After that white pieces are actually detected better than black ones, as we describe in subsection 4.3.1. Anyhow it is not very important for us to have a perfect detection ratio in this step, because the results are not used to generate the end result but only to fill in gaps in the previously detected intersections.

2.3.2 Using Hough circles

Go pieces respectively their representation in the thresholded images are relatively round and can be detected using *Hough circles* to some extent as long as the image is taken from a relatively high angle. If the camera is positioned near to the plane in which the board lies the circles become ellipses. In order to still match the tokens we have to choose the threshold, above which a circle center will be detected in the accumulator space, quite low. This results in many false positives if noise is located in some near-circle configuration by chance.

The centers of the detected circles are then added to intersections and duplicates (determined by their distance) are removed.

2.3.3 Using contours

Finding pieces with *Hough Transformation* is slow, though, and we were not content with the result to speed ratio. That is why we investigated further and chose to try analyzing the image topologically [Suz85] using OpenCV's *findContours* function.

We preprocess the image the same way as before. This results in blobs for each piece whose contours can be detected nicely. If those are approximately round they can then be fitted well into quadratic rectangles. If we encounter a rectangle which is about twice as long in one dimension as in the other, we split it in two. Then we discard all contours whose bounding box is not more or less quadratic. Lastly we filter the rectangles by their size to remove too small or too large squares which typically come from noise.

The centers of the remaining squares are then added to intersections just like the centers of the circles when using *Hough Transformation*.

2.4 Extrapolating the board

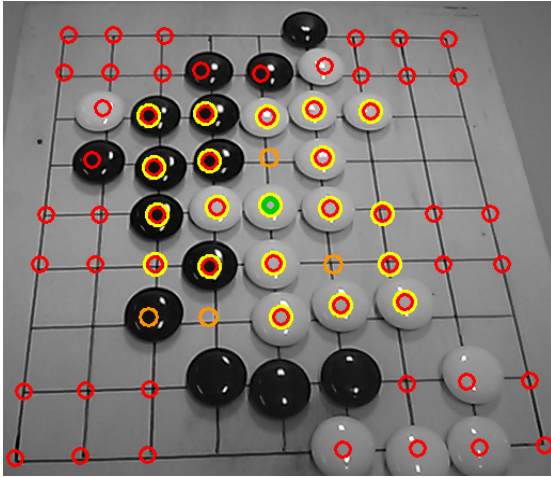
Having collected a set of probable intersections we now need to fill in gaps where neither lines nor pieces have been detected. Also, we need to classify if a detected intersection is valid and a part of the board. That is, we have to filter those which simply are intersections of prolonged grid lines with the outer edge of the board and other noise.

First we calculate the average angle of the horizontal lines from the previous step and rotate the set of intersections to justify them. This does not fix perspective distortion but it allows us to sort the intersections by rows.

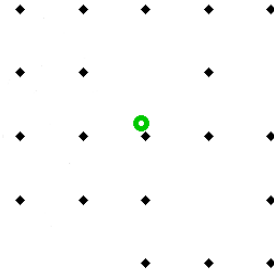
All intersections within a vertical window of 15 pixels of each other are considered to belong to one row. They are then sorted by row first and by column in second order.

Then we select a number of intersections in the center of the board by selecting all intersections inside a square around the image center. Hereby, we assume the innermost intersection on the board to be the closest (detected or yet undetected) one to the center of the image. This assumption can hardly be made in setups where the image is taken by bystanders or from randomly located cameras. We argue, though, that when trying to record a game a user will be able to adjust the camera location accordingly. Another possibility could be to prompt the user for the center of the board, which can be easily done via the touchscreen of the smart phone. We did not implement this approach, but it should yield equal quality.

If the number of selected intersections is too small (we found 20 to be a well working count), we increase the window size and try again.



(a) Red: All detected intersections; yellow: selected intersections; orange: virtual intersections; green: the board center



(b) The resulting model. Notice that undetected intersections are left out creating homography. Green: the image center

Figure 2.2: Building the model

Those intersections (shown in yellow in Figure 2.2a) are now modeled as a subset of actual board intersections. We determine the median distance between neighboring intersections (using the average worked less well). Then we iterate row-wise over the selected intersections from the top left to the bottom right. For every intersection we save the current row and column (on the board, not the image) and add a key point to our model at this location. If we encounter a gap between two intersections that is larger than the median distance we increase the column count accordingly. The same happens in y-direction with the row count. If we find an intersection to be an outlier to the left we increase the column count of all previous key points accordingly.

The interim result of this iteration is a model of a part of the board with equidistant key points where every key point corresponds to a selected intersection as shown in Figure 2.2b.

During the iteration we also add virtual intersections to fill in gaps in our intersection set (see the orange circles in Figure 2.2a). This is necessary in case the board's center intersection has not been detected. From this filled up intersection set we select the intersection closest to the image center and assume it is the center of the board. Using this intersection's row and column count we can shift our model to the correct position within the board.

Our updated interim result is now a model of a part of the board located correctly inside the complete model of the board. We can use this as input for the RANSAC algorithm [Fis81], as we made sure every key point corresponds to an intersection. Thus we can estimate a homography between the actual image and the hypothetical image of our modeled board.

This serves three purposes. First, RANSAC tolerates outliers, i.e., intersections which have been detected slightly displaced. Second, we can use the transformation information to rectify subsequent images and increase detection quality (see subsection 4.4.1). Lastly, the transformation matrix can also be used to warp a complete model of the board and lay it onto the image. After doing so we know the location of every intersection in our image – including those, which have not been detected previously.

2.5 Refining intersection locations

After the interpolation step the mean error is still quite high, because small offsets in the selected intersections result in large errors when interpolating from them. To fix these, we first shift interpolated intersections to a closeby measured intersection, if one is available. Then we perform another interpolation step with all those intersections which have been moved, i.e., in which we can be confident. This step can be repeated several times until a desired detection quality has been reached or it yields no further improvement. This is also key to detecting larger boards, where this step can be used to first find a nine-by-nine subgrid and then interpolate from it to larger boards. We did not implement this, though, due to the lack of a 19x19 board.

2.6 Classifying intersections

Having located the intersections we can evaluate the image at their positions to find whether there is a piece or not and (if applicable) its color. First, we use an adaptive threshold. To remove noise within the rectangles we perform a connected-component analysis similarly to the preprocessing step. The result is a binary image showing only the lines and black pieces. Highlights within the black pieces from reflections are still a problem but can be removed surprisingly well by erosion and dilation.

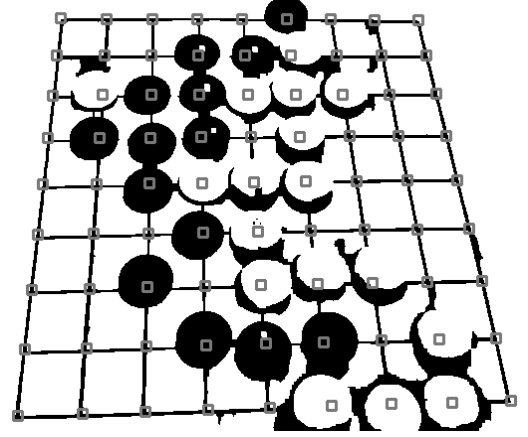


Figure 2.3: Detection windows on the thresholded, segmented image after highlight removal

Since we have rotated the intersections in section 2.4 our resulting model is rotated by the same amount. We undo this rotation now to realign the model with the actual input image. Finally we calculate the average pixel value in a window around each intersection and compare it to two thresholds, T_w and T_b , such that

$$\frac{\sum_{i=0}^I \sum_{j=0}^J w(i, j)}{I * J} = \begin{cases} > T_w + C & \Rightarrow \text{white} \\ < T_b & \Rightarrow \text{black} \\ \text{otherwise} & \Rightarrow \text{empty} \end{cases} \quad (2.1)$$

where C is a constant value that is added to the white threshold on the edge of the grid. The reason behind this is that the lines are not continued there and less black pixels are contributed by them.

2.7 Postprocessing

As we not only return the detected board but also intersections and piece locations, we have to realign them once more with the image. This is necessary because we have not just rotated them but also cropped the image in section 2.1. To undo this, we shift the intersections by the negative offset of the cropped image inside the input image.

The results still have to be smoothed over time. We decided to do this outside of the detection step, though. This has the advantage that the detector is agnostic of how its used – to simply analyze a single image or as a part of a larger recognition software. For details about this accumulation step see subsection 3.2.3.

We try to return only results which are sane. Thus we discard any result where one or more intersection lies outside of the input image or two intersections are too close to each other. In the first case it is obviously impossible to classify the intersection. In the second case no accurate value could be determined, because we evaluate the image in a window around each intersection. They would overlap if piece locations are too close. Therefore we do not use these results in our app.

Chapter 3

Creating the Android app

So far we have only described our detection algorithm, but not how we gather the data and interact with the user. Our goal was to create a simple user experience showcasing the algorithm rather than building a full featured app. Still we wanted to hide the technical details as far as useful to stay reasonably close to a usable end product.

The app was deployed and developed for a *Nexus 4* (also known as *LG-E960*) running *Android Lollipop 5.0.1* build *LRX22C*.

3.1 Tethering in the framework

First we will describe how we set up the framework and use it to fetch the input for our detector.

3.1.1 Adding the source dependencies

OpenCV for Android is available since September 2011 so we will not present in detail how we set it up. In short the process is as following: The Android bindings for OpenCV can be downloaded from the official site and be compiled into a library which can then be included into one's project's *lib*-folder. Alternatively the framework can be set as a dependency in Eclipse Android Development Tool and let it handle the compilation and dependency resolution, which is how we included the library.

One could build all the OpenCV native code for the framework oneself. This implies building it for every possible target architecture (ARMv6, ARMv7, x86...) and shipping it with the app. This creates large binaries, at the benefit of having a standalone application. On the other hand the recommended way is to depend on the *OpenCV Manager* app and ask the user on first start to install it, if necessary. This manager will then download the correct OpenCV native binaries in the required version and share it between all applications needing it. We chose to do the latter, at least for our prototype.

3.1.2 Gathering data

Android apps are structured into *Activities*. Those roughly correspond to the screens visible to a user: If an app has an overview screen, an editor screen and a preview screen, it will likely have three activities.

Each activity is built out of *Views* describing and implementing the user interface. The OpenCV framework provides a special *JavaCameraView*. This view offers a callback to *CvCameraListener2* interfaces for the delivery of camera frames and displays them on the screen. However, it does not expose access to the camera and locks the device. To have more control over the camera we subclassed it into *CameraManipulatingView*. This allowed us, for example, to experiment with locking the exposure and white balance to fix the issue of color aberration as described in section 2.3. Our final setup comprises a *DetectorActivity* which implements the *CvCameraListener2* interface and uses a *CameraManipulatingView*.

After startup we call the *OpenCV Manager* to load an instance of OpenCV 2.4.3. Upon completion our native code library is loaded and we start receiving frames from the *CameraManipulatingView* while still having access to camera parameters via it. We can retrieve OpenCV matrices from the frames — either in grayscale or RGBA color space.

Each image is then compared to the last one. If they differ significantly in more than 20 pixels, the resulting image is directly returned, thereby skipping the detector. This improves FPS and energy consumption while the user moves the camera. It also pre-filters some frames in which movement occurs, for example because one player is putting down a token.

3.2 Using the detector

3.2.1 Setup of the components

When an image passes the movement filter it is being analyzed. There exist two possible approaches. Either using OpenCV for Android or the Java Native Interface (JNI) to call compiled C/C++ code. As the first is mainly a Java wrapper around C++ code, using it also involves costly JNI calls. Therefore, it is more performant to implement the detector in C/C++ and have just one call into the detector. This is the way we chose to structure our application, because we can increase performance easily. It means, however, that our application has to be compiled and shipped for every target architecture separately using the Android NDK.

To really benefit from the performance increase we decided to use our detector as a black box and have only one call into it.

While we could allocate memory for input and output variables in the native part of our application and release it when the app is stopped, it is much easier to request the memory in the Java part and let the garbage collector clean up the unneeded memory. Even for OpenCV matrices, which have to be manually released to decrease their reference counter, this holds true, as another JNI call would be necessary in the `onStop` callback of the Activity. Therefore, all input/output-parameters are allocated or at least declared in the Java class and filled or instantiated in the native part.

3.2.2 In- and output to the detector

The first and most obvious parameter to pass is the input image to analyze. It is important to note that while OpenCV usually assumes that color images are in BGR(A)-format the Android bindings retrieve an RGBA-image from the camera. Thus, the image's color space has to be converted before passing it into the detector. As we encountered problems in numerous places when dealing with images whose color channels use 8-bit integers to encode their value, we decided to switch to floating point channels at this stage, too. There might be concerns that due to differences in floating point implementation some calculations might yield different results. We briefly looked into this and found that we either did

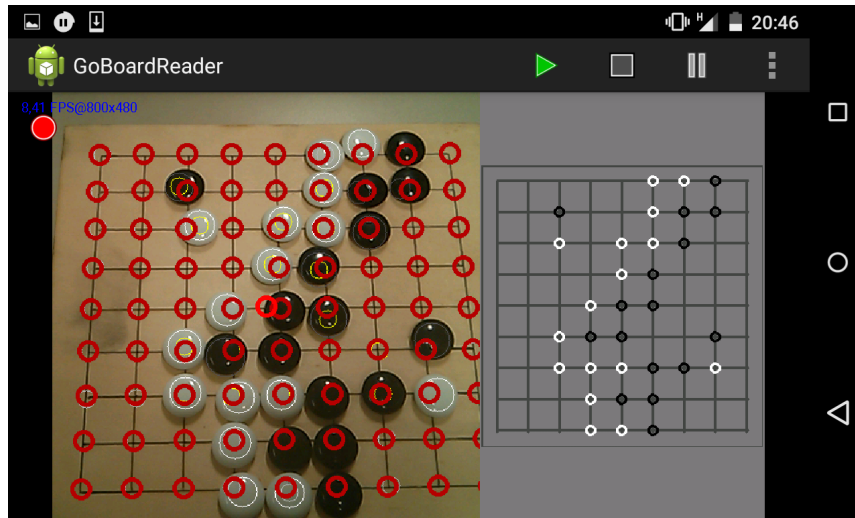


Figure 3.1: The app showing some debugging information and the detection result in a split screen while recording a game

no operation that needed this level of accuracy or it did not return different results.

The detection result is being written into a char array with an entry for every intersection of the board. It contains a 0, b or w character after detection, corresponding to an empty intersection, one with a black or with a white piece on it, respectively.

In order to perform the postprocessing steps as noted in section 2.5 we need to pass the previously detected intersection back into the detector, which is done via a `MatOfPoint2f`. This is a special kind of OpenCV matrix, consisting of only one column and n rows. It is used to quickly pass data between Java and native code. While Java lists would have to be copied when performing JNI calls, matrices use the same memory in both environments, which allows seamless data exchange.

All other parameters are there to output information from the detector. Among them are lists of detected intersections, pieces and intersections – all of which are there rather for debugging purposes than to give useful information to the user.

3.2.3 Using the detection result

Interestingly, the computed intersections of two consecutive frames somewhat differ. On the one hand, this is very convenient: it helps dealing with specular highlights as the classification windows are constantly being shifted slightly. On the other hand, sometimes the detection result will deteriorate completely. Even though the detector does some sanity checks, displaying and saving the detection results directly would result in pieces “flickering” between being detected and not detected. To counteract this, we count the number of times an intersection was classified as white or black in the last ten frames. If this counter reaches 5 or higher we assume a correct result and display it. If the game is being recorded and there was a change in the board configuration, we add the complete new configuration to a list in order to save it.

We simply save this list of char arrays as strings separated by newlines as output format when the user ends a recording session. This simple format allows easy transcoding and is even human readable.

3.3 User interaction design

The detection result still has to be displayed to the user. While often computer vision applications draw directly onto the analyzed image, we decided to show the detected information separately and only debugging output on the image. This way we balance showcasing a possible way how an app might be responding to the user and also give an insight of whether and how the detector works.

Android offers methods to draw custom views via Canvas and Paint objects similarly to OpenCV. However we wanted to give a comparable experience when using the detector on a desktop computer and via the app. Thus, we decided to draw the result by cropping the input image and draw the result next to it using OpenCV when being in detection mode.

Furthermore, we implemented a replay mode in which previously recorded games can be stepped through move by move. This allows direct control over the detection result and is a simple way to recapitulate a game.

Chapter 4

Evaluation and optimization

During the course of creating the application we tried different approaches to several interim steps. In this chapter we discuss how using different algorithms yielded different results.

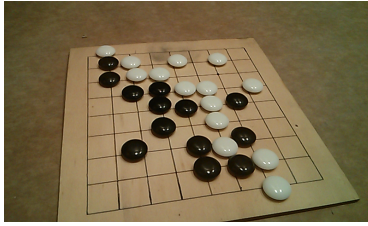
4.1 Dataset

In order to test and optimize our detector we created a set of 101 images and annotated them manually with the following information: each intersection on the board was tagged. Either as an empty intersection, an intersection with a white piece on it or one occupied by a black piece. Also, the location of all pieces was marked with a circle, i.e., with a center and an approximate radius.

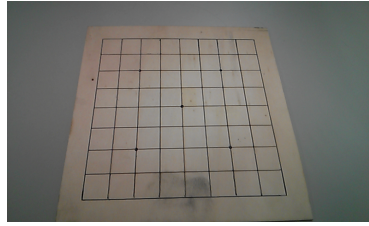
To have an image set that covers many possible game situations, we took pictures in three different lighting conditions and on different backgrounds. Each time we took images of an empty board without any pieces, a board with only some few tokens and a configuration like it often happens during endgame. That is, many pieces on the board and lots of them in a line.

Each board configuration was photographed from different angles in φ (azimuth) and θ (polar) direction and from two directions, i.e., facing the board from the side a player would see it and rotated by 90° as a third person would see the board.

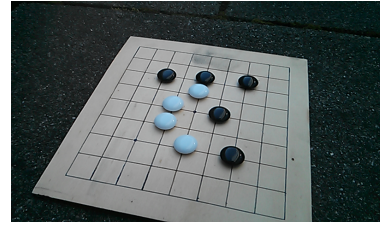
Our set consists of: 29 images taken in cold neon light on a plain gray desk; 26 images taken in the same light on a textured, brown carpet; 25 images taken on a sunny day in the shadow on a stone surface; 16 images in warm artificial



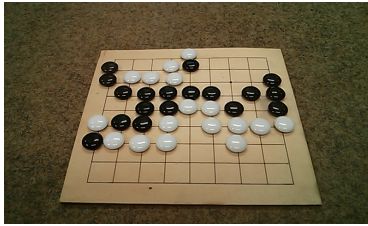
(a) Paper in warm light; camera left on medium height



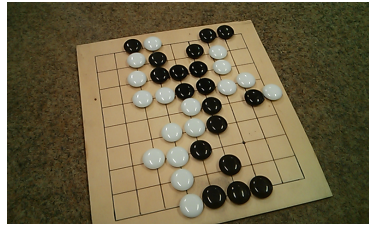
(b) Plain gray desk in neon light; camera centered high



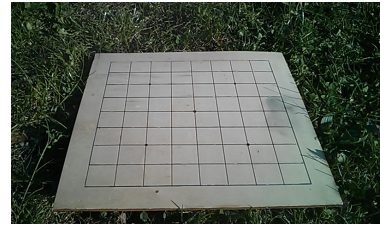
(c) Dark stone surface in the shadow; camera right high



(d) Brown carpet in neon light; camera centered low



(e) Like (d) but from bystanders' perspective



(f) On grass in sunlight; camera centered low

Figure 4.1: Some examples of different lighting conditions, angles and backgrounds as well as different piece count

light on a textured paper background. Furthermore we have taken 5 images in evening sunlight with the board lying in grass. See Figure 4.1 for some examples from the image set¹.

The images were recorded by starting the app normally. But instead of calling the detector, the input image was saved on the internal memory of the device. We did not take images using a camera or camera app to stay as close to the expectable input as possible.

In the beginning we saved images to PNG. Later we switched to persisting the images into YAML format and retook the previous images. This has the advantage that we can be absolutely sure that the input to our test instances is the same as it would be on a phone, as no PNG encoding and decoding takes place. Also, we first had problems because the OpenCV for Android function which persists images in PNG format presumably does not recognize that the camera image is encoded in RGB (see section 3.2). This led to different results on our desktop hardware than on our Android device.

¹Our complete image set can be downloaded from <http://t-animal.de/projects/bachelorThesis/>

31 of the images were randomly chosen as a test set evenly spread over all lighting conditions as well as piece and angle configurations. The grass images with pieces on the board were not part of the test set as they soon turned out to be very difficult. We included them in the training set, though. This set was used to improve our detector.

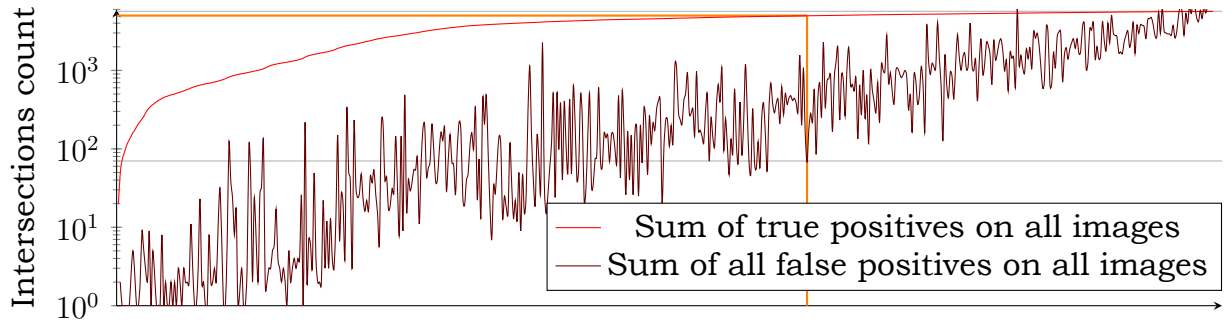
To do so we assumed that there is a global maximum for the overall quality of results when adjusting parameters of the used algorithms. Even if this assumption were wrong we optimized for a local maximum. Then, we tried to manually find optimal parameters for some randomly chosen images for every algorithm. When we were contempt and could not find parameter combinations which yielded better results on the few images we fine tuned them on all training images. For this we automatically created every combination of parameters in the vicinity of our manually optimized ones. Then we checked the results of those combinations on every image by brute force on a cluster of 45 octacore machines. We compared the results to the annotations and transferred the best performing parameters into our application.

4.2 Visible intersections

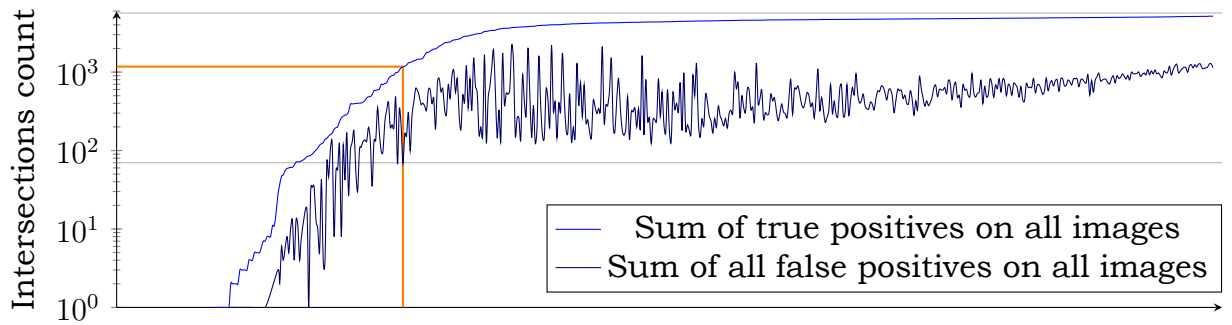
When evaluating the detection rate of visible intersections we first checked if the intersection was within the boundaries of the annotated board plus a padding of 15 pixels. If not, the intersection was considered uninteresting and did neither count positively nor negatively (we assumed it to be noise outside the board). If it was inside the board we searched for the nearest annotated intersection. If there was none within a range of 15 pixels the intersection was counted as a false positive. But if there was, then the annotated intersection was marked as detected and not considered as a possible match for other intersections. The threshold of 15 pixels was chosen roughly as a quarter of the average distance between two intersections and a third of the diameter of a piece as measured in some sample images.

4.2.1 Optimizations on the training set

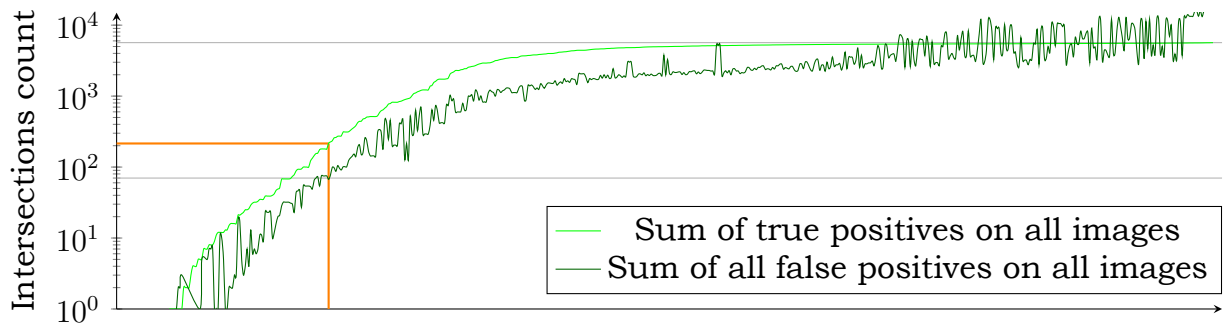
While optimizing the parameters used in our algorithm, we measured the quality as the ratio of detected to undetected intersections, also considering the number



Different parameter combinations, sorted by resulting number of correct positives
 (a) Representative subset of 605 combinations of 60481 tested for the Houghline detector



Different parameter combinations, sorted by resulting number of correct positives
 (b) Representative subset of 605 combinations of 58321 tested for the LSD detector.



Different parameter combinations, sorted by resulting number of correct positives
 (c) Representative subset of 605 combinations of 1440 tested for the FAST detector

Figure 4.2: The x-axis shows different combinations of parameters that we have evaluated. On the y-axis the true positives, false positives, totally available intersections (upper gray line) and number of analyzed images (lower gray line) per combination can be read off. The x-axis does not imply any order of the tested combinations; results have been simply sorted after number of correct intersections. We chose the parameter combination with the highest true positive rate whilst less false positives than evaluated files (orange line)

of false positives. Traditionally, one would calculate some ratio of true positives to false positives. However none of the usual measurement methods showed us a clear indication which combination of parameters to use, so we devised the following metric: we sorted the results by the number of correctly matched intersections from low to high. Then we chose the parameter set giving the highest number of true positives and at the same time having less or equally many false positives than the number of images analyzed ($|false_positives| \leq |input_images|$, i.e., statistically maximal one false positive per image). This is shown (albeit on a representative subset of all tested parameter combinations) in Figure 4.2, where the parameters we chose is marked in orange.

HOUGH

In comparison to the other algorithms detecting the lines using *Hough Transformation* performs best quality-wise. As Figure 4.2a shows, we still had to compromise when choosing the final parameter set. The one we chose resulted in 89.67% (5084 out of 5670) correct detection rate on our training set, with a total of 68 incorrect intersections (1.199% of available intersections).

The graphic shows also, that the quality of the line detection does not depend very much on the parameters used: there are only a few parameter combinations that yield very poor results, and hardly any that results in finding no intersection at all. This is generally a good thing, because it means that even if we chose a combination that is non-optimal under some circumstances its result quality does not collapse completely.

As would be expected the average number of false positives rises with the percentage of correctly found intersections. Luckily, there is enough variance that even at a high detection rate we can find parameter combinations with less than one false intersection per image on average.

LSD

As noted in subsection 2.2.1 the Line Segment Detector needs significant post-processing to give any reasonable results. Still it is easily outperformed by the variant using *Hough transformation*. In Figure 4.2b one can see that the amount of false positives does not drop below the number of input images for any parameter combination yielding more than about 25% detection rate. Furthermore,

		Hough	LSD	FAST
No pieces on the board	True positive rate	100%	30.3%	0.685%
	Precision	99.4%	94.2%	83.3%
Some pieces (7-13)	True positive rate	95.6%	11.4%	1.97%
	Precision	98.6%	91.2%	53.3%
Many pieces (27-34)	True positive rate	58.6%	9.02%	3.80%
	Precision	95.5%	94.0%	68.8%

Table 4.1: Quality of the tested algorithms on our test set. The *Hough Transformation* based algorithm dominates all categories.

the average ratio of correct results to false positives is nearly all the time worse than its *Hough* counterpart's. Only in the rightmost segment the ratio is better. The total number of wrong intersections ranges in the thousands here, though, making this segment unusable for the application.

The chosen parameter combination performs at a detection rate of only 20.53% (1164 out of 5670) with 69 false positives. To achieve a detection rate similar to the one of the *Hough* detector we would have to accept more than 10 false positives per image on average.

FAST

The last tested algorithm is the *FAST* corner detector. It performs even worse when used for detection of the intersections. It is the only detector that resulted in more false positives than correct intersections in some combinations. The ratio of false positives to correctly classified intersections is never better than when using *LSD* or *Hough transformation*.

The chosen parameter combination yields a mere 2.7% (155 out of 5670) correctly detected intersections with 70 wrongly placed ones. Overall the detector seems to stick too much to the lines themselves and places many keypoints around them and at the border of the black pieces.

4.2.2 Performance on the test set

To evaluate the performance of the three possible approaches we tested them on 31 images which had not been included in the training set.

HOUGH

The *Hough* detector produces quite good results on these images. When no tokens occlude the board (on the left in Figure 4.3a) the intersections are detected de facto perfectly. It does have some issues when some pieces lie on the board and its detection rate drops significantly (to about 60% of the performance without pieces) when there are many pieces on the board (on the right in Figure 4.3a). This was to be expected, though, and its performance is still better than the other two algorithms’.

LSD

The Line Segment Detector based approach detected only about 18% of the available intersections in our test set. That means its performance is significantly worse than the *Hough Lines Detector*’s, its true positive rate being only one fifth. Consequently its relative quality was even lower on the test set than on the training set (where it reached 22% of the quality of its counterpart).

At first glance the precision (the dark blue circles in Figure 4.3b) seems to be quite high. This is a result of how we chose the algorithms’ parameters, though. When comparing it to the first evaluated approach we see a clear decrease in precision, too.

FAST

The corner detector based approach obviously yielded the worst results. Interestingly, though, the percentage of correctly detected intersections rises slightly with the number of pieces on the board. Unfortunately this is only due to chance, not because the detector is suitable for this kind of work. The reason lies in the amount of pieces: they reduce the number of visible intersections. As the detector favors the edges in the image it becomes more probable for a keypoint to lie on an intersection, when less lines are visible.

Overall we dismissed the idea of using corner detectors after these results.

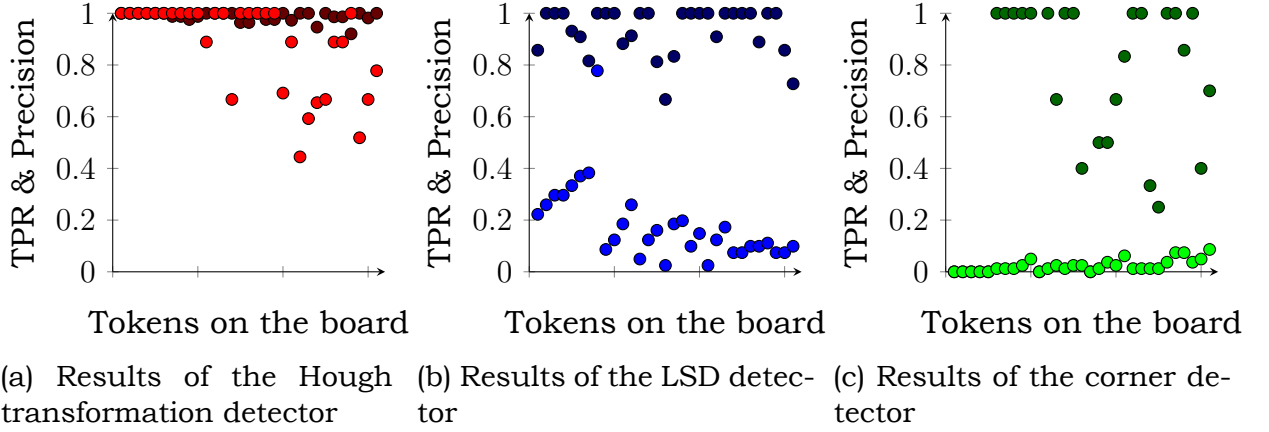


Figure 4.3: True positive rate (light color filled circles) and precision (dark color filled circles) per image in our test set.

4.3 Occluded intersections

Now that we have analyzed the performance of the line detectors let us examine the different approaches to detect pieces, respectively the intersections upon which they are placed.

As described in section 4.1 we had annotated all pieces with their center and radius. During evaluation we counted a token as correctly detected if its location was within 15 pixels of the annotation and its radius was wrong by at most 10 pixels. We did not care too much about the radius, though, as it did not really matter to us if the piece had been detected correctly in size. We rather strove to find the correct center of the token as we assumed this to be the location of an intersection.

4.3.1 Optimizations on the training set

As we explained in subsection 2.3.1 we used thresholding to get the input to both our detection methods (our contour based approach and *Hough Circle Transformation*). We optimized the thresholding parameters separately from the detection parameters as otherwise our search domain would have grown too much in size. We also optimized locating dark and light pieces separately. This is unproblematic because we do not work on the same data (hue and saturation channels and value channel) making it two different, independent tasks.

We chose the final parameter set like we did for the line detection algorithms by sorting after true positive rate and selecting the best scoring algorithm yielding less false positives than a specific number. We chose this threshold a lot lower, though, at a maximum of seven false positives for either color. We made this choice, because we found that wrongly located tokens tended to have a greater impact on detection quality than false positives in the line detection step. Both algorithms still performed good enough to provide a significant amount of additional intersections.

Contours

The idea of the first algorithm, as described in subsection 2.3.3, is to select quadratic rectangles around contours of blobs in the thresholded images. This works surprisingly well and we believe it could be improved further to perform as good as *Hough Circle* detection or even better, by using rotated rectangles. The parameter combination which we chose yielded 84.23% (438 of 520) detection rate with 3 false positives on the white pieces and 61.12% (316 of 517) detection rate with 7 false positives on the black pieces. All in all this results in a 72.71% (754 of 1037) detection rate with 10 false positives.

Hough

The more obvious approach is to use *Hough Circle* detection to locate the pieces as mentioned in subsection 2.3.2. This approach correctly detected 78.27% (407 of 520) with 5 false positives of the white pieces and 69.05% (357 of 517) with 6 false positives of the black pieces. Summed up this is a 73.67% (764 of 1037) detection rate with 11 false positives.

The reason why both approaches perform worse on the black pieces are specular highlights. They can occur on white pieces, too, but naturally are worse on black pieces, both in appearance and influence. After thresholding they appear as holes in the pieces, and since we use erosion and dilation to filter out speckles, their size increases. If they are cast by a linear or planar light source, they often span large portions of the black pieces. Then, they cause the piece depiction to become rectangular and considerably smaller. In extreme cases they even lead to black pieces vanishing completely.

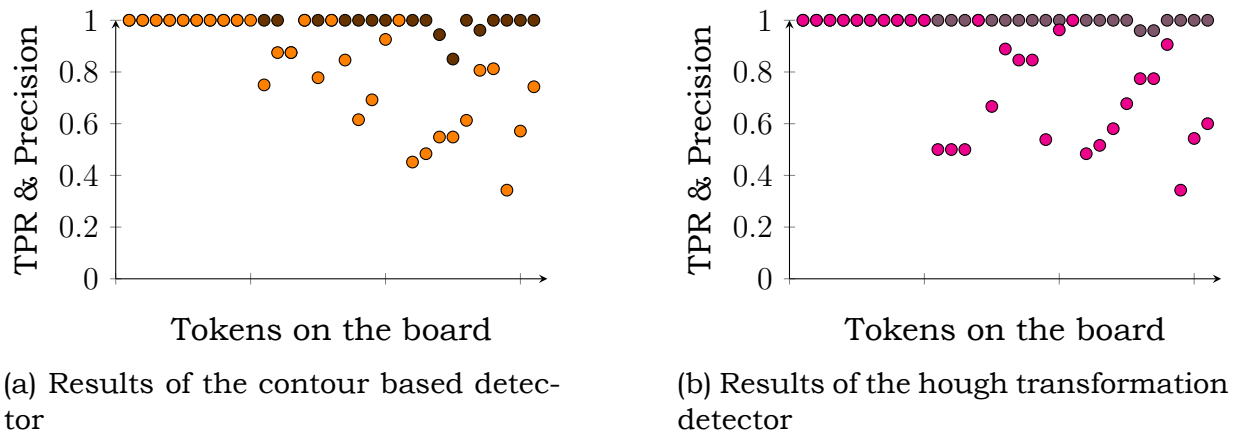


Figure 4.4: True positive rate (light color filled circles) and precision (dark color filled circles) per image in our test set.

4.3.2 Performance on the testing set

Interestingly, the contour based approach yields better results when there are less pieces on the board. This is due to how the algorithm handles two adjacent pieces which are connected. Not all of those are correctly split into two pieces and thus discarded, especially if the board is rotated. It cannot deal with more than two merged pieces, too. If this could be improved it would outperform its *Hough Transformation* based counterpart.

When summing up the results both perform equally well on our test set as Table 4.2 shows.

4.4 Pre- and postprocessing

We did not just evaluate different approaches to the detection of intersections, but also looked into how we could improve the detection by other means.

4.4.1 Rectifying perspective distortion

The camera angle changes only slightly when the mobile device is hand-held or changes not at all when it is fixed on some kind of stand. Thus it should be feasible to use the orientation of the board in a previous frame to correct the orientation in the current frame. This information must only be discarded when the camera moves drastically or when the result in the current frame is

		Contour	HOUGH
No pieces on the board	True positive rate Precision	N/A 1 false positive	N/A N/A
Some pieces (7-13)	True positive rate Precision	86.7% 98.6%	75.90% 100%
Many pieces (27-34)	True positive rate Precision	40.0% 99.2%	70.6% 99.2%
Total	True positive rate Precision	71.6% 97.8%	71.6% 99.3%

Table 4.2: Quality of the tested algorithms on our test set. When there are no pieces on the board neither rate can be calculated (TPR divides by zero, Precision becomes zero)

very bad due to the correction. Therefore we enabled the detector to receive a transformation matrix, apply it to the input image before any operation happens and transform the results back after all detection has taken place.

The source of the transformation is the inverse transformation matrix determined by the *RANSAC* step in our detection pipeline. This automatically incorporates lens distortion, too. The matrix has to be inverted because it is the matrix transforming a perfectly aligned model onto the distorted board in our image. As we want to do the opposite (transforming a distorted board to be rectangular) the transformation has to be performed in the other direction.

We did not have video test data and could not reuse information from a previous frame. In order to still test the approach we fed the transformation matrix back into a second detector run on the same image. Assuming the location of the camera does not change too much during two frames this substitution can be made. The first impressions of doing so were pretty promising. The images were transformed well and seemed to be taken directly from above the board.

However, having reached near-perfect accuracy on the test set by optimizing for images taken from an angle, the results were poorer than before. This does not mean that the step is a failure, though, but simply that the optimized parameters for the other pipeline steps are very well fitted for their purpose. When researching the task further it might be a good idea to incorporate this preprocessing step upfront.

4.4.2 Removing specular highlights

Specular highlights are a problem especially when they appear on black tokens. They do not only impair piece detection quality but also can be a problem when extracting the piece color. We looked into inpainting them by first determining a mask through thresholding on grayscale images with very high values. Then we tried applying Alexandru Telea’s inpainting algorithm [Tel04]. However, we had to dismiss the idea, since the speed of the method was too slow. As specular highlights are mainly a problem on black pieces we tried to simply set their color to black, which was fast but did not result in higher quality, as highlights occur on white tokens, too.

We did not look into this any further, but we think that by inpainting with the dominant color in the vicinity of the highlight, good results in reasonable time might be achievable.

4.4.3 Removing large outliers during detection

When testing our pipeline on an actual android device we recognized that two successive frames could yield completely different results. Mostly the differences were marginal but sometimes between two correctly classified images there was one which was detected very poorly.

Moving average over intersection results

To mitigate the effects of those frames we tried applying a moving average over the intersections. The previously detected intersection counted 75% while those from the current image contributed only 25%, i.e. $I_t = \frac{1}{4}I + \frac{3}{4}I_{t-1}$.

While this approach removed large outliers from the detection result on wrongly classified images, it also led to worse results. Large errors still have such a high influence that the result would be skewed for several frames.

Filtering obviously incorrect detection results

Our next approach was to filter out obviously wrong detection results. We considered a result to be wrong when at least one intersection lies outside of the image or if the distance between two intersections was smaller than five pixels. We then marked the whole board as unclassified (i.e., neither black, white or

empty) and did not count these steps when calculating the average over time as mentioned in section 3.2.

After implementing this approach we hardly ever saw incorrect results when capturing a game and so we kept it in our final application.

4.5 Speed on an Android device

We evaluated all algorithms for their speed on our development phone, too. To do so, we measured the current time in milliseconds before entering the respective function and afterwards. To compensate for other apps interfering with the application and scheduling we took an average of 100 samples. The individual results are shown in Figure 4.5a. All colored entries correspond to parts of our detector. The entry “create output” includes all steps outside of the native detector (counting responses over time and drawing the detection result on the output image). In Figure 4.5b we show how the total runtime of 1095.84 ms is built from the different parts of our final application.

The mobile phone we used for measurements is ranked mid range in several benchmarks [rAHK15, rPLI15] as its model is three years old as of this writing. That gives us confidence that our measurements are representative for a large portion of Android phones and performance will be better on many current phones.

4.5.1 Getting a baseline

It would be nice to achieve results in real time, i.e., at least 25 frames per second. From this follows that one complete receiving and processing cycle should take at most 40ms. This is in part limited by the maximum speed which the framework can read images from the camera. Our tests show that this alone takes 51.33ms, though. That means real time speed is out of the question on our hardware.

4.5.2 Comparing the runtime of different approaches

We then took speed measurements of the line detection methods. The Line Segment Detector outperforms the *Hough Transformation* by a factor of two on our testing hardware. Unfortunately its quality is also only half as good. If

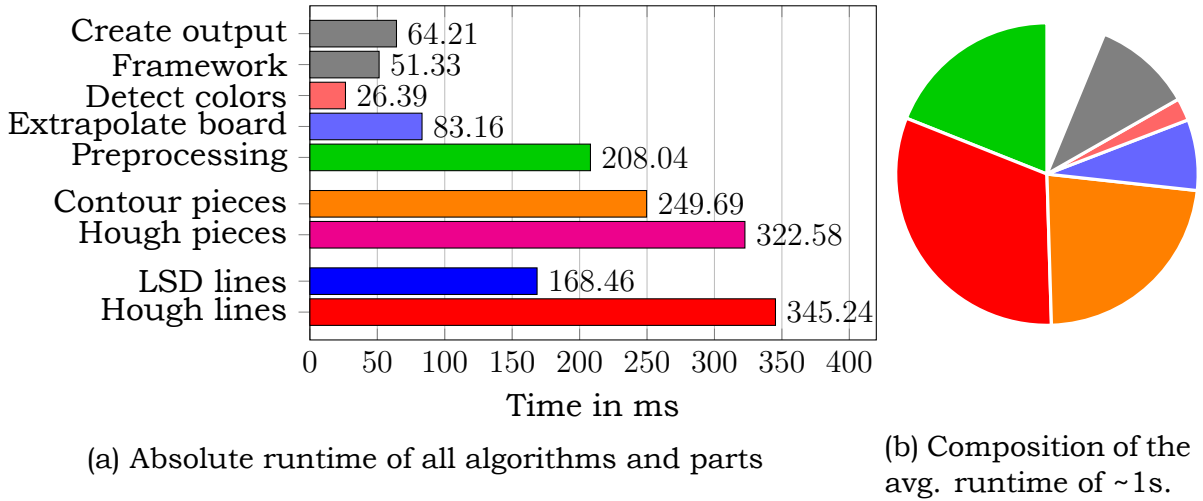


Figure 4.5: Runtime of different parts of our application

the postprocessing could be improved using the *LSD* would increase the frame rate by about 15%. We did not evaluate *FAST* corner detection to compare it, because it is simply unfit for the task.

We also looked in detail at the piece detection algorithms. Originally we created the contour based approach for the piece detection to increase the speed of our detector. This goal was reached as it performs at about 129% of the speed of the *Hough Circle detector*. But when looking at the absolute speed advantage of 72.9ms, it does not make a real difference considering the total runtime of about one second. As they perform with about the same quality, we still chose it over the *Hough* based approach.

4.6 Overall performance and live experiences

By now we have only tested the quality of each individual step separately. Finally we also evaluated how well our whole detection pipeline performed on our testing and training set.

To do so, we checked if all intersections were correctly classified regarding their location and color. An intersection was only counted as correct if both criteria were met. Results which had been marked as incorrect by our detector (see section 2.5) were not part of our calculations. Five images out of 70 in our training set have been marked so. After subtracting them, all but 48

	Training set	Test set
Number of images	70	31
Results marked incorrect	5	2
Accuracy localizing	99.1% (5217/5265)	100% (2349/2349)
Accuracy classifying	98.0% (5112/5217)	98.6% (2317/2349)
Accuracy for both chained	97.1% (5112/5265)	98.6% (2317/2349)

Table 4.3: Quality of our complete detection pipeline on our image sets

intersections were detected at the correct location. Out of the correctly located intersections 105 were classified incorrectly. When calculating the final number of correct (in location and color) intersections this makes a detection rate of 97.1% correct intersections.

On the test set our detector performed better. It discarded relatively less images and the locations in the other images were detected perfectly without any errors. 30 tokens have been missed and the intersection was regarded empty, though. Two more intersections were wrongly classified. This makes a total accuracy of 98.6%.

Another interesting topic is, how well the App performs under real world circumstances, with some camera movement and part time occlusions. To test this, we let KiGo (a KDE Go application²) generate sample games and reproduced them in different lighting conditions. In total there were 180 moves that could be recorded in six games, also counting the handicap pieces placed on the board before a game starts. While testing, the phone was placed in a mounting device similar to the one in Appendix A.

For these tests we used the final application as it was released to the Google Play Store³. In it we chose to use the *Hough Line* detector because it was the only well-performing line detector. For the detection of pieces we used the contour based algorithm due to its speed advantage.

Three test games took place on the same desk as in our training data. One of those was detected perfectly without any error. In one there was an intersection which was wrongly classified as occupied on several occasions. In the last game an empty intersection was classified as white but this was corrected when a

²<https://www.kde.org/applications/games/kigo/> accessed on 17-May-2015

³It can be installed at https://play.google.com/store/apps/details?id=de.t_animal.goboardreader

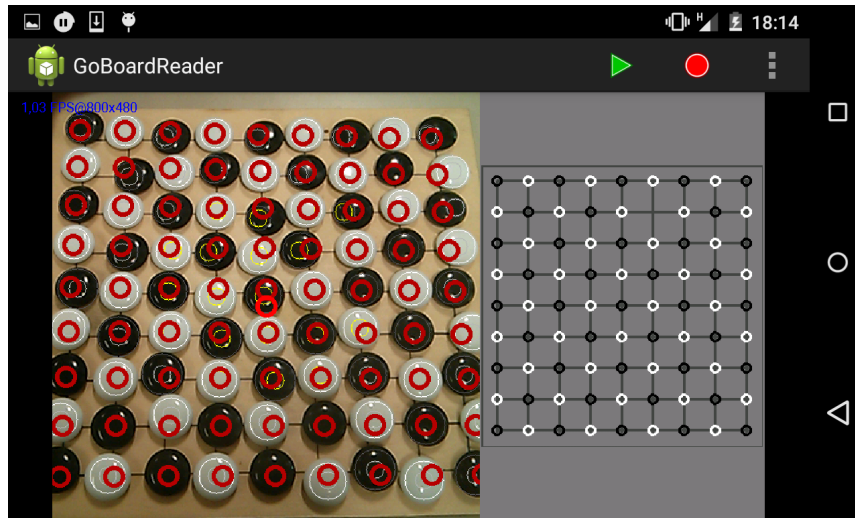


Figure 4.6: The performance on a hand-held Nexus 4 and a completely covered board

black piece was put on its location. But that also means 100% of all played pieces were detected correctly in those three games, only empty intersections were problematic.

Two more games were played under difficult lighting conditions in an inner courtyard. The opening in the ceiling cast large reflections on the black stones and strong winds shook the camera. In total there were three pieces which were not detected persistently, i.e., they vanished and reappeared several times. Two tokens were not detected at all. One move was detected too late and there were empty intersections which were classified wrongly. Under these difficult conditions 90.3% of all pieces on the board were detected correctly.

Finally we recreated a game on a desk near a window under neon lighting. Here we encountered one piece which was not detected persistently and one which was not detected at all. That makes a 93.8% detection ratio. Also, there was one piece which was detected but not played at the very end of the game, after both players had already passed.

After this we tried to max out the amount of pieces we could place on the board to see how well the algorithm performed under extreme circumstances. Hereby a detection was possible until the board was nearly completely covered in pieces. We then put a piece on each intersection in a checkerboard like pattern – alternating black and white. Even this completely abnormal configuration

could be detected with near 100% accuracy as Figure 4.6 shows. The results were very unstable, though. That means between a correctly classified frame lay several discarded ones, which did not contribute to the detection result but prolonged the detection time. Of course this is not a valid state of a game of Go, but it shows the power of using intersections when detecting densely occupied boards.

Chapter 5

Conclusion and outlook

5.1 Conclusion

We could show that it is possible to detect a Go board, extract the current game state and record a game with the limited resources of a mobile device. Our approach uses results of line detection to detect the grid and intersections. This yields high rates of detected intersections on empty to sparsely occupied boards. To improve our detector, we also add the center of roughly detected pieces to the set of intersections. From this we calculate the complete set of intersections starting from the center of board, which has to be provided by the user. Finally, we evaluate the image at their locations to classify each as black, white or empty.

The quality of our detector is comparable to other works in this field. It even outperforms them when there is a large number of pieces on the board. This allows us to start the detection mid-game or to stop it and continue it later and permits some movement of the camera. Our experiments show that we can detect the current game state under extreme circumstances. Even boards which are completely covered in pieces can be classified correctly.

Speed wise we can reach near real time performance on desktop hardware. On weaker mobile processors we reach about one frame per second, which beats previously existing algorithms.

Our contribution is a faster and reliable method for the detection of Go boards, which can handle densely occupied boards, too. We also showcased

a possible end user product by embedding the developed detector in an android app.

5.2 Outlook

Due to the lack of a larger board we could not test our algorithm on more than a 9-by-9-grid. We are confident that our approach can be applied to other sizes, too, with minimal adjustments.

Another area of work could be to automatically detect the center of the board. This would remove any need for user interaction and allow the application to be run completely autonomously.

There are still some issues with specular highlights, which could also be improved upon. If they could be painted in well and fast, it could improve piece detection as well as color detection. Another option would be to improve the color detection step, because it is more prone to errors from specular highlights. One possibility could be to switch from rectangular windows to some more sophisticated shape like circles or discs. Obviously this would not improve piece detection, but both inpainting and different evaluation shapes might be combined.

Even though our detector's quality is comparable to the one in Scher's, Crab's and Davis's paper [Sch08] combining both could remove persistent errors. The modeling and path finding step from their approach could be added to our detector, especially as it was developed for a more general use. It should not take too much time and could be performed right after the detection was stopped.

Smartphones offer a multitude of additional sensory information, which might be used to improve the detection quality. For example, if the camera height above the board is known, one could calculate a transformation matrix as in subsection 4.4.1, based on the tilt of the phone. This can be determined using the built in accelerometers or gyroscopes.

Appendix A

Simple mount for Nexus 4

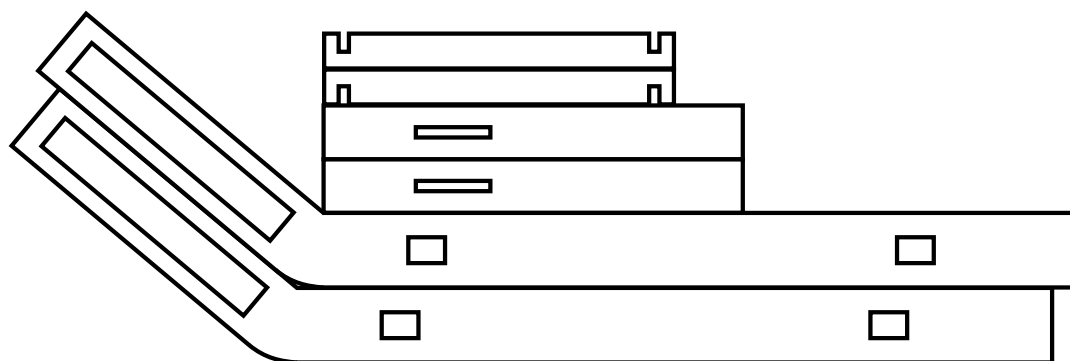


Figure A.1: A simple mount which can be printed out (at twice the size), cut out of thick cardboard or acrylic glass and put together as needed

List of Figures

2.1	The postprocessing steps necessary for the <i>LSD</i>	8
2.2	Building the model	12
2.3	Detection windows on the thresholded, segmented image after high-light removal	14
3.1	The app showing some debugging information and the detection result in a split screen while recording a game	20
4.1	Some examples of different lighting conditions, angles and backgrounds as well as different piece count	24
4.2	The x-axis shows different combinations of parameters that we have evaluated. On the y-axis the true positives, false positives, totally available intersections (upper gray line) and number of analyzed images (lower gray line) per combination can be read off. The x-axis does not imply any order of the tested combinations; results have been simply sorted after number of correct intersections. We chose the parameter combination with the highest true positive rate whilst less false positives than evaluated files (orange line) .	26
4.3	True positive rate (light color filled circles) and precision (dark color filled circles) per image in our test set.	30
4.4	True positive rate (light color filled circles) and precision (dark color filled circles) per image in our test set.	32
4.5	Runtime of different parts of our application	36
4.6	The performance on a hand-held Nexus 4 and a completely covered board	38

- A.1 A simple mount which can be printed out (at twice the size), cut out of thick cardboard or acrylic glass and put together as needed 43

List of Tables

4.1	Quality of the tested algorithms on our test set. The <i>Hough Transformation</i> based algorithm dominates all categories.	28
4.2	Quality of the tested algorithms on our test set. When there are no pieces on the board neither rate can be calculated (TPR divides by zero, Precision becomes zero)	33
4.3	Quality of our complete detection pipeline on our image sets . . .	37

Bibliography

- [AI92] T. K. ANDO ITARU: *GO GAME PLAYING DEVICE*, October 1992.
- [Bul01] B. Bulsink: *Device for detecting playing pieces on a board*, Jan. 2 2001, US Patent 6,168,158.
- [Fis81] M. A. Fischler, R. C. Bolles: *Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography*, *Communications of the ACM*, Bd. 24, Nr. 6, 1981, S. 381–395.
- [HIR99] Y. HIROSH: *ELECTRONIC GO BOARD SET*, April 1999.
- [Hir05] T. Hirsimäki: *Extracting Go Game Positions from Photographs*, *Helsinki University of Technology, Tech. Rep*, 2005.
- [MAM02] T. MAMORU: *AUTOMATIC GO RECORDING SYSTEM OF GO BOARD*, January 2002.
- [Mih39] F. Mihori, Z. Iwadō, J. K. Kankōkyoku: *Japanese Game of “Go”*, Board of tourist industry, Japanese government railways, Maruzen, 1939.
- [Mol10] E. Molla, V. Lepetit: *Augmented reality for board games*, in *Mixed and Augmented Reality (ISMAR), 2010 9th IEEE International Symposium on*, IEEE, 2010, S. 253–254.
- [rAHK15] AnTuTu Hong Kong: *Mobile Ranking*, <http://www.antutu.com/en/Ranking.shtml?cmd=mobile&page=1>, 2015, Accessed on 17-May-2015.
- [Ros06] E. Rosten, T. Drummond: *Machine learning for high-speed corner detection*, in *Computer Vision–ECCV 2006*, Springer, 2006, S. 430–443.

- [rPLI15] Primate Labs Inc.: *Android Benchmarks*, <https://browser.primatelabs.com/android-benchmarks>, 2015, Accessed on 17-May-2015.
- [Sch08] S. Scher, R. Crabb, J. Davis: *Making real games virtual: Tracking board game pieces*, in *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, IEEE, 2008, S. 1–4.
- [See10] A. K. Seewald: *Automatic Extraction of Go Game Positions from Images: A Multi-Strategical Approach to Constrained Multi-Object Recognition*, *Applied Artificial Intelligence*, Bd. 24, Nr. 3, 2010, S. 233–252.
- [Suz85] S. Suzuki: *Topological structural analysis of digitized binary images by border following*, *Computer Vision, Graphics, and Image Processing*, Bd. 30, Nr. 1, 1985, S. 32–46.
- [Tel04] A. Telea: *An image inpainting technique based on the fast marching method*, *Journal of graphics tools*, Bd. 9, Nr. 1, 2004, S. 23–34.
- [Tsu04] M. Tsutsumi: *AUTOMATIC GO-GAME RECORDING DEVICE*, June 2004.
- [VG12] R. G. Von Gioi, J. Jakubowicz, J.-M. Morel, G. Randall: *LSD: a line segment detector*, *Image Processing On Line*, Bd. 2, Nr. 3, 2012, S. 5.