Examensarbete

Improvement of recognition percentage and speed of ZXing based cellphone EAN-13 reader

av

Niklas Hayer

LIU-IDA/LITH-EX-G--10/021--SE

2013-04-12
Examensarbete

Improvement of recognition percentage and speed of ZXing based cellphone EAN-13 reader

av

Niklas Hayer

LIU-IDA/LITH-EX-G--10/021--SE

2013-04-12

Handledare: Johan Larsson
Examinator: Erik Berglund
1 Abstract

Prevas AB, a consulting company focused on industrial systems and embedded systems, are looking to expand their mobile phone competency and started two projects for this purpose. The first is a diploma work to create a proof-of-concept barcode reader running on Symbian mobile phones, and the second is a student project for the new Innovative Programming bachelors degree to create a functioning price comparison application using IMS\(^1\) around the proof-of-concept created earlier.

Both projects achieved their stated goals. However, due to none of the projects focusing on efficiency or theoretical correctness the resulting application is difficult to start, runs slowly, leaks memory and is codewise bloated. These defects prevent the application from being realistically deployable, and a third project is started to clean up the application and improve it. This report describes the solutions employed in that third project.

The barcode reader in the application was split into two parts: First is the actual application written in C++ responsible for capturing images, sending them to the decoding server and displaying the result to the user. Second is the decoding server, a separate application written in java that using a TCP-socket waits for images, decodes them, and returns the result.

The primary goal of this project is to not only improve, but actually integrate the decoder into the C++ application, massively improving speed and memory usage. This report was written to highlight the lessons learned during this project, to describe the algorithms and methods used to improve the decoder and to be of use to anyone else writing time-constrained and heavily optimized barcode or pattern recognition code for Symbian devices.

\(^1\)IP Multimedia Subsystem, a 3rd generation mobile architectural framework
## Contents

1 Abstract ................................................. 4

2 Introduction ............................................. 7
   2.1 Problem/Motivation ............................... 7
   2.2 Goal/Purpose ........................................ 7
   2.3 Target Audience ..................................... 7
   2.4 Glossary .............................................. 7
   2.5 Outline ................................................ 7
       2.5.1 Background ........................................ 7
       2.5.2 Requirements/Design .......................... 8
       2.5.3 Implementation ............................... 8
       2.5.4 Discussion ........................................ 8

3 Background ............................................... 9
   3.1 Sponsor ................................................ 9
   3.2 Technology ............................................ 9
   3.3 Legacy .................................................. 9
   3.4 EAN-13 ............................................... 10

4 Requirements/Design ..................................... 12
   4.1 Requirements ......................................... 12
   4.2 Goals .................................................. 12
   4.3 Design ................................................ 12

5 Implementation .......................................... 13
   5.1 Porting ................................................. 13
       5.1.1 Valarray ........................................... 13
   5.2 The ZXing Decoding Process ........................ 15
   5.3 The Binarizer Algorithm ............................ 17
       5.3.1 The GlobalHistogramBinarizer .................. 17
       5.3.2 A New Binarizer .................................. 17
       5.3.3 Implementation of BrightnessRefiningBinarizer 19
       5.3.4 Performance ........................................ 20
   5.4 Binarized format change in ZXing .................. 20
       5.4.1 BitFlipArray iterator ........................... 21
       5.4.2 Comparing functions using new binarized format 22
   5.5 Testing ................................................ 23
   5.6 Integration ............................................ 24
       5.6.1 Moving of functionality inside BarcodeScannerViewfinderView 24
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>Discussion</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>6.1 Feature-Limiting</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>6.2 Scanlines</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>6.3 Parallelizing</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>6.4 Smoothing</td>
<td>28</td>
</tr>
<tr>
<td>7</td>
<td>Result</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>7.1 Requirements</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>7.2 Goals</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>7.3 Screenshot</td>
<td>31</td>
</tr>
<tr>
<td>8</td>
<td>Appendix</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>8.1 EAN-13 Decoding Tables</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>8.1.1 First digit code table</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>8.1.2 First digit group decoding table</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>8.1.3 Second digit group decoding table</td>
<td>33</td>
</tr>
<tr>
<td>9</td>
<td>References</td>
<td>32</td>
</tr>
</tbody>
</table>
2 Introduction

2.1 Problem/Motivation
This project was conceived to address several problems with a preexisting application, primarily it's slow and unreliable barcode recognition system.

2.2 Goal/Purpose
The goal of this project is to remove all the major obstacles that prevent the application\textsuperscript{2} from being used by normal users. For specific goals please see subsection 4.1 on page 12.

2.3 Target Audience
This report is aimed mostly at professional programmers who are to continue working with the produced code or who are looking for information to write their own barcode decoder.

Experience with object-oriented programming and C/C++ in particular is expected. The reader should be familiar with asymptotic notation. Basic knowledge of image processing and RGB to HSV\textsuperscript{3} conversion is required to understand this report fully.

2.4 Glossary

\textbf{Scanline} A line drawn over an image containing a barcode. For an illustration of a scanline please see figure 7 on page 26.

\textbf{EAN} European Article Number, the most common barcode format for labeling of consumer goods.

\textbf{ZXing} Zebra Crossing, an open-source barcode decoder written in java, but ported to many other languages.

2.5 Outline
Here a brief description of each section is given.

2.5.1 Background
Background information for this very project. A short description on how EAN-13 barcodes work.

This report is written with all sections containing some background, and thus this section is not required reading to understand the implementation.

\textsuperscript{2}The working name for the application is PriceMaster. This name will be avoided in the remainder of this report. Conceiving a more suitable name is beyond the scope of this project.

\textsuperscript{3}Red-Green-Blue to Hue-Saturation-Value, two different systems to represent colors using integers.
2.5.2 Requirements/Design

A list of requirements and goals for this project is presented here. The design of the coding process is also very briefly described.

2.5.3 Implementation

Describes the implementation of the changes and additions to the application during this project. The implementations of the algorithms used for decoding barcodes in this application are presented and demonstrated here.

2.5.4 Discussion

Here the final result is discussed. Several dead-ends are discussed and the reasons for them not being implemented are outlined.
3 Background

3.1 Sponsor
Prevas AB are a Nordic IT company focusing on industrial IT and embedded systems. As high-powered mobile devices have become readily available and cheap it is becoming increasingly possible to use them to interact with industrial systems in new and exciting ways. Taking high security embedded systems or even cloud-based services and making them available to companies through interaction by mobile phone is an area with potential for expansion.

With this in mind, Prevas AB have sponsored this and other projects to build expertise and prepare solutions for potential customers looking to improve system interaction using cellphones.

3.2 Technology
• Carbide C++ 2.3.0
• Nokia E66 running S60 3:rd edition

3.3 Legacy
The original application [1] consisted of a single view and did nothing but barcode decoding using a C++-client Java-server architecture. After this a group of students were tasked with building a networked price comparison application around this core. While the student group succeeded with most set goals, two large problems remained.

Firstly, when the students began designing the interfaces of the application they had little to no experience designing for Symbian mobile devices. This led to design common in normal applications written for desktop PCs. Once the project had progressed far enough that these faults became apparent it was too late to modify the core architecture, and it was decided to ship feature-complete instead of cutting features to improve code-quality.

Secondly, the original barcode decoder [1] had been tested on a high powered Symbian smartphone. Once the student group tested the application on their more representative phone slow decoding and out-of-memory errors became issues. It was decided that improving the speed and efficiency of the decoder was a too big job for their limited resources, and so the problem was documented and left for future projects to solve.

Several of the improvements made to the decoder in this project were suggested by it’s original author [1].
3.4 EAN-13

Barcodes of the EAN-13\(^4\) type are the most common form of barcode used in the western world, and are a very close cousin to UPC\(^5\). This means that a EAN-13 decoder is able to successfully decode UPC barcodes, however a UPC decoder cannot decode a EAN-13 barcode.

EAN-style barcodes do not care for the color of the barcode, it only cares about the length of consistent color. This allows EAN barcodes, or segments thereof, to be described using a X-Y-Z sequence. Every number in this notation describes the length of color, black or white, compared to all the other lengths of color in the sequence. For example, if the start of any EAN-13 barcode is described in this notation it becomes 1-1-1, as there is a sequence of black-white-black where the three stripes are of equal length.

The EAN-13 barcode contains three markers. One 1-1-1 sequence at the beginning and end, as well as one 1-1-1-1-1 sequence in the center of the code.

EAN-13 barcodes consist of 13 digits, of these the first 12 are data and the last one is a checksum. There are three groups of digits for which different decoding rules apply:

1. The first digit.
2. The first digit group.

\(^4\)European Article Number 13. There exists the shorter EAN-8 code as well, but it is seldom used.
\(^5\)Universal Product Code.
3. The second digit group.

The second group is easy, just read the length of consistent color and look up the values, in this case 1-4-1-1 repeated five times and then 1-2-1-3 once, in the EAN-13 R-code table (provided in the appendix 8.1).

The first group is a bit more complicated. Here every number has two possible encodings, one from the L-code table and one from the G-code table. Now we have to remember in which order we found matches in the two tables, and use that order to look up our first digit in the first digit table (again see 8.1).

While the values in the decoding tables appear random they are not, instead they are chosen to increase the chance that any systematic problem results in the read code not matching the checksum. This is why the L and G codes are mirrored.
4 Requirements/Design

4.1 Requirements

1. No features shall be removed from the application.
2. There shall be no noticeable delay when the barcode scanner is activated from the application.
3. The user must not be forced to do any setup beyond installation before entering the application.

4.2 Goals

1. To improve the responsiveness on the barcode scanner.
2. To improve the decoding accuracy of the barcode scanner.

4.3 Design

The decoder is designed to split into four short steps. First the taken image undergoes two transformations; it is converted from color to grayscale, then from grayscale to black and white.

The result is cached, and then finally passed into several decoders that do the actual work. These processes are described in 5.2 on page 15.
5 Implementation

5.1 Porting

The C++ port of ZXing makes heavy use of the STL\(^6\), a library that is not provided by the Symbian C++ implementation. To minimize the number of format changes inside ZXing and to allow parts of the new port to be committed back up into the mainline ZXing C++ port, it was decided to use a third party STL library. When this project was undertaken, there was no official STL library for Symbian from Nokia\(^7\) and therefore a port of uSTL was selected to provide the needed objects. uSTL was ported to Symbian by Penrillan, a company focusing on mobile development. uSTL is provided by Penrillan under an MIT license.

The two functions missing from uSTL, namely `max()` and `lround()`, were easily added.

5.1.1 Valarray

uSTL was a very good fit for ZXing, with the exception of valarray. Valarray is a seldom used more primitive form of vector, and it was used here to hold binarized arrays. A partial implementation of valarray was written during this project to allow the old ZXing code to compile and run if needed. As the format of binarized data was changed the container was switched from valarray to vector as described in subsection 5.4 on page 20. This is the implementation of valarray written in this project:

```cpp
namespace std {
    template<class T>
    class valarray {
    public:
        valarray(const T & val, size_t n) {
            this->arrsize = n;
            this->arr = new T[n];
            for(size_t i=0; i!=n; ++i) {
                this->arr[i] = val;
            }
        }
        valarray(size_t & n) {
            this->arrsize = n;
            this->arr = new T[n];
        }
        ~valarray() {
            delete this->arr;
        }
    }
}
```

\(^6\)The C++ Standard Library.

\(^7\)This has changed, using the official STL implementation over uSTL is now recommended.
The complexities for different operations in this implementation of `valarray` can be seen in table 1.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Valarray</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>valarray(n)</code></td>
<td>c</td>
</tr>
<tr>
<td><code>valarray(val,n)</code></td>
<td>n</td>
</tr>
<tr>
<td><code>size_t size()</code></td>
<td>c</td>
</tr>
<tr>
<td><code>T&amp; operator[](size_t n)</code></td>
<td>c</td>
</tr>
<tr>
<td><code>T operator[](size_t n) const</code></td>
<td>c</td>
</tr>
</tbody>
</table>

Table 1: Complexities for the new `valarray` implementation
5.2 The ZXing Decoding Process

The process of decoding takes the barcode through several modules inside the ZXing decoder, illustrated in figure 3.

1. The \textit{LuminanceSource} is responsible for transforming the image into a grayscale representation. For performance reasons, the conversion to grayscale is made on-the-fly whenever the value of any pixel is requested. The \textit{LuminanceSource} is designed to be overloaded to allow the ZXing decoder to operate on several types of images from different inputs. In the coming example the input is a \textit{SymbianImageLoader}. This object reads a JPG-image from the phone’s main memory and transforms it into a bitmap. When used live, the \textit{LuminanceSource} is instead a \textit{SymbianLiveImageLoader}. That object is designed to receive bitmap cellphone camera images, and is very flexible concerning the format of the received bitmap. This allows the \textit{SymbianLiveImageLoader} to use whatever bit-depth the camera is most comfortable generating, saving Symbian from having to convert the bitmap and greatly improving speed.

2. A \textit{Binarizer} is created. The binarizer’s job is to transform pixels from the grayscale generated by the \textit{LuminanceSource} to a binarized representation\textsuperscript{8}.

3. A \textit{BinaryBitmap} is created. This object uses a \textit{Binarizer} to generate a binarized representation of the image and saves it into a cache. This allows several barcode decoders to be used on the same pixel row without having to use the potentially computationally expensive \textit{Binarizer} again.

\textsuperscript{8}Binarized meaning binary, that is only black or white.
4. The `BinaryBitmap` is passed to a `MultiFormatDecoder`. This object uses all available barcode decoders in the order from most-likely-to-match to least-likely-to-match.

The following code from the application test module illustrates this process exceptionally well, and is representative of how the same objects are used when decoding images live.

```cpp
std::string CTest::attemptBarcodeDecoding(const TDesC& aFilename) {
    SymbianImageLoader* rawImage = new SymbianImageLoader;
    TS_ASSERT_DIFFERS((int)rawImage, NULL);
    Ref<LuminanceSource> rawImageRef(rawImage);

    TInt errcodeForBitmapLoad = rawImage->loadBitmap(aFilename);
    TS_ASSERT_EQUALS(errcodeForBitmapLoad, KErrNone);

    BrightnessRefiningBinarizer* binarizer = new BrightnessRefiningBinarizer(rawImageRef);
    TS_ASSERT_DIFFERS((int)binarizer, NULL);
    Ref<Binarizer> binarizerRef(binarizer);

    BinaryBitmap* bitmap = new BinaryBitmap(binarizerRef);
    TS_ASSERT_DIFFERS((int)bitmap, NULL);
    Ref<BinaryBitmap> bitmapRef(bitmap);

    MultiFormatReader testSubject;
    try {
        Ref<Result> answer = testSubject.decode(bitmapRef);
        std::string result = answer->getText()->getText();
        return result;
    } catch (ReaderException re) {
        return re.what();
    }
}
```

Noteworthy here is that this code does not leak memory despite containing no visible deletes. A reference counting scheme is employed here using the `Ref`-objects to ensure that all memory is freed when no longer needed. `Ref`-objects greatly simplify working with the java-like code. Additionally, using `Ref`-objects makes it possible to avoid having to use Symbians `CleanupStack`\(^9\). This improves both code readability and makes the code portable.

`BinaryBitmap` and `MultiFormatDecoder` have been kept in the code despite ZXing having been reduced to only using the EAN-13 and EAN-8 decoders. The overhead these two objects impose is minimal, and far outweighed by being able to, with minimal effort, re-enable other decoders if needed in the future.

---

\(^9\)A special stack used by Symbian to ensure that all memory is freed if a out-of-memory error is encountered.
5.3 The Binarizer Algorithm

The Binarizer is tasked with taking the grayscale representation of the scanline and converting it into a binarized\(^\text{10}\) representation. This is accomplished by first finding a good cutoff value, and then using it to convert the line. The cutoff value can be imagined as a point in the grayscale where everything above becomes white and everything below becomes black\(^\text{11}\).

Finding a good cutoff is a computationally intensive task, and as such the algorithms around it are presented here in detail.

5.3.1 The GlobalHistogramBinarizer

The GlobalHistogramBinarizer was the only binarizer that was implemented in the original ZXing C++ port, and therefore it was ported to Symbian as-well. Testing however revealed that this binarizer was a major factor in the barcode decoders low hit average.

The algorithm behind the GlobalHistogramBinarizer was as follows:

1. Create a histogram with sixteen bars, referred to as buckets, and assigning each bucket a grayscale range\(^\text{12}\).

2. Iterate over the LuminanceSource and getting each pixel’s grayscale value and sorting the values into the buckets. See Fig. 4 for a graphical representation of the histogram with the X-axis representing brightness and the Y-axis representing number of occurrences.

\(^{10}\)A black and white representation where every pixel uses only one bit.

\(^{11}\)This might be confusing, please see fig 6 for a grayscale where different cutoff values are tried.

\(^{12}\)For an 8 bit unsigned grayscale the first bucket got the range 0 to 16, the second bucket got the range 17 to 32 and so on.
3. Iterate over the generated histogram and find the white and black tops. In this example these are buckets 2 and 14 respectively.

4. Search for the lowest bucket between the two tops, making this our cutoff value. In this example the bucket 8 is selected, resulting in 136\textsuperscript{13} becoming the selected cutoff.

This method is fast and efficient, however it has a fatal flaw: Pictures taken with cellphones are often blurry and this makes the middle part of the histogram overrepresented. For an example of the resulting histogram where the phone was moved while the camera was taking the picture see figure 5. In that case the binarized line contains too much black, and decoding fails.

5.3.2 A New Binarizer

When the \textit{GlobalHistogramBinarizer} turned out to under-perform, creating a new Binarizer became a priority. Since the major flaw in the old Binarizer was that it would output results containing massive over-representations of black or white, it was reasoned that a binarizer that tested cutoff values until one with a reasonable resulting black-to-white ratio was found would be the best candidate for implementation.

In figure 6 the basic operation of this new \textit{BrightnessRefiningBinarizer} can be seen. It, using a binary search, traverses the grayscale range until a cutoff that results in a good black-to-white ratio has been found. The example given

\textsuperscript{13}Assuming the grayscale is 8 bit unsigned.
5.3.3 Implementation of BrightnessRefiningBinarizer

The implementation of the BrightnessRefiningBinarizer is fairly straightforward code-wise.

1. Load the line into a cache converting the line from color to grayscale. This step is a large performance booster, as it removes the need to convert from color to grayscale for every attempt.

2. Determine the proper cutoff.

3. Use the cutoff to finally binarize the given row.

```cpp
Ref<BitArray> BrightnessRefiningBinarizer::estimateBlackRow(int y, Ref<BitArray> row)
{
    prepareRowContainer(row);
    fillCache(y);
    findOptimalBlackCutoffForCache();
    binarizeCacheToRow(row);
    /*smoothSingleBits(row);
    #ifdef _DEBUG
    dumpRow(row);
    #endif
    return row;
}
```

For the reasoning around why smoothSingleBits is implemented but deactivated please see subsection 6.4 on page 28.

The findOptimalBlackCutoffForCache() function works like this:

1. Find the percentage difference between black and white using the middle cutoff.

2. If there is too much black reduce the cutoff.

3. Else if there is too much white increase the cutoff.

---

14 Again assuming that the grayscale is 8 bit unsigned.
4. If we are within limits, return, marking the current cutoff as optimal.

5. If there is still room to adjust, go to step 1.

6. Return, marking the current value as optimal, despite it being out of tolerance.

```cpp
void BrightnessRefiningBinarizer::findOptimalBlackCutoffForCache() {
  blackCutoff = UCHAR_MAX / 2;
  int stepsize = UCHAR_MAX / 4;

  const int percentTolerance = 10;

  do {
    int percentage = calcBlackToWhitePercentage();
    if (percentage > 100 + percentTolerance)
      blackCutoff -= stepsize;
    else if (percentage < 100 - percentTolerance)
      blackCutoff += stepsize;
    else
      return;
    stepsize /= 2;
  } while (stepsize != 1);
  return;
}
```

### 5.3.4 Performance

The complexities of the new binarizer compared to the old can be seen in table 2. These complexities look worrying, however testing has determined that the new binarizer is able to perform as quickly as needed.

### 5.4 Binarized format change in ZXing

The most extensive change made to ZXing was to change the way that binarized scanlines are stored in memory.

```// *** Old Storage ***
bit BitArray[ImageWidth];
```

<table>
<thead>
<tr>
<th>Operation</th>
<th>GlobalHistogramBinarizer</th>
<th>BrightnessRefiningBinarizer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prepare the target container</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>Filling cache</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Finding cutoff</td>
<td>n</td>
<td>n * log n</td>
</tr>
<tr>
<td>Final conversion</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

Table 2: Comparison of binarizer operation complexities.
Not only was the container changed, but the way ZXing thinks about binarized scanline was changed as well. Consider the following short scanline:

```
// Binarized scanline
00001111111100000011000011111100
// Old representation
0x0FF030FC
// New representation
4, 8, 6, 2, 4, 6, 2
```

The new representation stores not if the individual pixel was black or white, but instead how long the line of consecutive color was. This format consumes slightly more memory in this example, but it has three decisive advantages:

Firstly, the amount of memory needed to store a scanline of higher resolution does not increase much or at all. The amount of memory instead increases with the number of switches between black and white in the scanline, and this number remains constant no matter how large the resolution of the source image is.

Secondly, saving any function only interested in the length of consecutive color from having to recompute it every time. This saves large amounts of execution time for the functions attempting to match bar width to decoding tables. This is not only relevant when attempting to decode digits, but also when searching for the start and end patterns to the barcode.

Thirdly, this way of representing data allows any function accessing the stored data to access it using iterators, a much simpler and cleaner way of searching through the scanline. For an example of this see the function at subsection 5.4.2, it reads the representation and fills it into four counters for easy access.

For the sake of completeness the complexities for different actions are listed in table 3.

### 5.4.1 BitFlipArray iterator

This is the iterator object contained inside `BitFlipArray`. It is used as a shared object passed round between the decoding functions inside ZXing.

```cpp
class BitFlipArray {
    /* ... */
}```
<table>
<thead>
<tr>
<th>Operation</th>
<th>BitArray</th>
<th>BitFlipArray</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finding start and end patterns</td>
<td>$n^2$</td>
<td>$n$</td>
</tr>
<tr>
<td>Prepare to decode single digit</td>
<td>$n$</td>
<td>$c$</td>
</tr>
<tr>
<td>Create representation</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>Memory consumption (best case)</td>
<td>$n$</td>
<td>$c$</td>
</tr>
<tr>
<td>Memory consumption (worst case)</td>
<td>$n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

Table 3: Complexities for operations using different storage formats.

```java
public:
  class Iterator {
    public:
      Iterator(int startingIndex, BitFlipArray& array);
      BitRange getNextBlock();
      bool atEnd();
    private:
      unsigned int currentIndex;
      BitFlipArray& sourceArray;
  };

  Iterator getIteratorStaringAfter(BitRange& match);
}
```

5.4.2 Comparing functions using new binarized format

Presented here is an after and before comparison between two functions that complete the same task. One using the new representation, the other using the old representation. Both functions are used by the EAN-13 digit pattern matcher to fill a counter container with the barcode lengths to look up.

```java
void OneDReader::fillCountersFromIterator(BitFlipArray::Iterator& it, int counters[], int countersCount) {
  for (int i = 0; i < countersCount; i++) {
    counters[i] = it.getNextBlock().length;
  }
}

void OneDReader::recordPattern(Ref<BitArray> row, int start, int counters[], int countersCount) {
  int numCounters = countersCount;
  for (int i = 0; i < numCounters; i++) {
    counters[i] = 0;
  }
  int end = row->getSize();
}  
```
```cpp
if (start >= end) {
    throw ReaderException("recordPattern: start >= end");
}

bool isWhite = row->get(start);
int counterPosition = 0;
int i = start;
while (i < end) {
    bool pixel = row->get(i);
    if (pixel ^ isWhite) { // that is, exactly one is true
        counters[counterPosition]++;
    } else {
        counterPosition++;
        if (counterPosition == numCounters) {
            break;
        } else {
            counters[counterPosition] = 1;
            isWhite ^= true; // isWhite = !isWhite;
        }
    }
    i++;
}
// If we read fully the last section of pixels and filed up our counters -- or filled
// the last counter but ran off the side of the image, OK. Otherwise, a problem.
if (!(counterPosition == numCounters ||
     (counterPosition == numCounters - 1 && i == end))) {
    throw ReaderException("recordPattern");
}
```

### 5.5 Testing

The project plan called for automated unit testing and the framework of choice became *SymbianOSUnit*, as it was the only noncommercial C++ testing framework for Symbian that was found.

While initially promising, a bug was found in the framework that caused the provided ASSERT functions to leak memory the first time they were used which resulted in the first test case always failing due to memory leaked. The following test case was used as a workaround to trigger this bug, and let all the following test cases complete normally.

```cpp
void CTest::testSkipMemoryLeak() {
    TS_ASSERT_DIFFERS(1, 2);
    TS_ASSERT_EQUALS(1, 1);
    TS_ASSERT_DIFFERS(1, 2);
    TS_ASSERT(1 == 1);
}
```
This workaround was only partially effective, as the logs always had be manually reviewed to see if any additional errors occurred during testing.

A larger problem to testing was that automated testing of a barcode decoder proved largely futile during the course of this project. The plan was to have a large amount of pictures containing barcodes stored on the phone’s memory card, and when testing run all the pictures through the decoder and output how large a percentage that were successfully decoded. While good in theory this plan was defeated by several factors: firstly evaluating the results was really hard. Seven barcodes were available to test with, so if the decode succeeded on 2/7 of all pictures it had either decoded 2/7 of the pictures of all the barcodes, or decoded all pictures of two barcodes. Determining which had occurred required manual evaluation, thus defeating the purpose of automated testing.

Secondly, loading images from the phone’s SD-card was surprisingly slow, and resulted in tests with many pictures taking too long to complete. The tests had to be run from the actual phone and could not be run on the emulator for technical reasons. When the number of pictures was scaled down to 12 in order to combat this problem the tests became inconclusive and again required manual evaluation.

Eventually automated testing of the decoder was abandoned, and instead a fully manual test was devised. A trained operator would start a clock, and then as quickly as possible scan all the seven barcodes. This provided a usable metric for the decoder’s efficiency and allowed changes to be evaluated relatively quickly.

5.6 Integration

5.6.1 Moving of functionality inside BarcodeScannerViewfinderView

The largest problem when adapting the viewfinder view to use the ZXing decoder was that many core functions in the view were controlled by system reserved functions. For example, the code to start the viewfinder was inside the `ScreenSizeChanged()` function. This had the unfortunate side effect of crashing the application if the phone was tilted while attempting to decode an image.

To mitigate this and to simplify any future work on this module functionality was moved to a few properly named functions.

```c
void SetupCamera();
void StartLiveCapturing();
void StopLiveCapturing();
void SnapFullsizePicture();
void CaptureFullsizePictureASAP();
```

--

15 The viewfinder is a preview screen that is continuously updated.
16 This function is called when the view is activated and when the screen changes orientation.
6 Discussion

This section discusses the design and the considerations that shaped the barcode decoder. Note that only the barcode decoder is discussed here. Changes to the rest of the application are motivated partly in the Background section on page 9 and the Requirement section on page 12, or are discussed extensively in the referenced literature.

6.1 Feature-Limiting

The ZXing barcode decoder is a complicated piece of software, and is able to interpret a wide array of barcodes. The ZXing decoder is also able to decode not only one-dimensional but also two-dimensional barcodes\(^\text{17}\) as well. During this project all decoders but EAN-13 and EAN-8 have been disabled.

This is for the simple reason that looking for any type of barcode, be it one or two-dimensional, uses processing power. In the case of two-dimensional barcodes, a large computational investment is required: it is not sufficient to only search a single scanline, the entire image has to be searched for key patterns. Previously, the ZXing decoder was geared towards using single high-resolution images known to contain a barcode. It was worth having this code active as it would be run only once. In most cases it was never run, as a more likely decoder, like EAN, had already found a barcode.

Now, however, the decoder relies on attempting to decode several pictures a second and most of these decodings failing is an expected outcome. Remember that if a barcode was not found by one of the first decoders, ZXing will try all decoders in the order of most common codes. This means that it takes significantly longer to fail to decode a barcode then it takes to succeed.

The code that reads other formats has however been kept reasonably intact, and can with small effort be re-enabled to allow ZXing to once again decode less common barcodes like Code 39\(^\text{18}\).

6.2 Scanlines

To improve the chance of a successful decoding, several scanlines are used when decoding an image, as shown in figure 7.

\(^{17}\)Previously referred to as matrix codes, the term is seldom used nowadays due to confusion with the Matrix franchise.

\(^{18}\)An old barcode symbology rarely used for consumer goods.
To remove clutter, the *RowSelector* helper object was created to keep track of which lines were to become scanlines. The ported decode function is presented below:

```cpp
Ref<Result> OneDReader::doDecode(Ref<BinaryBitmap> image) {
    Ref<BitArray> row(new BitArray(image->getWidth()));

    RowSelector rowsToDecode(image->getHeight(), 9);

    while (rowsToDecode.rowsLeftToProcess()) {
        int rowNumber = rowsToDecode.getNextRow();

        // Convert row from image to black&white
        try {
            row = image->getBlackRow(rowNumber, row);
        } catch (ReaderException re) {
            continue;
        }

        // Attempt decoding
        try {
            Ref<Result> result = decodeRow(rowNumber, row);
            return result;
        } catch (ReaderException re) {}  

        // Perhaps code is upside down?
        row->reverse();
        try {
            Ref<Result> result = decodeRow(rowNumber, row);
            return result;
        } catch (ReaderException re) {}  
    }

    throw ReaderException("OneDReader::doDecode failed");
}
```

The *RowSelector* starts from the center most line and then, in an alternating manner selects a line above or below the center line. This behavior allows the
decoder to take advantage of image distortions that might be present.

For examples see figures 8 and 9, here more pixels are under scanlines then if a single-scanline setup had been used and this increases the effective resolution while increasing the distance the barcode can be held from the phone. Most phones do not feature analog zoom and need a decimeter or more of distance from the barcode before they are able to capture a sharp image.

![Figure 8: A scanline that is at an angle to the barcode](image)

![Figure 9: A scanline that is sweeping an angled barcode](image)

The design choice of several lines has most likely come from the ZXing decoder’s previous Modus Operandi: working on a single high resolution image and emphasizing decoding hit-rate over speed. This design choice has however come into question. If our goal is to decode as many frames per second of moving video, why attempt to decode the same image several times?

One of the largest factors impacting real-time decoding speed\(^\text{19}\) is how sharp the image of the barcode is. As the barcode is fairly small it is reasonable to assume that if part of the barcode is out of focus, then the entire barcode is out of focus. Meaning that using multiple lines only slightly increases the hit-rate but multiplies the time it takes to attempt to decode and image.

The number of scanlines to be used need to be adapted to the number of frames per second produced by the capturing device and how many frames per second the phone can attempt to decode. It could be argued that this decoder should use only a single scanline, minimizing the time per image and instead relying on being able to process images with great speed.

\(^{19}\)Average attempts per barcode multiplied by time to decode single image.
This has not been explored in this project, as decoding speed and accuracy are considered to be well within acceptable limits. It might however become a relevant and simple modification if speed or accuracy need to be traded in the future.

6.3 Parallelizing

As seen in figure 7 on page 26, there are several parallel lines scanned to attempt to find the barcode. Many software engineers first reaction is that this is a good place to parallelize the algorithm. That is, have several threads each working on its own line and thereby speeding up decoding significantly. Parallelizing this code however has only negative impact.

While most modern cellphones use two processors, one x68 for the OS and one ARM for GSM, only the x68 processor is available to the operating system. Therefore, it is impossible to speed up decoding by using multiple threads. If we additionally assume that all threads receive equal time-slots, the time to decode the barcode will increase as the number of threads increases. This is illustrated in figure 10 for parallel execution and in figure 11 for serial execution.

As seen in these execution graphs, both the threaded and the unthreaded methods take equally long to complete. However, there is a good chance that the barcode was decoded using only the middlemost line and therefore not needing to even attempt to decode any other lines. A parallel implementation however will wait for all, or most, decodings to finish before finding out if one succeeded. Additionally, a parallel implementation would have to use much larger amounts of memory, as each thread needs its own buffers, and many more dynamic allocations, as memory cannot be reused between lines.

For these reasons the choice was made to not parallelize decoding, despite potential future payoff for future cellphones with multiple x86 cores.

6.4 Smoothing

Sometimes, when attempting to binarize the picture of a barcode a tiny, often only one or two pixels wide, region inside a black strip might be identified as white, or vice versa. This phenomenon is a bit hard to explain, so please study
By attempting to number the relative lengths of the bars, the upper barcode yields the sequence 1-4-1-1, which can be looked this up in the LG-table (Appendix 8.1 on page 33) and identified as the number 3. If however the lengths in the lower barcode are measured it gives us the sequence 10-9-1-30 which cannot remotely be matched to an entry in the table, hence decoding fails. This problem has always been present in Binarizers, however it has been worsened significantly by increasing the color depth used by the Binarizer. Three possible solutions to this problem were found:

The first is to smooth the bit-string after binarizing, removing any small anomalies. This approach has several advantages; There is no need to modify any modules in the decoding chain; It works in linear time and it is fairly simple to implement. Due to its advantages this method was implemented. However, after testing several severe problems were identified, where the biggest is that it is very hard to define how small an anomaly needs to be for smoothing out. A smoother needs to be able to handle the following scenarios:

1. A small image of which 90% is barcode.
2. A large image of which 90% is barcode.
3. A large image of which 10% is barcode.
4. The image very dark or very bright.

Several methods to identify incorrect sections were tested:

A. Any bar shorter then X% of the image width is wrong.

For an explanation of this notation please see subsection 3.4 on page 10.

The old Binarizer used 16 buckets. This made it fairly unlikely that two adjacent pixels would be sufficiently different to end up in different buckets. The new Binarizer effectively uses 256 buckets, drastically increasing the chance that two similar pixels end up colored differently [see subsection 5.3 on page 17].

The pattern matcher is generally permissive with this, and will allow for example the pattern 3-11-3-2 to match 1-4-1-1.
Any bar shorter then X pixels is wrong.

Any bar where its neighbors are five times longer then itself is wrong.

All methods tested failed to properly smooth one or more of the scenarios, as seen in table 4.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method A</td>
<td>FAIL</td>
<td>FAIL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Method B</td>
<td>FAIL</td>
<td>FAIL</td>
<td>FAIL</td>
<td></td>
</tr>
<tr>
<td>Method C</td>
<td>FAIL</td>
<td>FAIL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Smoothing methods tested for different scenarios.

The method C would be able to smooth well if it knew fairly exactly where the barcode started and ended. Any implementation first searching for the barcode and then smoothing would however run in quadratic or even cubic time, and would decrease performance significantly.

The second possible solution is to have the binarizer use a bias when binarizing. Assume that the pixels along a line are binarized from left to right. A biased binarizer would darken any pixel that is preceded by a back pixel, and lighten any pixel that is preceded by a white pixel when decoding the line. This approach is most likely the best solution to this problem. However, it has two major problems: Firstly, it has a tendency to, when the picture is a bit too dark or a bit too bright, flatten the entire image with a single color; Secondly it is not predictive. This means that if given the same picture twice, one a little darker that the other, the darker one could end up having more white then the brighter one. The first problem could be worked around with some parameter tweaking and minor modifications. However, the second one is in this application insurmountable as the BrightnessRefiningBinarizer (see subsection 5.3 on page 17) uses binary searching, depending on predictable behavior from the smoother.

The third solution is to take several pictures and combine these into one, eliminating any irregularities. This is a well tested approach for creating sharp and accurate images from blurry and noisy photos. However, it has several weak points that make it unsuitable for this application. For this method to work, the images need to be taken in very rapid succession or from a stationary camera photographing a reasonably stationary unchanging object. This approach is generally unsuited for mobile applications, as neither the camera nor the barcode stay very still and the camera is not able to capture images at the required rate.

The fourth solution is the simplest one, and the one that is deployed in this application. Generally, the lower the resolution of the image, the less the chance to get a stray line, as there are shorter strips. Additionally, the lower the resolution is the faster each image can be processed. Since the new decoder operates mostly on 240 pixel wide preview images, stray lines are seldom seen. If they occur they do not matter, as they are not present on the next picture.
7 Result

7.1 Requirements

The resulting application has been determined to meet all requirements by the sponsor.

7.2 Goals

Both the responsiveness and the accuracy of the barcode scanner have been improved to the point where the application could be used by a user.

7.3 Screenshot

In figure 13 the barcode scanning view can be seen in action. Note the horizontal red line assisting the user in placing the camera correctly.

Figure 13: A screenshot of the barcode scanning view
References


8 Appendix

8.1 EAN-13 Decoding Tables

8.1.1 First digit code table

<table>
<thead>
<tr>
<th>First Digit</th>
<th>First Digit Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LLLLLL</td>
</tr>
<tr>
<td>1</td>
<td>LLGLGG</td>
</tr>
<tr>
<td>2</td>
<td>LLGGLG</td>
</tr>
<tr>
<td>3</td>
<td>LLGG GL</td>
</tr>
<tr>
<td>4</td>
<td>LGGLGG</td>
</tr>
<tr>
<td>5</td>
<td>LGGGLG</td>
</tr>
<tr>
<td>6</td>
<td>LGGGGL</td>
</tr>
<tr>
<td>7</td>
<td>LGLGLG</td>
</tr>
<tr>
<td>8</td>
<td>LGLGGL</td>
</tr>
<tr>
<td>9</td>
<td>LGGLGL</td>
</tr>
</tbody>
</table>

8.1.2 First digit group decoding table

<table>
<thead>
<tr>
<th>Digit</th>
<th>L-code</th>
<th>G-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3 2 1 1</td>
<td>1 1 2 3</td>
</tr>
<tr>
<td>1</td>
<td>2 2 2 1</td>
<td>1 2 2 2</td>
</tr>
<tr>
<td>2</td>
<td>1 2 2 2</td>
<td>2 2 1 2</td>
</tr>
<tr>
<td>3</td>
<td>3 4 1 1</td>
<td>1 1 4 1</td>
</tr>
<tr>
<td>4</td>
<td>4 1 3 2</td>
<td>2 3 1 1</td>
</tr>
<tr>
<td>5</td>
<td>1 2 3 1</td>
<td>1 3 2 1</td>
</tr>
<tr>
<td>6</td>
<td>1 1 4 4</td>
<td>4 1 1 1</td>
</tr>
<tr>
<td>7</td>
<td>1 3 1 2</td>
<td>2 1 3 1</td>
</tr>
<tr>
<td>8</td>
<td>3 2 1 3</td>
<td>3 1 2 1</td>
</tr>
<tr>
<td>9</td>
<td>3 1 1 2</td>
<td>2 1 1 3</td>
</tr>
</tbody>
</table>

8.1.3 Second digit group decoding table

<table>
<thead>
<tr>
<th>Digit</th>
<th>R-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3 2 1 1</td>
</tr>
<tr>
<td>1</td>
<td>2 2 2 1</td>
</tr>
<tr>
<td>2</td>
<td>2 1 2 2</td>
</tr>
<tr>
<td>3</td>
<td>1 4 1 1</td>
</tr>
<tr>
<td>4</td>
<td>1 1 3 2</td>
</tr>
<tr>
<td>5</td>
<td>1 2 3 1</td>
</tr>
<tr>
<td>6</td>
<td>1 1 4 4</td>
</tr>
<tr>
<td>7</td>
<td>1 3 1 2</td>
</tr>
<tr>
<td>8</td>
<td>1 2 1 3</td>
</tr>
<tr>
<td>9</td>
<td>3 1 1 2</td>
</tr>
</tbody>
</table>
På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Niklas Hayer