KTH

EQ2440 Project Red
Visual-Based In-Flight File Transfer

by

Changyang Zhang

Charalampos Kalalas

Claire Lamé

Liu Xiuming

Panchamy Krishnan K

Thomas Fehrenbach

Teacher: Per ZETTERBERG

Assistant: Iqbal HUSSAIN

May 2013

"One test is worth a thousand expert opinions."

Wernher Von Braun

Abstract

KTH
Electrical School

by Changyang Zhang
Charalampos Kalalas
Claire Lamé
Liu Xiuming
Panchamy Krishnan K
Thomas Fehrenbach

This project targets a situation onboard a plane where two passengers, having Android smartphone devices, want to transmit a file from one phone's SD-card to the other. Radio transmission is prohibited onboard, since it may interfere with the aircraft navigation and communication systems. This leads to the need of finding an innovative way of transmission. This paper explains the development of such a technology, which is based on visual transmission via QR codes. Different QR codes have been implemented and tested for fast and reliable transfer of data. Our application can achieve a speed of more than 1 KB/s. The reliability depends on the QR code and the version selected. Testing shows that the best performance is attained with color QR codes.

Acknowledgements

This project could not have been performed without the patience, assistance and support of several individuals. First and foremost, we would like to thank our project supervisor, Per Zetterberg, for his valuable advise. We would also like to thank our project assistant, Iqbal Hussain, for helping us all through the project. We truly appreciated the great material and facilities provided. We would like to especially mention the microwave and coffee machine which helped us complete this project immensely.

Thanks also goes to the Green Team for the tough but fair competition, and the Yellow Team for their support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

This project targets a situation onboard an aircraft. Two passengers, who own Android smartphone devices, want to transfer a file, stored on one phone's SD-CARD to the other one. The use of radio transmitters and receivers is forbidden onboard, due to the potential risk of interferences with the aircraft navigation and communication systems.

To transmit the file, one possibility would be to use sound, which was explored in a project last year [1]. However, this solution comes at the cost of disturbing surrounding passengers. This year, we have implemented a system based on visual transmission, using QR codes. In our implementation, the screen is used as a visual transmitter, and the camera as a visual receiver.

1.2 Specifications

The project is divided into the theory, in order to design a relevant solution, Matlab simulations and the Android implementation. Programming is done on Matlab [2] and Eclipse [3] with Android Development Tools [4]. Dropbox and Git [5] are used to share the Matlab and Java code between the team members.

To fulfil basic requirements, our implementation can reliably transfer a small file (1Kb) at a speed of at least 100 bits/sec, with at least 100 bits/frame. This has to be done without using the existing open source library ZXing.

After achieving the basic requirements we have improved the transmission rate by increasing the transmission speed, and by adding more data in one QR code. We have designed our own standard of QR codes, away from the ISO/IEC 18004-2006 standard [6], in black and white, and in color.

1.3 Outline

This paper aims at explaining our solution and the choices we made. This introduction forms the first chapter.

The second chapter of this paper explains in detail how the data is encoded and placed in matrices to build QR codes. On the transmitter's phone, binary data is read from a chosen file on the SD card and encoded. Several schemes can be employed when encoding. The ISO/IEC 18004-2006 standard [6], defines several versions of QR codes and their capacities. To meet the basic requirements of 100 bits/frame, we have chosen the standard version number 2 for building the QR symbols. To meet the advanced requirements, we have implemented our own standard.

Chapter 3 describes the transformations used to process the image captured by the receiver's phone, in order to detect the QR codes. Indeed, the data is sent as QR codes from the first phone and captured by the second phone as an image. This image is processed to extract the QR code information as bit streams. Then, the extracted bit streams are decoded using QR code decoder. The major task in this part is to extract the QR code from the image in which the QR code may be tilted or distorted.

Chapter 4 details how the data is decoded from the bit streams to recover the original message, and generate the output file.

The Android prototype is described in chapter 5. It encompasses the development environment, a new framework for development, and explains the major parts of the Java implementation.

Finally, our implementation is analysed in terms of performances in chapter 6. Possible improvements and future work are discussed.

Chapter 2

Encoding QR Codes

This chapter describes the structure of QR codes, and the encoding procedure, that constructs a QR symbol from a message. Since several versions have been implemented, the standard version, based on the ISO/IEC 18004-2006 standard [6] is first presented, followed by the non-standard model.

2.1 Standard QR codes

Our initial implementation of QR codes was entirely based on the ISO/IEC 18004-2006 standard [6]. Following the principles of this standard, a QR symbol was created taking as inputs the desired encoded message, the version of the QR symbol (a parameter that is related to the size) and the error correction level.

2.1.1 General structure and specifications

According to the standard, the QR symbol consists of an array of nominally square modules arranged in an overall square pattern. As for the representation of data, a dark module is nominally a binary one and a light module is nominally a binary zero. In three (top left, top right and bottom left) corners of the symbol, special position detection patterns, named finder patterns, are located allowing for 360 degree (omni-directional) high-speed reading of the code. In addition, a wide range of sizes of symbol is provided; from 21×21 modules (version 1) till 177×177 (version 40) modules with an increasing step of four modules per side. Four different levels of Reed-Solomon error correction L, M, Q and H (in increasing order of capacity) are specified, allowing recovery of 7%, 15%, 25% and 30% of the symbol codewords respectively. Figure 2.1 illustrates the structure of a version 7 QR symbol.

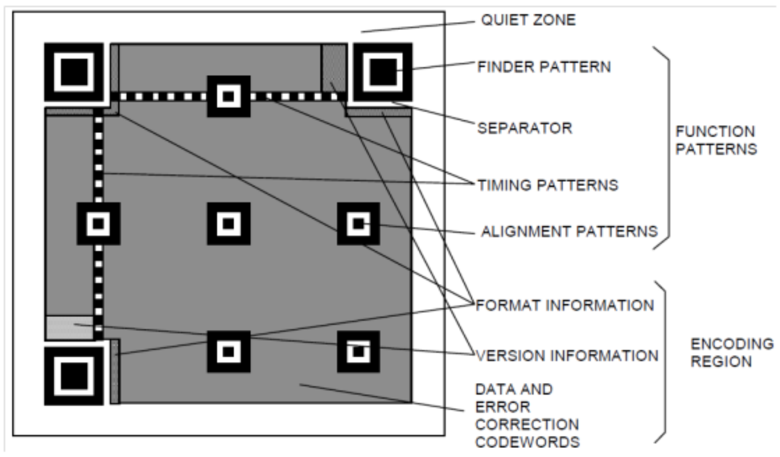


Figure 2.1: Structure of a QR symbol version 7 [6]

As shown in figure 2.1, the symbol structure is divided into two different parts; the function patterns and the encoding region. The symbol is also surrounded on all four sides by a quiet zone border.

2.1.2 Function patterns

Function patterns are used for the location of the symbol and the identification of its characteristics, in order to facilitate the decoding process. Function patterns are not used for encoding data.

2.1.2.1 Finder patterns

The three finder patterns are identical and are located at the upper left, upper right and lower left corners of the symbol. Each finder pattern may be viewed as three superimposed concentric squares and is constructed of dark 7×7 modules, light 5×5 modules and dark 3×3 modules. The ratio of module widths in each finder pattern is 1:1:3:1:1. The symbol is encoded in a way that similar patterns have a low probability of being encountered elsewhere inside the symbol, enabling rapid identification of the QR symbol in the field of view. The separator is a one module wide function pattern of all light modules that acts as a border between the finder patterns and the encoding region of the symbol.

2.1.2.2 Alignment patterns

The alignment patterns are reference patterns, placed at a specific positions for each particular version, and allow the QR reader to correct distortions when the code is bent or curved. They are applied in QR symbols version 2 or larger. Each alignment pattern may be viewed as three superimposed concentric squares; dark 5×5 modules, light 3×3 modules and a single central dark module. The number of alignment patterns depends on the symbol version and they are placed in fixed positions that are defined in Annex E of the standard [6].

2.1.2.3 Timing patterns

The timing patterns are alternating sequences of dark and light modules enabling module coordinates in the symbol to be determined. They are placed in two particular parts of the symbol; the horizontal timing pattern is located in the sixth[1] row of the symbol between the separators for the upper finder patterns and the vertical timing pattern is located in the sixth column of the symbol between the separators for the left-hand finder patterns.

2.1.3 Encoding region

On the other hand, the encoding region is the region of the symbol that is not occupied by function patterns and is available for encoding of data and error correction codewords, and for format and version information.

2.1.3.1 Format information

The format information is an encoded pattern containing information on symbol characteristics, essential to enable the rest of the encoding region to be decoded. It is placed next to the three finder patterns and provides information about the error correction level and the masking pattern that is applied on the data.

2.1.3.2 Masking

Masking is defined as a process of XORing the encoding region with a mask pattern, in order to provide the symbol with a more evenly balanced numbers of dark and light modules, and to reduce the occurrence of patterns which would interfere with fast processing of the image.

[1]Module positions are defined by their row and column coordinates in the symbol, in the form (i, j) where i represents the row (counting from the top downwards) and j the column (counting from left to right) in which the module is located, with counting beginning at 0. Module (0, 0) is therefore located at the upper left corner of the symbol [6]

2.1.3.3 Version information

The version information is a pattern that contains information on the symbol version, together with error correction bits for this data. Version information is applied to versions higher than seven.

2.1.3.4 Initialisations in the QR symbol

After specifying the version of the QR symbol, the function patterns are placed inside the symbol. Figure 2.2 shows this placement into a QR symbol version 7. Note that the module in position (4V + 9, 8), where V is the version number, is always dark and does not form part of the format information.

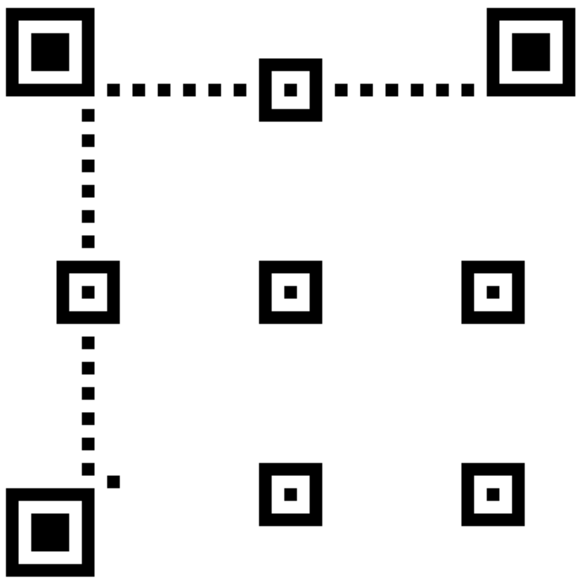


FIGURE 2.2: Function patterns in a QR symbol version 7

For this QR symbol, the unavailable area for data placement is illustrated in figure 2.3. This area consists of the previously defined function patterns, together with the areas for the format and the version information.

2.1.4 Encoding procedure

As far as the encoding procedure is concerned, this process can be viewed as a combination of two particular sub-processes; the initialisations regarding the QR symbol and its fulfilment with the input data.

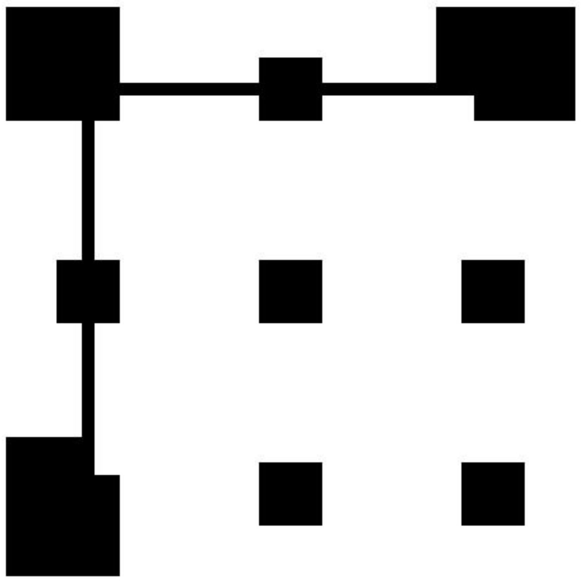


FIGURE 2.3: Unavailable area in a QR symbol version 7

For the representation of each character into a bit string, the byte mode is used where each character corresponds to an 8-bit codeword. In this mode, one 8-bit codeword directly represents the byte value of the input data character.

2.1.4.1 Initialisation

By properly choosing the desired version of QR symbol, several parameters are initialised. At first, the size of the QR symbol, which corresponds to the number of square modules in the horizontal and vertical direction, is determined according to Table 1 of the standard [6]. The same table provides information about the remainder bits for each version which are modules not fulfilled with data that are used in order to fill empty positions of the symbol encoding region after the construction of the final bit sequence. They are used when the area of the encoding region available for symbol characters does not divide exactly into 8-bit symbol characters.

Next, when the version and the error correction level are specified by the user, the total number of codewords, the number of error correction codewords and the number of blocks, in which the data codewords will be divided, are determined as defined in Table 9 of the standard [6]. Data codewords are divided into blocks in each of which Reed-Solomon error correction scheme is applied. Using Reed-Solomon encoder, redundancy is added to the encoded information, and error detection and correction is possible when decoding the data codewords.

2.1.4.2 Reed-Solomon data encoder

The Matlab and Java implementations of the Reed-Solomon encoder differ[2]. Indeed, in Matlab, two functions are called for encoding the data. The *gf(x, 8)* function creates a Galois array of 2 to the power of 8 elements, from the data codewords per block. The *rsenc(x, n, k)* function is called for encoding the data codewords per block, using an [n, k] encoder, where n is the number of final codewords per block after encoding, and k, the number of data codewords per block needed to encode.
In Java, the whole Reed-Solomon encoder had to be implemented. The tutorial proposed by Wikiversity [7] was really helpful to understand how to implement it. Before encoding the data, the operations used in the Galois field had to be defined. These operations are based on polynomial definitions. First, a table of the Galois field primitive elements has been created. Then, the division and multiplication operations, based on logarithms, are defined, using the previous table. Several operations on polynomials, which coefficients are Galois field primitive elements, have also been defined, as preliminary to the encoder implementation. The Reed-Solomon encoder uses a generator polynomial, that is dynamically generated, according to the number of error correction symbols (i.e the level of correction for the standard QR version 2). An XOR operation is then performed between the byte array to encode and the generator polynomial, which is equivalent to a long division. The remainder of the division is concatenated with the original byte array. The result forms the encoded data codewords.

The higher the error correction level is chosen, the larger the amount of error correction codewords inside the QR symbol is. Hence, there is a trade-off between the number of data codewords and the error correction capability.

2.1.4.3 Data codewords' construction

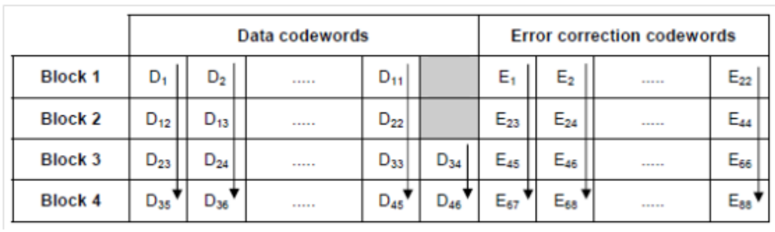
Having obtained the values of all the parameters set according to the standard specifications, the construction of the data codewords can be proceeded. Firstly, the input set of characters is converted into a binary bit stream where each character is represented with an 8-bit sequence (byte mode). Four bits indicating the byte mode, according to Table 2 of the standard[6], are added in the beginning of the data sequence and this prefix is followed by another bit sequence that is called character count indicator representing the number of input data characters in its binary equivalent form. According to Table 3 in the standard [6], this sequence has either a length of 8 bits (for versions lower than 10) or 16 bits (for versions higher than 10). The binary data sequence is placed afterwards,

[2]The encoded messages that result from the Reed-Solomon encoding process are different in Java and Matlab, since the generator polynomials used are different. In spite of that, both standard version 2 QR codes, generated by either Matlab or Java, are compatible with a professional QR reader application.

while a so called "terminator bit sequence" of four zeros is added in the end of this binary sequence.
Hence, the bit stream contains respectively the mode indicator, the character count indicator, the data bit stream and the terminator bit pattern. The resulting stream is divided into 8-bit codewords and, if necessary, is extended with the padding codewords 11101100 and 00010001 which are added alternately in order to fill the data codewords capacity required for the version and error correction level.

The resulting series of codewords, the data codeword sequence, is then processed to add error correction codewords to the message. According to the standard, Reed-Solomon error control coding is used to detect and correct errors. A series of error correction codewords is generated, as explained in the Reed-Solomon section, which are added to the data codeword sequence, in order to enable the symbol to withstand damage without loss of data. There are four user-selectable levels of error correction, as previously mentioned, offering the capability of recovery from different amounts of damage.
More specifically, the codeword sequence is divided into the required number of blocks in order to enable the error correction algorithm to be processed. The number of blocks depends on the choice of the version and error correction level. The error correction codewords are generated applying the error correction algorithm, as analytically described previously, separately for each block, appending the error correction codewords to the end of the data codeword sequence. The data and error correction codewords are then interleaved with a blockwise method and the final message sequence is produced. These data and error correction blocks fill exactly the symbol codeword capacity; however, as it was mentioned previously, if the number of modules available for data and error correction codewords is not an exact multiple of 8, there may be a need for 3, 4 or 7 remainder bits to be added to the final message bit stream in order to fill exactly the number of modules in the encoding region.

Data and error correction codewords are taken from each block in turn and all the data codewords are placed in the symbol before the first error correction codeword. Figure 2.4 [6] illustrates the way the final message sequence is constructed. In this example, the sequence is D_1, D_{12}, D_{23}, D_{35}, D_2, D_{13}, D_{24}, D_{36}, $\ldots$, E_1, E_{23}, E_{45}, E_{67},$\ldots$



	Data codewords					Error correction codewords			
Block 1	D_1	D_2	.....	D_{11}	▨	E_1	E_2	.....	E_{22}
Block 2	D_{12}	D_{13}	.....	D_{22}	▨	E_{23}	E_{24}	.....	E_{44}
Block 3	D_{23}	D_{24}	.....	D_{33}	D_{34}	E_{45}	E_{46}	.....	E_{66}
Block 4	D_{35}	D_{36}	.....	D_{45}	D_{46}	E_{67}	E_{68}	.....	E_{88}

FIGURE 2.4: Division in blocks and interleaving process [6]

2.1.4.4 Data placement in the encoding region

After the construction of the final message sequence, the placement in the encoding region inside the QR symbol follows, beginning from the bottom right corner of the symbol according to a zig-zag algorithm[6]. Based on this, the bits of the message sequence (starting with the most significant bit) are placed in the two module wide columns alternately upwards and downwards, from the right to the left of the symbol. In each column the bits are placed alternately in the right and left modules, moving upwards or downwards according to the direction of placement, changing direction at the top or the bottom of the column. Each bit is placed in the first available module position. Following these principles, figure 2.5 illustrates the QR symbol version 7 and high error correction level after the placement of the message sequence. The characters that are encoded are *"Hello, this is version 7"*.

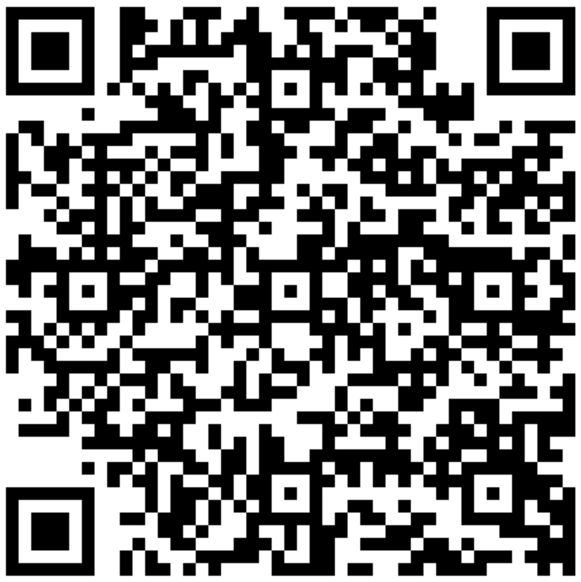


FIGURE 2.5: Data placement inside a QR symbol version 7

2.1.4.5 Data masking

The final procedure before the QR symbol takes its final form refers to the data masking. A particular matrix pattern is applied through the XOR operation in the encoding region (excluding the format information and the version information), in order for dark and light modules to be arranged in a well balanced manner in the symbol. In addition, the module pattern 1011101 particularly found in the finder pattern should be avoided in other areas of the symbol as much as possible. There are eight masking patterns proposed by the standard. Therefore, a three-bit identifier of the data masking pattern applied to the symbol is included in the format information in order the correct mask to be removed in the decoding procedure. Figure 2.6 shows the previous QR symbol after the masking pattern with reference 010 is applied on it.

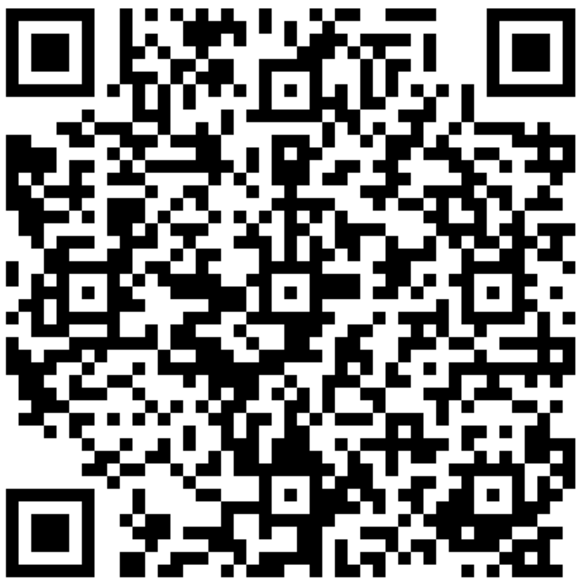


Figure 2.6: Data placement inside a QR symbol version 7 after applying masking

2.1.4.6 Format and version placement

After the placement of the format and version information the QR symbol takes its final form in figure 2.7. The format information is a 15-bit sequence that contains 5 data bits and 10 error correction bits calculated using the (15, 5) BCH code. The first two data bits refer to the error correction level of the symbol and the other three data bits contain the data mask pattern reference for the pattern selected.

2.1.4.7 Reed-Solomon format encoder

In Matlab, two functions are called to encode the format information data bit. The *gf(data bits, 1)* function creates a Galois array of 2, from the data bits. Then, the *bchenc(data bits, n, k)* function is called which encodes the data bits, using an [n, k] encoder, where n is the number of bits after encoding, and k, the number of data bits to encode. In Java, the format information is encoded using BCH codes, which is a generalisation of Reed-Solomon encoding. QR format information encoding uses a fixed generator. The format information is encoded by performing the XOR operation with the generator.

On the other hand, the version information, which is included in symbols with version higher than seven, consists of an 18-bit sequence containing 6 data bits, with 12 error correction bits calculated using the (18, 6) Golay code. The particular values of this field are found in Annex D of the standard [6].

Figure 2.7: Final form of the encoded QR symbol version 7

The format and version information are positioned twice each inside the symbol in order to provide redundancy since its correct decoding is essential to the decoding of the complete symbol.

2.2 Non-standard QR codes

While moving from Matlab implementation towards Android implementation, in order to meet the advanced requirements for the project, we decided to differentiate from the ISO/IEC 18004-2006 standard, and create our own QR symbol that will be used for encoding and decoding the desired message.

2.2.1 Main characteristics

The main characteristics of this new QR symbol are the four finder patterns that are located in the four corners of the symbols and the total lack of the alignment patterns inside the symbol. The four finder patterns are used in order to apply the perspective transformation for the symbol detection while the image is processed, which might mean less data encoded in the QR. However, the drawback of encoding less data is counteracted by the absence of alignment patterns which are of no use since the symbol that is transmitted from the flat device screen is not subject to any kind of bent or curvature.

Furthermore, compared to the QR standard form, this new symbol does not contain any version or format information next to the finder patterns but the quiet zone around them is retained in order to set the boundaries with the data placement.

The size of the QR symbol takes any even values that are larger than 24 and the concept of the version that is related with the size of the symbol is now discarded. The even number of the size results from the fact that the seventh column of the symbol, that previously contained the timing pattern, is now used for data placement and also because the data fulfilment follows the same zig-zag pattern as in the standard.

Moreover, according to the standard, format information contains information about the error correction level and the mask that is used on the data. On the contrary, in our new QR symbol, the error correction level is determined with a procedure that will be explained later and also a fixed masking pattern is applied on the data. The available masking patterns are the same that are used in the standard [6]. The timing patterns are not included either in this new implementation; instead, these areas are fulfilled with data. Figure 2.8 shows the simplified structure of an 80×80 QR symbol that was used in this implementation.

2.2.2 Amount of data encoded

In this implementation, the maximum value of size according to the ISO/IEC 18004-2006 standard (177×177 for version 40) can now be theoretically exceeded and the limitation to this parameter refers to the image processing part with the use of the perspective transformation for a selected resolution set on the phone device. According to the description of the non-standard QR symbol the modules that correspond to the four finder patterns are unavailable for data placement and this fact is indicated by the appropriate modification of the unavailable area for data end error correction codewords

FIGURE 2.8: Function patterns of an 80×80 QR symbol

placement. In order to calculate the number of data codewords that fit inside the QR symbol we follow the following procedure: given that the data is encoded with 8 bits per character, we divide the number of available modules with the number 8 and we set the integer part of this division equal to the total number of codewords in the symbol and the remainder as remainder bits that are unavailable for data placement since no codeword can fit there. The error correction level is determined as follows: we define that the error correction codewords should be approximately 50% of the total number of codewords. Since for the Reed-Solomon encoding the difference between the unencoded and encoded codewords should be an even number, the number of error correction codewords should be even. Therefore, we define four different cases for the total codewords (N), data codewords (D), error correction codewords (E) and possible remainder codewords ($R = N - D - E$), as explained in the algorithm below:

```
% If N is an even number:
if( num_codetotal%2 == 0 ){
    % If N/2 is an even number,
    % then we set D = N/2, E = N/2, R = 0:
            if( (num_codetotal/2)%2 == 0 ){
                    num_codeE = num_codetotal/2;
                    num_codeD = num_codeE;
                    num_codetotal = num_codetotal - 0;
                    num_bitsremain = num_bitsremain + 0*8;
            }
    % If N/2 is an odd number,
    % then we set D = (N/2) - 1, E = (N/2) - 1, R =2:
            else{
                    num_codeE = num_codetotal/2 - 1;
                    num_codeD = num_codetotal/2 - 1;
                    num_codetotal = num_codetotal - 2;
```

```
                        num_bitsremain = num_bitsremain + 2*8;
                }
        }
    % If N is an odd number:
        else{
        % If (N-1)/2 is an even number,
        % then we set D = (N-1)/2, E = (N-1)/2, R = 1.
                if( ((num_codetotal-1)/2)%2 == 0 ){
                        num_codeE = (num_codetotal-1)/2;
                        num_codeD = (num_codetotal-1)/2;
                        num_codetotal = num_codetotal - 1;
                        num_bitsremain = num_bitsremain + 1*8;
                }
        % If (N-1)/2 is an odd number,
        % then we set D = ((N-1)/2) - 1, E = ((N-1)/2) - 1, R = 3.
                else{
                        num_codeE = (num_codetotal-1)/2 - 1;
                        num_codeD = (num_codetotal-1)/2 - 1;
                        num_codetotal = num_codetotal - 3;
                        num_bitsremain = num_bitsremain + 3*8;
                }
        }
```

Therefore we properly modify the unavailable area for data placement in the flag matrix if R is not equal to zero. The four finder patterns together with the separator area result in $8*8*4$ unavailable modules. However, this number corresponds to finder patterns with ratio 1:1:3:1:1 as defined according to the standard. This ratio becomes flexible in our implementation since the size of the finder patterns should be increased when dealing with large sizes of QR symbols in order the finder patterns to be easily detected when applying the perspective transformation. Hence, a parameter n is inserted in order to indicate this size. Table 2.1 shows the values of this parameter.

Finder Pattern Ratio	**Values of parameter n**
1:1:3:1:1	1
3:3:9:3:3	3
5:5:15:5:5	5

Table 2.1: Flexible finder pattern ration in non-standard version

2.2.3 Available area for data placement

Since the size of the QR symbol should be an even number, as mentioned previously, the expression $size^2 - 8 \times 8 \times 4 \times n$ that defines the available area for data placement is also an even number and the remainder of the division of the above expression with eight gives also an even number. Thus, adding to these remainder bits, the R codewords we get the "total remainder area" that is also not used for data placement. This area is placed on the left side of the QR symbol and in particular on the first two columns; however, if this is not enough we use the next two columns or part of them. Figure 2.9 shows the unavailable area for the previously defined QR symbol.

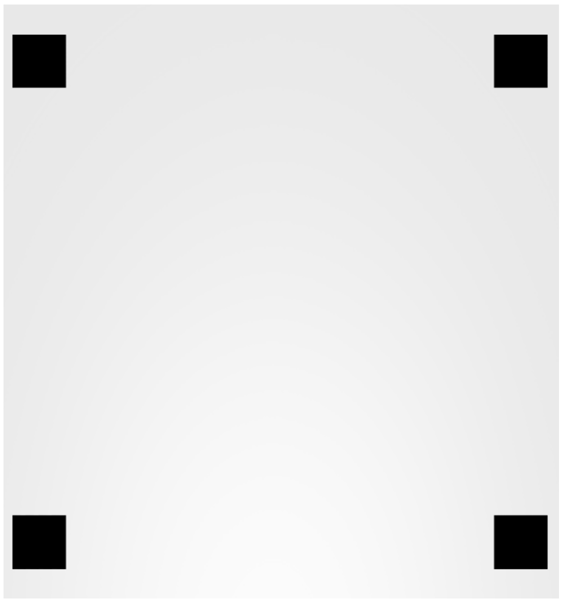


FIGURE 2.9: Unavailable area of an 80×80 QR symbol

2.2.4 Data sequence construction and placement

As far as the data sequence construction is concerned, the first two bytes of the data stream are used for the character count indicator. If the number of data is less than the maximum acceptable number of data codewords, padding codewords are added while the padding process remains the same as in the standard. In addition, we divide the data codewords into blocks of 20. The incentive behind this value was the observation of the amount of data in each block in Table 9 of the standard [6] for all the versions for the highest error correction level. The arrangement of the data codewords into blocks is carried out as in figure 2.4; however, in this non-standard implementation all the blocks, except for the last one, have the same amount of data codewords and the final block has a data length of 20 plus the remainder of the division of the number of data codewords with 20. We then apply Reed-Solomon encoding for each block and interleave the data with the same method used in the standard, constructing the final message sequence. In order to fulfil the symbol with this sequence the same zig-zag algorithm is used as described in the standard. Figure 2.10 illustrates the 80×80 QR symbol after the data placement is accomplished.

By applying the masking pattern with reference 001[6] is applied on the QR symbol, we get the final form of the QR symbol in figure 2.11

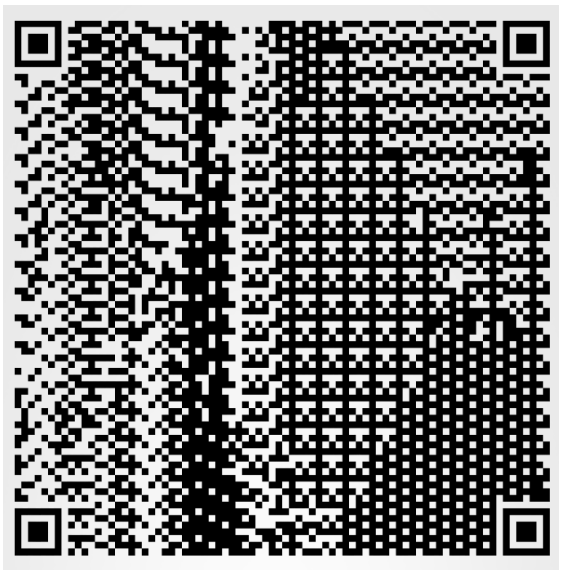


Figure 2.10: Data placement in an 80×80 QR symbol

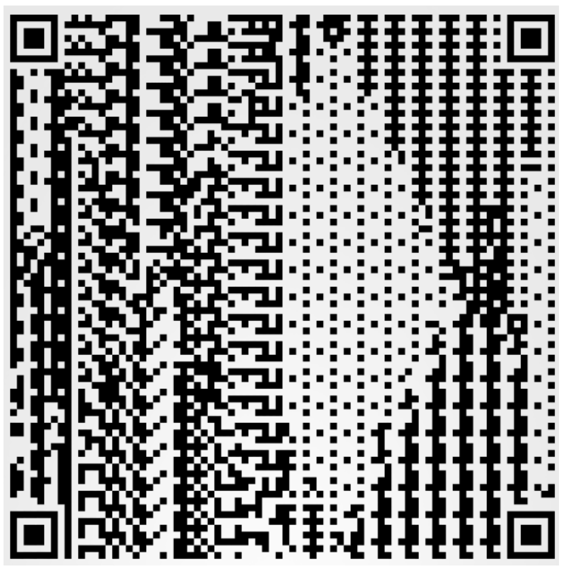


Figure 2.11: The 80×80 QR symbol after applying the mask

2.2.5 Colourful QR codes

Moving away from the 1:1 module-bit representation, we created colour QR codes with 8 different colours. In particular, we used a 1:3 module-bit representation; that means that in each square module of the QR symbol, 3 bits of the message sequence are attached and therefore the overall capacity of the symbol can be three times increased. Table 2.2 indicates the colours used in our implementation and their corresponding bit sequence.

Another difference that occurs is that now the QR matrix is initialised as a byte matrix instead of a boolean matrix used in the black and white representation. Each colour has its own byte value which is represented in every particular module of the symbol.

Bit Sequence	**Color**
000	Black
001	Blue
010	Green
011	Cyan
100	Red
101	Magenta
110	Yellow
111	White

Table 2.2: Bits - Colour representation

These values are assumed to be known in the decoding procedure in order to correctly link colours to bits. Figure 2.12 illustrates an 80×80 colourful QR symbol with finder pattern ratio 1:1:3:1:1.

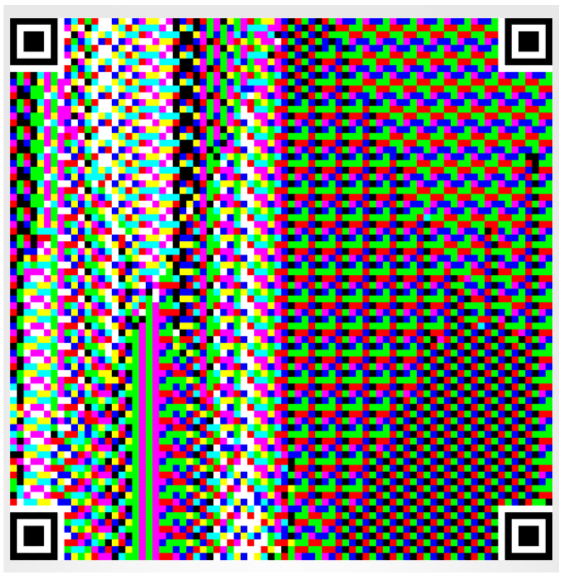


Figure 2.12: The 80×80 colourful QR symbol with finder pattern ratio 1:1:3:1:1

2.2.6 Rectangular QR codes

Using different number of square modules in the horizontal and vertical direction we can obtain rectangular codes. With this way we can exploit the fact of representing the QR symbol using most of the width of the device screen while in the case of squared QR symbols that was not possible. The restriction regarding the even number of size in both directions remains. Figure 2.13 illustrates an 80×120 rectangular QR symbol with finder pattern ratio 3:3:9:3:3.

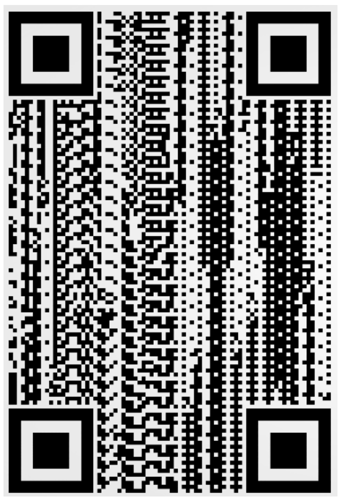


FIGURE 2.13: The 80 × 120 rectangular QR symbol with finder pattern ratio 3:3:9:3:3

2.2.7 Colourful rectangular QR codes

Combining the concepts of a colourful and a rectangular QR code, towards the aim of maximising the amount of data inside the symbol, the result is illustrated in figure 2.14 where a colourful rectangular 30 × 90 QR symbol is depicted with finder pattern ratio 1:1:3:1:1.

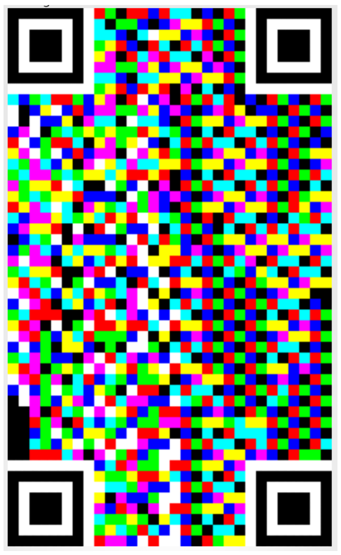


FIGURE 2.14: The 30 × 90 colourful QR symbol with finder pattern ratio 1:1:3:1:1

Chapter 3

Image Processing

In this chapter the image processing needed to detect and read QR codes is discussed. For this purpose, OpenCV is used, for the Java implementation, which is a powerful computer vision libary. It is freely available under the open source BSD license.

3.1 Preprocessing

3.1.1 Filtering

For the blurring, a median filter is used. The median filter runs through each element of the image and replaces each pixel with the median of its neighbouring pixels. In the OpenCV implementation of this filter, neighbourhood is defined as a square around the evaluated pixel.

The code in OpenCV is as follows, featuring the source image (*src*), destination image (dst) and the linear aperture size (*ksize*):

```
Imgproc.medianBlur(src, dst, ksize);
```

3.1.2 Thresholding

For the thresholding, an adaptive threshold is used, to compensate changes in brightness over the image. To implement it, the adaptiveThreshold function is called in OpenCV. Therefore, the average brightness of the neighbouring pixels is used to determine a good threshold.

The code called in OpenCV is as follows:

```
Imgproc.adaptiveThreshold(src, dst, maxValue, adaptiveMethod,
                          thresholdType, blockSize, C);
```

The parameters are defined as follows:

- **src** is the gray scale source image,
- **dst** is where the binarized image will be written to,
- **maxValue** is the value assigned to pixels that meet the condition,
- **adaptiveMethod** is the adaptive thresholding algorithm to be used. In our case a simple mean over all the neighbouring pixels (Imgproc.ADAPTIVE_THRESH_MEAN_C),
- **thresholdType** is the thresholding type to be used; in our case Imgproc.THRESH_BINARY,
- **blockSize** is the size of a pixel neighbourhood used to calculate the threshold (we use around 1/4 of the image size),
- **C** is the constant subtracted from the mean or weighted mean; in our case 0.

3.2 Finding the Finder Patterns

After adaptive thresholding, the next task is to detect the QR code from the image. For this, we need to locate where the QR code is in the image. This is done by using one of the marked features of the QR code- finder patterns. According to the international standard of QR code encoding [?], the ratio between the black and white modules in the finder patterns is 1:1:3:1:1. Figure 3.1 shows an example of a QR pattern. The QR code are extracted from the image using the finder patterns, and the extraction procedure encompasses the following steps:

First Scan pixels of each row of the image (processing time can be decreased by scanning every n^{th} row instead of each row) and record the length of the black and white modules. Figure 3.1 shows the row scanning.

The pseudo code to calculate the length of the modules of a vector is given below:

```
For i=1:length_Vector-1
    If Vector(i)==Vector(i+1)
        length_modules(k)+=1;
    Else
        k=k+1;
    End
End
```

Second Take every consecutive 5 elements of this length recording of the row and divide it by the sum of these 5 elements to calculate the ratio.

The pseudo code for computing the ratio is given below:

```
For i=1:length_Modules-4
    Refer = length_modules(i:i+4);
    Ratio = Refer/sum(Refer);
End
```

Third Check whether the ratio is equal to 1:1:3:1:1 and also check whether 1st element in the 5 elements is the length recordings of a black module as finder patterns always start with black module.

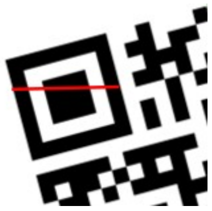


Figure 3.1: Row Scanning

Fourth If the ratio is found, the center position of the middle black block is found and the length of the black and white modules in that column is recorded. Figure 3.2 shows the column scanning.Repeat the previous 2 steps for finding the ratio and checking for the finder patterns.

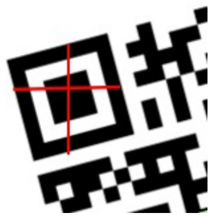


Figure 3.2: Column Scanning

Fifth Again find the center position of the middle black block in the column and check whether this coincides with the row. If it coincides, that is the center of the finder patterns. If it doesn't coincide, check the next row until the center is found as shown in Figure 3.3. Repeat the steps for the entire image.

Sixth The algorithm may find centers that are not those of the finder patterns. In order to eliminate those, the pattern size can be used. Pattern size is the sum of the length of black and white modules in the finder pattern either along the row or column. The pattern size of the finder patterns can be assumed to be larger than all the wrong patterns with ratio 1:1:3:1:1. Also we know that only three finder

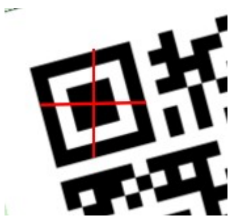
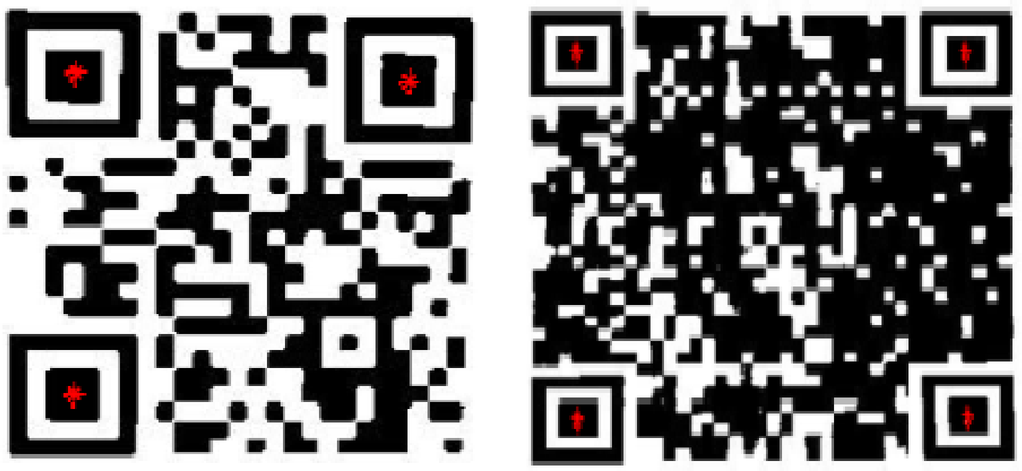


FIGURE 3.3: Adjusting to center position



(a) Standard QR Code (b) Non-standard QR Code

FIGURE 3.4: Finder Pattern Centers

patterns are there (4 finder patterns for non-standard) and the algorithm can be adjusted accordingly. An example showing the detected centers in the standard and non-standard QR codes is shown in Figure 3.4.

Finding the ratio in the finder pattern algorithm is the function that takes up the longest processing time as for each row, each pixel has to be checked with the previous pixel to compute the length of the modules. The algorithm have been tested to make it faster by checking every 10^{th} row instead of each row and instead of checking the entire column, only a portion around the center position of the row have been checked to compute the center of the finder patterns.

The pseudo code for finding the finder patterns is given below:

```
Thresholded Image
For i=1:num_rows
    Center_row=Ratio(Image(i,:));
    Center_column=Ratio(Image(:,Center_row));
    If Center_column==i && Image(i,start)==0
        Center = [i , Center_row];
    End
End
```

3.3 Geometric Rectification

Geometric rectification is needed, since the image of the QR code may be distorted (i.e. tilted with an angle). The image must be transformed or corrected to a square, as shown in Figure 3.6. To complete geometric rectification, the finder pattern centers can be used to correct the image and extract the bits. Three different approaches have been tried. The first approach was a mathematical one where the angle θ at which the QR code is tilted to correct the distortions are used, as shown in Figure 3.5. This was implemented only in Matlab, and works for all versions given that the finder pattern centers are found.

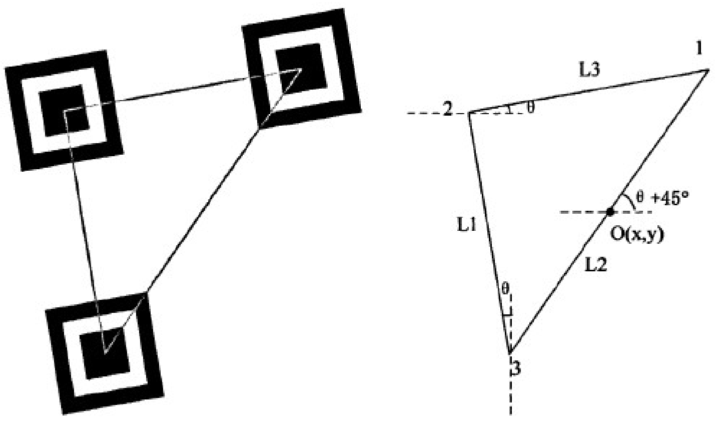


FIGURE 3.5: Geometric Correction using the tilted angle[8]

The other two approaches are based on transformations. In the standard QR code version, comprising three finder patterns, affine transformation is implemented to correct the image and then find the alignment pattern center in the QR code, as there might be some geometric distortions. This was implemented in both Matlab and Java for standard Version 2 QR codes only. For the non-standard QR codes, the perspective transformation is used for geometric corrections. With the four finder pattern centers, we have used perspective transformation since the latter is able to correct more distortions than affine transformations. The method used to implement these transformations on the distorted image to extract the bits is explained briefly in the following sections.

3.3.1 Affine Transformation

An affine transformation is a transformation which preserves straight lines and ratios of distances between points lying on a straight line. After the transformation, the sets of parallel lines will remain parallel to each other, straight lines remain straight[9].

Assuming that there are two affine spaces which are the spaces before and after the transformation, an affine map is built to describe the relation between the spaces. An affine map is the composition of two functions: a translation and a linear map. In ordinary vector algebra, matrix multiplication is used to represent linear maps and vector addition is used to represent translations.

$$\overrightarrow{y} = f(\overrightarrow{x}) = A\overrightarrow{x} + \overrightarrow{b}$$

As the affine map f above, where $\overrightarrow{y}$ and $\overrightarrow{x}$ are two affine spaces, the multiplication matrix A represents the linear map and the addition vector $\overrightarrow{b}$ represents the translation. All the points in one affine space can be computed by using the affine map f and the corresponding points in the other affine space.

In order to find the multiplication matrix A and the addition vector $\overrightarrow{b}$, an augmented matrix, which is called the transformation matrix, is built to simplify the computation.

$$\begin{bmatrix} \overrightarrow{y} \\ 1 \end{bmatrix} = \begin{bmatrix} A & \overrightarrow{b} \\ 0, ..., 0 & 1 \end{bmatrix} \begin{bmatrix} \overrightarrow{x} \\ 1 \end{bmatrix}$$

There are six unknown elements in the transformation matrix, because A is a 2-by-2 matrix and $\overrightarrow{b}$ is a 2-by-1 matrix. At least 3 control-point pairs are needed to obtain a solution for the 6 unknown coefficients.

In our case, the $\overrightarrow{x}$ space is the original picture, and $\overrightarrow{y}$ is the space needed after the transformation. All the values of the points in $\overrightarrow{y}$ can be extracted from $\overrightarrow{x}$ based on the transformation matrix. In order to compute this matrix, three control-point pairs are needed in case of affine transformation. And the control-points in the $\overrightarrow{x}$ space are the center-points of the finder patterns, recovered from the detection part, which are the input_points matrix and the corresponding center-points in the $\overrightarrow{y}$ space is the base_points in the following Matlab code. For version 2 QR code, $\overrightarrow{y}$ should be a 25*25 matrix and the position of the three points in the $\overrightarrow{y}$ space are (4,4), (4,22) and (22,4) which are the base_points matrix in the code.

The code to do affine transformation in Matlab is given below:

```
tform=cp2tform(input_points,base_points,'affine');
result=imtransform(image,tform);
```

With OpenCV the same can be done on the Phone using the following piece of code:

```
afTrans = Imgproc.getAffineTransform(input_points, base_points);
Mat result = new Mat();
Size size = new Size(290, 290);
Imgproc.warpAffine(image, result, afTrans, size);
```

In Figure 3.6, one can notice that the result after the transformation is not good enough, especially in the lower left of the code since there is not any control-point in that area.

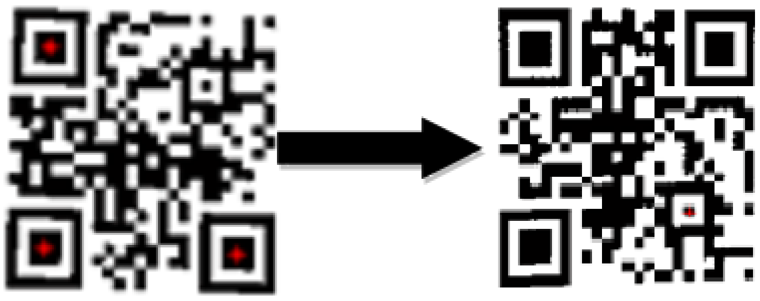


FIGURE 3.6: Affine Transformation

In order to extract the bits from the code, the picture needs to be corrected. This can be done by using the alignment pattern. The method used to find the alignment pattern is similar to the one for finder patterns, except that the ratio of the alignment pattern is 1:1:1:1:1 instead of 1:1:3:1:1, and the search window for the patterns can be localised as the approximate location of the alignment pattern is known, after the affine transformation using the three control-points. In Figure 3.6, the centre of the alignment pattern is marked, which is found by scanning the QR code obtained after the affine transformation.

After detecting the alignment pattern, the bits can be extracted based on the position of the alignment pattern and the finder patterns which are used as the reference points. Since the finder pattern sizes are also recorded, the width of each module can be computed. The code is divided into four parts, and the bits are extracted adaptively based on the different width of each module and the position of the reference point in every area as shown in Figure 3.7.

3.3.2 Perspective Transformation

The perspective transformation describes the changes in a perspective projection, when the point of view of the observer changes. The incidence and the cross-ratio of the image are preserved in the transformation [10].

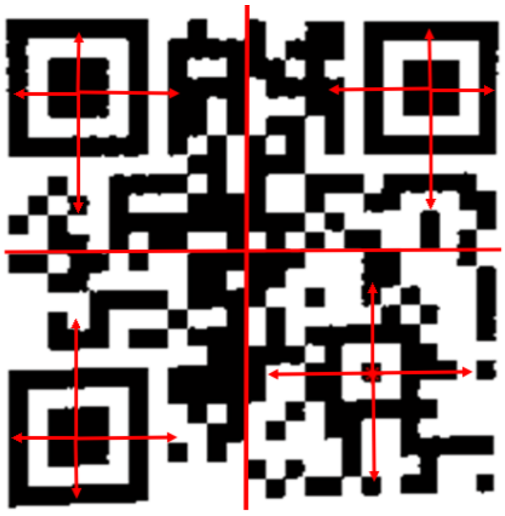


FIGURE 3.7: Extraction of bits after the Affine Transformation

After the transformation, straight lines remain straight, but not parallel lines. In this case, affine transformations are a subset of perspective transformations, which is shown in the Figure 3.8.

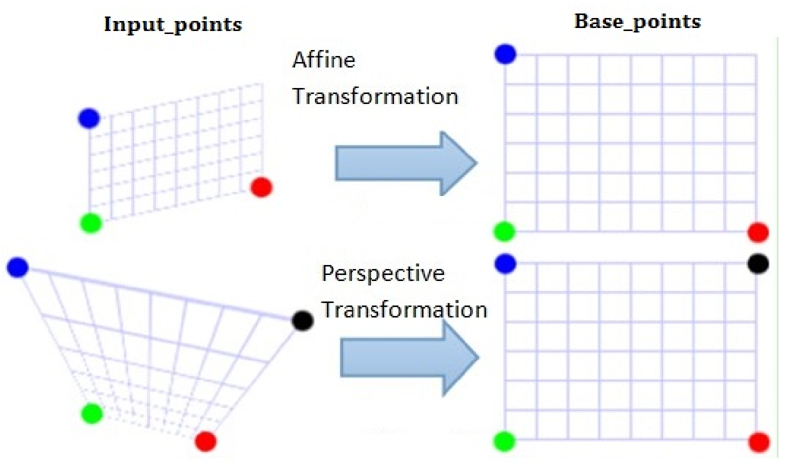


FIGURE 3.8: Geometric Correction using the transformations

$$p_a = \begin{bmatrix} \mathrm{x}_a \\ \mathrm{y}_a \\ 1 \end{bmatrix}; p_b\text{'} = \begin{bmatrix} w'\mathrm{x}_b \\ w'\mathrm{y}_b \\ w' \end{bmatrix}; H_{ab} = \begin{bmatrix} \mathrm{h}_{11} & \mathrm{h}_{12} & \mathrm{h}_{13} \\ \mathrm{h}_{21} & \mathrm{h}_{22} & \mathrm{h}_{23} \\ \mathrm{h}_{31} & \mathrm{h}_{32} & \mathrm{h}_{33} \end{bmatrix}; p_b\text{'} = H_{ab}p_a$$

Similar to the affine transformation, the p_a represents the space before the transformation and p_b represents the one after it and H is the transformation matrix. There are nine unknown elements in this case, and at least 4 control-point pairs are needed to solve them. The four center-points of the finder patterns are the 4 control-points. In the following Matlab code, the input_points matrix is the position of the four center-points of the original picture of the non-standard QR code, and the base_points are the position of them in the space after the transformation. For version 2, the base-points are (4,4), (4,22), (22,4) and (22,22).

The code to do perspective transformation in Matlab is given below:

```
tform=cp2tform(input_points,base_points,'projective');
result=imtransform(image,tform);
```

With OpenCV the same can be done on the Phone using:

```
pTrans = Imgproc.getPerspectiveTransform(input_points, base_points);
Mat result = new Mat();
Size size = new Size(MODULESH*STEP, MODULESV*STEP); //Size depending on QR version
Imgproc.warpPerspective(image, result, pTrans, size);
```

In Figure 3.9, one can see that the result after the perspective transformation is much better than the one after the affine transformation since there are more control-points in this transformation, and no correction is needed. The bits can be extracted directly from the result of the perspective transformation and the process is much faster, which are some of the advantages of using the non-standard QR codes over the standard QR codes.

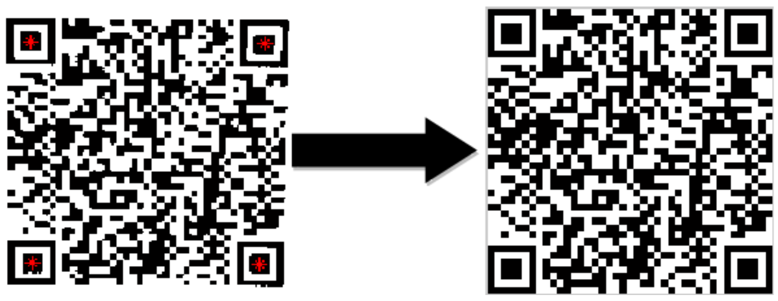
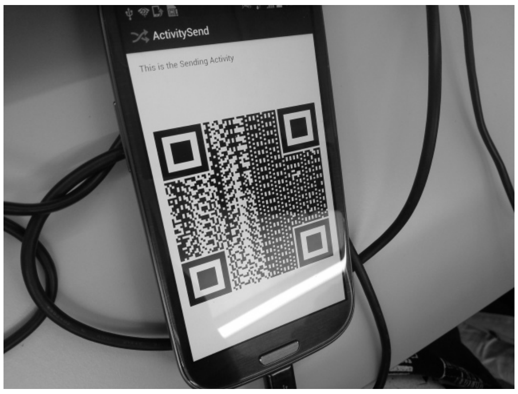


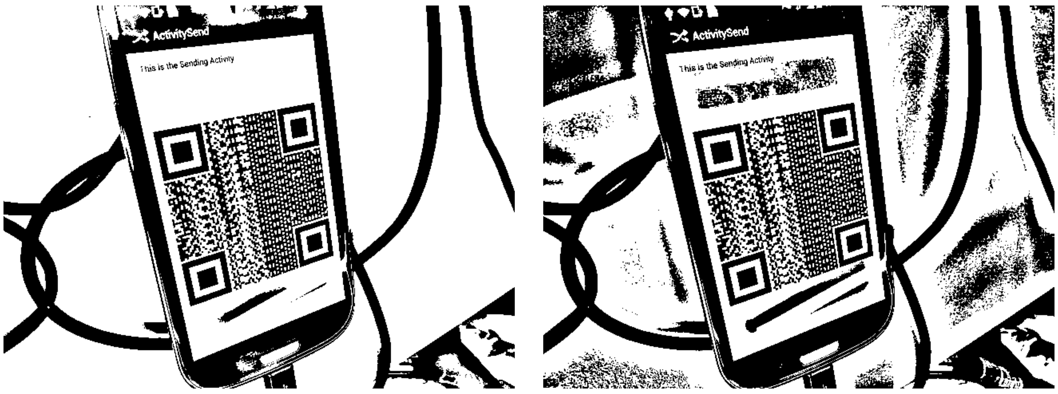
FIGURE 3.9: Perspective Transformation

3.4 Example from Phone

In this section a visual impression of the image processing done on the phones is presented. In Figure 3.10(a), the image recorded by the camera is shown. First the preprocessing is run (section 3.1), which includes noise reduction by blurring the image and applying an adaptive threshold. The result can be seen in Figure 3.10(b). Here the effect of applying a slight offset to the threshold can be seen on the surrounding table when comparing with Figure 3.10(c) where the offset is set to 0. The completely white areas



(a) Camera Image



(b) Thresholding (c) Thresholding No Offset

FIGURE 3.10: Preprocessing

are now noisy. This is bad for two reasons. Those areas contain a lot of state changes (black/white). This slows down our algorithm to detect the Finder Patterns as described in section 3.2. Those areas might also by chance contain the ratio of a finder pattern. This causes problems during further processing because we need to sort and assign the patterns before we do the Affine or Perspective transformation. See section 3.3.

In Figure 3.11, the image processing after detecting the QR code is shown. In Figure 3.11(a), it becomes clear why a simple affine transformation does not allow us to retrieve the data. It only achieves reconstruction along the lines between the finder patterns used. The perspective transformation in Figure 3.11(b) results in an undistorted QR code. Finally we apply another adaptive threshold before reading out the module values from Figure 3.11(c).

3.5 Color

To increase the transmission rate, the data encoded in one QR code can be increased. This can be done by implementing colors in the QR code which can be done in various

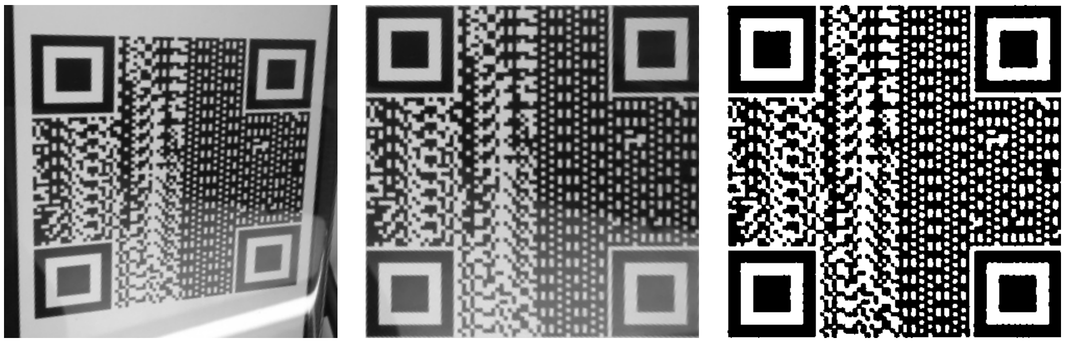

(a) Affine (b) Perspective (c) Binarization

FIGURE 3.11: Image Transformation

ways. There are different types of color models available for this purpose like RGB and HSV color model. The 2, 4, 8 and 16 colors have been implemented in both of these models, in Matlab, and 8 colors in Java. The data transmission rate can be increased n times depending on the number of colors used for encoding which is 2^n. Therefore, for 8 colors the data encoded in a single QR code can be increased up to 3 times that of a black and white QR code. The salient features of the QR code comprising the format information, version information, finder patterns, do not have any color information in them. The models are briefly described in the following sections.

3.5.1 RGB color

In this color model, the primary colors -Red, Green and Blue- are added in various proportions to produce various colors. To form a color, the R, G and B channel are superimposed with an arbitrary intensity which varies from fully off to fully on. Zero intensity in all the channels give a black color and full intensity in all gives white. An 8 color system is shown in the Figure 3.12. The codewords corresponding to the 8 colors is given in the Table 3.1.

Codewords	**R**	**G**	**B**
⬛ 000	0	0	0
🟦 001	0	0	1
🟩 010	0	1	0
■ 011	0	1	1
🟥 100	1	0	0
🟪 101	1	0	1
🟨 110	1	1	0
111	1	1	1

TABLE 3.1: Codewords: 8 RGB color system

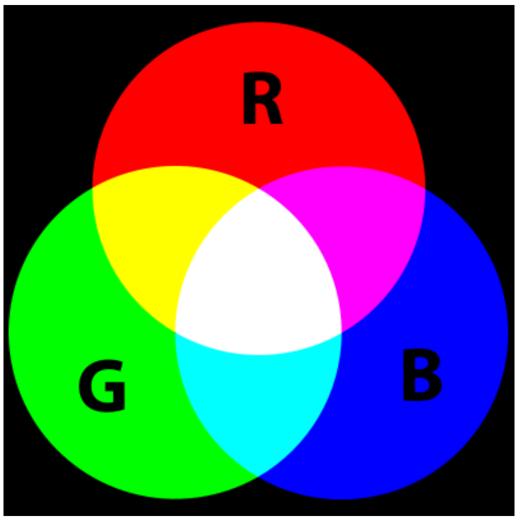


Figure 3.12: 8 RGB color system[11]

Encoding the QR code with these values is simple, but extracting the color QR code from the image with less errors is the difficult part. All the color models are device dependent -the RGB values are detected by different devices differently. For an 8 color system, an adaptive thresholding can be applied to each channel to binarise all the three channels and then superimpose them together.

For a 16 colors system, binarisation of the channels is not possible as one more level is added to the intensities: the grey level (0.5). A possible color system for 4 bits codeword is shown in the Table 3.2. This color system was developed after many testing based on the capability to differentiate between the colors. The gray code system is implemented to reduce the bit errors, i.e. for every 0.5 level difference in a channel, the codewords have a 1 bit difference. To decode the features of the QR code that contains no color information like the version, format, finder patterns, etc; in one channel -Green (comparatively RB channel provided more distinct colors than RG and GB)- only 2 levels were assigned so that binarisation can be done on the green channel, by using adaptive thresholding and the features extracted without any errors.

Another problem while implementing the 16 colors was that the colors were wrongly decoded which increases the errors. This may be because of a constant offset as the RGB color is device dependent or because of too much noise in the image. One method for decreasing the error was to use local thresholding with different reference points. The extracted QR code from perspective transformation is divided into 4 and thresholding is applied in each area based on the mean of the RGB values of the white modules in the finder patterns in that area. But as the version increases, there are more errors. So, we used more reference points; 2-by-2 white modules in the place of alignment patterns and the mean value of these are taken as the reference. For version 10, the extracted

Codewords	**R**	**G**	**B**
🟨 0000	1	1	0
🟥 0001	1	0	0.5
🟫 0010	0.5	0	0
🟥 0011	1	0	0
🟦 0100	0	0	0.5
🟩 0101	0	1	0
⬛ 0110	0	0	0
🟩 0111	0.5	1	0
⬜ 1000	1	1	1
🟨 1001	1	1	0.5
🟪 1010	0.5	0	1
🟪 1011	1	0	1
🟥 1100	1	0	0.5
🟦 1101	0	1	1
🟦 1110	0	0	1
🟦 1111	0.5	1	1

Table 3.2: Codewords: 16 RGB color system

QR code is divided into 9 parts and the 5 reference points excluding the white modules of the finder patterns used for thresholding are marked in Figure 3.13.

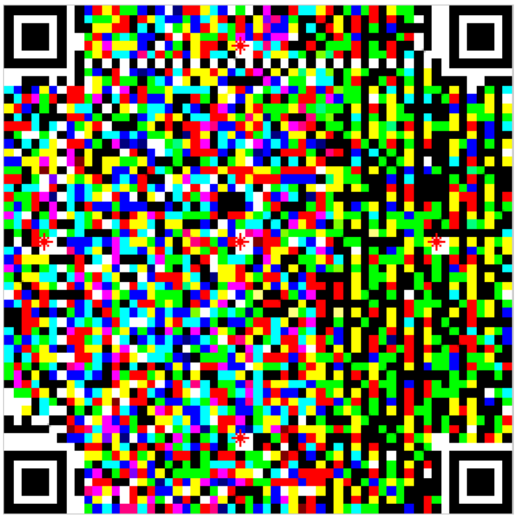


Figure 3.13: 16 colors with 5 reference points marked

Another improvement for color QR code decoding was to increase the scale of the perspective transformation. For the black and white non-standard QR codes, the extraction of bits from the perspective transformation has a many-to-one relation in which the n modules are transformed into 1 module. This may not work well with colors as the color may be wrongly mapped. So instead of transforming from many modules to 1, we tried different scales and taking the mean of them to produce a better result. The perspective transformation with a scale of 7 is shown in Figure 3.14. After this, the mean of each of these 7x7 modules are taken to extract the bits and then they are thresholded into its proper color components to get the QR bits shown in Figure 3.13.

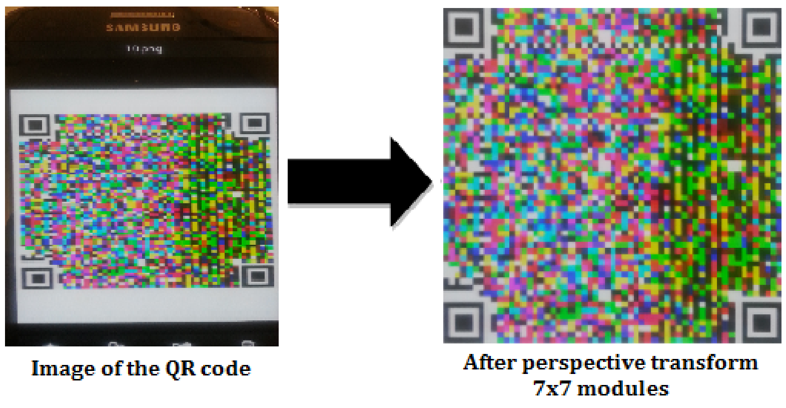


FIGURE 3.14: Perspective Transformation with a scale of 7

3.5.2 HSV color

Another color model is the HSV color model. This is a different representation of points in RGB color model. One of the problem faced by using RGB in higher versions of QR code was the variation in brightness along the image. This adversely affected the thresholding as the part where the brightness varies rapidly are more prone to errors due to the wrong thresholding. The use of reference points described in the RGB section is one of the solutions tested. Another solution was the implementation of HSV color model instead of the RGB color model.

HSV stands for *hue*, *saturation* and *value*. *Hue* is the attribute that corresponds to the colors or a combination of two colors and the values can vary from 0 to 360. The hue value 0 and 360 corresponds to the same color. *Saturation* is the colorfulness of a color relative to its brightness and *value* is the brightness value of a hue color. The hsv attributes are shown in Figure 3.15. For white color, the hsv value is 001 and 000 for black.

The hsv color model was implemented for 2, 4, 8 and 16 colors QR code. The v of hsv color is used for encoding the features of QR code like the format, version information, finder patterns, etc. The problem with the brightness variation is taken care of by not putting any information in the v channel of hsv for the data codewords. The codewords are encoded using h values alone.

A possible 16 colors hsv system with a step of 20 is given in the Table 3.3. The QR code corresponding to the system is given in Figure 3.16(a) . It can be seen that the colors

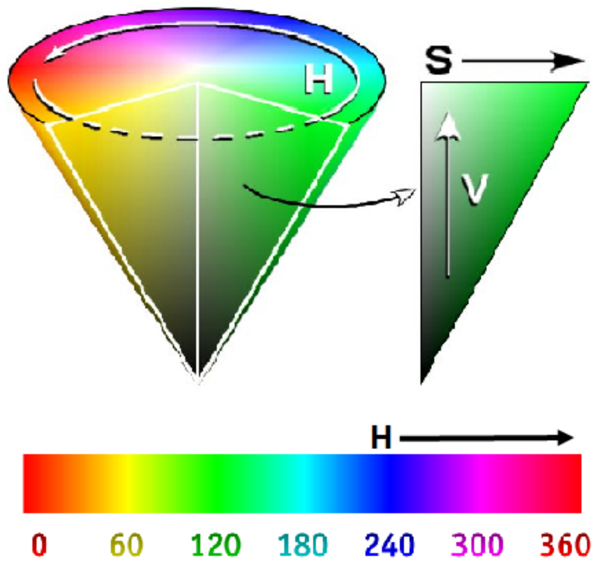


Figure 3.15: Attributes of HSV color model

is not that distinctive and this leads to the receiver perceiving different color from that of the encoded one as the color system are device dependent. Also, thresholding the h channel with a step of 20 is not that effective as some colors may be wrongly thresholded. Binarisation is done on the other two channels, s and v, which is quite reliable and easy. The thresholding in the h channel is the channel which is prone to errors.

Codewords	H	S	V
🟥 0000	0	1	1
🟧 0001	20	1	1
🟧 0010	40	1	1
🟨 0011	60	1	1
🟩 0100	80	1	1
🟩 0101	100	1	1
🟩 0110	120	1	1
🟩 0111	140	1	1
🟩 1000	160	1	1
🟦 1001	180	1	1
🟦 1010	200	1	1
🟦 1011	220	1	1
🟦 1100	240	1	1
🟪 1101	260	1	1
🟪 1110	280	1	1
🟪 1111	300	1	1

Table 3.3: Codewords: 16 HSV color system

In order to remove the problem of small steps, i.e less distinctive colors; using bigger steps with different values in the s and v channels has been tried. For 16 colors, the

h channel is divided into 8 providing higher step size and adding 2 levels in s and v instead of one for encoding the data codewords. Different configuration for the system were checked. Refer to Appendix B to view 16 HSV color system with 2 level s and v values. The corresponding QR codes are given in Figure 3.16(b) and Figure 3.16(c) respectively. But encoding information in the s and v channels mean binarisation cannot be done in these channels and this gives more errors as we go to higher versions. Also in Figure 3.16(b), the colors are too light and similar making it difficult to distinguish. The implementation of 2 level in the v channel makes the colors more distinctive but now separate thresholding must be done in both h and v and an optimised solution for thresholding has not yet been achieved.

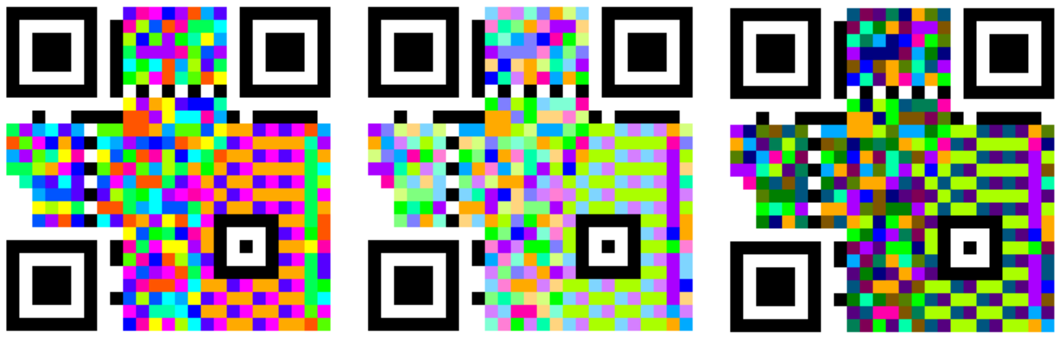

(a) Varying Hue (b) Varying Hue & Saturation (c) Varying Hue & Value

FIGURE 3.16: 16 HSV Color QR Codes

Chapter 4

Decoding QR Codes

This chapter explains the way the data is decoded from the bit streams in order to recover the original message and generate the output file. First, the standard QR version's decoding is presented, followed by the non-standard QR versions' decoding.

4.1 Standard QR codes

The decoding process of the received QR symbol follows the reverse steps of the encoding procedure in order to identify the encrypted data characters.

4.1.1 QR symbol version

More specifically, after the image processing part, the version of the QR symbol is retrieved, according to Table 1 of the standard [6], from the QR symbol's size, which represents the number of modules in the horizontal and vertical direction. This value is compared with the value retrieved from the version information field for versions higher than 7. By then, the identification of the version of the symbol, the function patterns, the encoding region and its locations inside the QR symbol, are known.

4.1.2 Reed-Solomon format information decoding

The next step refers to the reading of the format information which is placed in two different locations inside the symbol. By applying the BCH decoding, we get the two bits that refer to the error correction level and the three bits of the data masking pattern reference. In Matlab, the functions *gf()* and *bchdec()* are called, whereas in Java, the entire decoder had to be implemented.

To decode the format information, an XOR operation is performed between the encoded format and the polynomial generator. If the remainder is zero, then there are no errors in the format information. Otherwise, the format is damaged. In the latter case, a mechanism is used to determine which format was the most likely to have been the original one. This mechanism uses the Hamming distance. Since there are only 32 possible format codes, each of these 32 potential format are compared, in terms of bits-differences (the Hamming metric), with the received damaged format. The one with the least differences is the format that had been transmitted. If several (two or more) possible format are equally distant to the received format, then the format cannot be unambiguously corrected and an error message is generated. The whole decoding process is hampered in that situation, since the format contains essential information for decoding the rest of the encoded data.

4.1.3 Error correction level and masking pattern

After this step, the error correction level and the data masking pattern that was used in the encoding procedure are revealed. Since now the version and the error correction level are known, the number of total codewords, the number of error correction codewords and the number of blocks are determined with the use of Table 9 of the standard [6].

Then, we release the data masking by XORing the encoding region bit pattern with the data mask pattern the reference of which has been extracted from the format information. Using the same principles and rules for the placement of the data and error correction codewords inside the symbol, the interleaved message sequence is extracted.

4.1.4 Reed-Solomon data decoder

The coherence of the codewords is restored by removing the existing interleaving and the codewords are divided into blocks to prepare for the Reed-Solomon error correction scheme to be applied[1]. During this procedure, possible errors are detected using the error correction codewords corresponding to the level used and if any error is detected, it is corrected.

The data codewords decoding process encompasses several steps. First, a syndrome is calculated. If this syndrome is equal to zero, there are no errors in the data codewords, otherwise, the syndrome contains the information necessary to locate the errors. If the

[1]For the Reed-Solomon encoder, as explained in chapter 2, the Matlab and Java implementation differ.

data appears to be damaged, the second step is to locate the errors. The Berlekampâ-Massey algorithm is used; it gives the position of the errors in the byte array representing the message to correct. Then, the errors can be corrected, using the Forney algorithm, which calculates the correction values. Finally, the message is corrected. In some cases, the number of errors detected exceeds the number of errors that can be corrected according to the error level. In that case, an error message is generated and the data cannot be corrected.

4.1.5 Message reconstruction

The corrected data codewords which are now placed in the right order are converted in a binary bit stream, the first four bits of which indicate the byte mode that is used to represent the codewords in bit sequence. The next 8 or 16 bits (depends on whether the version used is lower or higher than ten) of the binary bit stream are the character count indicator which provides information about the length of input data characters. According to the value of this field, we identify the part of the binary stream that refers to the data characters and omit the part that refers to the terminator sequence and the padding codewords. The last part of the decoding procedure consists of splitting the real data stream into 8-bit codewords and converting them to characters.

The block diagram in figure 4.1 summarises the decoding steps[6].

After applying the decoding process, figure 4.2 illustrates the output of a QR symbol version 35 with high error correction level.

4.2 Non-standard QR codes

The decoding procedure for the non-standard QR codes follows the same logic as in the case of QR codes which are created according to the standard.

In particular, after the identification of the QR symbol through the image processing part, we get the size of the symbol in both horizontal and vertical directions. These values represent the number of square modules used and are not equal in the case of rectangular QR codes. Having these values together with the ratio used in the four finder patterns which is assumed that it is known in the receiver, we identify the size of the finder patterns and the encoding region where data and error correction codewords are located.

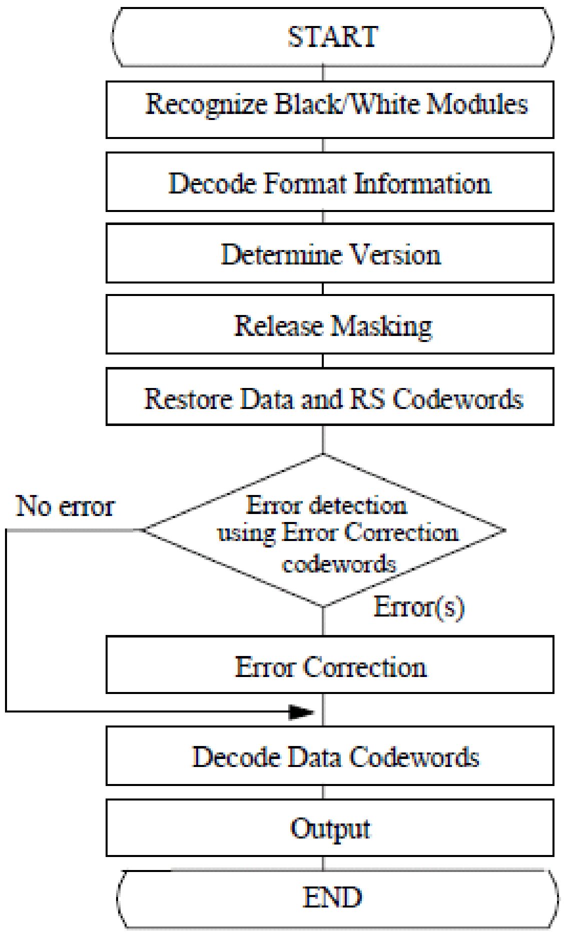
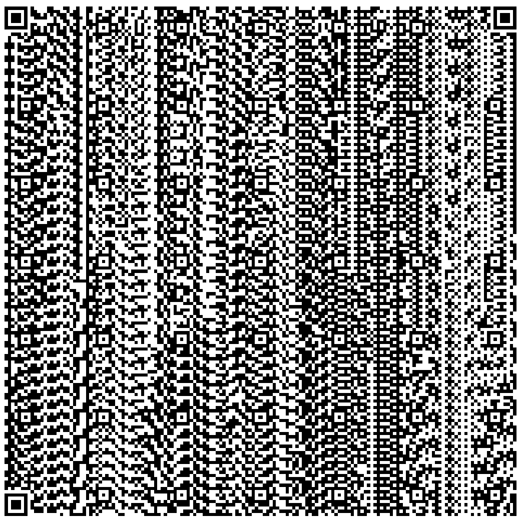
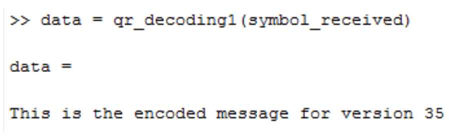

START
Recognize Black/White Modules
Decode Format Information
Determine Version
Release Masking
Restore Data and RS Codewords
No error
Error detection using Error Correction codewords
Error(s)
Error Correction
Decode Data Codewords
Output
END

FIGURE 4.1: QR symbol decoding steps [6]

4.2.1 Mask release and decoding region identification

The first step consists of releasing the data masking by XORing the encoding region bit pattern with the data mask pattern the reference of which is assumed known in the receiver. According to the analysis in the encoding part of the non-standard QR codes and given that the codewords consist of 8 bits, we get the total number of codewords that exist inside the symbol and also the remainder area where a codeword cannot fit. Depending on the number of total codewords and assuming that the error correction codewords should be approximately 50% of the total number of codewords, we identify the number of data codewords, the number of error correction codewords and possible remainder codewords where no data or error correction codewords are placed, following the same procedure as described in the encoding part. This extended now remainder area is defined together with the finder patterns as unavailable and is skipped while running through the QR symbol to retrieve the data and error correction codewords.





```
>> data = qr_decoding1(symbol_received)

data =

This is the encoded message for version 35
```

FIGURE 4.2: Decoding message from a QR symbol version 35

4.2.2 Data and error correction extraction

This is exactly what constitutes the next step of the decoding procedure where using the same principles and rules for the placement of the data and error correction codewords inside the symbol, the interleaved message sequence is extracted. Starting from the bottom right corner of the symbol and skipping the unavailable areas, we continue scanning it until we get the total bit sequence that corresponds to the total number of codewords that we already know from the previous step. In case of colour QR codes, assuming that the 3-bit to colour representation used while encoding is known in the receiver, we translate the colour value in each particular module into 3 bits while retrieving the final bit sequence.

4.2.3 Interleaving release and Reed-Solomon decoder

After the completion of this task, our goal is now to release the interleaving from the bit sequence and divide the codewords into blocks in order to apply the Reed-Solomon error correction scheme. The number of the blocks is determined by the already known number of data codewords, assuming that each block, except for the last one, should contain 20 data codewords. The last block has a data length of 20 plus the remainder of the division of the number of data codewords with 20. For each block, the Reed-Solomon error correction scheme is applied and corrected data codewords are obtained in the output of this procedure.

4.2.4 Message reconstruction

Furthermore, by checking the first two bytes of codewords sequence, which represent the character count indicator, we retrieve the number of the input data characters existing in the sequence and skip possible padding codewords while representing the data characters in the output.

Chapter 5

Android Prototype

This section describes the Android application that has been developed. An overview of the application's user interface and the framework for Java development is given in the first two sections. The third section explains the Java implementation, giving details about the different classes and functions, and their interactions.

5.1 User Interface

Appendix A features four pictures of the User Interface (UI).

Figure A.1 shows the UI's main window, where the user can either choose to transmit ("send" button) or receive ("receive" button) a file, or a message. Pressing "send" in the main UI window starts the transmission with the parameters set in the settings. Pressing "receive" launches the camera and displays the capture window. The decoding starts after pressing "start" in the capture window. When pressing the left-hand-side bottom button on the Android smartphone, a selection window is displayed. The "Settings" button allows the user to access the settings window. The "Read File" and "Write File" buttons enable the user to browse the file they want to transmit or receive, on the SD card. The file browser is shown in figure A.2.

Figure A.3 shows the settings window. Several parameters can be chosen by the user. The "debug mode" is made for the developer to display the chosen settings on the Log console, when testing the application. The "Multi Frame mode" specifies that a file, contrary to a single message, is to be transmitted or received. If the "Multi Frame" button is not ticked, a "String for Encoder" must be specified, in order to transmit a single message. On the contrary, if the "Multi Frame" button is ticked, the location of the chosen file for reading or writing is set in the previous window, through the file browser.

The "Version" button displays a window, shown in figure A.4, where four different QR versions can be chosen: standard version 2, non-standard squared black and white, non-standard rectangular black and white, and non-standard squared colourful. Depending on the version chosen, the transmission rate and reliability vary, which is analysed in Chapter 6.

It is important to notice that the "Multi Frame" and "Version" settings must be configured equally on both transmitting and receiving phones for the application to work properly.

5.2 New framework

A new version of the framework, different from the FrameWork given at the beginning of the project, has been developed by one of the team members.

The code is organised in four separate packages. The main package comprises the classes ActivitySend.java and ActivityCapture.java that launch the transmission and the reception respectively; the versions' parameters; the user interface settings; and the camera settings. The three other packages contain files specific for the different QR versions that have been implemented: one package for the standard QR version number 2, one for the non-standard black and white QR version, and one for the non-standard colourful QR version.

In addition, the framework includes the Open Source Computer Vision Library (Open CV) [12]. OpenCV is a library of programming functions, developed originally by Intel, and mainly aimed at real-time image processing.

5.3 Java implementation

This section explains how the Java code is organised and how the different classes that have been implemented interact which each other.

To make the following explanations simpler and less redundant, distinction between the versions used is made only when part of the process is specific to a given version. Otherwise, when no distinction is made, the explanations are valid for all the versions.

5.3.1 Transmission

Figure 5.1 shows the interactions between the implemented classes when transmitting a message, or a file. The red boxes and arrows are specific to the multi-frame mode.

5.3.1.1 Single Message

The different steps in the transmission process for a single message are described below. These steps correspond to the numbers in blue in figure 5.1.

First The *ActivitySend.java* file contains the functions that initialise the transmission. The version selected and the message to encode, set by the user, are retrieved from the user interface's settings. In the *init()* function, the size of the QR code and the finder patterns are set according to the version chosen. Still depending on the version chosen, the proper function for encoding (*encodeV2(), encodeNonStandard(), encodeColorful()*) is called in the *onCreate()* function.

Second In the *encode()* function, the bytes are retrieved from the text message to encode, and passed as argument to a newly created instance of the *Encoder.java* class. This encoder encodes the data using the ReedSolomon encoder, places the data in a boolean matrix, masks the data, and adds the format information. The Encoder returns the generated QR code, as a boolean matrix.

Third The *updateCode()* function is called with the boolean matrix generated by the Encoder as an argument. This function is in charge of displaying the QR code on the phone's screen, using the *draw()* function.

5.3.1.2 Multi Frame

The different steps when transmitting a file are described below. They correspond to the numbers in red in figure 5.1.

First The version and the file to transmit are retrieved from the user settings. The parameters specific to the chosen version (QR code's size, finder patterns' size, functions to call) are set in the *ActivitySend* class.

Second The *encode()* function, called when the "send" button is pressed by the user, initialises and starts a new instance of the *ReadFromFile* thread. The *ReadFromFile* thread reads the file to transmit, and splits it into data pieces, which length depends on the maximum amount of data the QR code generated for the chosen version can support.

Third The *ReadFromFile* thread creates a new instance of the *Encoder* class, that generates the QR codes from the data pieces.

Fourth The generated QR codes are stored in a "symbol buffer", by calling the *addSymbols()* function, defined in *ActivitySend.java.* A new thread, thread t1, launched in the *encode()* function, checks the "symbol buffer" every 600 ms, meanwhile thread t1 sleeps. If the "symbol buffer" is not empty, thread t1 removes the QR codes from it. Then, thread t1 warns the UI thread, via the handler, that it has removed a QR code element from the "symbol buffer", so that the UI thread can display that QR code on the screen. Consequently, QR codes are displayed on the screen every 600 milliseconds.

Fifth When the UI thread receives a notification from the handler that a QR code has been dequeued by thread t1, and can be displayed on the screen, the UI thread calls the *updateCode()* function that draws the QR code on the phone's screen.

When the "symbol buffer" containing the QR codes has been emptied, which means that all the QR codes have been displayed on the screen, thread t1 that reads the "symbol buffer" stops reading it, and warns the UI thread to display a "Transmission completed" message on the screen. The *stop()* function is called to stop the threads properly and reset the "symbol buffer". A new file can then be transmitted.

5.3.2 Reception

Figure 5.2 shows the interactions between the implemented classes when receiving a message or a file. The red boxes and arrows are specific to the multi-frame mode.

5.3.2.1 Single Message

First The settings (QR version for decoding) are retrieved from the UI settings.

Second When the "decode" button is pressed by the user, the *decode()* function is called. Pictures are taken by the camera, and put in the *imageQUEUE* buffer (done in the *onPictureTaken()* function). A thread is then launched that checks regularly (every 100 ms) if the *imageQUEUE* is not empty and if not, removes the first image from the *imageQUEUE.* The image is processed by the *QRReader* in order to find the patterns, and to retrieve the bits from the QR code. If the image processing is successful, which means that the finder patterns have been successfully detected, the bits are retrieved from the image and the bit matrix is decoded by the *Decoder.* The initial message can be restored. Pictures are taken by the camera until the message is successfully decoded, even though the message contains errors. Taking only one picture would not be reliable, since the finder patterns and decoding process are not 100% reliable, and printing a null message on the screen has to be avoided.

Third The thread warns the UI thread, via the handler, that the initial message has been restored and can be displayed on the screen. The UI thread displays the message on the screen using the *writeMessage* function.

5.3.2.2 Multi Frame

First The settings (file to write to, and QR version for decoding) are retrieved from the UI settings.

Second and Third When the "decode" button is pressed by the user, the *decode()* and the *cameraOn()* functions are called.
In the *cameraOn()* function, a thread *tCamera* is launched that takes pictures. Pictures are then stored in the imageQUEUE buffer (done in the *onPictureTaken()* function). For better performances, pictures at taken by the camera at a dynamic frequency, depending on the number of pictures stored in the *imageQUEUE* buffer. The frequency is adapted as follows:

$$frequency = 140ms + 20 * imageQUEUE.size()$$

The *decode()* function creates new instances of the *QRReader*, containing the image processing functions, and the *ReadQR* and *DecodeAndDisplay* threads.

Fourth The *ReadQR* thread is mainly in charge of processing the images captured by the camera, in order to retrieve the boolean matrices from them. To do so, the *ReadQR* calls the relevant functions in the *QRReader* instance, which processes the image and returns the boolean matrix if the process is successful.
Only one instance of the *ReadQR* thread is launched, however, images are captured roughly every 140 to 200 ms, which means that a way to update the images processed by the *QRReader* had to be found. Moreover, the image update should not be done in the middle of another image processing, for consistency. Therefore, a synchronisation mechanism, based on mutex and wait/notify calls, has been established between the camera thread *tCamera* and the *ReadQR* thread.

Here is the pseudo-code of the *ReadQR* class:

```
Run function of the ReadQR class
while stop is false
    busy is true
    process the image
    if success
        get the boolean matrix
        put the matrix in the queue
        notify DecodeAndDisplay thread that new element in the queue
    busy is false
    wait (blocking operation)
```

Here is the pseudo-code of the onPictureTaken() function, called when a new picture has been taken by the camera:

```
IF imageQUEUE is not full
    add the image to the imageQUEUE
```

A new thread, thread t2, is in charge of emptying the *imageQUEUE* and update the frame in the *ReadQR*:

```
while imageQUEUE not empty
    IF ReadQR is not busy
        new image = poll the first image from the imageQUEUE
        update the new image the QRReader
        notify the ReadQR thread (wakes the ReadSR thread up)
    sleep 50 ms
```

Thread t2 has been added to avoid the *imageQUEUE* from getting too large, and exploding in case a large file is transmitted. Since the *imageQUEUE* is checked every 50 ms by t2, the *imageQUEUE* never contains more than one element.

Fifth The boolean matrix, retrieved from the image processed by the *QRReader*, is put in a queue. This queue is an instance of the *Queue* class, shared by both the *ReadQR* and the *DecodeAndDisplay* threads.

Sixth and Seventh The *DecodeAndDisplay* thread is mainly in charge of decoding the QR codes and writing the decoded messages to the file the user specified. The *DecodeAndDisplay* thread removes boolean matrices from the queue, gives them to the *Decoder*, which decodes them and returns back the decoded message.

Eighth The decoded message is written to the specified file on the SD Card.

At the end of the reception, the user presses the stop button that properly stops all the running threads, and makes the implementation clean for receiving or transmitting a new file/message. It is to notice that when the stop button is pressed, thread t2 continues running until it has emptied the *imageQUEUE*, so that the entire queue has been processed.

The multi-threading implementation unburdens the main thread and increases the speed of the decoding process, since the image processing and the QR code decoding are done in parallel.

5.3.3 Reliability

When transmitting multiframes, QR codes are displayed on the screen every 600 milliseconds. On the receiver's phone, pictures are taken roughly every 140 ms. Therefore, up to four pictures of the same QR can be taken by the camera.
To increase the image processing reliability, a frame number is added at the beginning of each transmitted frame, in the *ReadFromFile* class. When decoding, the best frame, amongst the several frames containing the same frame number, is written to the file. The best frame is the one containing the least error number, which is calculated by the Reed-Solomon decoder. The frames containing too much errors are discarded without being processed to gain CPU processing.
However, no mechanism has been implemented to recover a missing frame.

FIGURE 5.1: Encoding process: interaction between classes and functions' calls.

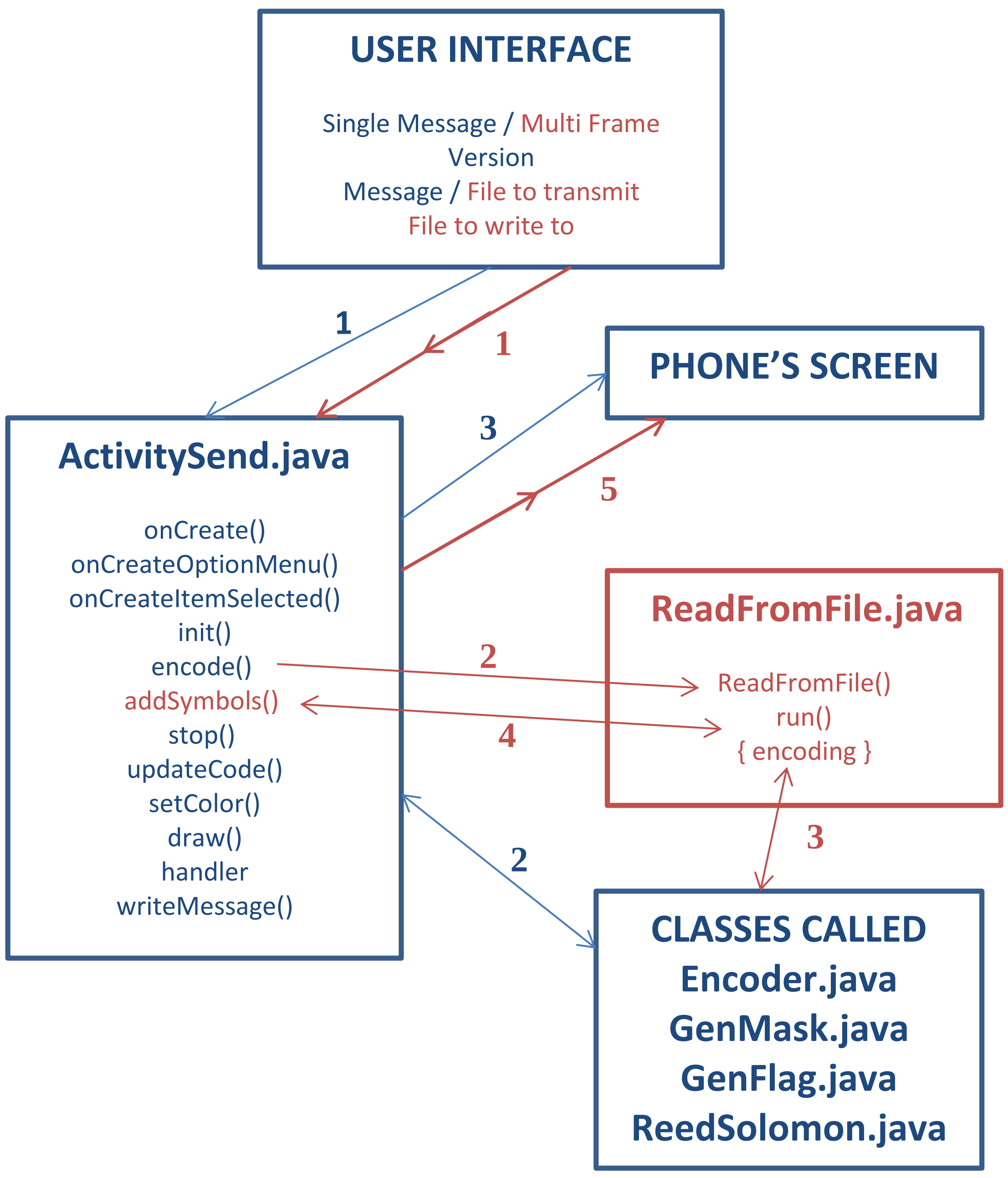
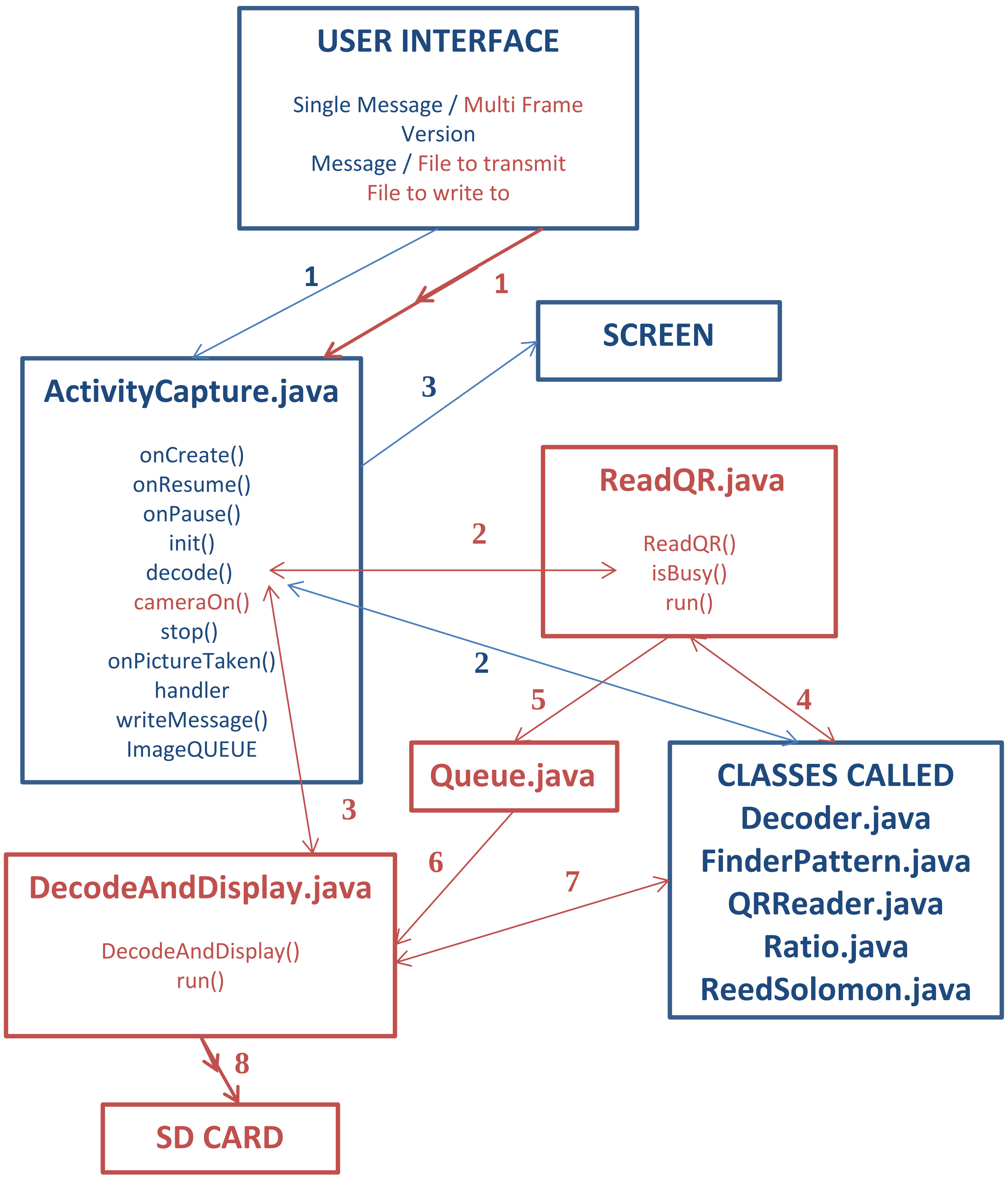


Figure 5.2: Decoding process: interactions between classes and functions' calls.



Chapter 6

Results and Discussion

This chapter aims at analysing our implementation and results, in terms of transfer speed and reliability. Future possible work for research and Android application improvements are also discussed.

6.1 Performance

In this section, the performances of the application are discussed. The transfer speed and reliability for the different versions implemented are compared and analysed.

6.1.1 Transfer Speed

The following table summarises the transmission speed of the four QR versions that have been implemented. The frequency (in ms) is how fast the QR codes are changed on the phone's screen.

Version	bytes/frame	frequency (ms)	speed (Kb/s)	speed (bytes/s)
version 2	13	700	0.019	19
NS square B&W	220	1000	0.22	220
NS rectangular B&W	660	1000	0.66	660
NS square colorful	830	700	1,19	1185

Table 6.1: Comparison between the four QR versions' transmission speed

The basic requirements are attained with the standard version 2; more than 100 bits/frame to be transmitted at a speed of at least 100 bits/s.
The non-standard versions allow far higher transfer speeds, the highest one being achieved

with the colorful version. The transmission in that case exceeds 1 Kbyte/s.
The amount of data encoded in one QR depends on the maximum amount of data the version used for that QR can support. As for the frequency, tests have been conducted and a compromise between speed and reliability has been made. Furthermore, increasing the frequency in displaying the QR codes, results in taking less pictures of the same frame, which makes reliability less accurate (same frames are captured several times so that the best one is chosen and written to the SD card).

6.1.2 Reliability

In the Java implementation, reliability has been improved by taking several pictures of the same QR codes, so that same frames, i.e. having the same frame number, are then compared. The best frame is selected to be written to the SD-card.

Concerning the color version, the 16 colors works well for smaller versions of QR code, however the reliability decreases as the version increases because the size of a single module is reduced thus making the colors difficult to distinguish. Therefore, the 16 colors are implemented in Matlab only as it needs further research, and the 8 colors are implemented in both Matlab and Java. The performance of the 16 colors can be improved by testing other methods like differential encoding, QAM, etc. which can be viewed as a future work of this project.

High reliability is hard to achieve with lots of data in one code and fast transition times. So there is always a tradeoff between reliability and speed. The main limitation here is the inability to retransmit and acknowledge frames (ARQ). The sizes and speeds we use ensure that the transmittion is successful most of the times when transmitting files of up to 50kb.

Especially for the rectangular version the perspective transform is the main issue with reliability. Because as the modules are getting smaller and the size between the reference-points gets bigger, the accuracy with which we can eliminate the distortions is not good enough to reliably locate the modules. The perspective transform is unable to perfectly reconstruct the QR code as the image projection of the camera is not perfect due to lens distortion.

6.2 Limitations

In this section, the current limitations of the application and our non standard approach are discussed. There are three main limitations: the resolution, the processing time and the image transformation.

The resolution is an obvious one and is closely related to the processing time. The resolution for all versions is currently fixed at 640x480. Taking into account at least 3x3 pixels per module are needed to reliably decode an image this puts up a barrier around $(480/3) * 0.8 \approx 130$ modules. This is taking a margin of 20% around the image into account. Stretching that limitation to some extent has been tried by using rectangular codes.

The processing time is mainly limited by the image processing. Image processing takes around 100 to 200 ms depending on the version used. The decoding including Reed-Solomon only takes 10 to 20 ms on a good picture, and around 430 to 470 ms if the Reed-Solomon decoder has a lot of errors to correct.

The perspective transformation works well even when recording with an angle. There is however a limit due to lens distortion which cannot be corrected. A more sophisticated transformation or realignment procedure through additional patterns would improve the decoding.

6.3 Future work

As far as reliability is concerned, a acknowledgement channel could be implemented in future work. Indeed, in the current Java implementation, when two frames in a row are missed (because of finder patterns not found, or too many errors to correct in the Reed-Solomon), the decoding process is stopped. Nothing is done to recover missing frames. An idea could be to acknowledge the frames that are successfully received, using the camera flash. The transmitting phone would then either retransmit the frame, or pursue the transmission with the next QR code, if a flash is received as an acknowledgement.

Concerning the colorful QR code version, HSV color coding with 16 colors worked well for the smaller versions of QR codes; till version 6 QR codes. The 8 HSV colors worked till version 10. Since the RGB color model provided higher reliability and performance than the HSV model, the later work was focused on RGB color model. One of the future work of this project can be to increase the reliability of HSV color model for QR code encoding and decoding as it provides more colors than the RGB. Moreover, the brightness dependence of the QR code can be neglected by not putting any information in the v channel which can make it more tolerant to some type of errors such as errors due to the lighting in the background.

Another possible area of improvements concerns support for higher camera resolutions. The main difficulty here is processing time and making use of the gain in resolution with codes that contain more modules. To achieve this a fast search for the finder patterns can be done on a lower resolution version of the image after which a perspective transform on the high resolution image can be done. Through alignment patterns in the code this first transformation can then be refined and the modules can be read.

Furthermore, the error correction level in our project is fixed at 50%, which means half of the data codewords are covered by the error correction codewords. However, in ideal case, the error correction level should depend on the noise level of the environment. For example, if the transfer is performed in the outdoor environment, then source of the noise comes from the sunlight and it will cause a strong reflection on the screen. In this scenario, the error correction level has to be higher. One method is to use the light sensor of the smartphone. If the strength of the detected light is strong, the number of error correction is set to a high level (the optimal percentage should be determined according to the outcome of this practical test).

Chapter 7

Conclusion

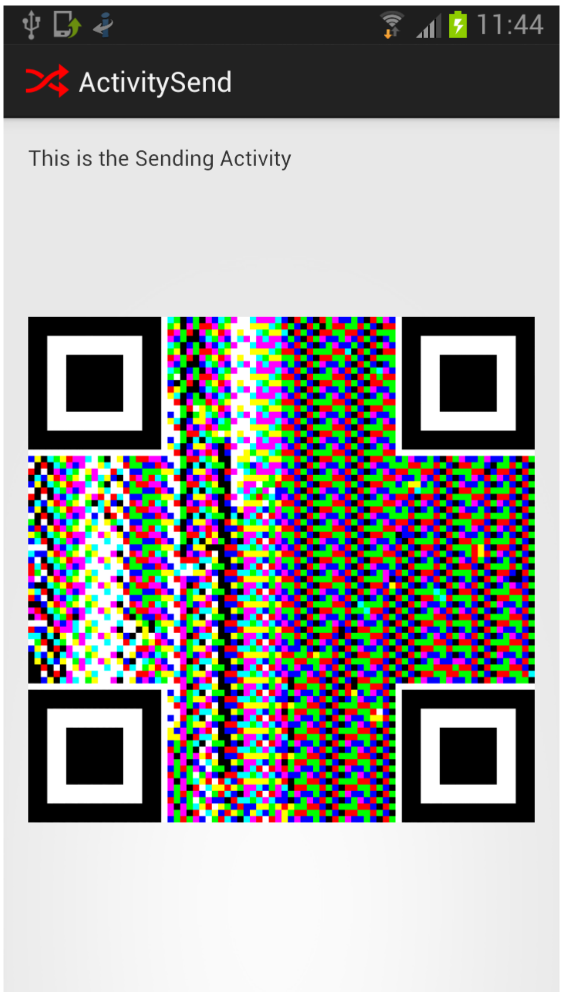
During this project, an in-flight file transfer application has been developed successfully. The Android application relies on QR codes to transmit files from one phone's SD-card to another, so that radio propagation is avoided onboard the aircraft. This project has been divided into two main parts: the Matlab implementation and the Android application development. In Matlab, different QR versions have been implemented and algorithms have been tested. In Matlab, all the standard versions from 2 to 40 have been implemented, the non-standard square version, in black and white, and the color version, up to 16 colors. In Java, the tested Matlab codes have been implemented as an Android application. Focus has been put on the reliability and the transmission speed, so that a compromise has been reached between both.

Several QR codes versions have been implemented and compared. First, the version 2, according to the international standard has been explored. This version fulfils the basic requirements for this project, since it supports transmission of 100 bits/seconds. However, in order to meet advanced requirements and to increase the transmission speed, non-standard versions have been developed. The square black and white version, the rectangular one and the color QR code version have been implemented in Java. With all the versions, images, text files, pdf, and music can be transferred reliably, depending on the version used. The best reliability of transfer is achieved with the color version, according to tests conducted. The color version is also the one supporting the highest transmission rate.

This project provided us with a great platform for learning, on different levels. The Android platform and the Matlab software allowed us to apply our theoretical knowledge to a practical application. Moreover, this project has helped us to work as a team in an organised and communicative manner. Finally, the weekly deadlines and project meetings have helped us planning our work efficiently.

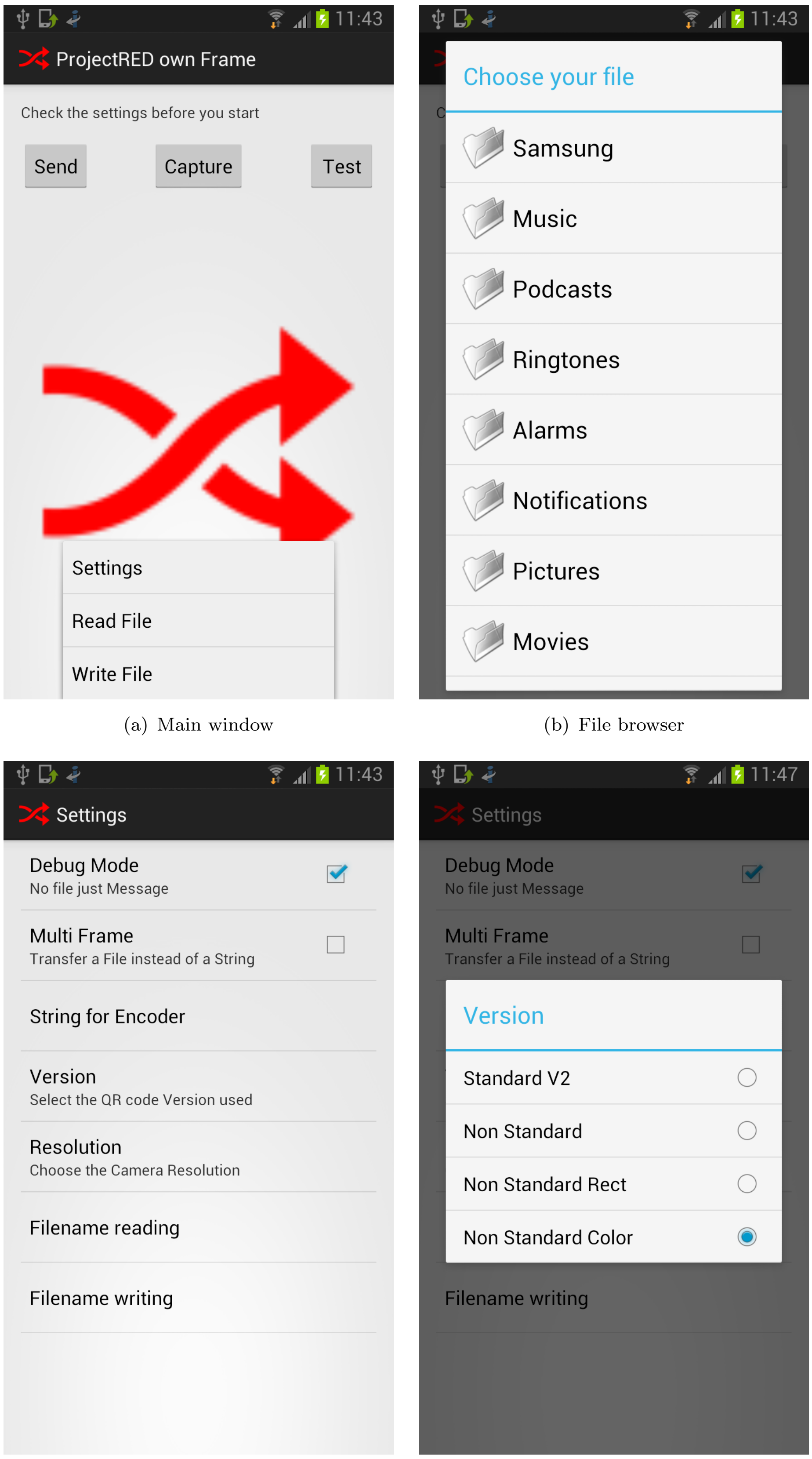
Appendix A

User Interface Captures



This is the Sending Activity

Figure A.1: Sending Interface



(a) Main window (b) File browser

11:43
Settings

Debug Mode ✓
No file just Message

Multi Frame ☐
Transfer a File instead of a String

String for Encoder

Version
Select the QR code Version used

Resolution
Choose the Camera Resolution

Filename reading

Filename writing

11:47
Settings

Version

Standard V2 ○

Non Standard ○

Non Standard Rect ○

Non Standard Color ●

(c) Settings window (d) QR versions

FIGURE A.2: User Interface

Appendix B

Image Processing

Codewords	**H**	**S**	**V**
🟥 0000	0	1	1
🟧 0001	40	1	1
🟩 0010	80	1	1
🟩 0011	120	1	1
🟩 0100	160	1	1
🟦 0101	200	1	1
🟦 0110	240	1	1
🟪 0111	280	1	1
🟥 1000	0	0.5	1
🟨 1001	40	0.5	1
🟨 1010	80	0.5	1
🟩 1011	120	0.5	1
🟩 1100	160	0.5	1
🟦 1101	200	0.5	1
🟦 1110	240	0.5	1
🟪 1111	280	0.5	1

Table B.1: Codewords: 16 HSV color system with varying hue & saturation

Codewords	**H**	**S**	**V**
🟥 0000	0	1	1
🟧 0001	40	1	1
🟩 0010	80	1	1
🟩 0011	120	1	1
🟩 0100	160	1	1
🟦 0101	200	1	1
🟦 0110	240	1	1
🟪 0111	280	1	1
🟥 1000	0	1	0.5
🟫 1001	40	1	0.5
🟩 1010	80	1	0.5
🟩 1011	120	1	0.5
🟩 1100	160	1	0.5
🟦 1101	200	1	0.5
🟦 1110	240	1	0.5
🟪 1111	280	1	0.5

Table B.2: Codewords: 16 HSV color system with varying hue & brightness

Bibliography

[1] KTH: Green Team. In flight file transfer. 2012. URL http://www.s3.kth.se/signal/project_course/2012/green/.

[2] Mathworks. Matlab R2012b. URL http://www.mathworks.se/products/matlab/.

[3] Eclipse: multi-language integrated development environment. URL http://www.eclipse.org/.

[4] ADT: Android developer tools. URL http://developer.android.com/tools/index.html.

[5] Git: distributed version control system. URL http://git-scm.com/.

[6] ISO/IEC 18004. Information technology – Automatic identification and data capture techniques – QR code 2005 bar code symbology specification. *International Standard*, (Second Edition), September 2006. URL http://www.iso.org.

[7] Wikiversity. Reedâsolomon codes for coders. URL http://en.wikiversity.org/wiki/Reed-Solomon_codes_for_coders.

[8] Yuxi Tang. 基于DSP的QR像 研究与 (Study and Implementation of QR code Image Recognition based on DSP), 2009.

[9] Wikipedia: Affine transformation. URL http://en.wikipedia.org/wiki/Affine_transformation.

[10] Wikipedia: Homography. URL http://en.wikipedia.org/wiki/Projective_transformation.

[11] Wikipedia: Additive color mixing. URL http://en.wikipedia.org/wiki/File:AdditiveColor.svg.

[12] OpenCV: Open source computer vision library. URL http://opencv.org/.