University of Zurich[UZH]

BACHELOR THESIS – Communication Systems Group, Prof. Dr. Burkhard Stiller

Data Transfer Using a Camera and a Three-Dimensional Code

Jeton Memeti
Stalden, Switzerland
Student ID: 07-722-408

Supervisor: Flávio Santos, Martin Waldburger, Prof. Dr. Burkhard Stiller
Date of Submission: April 6, 2012

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Bachelor Thesis
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Abstract

One- and two-dimensional barcodes have become very popular in the last years and decades and are widely used to identify products and services. Recently, two-dimensional barcodes like QR code are also used to transfer a hyperlink from a magazine or poster to a smartphone.

To overcome the low storage capacity of two-dimensional barcodes, the time can be introduced as a third dimension. Instead of one, a sequence of barcodes can be used to transfer an amount of data therefore. This would create a new application area. One could transfer for example files from a personal computer to a smartphone in cases where a wired or wireless connection is not available.

The goal of this thesis is to design and implement the whole workflow on the receiver side. This contains capturing the 3D barcode, recognizing and reading the sequence of 2D matrices, and retrieving the original content. An important criterion though is the maximization of the transfer rate respecting the reliability. Furthermore, adversarial conditions have to be identified, tested and documented.

The implemented prototype achieves a theoretical throughput of 12'288 bytes per 30 seconds, which means about 430 bytes/s. Interfering factors like distortion, rotation and light reflectance have been evaluated and considered in the implementation.

Future extensions of the prototype could for example compensate the occurrent fish-eye effect and aim for larger barcodes, which could help increasing the throughput. Auspicious in this regard are the announced API changes and improvements in Android 4.0, e.g. taking pictures faster due to a shorter burst delay.

Zusammenfassung

Ein- und zweidimensionale Barcodes haben in den letzten Jahren und Jahrzehnten sehr an Popularität gewonnen und werden vielerorts zum Identifizieren von Produkten und Dienstleistungen genutzt. Zweidimensionale Barcodes wie zum Beispiel QR Codes werden neuerdings auch dazu verwendet, schnell einen Hyperlink von einer Zeitschrift oder einem Plakat auf ein Smartphone zu übertragen.

Um das geringe Speichervermögen der zweidimensionalen Barcodes zu überwinden, kann man die Zeit als dritte Dimension hinzunehmen. Anstelle von einem Barcode, der eine bestimmte Information enthält, hätte man dann eine Sequenz von Barcodes. Dies würde einen neuen Anwendungsbereich schaffen. Beispielsweise könnte man Dateien von einem Computer auf ein Smartphone übertragen, falls mal kein USB-Kabel oder Netzwerk vorhanden ist.

Das Ziel dieser Arbeit ist auf der Empfängerseite den ganzen Prozess zu designen und implementieren. Der Empfäger umfasst die Schritte von der Aufnahme des 3D Barcodes über die Erkennung der einzelnen 2D Matrizen bis hin zu deren Dekodierung und zum Wiederherstellen des ursprünglichen Inhalts. Ein wichtiges Kriterium dabei ist die Maximiserung der Übertragungsrate unter Berücksichtigung der Zuverlässigkeit. Ferner müssen Störfaktoren ermittelt, getestet und dokumentiert werden.

Der entwickelte Prototyp erreicht eine theoretische Durchsatzrate von 12'288 Bytes pro 30 Sekunden, sprich ca. 430 Bytes/s. Störfaktoren wie Verzerrung, Rotation oder auch reflektierende Lichter wurden hinreichend getestet und auch so weit möglich behoben.

Künftige Erweiterungen des Prototyps könnten beispielsweise den auftretenden Fischaugeneffekt kompensieren und somit die Verwendung von noch grösseren Barcodes anstreben, wodurch der Durchsatz erheblich gesteigert werden kann. Vielversprechend sind diesbezüglich auch die angekündigten API Erweiterungen und Verbesserungen in Android 4.0, wie zum Beispiel die schnellere Aufnahme von Fotos.

Acknowledgments

I would like to thank all the people involved in this thesis. First of all, I thank professor Burkhard Stiller for giving me the opportunity to write this thesis at the Communication Systems Group. Many thanks also go to Flávio Santos and Martin Waldburger for providing the idea of this thesis and their great support concerning the implementation, design decisions and this written report.

Contents

Chapter 1

Introduction and Motivation

Products and services are often identified by numeric codes. This is motivated by the need to quickly find and reference objects. In order to enable machines to read these identifiers, alternative methods were proposed to create codes using visual representation. Barcodes [11] are a successful example of codes that use visual representation. They use a one-dimensional code to represent numeric values. Barcodes are still very popular and can be found in the majority of products nowadays.

In the last decade, two-dimensional codes started gaining popularity. The main motivation for their creation was the need for more flexible representation, which allows storing also text and binary content. Thus, in order to extend the storage capacity of one-dimensional barcodes, the second dimension allows the use of a matrix barcode instead of aligned bars. The most popular example of 2D codes is known as QR (Quick Response) [7] codes. However, QR codes store at most 3KiB of binary data [9] , depending on the version and error correction level.

As someone may not know – even if seen many times before – what these one or two-dimensional codes mentioned above look like, the two following figures show examples of them. Figure 1.1 shows a barcode[1] for the text *"Data Transfer 3D Code"*. The same information is represented by the QR code[2] in Figure 1.2.

Figure 1.1: Barcode example

Figure 1.2: QR code example

[1] generated by `http://barcode.tec-it.com/` (Code 128)
[2] generated by `http://qrcode.kaywa.com/` (no information about which version was used)

1.1 Motivation

The motivation of this thesis is to introduce a third dimension for circumventing the capacity limitation of 2D codes. Thus, a digital display can show a sequence of two-dimensional codes (through an animation), which uses the time as a third dimension. Since most of previous codes use black and white patterns to represent information, extended models can also consider the use of multiple colors to increase their storage capacity. Hence the investigation area is about how to represent animated 2D codes and which algorithms can be applied in their recognition.

Using high-capacity 3D codes may open a new branch of research. The development of this novel technology involves inter-disciplinary fields of Computer Sience. In the first step, the content must be encoded to generate the animated 2D codes. In the second step, the digital camera must film the animated code and extract the sequence of two-dimensional matrices. This step requires advanced Image Processing/Recognition techniques. In the third step, a decoding process, similar to the encoding, is required to get the content back.

In order to complete the previous three steps, two main entities are needed: the *sender* and the *receiver*. The former must be able to reproduce, through a *digital display*, the sequence of 2D codes. It is important to notice that these codes might be generated by the sender or a third-party device, e.g., computer. The latter must be equipped with a *digital camera* and a software to decode the animation. Figure 1.3 illustrates this process. In the rest of this thesis, *encoder* defines the algorithm that reads a content (a) and generates an animated 2D code (b), the *recognizer* the algorithm that reads a sequence of frames (captured by the digital camera (c)) and generates a sequence of 2D matrices (d), and the *decoder* the algorithm that reads the sequence of 2D matrices and outputs the original content (e).

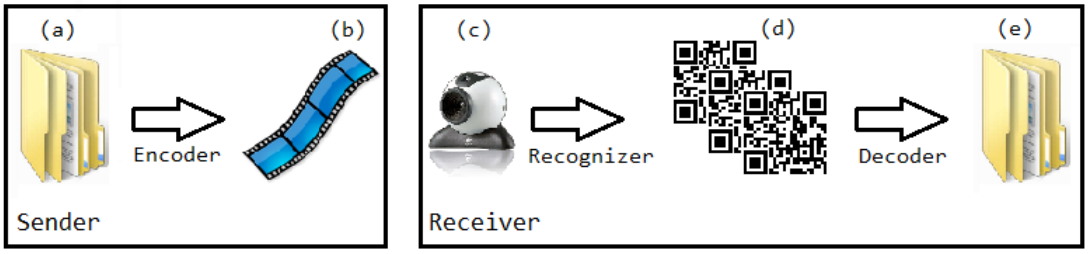

Figure 1.3: Workflow from content encoding to decoding

The construction of the aforementioned system fosters different kinds of applications. On one hand, a user with a personal computer and a portable device equipped with a digital camera, such as a smartphone, can transfer files from the PC to the smartphone even without cables or Bluetooth connectivities. On the other hand, show windows and museum galleries, for example, can take advantage of high-capacity 3D codes to send relevant, dynamic information to tourists holding a smartphone. Accordingly, this thesis is motivated by the need for an efficient algorithm to recognize animated 2D codes, which allows digital displays to transfer large amounts of data in a reliable way.

1.2 Description Of Work

This thesis endorses two areas of work. The first includes the design and implementation of the recognizer module. The second includes the implementation of the decoder module. It is assumed that the encoder is already implemented, that is, the input for the system is always an animated 2D code, not the raw content. The focus of this thesis is on the recognition algorithm. Initially, the objective is to have the complete workflow running, i.e., a simple operational version of the recognizer and the decoder modules. Subsequently, the objective is to improve the solution to achieve high transfer rates, even in adversarial conditions.

Since the encoder is assumed to be previously defined, the investigation work shall start from the recognizer module. Recognizers employed by QR codes may be used as a base model to the one proposed in this thesis description. Error correction techniques are out of the scope of this thesis and can be ignored. Furthermore, a more simplistic encoder, which only represents the content as a sequence of bicoloured 2D matrices (like a chessboard), can speed up recognition algorithms and shall be considered. It is desirable to consider at least these two representations in this work.

The prototype shall be compatible with at least one Android device. The application is supposed to read an animated 2D code and exhibit the resulting content. The encoded contents are provided with their hashes and the decoded contents must produce the same hashes to be considered valid. The evaluation shall be complete in the sense that adversarial conditions need to be considered, e.g., low light conditions. Furthermore, the system boundaries shall be pushed in order to achieve high transfer rates between the sender and the receiver.

1.3 Thesis Goals

Driven by the description of work outlined, the following determine key goals for this thesis:

- Make it (the workflow) work.
- Make it work fast (condition: acceptable reliability).

These two overall goals can be divided into the following three more specific goals:

- **Definition of the format of the 2D matrices:** The input for the system is expected to be image frames, i.e., animation, which are captured by a digital camera. These frames store the data to be decoded. A format shall be defined to enable recognition algorithms to decode information. As a starting point, regular QR codes shall be used, but additional formats shall be considered too. The main goal is to ease the image recognition and decoding process. The adoption of error correction codes is not a requirement.

- **Implementation of the recognizer and the decoder:** The recognizer shall read a sequence of image frames from the digital camera, convert it to a sequence of 2D matrices, and the decoder converts these matrices to the original content. In the first step, a matrix of pixels (a frame) is provided and the 2D matrix is generated (following the predefined format). In the second step, the 2D matrix is decoded and the ouput is stored on the device. Since multiple frames shall be provided as input, the decoding process is expected to execute many times. It is important to considered that due to performance issues.
- **Evaluation under adversarial conditions:** The prototype implemented shall be evaluated and documented. The idea is to delineate the efficacy and efficiency of the implemented algorithms. The evaluation shall be conducted using at least one device. The main goal is to quantify the maximum transfer rate achieved by the proposed algorithms.

1.4 Thesis Outline

The next chapters address the aforementioned goals by showing the design choices, decisions, and where suitable also tests scenarios and their results. Chapter 2 gives background information on barcodes in general and shows related work with regard to 3D barcodes. Chapter 3 depicts the implemented research method which has been used to split the complexity of the overall system into smaller, manageable parts. The design choices and decisions concerning the first two specific thesis goals are covered in Chapter 4. Chapter 5 describes the key parts of the implemented prototype. The third specific thesis goal, namely the evaluation, is covered in Chapter 6. Finally, Chapter 7 summarizes the key findings and draws a conclusion. It also shows open issues concerning future work.

Chapter 2

Background and Related Work

The first specific goal of this thesis is to define the format of the 2D matrices. In order to specify what characteristics such a 2D matrix should have, Section 2.1 provides background information on barcodes. Section 2.2 shows an approach with a similar idea to this thesis, which could help to achieve the thesis goals.

2.1 Background: Barcodes

This section covers the one-dimensional (2.1.1) as well as two-dimensional barcodes (2.1.2) to see what symbologies are available, what their characteristics are and how they are recognized. The focus here is on the advantages of the different symbologies, which finally should lead to an effective symbology for this thesis.

2.1.1 Linear (1D) Barcodes

There are dozens of different linear barcode symbologies that have become a standard in various areas of application. The name one-dimensional or linear barcode describes the way they are recognized respectively read — in one dimension, presumably from left to right. Since the recognition algorithms are not open source and even not part of the standards of a given symbology, it is hard to find out how they really recognize a barcode. But it can be assumed that all one-dimensional barcodes are similar concerning their recognition technique. The recognition algorithms at some point need to start by detecting and analysing the starting bar on the left side.

The linear barcode formats have also other characteristics – like character set, specified length, check digit, etc. – which need to be analysed as well.

- **Color:** All linear barcodes use two colors to represent information, the color of the symbol and the color of the background (where the symbol is printed on) [21]. There is one restriction in the choice of these colors. Nevertheless The Global Standards

1 specification states that the contrast difference between the two colors has to be at least 20% [10]. An interesting fact is that no 1D barcode format uses different colors to increase the storage capacity.

- **Symbology:** Barcodes are called so because bars are used to store information. Technically the thickness of the bars and spaces between bars build up a readable code. There are basically two alternatives how 1D barcodes are composed. The most popular barcodes (the EAN and UPC variations among others [10]) use start and stop characters to identify where the code starts or respectively ends. In the case of different code sets (like in Code 128 [5]) they are also used to describe which code set has been used to encode the information. Some barcode formats (i.a. DP Leitcode[1] [19], Australian Post Custom [19]) do not use this indication and represent only the content in form of bars.

- **Check digit:** The majority of linear barcode symbologies include check digits to verify if the content was read (or written) correctly or not.

- **Length:** EAN and UPC and their variations as the most popular barcodes are able to encode a fixed number of characters. EAN13 and UPC-A for example have both a length of 12 digits. EAN128 (or GS1-128) [10] supports a length up to 48 data characters. Other symbologies have a variable length but are nevertheless limited to an upper bound [21, 10].

- **Character set:** Most 1D barcodes represent only numeric characters (i.a. EAN8, EAN13, UPC-A, UPC-E, ISBN [13]). There are also formats which have a larger character set and support a subset of alphanumeric characters (Code 39, Code 93 [21]) with a few being able to represent even the whole ASCII table[2], which means ability to represent up to 128 characters (i.a. Code 128, 3 of 9 Ext [21], EAN128).

2.1.2 Matrix (2D) Barcodes

The amount of two-dimensional barcode symbologies is not as high as the amount of their forerunner, the 1D barcode. However, they became very popular due to their higher storage capacity. Compared to linear barcodes, 2D barcodes represent information in a two-dimensional area with the important difference that multiple rows contain data. In contrast to 1D barcodes, matrix barcodes vary in their symbology, that means the way they represent information. There are different approaches of two-dimensional barcodes.

QR Code [7], DataMatrix [10] and Aztec Code [1] build up one group of matrix barcodes, which use a similar approach. They contain dark and light square data modules — hence do not use different colors — to represent information. An important characteristic of these three symbologies is the use of a finder pattern. The following three figures show example barcodes for the three symbologies mentioned, highlighting the finder pattern in red color. Figure 2.1 shows the finder pattern of the QR Code[3], Figure 2.2 shows the

[1] Utilization from customers of "Deutsche Post AG"

[2] see the ASCII table at `http://www.asciitable.com/`

[3] *Data Transfer 3D Code* generated by `http://qrcode.kaywa.com/`

one of the DataMatrix[4]. The Aztec Code has beside of the finder pattern also orientation marks which can be used to determine the angular orientation of a barcode [1]. This is a characteristic that QR Code and DataMatrix do not have. Figure 2.3 shows the finder pattern (in red) and the orientation marks (in green) of an example Aztec Code[5]. Another characteristic that 2D barcodes have is a blank quiet zone outside the bounds

Figure 2.1: QR Code finder pattern

Figure 2.2: DataMatrix finder pattern

Figure 2.3: Aztec Code finder pattern

of the symbol to distinguish the symbol from other data around. While QR Code and DataMatrix have such a quiet zone [10, 8], the Aztec Code symbology prescribes no quiet zone around the symbol [20]. QR Codes have the highest storage capacity in this group which is 2953 Bytes [9]. Aztec Codes have a maximum storage capacity of 1996 Bytes [1] and DataMatrix 1556 Bytes [10]. All these capacities refer to the largest version of the corresponding symbology.

Another group of matrix barcodes with a slightly different approach is made up by Moseycode [16] and MaxiCode [23]. They also contain dark and ligth data modules (or greyscale contrast), but in comparison to the first group, these two matrix barcodes use other symbols like squares. Moseycode uses dots for the data as well as for the finder pattern [16]. MaxiCode on the other hand uses hexagonal modules for the data and circles for the finder pattern [3]. Figure 2.4 shows an example of the Moseycode. The finder pattern is highlighted red, the orientation marks are green. Figure 2.5 shows an example of the Maxicode[6]. The red marked circles are the finder pattern, also referred as ”bulls-eye”. MaxiCode uses also orientation marks to find the right rotation of the barcode. The green cells are always black and the yellow ones are always white. This means that 18 modules are responsible for the orientation [23].

Moseycode has a raw data capacity of only 96 bits, but it is only used to support realtime spatial interaction [16] and not to store information. MaxiCode on the other hand has a storage capacity of about 93 Bytes (93 ASCII characters each stored in 8 bits [18]).

A matrix barcode with properties that the other 2D barcodes do not have is the Color Ultra Code [14]. It is the only 2D barcode which uses different colors to encode information. The columns use additive and subtractive colors alternating [14]. This means the first column uses the additive colors red, green and blue, the second the subtractive colors cyan, magenta and yellow, the third column again the additive colors and so on. Doing so it

[4] *Data Transfer 3D Code* generated by `http://datamatrix.kaywa.com/`
[5] *Data Transfer 3D Code* generated by `http://www.netzwolf.info/barcode/aztec/`
[6] *Data Transfer 3D Code* generated by `http://barcode.tec-it.com/?LANG=de`

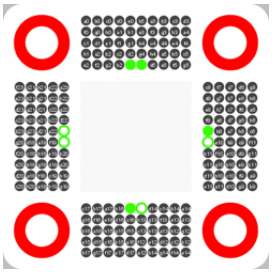

Figure 2.4: Moseycode finder pattern [16]

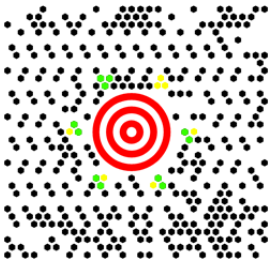

Figure 2.5: MaxiCode finder pattern

can be avoided that the same color stands twice somewhere. Figure 2.6 shows an example Color Ultra Code. The black bars on the left and right side indicate start respectively stop characters. The horizontal black bar is used as orientation mark [14]. The storage capacity of the Color Ultra Code is 1175 Bytes [4].

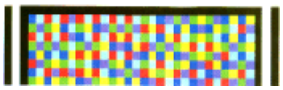

Figure 2.6: Color Ultra Code example [14]

Table 2.1 summarizes the characteristics of the matrix barcodes described above, namely the shape of the data modules, the number of colors they use, the usage of orientation marks, the usage of a quiet zone and finally their storage capacity.

Symbology	**Data modules**	**Colors**	**Orientation marks**	**Quiet zone**	**Storage capacity (in Bytes)**
QR Code	squares	2 (dark and light)	implicit	prescribed	2953
DataMatrix	squares	2 (dark and light)	implicit	prescribed	1996
Aztec Code	squares	2 (dark and light)	explicit	not prescribed	1556
Moseycode	dots	2 (dark and light)	explicit	not indicated	12
MaxiCode	hexagonal modules	2 (dark and light)	explicit	not indicated	93
Color Ultra Code	squares	3 (additive and subtractive colors alternating)	explicit	not indicated	not indicated

Table 2.1: Characteristics of the matrix barcodes

2.2 Related Work

In 2007 Langlotz and Bimber from the Bauhaus-University Weimar invented a 3D barcode, using the time as third dimension by creating an animated GIF of several 2D barcodes[7]. They called their invention *Unsynchronized 4D Barcodes* [17], counting the color as an additional dimension. Using red, green and blue DataMatrix barcodes, they were able to encode three different 2D barcodes simultaneously into each frame of a displayed sequence. Their goal was to increase the robustness of the system by adding redundancy instead of increasing the transfer rate. Displaying every 2D barcode during three frames increases the recognition probability and is less fault-prone [17]. Since their work is based on DataMatrix barcodes, the recognition and decoding techniques depend on DataMatrix recognition, more precisely on the Symbian Semacode library[8] they applied. Unfortunately there can be no insight gained into the recognition and decoding algorithms. It is important to mention that this is the only 3D barcode approach so far.

Since the storage capacity of the 1D barcodes is marginal, they are neglected in the process of this thesis. 2D barcodes overcome the capacity problem and are therefore taken into consideration when it comes to define a suitable format for the 2D matrices later in this report. The *Unsynchronized 4D Barcodes* approach by Langlotz and Bimber has unfortunately a low impact to this thesis. First of all, they used DataMatrix barcodes which prescribes the format and the decoder. This delimits the space of improvement concerning the throughput maximization. Second, the hardware has dramatically changed in the last five years, especially in the mobile phone sector. Therefore, the new hardware allows to research in other dimensions, e.g. increasing the storage capacity of a matrix by using a higher camera resolution. Nevertheless, the approach by Langlotz and Bimber has shown, that the first key goal of this thesis, namely to make the workflow work, is feasible.

In order to achieve the thesis goals specified before, each one has to be designed, implemented, and tested from scratch. The only goal which has an impact from the related work is the definition of the format of the 2D matrices, since the key design criteria can be specified based on existing 2D symbologies.

[7] see `http://140.78.90.140/medien/ar/PhoneGuide/results.htm` for the animated GIF
[8] `http://www.semacode.com`

Chapter 3

Research Method

Considering the workflow in Figure 1.3 (p. 2), the work description in Section 1.2, and the thesis goals (see Section 1.3), the research problem of this thesis can be summarized into the following two main tasks:

- Make the workflow work.
- Make the workflow work as fast as possible.

Given that Langlotz and Bimber already implemented such an application for mobile phones in 2007 (see Section 2.2), they have shown that it can be done, therefore, the first problem in the list above is not a research question anymore in principle. Hence, this thesis will focus on the second problem, i.e. the question of *how high the maximum achievable transfer rate is*. Nevertheless, reliability is still an issue as a sort of left-over from the the first problem and has to be considered too. In order to answer this question, one has to push the boundaries of today's mobile phones, thus, it is an explorative research.

The first step in answering the question of the maximum transfer rate is to apply an overall approach. In order to make the whole workflow as fast as possible, each subproblem needs to be analysed and optimized. The complexity of the overall system – e.g. recognize a 2D matrix, find the modules, identify the colors, has to work on an Android device – needs to be broken down into smaller parts as well. Therefore, the divide-and-conquer approach is an ideal solution for this problem. By analysing the workflow in Figure 1.3, the following list of functional steps can be derived:

- Encode data in a 3D barcode.
- Show 3D barcode on a screen.
- Record 3D barcode for a while with the camera of a smartphone.
- Find relevant content in the recorder material.
- Decode data from the recorded material.

These functional steps can be split into even smaller steps, i.e. atomic steps. Since the first two steps in the list above belong to the sender, only the functional steps on the receiver side are analysed. Using the divide-and-conquer approach, the following atomic steps can be identified for the receiver:

(A) Launch the application and initialize the camera.

(B) Request auto focus and auto white balance from the camera (in order to avoid blurry or too dark/bright images).

(C) Record a 3D barcode from a screen.

(D) Save the recorded data to the file system.

(E) Open the saved (or recorded if saving is not necessary) data in order to process it.

(F) Recognize the 2D matrices.

(G) Decode the 2D matrices.

All these atomic steps are meaningful for the environment of a modern smartphone. The next chapter will shed light on the specific problems to be solved concerning these atomic steps and the range of options being analysed and depicted.

The following describes the explorative aspect of this research problem. First of all, a suitable format of the 2D matrix has to be designed. To break down the complexity of the overall approach, the recognition of the matrices is tested on a desktop computer since it provides an easier debugging functionality. The next step is to test the recognition still on a desktop computer, but with pictures captured with a digital camera rather than bitmaps created on a PC. Findings from these tests lead to improvements of the recognition algorithm and have an influence on the barcode format, i.e. the quiet zone has to be double as large as the bars in the finder pattern.
Once the recognition is reliable enough, the decoder module has to be designed and implemented. This is again done on a desktop computer. Decoding a 2D matrix requires to determine the position of each module in the first place. This has again an influence on the format, it delimits namely the upper boundary of how much modules a matrix may contain. The next step then is to determine the color of each module in a captured frame.
Another explorative aspect is to evaluate which of the three possible ways to obtain data from the camera is the most efficient one. The decision taken has a great impact on the overall solution, e.g. how many frames can be transmitted per second, how high is the throughput.
After the design, implementation and testing of the functionalities mentioned above on a desktop computer, the code has to be migrated to the Android platform. Furthermore, the whole workflow from content encoding to decoding has to be assured to work.

Each of the steps mentioned above comes along with different problems. The recognizer module has to reduce the different colors coming from the camera in order to recognize a 2D matrix. It has also to deal with rotated barcodes. The decoder has to handle distorted barcodes as well, besides of the rotation and color reduction problem. There are also lots

of problems to solve concerning the Android API when it comes to record data with the camera, for example how to grab a frame from a recorded video.

Finally, the algorithms have to be validated under adversarial conditions, e.g. dark and bright conditions. The goal is to quantify the maximum transfer rate achieved by the proposed algorithms. The identification of the bottleneck may speed up future improvements of the prototype.

Chapter 4

Design Choices

This chapter addresses the problems described in the list of atomic steps in Chapter 3. It shows the range of options which have to be analysed and which option was finally depicted to solve the underlying problem, that is, an atomic step. Section 4.1 addresses one of the thesis goals (see Section 1.3), i.e. the format of the 2D matrices. The core parts of this thesis are the recognizer and decoder module, explained in Section 4.2 and 4.3 respectively. Section 4.4 handles the different ways available to obtain data from the mobile phone's camera in order to record a 3D barcode from a screen.

4.1 Barcode Format

Based on the background information about barcodes in Section 2.1 and on the symbologies analysed, Section 4.1.1 proposes characteristics for a symbology with the goal to maximize the throughput of the overall system. After knowing the adequate characteristics, Section 4.1.2 specifies the barcode format designed and used in this thesis.

4.1.1 Deduction For A Good Symbology

Since linear barcodes provide a diminutive storage capacity compared to matrix barcodes, only 2D barcodes are taken into consideration for the proposal symbology. It must be mentioned that the derived characteristics are not more than assumptions. To argue for example which colors are best to use one could write a bachelor thesis about this subject alone.

- **Quiet zone:** The need for a quiet zone depends on the usage intends of a given symbology. If a symbol is expected to stand alone on a given surface no quiet zone would be necessary. Given the case that a symbol may stand between other figures or text than a quiet zone is a must to make it possible to distinguish the symbol's data from other data around. As a drawback this results in less usable space for raw data but makes it easier to recognize and decode the symbol correctly.

- **Finder patter:** A finder pattern is a must for every symbology. The reading or recognizing device needs to know if a barcode was found or if the device is pointing to some random picture or text. With respect to the storage capacity the finder pattern should use as less space as possible. QR Code (see Figure 2.1 on page 7) and DataMatrix (see Figure 2.2 on page 7) both use a finder pattern which delimits the symbol's boundaries implicitly — in contrast to Aztec Code (see Figure 2.3 on page 7) whose finder pattern is in the center of the symbol. Based on the fact that QR Code and DataMatrix have the highest storage capacity, a symbology with the goal to maximize the throughput should use a similar finder pattern.

- **Orientation marks:** Another benefit of the QR Code and DataMatrix finder patterns is that they imply the orientation of the symbol, hence no additional orientation marks are needed. This allows to use more data modules for raw data instead of wasting some of them for orientation marks. Considering the storage capacity of a symbology orientation marks should be made obsolete by using an adequate finder pattern which implies the orientation of the symbol.

- **Data modules:** Section 2.1.2 presents 2D barcodes with different data module shapes like squares, dots and hexagonal symbols. There are several reasons to favour squares over dots and hexagonal symbols. First, squares are easier to display by LCD displays for very small modules, since the sender in the data transfer proposed in this thesis will be achieved by LCD displays. Another reason to favour squares is that they use the space provided for the payload in an optimal way. Using dots or circles for example leads to unused or wasted space within the barcode. Finally, the usage of square data modules seems to be the best approach, since most matrix barcodes and at the same time the ones with the highest storage capacity use squares.

- **Colors:** In order to increase the storage capacity of a barcode and therefore may be as well the throughput (depending on the decoding speed and accuracy), colors should be used instead of only dark and light data modules. Since the modules encode a particular number of bits, the number of colors used has to be a power of two. Using 2 colors for example results in 1 bit encoding per module, 4 colors results in 2 bits, 8 colors in 3 bits etc. The number of colors one can use for the modules obviously reaches an upper limit when reliability comes into play. Since the modules need to be captured by a camera in order to decode the barcode, using lots of colors for the encoding results in more difficulties and even in more errors in the decoding part. In short, the more colors are used the less distinguishable are they from each other.

4.1.2 Format Of The 2D Matrices

There is one major requirement on the format of the 2D matrices: it should lead to a high throughput. This coarse requirement can be split into more subtle ones as follows:

- It should provide a fast recognition and decoding.

- It should provide a high storage capacity.

- It should be robust against false positives.

Figure 4.1 shows the invented 2D matrix format based on the barcode characteristics identified and described in Section 4.1.1 and aiming to meet the requirements specified above. The actually encoded data is of no relevance at this point since it contains just random data. The grey border surrounding the barcode is not part of the format but only inserted to emphasize the quiet zone.

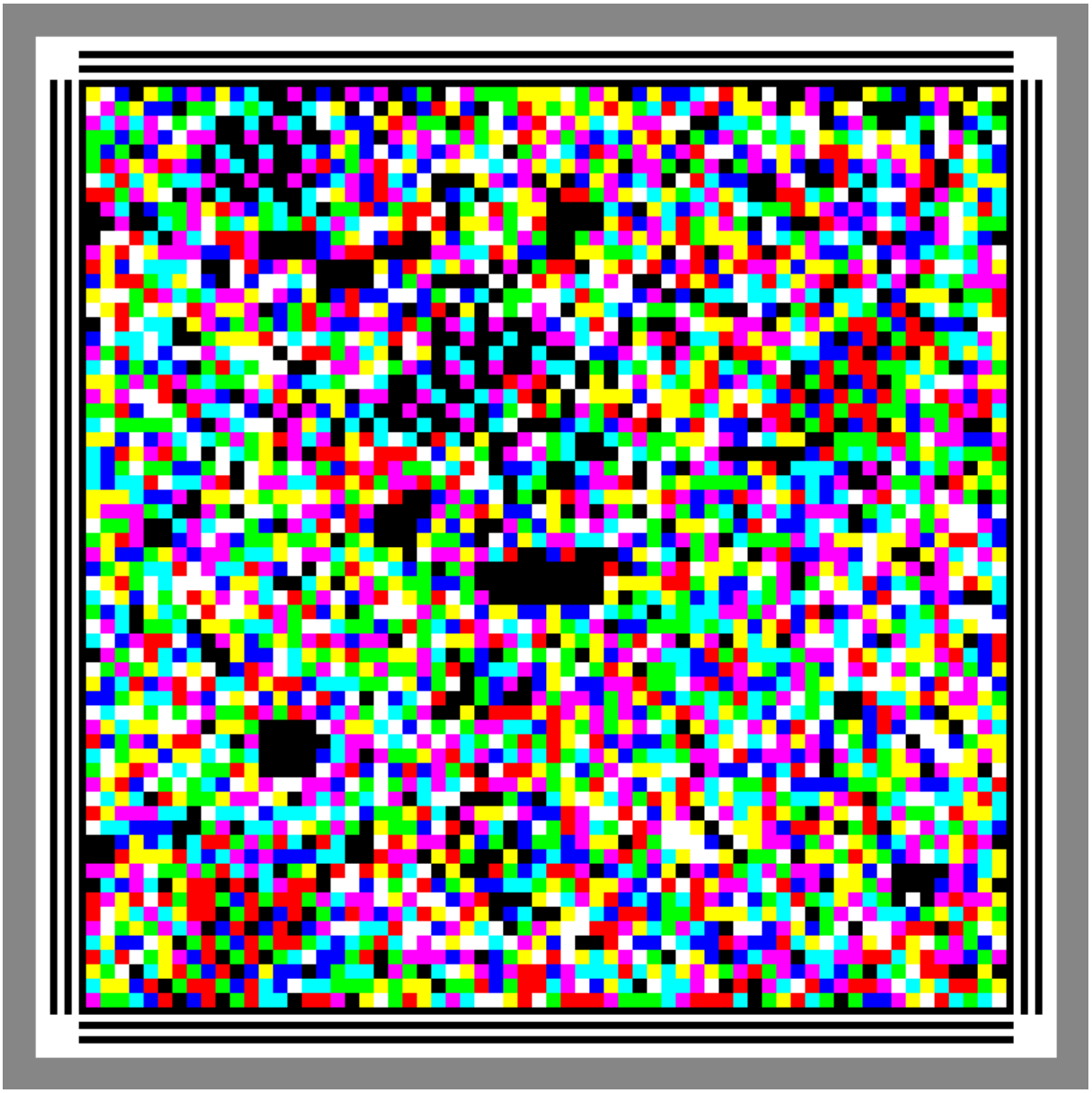

Figure 4.1: Format of the 2D matrices

It is important to mention that the development of the encoder is not part of this thesis. Determining the design criteria and making some design choices – e.g. quiet zone, data modules – has nevertheless been influenced by findings in this thesis.

The following list describes each dimension of this format on the basis of the deductions in Section 4.1.1:

- **Quiet zone:** In order to allow the usage of 3D barcodes in places where there is also other content (e.g. as an animated gif file on a website) and to eliminate interference factors (e.g. other figures or text around the 2D matrix) a quiet zone has been implemented. It does not have to be white like in Figure 4.1 but can be any bright color instead. Dark colors on the other hand have to be avoided since this may lead to misinterpretation (more on this follows in Section 4.2).

  The first version used to have a quiet zone with the same width as the bars in the finder pattern. After some experiments with this prototype it turned out that this format caused too much false positives. Especially the screen borders were misinterpreted as the first black bar of the finder pattern, in cases where the captured frame happened to record not only the 2D matrix but as well the screen border. This can of course appear in other cases as well, for example when there is an additional black bar on one side of the 2D matrix. The solution to this problem is to double the width of the quiet zone. As a result the final version prescribes for the quiet zone the double width as for the bars in the finder pattern.

- **Finder pattern:** The format of the 2D matrices implements a finder pattern which delimits the symbol's boundaries explicitly. At the time when the format was defined, it had not fully been evaluated how much modules the barcode would contain, hence how much such a barcode could gain in size. The bigger a barcode is, the harder is it to recognize a barcode when the finder pattern only consists of reference points, as it is the case for QR Codes (see Figure 2.1, p. 7).

  Tests of this finder pattern (on each side three black and two white bars alternating) in combination with the quiet zone described above have shown, that this format is very robust against false positives. In order to not waste too much space but still be reliable enough, it is sufficient when the bars have only half of the width compared to a module.

  Using this finder pattern also simplifies the implementation of the recognizer module. Since the recognition algorithms of the 2D barcodes mentioned in Section 2.1.2 are all not open source, the recognizer has to be implemented from scratch. This finder pattern assures a simpler recognition algorithm by finding two points on each side of the barcode (see Section 4.2 for the recognition algorithm).

- **Orientation marks:** The invented format does not contain orientation marks. This implies that the recorded matrices are rotated less than 45 degrees – as theoretical upper limit. The field of application of 3D barcodes allows to make this kind of limitations without suffering the loss of quality or usability of the overall system.

- **Data modules:** According to the advantages of square data modules over dots or hexagonal symbols explained in Section 4.1.1, this format prescribes square data modules. Based on the prescription concerning the colors, each module encodes 3 bits. To avoid part of bytes as a result of the decoding, the number of total modules within a 2D matrix has to be divisible by 8. This means that each decoded 2D matrix results in an integral number of bytes.

Furthermore, the format specifies the same amount of modules in the width as in the height of the 2D matrix. The decoder algorithm prescribes, that the number of modules in the width (respectively height) has to be a power of 2 in order to decode the content accordingly (see Section 4.3 for more information).

Therefore, this format prescribes 4096 (= 64*64) modules for each 2D matrix (of course this does not mean that each barcode has to use all of the 4096 modules). This is the largest amount of modules, which still can be decoded reliably. Chapter 6 shows the results of the corresponding tests with a larger amount of modules.

The format does not specify how big a module has to be in terms of pixels. Nevertheless, the recognizer and decoder module rely on a given width of the bars in the finder pattern and the modules inside of the 2D matrices in order to decode a barcode accordingly. The user who is recording the 3D barcode has to ensure that he is close enough to the sender. Chapter 6 indicates in which interval the distance between the sender and receiver can be.

- **Colors:** According to the conclusion on Section 4.1.1 concerning the colors, the number of colors used has to be a power of 2 to encode a sequence of bits. Given the fact that LCD displays (sender) use a red, green and blue cell per pixel to display it [22] and digital camera sensors (receiver) on the other hand section the incoming light – in order to convert the analogue signal into a digital one – again in red, green or blue portions [6], these three colors seem to be best distinguishable. Black and white of course are the most distinguishable colors, since black means no and white full light source.

  Cyan, yellow and magenta – the mixtures of the additive colors red, green and blue – are supposed to be well distinguishable, since each one of this colors is composed by two of the three basic additive colors. So using the additive colors red, green and blue – and their mixtures black, white, cyan, magenta and yellow – 8 different colors are able to encode 3 bits in one data module. Therefore, the format of the 2D matrices prescribes the usage of these 8 colors. Using more colors, e.g. 16, would double the information content, but lead to a worse recognition and more faults.

- **Offset information:** When there is less information to transmit than a barcode is capable of storing (or when the last frame of the 3D barcode is not used to capacity), an offset information is needed to tell the decoding algorithm where to stop reading. For that purpose an offset indication has to be included into the format of the 2D matrices. There are basically two possible ways to address this problem. The offset information can be encoded in the barcode itself or outside of the payload information. The latter alternative has been chosen in order to not waste the payload capacity.

  The four corners of the barcode are used to store the offset information, which is shown in Figure 4.2. Each corner is in one of the 8 different colors. Reading the four corner modules in the right sequence (*A, B, C,* and *D*) results in an integer of 12 bits (since each module stores 3 bits). This 12 bit number in binary representation can be converted to a number in decimal representation from 0 to 4095. This means that 4096 different numbers can be represented by the four corners. The result of the conversation tells the decoder algorithm which module is the last one to read.

The example in Figure 4.2 would result in A=111, B=111, C=111, and D=111[1]. This 12 bit number can be converted to 4095 in decimal representation, which means that all 4096 modules have to be read in this specific 2D matrix.

This alternative with the four corner points has one limitation: it allows a maximum of 4096 modules within a barcode. Given the case that more modules could be processed and therefore encoded, another approach would have to be chosen (e.g. encoding the offset information in the content of the barcode).

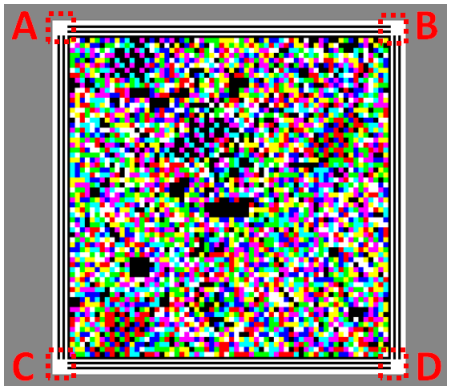

Figure 4.2: Offset information of the 2D matrix format

4.2 Recognizer Module

After defining the format of the 2D matrices, the next step according to the workflow in Figure 1.3 (p. 2) is to think up the recognizer module. The recognizer reads a sequence of frames (captured by the camera) and generates a sequence of 2D matrices.

The recognizer obviously depends on the format specification. It has to know how the format looks like in order to recognize a barcode within a frame. Based on the format shown in Figure 4.1 (p. 17), the main idea for the recognizer is to find two points on each of the four sides of the barcode. If there can be found two points on each side – this means eight points in total – which correspond to the finder pattern, then the actual frame contains a barcode and its exact location can be determined. Once these points are found, one can draw a line through the two points on each side. The resulting four lines intersect in four points, namely the corner points which are important for the decoder module (see Section 4.3). These four lines also delimit the content of a barcode, so the decoder can process the payload. Figure 4.3 shows this explained idea. The red dots represent the two points on each side with the lines drawn through them (in blue). In order to keep the figure simple, there was no content drawn.

When working with frames which were not captured by a camera but created on and loaded from a computer, it is easy to recognize the finder pattern in a frame. To find

[1]To encode the colors a similar approach to the RGB color space is used. Therefore, decoding these eight colors results in the following three bit sequences: black=000, red=100, green=010, blue=001, yellow=110, magenta=101, cyan=011, and white=111.

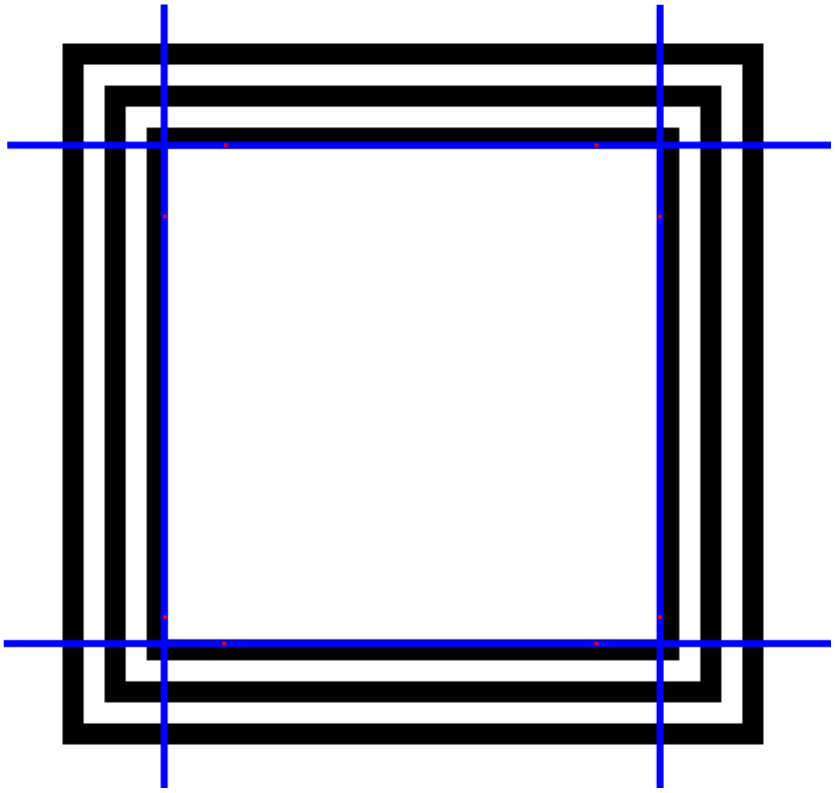

Figure 4.3: Finding two points on each side and drawing the lines

the two points on the top side for example, the algorithm would start on the top of the image and read in two separate columns the pixels sequentially. Then it has to compare if the pixel read is black or white. When an amount of x black (1st black bar), white (1st white bar), black (2nd black bar), white (2nd white bar), and black pixels (3rd black bar) is read, the finder pattern has successfully been found. This process then has to be repeated two times and on the three other sides as well in order to find all eight relevant points. Figure 4.4 illustrates how this algorithm would proceed. The green lines indicate at which columns (or rows) the algorithm reads the pixels. The red dots indicate that at the given point a match was found.

Based on the workflow in Figure 1.3, the receiver has to handle frames coming from a digital camera. And here is where some difficulties appear. First of all, the pixels can not easily be distinguished between black or white. Figure 4.5 shows a detail of a frame. The white bar is grey and the black bar is not clearly black. Besides of the color shift there are also other interferences like green and red areas on this picture. To deal with this inconvenient fact, the input frame has to be processed. Since the recognizer algorithm distinguishes only between black and white pixels, the 16.8 million colors[2] have to be reduced to two. This can be achieved by counting up the values of the red, green and blue channel and comparing the average of the sum to a threshold. A reference value to use as a threshold is 128, since this is the medium of the maximum value 255. If the average is less than the threshold, the pixel is set to black, otherwise it is set to white. The result

[2]Each pixel holds a value between 0 and 255 for the red, green and blue channel. This results in 16'777'216 (= 256*256*256) different colors a pixel can have.

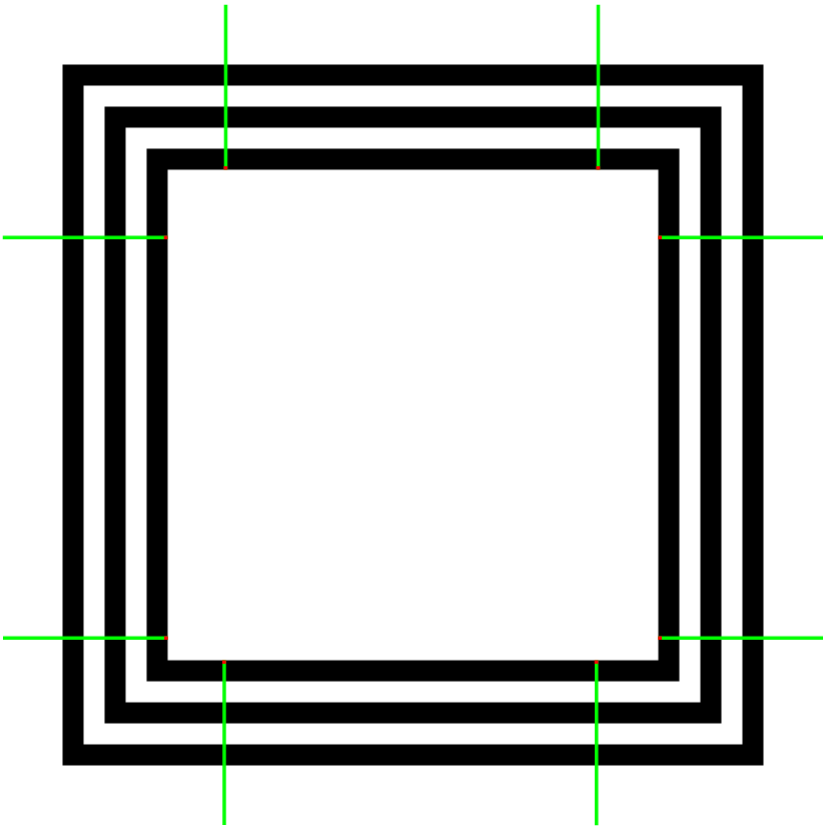

Figure 4.4: Proceeding of the recognizer algorithm

Figure 4.5: Colors when capturing frames with a camera

of this procedure is that each pixel of the frame is black or white afterwards, depending on the original RGB values.

Figure 4.6 shows the interaction of the color reduction and recognizer algorithm explained above on a frame captured by the camera. Before starting the search for the finder pattern,

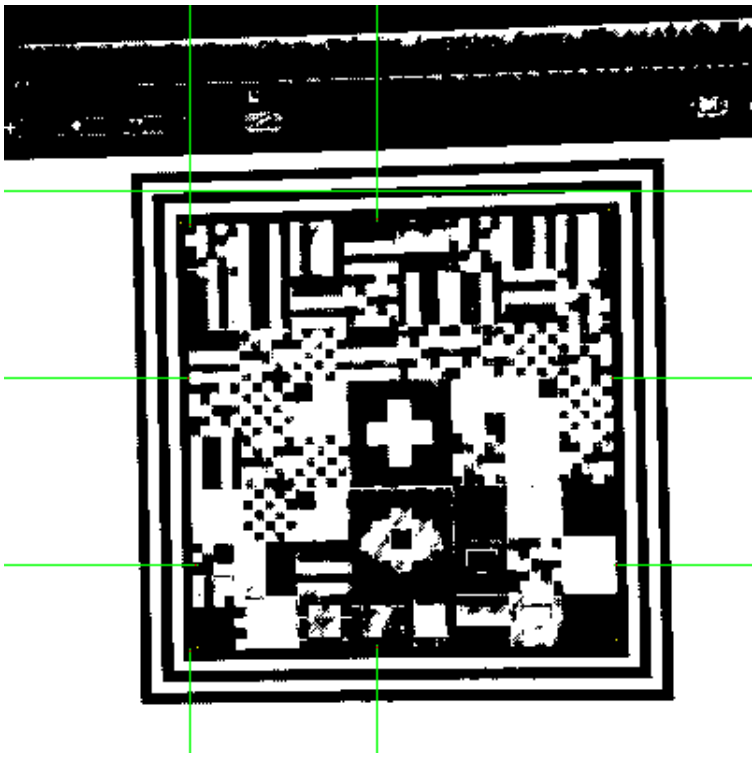

Figure 4.6: Color reduction and recognizer algorithm

the frame's colors are reduced to only black and white. As a result, Figure 4.6 contains only black and white pixels. Then, the recognizer algorithm slices the frame into four parts on each side before starting the search. If for example the frame in Figure 4.6 has a height and width of 400px each, the recognizer would analyse the column at 100px, 200px, and 300px to find the two relevant points on the top – and analog for the three other sides. The recognizer algorithm is again indicated with the green lines. There are also four yellow points on Figure 4.6, which are the calculated corner points. The recognizer makes three attempts on each side in order to find the two relevant points, since one attempt fails if the barcode in the input frame is not large enough. Therefore, this approach requires that the barcode fills up at least 50% of the frame.

A first improvement of this procedure was achieved by changing the color reduction behaviour. Since only the pixels in the corresponding rows or columns are analysed, there is no need to process the whole frame. So instead of reducing the colors of the whole frame before starting the search for the finder pattern, only the pixels in the corresponding column (or row respectively) are processed. Furthermore, the colors are not reduced in advance, but each pixel is processed on demand. This avoids that pixels on a given column/row are processed, but never read. A further advantage of this improvement is that the input frame has not to be changed. Replacing the color values in the memory – or the file system – is not needed anymore. Hence, this improvement saves time.

A weak spot of this approach is the fixed threshold used to separate the pixels into black or white. The threshold 128 obviously worked well for the case in Figure 4.6, but it surely will not work for any input frame. If the sender has a bright light source, then using this threshold would produce too much white pixels. A sender with a dark light source in combination with this threshold would on the other hand result in too much black pixels. In both cases it is unlikely that the recognizer algorithm will find a barcode within a frame. Figure 4.7 shows an example frame which contains only the finder pattern of the barcode. It was captured from a screen using the digital camera of a mobile phone[3], which corresponds to the workflow given. Figure 4.8 shows what the color reduction algorithm produces, when the fixed threshold of 128 is applied[4]. Obviously, the fixed threshold will

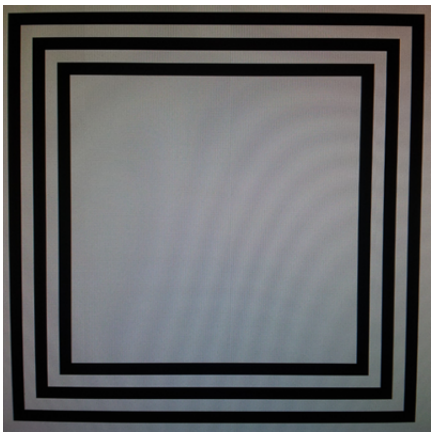

Figure 4.7: Example frame captured by the camera

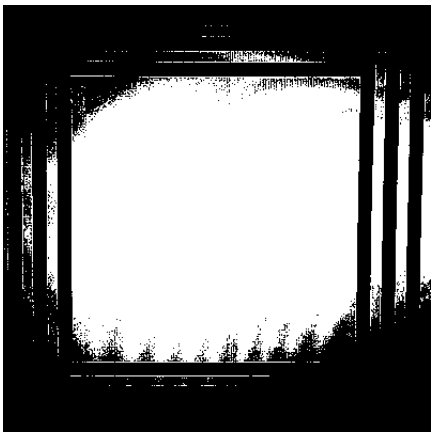

Figure 4.8: Color reduction using threshold 128

not work in practice. To overcome this difficulties another approach is needed. Instead of using a fixed threshold, the recognizer should estimate how dark (or bright respectively) a frame is and apply a dynamic threshold. This new algorithm starts with a small threshold, i.e. 20, the search for the finder pattern. Assuming that the barcode is at least 50% of the frame's size, the recognizer algorithm hits the finder pattern if it analyses the pixels in the middle column. In order to save time, the recognizer attempts to find only the top side of the barcode – the three other sides are ignored. Furthermore, it searches only in the upper 50% of the frame. This also requires that the barcode fills up at least 50% of the frame. The green line in Figure 4.9 shows which pixels the recognizer algorithm analyses at the most in order to estimate the image brightness. If the recognizer cannot find the finder pattern with the given threshold, the threshold is increased by a value of 10. This is repeated until the finder pattern was found or the threshold exceeds the value of 160. If the threshold gets larger than 160, than the frame is not expected to contain a barcode.

No matter which value is used as threshold for the color reduction, there will never be a perfect solution. A perfect solution in this sense means that the black and white bars have always the same width. Therefore, the recognizer algorithm has to take into account that the black bars can be larger than the white ones or vice versa. Figure 4.10 and 4.11 show acceptable outputs based on the input frame (see Figure 4.7). However, the

[3]the built-in camera of the Samsung Galaxy SII

[4]This figure has been created only for illustration purposes. The color reduction algorithm would not

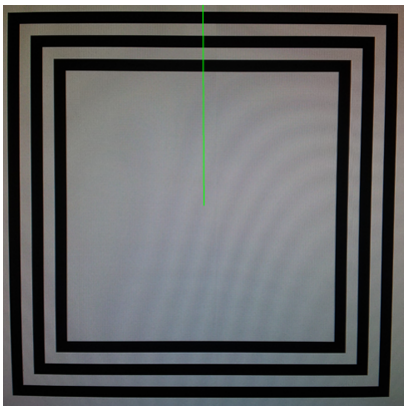

Figure 4.9: Procedure of the brightness estimation algorithm

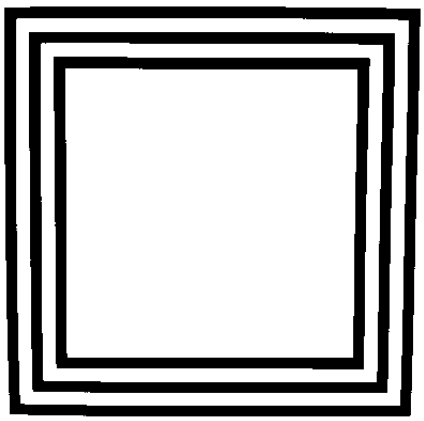

Figure 4.10: Color reduction using threshold 20

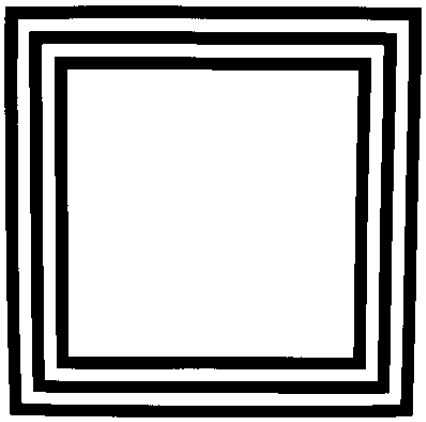

Figure 4.11: Color reduction using threshold 70

recognizer algorithm has to pick the best possible threshold value. In order to decide whether a threshold is acceptable or not, the ratio between the black and white bars has to be compared. Therefore, the recognizer algorithm allows a deviation of maximum 10% between the black and white bars. If the deviation is larger than 10%, then the corresponding threshold value is refused.

After improving the color reduction algorithm, there is another problem to be solved concerning the recognizer. The actual algorithm is fault-prone when it comes to rotated barcodes. If the user rotates the mobile phone while recording the 3D barcode, the 2D matrices will be rotated. When the recognizer attempts to find the two relevant points on any side, it might happen that the points are found on the wrong side. Figure 4.12 shows where the problem of the actual algorithm lies. In this case, the recognizer attempts to find the two relevant points on the top. Therefore, the two red points are supposed to be on the top side of the barcode. Unfortunately, the left point belongs to the left side of the barcode. The corner points needed by the decoder algorithm obviously cannot be calculated accurately by using these points. Instead of constricting the supported degree of rotation, the algorithm has to be improved.

process the whole frame but only the pixels analysed.

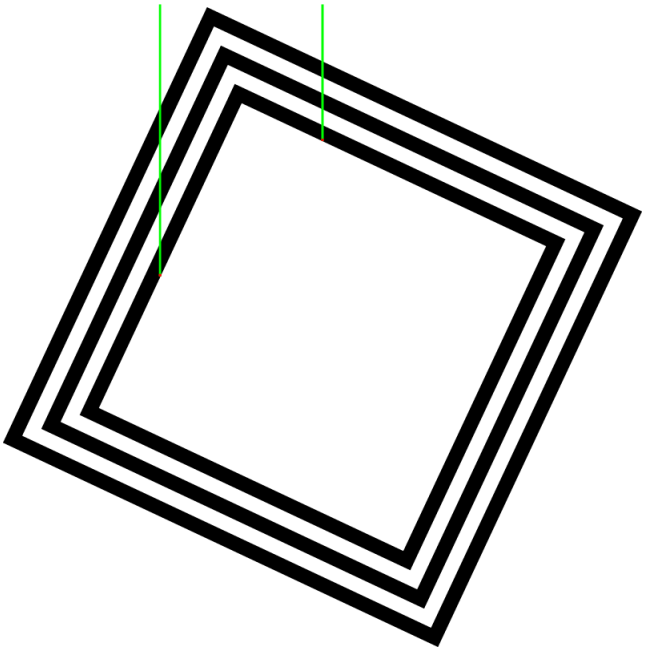

Figure 4.12: Points recognized on wrong side in case of rotations

The example in Figure 4.12 shows that when one point is found on the wrong side, the accumulated width of the bars (three black and two white bars) changes dramatically. The accumulated width on the left side is for example much larger than the accumulated width on the right. This fact can be used to design a new algorithm. Therefore, the new recognizer algorithm starts in the middle column (respectively row) with the search, similar to the threshold estimation approach described above. Figure 4.13 illustrates the new approach. To keep the example clear, the figure shows only the algorithm to find the two points on the top bar. However, the approach works analog to find the points on the three other sides. The brown line indicates at which column the algorithm starts searching the finder pattern, i.e. the middle column of the frame. Since the same column has been used to calculate the threshold for the color reduction, there has to be a match with the finder pattern. Otherwise the algorithm would stop before. The recognizer algorithm then continues searching on the left side of the brown line by moving 10 pixels ahead. The search is continued until the accumulated width of the bars changes dramatically or there is no match with the finder pattern. The latter would be the case on the left side, indicated with the yellow line. The second white and the third black bar are much larger here. Therefore, one point has successfully been found on the top side (indicated by the last green line on the left). To find also the second point, the search is continued on the right side of the brown line using the same technique. Apparently, there are more iterations on the right side. Nevertheless, the algorithm stops at some point on the right side as well (the yellow line on the right). Thereby, the two relevant points on the top bar are found.

The new recognizer algorithm is obviously slower, since there are much more iterations – and they have to be repeated on all four sides. Nevertheless, there is one reason to favour this approach over the previous one, namely the accuracy. Therefore, the improved recognizer algorithm requires that each edge of the barcode is in one quadrant of the frame.

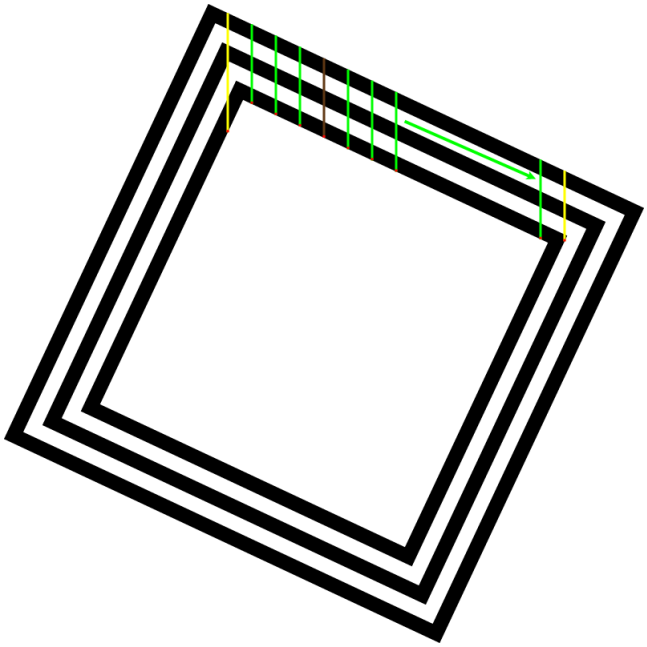

Figure 4.13: New approach to find the two relevant points on each side

One could also ask, if it would be sufficient to find only one point on the left and one on the right of the brown line. The reason why the iteration is repeated so many times is to find two points as far away as possible. This assures that the calculated corner points are more accurate, and these points are crucial for the decoder algorithm described in the following section.

4.3 Decoder Module

Once a sequence of 2D matrices is recognized in a sequence of frames, the last step according to the workflow in Figure 1.3 (p. 2) is to read that sequence of 2D matrices and output the original content. The latter task is accomplished by the recognizer module.

First, the recognizer depends on the format specification. It has to know how much modules are in the width and height of the barcode, how many colors are used to encode information and which color represents which bit sequence. Furthermore, it has to know if it should read all modules or just a part of them, which is indicated by the offset information. Second, the recognizer depends on the corner points of the 2D matrix calculated by the recognizer module.

For each module of the 2D matrix, the recognizer has to solve the following two problems:

1. Determine the position of a module.
2. Read that module.

These two subproblems are described in Subsection 4.3.1 and 4.3.2 respectively.

4.3.1 Determine The Position Of A Module

Since each module has to be read, the position of the modules has to be determined afore. 2D matrices created and loaded from a PC rather than captured by a camera follow a linear rule. This means that each module of a 2D matrix has the same height and width. Rotated barcodes are more difficult to determine the modules positions. But simple trigonometric functions solve these problems. Figure 4.14 shows a rotated barcode example. It contains just the third black bar and only four modules in order to keep the example clear. The points *A, B, C,* and *D* would be the corner points calculated by the recognizer module and are assumed to be calculated. In order to determine the position of the yellow module one would have to find the coordinates of the points *A, E, F,* and *G*. This example shows how one can find the coordinates of the point *E*. The calculations of the other points will not be shown, since the approach remains the same. The hypotenuse

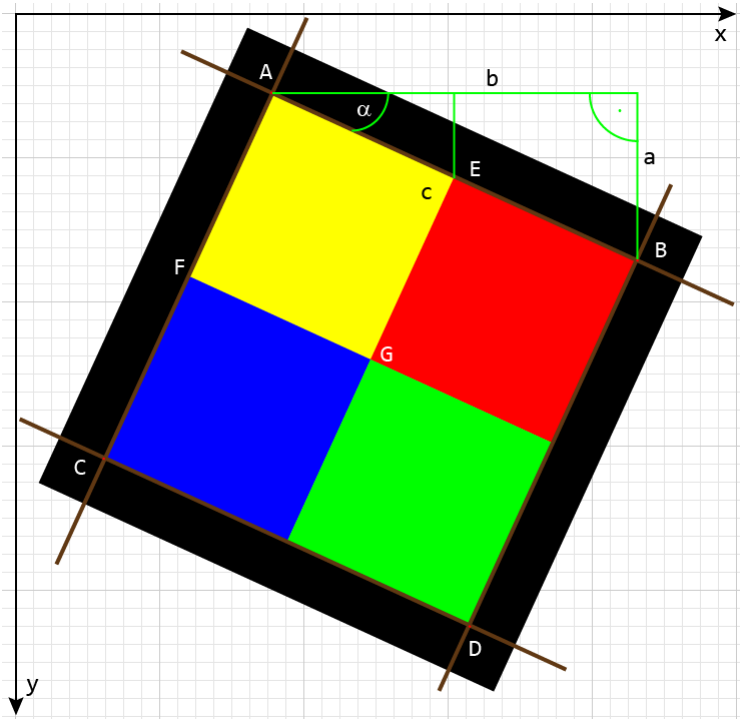

Figure 4.14: Determining the position of a module in a rotated barcode

c can be calculated using the Pythagorean theorem: $c = \sqrt{a^2 + b^2}$. Since the decoder module knows, how much modules the barcode contains in the width, c can be divided by the number of modules to get the width of the yellow module, i.e. the line segment $\overline{AE}$. The angle α can be calculated by solving the following equation: $\alpha = \tan^{-1}(\frac{a}{b})$. Once $\overline{AE}$ and α are calculated, one can figure up the coordinates of the point E as shown in equation (4.1).

$$\begin{aligned} E_X &= A_X + \cos\alpha * \overline{AE} \\ E_Y &= A_Y + \sin\alpha * \overline{AE} \end{aligned} \tag{4.1}$$

These seem to be a lot of calculations, especially when having a barcode with more than four modules. But since a computer is doing these calculations, they are executed fast.

The problem of determining the position of a module arises, when dealing with 2D matrices which where captured by a camera. Besides of rotation, there is another problem to solve, namely distortion. Since it can not be assumed that the user holds the mobile phone parallel to the sender, i.e. the screen, the captured barcodes might be distorted. Figure 4.15 shows the distortion effect using the example of a chessboard. The fields in the first row are obviously larger than the fields in the eighth row.

Figure 4.15: Distortion using the example of a chessboard[5]

To solve this problem, i.e. to determine the positions of the modules, a geometrical approach has been chosen. An interesting fact concerning distorted squares is that the diagonals intersect in the balance point. Drawing a line through the diagonal intersection and the horizontal vanishing point and another line through the diagonal intersection and the vertical vanishing point quarters the square exactly.
In order to illustrate this approach a barcode with 16 modules has been created and captured with a digital camera. To keep the example clear, only the third black bar of the finder pattern has been drawn. The offset information has been neglected as well. Furthermore, the barcode has been captured from an extreme angel in order to have the vanishing points not too far away. Figure 4.16 illustrates this idea. The points *A, B, C,* and *D* are again the corner points of the barcode determined by the recognizer module. Once the decoder algorithm receives these four points, he can proceed determining the modules. The intersection of the line through $\overline{AB}$ and the one through $\overline{CD}$ is the horizontal vanishing point, indicated with the letter *H*. The vertical vanishing point V is the intersection of the lines through $\overline{AC}$ and $\overline{BD}$ respectively. These two vanishing points are crucial for the proceeding of the algorithm and are used for the next steps. The next task of the decoder algorithm is to calculate the balance point. As mentioned before, the balance point is the intersection of the diagonals. Therefore, the intersection of the lines through $\overline{AD}$ and $\overline{BC}$ returns the point *E*. By drawing a line through $\overline{VE}$ and another one through $\overline{HE}$ the barcode gets quartered, this means there are now four smaller matrices. This process of quartering a matrix can be repeated as many times as needed. In the example in Figure 4.16 it has to be repeated four times, namely for each of the four squares

[5]Figure taken from `http://wiki-schacharena.de/images/6/64/Schachbrett.jpg`, accessed on March 26, 2012.

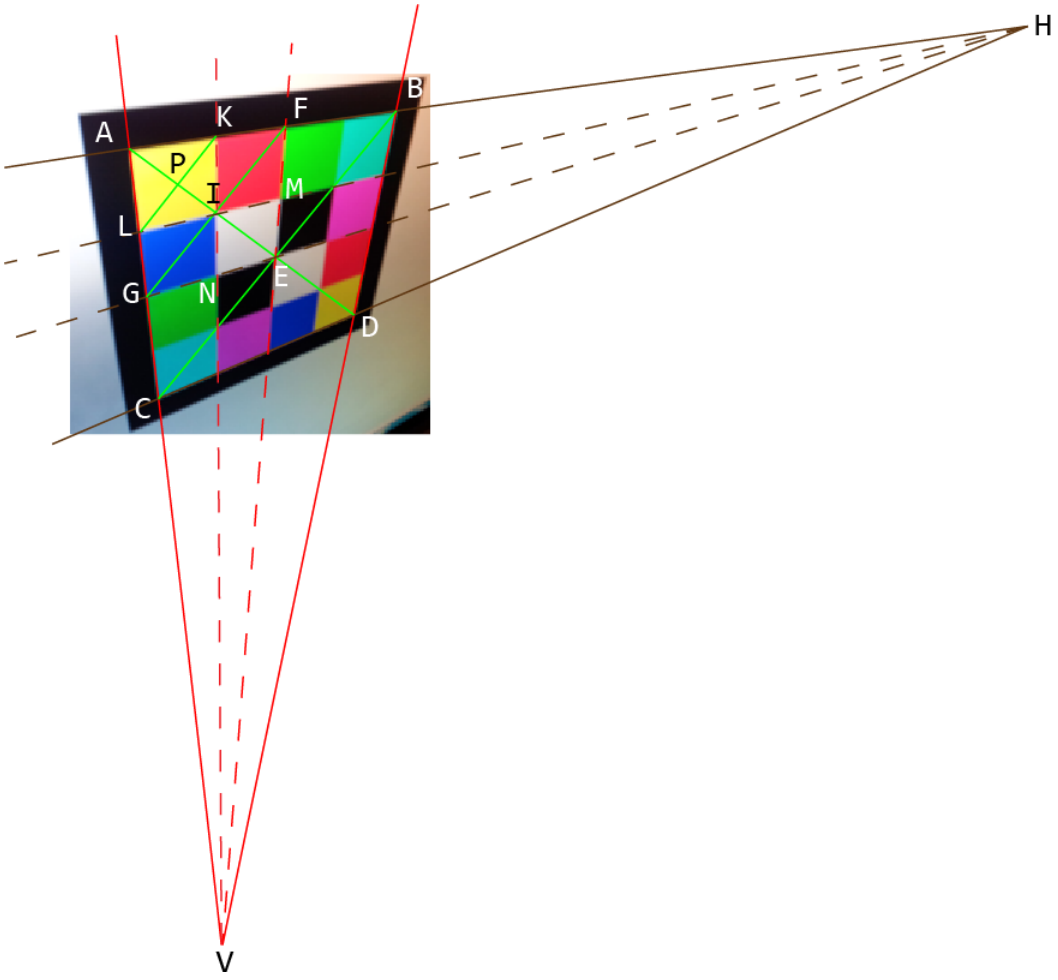

Figure 4.16: Determine the position of modules in distorted barcodes

determined in the first step. Figure 4.16 illustrates this procedure for the top left square (with the corner points *A, F, G, E*). The diagonals of this square (the line segments $\overline{AE}$ and $\overline{FG}$) intersect in point *I*. This matrix can again be quartered by drawing a line through $\overline{VI}$ and $\overline{HI}$ respectively. Based on the number of modules the whole matrix has, the recursive algorithm has reached his end here for this sub matrix. As a result, the modules of the sub matrix *AFGE* have the following corner points:

- The yellow module has the corner points *A, K, L,* and *I*. (*A* was a corner point of the sub matrix and is known, *I* has been calculated as diagonal intersection. The two unknown points so far are *K* and *L*. *K* is the intersection point of the two lines through $\overline{AF}$ and $\overline{VI}$. *L* is the intersection point of the two lines through $\overline{AG}$ and $\overline{HI}$.)

- The red module has the corner points *K, F, I,* and *M*. (The point which has to be calculated here is *M*. It is the intersection of the two lines through $\overline{FE}$ and $\overline{HI}$.)

- The blue module has the corner points *L, I, G,* and *N*. (*N* is the only point which has to be calculated, and it is the intersection of the two lines through $\overline{GE}$ and $\overline{VI}$.)

- The white module has the corner points *I, M, N,* and *E.* (All these points have been calculated before and are therefore known.)

The approach is similar for the other sub matrices and modules and is not explained in detail here.

This is a good example how the divide-and-conquer approach helps solving the problem by dividing it into smaller ones. An advantage of this new decoder algorithm is that rotation has not to be considered. Instead rotation is solved implicitly. But this approach has also two drawbacks. First of all, the number of modules in the width and height of the 2D matrix has to be a power of 2 due to the recursive approach. This is where the constraint specified in Subsection 4.1.2 comes from. Second, the recursive algorithm does not read the modules sequentially, i.e. from left to right. This is not a real problem, since the encoder can write the data in the same order the decoder will read them. Hence, the sequence is assured.

4.3.2 Read A Module

Each module determined in the manner described above needs to be read in order to output the original content of the decoded data. To avoid keeping to much information in the memory of the mobile phone – i.e., the position of each module – each module is subsequently read after its position is determined. Based on the approach with the vanishing points described above, the decoder algorithm uses a similar approach to read a given module. Figure 4.16 illustrates how the algorithm proceeds to read the top left module – the yellow module *A, K, L, I* – in the 2D matrix. The two lines through $\overline{AI}$ and $\overline{KL}$ respectively intersect in the point *P*. This is again the balance point of the given quadrangle. Once the balance point of a given module is calculated, one can apply a matrix to read an amount of pixels. Figure 4.17 shows a detail of Figure 4.16, where a 3*3 matrix is applied to read 9 pixels in that module. The black pixel in the middle of the matrix illustrates the calculated diagonal intersection point. The average of these 9 pixels is than used to determine the color of the corresponding module. Experiments have shown that using the median instead of the arithmetic average results in more inaccurate color detection. The corresponding results are shown in Appendix D. Scenario 7 uses the average approach and has 32 failures in total, whereas scenario 9 with the median approach has 41 failures. An interesting fact about pictures captured with a digital camera is that the pixels do not have the color (or value) they appear to have. There is hardly a pixel in the yellow module which has the correct RGB value, i.e. R=255, G=255 and B=0. The same problem has already been shown in Figure 4.5 (p. 22). Therefore, a couple of pixels of a module have to be taken into account when reading a given module. It is not enough to read only one pixel, e.g. the balance point. A rule of thumb concerning this problem is that the more pixels you read, the more reliable is the result. Nevertheless, there are two problems which have to be taken into account. First, the balancing point of the modules is often not as precise as in Figure 4.17. The larger the reading matrix is, the higher is the risk to cross the borders of a module and therefore to read pixels of an adjacent module. Second, LCD displays show interferences when capturing a frame with a digital camera. These two problems can be examined in the detail of a picture captured

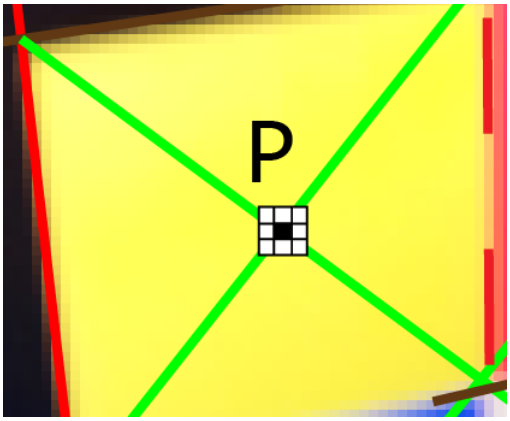

Figure 4.17: 3*3 matrix to read a module

from a LCD display, shown in Figure 4.18. This example applies a 7*3 matrix to read the

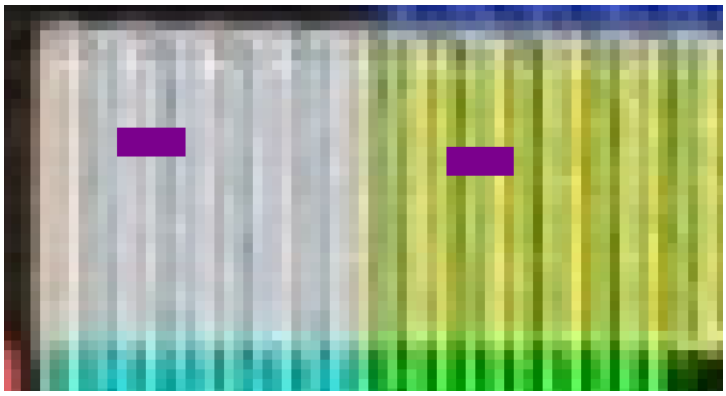

Figure 4.18: Display interferences and inaccurate balancing points

pixels within a module, which is indicated with a violet bar. A 3*3 matrix could point to an interference by chance. In the worst case, the yellow module could be identified as green. These observations lead to the conclusion that a flat bar (i.e. matrix) has to be used when reading the pixels of a module. The usage of a 9*3 matrix has been tested but dropped, since the calculated balancing points might be even more inaccurate than in Figure 4.18. Furthermore, the usage of a 9*3 matrix results not in less failures, which is shown in Appendix D (see scenarios 10 and 11).

As already mentioned, the color determination is a great obstacle when reading modules. The problems the recognizer module had to solve – i.e., the color reduction – are even more complex in this case. The 16.8 million possible input colors have to be reduced to eight colors, instead of only two. As seen in Figure 4.18, the white module for example has different coloured pixels.
The first approach of the color recognition is similar to the first version of the recognizer module, i.e. to use a fixed threshold of 128. The reading matrix returns an amount of pixels belonging to a module. Thereafter, the average of each color channel – i.e. red, green, and blue – is calculated separately. Having a matrix with n pixels, the average value for the red channel of a given module for example would be: $red_{average} = \frac{\sum_{i=1}^{n} red_i}{n}$. If the

calculated average is less than the threshold, it is set to 0, otherwise to the maximum value 255. Repeating these steps for the green and blue averages as well gives the estimated color for the whole module. Based on the fact that the threshold value of 128 has not worked for the recognizer module in order to separate the pixels between black and white, it surely cannot work for the decoder module neither. The 2D matrix may contain black and white modules – besides of others – and these need to be recognized as well. Dark pictures for example will need a threshold lower than 128, bright images on the other hand need a higher one.
The second approach tries to get rid of the fixed threshold. Reference values for black and white help adjusting the brightness of the image and therefore allow to use a dynamic threshold for each captured 2D matrix. Since the recognizer module reads black and white pixels when recognizing a 2D matrix – i.e. the black and white bars – these values can be used. Therefore, the white reference point is the calculated average of the white pixels read, and the black reference point the average of the black pixels read. A characteristic of captured frames is that black areas do not have each color channel equals to zero, nor do white areas have each color channel equals to 255. Hence, the pixels do not use up the whole space from 0 to 255. Therefore, the relative black and white points from a captured frame have to be mapped to absolute values as shown in Figure 4.19. The capital letters B and W stand for black and white respectively. The index letters R,G, and B indicate the corresponding color channel. Since a black (or white) pixel has equal values for each of the three color channels in theory but has different values in practice, they have to be separated for the mapping. The color of each module – which is given by calculating the

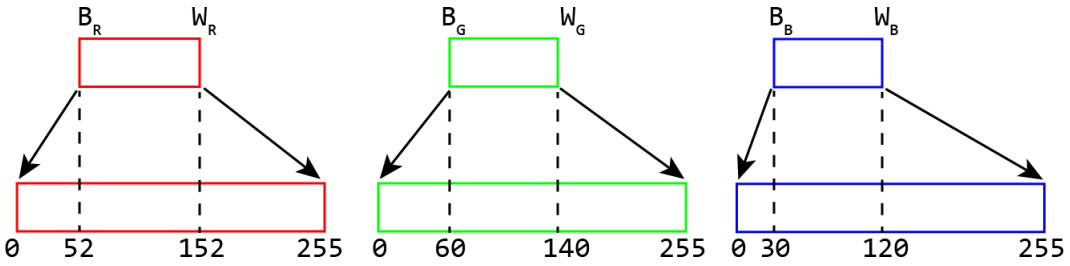

Figure 4.19: Mapping the relative RGB values of the black and white reference points to absolute values

average of the matrix applied – has now to be transformed in order to get the absolute value. Given a pixel x the transformation for each color channel can be conducted using equation (4.2). Again, the indices refer to the color channel and the capital letters B and W to the black and white reference points. The algorithm using equation (4.2) has to assure that the transformed values do not go beyond 255 or beneath 0.

$$\begin{aligned} red_{transformed} &= \frac{255 * (x_R - B_R)}{W_R - B_R} \\ green_{transformed} &= \frac{255 * (x_G - B_G)}{W_G - B_G} \\ blue_{transformed} &= \frac{255 * (x_B - B_B)}{W_B - B_B} \end{aligned} \tag{4.2}$$

In order to identify which threshold has to be used to differentiate the transformed colors, a barcode with 4096 modules has been created and thereafter captured by a camera. Each

module color has been read with the approach described above, and the values have been plotted in a 3D diagram. Figure 4.20 shows the 3D diagram. Each cluster represents the modules with the same color, e.g. the yellow triangles in the diagram represent the yellow modules in the 2D matrix. However, to make the white modules visible, they are drawn in grey. To specify the thresholds for the different color channels, a more detailed view is

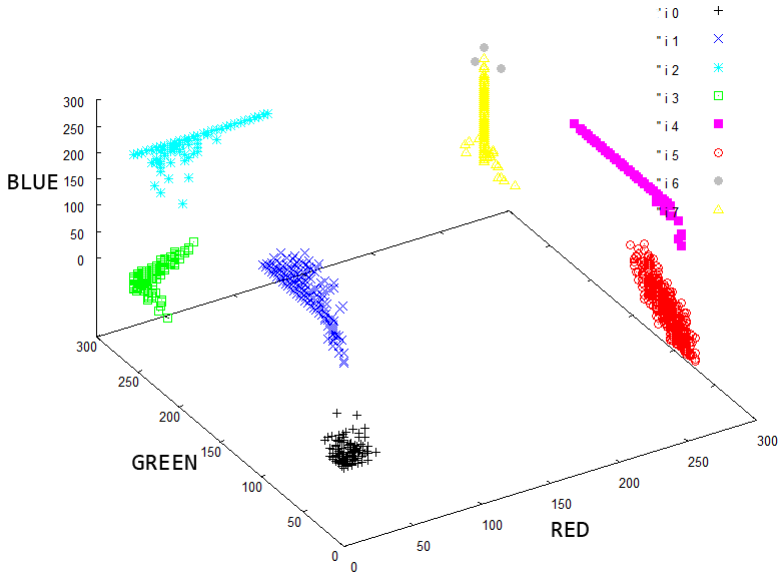

Figure 4.20: 3D diagram of the module colors

necessary. Figure 4.21 shows a detail of that diagram with the red color channel on the x-axis and the blue channel on the y-axis. The green channel is neglected in this figure. What is visible in this figure is what has been stated before. There are for example a

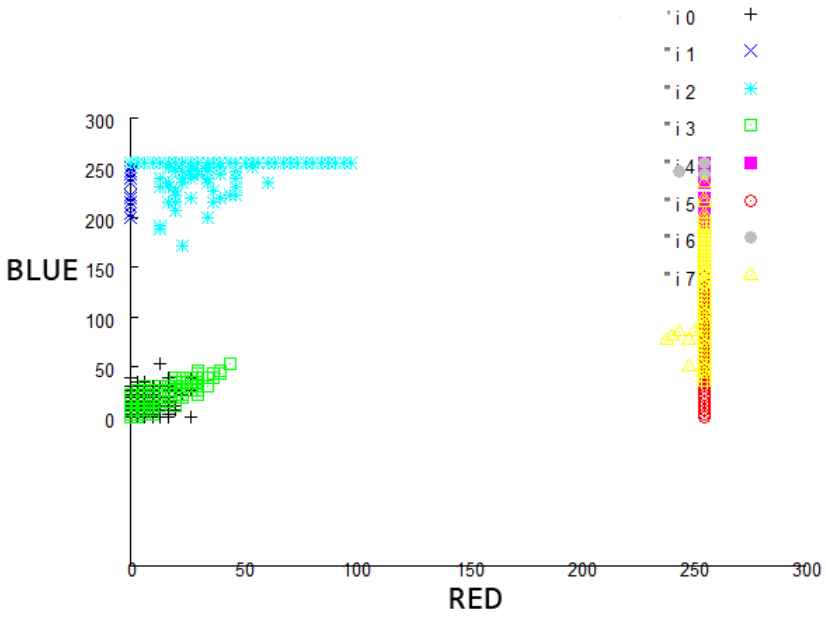

Figure 4.21: Plotted module colors in a blue-red diagram

lot of cyan modules, which have a relatively high value in the red channel (about 100),

even if cyan is not composed of red in theory. The pleasant conclusion however is, that the modules which have a red channel in their color are well separated from the others. Based on Figure 4.21 and the test results shown in Appendix D, the value 160 is used as threshold to distinguish if a module is composed of red or not. Figure 4.22 shows a detail of the 3D diagram, but this time with the green color channel on the x-axis instead of the red one. The y-axis is unchanged, that means it represents the blue channel. Therefore, the red channel is neglected in this figure. The clusters are not so well separated in this

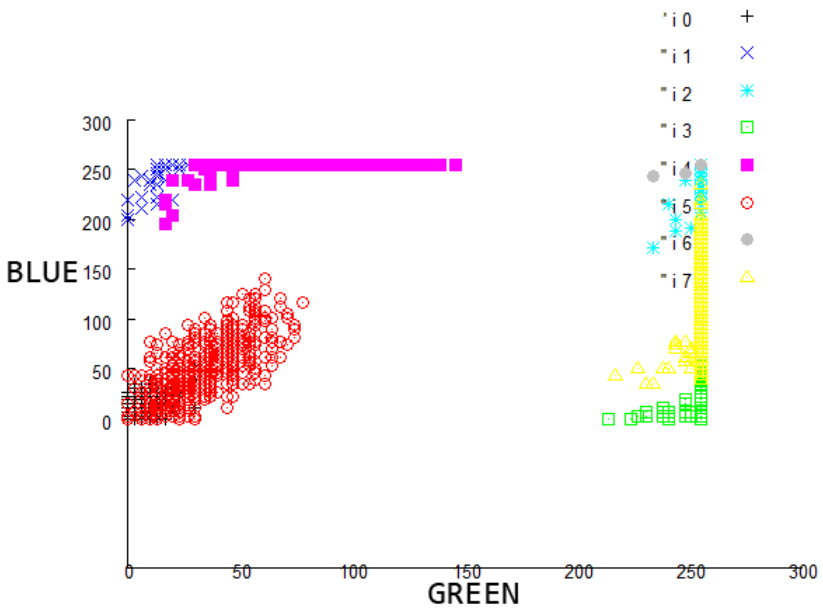

Figure 4.22: Plotted module colors in a blue-green diagram

figure. There are for example a lot of magenta modules with a very high value in the green channel, even if they are not composed of green. Nevertheless, there can still be determined a threshold for the green channel in the manner that it separates the clusters which really are composed of green from the ones which are not. Figure 4.22 and the test results in Appendix D show, that the threshold for the green value has as well to be set to 160. The last threshold to determine is the one for the blue channel. Unfortunately, there is no threshold which would serve for all clusters. This can be observed in Figure 4.22. One cannot draw a horizontal line in the manner that it separates the clusters which are composed by blue (e.g. white and cyan) from the ones which are not (e.g. red and yellow). Therefore, this problem has to be split into the following four sub-problems:

a) Red $\leq$ 160 and green $\leq$ 160

b) Red $\leq$ 160 and green $>$ 160

c) Red $>$ 160 and green $\leq$ 160

d) Red $>$ 160 and green $>$ 160

Each of these sub-problems needs a special treatment concerning the blue channel. That means that each of the cases *a-d* must have a different threshold value. Case *a* has two

possible outputs: if the blue value is beyond the threshold, the module is blue, otherwise it is black. This is shown in Figure 4.24. Case *b* has as well two possible outputs, namely cyan if the blue value is beyond the threshold, and green if the blue value is beneath. Figure 4.23 illustrates this correlation. Based on these figures and the test results in

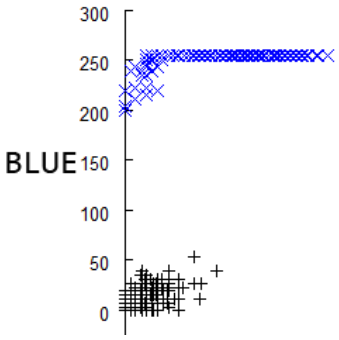

Figure 4.23: Distinction between black and blue

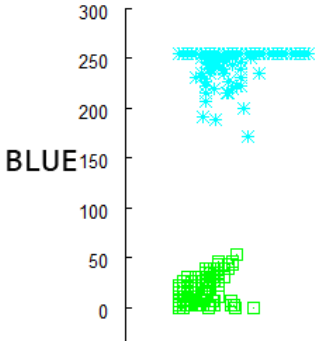

Figure 4.24: Distinction between cyan and green

Appendix D, in both cases a threshold of 150 is applicable for the blue channel. The result of case *c* is magenta or red, depending on whether the blue value exceeds the threshold or not. Figure 4.25 illustrates the range of possible values which can be chosen as threshold for this case. Since there are red modules with the blue value around 150, this threshold has to be slightly higher. Based on the tests shown in Appendix D, a threshold of 165 returns the best possible results. The most difficult sub-problem is given in case *c* – the distinction between white and yellow. As shown in Figure 4.26 the threshold has to be close to 250. The bigger problem is, that there is an overlapping. If for example a threshold of 250 is chosen for the blue channel, then it happens that white modules are mapped to yellow and vice versa. To overcome this problem, a different approach has turned out to be successful. In order to solve this sub-problem, the original RGB values – that means the values before adjusting the brightness based on the black and white reference points – are taken into consideration. These original values show that even if the values of the three color channels vary for a white module, the ratio between them is still in a given range. The tests in Appendix D have shown that if the original blue value is higher than 70% of the lowest value of the two other channels, than it has to be a white module. In mathematical terms if the condition $b_{original} > 0.7 * min\{red_{original}, green_{original}\}$ is true, then it is a white module, otherwise it is a yellow one.

Since each frame (or 2D matrix) is brightness adjusted using the black and white reference points, the thresholds described above are valid for each captured frame. The thresholds can be summarized as follows[6]:

- $Threshold_{red} = 160$
- $Threshold_{green} = 160$
- rg (black or blue): $threshold_{blue} = 150$

[6]A capital letter means that the corresponding value is above the threshold. For example 'rG' means that red is below the threshold and green is above the corresponding threshold.

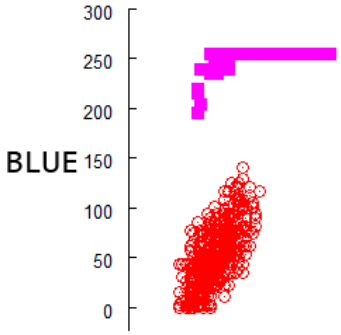

Figure 4.25: Distinction between magenta and red

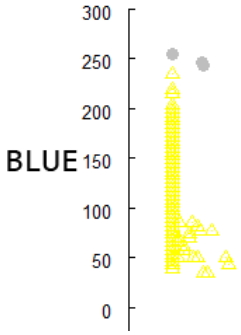

Figure 4.26: Distinction between white and yellow

- rG (green or cyan): $threshold_{blue} = 150$
- Rg (red or magenta): $threshold_{blue} = 165$
- RG (yellow or white): $threshold_{blue} = 0.7 * min\{red_{original}, green_{original}\}$

It has to be mentioned, that these thresholds were tested only on one screen. In order to specify whether these thresholds are generally valid or only for the tested screen, further tests are inevitable.

4.4 Obtaining Camera Data

There are three different ways to obtain data from the built-in camera of an Android mobile phone, namely via:

- preview frames (subsection 4.4.1),
- image capturing (subsection 4.4.2),
- or video recording (subsection 4.4.3).

This section covers the power of each approach as well as its problems in the corresponding subsection. Subsection 4.4.4 then shows which approach has finally been selected to proceed with in this thesis.
In order to achieve the highest possible throughput, each of the three approaches mentioned above needs to be tested and experimented with. The research method mentioned in Chapter 3 – the divide-and-conquer approach – helps finding the approach with the highest throughput. The main idea is to slice each of the three possible ways to obtain data from the camera in its atomic steps, to optimize them and to measure the output for a given time period. For these three tests the atomic steps of each approach were put together into a sequence of 30 seconds to make the throughputs comparable.
All these experiments were conducted using a single thread, that means each task is executed sequentially after the previous one has finished. The barcodes contain random data. Furthermore, the mobile phone has been fixed while recording.

4.4.1 Preview Frames

When starting the camera application on an Android device live pictures of the environment – or more precisely of the region where the camera is pointing to – are shown on the display. These live pictures are called preview frames. The Android API offers the functionality to receive the preview frames and to process the raw camera data.

To conduct the test series and to examine how high the throughput – in terms of bytes per 30 seconds – is, a suitable barcode has to be created first. Given the fact that preview frames have the lowest resolution[7] compared to images[8] and videos[9], a barcode with less modules has to be used to keep the test comparable and fair. It is obvious that previews do not contain the same amount of information – or can not be used to transfer it – like images due to their much lower resolution.
Given the highest preview resolution of 800*480 pixels, a barcode has to be created and used for the test series in the way that it uses as much as possible of the space available. The optimal case would be to have a square barcode of 480*480 pixels. This can be assured by moving the recording device so close to the sender, that the barcode is still captured entirely. This means that the size of the barcode displayed on the screen is not of importance in this case, it only matters how big the size of the recorded barcode is. Based on the current barcode format (4.1.2) and the decoder module (4.3), this barcode has to meet some requirements. First of all, the quiet zone and the finder pattern has to correspond to the symbology[10]. And second of all, the number of modules in the width (or height respectively) has to be a power of 2. By using a barcode with 8*8 modules, the recorder modules have a width (respectively height) of about 37 pixels. This can be retraced by looking at equation (4.3).

$$\begin{aligned} \underbrace{5 * 0.5 * x}_{\text{finder pattern on top}} + \underbrace{8 * x}_{\text{8 modules}} + \underbrace{5 * 0.5 * x}_{\text{finder pattern on bottom}} &= 480px \\ x &= 36.92px \end{aligned} \tag{4.3}$$

Figure 4.27 shows the barcode which was displayed by a LCD display and captured with the mobile device to conduct the test series.

Figure 4.27: Barcode for the preview frames test series

The next step in the experiment is to divide the workflow into atomic steps. By analysing the workflow of the application when using preview frames to capture the camera data and based on the list of atomic steps derived in Chapter 3, the following list of atomic steps results for the preview frames:

[7] for the Samsung Galaxy SII the highest preview resolution is 800*480 pixels
[8] highest image resolution is 3264*2448 pixels
[9] highest video resolution is full HD, which means 1920*1080 pixels
[10] the padding information was ignored for these tests

(A) Launch the application and initialize the camera

(C1) Request the first preview frame

(C2) Request another preview frame

(E) Create bitmap object

(F) Detect borders

(G) Decode content

Step A measures the time it takes from clicking to start the application until the camera is ready to return the first preview frame. Auto focus has not to be requested explicitly when dealing with preview frames (step B), therefore it is ignored in this test. After some experiments, the results showed that requesting the first preview frame takes much longer than requesting any other frame later. So this is separated into the atomic steps C1 and C2. Saving the preview frames to the file system is not needed, since they have a low resolution and can be kept in memory therefore. This means, that step D is ignored as well. Once the camera returns the raw data it has to be converted to a bitmap object[11] to process it (e.g. iterate over the pixels, get the RGB values of a pixel), which is indicated with the letter E. When the bitmap object is created, the *real* work begins, namely recognizing a barcode within a bitmap (step F) and decoding the content (step G) if a barcode was found.

Measuring the time for each of these atomic steps over 10 iterations results in the diagram shown in Figure 4.28. It shows the arithmetic mean and the standard deviation for each atomic step. The detailed results for this test series can be found on Appendix A.

After calculating the statistical data for each atomic step, the next step in the experiment is to combine each step into a logical sequence of maximum 30 seconds. To solve this problem for the preview frames the following equation has to be solved:

$$\underbrace{A}_{a} + \underbrace{C1 + E + F + G}_{b} + x * \underbrace{(C2 + E + F + G)}_{c} \leq 30s \tag{4.4}$$

Equation (4.4) describes the workflow for the preview frames as follows: First, the application has to be launched and the camera initialized (a). Then, the first preview frame is requested, converted to a bitmap object, and processed to find the barcode and decode it (b). Finally, requesting another frame, converting and processing it (c) has to be repeated x times. Using the arithmetic averages from Figure 4.28 for the atomic steps and equation (4.4), the following inequation can be formed:

$$\begin{aligned} a + b + x * c &\leq 30s \\ a &= 1.1047s \\ b &= 0.9706s \\ c &= 0.7347s \end{aligned} \tag{4.5}$$

[11] This conversion is needed since preview frames are captured in the YUV color space and need to be converted into the RGB color space in order to process the data accordingly. For more information see the Android API reference at `http://developer.android.com/reference/android/graphics/YuvImage.html`

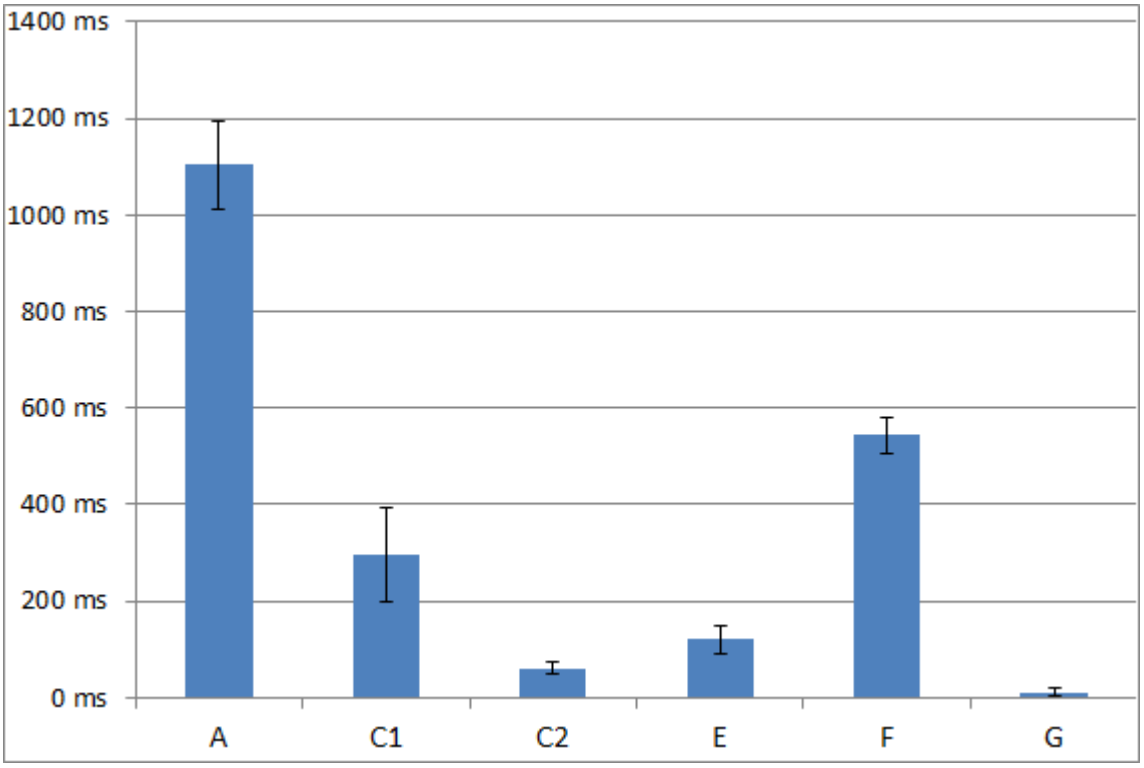

Figure 4.28: Mean and standard deviation for preview frames

Solving the maximization problem for x in inequation (4.5), the maximum value of x is 37. This results in 38 processed barcodes (1 in b and 37 in c) within 30 seconds. Having 38 barcodes with each 64 modules and 3 bit per module, this results in a throughput of 912 bytes (or 0.89 kB) for the preview frames.
The bottleneck in this approach is the recognizer module, i.e. step F (apart from A and B which have to be executed only once, which are part of the Android API and cannot be avoided or optimized).

4.4.2 Image Capturing

The second approach which can be used to obtain data from the camera of an Android device is taking pictures. Similar to the experiment activities described in Section 4.4.1, the first step here is as well to create a suitable barcode. Since the highest image resolution of the Samsung Galaxy SII is 3264*2448 pixels, there can be used a much bigger test barcode. Therefore, the optimal barcode size would be 2448*2448 pixels. Again, this is assured by moving the recording device so close to the sender, that the barcode is still captured entirely. This barcode has to meet the same requirements as the barcode for the preview frames test. Due to the higher resolution, a barcode with 64*64 (= 4096) modules can be used to test the throughput. As a result, the recorded modules have a width of about 35 pixels, as calculated in equation (4.6).

$$\begin{aligned} \underbrace{5 * 0.5 * x}_{\text{finder pattern on top}} + \underbrace{64 * x}_{\text{64 modules}} + \underbrace{5 * 0.5 * x}_{\text{finder pattern on bottom}} &= 2448px \\ x &= 35.48px \end{aligned} \tag{4.6}$$

Since the difference between the modules in the preview frames test and this one is diminutive, the tests and therefore the result in terms of throughput per 30 seconds stays comparable and fair. This is as well the criterion on which the number of modules in the corresponding barcodes have to be chosen. Figure 4.29 shows the barcode which was displayed by a LCD display and captured with the mobile device to conduct the test series.

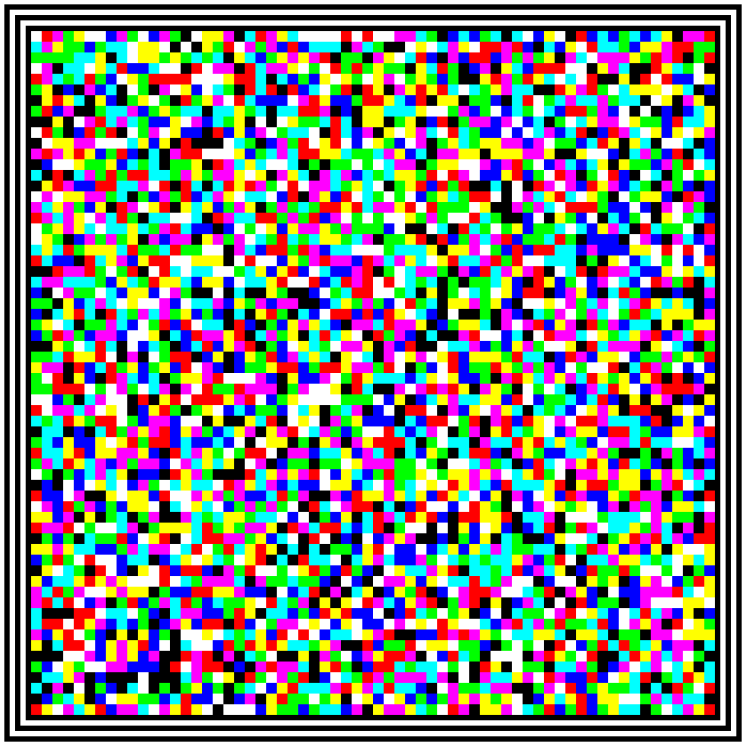

Figure 4.29: Barcode for the image capturing test series

Similar to the tests for preview frames, the second step in this test series is as well to divide the workflow into atomic steps based on the list in Chapter 3. When taking pictures rather than using preview frames, a quite different list of atomic steps can be derived:

(A) Launch the application and initialize the camera

(B) Request auto focus and auto white balance

(C) Capture an image

(D) Save image to the file system

(E) Create bitmap object

(F) Detect borders

(G) Decode content

Step A is the same as the step A for preview frames. An additional step compared to the preview steps is requesting auto focus and auto white balance (B). While this is done implicitly for the preview frames, it has to be requested for images. Other than the preview frames, capturing the first image or a subsequently one does not differ in statistical means. Therefore, there is only one atomic step for capturing images (C). Hence another additional step is saving the image to the file system (D). The Android API offers no functionality to convert a captured image into a bitmap on the fly. It has to be saved to the file system first and then loaded and converted to a bitmap object (E) afterwards. The two last atomic steps are again detecting a barcode in a bitmap (F) and decoding the content (G).

The resulting arithmetic mean and standard deviation of these atomic steps measured over 10 iterations are shown in Figure 4.30. The detailed results for this diagram can be found on Appendix B.

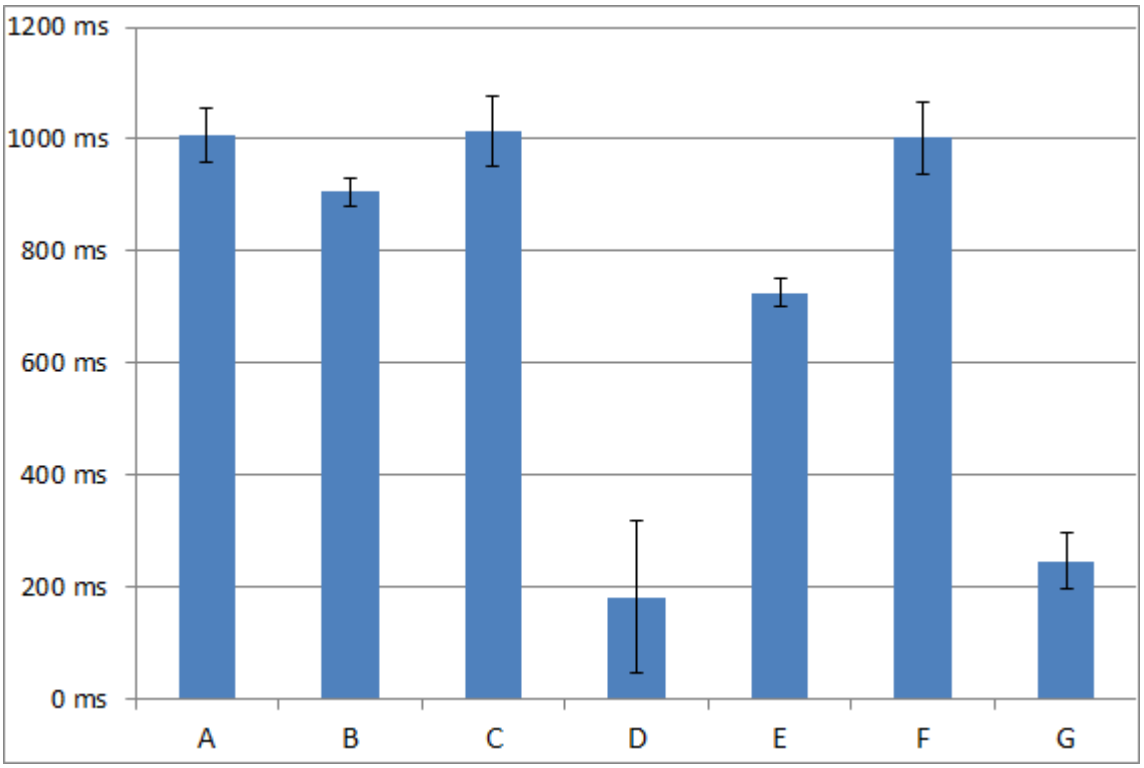

Figure 4.30: Mean and standard deviation for image capturing

To calculate the throughput – in terms of bytes/30s – a slightly different equation compared to (4.4) has to be used, due to differences in the workflow. The combination of atomic steps into a sequence of maximum 30 seconds results in the following inequation:

$$\underbrace{A + B}_{a} + x * \underbrace{(C + D + E + F + G)}_{b} \leq 30s \tag{4.7}$$

Equation (4.7) describes the following workflow of image capturing: First, the application is launched and the camera initialized. Once this is done, auto focus and auto white

balance are requested. Since these two atomic steps appear in the beginning and only once, they are counted together (a). For each barcode, i.e. 2D matrix, the atomic steps C (capture an image), D (save image to the file system), E (create bitmap object), F (detect borders) and G (decode content) has to be executed sequentially. All these steps are summarized as b. To process x barcodes, these steps have to be repeated x times. With the help of inequation (4.8) and the arithmetic averages from Figure 4.30 for the atomic steps, the following inequation can be set up:

$$\begin{aligned} a + x * b &\leq 30s \\ a &= 1.9126s \\ b &= 3.1676s \end{aligned} \tag{4.8}$$

The solution for x in the maximization problem in (4.8) is 8. This means that 8 barcodes can be processed in at most 30 seconds with the image capturing approach. 8 barcodes with each 4096 modules and 3 bit per module means that the throughput of this approach is 12288 bytes (or 12 kB).
The atomic steps with the longest duration in this approach are taking a picture (C) and the recognizer module (F). While step C is an API functionality and therefore can not be optimized, an improvement of step F could be achieved. A solution could be to let run these two steps in parallel. Nevertheless, the throughput of the image capturing approach is more than 13 times higher than the throughput of the preview frames approach.

4.4.3 Video Recording

The last approach in obtaining data from the camera is recording video. The mobile devices nowadays have high video resolutions, the Samsung Galaxy SII for example is able to record video in full HD quality[12]. The idea of this approach is to record a video of the animated 2D barcodes, assuming that the transition rate of the 2D barcodes is equals to the frames-per-second rate of the recording device. After this, each frame can be grabbed and processed. The main advantage of this approach might be to overcome the long burst delay of image capturing (the time it takes between capturing two images) and hence to achieve a higher throughput.

Based on the highest video resolution, i.e. 1920*1080 pixels, the optimal barcode size would be 1080*1080. This requirement can be met by positioning the recording device in the right distance to the screen. Taking into consideration that the optimal size for the barcodes in video recording is about half the size of the images and about double the size of the previews, the number of modules should be somewhere in the middle. Since the number of modules in the width (or height) has to be a power of 2, a barcode with 32*32 (= 1024) modules can be used for this test. This results in a width of about 29 pixels for each recorded module, as shown in equation (4.9).

$$\begin{aligned} \underbrace{5 * 0.5 * x}_{\text{finder pattern on top}} + \underbrace{32 * x}_{\text{32 modules}} + \underbrace{5 * 0.5 * x}_{\text{finder pattern on bottom}} &= 1080px \\ x &= 29.19px \end{aligned} \tag{4.9}$$

[12] 1920*1080 pixels

The resulting module width is slightly smaller compared to them from the previews and images. Therefore, the test results will not be 100% fair. Given the fact, that using another amount of modules would vary even more, this is the only possible way to conduct the test series for the video recording approach. Figure 4.31 shows the barcode used to conduct the test series.

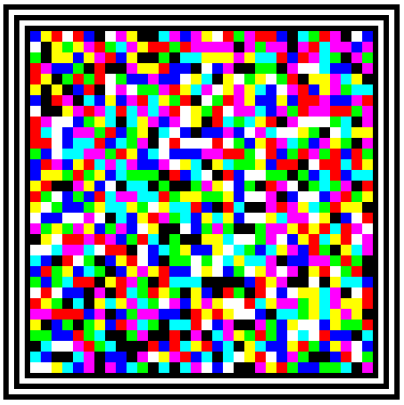

Figure 4.31: Barcode for the image recording test series

After creating a suitable barcode, the second step in the test series is to divide the workflow into atomic steps. Based on Chapter 3 and the assumption in the beginning of this section, i.e. all steps are executed sequentially so no tasks run on parallel, the following list of atomic steps can be derived for the video recording approach:

(A) Launch the application and initialize the camera

(C) Record a video

(E1) Grab a frame

(E2) Create bitmap object

(F) Detect borders

(G) Decode content

Step A in this approach is equals to the one from the previous two approaches, i.e. launching the application and initializing the camera. Auto focus is not an issue when recording videos since it is done implicitly, therefore step B is ignored in this list. Step C varies from the previews and images approaches, since first the whole 3D barcode has to be recorded before starting to process each frame. Step D measures the time it takes to store the recorded data on the file system. However, this step is ignored for this approach. The reason is that the atomic steps C and D cannot be split in order to measure them separately. Each recorded frame is directly written to the file system and is not kept in memory for further processing. Step E on the other hand has to be split into two different atomic steps, namely E1 (grab one frame from the video) and E2 (create a bitmap object).

Once these steps are executed, the 2D matrix has to be found within a frame (step F) and has to be decoded (step G).

To measure all these steps separately, there has an application to be developed – as it had for the previews and images approach as well. Unfortunately, recording and processing video – in contrast to image capturing or requesting preview frames – is not straightforward and causes many problems to solve.
The first problem arises with regard to step C. Applications developed by using the Android API – or more precisely the Android SDK[13] – cannot record videos in full HD quality. This affects among others Android devices also the Samsung Galaxy SII. Even if the hardware of these devices support the recording of HD videos, there is a limitation on the software side. The manufacturers overcome this problem by developing their built-in camera applications in a low-level programming language, i.e. C. Building the test application in this way nevertheless would be too much effort just for the test purposes and was neglected therefore. The alternative would be to use the highest video resolution supported by the Android API in combination with the Samsung Galaxy SII, which is 720x480 pixels. This has also been ignored since it is lower than the previews resolution and would be no benefit.
The second problem arises with regard to step E1 and has its source in the lack of functionality concerning the Android API. It offers one function to grab frames from a video via `MediaMetadataRetriever.getFrameAtTime(long microseconds)`[14]. In contrast to what one might expect, this method does not what it states. It returns only key frames, that means about 1 frame per second. In a video with 30 fps[15] only 1 frame can be grabbed, the other 29 frames are not retrievable. The big advantage of video recording, that is the number of different pictures captured within a second, is lost in this sense. Therefore, a third party software had to be used. According to the tutorial on the website `www.quack-ware.com` [15], FFmpeg[16] can be integrated in order to grab the frames. By doing so, the problem of grabbing frames can be solved.

The video recording approach has nevertheless been tested, since these problems could be fixed in the near future. Unfortunately, the test series could not take into consideration the steps A and C but only measured the execution times for the steps from E1 to G. Therefore, the test scenario is as follows:

1. With the built-in application record a 3D barcode for 30 seconds (despite that it will not be used at whole).

2. With the test application for the video approach measure the times for:

   (E1) Grab a frame

   (E2) Create a bitmap object

   (F) Detect borders

[13] `http://developer.android.com/sdk/index.html`

[14] `http://http://developer.android.com/reference/android/media/MediaMetadataRetriever.html#getFrameAtTime(long)`

[15] frames per second

[16] `http://ffmpeg.org`

(G) Decode content

The resulting arithmetic mean and standard deviation of these atomic steps measured over 10 iterations are shown in Figure 4.32. The detailed results for this diagram can be found on Appendix C.

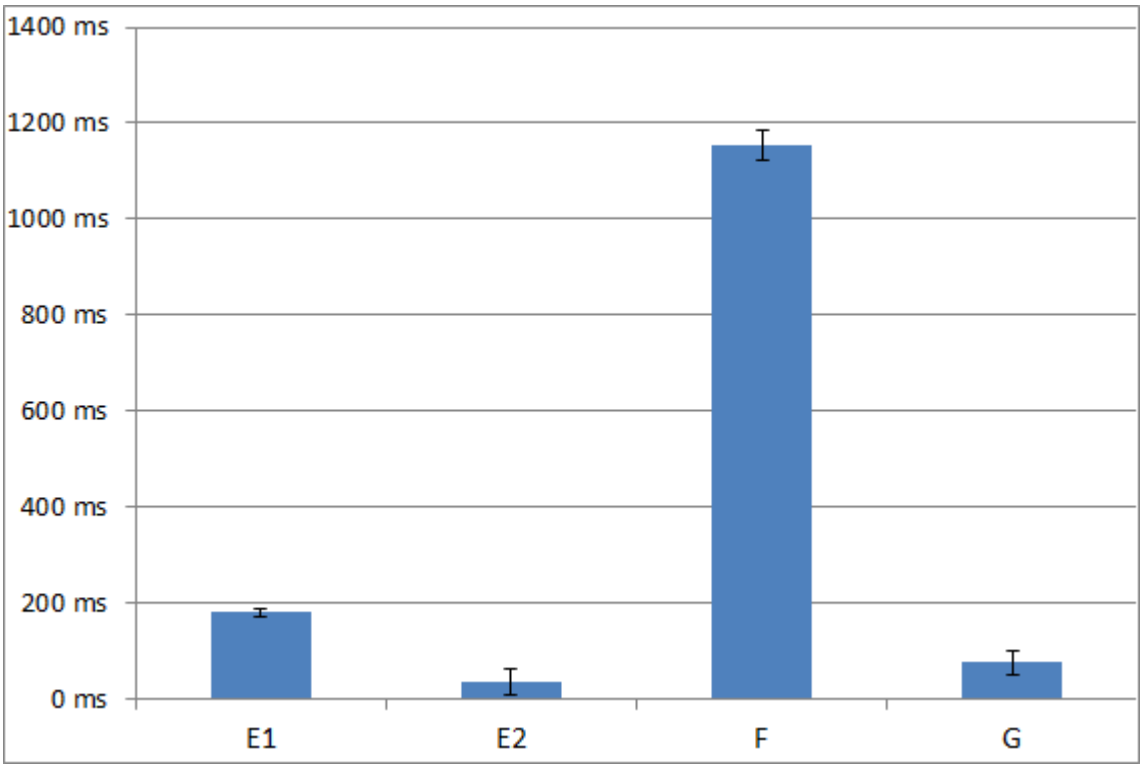

Figure 4.32: Mean and standard deviation for video recording

To put these values into a sequence of 30 seconds and therefore to calculte the throughput in bytes, further assumptions have to be made:

- **Assumption 1:** Launching the application and initializing the camera takes as long as it takes for the image capturing application (step A).
- **Assumption 2:** To record x seconds it takes exactly x seconds (step C).
- **Assumption 3:** The camera records always 30 frames per second.
- **Assumption 4:** The transition rate of the 2D barcodes is equals to the fps rate of the recorded video.

Assumption 1 is surely a lower bound for the time the launching and initialization really takes. The integration of FFmpeg enlarges the application size from about 2MB up to 30MB. Since there are additional objects which need to be initialized in contrast to the image approach application, the launch obviously has to take longer. But this is hard to express in terms of seconds. *Assumption 2* on the other hand is a realistic estimation.

If recording would take a long time – for the tasks like storing on the file system, etc. – then the longer one would record the larger would this overhead be. This is not the case, because even after recording for a couple of minutes, the output file is directly accessible. *Assumption 3* holds for the cases where the camera is not forced to auto focus or auto white balance. Tests with the Samsung Galaxy SII have shown that in these cases the frames-per-second rate decreases down to about 24. This assumption is as well realistic, since it can be assumed that the recording device points during the whole recording phase to the same source, i.e. a screen, with the same light conditions and distance. *Assumption 4* is only relevant to measure the output of this approach. It does not state if such a high transition rate is achievable or not.

After measuring the execution time and calculating the statistical data for the foresaid atomic steps, the next step is to combine them into a sequence of maximum 30 seconds. Including assumption 1 and 2 into this sequence results in the following inequation:

$$\begin{aligned} A + C + x * (E1 + E2 + F + G) &\leq 30s \\ \underbrace{C = x * \frac{1}{30}s}_{\text{Assumptions 1 and 2}} & \\ \Rightarrow A + x * \frac{1}{30}s + x * \underbrace{(E1 + E2 + F + G)}_{a} &\leq 30s \end{aligned} \tag{4.10}$$

Equation (4.10) describes the following workflow: First, the application has to be launched and the camera initialized (A). Since this could not be tested for the video recording approach, the arithmetic average from the image capturing approach was taken to solve this equation. The next step is to record a 3D barcode. The main idea is to record as much frames as there can be processed within 30 seconds. Therefore, C is replaced in equation (4.10) by the assumptions 1 and 2. Once the 3D barcode is recorded, for each frame, i.e. 2D matrix, the steps E1, E2, F and G have to be executed sequentially. These steps are summarized as b. Based on this description, on equation (4.10) and on the arithmetic averages from Figure 4.32 for the atomic steps, the following inequation can be set up:

$$\begin{aligned} A + x * \tfrac{1}{30}s + x * b &\leq 30s \\ A &= 1.0067s \\ b &= 1.4425s \end{aligned} \tag{4.11}$$

The solution for x in the maximization problem in (4.11) is 19. This means that 19 barcodes can be processed in at most 30 seconds with the video recording approach. 19 barcodes with each 1024 modules and 3 bit per module means that the throughput of this approach is 7296 bytes (or 7.125 kB).
Based on the assumptions and on Figure 4.32, the atomic step with the longest duration in this approach is the recognizer module (F). An improvement of step F would lead to a higher throughput in this approach as well. Nevertheless, it is obvious that this calculated throughput is clearly smaller than the throughput of the image capturing approach.

4.4.4 Chosen Approach

The previous subsections calculated the results shown in Table 4.1 for the three different approaches. Since the highest throughput could be achieved by taking pictures, this approach was selected to continue with in this thesis.

Approach	Throughput (in bytes per 30s)
Preview frames	912
Image capturing	12288
Video recording	7296

Table 4.1: Summarized throughput results for all the three different approaches

Chapter 5

Solution Design and Implementation

The previous chapter explained the principle design choices – e.g. how to obtain data from the camera – and the algorithms for the recognizer and decoder module. What still has not been described is the mode of operation of the overall application, i.e. how these pieces work together. After describing the framework in Section 5.1, Section 5.2 specifies the solution design and implementation. Section 5.3 depicts then the user interface of the application.

5.1 Framework

The framework can be split into two parts, namely into a software and a hardware part. Subsection 5.1.1 specifies the software side of the framework, i.e. the Android API. The hardware side is described in Subsection 5.1.2, that is the Android device which has been used in the whole process of this thesis. Both, the software and the hardware, have major impacts on the design choices as well as on the final solution. It is important to mention here that the final solution has only been tested for this framework. It is not guaranteed that the solution will also work with other devices or on other Android platforms.

5.1.1 Software

According to the description of work in Section 1.2, the prototype shall be compatible with at least one Android device. Therefore, the operating system of the mobile device and the software framework are prescribed for this thesis. The platform (or API level) are determined by the hardware – the Android device – made available by the Communication Systems Group. Hence, the highest possible platform version is Android 2.3.3 (or API level 10[1]). Since the API level 9 offers all functionalities needed by the final solution, it has been set as minimum SDK requirement.

[1] see `http://developer.android.com/guide/appendix/api-levels.html` for the correlation between platform version and API level

To develop applications for Android devices in an integrated environment, Google offers a plugin for the Eclipse IDE called Android Development Tools (ADT)[2]. The ADT requires that the Android SDK[3], a compatible version of Eclipse, and the Java Development Kit[4] are available on the development computer. Once these tools are installed and configured, the development of Android applications is straightforward using the Java programming language. The code can than be tested on an Android Virtual Device (AVD) or directly on a connected physical device, i.e. a mobile phone.

5.1.2 Hardware

The hardware, i.e. the mobile phone, used for development and testing purposes in this thesis is the Samsung I9100 Galaxy SII[5]. As already mentioned, this is the only Android device which has been tested to work with the prototype. Table 5.1 shows the hardware specifications relevant to the the design choices.

Display size	480*800 pixels
Primary camera	8 MP, 3264*2448 pixels
Video resolution	1080*1920 pixels at 30 fps
CPU	Dual-core 1.2 GHz Cortex-A9

Table 5.1: Relevant hardware specification [12]

5.2 Application Design

This section explains how the algorithms described before work in order to receive a 3D barcode and retrieve the original content. Therefore, the application design of the prototype is covered within this section. Figure 5.1 shows the class diagram of the prototype. To keep the diagram clear, the attributes and methods of the corresponding classes are hidden. To understand the collaboration of these modules, each class or logical group of classes is described in detail now.

In Android an Activity is an application component that provides a screen users can interact with, e.g. take a photo. Since an application can consist of multiple activities, one activity is specified as the main one which is presented to the user when launching the application for the first time [2]. `CameraActivity` is the main – and only – activity of the prototype and specifies the look and feel of the application. The user interface is covered in detail in Section 5.3. Given the case that the mobile device does not own a camera, a corresponding error message is shown and the application automatically shuts down.

[2] `http://developer.android.com/sdk/eclipse-adt.html`
[3] `http://developer.android.com/sdk/installing.html`
[4] `http://java.sun.com/javase/downloads/index.jsp`
[5] `http://www.samsung.com/global/microsite/galaxys2/html/`

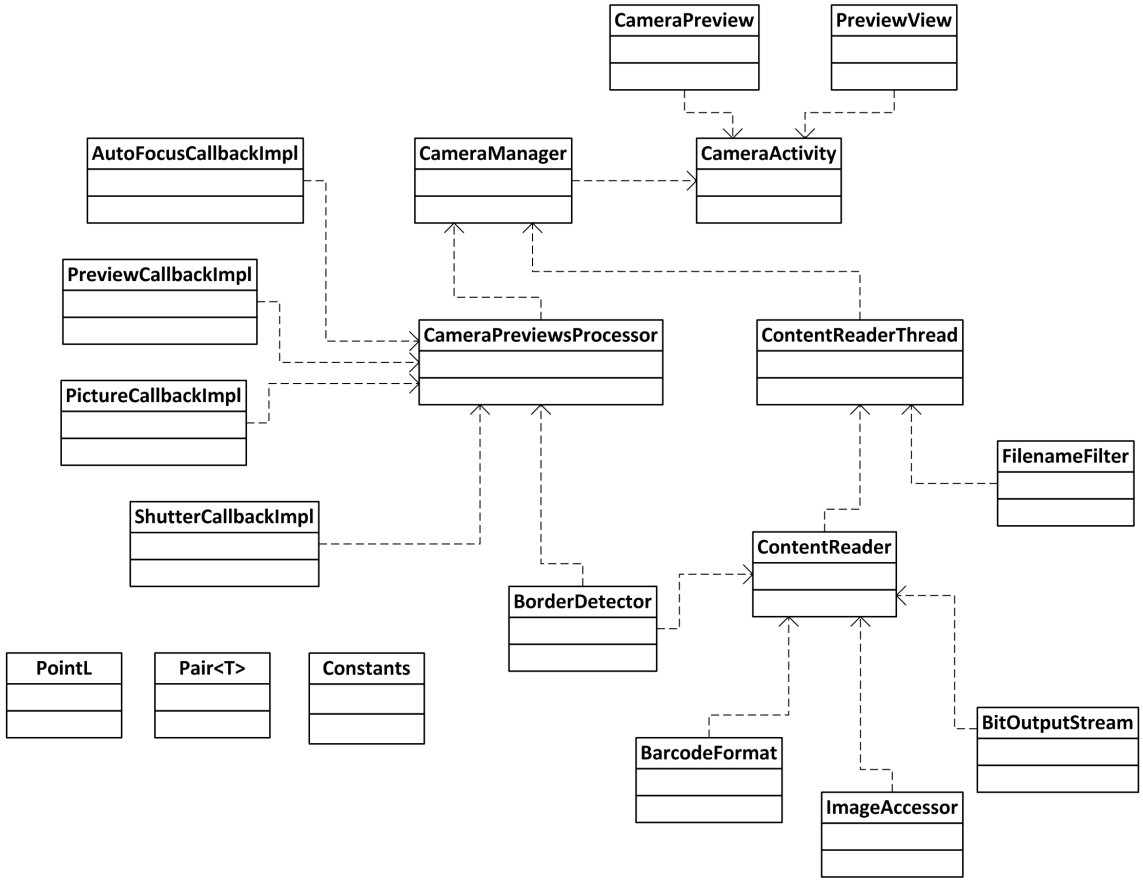

Figure 5.1: Class diagram of the prototype

Once the application is launched, live camera previews are presented to the user which help pointing to a 3D barcode. This can be achieved with a SurfaceView[6] in Android. The `CameraPreview` class is responsible therefore. It handles the proper camera allocation and release. If the camera is not released correctly when the application is destroyed (i.e. quitted) the camera is no longer accessible and the mobile phone has to be rebooted. This class is based on the CameraPreview class in the Android Developer Guide[7].
`PreviewView` is the last user interface class (as well shown in Section 5.3). It supports the user with several information, e.g. where to position the barcode, if decoding is in progress, etc.

The `CameraManager` is the entity which handles the camera access. Taking pictures, or requesting auto focus and previews is exclusively handled by this class. This avoids improper access or release of the camera. It is also the tie between the controller and view classes (thinking of the model-view-controller architectural pattern) and the two threads.

Once the afore mentioned classes are initialized, the CameraManager instantiates a new object of the type `CameraPreviewsProcessor`. The starting point of this instance is to request auto focus from the camera (by delegating it to the CameraManager).
To request auto focus adjustment from the camera Android uses callbacks. One has to

[6]http://developer.android.com/reference/android/view/SurfaceView.html
[7]See section *Creating a preview class* on `http://developer.android.com/guide/topics/media/camera.html`

request auto focus and as far as the camera auto focus completes, the corresponding method is called. Therefore, the AutofocusCallback[8] interface has to be implemented. This is done in the `AutofocusCallbackImpl` class. Initially, AutofocusCallbackImpl requests a preview frame.
This is done in the same manner, i.e. via a Callback. Therefore, the PreviewCallback[9] interface is implemented by the `PreviewCallbackImpl` class. If the preview frame does not contain a barcode (or it cannot be recognized) again auto focus and a preview frame are requested. This is repeated until a barcode is found or the tenth unsuccessful attempt (it is supposed that after the time elapsed the user is pointing to a 3D barcode). Thereafter, the CameraPreviewProcessor changes to decoding mode and requests for the last time auto focus – since the user is not supposed to change his position now and hold the distance between the mobile phone and the sender – and starts taking pictures and storing them on the file system until the application is destroyed or paused. The directory in which the pictures are saved is specified in the Constants class.
The Callback approach is also used for pictures. `PictureCallbackImpl` implements the PictureCallback[10] interface and handles arriving pictures. To assure that the user is still seeing where he points the mobile phone on, the preview has to be started again. Due to the burst delay the user interface freezes for almost a second which means that the live preview does not work for this time period. Unfortunately, this cannot be overcome. Once the PictureCallbackImpl receives a picture, he stores it on the file system, notifies the ContentReaderThread that a new picture is ready to be decoded and takes immediately a new one. Since a user can set the rate at which the pictures are taken, the CameraPreviewsProcessor has to wait if the corresponding time has not elapsed.
In order to avoid the shutter sound of the camera, the `ShutterCallbackImpl` class has been implemented. Unfortunately, this has no effect on the shutter sound which is still played. Since this malfunction could be fixed in future Android releases, it is kept in the prototype.

As mentioned, the prototype works with two threads in parallel. Based on the dual core processor of the Samsung Galaxy SII, real parallelism can be attained. While the main thread is responsible for the user interface and taking pictures, the decoding is done on a separate thread, i.e. `ContentReaderThread`. This second thread looks up the directory where the first one saves the pictures. A `FilenameFilter` assures that only JPEG files are taken into consideration. If the ContentReaderThread cannot find any picture in the given directory, it waits until it gets notified by the CameraPreviewsProcessor that a picture has been stored. Otherwise, it starts with the decoding.
The `ContentReader` is the main entity responsible for the decoding of a barcode. Using an instance of the `BorderDetector` – which corresponds to the recognizer module described in Section 4.2 – the given picture is searched for a barcode. Given the case that the analysed picture contains a barcode and it is recognized, the BorderDetector returns eight points, two for each side of the barcode. With the help of the `ImageAccessor` the four important corner points for the ContentReader – which corresponds to the decoder

[8] see `http://developer.android.com/reference/android/hardware/Camera.AutoFocusCallback.html`
[9] see `http://developer.android.com/reference/android/hardware/Camera.PreviewCallback.html`
[10] see `http://developer.android.com/reference/android/hardware/Camera.PictureCallback.html`

module described in Section 4.3 – can be calculated. After reading the offset information, the ContentReader processes the modules of the barcode. The `BarcodeFormat` class helps mapping the colors to bit sequences, e.g. black to '000'. The resulting three bits are then written to an output file using the `BitOutputStream`[11] class. Since only bytes can be written to a file, this class buffers the bits and writes them to the file as soon as eight bits are available. If a barcode is not recognized in a picture or after it has been decoded, the ContentReader deletes the corresponding file from the file system in order to minimize the storage usage.
Given that the usage of this application is not limited to file transfer only but could also be used for streaming, no file transfer protocol has been implemented. Transferring files depends on some kind of a protocol in order to transmit the file name for example. To make the prototype – and therefore the whole workflow – work with files, the actual version supports only JPEG files to be transferred.

Furthermore, three utility classes are part of the prototype. First, the `Constants` class which contains among others the directory for the pictures and the output file. The second utility class is `PointL`. This class is used to store the vanishing points of the ContentReader in order to avoid integer overflow which happens often with little distorted barcodes (and the resulting huge coordinates of the vanishing points). And the last utility class is the generic class `Pair<T>`. It is used to store two objects of the same type, basically the two points on each side of a barcode in order to keep the sides apart.

5.3 User Interface

The user interface of the prototype is quite simple. As mentioned in Section 5.2, there are three classes responsible for the user interface. The CameraPreview presents live preview frames to the user. These frames are the first layer of the user interface. The second layer is given by the PreviewView class, which is added on top of the first one. The latter shows to the user where he has to position the barcode in order to capture decodable barcodes. Figure 5.2 illustrates the user interface when starting the application. The red borders and the text message in the bottom show that decoding is not in progress, therefore previews are searched for a barcode. The area around the red borders is at a given degree transparent. The two dashed lines are intended to quarter the area in order to allow the user to position each corner into the corresponding quarter. This allows to meet the requirement of the recognizer module. As it can be seen in Figure 5.2, a part of the user interface on the right side is black. This is due to the different ratios of images, previews and the screen resolution of the Samsung Galaxy SII. In order to see on the previews what really will be captured in an image, the previews and images must have the same ratios. According to the image resolution of 3264*2448 – which corresponds to a 4:3 ratio – a suitable ratio has to be chosen for the previews. This can be achieved by using the 640*480 resolution for the preview frames. This results in 160 pixels unused space on the mobile phone display, which has a resolution of 800*480 pixels. The built

[11] This class has been implemented by Owen Astrachan and was found on `http://www.cs.duke.edu/csed/poop/huff/spring05/code/BitOutputStream.java`

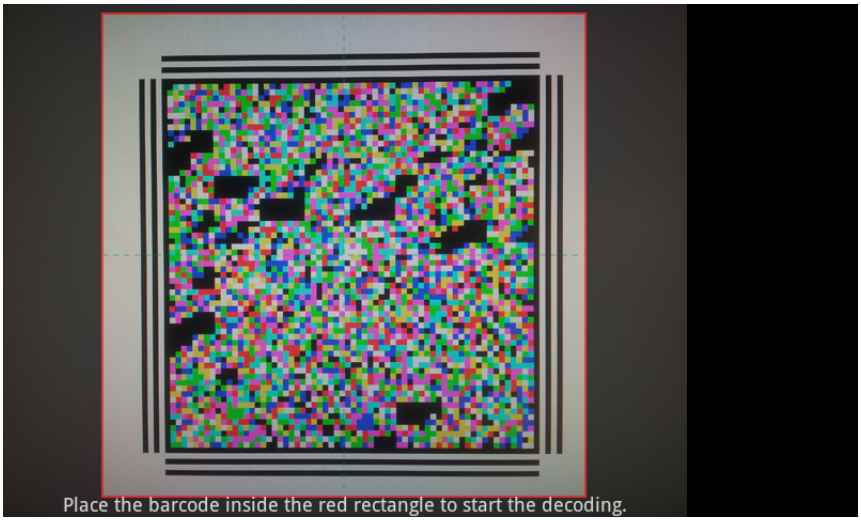

Figure 5.2: User interface appearance when starting the application

in camera application of Samsung uses a similar approach by positioning buttons on this extra space.

Once the application changes to decoding state, feedback is given to the user as illustrated in Figure 5.3. The differences are the text message on the bottom and the border color which changes from red to green.

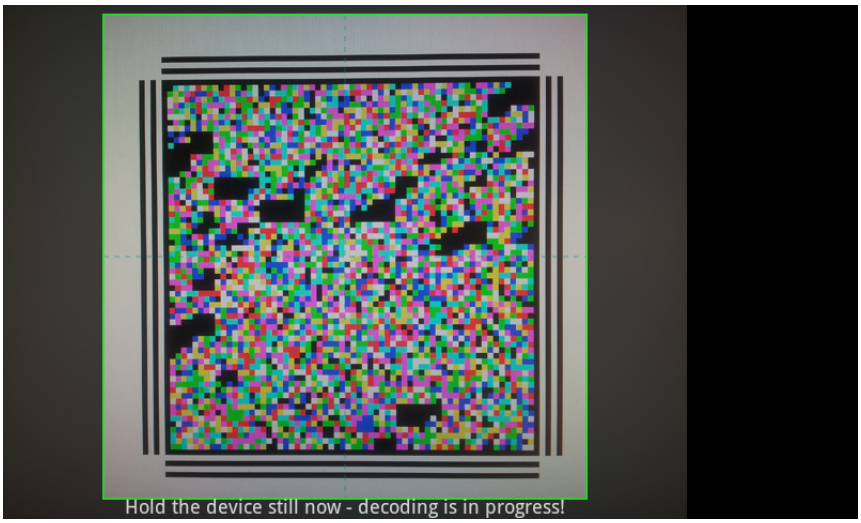

Figure 5.3: User interface appearance when decoding is in progress

The last nameable functionality of the user interface is the input dialog as shown in Figure 5.4. It is accessible via the menu button on the mobile phone, than clicking on the `Settings` button. The user can enter in the input dialog the number of frames which should be captured per second (according to the transition rate of the encoded 3D barcode). Since the burst delay of the camera last almost one second, this value has to be less than one. Entering for example the value 0.5 makes the application taking and decoding a picture every 2 seconds.

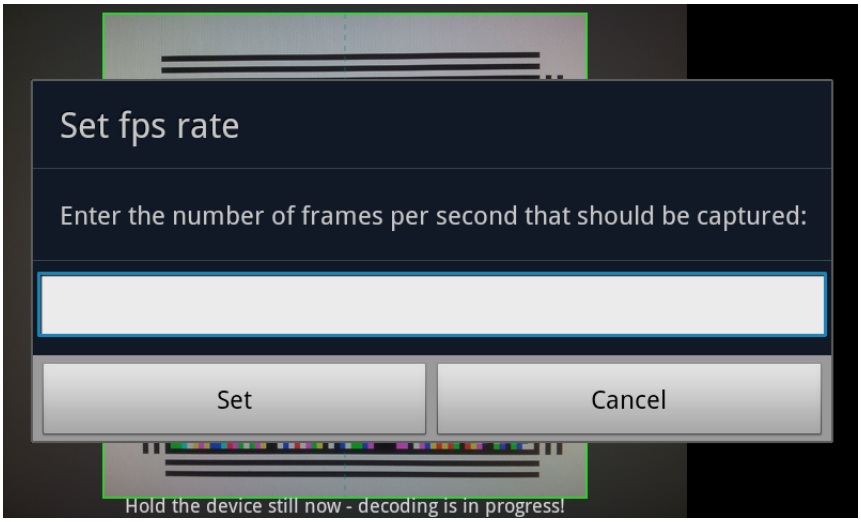

Figure 5.4: Input dialog for the fps rate

Chapter 6

Evaluation

After having explained the design choices and the solution design and implementation, this chapter addresses the evaluation of the implemented algorithms. The goal is to show the boundaries of the final solution. In contrast to the tests in Section 4.4 the main criterion here is the reliability instead of throughput. Section 6.1 describes the test environment in which the evaluation tests were conducted to allow reproducing the tests. Section 6.2 shows the reliability of the algorithms in an ideal case and derives the maximum amount of modules in a barcode. Section 6.3 then discusses the test results.

6.1 Test Environment

Besides of depicting the test environment this section also explains the test series in detail by showing the different dimensions tested and how they were changed to conduct a given test series. The goal is to show where the limits of the actual implementation are concerning each dimension separately.

In order to test each interfering dimension separately, there has to be assured that only the concerning dimension is changing from one test picture to the other. To avoid interference factors like "shaky" hands the mobile device has to be installed statically. Figure 6.1 shows a picture of the laboratory. To display the barcodes which have to be captured by the mobile phone, a Samsung SyncMaster 223BW screen has been used. The receiver on the other hand is a Samsung Galaxy SII mobile phone. The distance between the sender and receiver for all the test (except of the *distance to the screen* test) is 50cm.

There are several dimensions which affect the recognition and decoding rate. To map this interferences to numbers – e.g. how many degrees can the mobile phone be rotated so that the barcode gets recognized and decoded correctly – each of these dimensions has to be tested separately. This means that all the other dimensions stay unchanged. In order to avoid outliers, for each dimension three different pictures are captured. The following list of dimensions might not be complete, but surely contains the most frequently appearing interferences observed:

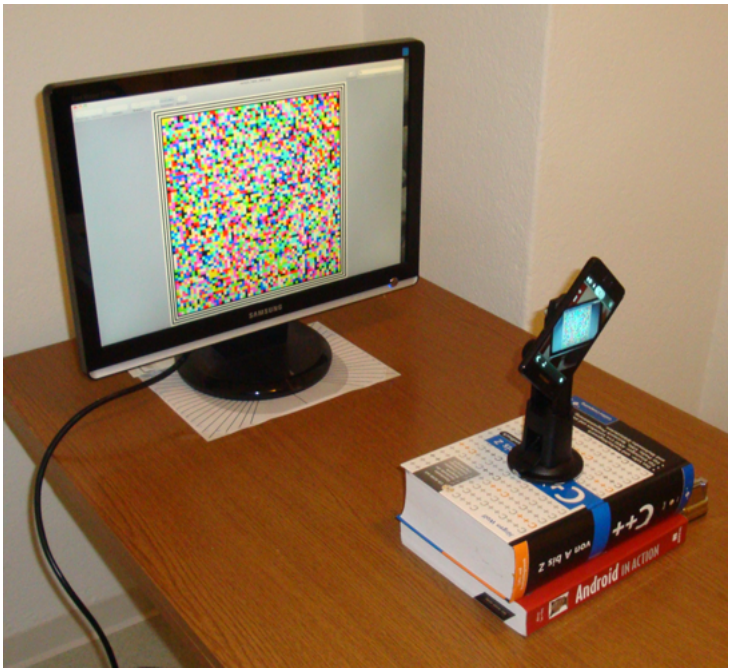

Figure 6.1: Test Environment

- **Rotation:** The actual implementation theoretically allows rotations up to 45 degrees (not included). To evaluate how high the rotation is allowed to be in practice in order to achieve reliable results, the following rotation values are tested: ± 10°, 15°, 20°, 30°, and 40°[1]. To conduct this test series the mobile phone is rotated on the fixture according to the specified degrees above. Figure 6.2 shows what is meant by rotation.

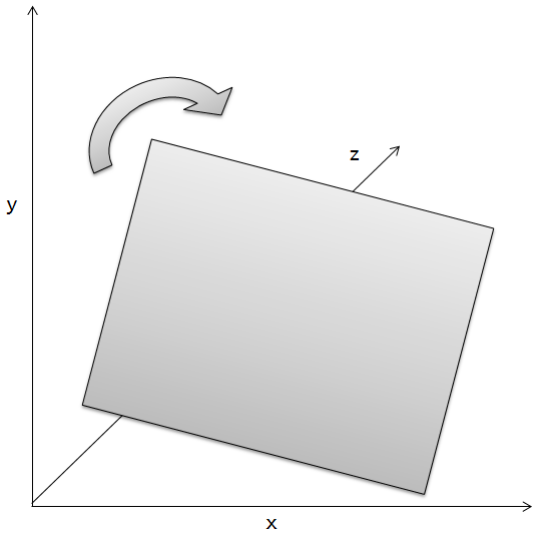

Figure 6.2: Rotation

- **Exposure:** The current mobile phones have cameras with a good exposure correc-

[1]where '-' means counter-clockwise and '+' clockwise rotation

tion[2]. To show how the recognizer and decoder modules behave in very dark or very bright environments, the built in exposure is set manually in the interval ± 0.5, 1.0, 1.5, and 2.0, where -2.0 means the darkest possible exposure and +2.0 the brightest exposure respectively.

- **Distortion:** Distortion can be separated into vertical and horizontal distortion. Vertical distortion appears when a user rotates the mobile phone on the x-axis (thinking of a three-dimensional space) as shown in Figure 6.3. Horizontal distortion corresponds to rotation on the y-axis and is illustrated in Figure 6.4.

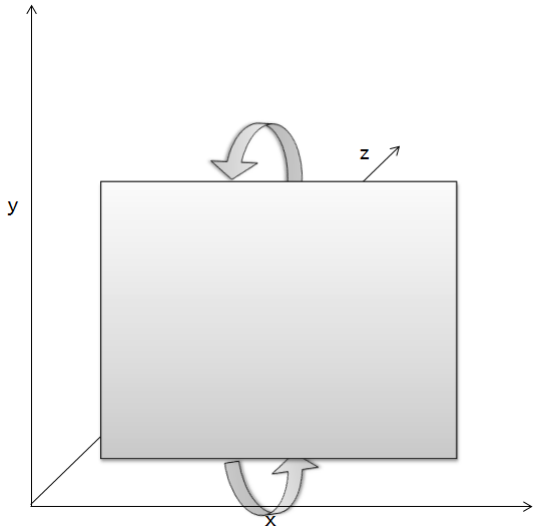

Figure 6.3: Vertical distortion

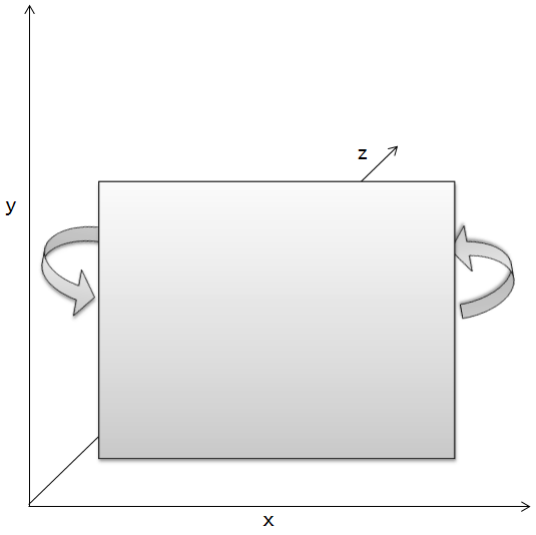

Figure 6.4: Horizontal distortion

In order to evaluate the highest possible horizontal distortion, the following rotation values are tested: ± 10°, 15°, 20°, and 30°. Instead of rotating the mobile phone, horizontal distortion can be simulated by rotating the screen and holding the mobile phone fix.
To test vertical distortion either the mobile phone or the screen have to be rotated on the x-axis. Unfortunately, the fixture shown in Figure 6.1 does not allow such a rotation in the sense that the screen is still captured by the camera. Nor is the screen rotatable in both directions (i.e. up and down). Conducting these tests would require a professional test environment. Therefore, the tests for the vertical distortion are not conducted with the fixture but the mobile phone is hold with the hands and rotated according to the test scenario. By using a cord it is assured that the distance to the screen stays always the same. The following rotation values are tested for vertical distortion: ± 10°, 15°, and 20°.

- **Light reflection:** This is one of the hardest problems to solve in this list of dimensions. Light reflection occurs mostly with glare-type-displays, especially when a light source is pointing to the screen and gets reflected to the camera lens. To evaluate this impact, the first scenario takes pictures in a dark environment with no other light source than the screen. Second, a room light on the floor is switched

[2]which means a brightness correction

on before capturing another series of pictures. As a third scenario a table lamp is switched on so it points to the screen and its light gets reflected towards the lens.

- **Auto focus:** As observed, auto focus has a big impact on the captured picture. When auto focus does not work correctly, the modules and the borders are too blurry, which affects the recognizer as well as the decoder module negatively. Unfortunately, one can not set at which degree the build-in auto focus mechanism should work. But it can be influenced, by setting the auto focus manually to a scene behind the screen or to a scene close to the camera. One test scenario is with auto focus, a second one is with the focus pointing to a scene behind the screen, and the third scenario is with the focus pointing to a scene close to the camera (closer than the screen). By analysing the output it can be shown what influences in terms of reliability a malfunctioning auto focus has.

- **Moving the device:** For this test scenario the mobile device is uninstalled from its static fixture and hold with the hand by a user. The first scenario is to hold the mobile device as calm as possible. The analyses of this scenario shows if the application can be used in a real context. In a second scenario then three pictures are captured by shaking the mobile device a bit, which can be called a stress test.

- **Distance to the screen:** The distance from the mobile device to the screen is the last dimension in this list. The user interface limits the user how close he can get to the screen. The question on the other hand is how big the distance to the screen can be that decodable barcodes are still captured. This test scenario analyses for the distances 50cm, 55cm, 60cm, 65cm, and 70cm the three pictures taken and shows the maximum distance. This is done by moving the mobile device fixture to the distance in the interval above.

6.2 Ideal Case

An ideal case in terms of the dimensions described afore is to have no rotation, zero exposure, no distortion, no light reflection (or no other light source than the screen), appropriate focus, a fixed mobile device, and a distance to the screen which allows to capture the barcode in that way that it fills up the whole picture (corresponds to a distance of 30cm with the given test hardware). Having eliminated all interfering effects in this way one can test how much the maximum amount of modules in a barcode can be. According to Scenario 1 in Appendix E the largest barcode which allows a reliable decoding has 64 modules in the width (height respectively). Using barcodes with 128*128 modules results in lots of failures due to a fish-eye effect of the camera. Therefore, the actual decoder algorithm does not succeed in determining the position of each module. This effect can also be observed with 64*64 barcodes, but it is not a problem since the modules are four times as large.
Since 128*128 barcodes are not reliable even in an ideal case, they are neglected for the further tests and the prototype as well. Instead, 64*64 barcodes are used.

6.3 Discussion Of The Test Results

This discussion is based on the test cases described in Section 6.1 and the results shown in Appendix E. Each test dimension is discussed sequentially, starting with rotation. The tests have shown that for the rotation degrees ± 10°, 15°, 20°, and 30° the decoder algorithm is very reliable. The recognizer algorithm on the other hand could be more accurate sometimes and therefore decrease the number of faults. One outlier is given in the case of + 30° with 319 faults. The problem lies on the picture quality there, i.e. inaccurate focus and white balancing. In opposition to the theoretical value of ± 45° of rotation, the recognizer algorithm fails to recognize ± 40° rotated barcodes. An interesting fact is that the decoder algorithm can successfully process these barcodes once the corner points are found.

The next test dimension is exposure. Obviously, both modules – the recognizer as well as the decoder – succeed in processing the overexposured and underexposured barcodes.

Horizontal distortion is the next dimension which has been tested. An interesting fact is that increasing the rotation degrees increases also the number of failures. This can be explained by the decreasing module size. The higher the rotation is, the smaller get the modules on the side which is farer away. This increases the chance to cross the modules borders in turn. Nevertheless, ± 10° and 15° degrees of horizontal distortion are well managed by the recognizer and the decoder algorithm.

The results of the vertical distortion tests are not absolutely reliable, since these pictures are not taken while the mobile phone has been fixed. Therefore, the indicated rotation degrees are approximate values. Compared to horizontal distortion, the same rule is here valid as well: increasing the rotation degrees increases also the number of failures. Assuming that the recognizer algorithm finds the corner points in each picture, the decoder succeeds in retrieving the content with a failure rate of maximum 2%.
Horizontal and vertical distortion have also another problem source in common. When the camera is not pointing straight to the screen, some areas get darker. LCD screens indicate therefore specific angles as maximum values from which the content is still visible. In the case of distortion the angle does not change, bot one side is closer to the camera lens than the other. Using only one black and white reference point for these different areas is likely to fail.

The tests of light reflection does not reflect the results observed on another screen, i.e. the screen of an Apple MacBook Pro. The Samsung test screen is less glossy and therefore less fault-prone. This is the reason why the test returns good results for the scenarios with one and two different light sources.

The movement test, especially the scenario with the calm held device, represents a realistic case and returns good results. This is how a user would try to transfer data using this prototype. The “shaky“ scenario is not realistic any more but can be called a stress test instead. Since the pictures are very blurry, it is even hard to set the corner points manually. Therefore, the bad results for this scenario are not astonishing.

The last dimension tested is the distance to the screen. The higher the distance gets, the smaller become the modules and the decoder algorithm crosses the modules borders. This

ends up in more failures. Nevertheless, the distances of 50cm and 55cm produce errors in a reasonable scale.

Summarizing the test results, the following specify the limits of the prototype:

- Rotation is supported up to ± 30°.
- Over and underexposured pictures are not a problem.
- Horizontal and vertical distortion are supported up to ± 15°.
- Light reflections are no problem when the sender is not too glossy.
- The mobile phone has not to be fixed in order to transfer data. It can be held with the hands but does not allow shaking of the device (the same as when taking pictures).
- The distance to the screen should be as close as possible. The barcode should fill up the whole picture. Distances greater than 55cm produce too much errors.

Since these interferences do not appear segregated but several at the same time, further tests are needed in order to specify the limits more accurate. In all likelihood these limits will have to be downgraded.

Chapter 7

Summary and Conclusions

Chapter 1 specified two areas of work for this thesis, namely the design and implementation of the recognizer as well as decoder module. This has been formulated as the following key goals:

- Make the workflow work.
- Make it work fast.

Due to the explorative nature of this thesis, solving a problem had impacts on other solved and unsolved problems. Therefore, the following is not meant to describe the chronological sequence.
In order to achieve the first goal, different kinds of problems had to be solved. First, a suitable format of the 2D matrices had to be invented based on existing symbologies. Second, the recognizer module had to be implemented. The main problem with the recognizer is to estimate the brightness of the picture, which is crucial for the color reduction algorithm and therefore, to recognize a barcode within a picture. The third piece of work consist of the decoder module. Determining the positions of the modules and reading the colors accordingly represent the main difficulties here. In addition to the complexity of the overall system, the prototype had to be implemented for the Android operating system. An operating system, which seems to be in the fledgling stages (at least the platform version 2.3.3).
The second goal, i.e. the question what the maximum achievable throughput is, required evaluations of the three different approaches (preview frames, image capturing, and video recording). The findings here had as well an impact on the first goal and the implementation of the overall system.

It is safe to say that the first goal has been achieved on the condition that a reliability of 100% could not be accomplished. Adversarial conditions like distortion, rotation, and light reflectance among others appear mostly all at the same time and have an impact on the decoded content. This is not fatal when using the prototype for streams. File transfer on the other hand is nearly impossible. One inaccurately decoded module has for example an impact on the whole image when transferring JPEG files.

The maximum throughput achieved by this prototype is 12'288 bytes per 30 seconds, which means about 430 bytes/s. Assuming that the recognition and decoding is executed in parallel, the prototype can at most double this throughput, which would result in 860 bytes/s. Compared to speech (telephone quality) which requires a data rate of 64 kbit/s, this prototype could not be used to transfer some voice for example. Transferring a file of 1MB would take more than 20 minutes in the best case. Therefore, it is doubtful that 3D barcodes as implemented in this thesis will establish itself.

A glimmer of hope concerning the throughput is that Android 4.0 is putting an emphasis on taking images faster[1]. Since the bottleneck of the prototype is the capturing of images, this could help increase the throughput.
Another approach with the same goal is a compensation of the fish-eye effect. As mentioned, 128*128 barcodes cannot be used since this effect could not be overcome. Future versions of this prototype can implement algorithms to compensate the fish-eye effect. A larger dimension of the barcode might increase the throughput considerably.
In order to evaluate the thresholds used for the color distinction, further tests on different devices on the sender as well as on the receiver side are inevitable. The actual prototype uses the thresholds evaluated only on one display and digital camera.
And finally, to address the reliability problem of the actual prototype, future versions could apply error detection and correction techniques.

[1] see `http://developer.android.com/sdk/android-4.0-highlights.html`

Bibliography

[1] A. Longacre, Jr. et al.: *Two Dimensional Data Encoding Structure and Symbology for use with Optical Readers*, US Patent 5591956, January 1997.

[2] Android Developers: *Activities*, `http://developer.android.com/guide/topics/fundamentals/activities.html`, April 2012.

[3] Association for Automatic Identification and Mobility: *MaxiCode*, `https://www.aimglobal.org/estore/ProductDetails.aspx?productID=25`, October 2011.

[4] Association for Automatic Identification and Mobility: *Ultra Code Overview*, `http://www.aimglobal.org/standards/symbinfo/ultracode_overview.asp`, October 2011.

[5] Barcode Island: Code 128 Symbology, `http://www.barcodeisland.com/code128.phtml`, October 2011.

[6] Cambridge In Colour: Digital Camera Sensors, `http://www.cambridgeincolour.com/tutorials/camera-sensors.htm`, December 2011.

[7] Denso Wave: QR Code - Official website, `http://www.denso-wave.com/qrcode/index-e.html`, October 2011.

[8] Denso Wave: QR Code Area, `http://www.denso-wave.com/qrcode/qrgene4-e.html`, October 2011.

[9] Denso Wave: Version and Maximum capacity table, `http://www.denso-wave.com/qrcode/vertable4-e.html`, October 2011.

[10] Global Standards 1: Allgemeine GS1 Spezifikationen Version 11, `http://www.gs1.ch/de/leistungsbereiche/identification-communication/standardisation/GeneralSpecification/pdf/GenSpec_v11_DE.pdf`, October 2011.

[11] Global Standards 1: GS1 Bar Code Verification for Linear Symbols, `http://www.gs1.org/docs/barcodes/GS1_Bar_Code_Verification.pdf`, May 2009.

[12] GSMArena: Samsung I9100 Galaxy S II - Full phone specifications, `http://www.gsmarena.com/samsung_i9100_galaxy_s_ii-3621.php`, April 2012.

[13] H. Walravens: *ISBN International Standard Book Number*, Berlin, Germany, 2010.

[14] Personal Computer Organisation: Color Ultra Code, http://www.pco-barcode.de/staticsite/staticsite.php?menuid=344&topmenu=296, October 2011.

[15] Quack-ware.com: Include FFMpeg in Android and grab a frame from a 3gp video, http://www.quack-ware.com/2011/include-ffmpeg-in-android-and-grab-a-frame-from-a-3gp-video/, October 2011.

[16] T. Gibara: *Moseycode Symbology*, http://www.tomgibara.com/android/moseycode/symbology, October 2011.

[17] T. Langlotz, O. Bimber: *Unsynchronized 4D Barcodes*, International Symposium on Visual Computing, pp. 363-374, 2007.

[18] TALtech: Maxicode, http://www.taltech.com/support/entry/maxicode, October 2011.

[19] TEC-IT: Barcodes, http://www.tec-it.com/Download/PDF/Barcode_Reference_DE.pdf, October 2011.

[20] TEC-IT: Overview: 2D Barcode Symbologies, http://www.tec-it.com/en/support/knowbase/symbologies/barcode-overview/2d-barcodes/Default.aspx, October 2011.

[21] TEC-IT: Overview: Linear Bar Code Symbologies, http://www.tec-it.com/en/support/knowbase/symbologies/barcode-overview/linear/Default.aspx, October 2011.

[22] Tom's Hardware: Der Aufbau der Bildpunkte eines TFTs, http://www.tomshardware.de/TFT-LCD-Funktionsweise,testberichte-49-7.html, January 2012.

[23] Y. Wang, et al.: *Maxicode data extraction using spatial domain features*, US Patent 5637849, June 1997.

Abbreviations

EAN	European Article Number
QR Code	Quick Response Code
UPC	Universal Product Code

Glossary

EAN/UPC Symbology A family of barcodes including EAN-8, EAN-13, UPC-A, and UPC-E barcodes. Although UPC-E Bar codes do not have a separate symbology identifier, they act like a separate symbology through the scanning application software.

Symbology A defined method of representing numeric or alphabetic characters in a bar code; a type of bar code.

List of Figures

List of Tables

Appendix A

Test Results For The Atomic Steps - Preview Frames

(A) Launch the application and initialize the camera

1	2	3	4	5	6	7	8	9	10
1180ms	1038ms	1228ms	1010ms	951ms	1175ms	1154ms	1183ms	1060ms	1068ms

Arithmetic average: 1104.7 ms
Standard deviation: 91.14 ms

(C1) Request the first preview frame

1	2	3	4	5	6	7	8	9	10
232ms	280ms	220ms	207ms	210ms	200ms	394ms	450ms	370ms	398ms

Arithmetic average: 296.1 ms
Standard deviation: 96.53 ms

(C2) Request another preview frame

1	2	3	4	5	6	7	8	9	10
79ms	45ms	60ms	67ms	60ms	36ms	69ms	68ms	62ms	56ms

Arithmetic average: 60.2 ms
Standard deviation: 12.36 ms

(E) Create bitmap object

1	2	3	4	5	6	7	8	9	10
195ms	85ms	112ms	129ms	104ms	106ms	113ms	112ms	121ms	124ms

Arithmetic average: 120.1 ms
Standard deviation: 29.02 ms

(F) Detect borders

1	2	3	4	5	6	7	8	9	10
612ms	555ms	515ms	562ms	570ms	492ms	553ms	565ms	509ms	508ms

Arithmetic average: 544.1 ms
Standard deviation: 37.02 ms

(G) Decode content

1	2	3	4	5	6	7	8	9	10
22ms	29ms	6ms	10ms	6ms	6ms	7ms	6ms	6ms	5ms

Arithmetic average: 10.3 ms
Standard deviation: 8.23 ms

Appendix B

Test Results For The Atomic Steps - Image Capturing

(A) Launch the application and initialize the camera

1	2	3	4	5	6	7	8	9	10
980ms	995ms	979ms	1003ms	1019ms	1129ms	998ms	1034ms	960ms	970ms

Arithmetic average: 1006.7 ms
Standard deviation: 48.41 ms

(B) Request auto focus and auto white balance

1	2	3	4	5	6	7	8	9	10
920ms	910ms	904ms	950ms	909ms	910ms	910ms	858ms	875ms	913mss

Arithmetic average: 905.9 ms
Standard deviation: 24.74 ms

(C) Capture an image

1	2	3	4	5	6	7	8	9	10
1069ms	987ms	897ms	982ms	1112ms	979ms	1012ms	999ms	1020ms	1080ms

Arithmetic average: 1013.7 ms
Standard deviation: 61.39 ms

(D) Save image to the file system

1	2	3	4	5	6	7	8	9	10
129ms	130ms	222ms	126ms	97ms	162ms	104ms	164ms	126ms	557ms

Arithmetic average: 181.7 ms
Standard deviation: 136.64 ms

(E) Create bitmap object

1	2	3	4	5	6	7	8	9	10
790ms	714ms	716ms	705ms	731ms	732ms	704ms	714ms	746ms	704ms

Arithmetic average: 725.6 ms
Standard deviation: 26.57 ms

(F) Detect borders

1	2	3	4	5	6	7	8	9	10
1149ms	933ms	937ms	950ms	965ms	1049ms	1001ms	1032ms	985ms	1016ms

Arithmetic average: 1001.7 ms
Standard deviation: 65.2 ms

(G) Decode content

1	2	3	4	5	6	7	8	9	10
382ms	236ms	235ms	220ms	220ms	219ms	251ms	220ms	217ms	249ms

Arithmetic average: 244.9 ms
Standard deviation: 49.82 ms

Appendix C

Test Results For The Atomic Steps - Video Recording

(E1) Grab a frame

1	2	3	4	5	6	7	8	9	10
200ms	176ms	175ms	172ms	177ms	177ms	178ms	173ms	186ms	182ms

Arithmetic average: 179.6 ms
Standard deviation: 8.26 ms

(E2) Create bitmap object

1	2	3	4	5	6	7	8	9	10
40ms	40ms	43ms	22ms	22ms	44ms	23ms	46ms	24ms	47ms

Arithmetic average: 35.1 ms
Standard deviation: 10.87 ms

(F) Detect borders

1	2	3	4	5	6	7	8	9	10
1238ms	1136ms	1136ms	1148ms	1135ms	1167ms	1139ms	1158ms	1135ms	1142ms

Arithmetic average: 1153.4 ms
Standard deviation: 31.64 ms

(G) Decode content

1	2	3	4	5	6	7	8	9	10
142ms	64ms	62ms	62ms	61ms	64ms	61ms	74ms	77ms	77ms

Arithmetic average: 74.4 ms
Standard deviation: 24.65 ms

Appendix D

Test Results Different Thresholds And Reading Techniques

Scenario 1:

Reading matrix: 5x5 pixels

For the black/white reference points calculate average for each color channel separately

Thresholds: R=G=160

- rg (black or blue): B= 150
- rG (green or cyan): B=150
- Rg (red or magenta): B=170
- RG (yellow or white): B=200

Test_000.jpg	1: Original Average: R: 125, G: 121, B: 54 1: Mapped Values: R: 255, G: 255, B: 213 1: yellow decoded as white 2: Original Average: R: 131, G: 127, B: 53 2: Mapped Values: R: 255, G: 255, B: 209 2: yellow decoded as white 3: Original Average: R: 125, G: 124, B: 52 3: Mapped Values: R: 255, G: 255, B: 204 3: yellow decoded as white 4: Original Average: R: 121, G: 124, B: 53 4: Mapped Values: R: 255, G: 255, B: 209 4: yellow decoded as white 5: Original Average: R: 110, G: 126, B: 52 5: Mapped Values: R: 255, G: 255, B: 204 5: yellow decoded as white 6: Original Average: R: 126, G: 127, B: 62 6: Mapped Values: R: 255, G: 255, B: 246 6: yellow decoded as white 7: Original Average: R: 132, G: 131, B: 60 7: Mapped Values: R: 255, G: 255, B: 238 7: yellow decoded as white 8: Original Average: R: 120, G: 124, B: 57 8: Mapped Values: R: 255, G: 255, B: 225 8: yellow decoded as white number of failures: 8
Test_001.jpg	1: Original Average: R: 90, G: 92, B: 76 1: Mapped Values: R: 207, G: 228, B: 191 1: white decoded as yellow 2: Original Average: R: 105, G: 19, B: 67 2: Mapped Values: R: 245, G: 34, B: 167 2: magenta decoded as red 3: Original Average: R: 104, G: 86, B: 78 3: Mapped Values: R: 242, G: 212, B: 196 3: white decoded as yellow 4: Original Average: R: 116, G: 13, B: 67 4: Mapped Values: R: 255, G: 18, B: 167 4: magenta decoded as red number of failures: 4
Test_002.jpg	1: Original Average: R: 160, G: 156, B: 75 1: Mapped Values: R: 255, G: 255, B: 203 1: yellow decoded as white 2: Original Average: R: 156, G: 158, B: 79 2: Mapped Values: R: 255, G: 255, B: 214 2: yellow decoded as white number of failures: 2
Test_003.jpg	number of failures: 0
Test_004.jpg	number of failures: 0
Test_005.jpg	number of failures: 0
Test_006.jpg	number of failures: 0
Test_007.jpg	number of failures: 0
Test_008.jpg	number of failures: 0

Scenario 2:

Reading matrix: 5x5 pixels

For the black/white reference points calculate average for each color channel separately

Thresholds: R=G=160

- rg (black or blue): B= 150
- rG (green or cyan): B=150
- Rg (red or magenta): B=170
- RG (yellow or white): B=214

Test_000.jpg	1: Original Average: R: 126, G: 127, B: 62 1: Mapped Values: R: 255, G: 255, B: 246 1: yellow decoded as white 2: Original Average: R: 132, G: 131, B: 60 2: Mapped Values: R: 255, G: 255, B: 238 2: yellow decoded as white 3: Original Average: R: 120, G: 124, B: 57 3: Mapped Values: R: 255, G: 255, B: 225 3: yellow decoded as white number of failures: 3
Test_001.jpg	1: Original Average: R: 90, G: 92, B: 76 1: Mapped Values: R: 207, G: 228, B: 191 1: white decoded as yellow 2: Original Average: R: 105, G: 19, B: 67 2: Mapped Values: R: 245, G: 34, B: 167 2: magenta decoded as red 3: Original Average: R: 90, G: 97, B: 83 3: Mapped Values: R: 207, G: 241, B: 209 3: white decoded as yellow 4: Original Average: R: 97, G: 104, B: 84 4: Mapped Values: R: 225, G: 255, B: 212 4: white decoded as yellow 5: Original Average: R: 104, G: 86, B: 78 5: Mapped Values: R: 242, G: 212, B: 196 5: white decoded as yellow 6: Original Average: R: 116, G: 13, B: 67 6: Mapped Values: R: 255, G: 18, B: 167 6: magenta decoded as red 7: Original Average: R: 111, G: 95, B: 82 7: Mapped Values: R: 255, G: 236, B: 207 7: white decoded as yellow number of failures: 7
Test_002.jpg	number of failures: 0
Test_003.jpg	number of failures: 0
Test_004.jpg	number of failures: 0
Test_005.jpg	1: Original Average: R: 166, G: 141, B: 125 1: Mapped Values: R: 239, G: 211, B: 205 1: white decoded as yellow number of failures: 1
Test_006.jpg	1: Original Average: R: 173, G: 150, B: 133 1: Mapped Values: R: 236, G: 216, B: 204 1: white decoded as yellow number of failures: 1
Test_007.jpg	number of failures: 0
Test_008.jpg	1: Original Average: R: 192, G: 189, B: 169 1: Mapped Values: R: 222, G: 235, B: 210 1: white decoded as yellow 2: Original Average: R: 207, G: 181, B: 168 2: Mapped Values: R: 245, G: 223, B: 208 2: white decoded as yellow number of failures: 2

Scenario 3:

Reading matrix: 9x3 pixels

For the black/white reference points calculate average for each color channel separately

Thresholds: R=G=160

- rg (black or blue): B= 150
- rG (green or cyan): B=150
- Rg (red or magenta): B=170
- RG (yellow or white): B=200

Test_000.jpg	1: Original Average: R: 109, G: 119, B: 52 1: Mapped Values: R: 255, G: 255, B: 204 1: yellow decoded as white 2: Original Average: R: 112, G: 121, B: 52 2: Mapped Values: R: 255, G: 255, B: 204 2: yellow decoded as white 3: Original Average: R: 116, G: 126, B: 53 3: Mapped Values: R: 255, G: 255, B: 209 3: yellow decoded as white number of failures: 3
Test_001.jpg	1: Original Average: R: 94, G: 94, B: 79 1: Mapped Values: R: 217, G: 233, B: 199 1: white decoded as yellow 2: Original Average: R: 104, G: 19, B: 68 2: Mapped Values: R: 242, G: 34, B: 170 2: magenta decoded as red 3: Original Average: R: 107, G: 86, B: 79 3: Mapped Values: R: 250, G: 212, B: 199 3: white decoded as yellow number of failures: 3
Test_002.jpg	number of failures: 0
Test_003.jpg	number of failures: 0
Test_004.jpg	number of failures: 0
Test_005.jpg	number of failures: 0
Test_006.jpg	number of failures: 0
Test_007.jpg	number of failures: 0
Test_008.jpg	number of failures: 0

Scenario 4:

Reading matrix: 9x3 pixels

For the black/white reference points calculate average = (R+G+B)/3 (do not handle each channel separately)

Thresholds: R=G=160

- rg (black or blue): B= 150
- rG (green or cyan): B=150
- Rg (red or magenta): B=170
- RG (yellow or white): B=200

Test_000.jpg	number of failures: 0
Test_001.jpg	1: Original Average: R: 94, G: 94, B: 79 1: Mapped Values: R: 231, G: 231, B: 192 1: white decoded as yellow 2: Original Average: R: 104, G: 19, B: 68 2: Mapped Values: R: 255, G: 36, B: 163 2: magenta decoded as red 3: Original Average: R: 103, G: 18, B: 70 3: Mapped Values: R: 255, G: 33, B: 169 3: magenta decoded as red 4: Original Average: R: 107, G: 86, B: 79 4: Mapped Values: R: 255, G: 210, B: 192 4: white decoded as yellow 5: Original Average: R: 115, G: 13, B: 69 5: Mapped Values: R: 255, G: 20, B: 166 5: magenta decoded as red number of failures: 5
Test_002.jpg	number of failures: 0
Test_003.jpg	number of failures: 0
Test_004.jpg	number of failures: 0
Test_005.jpg	1: Original Average: R: 167, G: 140, B: 123 1: Mapped Values: R: 255, G: 212, B: 181 1: white decoded as yellow 2: Original Average: R: 178, G: 41, B: 116 2: Mapped Values: R: 255, G: 29, B: 168 2: magenta decoded as red number of failures: 2
Test_006.jpg	1: Original Average: R: 177, G: 151, B: 135 1: Mapped Values: R: 255, G: 217, B: 189 1: white decoded as yellow number of failures: 1
Test_007.jpg	1: Original Average: R: 241, G: 124, B: 237 1: Mapped Values: R: 255, G: 162, B: 255 1: magenta decoded as white number of failures: 1
Test_008.jpg	1: Original Average: R: 206, G: 179, B: 166 1: Mapped Values: R: 255, G: 217, B: 196 1: white decoded as yellow number of failures: 1

Scenario 5:

Reading matrix: 9x3 pixels

For the black/white reference points calculate average for each color channel separately

Thresholds: R=G=160

- rg (black or blue): B= 150
- rG (green or cyan): B=150
- Rg (red or magenta): B=170
- RG (yellow or white):
  B= original(min{R, G}) * 0.7

Test_000.jpg	number of failures: 0
Test_001.jpg	1: Original Average: R: 104, G: 19, B: 68 1: Mapped Values: R: 242, G: 34, B: 170 1: magenta decoded as red number of failures: 1
Test_002.jpg	number of failures: 0
Test_003.jpg	number of failures: 0
Test_004.jpg	number of failures: 0
Test_005.jpg	number of failures: 0
Test_006.jpg	number of failures: 0
Test_007.jpg	number of failures: 0
Test_008.jpg	number of failures: 0

Scenario 6:

Reading matrix: 9x3 pixels

For the black/white reference points calculate average for each color channel separately

Thresholds: R=G=160

- rg (black or blue): B= 150
- rG (green or cyan): B=150
- Rg (red or magenta): B=165
- RG (yellow or white):
  B= original(min{R, G}) * 0.7

Test_000.jpg	`number of failures: 0`
Test_001.jpg	`number of failures: 0`
Test_002.jpg	`number of failures: 0`
Test_003.jpg	`number of failures: 0`
Test_004.jpg	`number of failures: 0`
Test_005.jpg	`number of failures: 0`
Test_006.jpg	`number of failures: 0`
Test_007.jpg	`number of failures: 0`
Test_008.jpg	`number of failures: 0`

Scenario 7:

Reading matrix: 9x3 pixels

For the black/white reference points calculate average for each color channel separately

Thresholds: R=G=160

- rg (black or blue): B= 150
- rG (green or cyan): B=150
- Rg (red or magenta): B=165
- RG (yellow or white):
  B= original(min{R, G}) * 0.7

Test_010.jpg	1: Original Average: R: 165, G: 33, B: 98 1: Mapped Values: R: 255, G: 9, B: 163 1: magenta decoded as red 2: Original Average: R: 157, G: 31, B: 95 2: Mapped Values: R: 245, G: 5, B: 157 2: magenta decoded as red 3: Original Average: R: 12, G: 120, B: 82 3: Mapped Values: R: 0, G: 180, B: 131 3: cyan decoded as green number of failures: 3
Test_011.jpg	number of failures: 0
Test_012.jpg	1: Original Average: R: 83, G: 12, B: 49 1: Mapped Values: R: 217, G: 17, B: 143 1: magenta decoded as red 2: Original Average: R: 1, G: 5, B: 42 2: Mapped Values: R: 0, G: 0, B: 121 2: blue decoded as black 3: Original Average: R: 3, G: 71, B: 42 3: Mapped Values: R: 0, G: 188, B: 121 3: cyan decoded as green 4: Original Average: R: 3, G: 68, B: 50 4: Mapped Values: R: 0, G: 179, B: 146 4: cyan decoded as green 5: Original Average: R: 90, G: 12, B: 55 5: Mapped Values: R: 237, G: 17, B: 162 5: magenta decoded as red 6: Original Average: R: 1, G: 5, B: 46 6: Mapped Values: R: 0, G: 0, B: 133 6: blue decoded as black 7: Original Average: R: 1, G: 7, B: 49 7: Mapped Values: R: 0, G: 2, B: 143 7: blue decoded as black 8: Original Average: R: 4, G: 74, B: 49 8: Mapped Values: R: 0, G: 197, B: 143 8: cyan decoded as green 9: Original Average: R: 8, G: 79, B: 49 9: Mapped Values: R: 2, G: 211, B: 143 9: cyan decoded as green 10: Original Average: R: 95, G: 12, B: 55 10: Mapped Values: R: 252, G: 17, B: 162 10: magenta decoded as red 11: Original Average: R: 6, G: 78, B: 49 11: Mapped Values: R: 0, G: 208, B: 143 11: cyan decoded as green 12: Original Average: R: 14, G: 85, B: 48 12: Mapped Values: R: 20, G: 228, B: 140 12: cyan decoded as green 13: Original Average: R: 5, G: 79, B: 49 13: Mapped Values: R: 0, G: 211, B: 143 13: cyan decoded as green 14: Original Average: R: 1, G: 4, B: 50 14: Mapped Values: R: 0, G: 0, B: 146 14: blue decoded as black number of failures: 14

Test_013.jpg	1: Original Average: R: 159, G: 32, B: 91 1: Mapped Values: R: 253, G: 7, B: 151 1: magenta decoded as red number of failures: 1
Test_014.jpg	number of failures: 0
Test_015.jpg	number of failures: 0
Test_016.jpg	1: Original Average: R: 252, G: 139, B: 244 1: Mapped Values: R: 255, G: 161, B: 255 1: magenta decoded as white 2: Original Average: R: 253, G: 139, B: 250 2: Mapped Values: R: 255, G: 161, B: 255 2: magenta decoded as white 3: Original Average: R: 248, G: 139, B: 245 3: Mapped Values: R: 255, G: 161, B: 255 3: magenta decoded as white 4: Original Average: R: 242, G: 139, B: 246 4: Mapped Values: R: 255, G: 161, B: 255 4: magenta decoded as white number of failures: 4
Test_017.jpg	1: Original Average: R: 242, G: 225, B: 159 1: Mapped Values: R: 255, G: 255, B: 158 1: yellow decoded as white 2: Original Average: R: 227, G: 213, B: 150 2: Mapped Values: R: 255, G: 242, B: 144 2: yellow decoded as white 3: Original Average: R: 223, G: 220, B: 162 3: Mapped Values: R: 250, G: 253, B: 163 3: yellow decoded as white 4: Original Average: R: 231, G: 231, B: 166 4: Mapped Values: R: 255, G: 255, B: 170 4: yellow decoded as white number of failures: 4
Test_018.jpg	1: Original Average: R: 234, G: 147, B: 218 1: Mapped Values: R: 255, G: 170, B: 255 1: magenta decoded as white 2: Original Average: R: 243, G: 146, B: 235 2: Mapped Values: R: 255, G: 167, B: 255 2: magenta decoded as white 3: Original Average: R: 244, G: 147, B: 234 3: Mapped Values: R: 255, G: 170, B: 255 3: magenta decoded as white 4: Original Average: R: 248, G: 93, B: 143 4: Mapped Values: R: 255, G: 60, B: 195 4: red decoded as magenta 5: Original Average: R: 251, G: 94, B: 148 5: Mapped Values: R: 255, G: 62, B: 205 5: red decoded as magenta 6: Original Average: R: 236, G: 123, B: 140 6: Mapped Values: R: 255, G: 121, B: 188 6: red decoded as magenta 7: Original Average: R: 12, G: 231, B: 125 7: Mapped Values: R: 0, G: 255, B: 156 7: green decoded as cyan 8: Original Average: R: 16, G: 232, B: 126 8: Mapped Values: R: 0, G: 255, B: 158 8: green decoded as cyan 9: Original Average: R: 252, G: 85, B: 138 9: Mapped Values: R: 255, G: 44, B: 184 9: red decoded as magenta 10: Original Average: R: 16, G: 228, B: 132 10: Mapped Values: R: 0, G: 255, B: 171 10: green decoded as cyan 11: Original Average: R: 248, G: 118, B: 136 11: Mapped Values: R: 255, G: 111, B: 180 11: red decoded as magenta 12: Original Average: R: 240, G: 148, B: 241 12: Mapped Values: R: 255, G: 172, B: 255 12: magenta decoded as white 13: Original Average: R: 13, G: 240, B: 132 13: Mapped Values: R: 0, G: 255, B: 171 13: green decoded as cyan

```
14: Original Average: R: 31, G: 158, B: 207
14: Mapped Values:    R: 0, G: 192, B: 255
14: blue decoded as cyan
15: Original Average: R: 55, G: 236, B: 133
15: Mapped Values:    R: 0, G: 255, B: 173
15: green decoded as cyan
16: Original Average: R: 247, G: 92, B: 154
16: Mapped Values:    R: 255, G: 58, B: 218
16: red decoded as magenta
17: Original Average: R: 253, G: 98, B: 163
17: Mapped Values:    R: 255, G: 70, B: 237
17: red decoded as magenta
18: Original Average: R: 238, G: 149, B: 239
18: Mapped Values:    R: 255, G: 174, B: 255
18: magenta decoded as white
19: Original Average: R: 36, G: 145, B: 214
19: Mapped Values:    R: 0, G: 165, B: 255
19: blue decoded as cyan
20: Original Average: R: 18, G: 221, B: 140
20: Mapped Values:    R: 0, G: 255, B: 188
20: green decoded as cyan
21: Original Average: R: 247, G: 125, B: 142
21: Mapped Values:    R: 255, G: 125, B: 192
21: red decoded as magenta
22: Original Average: R: 13, G: 242, B: 125
22: Mapped Values:    R: 0, G: 255, B: 156
22: green decoded as cyan
23: Original Average: R: 249, G: 90, B: 131
23: Mapped Values:    R: 255, G: 54, B: 169
23: red decoded as magenta
24: Original Average: R: 253, G: 90, B: 135
24: Mapped Values:    R: 255, G: 54, B: 177
24: red decoded as magenta
25: Original Average: R: 239, G: 149, B: 244
25: Mapped Values:    R: 255, G: 174, B: 255
25: magenta decoded as white
26: Original Average: R: 242, G: 150, B: 243
26: Mapped Values:    R: 255, G: 176, B: 255
26: magenta decoded as white
27: Original Average: R: 233, G: 158, B: 242
27: Mapped Values:    R: 255, G: 192, B: 255
27: magenta decoded as white
28: Original Average: R: 239, G: 89, B: 158
28: Mapped Values:    R: 255, G: 52, B: 227
28: red decoded as magenta
29: Original Average: R: 249, G: 90, B: 138
29: Mapped Values:    R: 255, G: 54, B: 184
29: red decoded as magenta
30: Original Average: R: 240, G: 151, B: 245
30: Mapped Values:    R: 255, G: 178, B: 255
30: magenta decoded as white
31: Original Average: R: 246, G: 114, B: 140
31: Mapped Values:    R: 255, G: 103, B: 188
31: red decoded as magenta
32: Original Average: R: 243, G: 85, B: 141
32: Mapped Values:    R: 255, G: 44, B: 190
32: red decoded as magenta
33: Original Average: R: 241, G: 120, B: 148
33: Mapped Values:    R: 255, G: 115, B: 205
33: red decoded as magenta
34: Original Average: R: 247, G: 85, B: 130
34: Mapped Values:    R: 255, G: 44, B: 167
34: red decoded as magenta
35: Original Average: R: 12, G: 237, B: 123
35: Mapped Values:    R: 0, G: 255, B: 152
35: green decoded as cyan
36: Original Average: R: 237, G: 102, B: 138
36: Mapped Values:    R: 255, G: 78, B: 184
36: red decoded as magenta
37: Original Average: R: 245, G: 88, B: 142
37: Mapped Values:    R: 255, G: 50, B: 192
37: red decoded as magenta
38: Original Average: R: 51, G: 60, B: 125
```

	38: Mapped Values: R: 0, G: 0, B: 156 38: black decoded as blue 39: Original Average: R: 240, G: 116, B: 141 39: Mapped Values: R: 255, G: 107, B: 190 39: red decoded as magenta 40: Original Average: R: 251, G: 90, B: 145 40: Mapped Values: R: 255, G: 54, B: 199 40: red decoded as magenta 41: Original Average: R: 243, G: 100, B: 136 41: Mapped Values: R: 255, G: 74, B: 180 41: red decoded as magenta number of failures: 41

Using as reference points a module inside of the barcode instead of the borders for Test_018.jpg:

Test_018.jpg	1: Original Average: R: 31, G: 158, B: 207 1: Mapped Values: R: 0, G: 161, B: 255 1: blue decoded as cyan 2: Original Average: R: 247, G: 92, B: 154 2: Mapped Values: R: 255, G: 56, B: 171 2: red decoded as magenta 3: Original Average: R: 253, G: 98, B: 163 3: Mapped Values: R: 255, G: 65, B: 186 3: red decoded as magenta 4: Original Average: R: 219, G: 63, B: 145 4: Mapped Values: R: 244, G: 9, B: 156 4: magenta decoded as red 5: Original Average: R: 233, G: 158, B: 242 5: Mapped Values: R: 255, G: 161, B: 255 5: magenta decoded as white 6: Original Average: R: 239, G: 89, B: 158 6: Mapped Values: R: 255, G: 51, B: 178 6: red decoded as magenta number of failures: 6

Scenario 8:

Reading matrix: 7x3 pixels

For the black/white reference points calculate average for each color channel separately

Thresholds: R=G=160

- rg (black or blue): B= 150
- rG (green or cyan): B=150
- Rg (red or magenta): B=165
- RG (yellow or white):
  B= original(min{R, G}) * 0.7

Test_010.jpg	1: Original Average: R: 165, G: 32, B: 97 1: Mapped Values: R: 255, G: 7, B: 161 1: magenta decoded as red 2: Original Average: R: 158, G: 32, B: 95 2: Mapped Values: R: 247, G: 7, B: 157 2: magenta decoded as red 3: Original Average: R: 12, G: 120, B: 82 3: Mapped Values: R: 0, G: 180, B: 131 3: cyan decoded as green number of failures: 3
Test_011.jpg	number of failures: 0
Test_012.jpg	1: Original Average: R: 83, G: 12, B: 49 1: Mapped Values: R: 217, G: 17, B: 143 1: magenta decoded as red 2: Original Average: R: 1, G: 5, B: 42 2: Mapped Values: R: 0, G: 0, B: 121 2: blue decoded as black 3: Original Average: R: 2, G: 71, B: 42 3: Mapped Values: R: 0, G: 188, B: 121 3: cyan decoded as green 4: Original Average: R: 4, G: 68, B: 50 4: Mapped Values: R: 0, G: 179, B: 146 4: cyan decoded as green 5: Original Average: R: 89, G: 12, B: 54 5: Mapped Values: R: 234, G: 17, B: 159 5: magenta decoded as red 6: Original Average: R: 1, G: 5, B: 46 6: Mapped Values: R: 0, G: 0, B: 133 6: blue decoded as black 7: Original Average: R: 1, G: 7, B: 49 7: Mapped Values: R: 0, G: 2, B: 143 7: blue decoded as black 8: Original Average: R: 4, G: 74, B: 49 8: Mapped Values: R: 0, G: 197, B: 143 8: cyan decoded as green 9: Original Average: R: 8, G: 79, B: 48 9: Mapped Values: R: 2, G: 211, B: 140 9: cyan decoded as green 10: Original Average: R: 95, G: 12, B: 55 10: Mapped Values: R: 252, G: 17, B: 162 10: magenta decoded as red 11: Original Average: R: 5, G: 79, B: 49 11: Mapped Values: R: 0, G: 211, B: 143 11: cyan decoded as green 12: Original Average: R: 6, G: 79, B: 50 12: Mapped Values: R: 0, G: 211, B: 146 12: cyan decoded as green 13: Original Average: R: 15, G: 85, B: 47 13: Mapped Values: R: 22, G: 228, B: 137 13: cyan decoded as green 14: Original Average: R: 6, G: 80, B: 48 14: Mapped Values: R: 0, G: 214, B: 140 14: cyan decoded as green 15: Original Average: R: 1, G: 4, B: 50

	15: Mapped Values: R: 0, G: 0, B: 146 15: blue decoded as black 16: Original Average: R: 98, G: 95, B: 66 16: Mapped Values: R: 255, G: 255, B: 197 16: white decoded as yellow number of failures: 16
Test_013.jpg	1: Original Average: R: 159, G: 32, B: 91 1: Mapped Values: R: 253, G: 7, B: 151 1: magenta decoded as red number of failures: 1
Test_014.jpg	number of failures: 0
Test_015.jpg	number of failures: 0
Test_016.jpg	1: Original Average: R: 253, G: 139, B: 244 1: Mapped Values: R: 255, G: 161, B: 255 1: magenta decoded as white 2: Original Average: R: 253, G: 140, B: 250 2: Mapped Values: R: 255, G: 163, B: 255 2: magenta decoded as white 3: Original Average: R: 249, G: 139, B: 245 3: Mapped Values: R: 255, G: 161, B: 255 3: magenta decoded as white 4: Original Average: R: 242, G: 139, B: 246 4: Mapped Values: R: 255, G: 161, B: 255 4: magenta decoded as white number of failures: 4
Test_017.jpg	1: Original Average: R: 225, G: 211, B: 151 1: Mapped Values: R: 253, G: 239, B: 146 1: yellow decoded as white 2: Original Average: R: 221, G: 219, B: 164 2: Mapped Values: R: 247, G: 251, B: 166 2: yellow decoded as white 3: Original Average: R: 233, G: 233, B: 170 3: Mapped Values: R: 255, G: 255, B: 176 3: yellow decoded as white number of failures: 3
Test_018.jpg*	1: Original Average: R: 249, G: 91, B: 153 1: Mapped Values: R: 255, G: 54, B: 170 1: red decoded as magenta 2: Original Average: R: 253, G: 97, B: 162 2: Mapped Values: R: 255, G: 64, B: 185 2: red decoded as magenta 3: Original Average: R: 219, G: 63, B: 145 3: Mapped Values: R: 244, G: 9, B: 156 3: magenta decoded as red 4: Original Average: R: 242, G: 88, B: 159 4: Mapped Values: R: 255, G: 49, B: 180 4: red decoded as magenta number of failures: 4

* Using as reference points a module inside of the barcode instead of the borders:

Scenario 9:

Reading matrix: 9x3 pixels

For the black/white reference points calculate the median for each color channel separately (instead of average!)

Thresholds: R=G=160

- rg (black or blue): B= 150
- rG (green or cyan): B=150
- Rg (red or magenta): B=165
- RG (yellow or white):
  B= original(min{R, G}) * 0.7

Test_010.jpg	1: Original Average: R: 165, G: 33, B: 98 1: Mapped Values: R: 255, G: 9, B: 163 1: magenta decoded as red 2: Original Average: R: 158, G: 32, B: 95 2: Mapped Values: R: 247, G: 7, B: 157 2: magenta decoded as red 3: Original Average: R: 12, G: 121, B: 84 3: Mapped Values: R: 0, G: 182, B: 135 3: cyan decoded as green number of failures: 3
Test_011.jpg	number of failures: 0
Test_012.jpg	1: Original Average: R: 84, G: 13, B: 50 1: Mapped Values: R: 220, G: 20, B: 146 1: magenta decoded as red 2: Original Average: R: 1, G: 5, B: 42 2: Mapped Values: R: 0, G: 0, B: 121 2: blue decoded as black 3: Original Average: R: 3, G: 72, B: 44 3: Mapped Values: R: 0, G: 191, B: 127 3: cyan decoded as green 4: Original Average: R: 4, G: 69, B: 51 4: Mapped Values: R: 0, G: 182, B: 149 4: cyan decoded as green 5: Original Average: R: 90, G: 12, B: 55 5: Mapped Values: R: 237, G: 17, B: 162 5: magenta decoded as red 6: Original Average: R: 1, G: 6, B: 46 6: Mapped Values: R: 0, G: 0, B: 133 6: blue decoded as black 7: Original Average: R: 0, G: 7, B: 49 7: Mapped Values: R: 0, G: 2, B: 143 7: blue decoded as black 8: Original Average: R: 5, G: 75, B: 49 8: Mapped Values: R: 0, G: 199, B: 143 8: cyan decoded as green 9: Original Average: R: 7, G: 79, B: 49 9: Mapped Values: R: 0, G: 211, B: 143 9: cyan decoded as green 10: Original Average: R: 96, G: 11, B: 56 10: Mapped Values: R: 255, G: 14, B: 165 10: magenta decoded as red 11: Original Average: R: 4, G: 79, B: 49 11: Mapped Values: R: 0, G: 211, B: 143 11: cyan decoded as green 12: Original Average: R: 16, G: 85, B: 49 12: Mapped Values: R: 25, G: 228, B: 143 12: cyan decoded as green 13: Original Average: R: 6, G: 80, B: 49 13: Mapped Values: R: 0, G: 214, B: 143 13: cyan decoded as green 14: Original Average: R: 99, G: 96, B: 67 14: Mapped Values: R: 255, G: 255, B: 200 14: white decoded as yellow

	number of failures: 14
Test_013.jpg	1: Original Average: R: 159, G: 32, B: 92 1: Mapped Values: R: 253, G: 7, B: 153 1: magenta decoded as red number of failures: 1
Test_014.jpg	number of failures: 0
Test_015.jpg	1: Original Average: R: 209, G: 85, B: 203 1: Mapped Values: R: 255, G: 163, B: 255 1: magenta decoded as white 2: Original Average: R: 217, G: 44, B: 68 2: Mapped Values: R: 255, G: 61, B: 167 2: red decoded as magenta 3: Original Average: R: 215, G: 54, B: 70 3: Mapped Values: R: 255, G: 86, B: 172 3: red decoded as magenta number of failures: 3
Test_016.jpg	1: Original Average: R: 246, G: 139, B: 244 1: Mapped Values: R: 255, G: 161, B: 255 1: magenta decoded as white 2: Original Average: R: 251, G: 139, B: 247 2: Mapped Values: R: 255, G: 161, B: 255 2: magenta decoded as white 3: Original Average: R: 254, G: 139, B: 245 3: Mapped Values: R: 255, G: 161, B: 255 3: magenta decoded as white 4: Original Average: R: 252, G: 139, B: 250 4: Mapped Values: R: 255, G: 161, B: 255 4: magenta decoded as white 5: Original Average: R: 254, G: 140, B: 251 5: Mapped Values: R: 255, G: 163, B: 255 5: magenta decoded as white 6: Original Average: R: 250, G: 139, B: 245 6: Mapped Values: R: 255, G: 161, B: 255 6: magenta decoded as white 7: Original Average: R: 253, G: 139, B: 250 7: Mapped Values: R: 255, G: 161, B: 255 7: magenta decoded as white 8: Original Average: R: 249, G: 140, B: 245 8: Mapped Values: R: 255, G: 163, B: 255 8: magenta decoded as white 9: Original Average: R: 242, G: 139, B: 246 9: Mapped Values: R: 255, G: 161, B: 255 9: magenta decoded as white number of failures: 9
Test_017.jpg	1: Original Average: R: 242, G: 227, B: 159 1: Mapped Values: R: 255, G: 255, B: 158 1: yellow decoded as white 2: Original Average: R: 229, G: 214, B: 154 2: Mapped Values: R: 255, G: 244, B: 151 2: yellow decoded as white 3: Original Average: R: 218, G: 223, B: 163 3: Mapped Values: R: 242, G: 255, B: 165 3: yellow decoded as white 4: Original Average: R: 204, G: 224, B: 144 4: Mapped Values: R: 219, G: 255, B: 135 4: yellow decoded as white 5: Original Average: R: 221, G: 223, B: 155 5: Mapped Values: R: 247, G: 255, B: 152 5: yellow decoded as white 6: Original Average: R: 231, G: 230, B: 167 6: Mapped Values: R: 255, G: 255, B: 171 6: yellow decoded as white number of failures: 6
Test_018.jpg*	1: Original Average: R: 251, G: 87, B: 152 1: Mapped Values: R: 255, G: 48, B: 168 1: red decoded as magenta 2: Original Average: R: 254, G: 94, B: 161 2: Mapped Values: R: 255, G: 59, B: 183 2: red decoded as magenta 3: Original Average: R: 16, G: 229, B: 144 3: Mapped Values: R: 0, G: 255, B: 155

	3: green decoded as cyan 4: Original Average: R: 219, G: 64, B: 146 4: Mapped Values: R: 244, G: 11, B: 158 4: magenta decoded as red 5: Original Average: R: 245, G: 88, B: 159 5: Mapped Values: R: 255, G: 49, B: 180 5: red decoded as magenta number of failures: 5

* Using as reference points a module inside of the barcode instead of the borders:

Scenario 10:

Reading matrix: 9x3 pixels

For the black/white reference points calculate average for each color channel separately

Taking a bigger area for the white/black reference points

Thresholds: R=G=160

- rg (black or blue): B= 150
- rG (green or cyan): B=150
- Rg (red or magenta): B=165
- RG (yellow or white):
  B= original(min{R, G}) * 0.7

Test_010.jpg	1: Original Average: R: 157, G: 31, B: 95 1: Mapped Values: R: 253, G: 11, B: 163 1: magenta decoded as red 2: Original Average: R: 12, G: 120, B: 82 2: Mapped Values: R: 0, G: 189, B: 135 2: cyan decoded as green number of failures: 2
Test_011.jpg	1: Original Average: R: 3, G: 4, B: 48 1: Mapped Values: R: 0, G: 0, B: 150 1: blue decoded as black 2: Original Average: R: 3, G: 4, B: 47 2: Mapped Values: R: 0, G: 0, B: 147 2: blue decoded as black 3: Original Average: R: 89, G: 10, B: 49 3: Mapped Values: R: 252, G: 17, B: 154 3: magenta decoded as red 4: Original Average: R: 4, G: 68, B: 48 4: Mapped Values: R: 0, G: 189, B: 150 4: cyan decoded as green number of failures: 4
Test_012.jpg	1: Original Average: R: 83, G: 12, B: 49 1: Mapped Values: R: 236, G: 21, B: 149 1: magenta decoded as red 2: Original Average: R: 1, G: 5, B: 42 2: Mapped Values: R: 0, G: 0, B: 125 2: blue decoded as black 3: Original Average: R: 3, G: 71, B: 42 3: Mapped Values: R: 0, G: 205, B: 125 3: cyan decoded as green 4: Original Average: R: 1, G: 5, B: 46 4: Mapped Values: R: 0, G: 0, B: 139 4: blue decoded as black 5: Original Average: R: 1, G: 7, B: 49 5: Mapped Values: R: 0, G: 6, B: 149 5: blue decoded as black 6: Original Average: R: 4, G: 74, B: 49 6: Mapped Values: R: 0, G: 214, B: 149 6: cyan decoded as green 7: Original Average: R: 8, G: 79, B: 49 7: Mapped Values: R: 9, G: 230, B: 149 7: cyan decoded as green 8: Original Average: R: 6, G: 78, B: 49 8: Mapped Values: R: 3, G: 227, B: 149 8: cyan decoded as green 9: Original Average: R: 14, G: 85, B: 48 9: Mapped Values: R: 27, G: 248, B: 145 9: cyan decoded as green 10: Original Average: R: 5, G: 79, B: 49 10: Mapped Values: R: 0, G: 230, B: 149 10: cyan decoded as green number of failures: 10
Test_013.jpg	1: Original Average: R: 159, G: 32, B: 91 1: Mapped Values: R: 255, G: 12, B: 156

	1: magenta decoded as red number of failures: 1
Test_014.jpg	number of failures: 0
Test_015.jpg	number of failures: 0
Test_016.jpg	number of failures: 0
Test_017.jpg	1: Original Average: R: 242, G: 225, B: 159 1: Mapped Values: R: 255, G: 255, B: 167 1: yellow decoded as white 2: Original Average: R: 227, G: 213, B: 150 2: Mapped Values: R: 255, G: 247, B: 153 2: yellow decoded as white 3: Original Average: R: 223, G: 220, B: 162 3: Mapped Values: R: 255, G: 255, B: 172 3: yellow decoded as white 4: Original Average: R: 231, G: 231, B: 166 4: Mapped Values: R: 255, G: 255, B: 179 4: yellow decoded as white number of failures: 4
Test_018.jpg	1: Original Average: R: 31, G: 158, B: 207 1: Mapped Values: R: 0, G: 161, B: 255 1: blue decoded as cyan 2: Original Average: R: 247, G: 92, B: 154 2: Mapped Values: R: 255, G: 56, B: 171 2: red decoded as magenta 3: Original Average: R: 253, G: 98, B: 163 3: Mapped Values: R: 255, G: 65, B: 186 3: red decoded as magenta 4: Original Average: R: 219, G: 63, B: 145 4: Mapped Values: R: 244, G: 9, B: 156 4: magenta decoded as red 5: Original Average: R: 233, G: 158, B: 242 5: Mapped Values: R: 255, G: 161, B: 255 5: magenta decoded as white 6: Original Average: R: 239, G: 89, B: 158 6: Mapped Values: R: 255, G: 51, B: 178 6: red decoded as magenta number of failures: 6

Scenario 11:

Reading matrix: 7x3 pixels

For the black/white reference points calculate average for each color channel separately

Taking a bigger area for the white/black reference points

Thresholds: R=G=160

- rg (black or blue): B= 150

- rG (green or cyan): B=150

- Rg (red or magenta): B=165

- RG (yellow or white):
  B= original(min{R, G}) * 0.7

Test_010.jpg	1: Original Average: R: 158, G: 32, B: 95 1: Mapped Values: R: 255, G: 13, B: 163 1: magenta decoded as red 2: Original Average: R: 12, G: 120, B: 82 2: Mapped Values: R: 0, G: 189, B: 135 2: cyan decoded as green number of failures: 2
Test_011.jpg	1: Original Average: R: 3, G: 4, B: 48 1: Mapped Values: R: 0, G: 0, B: 150 1: blue decoded as black 2: Original Average: R: 3, G: 4, B: 47 2: Mapped Values: R: 0, G: 0, B: 147 2: blue decoded as black 3: Original Average: R: 89, G: 10, B: 49 3: Mapped Values: R: 252, G: 17, B: 154 3: magenta decoded as red 4: Original Average: R: 4, G: 68, B: 48 4: Mapped Values: R: 0, G: 189, B: 150 4: cyan decoded as green number of failures: 4
Test_012.jpg	1: Original Average: R: 83, G: 12, B: 49 1: Mapped Values: R: 236, G: 21, B: 149 1: magenta decoded as red 2: Original Average: R: 1, G: 5, B: 42 2: Mapped Values: R: 0, G: 0, B: 125 2: blue decoded as black 3: Original Average: R: 2, G: 71, B: 42 3: Mapped Values: R: 0, G: 205, B: 125 3: cyan decoded as green 4: Original Average: R: 89, G: 12, B: 54 4: Mapped Values: R: 255, G: 21, B: 165 4: magenta decoded as red 5: Original Average: R: 1, G: 5, B: 46 5: Mapped Values: R: 0, G: 0, B: 139 5: blue decoded as black 6: Original Average: R: 1, G: 7, B: 49 6: Mapped Values: R: 0, G: 6, B: 149 6: blue decoded as black 7: Original Average: R: 4, G: 74, B: 49 7: Mapped Values: R: 0, G: 214, B: 149 7: cyan decoded as green 8: Original Average: R: 8, G: 79, B: 48 8: Mapped Values: R: 9, G: 230, B: 145 8: cyan decoded as green 9: Original Average: R: 5, G: 79, B: 49 9: Mapped Values: R: 0, G: 230, B: 149 9: cyan decoded as green 10: Original Average: R: 15, G: 85, B: 47 10: Mapped Values: R: 30, G: 248, B: 142 10: cyan decoded as green 11: Original Average: R: 6, G: 80, B: 48 11: Mapped Values: R: 3, G: 233, B: 145 11: cyan decoded as green

	12: Original Average: R: 98, G: 95, B: 66 12: Mapped Values: R: 255, G: 255, B: 205 12: white decoded as yellow number of failures: 12
Test_013.jpg	1: Original Average: R: 159, G: 32, B: 91 1: Mapped Values: R: 255, G: 12, B: 156 1: magenta decoded as red number of failures: 1
Test_014.jpg	number of failures: 0
Test_015.jpg	number of failures: 0
Test_016.jpg	1: Original Average: R: 253, G: 140, B: 250 1: Mapped Values: R: 255, G: 162, B: 255 1: magenta decoded as white number of failures: 1
Test_017.jpg	1: Original Average: R: 225, G: 211, B: 151 1: Mapped Values: R: 255, G: 243, B: 154 1: yellow decoded as white 2: Original Average: R: 221, G: 219, B: 164 2: Mapped Values: R: 255, G: 255, B: 176 2: yellow decoded as white 3: Original Average: R: 233, G: 233, B: 170 3: Mapped Values: R: 255, G: 255, B: 185 3: yellow decoded as white number of failures: 3
Test_018.jpg	1: Original Average: R: 249, G: 91, B: 153 1: Mapped Values: R: 255, G: 54, B: 170 1: red decoded as magenta 2: Original Average: R: 253, G: 97, B: 162 2: Mapped Values: R: 255, G: 64, B: 185 2: red decoded as magenta 3: Original Average: R: 219, G: 63, B: 145 3: Mapped Values: R: 244, G: 9, B: 156 3: magenta decoded as red 4: Original Average: R: 242, G: 88, B: 159 4: Mapped Values: R: 255, G: 49, B: 180 4: red decoded as magenta number of failures: 4

Appendix E

Test Results Evaluation

Scenario 1: Testing different amount of modules within a barcode

Test scenario: **Different amount of modules**	Picture number	Number of failures (corner points programmatically detected)	Number of failures (corner points manually entered)*
64x64	1	0	
	2	0	
	3	0	
128x128	1	Recognizer module fails	729
	2	Recognizer module fails	875
	3	Recognizer module fails	701

*If the recognizer module fails or if the detected corner points are inaccurate, the test is repeated with manually entered corner points. Rows which do not hold a value in the corresponding row had already accurate detected corner points.

Scenario 2: Testing rotated barcodes

Test scenario: **Rotation**	Picture number	Number of failures (corner points programmatically detected)	Number of failures (corner points manually entered)
+10°	1	16	1
	2	0	
	3	0	
+15°	1	110	10
	2	0	
	3	4	
+20°	1	0	
	2	11	
	3	8	
+30°	1	319	
	2	50	
	3	1	
+40°	1	Recognizer module fails	1
	2	Recognizer module fails	0
	3	Recognizer module fails	2
-10°	1	0	
	2	8	1
	3	0	
-15°	1	2	0
	2	0	
	3	6	0
-20°	1	0	
	2	0	
	3	1	
-30°	1	0	
	2	0	
	3	0	
-40°	1	Recognizer module fails	0
	2	Recognizer module fails	0
	3	Recognizer module fails	0

Scenario 3: Testing exposure values

Test scenario: **Exposure**	Picture number	Number of failures (corner points programmatically detected)	Number of failures (corner points manually entered)
+0.5	1	0	
	2	0	
	3	0	
+1.0	1	0	
	2	0	
	3	0	
+1.5	1	0	
	2	0	
	3	0	
+2.0	1	0	
	2	0	
	3	0	
-0.5	1	0	
	2	0	
	3	0	
-1.0	1	0	
	2	0	
	3	1	
-1.5	1	0	
	2	0	
	3	1	
-2.0	1	0	
	2	0	
	3	0	

Scenario 4: Testing horizontal distortion

Test scenario: **Horizontal distortion**	Picture number	Number of failures (corner points programmatically detected)	Number of failures (corner points manually entered)
+10°	1	0	
	2	2	
	3	0	
+15°	1	0	
	2	0	
	3	1	
+20°	1	67	64
	2	78	76
	3	79	60
+30°	1	283	
	2	Recognizer module fails	292
	3	Recognizer module fails	288
-10°	1	6	7
	2	32	2
	3	9	2
-15°	1	27	
	2	24	
	3	4	
-20°	1	141	113
	2	100	105
	3	141	114
-30°	1	580	270
	2	423	304
	3	413	296

Scenario 5: Testing vertical distortion

Test scenario: **Vertical distortion**	Picture number	Number of failures (corner points programmatically detected)	Number of failures (corner points manually entered)
+10°	1	12	5
	2	91	1
	3	21	31
+15°	1	Recognizer module fails	1
	2	Recognizer module fails	0
	3	Recognizer module fails	0
+20°	1	Recognizer module fails	1
	2	Recognizer module fails	3
	3	Recognizer module fails	81
-10°	1	0	
	2	2	
	3	2	
-15°	1	0	
	2	11	0
	3	0	
-20°	1	0	
	2	43	75
	3	8	

Scenario 6: Testing light reflections

Test scenario: **Light reflection**	Picture number	Number of failures (corner points programmatically detected)	Number of failures (corner points manually entered)
1_light_source	1	0	
	2	0	
	3	2	0
2_light_sources	1	1	
	2	0	
	3	0	

Scenario 7: Testing movement of the device

Test scenario: **Moving the device**	Picture number	Number of failures (corner points programmatically detected)	Number of failures (corner points manually entered)
calm	1	13	0
	2	2	
	3	0	
shaky	1	Recognizer module fails	
	2	Recognizer module fails	
	3	880	

Scenario 8: Testing the distance to the screen

Test scenario: **Distance to screen**	Picture number	Number of failures (corner points programmatically detected)	Number of failures (corner points manually entered)
50cm	1	6	2
	2	0	2
	3	7	7
55cm	1	3	
	2	0	
	3	8	
60cm	1	27	15
	2	7	30
	3	17	25
65cm	1	10	17
	2	12	19
	3	13	19
70cm	1	22	
	2	96	
	3	9	

Appendix F

Installation Guidelines

This guideline describes how to build and install the application from the command line[1].

Prerequisites:

- JDK[2]
- Ant[3]
- Android SDK[4]
- Installed drivers for the corresponding mobile phone.

Assure to have set the environment variables to specify the paths to each directory. The following instruction assumes that the environment variables are correctly set.

1. Uncompress the *Code3D.rar* archive into a directory. The further points assume that the archive is uncompressed to *C:/Code3D*.
2. Open a command-line and run:
   `android update project --target 11 --name Code3D --path C:/Code3D`
   This creates the *build.xml* file for Ant.
3. In the command-line navigate to *C:/Code3D*.
4. Run:
   `ant debug`
   This compiles the source files and creates the debug .apk file inside the project *bin/* directory, named *Code3D-debug.apk*.
   Building the application in debug mode avoids signing it. The application can therefore not be released (e.g. on the Market), but can be installed and run on a device.

[1] according to `http://developer.android.com/guide/developing/building/building-cmdline.html`

[2] `http://www.oracle.com/technetwork/java/javase/downloads/index.html`

[3] `http://ant.apache.org/`

[4] `http://developer.android.com/sdk/index.html`

5. Connect the mobile phone with the computer via USB. Assure that non-Market applications can be installed (usually under *Settings−− >Applications−− >Allow installation of non-Market applications*).

6. In the command-line run:
   `adb -d install C:/Code3D/bin/Code3D-debug.apk`
   The application should be installed (named *Code3D*) on the mobile phone now and can be executed.

Appendix G

Contents Of The CD

The CD-ROM contains the following files:

- **Zusfsg.pdf**
  German version of the abstract
- **Abstract.pdf**
  English version of the abstract
- **Bachelorarbeit.pdf**
  PDF version of this thesis
- **Code3D.rar**
  Source code of the prototype
- **Thesis.rar**
  Figures and LaTeX source files of this thesis