

Contents

- 1 Getting started
- 2 Algorithms
- 3 Optimization
- 4 Source code
- 5 Test application

Getting started

If you don't have much knowledge of MPEG, this is a good place to start:

<http://en.wikipedia.org/wiki/MPEG-1>

Also the ISO/IEC 11172-2 is an excellent reference. Unfortunately you have to pay for it:

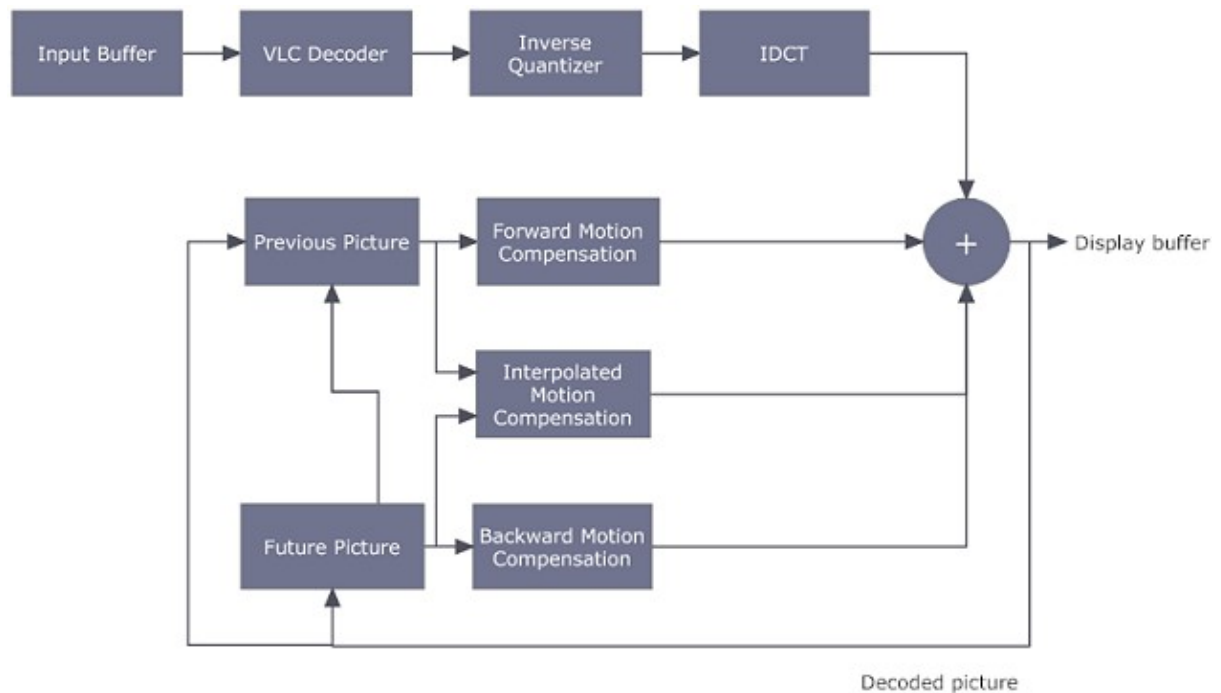
http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22411

The standard is divided in five sections. For this project using section 2 (Video) will be enough.

Algorithms

Here's an overview of the decoding process, and the classes involved.

MPEG_decoder_in_Java_ME



InputStream.java: The video stream is a bit stream, so we need to be able to retrieve an arbitrary amount of bits. For example, the Sequence Header contains the width and height of the frame, both 12-bits quantities. Some of the fields are variable length, so this class is used heavily by the VLC decoder.

```

private void parseSequenceHeader() throws IOException {
    int sequenceHeaderCode = mInput.getBits(32);

    mWidth = mInput.getBits(12);
    mHeight = mInput.getBits(12);
    // ...
}

```

Vlc.java: As already mentioned, some fields are coded using Huffman variable length codes (ie. frequent items are represented by shorter codes). The problem here is determining how long each code is. The easiest (and naïve) approach would be reading a bit at a time, but needless to say, it is not performance wise, as it involves multiple shifts and masking each time.

The approach used here is making use of tables. As an example, let's consider the variable length code for dct_dc_size luminance (Table 2-B.5a in ISO/IEC 11172):

VLC code	dct_dc_size_luminance
100	0
00	1
01	2
101	3
110	4
1110	5
11110	6
111110	7
1111110	8

MPEG_decoder_in_Java_ME

The strategy here is *peeking* 7 bits (the longest variable length code) and using it as an index in the following table:

index	value	length
100xxxx	0	3
00xxxxx	1	2
01xxxxx	2	2
101xxxx	3	3
110xxxx	4	3
1110xxx	5	4
11110xx	6	5
111110x	7	6
1111110	8	7

So when reading a bit sequence like "1110101" we know this is the "1110" code, whose value is 5 and length 4 bits. With this information we can remove these bits from the stream and proceed.

A problem with this approach can also be seen from the example above. We need an array of 128 items (each one holding value and length) for a table of 9 elements. That's a lot of redundancy (about 14:1). A way to deal with this is using hierarchical tables. We choose a convenient value and create an extra level. For example, let's discard the 3 least significant bits and use the remaining bits as an index. This will leave us with two tables: one with 16 entries (2^4) and the other one with 8 entries (2^3). The last entry of the first table will contain a *escape code* that will indicate we should use the other table. Now we have $16 + 8 = 24$ entries which yields a much better 3:1 ratio, at the expense of some extra computation.

Here's the code:

```
// Each byte consists of a value|length pair
private static final short[] dct_dc_size_luminance = {
    0x12, 0x12, 0x12, 0x12, 0x22, 0x22, 0x22, 0x22,
    0x03, 0x03, 0x33, 0x33, 0x43, 0x43, 0x54, 0x00
};

private static final short[] dct_dc_size_luminance1 = {
    0x65, 0x65, 0x65, 0x65, 0x76, 0x76, 0x87, 0x00
};

public int decodeDCTDCSizeLuminance(InputBitStream input) throws IOException {
    int index = input.nextBits(7);
    int value = dct_dc_size_luminance[index >> 3];

    if (value == 0)
        value = dct_dc_size_luminance1[index & 0x07];

    input.getBits(value & 0xf);

    return value >> 4;
}
```

Idct.java: MPEG uses 8x8 DCT to convert data in time domain to data in frequency domain. This is a critical part as this code is called everytime a block is decoded. Use of fixed point math is made both for efficiency and due to constraints in CLDC 1.0

The equation for the 2D IDCT is

$$f(x,y) = \frac{2}{\sqrt{MN}} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} C(m)C(n)F(m,n) \cos \frac{(2x+1)m\pi}{2M} \cos \frac{(2y+1)n\pi}{2N}$$

$$\begin{cases} C(m), C(n) = \frac{1}{\sqrt{2}} & m, n = 0 \\ C(m), C(n) = 1 & \text{otherwise} \end{cases}$$

where $F(m,n)$ is the DCT of the signal $f(x,y)$ and $M = N = 8$

MotionVector.java: used to represent and calculate the forward and backward motion vectors.

Picture.java: A picture consists of three rectangular matrices of eight-bit numbers; a luminance matrix (Y), and two chrominance matrices (Cr and Cb). This class also provides functions for copy, compensation and interpolation of blocks. Each pel is an *unsigned* 8 bit sample. Unfortunately Java doesn't support such type, so a *short* type is used instead.

Bitmap.java: This class performs the conversion from Y'CbCr 4:2:0 to RGB.

```

/*
 * A Bitmap stores a video frame ready to be displayed
 */
public class Bitmap {
    private int mWidth;
    private int mHeight;

    public int[] mRgb;

    public Bitmap(int width, int height) {
        mWidth = width;
        mHeight = height;

        mRgb = new int[mWidth * mHeight];
    }

    private final int C1 = 0x166E9; // 1.402 * 2^16
    private final int C2 = 0x5819; // 0.34414 * 2^16
    private final int C3 = 0xB6D1; // 0.71414 * 2^16
    private final int C4 = 0x1C5A1; // 1.772 * 2^16

    /*
     * Perform Y'CbCr 4:2:0 to RGB conversion
     */
    public void transform(Picture picture) {
        // We process two lines at a time
        int size = (mWidth * mHeight) >>> 2;

        int index1 = 0; // First luma line
        int index2 = mWidth; // Second luma line

        for (int i = 0; i < size; ++i) {
            int cb = picture.mCb[i] - 128;
            int cr = picture.mCr[i] - 128;

            int c1cr = C1 * cr;
            int c2cb = C2 * cb;
            int c3cr = C3 * cr;

```

MPEG_decoder_in_Java_ME

```
int c4cb = C4 * cb;

/*
 * Apply CbCr to four neighboring luma samples
 */
for (int j = 0; j < 2; ++j) {
    int y = picture.mY[index1] << 16;    // 2^16

    int r = y + c1cr;
    int g = y - c2cb - c3cr;
    int b = y + c4cb;

    // Clamp rgb values into [0-255]
    b    >>= 16;
    b    = b > 0xff? 0xff : b < 0? 0 : b & 0x000000ff;

    g    >>= 8;
    g    = g > 0xff00? 0xff00 : g < 0? 0 : g & 0x0000ff00;

    r    = r > 0xff0000? 0xff0000 : r < 0? 0 : r & 0x00ff0000;

    mRgb [index1++] = (r | g | b);

    y    = picture.mY[index2] << 16;    // 2^16

    r    = y + c1cr;
    g    = y - c2cb - c3cr;
    b    = y + c4cb;

    // Clamp rgb values into [0-255]
    b    >>= 16;
    b    = b > 0xff? 0xff : b < 0? 0 : b & 0x000000ff;

    g    >>= 8;
    g    = g > 0xff00? 0xff00 : g < 0? 0 : g & 0x0000ff00;

    r    = r > 0xff0000? 0xff0000 : r < 0? 0 : r & 0x00ff0000;

    mRgb [index2++] = (r | g | b);
}

// Next two lines
if (index1 % mWidth == 0) {
    index1    += mWidth;
    index2    += mWidth;
}
}
}
```

Queue.java: A straight forward unbounded queue, used as a communication between the decoder (producer) and renderer (consumer) threads.

Decoder.java: The decoder itself, as described by the ISO/IEC 11172-2 standard. Here's an excerpt:

```
public void start() throws IOException {
    nextStartCode();

    /*
     * A video sequence starts with a sequence header and is
     * followed by one or more groups of pictures and is ended
```

MPEG_decoder_in_Java_ME

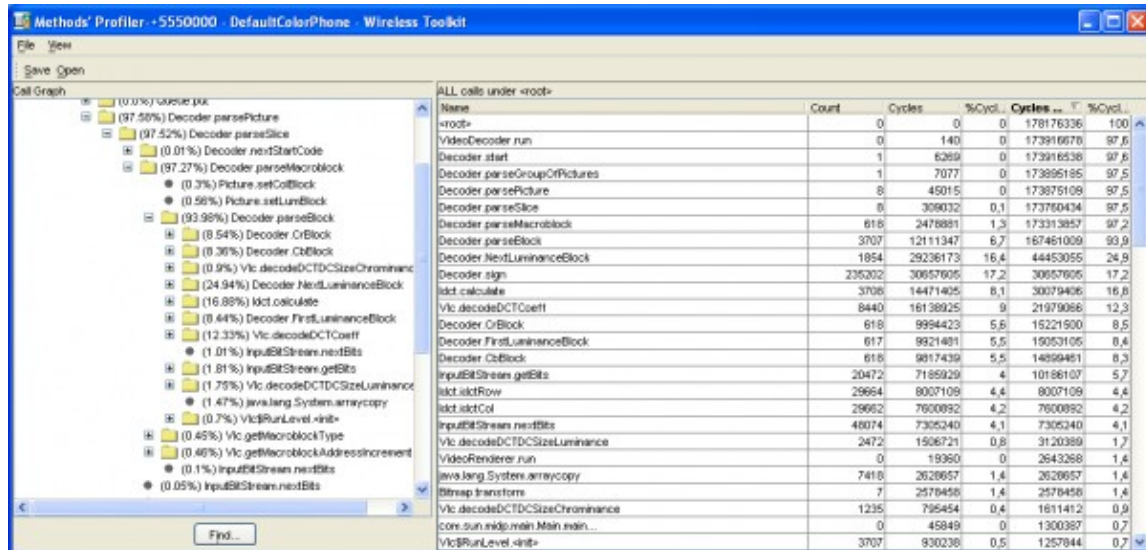
```
* by a SEQUENCE_END_CODE. Immediately before each of the  
* groups of pictures there may be a sequence header.  
*/
```

```
do {  
    parseSequenceHeader();  
  
    mRenderer.setSize(mWidth, mHeight);  
  
    mPictureStore[0] = new Picture(mMacroblockWidth, mMacroblockHeight);  
    mPictureStore[1] = new Picture(mMacroblockWidth, mMacroblockHeight);  
    mPictureStore[2] = new Picture(mMacroblockWidth, mMacroblockHeight);  
  
    do {  
        parseGroupOfPictures();  
    } while (mInput.nextBits(32) == GROUP_START_CODE);  
  
} while (mInput.nextBits(32) == SEQUENCE_HEADER_CODE);  
  
int sequenceEndCode = mInput.getBits(32);  
}
```

Player.java: A very simple MIDlet used for example purposes. It takes care of displaying the frames in display order using a very simple approach (other codecs would require a proper Playout Buffer). No other task is performed.

Optimization

There's certainly plenty of room for optimization. The aim was making the code as clear as possible, without ignoring basic performance issues. The following screen shows where the bottlenecks are:



Some possible optimizations:

- IDCT: handle special cases (ie. single coefficient)
- YCbCr-RGB: use table driven multiplication

MPEG_decoder_in_Java_ME

- General Java optimizations: loop unrolling, avoid use of casts, object reutilization, etc.

The decoder was tested on a Nokia 7610, getting 4-5 fps for a 160x120 video.

Source code

Download [source code](#)

Test application

Download [test application](#)