Image Based Data Transmission between Smart Phones and Computers

Wei Fang
Department of Electrical Engineering
Stanford University
Stanford, CA
maxfang0000@gmail.com

***Abstract*—This document proposes a method to extend 2D barcodes so that smart phones and computers can make use of the extended code to exchange information at reasonable data rates.**

Keywords- barcode; data rate; animated; smart phone

I. INTRODUCTION

People have been using barcodes to exchange small amounts of information for decades. As an extension of linear barcode, 2D barcodes which can hold more data have been popular and enabled larger amounts of data like a URL to be exchanged.

In this report, I will present the utilization of another two available features of the popular 2D barcode presenter and reader – smart phones, to make barcodes to be able to hold more data. The key point is to make use of different colors and time-varying coding.

Design of the code is explained in detail. A code reader on the smart phone side has been implemented. Problems currently existing in the process and possible future improvements are also discussed.

II. DESIGN OF THE CODE

Currently, only 1 type of code is implemented. The implemented code is a 64-by-96-block code. The code only makes use of 9 colors. The code contains the following regions: finder patterns, header, and data field. Each of the regions is explained in detail in the following paragraphs. A sample code with mark-ups of regions is shown in Figure 1.

A. Selection of Color

Because the raw image I received from the camera driver is in the YUV space, my criteria of color selection is to separate the colors as much as possible in the YUV space. The UV plane at different Y values is shown in Figure 2. Note that not every point in the 3D YUV cube is a valid color.

The 9 colors used in the code are: black, white, green, magenta, red, cyan, dark green, dark magenta and dark blue. 4 colors: green, magenta, red and cyan are used to represent data bits. Each of the 4 colors can represent 2 bits of binary data. On some displays, green can appear similar to cyan and red can appear similar to magenta, when I assign the 2-bit values to colors, I made sure that the color of 00 and11 are not similar, 10 and 01 are not similar, so that the chance of 2-bit error can be reduced.

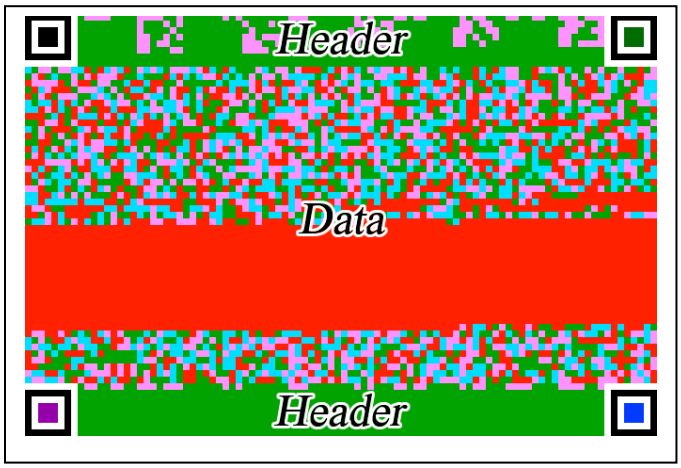

Figure 1. Sample code

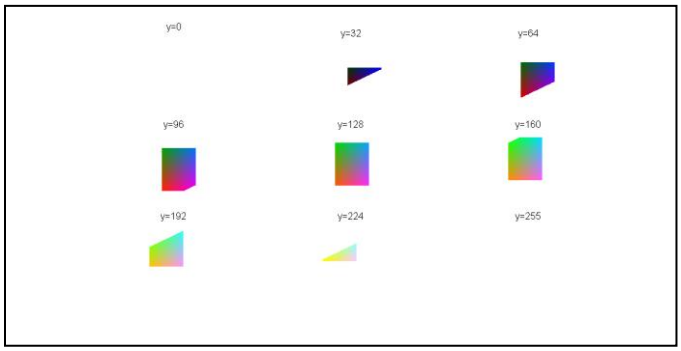

Figure 2. Rendering of UV planes at different Y values

B. Finder Patterns

The finder patterns are borrowed from QR code. This form of pattern enables a fast detection of the existence and position of the code: no matter the code is placed skewed or not, a scan line crossing the central square's opposing edges will always

Identify applicable sponsor/s here. *(sponsors)*

detect a 1 black, 1 white, 3 black, 1 white, 1 black pattern. This property of the pattern is illustrated in Figure 3.

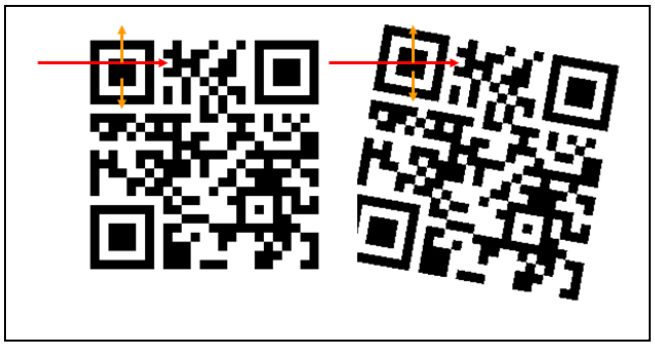

Figure 3. 1-1-3-1-1 pattern of the finder pattern

As an extension of the QR code's finder patterns, I used dark green, dark magenta and dark blue to render the 3 central squares on upper right, bottom left and bottom right corners. This will make a smart phone with camera to detect the direction of the code easier and faster.

C. Header

The header is divided into 2 parts, one sitting at the top of the code, the other at the bottom of the code. Both the top and bottom region are 8 by 80. The header contains important control information for the transmission to happen. These include: size of the file to be transmitted and name of the file.

When the file is longer than the limit of size the grid can hold, multiple frames of code have to be employed. You can think of a frame as a page of code. The transmitter will "play" the code frame-by-frame so that the receiver can eventually collect all the frames after some time. In this case, the header also has the responsibility to present the total number of frames of the long file and the frame number the frame currently being played.

Because of the importance of the header, only 2 colors with big separation: green and magenta are used. Each block will only represent 1 bit in the header. The crucial fields of the header, i.e. file size, number of frames, current frame number, are stored in the first line of the top part, and repeated in the first line of the bottom part.

The detailed design of the header is shown in Table 1 and 2.

TABLE I. BITS IN THE TOP HEADER

Bits	Content
0-15	Fixed pattern A
16-25	Current frame number
26-35	Total number of frames
36-55	Length of the file in bytes
56-63	Length of file name in bytes
64-79	Fixed pattern B
80-	File name in UTF-16

TABLE II. BITS IN THE BOTTOM HEADER

Bits	Content
0-15	Fixed pattern B
16-63	Repeatition of top header
64-79	Fixed pattern A
80-	File name (continued)

D. Data Field

The data field is 48-by-96 in size, containing binary data and error correction bits. There is always a data correction block after 2 data blocks. The currently implemented error correction algorithm is a Hamming code, adding parity 4 bits to every 8 data bits. It is good to correct up to 1 bit error in every 8 bits.

The data field can hold 1152 raw bytes. Since each byte comes with half a byte of error correction information, 1 frame of the data field is good to hold 768 bytes of data.

III. IMPREMENTATION DETAILS

The receiver algorithm is implemented on an Android phone with camera. Since it's not a computationally powerful system, many parts of the implementation have concerns on efficiency.

Basically the receiver will go through the following steps for each incoming image: locating the code, calculating positions of code blocks, read the header and data. The first 2 steps will be explained in detail in the following paragraphs.

A. Locating the Code

As described in part II (B), this step will make use of the finder patterns. The algorithm will read the Y channel of the image line by line. An empirical threshold is set and whenever a transition of luminance happens, it will buffer the distance between the current transition and the previous transition. If the recent buffer data is consistent with the 1-1-3-1-1 ratio, the algorithm will also check the vertical direction. If both directions satisfy the pattern, the algorithm will output the position of the center of the pattern, and the size of the central square. The size of the central square is very important to the following step.

Relevant code of this step can be found in the following path of the submitted zip archive:

BarcodeReceiver/jni/NativeProcessor.cpp, Line 476

Function: findFinderPattern

B. Calculating Positions of Code Blocks

Because nobody can ensure the code is placed parallel to the camera's image plane, and parallel to the image plane's x and y axes, I have to map the positions of data blocks to the positions of the received image.

With the positions of the 4 finder patterns, this problem appears to be a linear interpolation of the positions of the 4 finder patterns according to the positions of the finder patterns

on the code. However, this is not true. For a 2D code which is not so dense, this might be doable, however, it is theoretically wrong and will cause problem on such a dense code.

Because usually the code is not placed in parallel to the camera's image plane, distance from the code plane to the image plane changes across the image. The parts of code closer to the image plane will end up in bigger blocks, while the parts farther will end up in smaller blocks. On the other hand, the simple linear interpolation will end up with equally divided blocks in x and y directions. The linear interpolation reconstruction is compared to the real case in Figure 4.

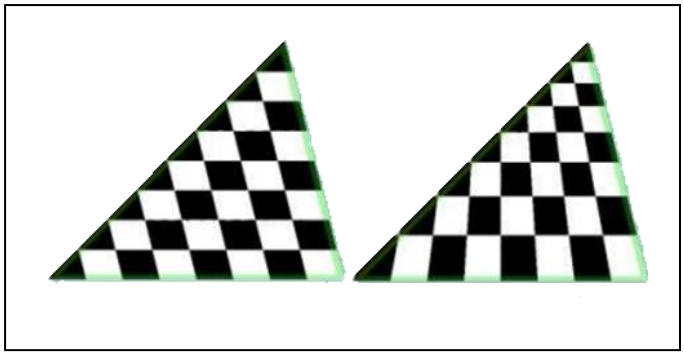

Figure 4. Linear interpolation (left) vs. real case

This is where I make use of the size of the central squares in the finder patterns. My assumption is that the code is placed in a plane so that the distance will change linearly as the position changes across the plane.

For example, if I'm finding block (y0, x0) on the image, I will first interpolate the block size to get the straight lines representing y=y0 and x=x0. Then, I will use the vector form of straight lines to solve for the intersection of the 2 straight lines. Because all the values are 2 dimensional, the operation is equivalent to inverting a 2 by 2 matrix and multiplying it by a 2 by 1 vector.

Relevant code of this step can be found in the following path of the submitted zip archive:

BarcodeReceiver/jni/NativeProcessor.cpp, Line 651

Function: translateCoordinate

BarcodeReceiver/jni/NativeProcessor.cpp, Line 127

Function: ProcessFrame

Because this is the computationally heavy part, I optimized this part by caching a lot of values and avoiding floating point operations as much as possible. The C++ code may not appear readable. There is an equivalent Matlab code in the following path of the submitted zip archive:

Matlab/transform.m

The effect of this step is shown in Figure 5. On the other hand, if the estimation of depth is not done properly, the result will end up totally wrong. A wrong example is shown in Figure 6, where the size of the top left central square is measured 2 pixel smaller. You can see clearly in Figure 6, many scan lines go into wrong rows of blocks. The scan lines will be in-sync in the beginning, gradually get out of phase in the middle and get back in-sync again in the end.

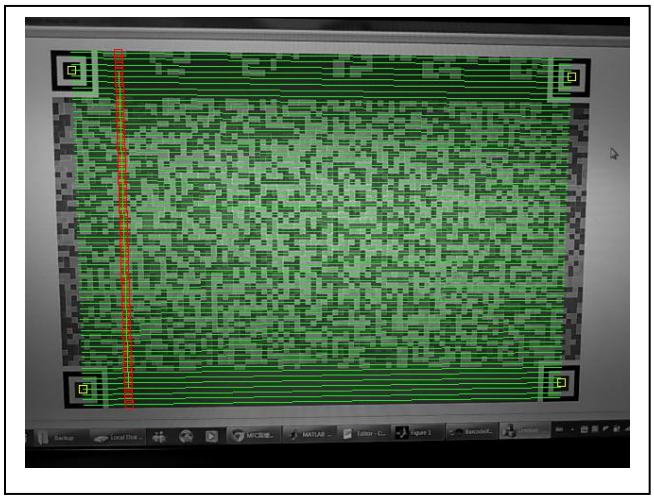

Figure 5. Correct algorithm

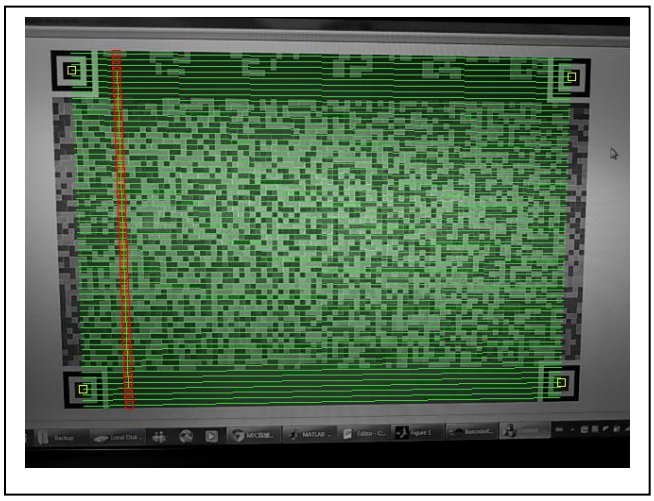

Figure 6. 2 pixel smaller estimation of the top left square

IV. Experiment Results and Future Improvements

The algorithm can work at 15 frames per second, yielding an optimal data rate of up to 90kbps. However, it drops some of the frames in the sequence so that it takes longer than the ideal time. I tested the algorithm on a 15kBytes file and the transmission time takes between 2 seconds and 6 seconds. (It is already an improved algorithm different from the one in the poster session)

The main reason of dropping frames lies in the finder pattern detection part. The algorithm fails at times so that the following part cannot run. It might be the empirical threshold making the trouble. A future algorithm should spend some time estimate the exact luminance of black and white blocks.

Acknowledgment *(Heading 5)*

This is a single person project and all the parts are done by myself.

REFERENCES

[1] QR Code: http://en.wikipedia.org/wiki/QR_code

[2] Parity check: http://en.wikipedia.org/wiki/Parity_bit

[3] Hamming Code: http://en.wikipedia.org/wiki/Hamming_code