An Investigation into the Possibilities of Implementing OGG Vorbis in the J2ME MIDP Environment

Daniel Nilsson

October 1, 2008
Master's Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Thomas Johansson
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

The purpose of this Master Thesis is to investigate the possibilities of implementing an OGG Vorbis decoder using Java ME. The primary investigation was to implement a decoder from scratch using the specifications available for OGG encapsulation and Vorbis Audio Format. This was the primary choice since Vimio, the company that the Master Thesis was done at, wanted a decoder that they were able to use without any licensing problems. Once the decoder was implemented it showed that it was not able to decode at a reasonable speed. Some optimization was done but, since floating operation in Java ME is really slow, it did not improve the decoder that much.

An attempt at implementing an Integer-based solution was started since the major time waste was the mathematical operations used on float numbers. But since the time ran out the implementation of the Integer-based solution was left unfinished.

An Investigation into the Possibilities of Implementing OGG Vorbis in the J2ME MIDP Environment

Contents

1 Introduction **1**

2 Problem Description **3**

2.1 Problem Statement 3

2.2 Goals and Purposes 3

2.3 Methods 3

2.4 Related Work 3

3 Compression techniques **5**

3.1 Huffman entropy coding algorithm 5

3.1.1 Encoding 5

3.1.2 Decoding 6

3.2 Discrete Cosine Transform 9

3.2.1 Modified Discrete Cosine Transform 9

3.2.2 Inverse Modified Discrete Cosine Transform 9

4 Formats **11**

4.1 The OGG encapsulation format 11

4.2 The Vorbis Audio format 12

4.2.1 Identification header 12

4.2.2 Comment header 12

4.2.3 Setup header 12

4.2.4 Audio packet 13

5 Implementation **17**

6 Platform **21**

6.1 Java ME 21

6.2 MIDP 21

7 Results **23**

8 Conclusions **25**
8.1 Future work 25

9 Acknowledgements **27**

References **29**

List of Figures

3.1 Huffman decision tree example 7

4.1 Equal size window overlap add. Output will be from middle of first to middle of second. 15

4.2 Unequal size window overlap add. Output will be from middle of first to middle of second. 16

List of Tables

3.1 Huffman object and frequency table 5
3.2 Huffman decoding example table 8

Chapter 1

Introduction

During the last couple of years the mobile phone has gone from a single purpose handheld device to a multi-functional device handling calls, playing media, taking pictures, surfing the internet, etc. The mobile phone as a media device handles both music and video. In these kinds of media there are many different media formats delivering sound and video at different quality. One of the formats for playing sound is the Vorbis format. This format is designed by a foundation and is free from patent and royalties. The format has a better compression then for example MP3 which means that for the same file size you get better audio with Vorbis then with MP3.

Since this compression format seems to be a format being used more and more, it could be interesting to examine the possibilities of decoding OGG Vorbis songs using a mobile phone. In order to have a solution that works in all the different devices the decoder needs to be implemented in Java ME. But is this possible? Many of the decoders for the mainstream media formats has problems when using Java in low memory devices. The purpose of this master thesis is to examine if an implementation to decode OGG Vorbis audio can be used with J2ME. A reason why this audio format is interesting is that there is a lot of content available in OGG Vorbis format and if a decoder could be implemented, the content could be used without re-encoding to another format which would save alot of time and space and the quality of the content would stay the same.

This work is done at Vimio AB in Umeå. Vimio is a company that develops mobile telecom applications and mobile service design. Among the products that are currently designed live TV and jukebox services for mobile phone use is perhaps the most interesting.

The report will continue by first describe the problem at hand and the goals and methods for this thesis. The chapter after that will be dedicated for explaining the theories that are necessary for understanding the techniques that the OGG Vorbis codec is using. Once the techniques are explained the next chapter will discuss the implementation of the decoder and how the implementation were planned and executed. The fifth chapter will show the result of the thesis. Any restrictions and limitations that where made from the original specification will be presented in the conclusion chapter where also future work and some discussion about the achievements of the thesis.

Chapter 2

Problem Description

2.1 Problem Statement

The problem is to evaluate the possibilities for a J2ME implementation of the OGG Vorbis decoder that can be used to decode audio in real-time using very limited amount of memory.

2.2 Goals and Purposes

The goal and purpose of this thesis is to evaluate if an OGG Vorbis decoder using J2ME is possible to implement and to show what limitations if any it has.

2.3 Methods

In order to examine the possibilities of implementing the decoder in J2ME a decoder must be implemented according to the specification availble at xiph.org and tested. To understand how the decoder works information must be gathered about the OGG Vorbis audio format and the J2ME library. With this knowledge a decoder design is done and then the implementation begins. If a decoder is implemented successfully and if possible some optimization of memory usage and execution time is done the decoder will be tested for memory usage and execution time. The definition of a usable decoder should be a decoder that decodes the audio in real-time with minimum usage of memory.

2.4 Related Work

There is a java based decoder called Jorbis. This decoder is not created with concerns about memory limitations and processor power and is therefore not of high interest in this thesis. An integer based solution of the decoder is available at xiph.org. It is called Tremor low-memory branch reference decoder and it is implemented in C. This could be used and modified within certain license restrictions.
There is a media library for Java ME called MMAPI, but this library do not support Vorbis and the usability of this library is not that good.

Chapter 3

Compression techniques

The advantage sound compression format such as MP3 and OGG Vorbis have against uncompressed sound is that the same sound quality can be compressed to a file that is smaller in size then a uncompressed sound file would be. The compression techniques that are interesting regarding the OGG Vorbis decoder are explained in the following sections.

3.1 Huffman entropy coding algorithm

The Huffman entropy coding algorithm is used for lossless data compression. The input to the algorithm is a stream of objects and the output is a list of bitstrings that represents the codeword for each unique object. The reason of using the Huffman entropy encoding algorithm is that it is the most efficient compression method available for representing reoccuring objects in a stream[12]. No other kind of mapping of different strings will generate a shorter output size. If the symbols are sorted according to frequency the algorithm works in linear time. This algorithm is used very widely for example with the jpeg image format and mpeg layer 3 (MP3) music format. [12] [1]

3.1.1 Encoding

The encoding process starts with determining the frequency of occurrence of each object in the stream. This is done by counting the number of times each object is used in the stream. For better understanding the encoding process a step by step encoding example will be used with the objects and frequencys according to table 3.1.1. Once the frequency of each object is counted the way to determine the codeword for each object is

Object	*Frequency*
A	35
B	7
C	65
D	12
E	18

Table 3.1: Huffman object and frequency table

to construct a binary tree called *Decisiontree*. The construction of the *Decisiontree* has a inital step of making all objects into leaf nodes. The first step of constuction is to place the leaf nodes in ascending order according to frequency. The rest of the construction is to combine the two subtrees with the lowest frequency to a new subtree by adding a node and placing the two subtrees as the children of the node. The new frequency of this subtree is set to the sum of the frequencys of the trees and the subtree is then placed in the right place in the list. This step will be repeated until there is just one subtree left. If the number of unique objects in the stream is N the number of combinding steps will be $N - 1$. In our encoding example the construction of the *Decisiontree* will be completed in four steps. The entire construction of the *Decisiontree* is showed in Figure 3.1. In the initial step of the *decisiontree* construction in Figure 3.1 the objects $A - E$ are made into leaf nodes. Step one sorts the subtrees in a ascending order according to their frequency. Then the steps two to five takes the two subtrees with the smallest frequency and constructs a new subtree of them which is inserted in the right place according to its calculated frequency. The fifth step only contains one subtree thus the *Decisiontree* is completed and stored for decoding.

3.1.2 Decoding

The decoding process uses a bitstream as input. The decoder uses the decisiontree built in the encoder to extract the objects. The first step of decoding is to set the root of the tree as current node. The decoder then reads a bit from the stream and selects the left child node as current node if the bit is a zero and the right child node if the bit is 1. This step is then repeated until the current node is a leaf node. When this happen the decoder returns the object represented by that leaf and starts over from step one. This process is repeated until all the bits in the stream are read. When the whole bitstream is read the decoding process is completed. For example if the input bitstream for the decoder is 11111100101111 and the *Decisiontree* used is the one returned from the example in figure 3.1 the output would be the objects D, E, C, A, D. A step-by-step walkthrough of this example is showed in table 3.1.2.

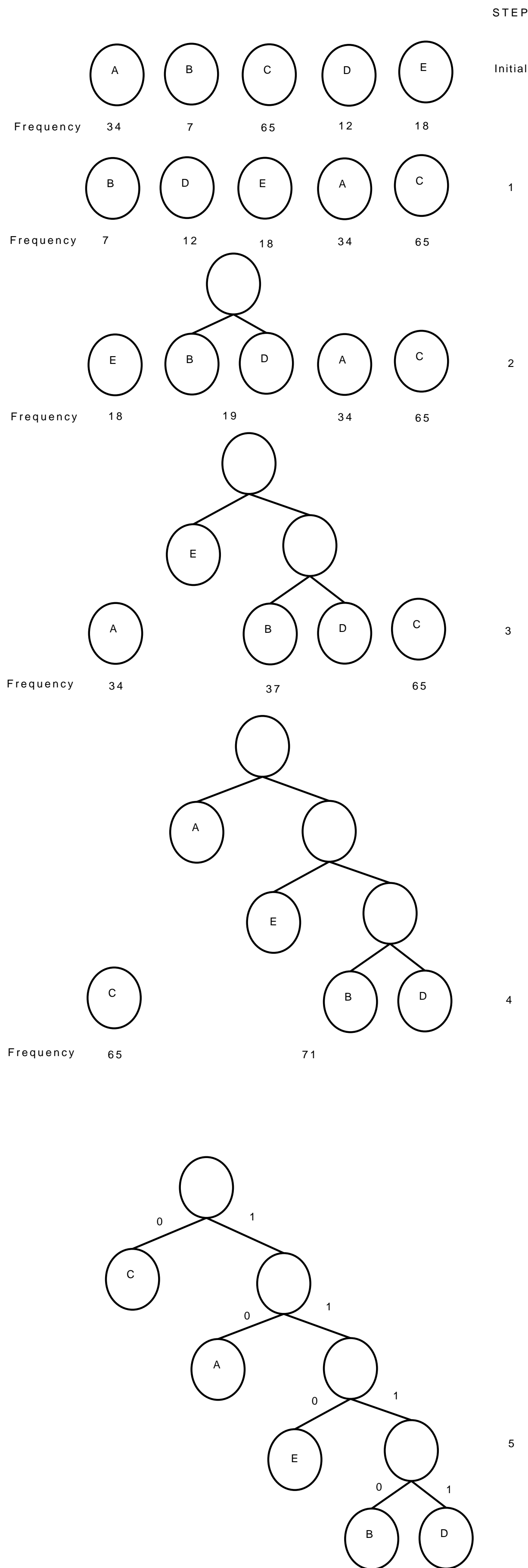

Figure 3.1: Huffman decision tree example

Decoder action	Move to node	Remaining stream
Init	Root	11111100101111
Read 1	Right child	1111100101111
Read 1	Right child	111100101111
Read 1	Right child	11100101111
Read 1	Right child	1100101111
Return D	Root	1100101111
Read 1	Right child	100101111
Read 1	Right child	00101111
Read 0	Left child	0101111
Return E	Root	0101111
Read 0	Left child	101111
Return C	Root	101111
Read 1	Right child	01111
Read 0	Left child	1111
Return A	Root	1111
Read 1	Right child	111
Read 1	Right child	11
Read 1	Right child	1
Read 1	Right child	empty
Return D	Root	empty
Finished		

Table 3.2: Huffman decoding example table

3.2 Discrete Cosine Transform

Discrete Cosine Transform (DCT) is a technique for converting a signal into elementary frequency components. It is often used for lossy data compression which means compressions that generate data not equal to the original, but close enough to be useful. This kind of data compression is often used in image and audio compression such as JPEG and MPEG. There are seven different forms of the DCT called DCT-1 to DCT-7 where DCT-2 is the far most popular for compression and the DCT-3 is the inverse function of the DCT-2. [11]
The formal definition of the DCT-2 can be seen below
the N real numbers $x_0..x_{N-1}$ are transformed to the N real numbers $X_0..X_{N-1}$. If $k = 0..N-1$ the equation 3.1 makes the transformation.

$$X_k = \sum_{n=0}^{N-1} x_n cos(\frac{\pi}{N}(n + \frac{1}{2})k)) \tag{3.1}$$

To inverse the DCT-2 transform and restore an approximation of the original signal the transformation formula DCT-3 is used. The definition for DCT-3 is as follows [11]:
The N real numbers $X_0..X_{N-1}$ are transformed to the N real numbers $x_0..x_{N-1}$. If $k = 0..N-1$ the equation 3.2 makes the transformation.

$$x_k = \frac{1}{2}x_0 + \sum_{n=0}^{N-1} x_n cos(\frac{\pi}{N}n(k + \frac{1}{2}))) \tag{3.2}$$

3.2.1 Modified Discrete Cosine Transform

The MDCT is a modification of DCT-4 for overlapped data. The transform takes two consecutive blocks of data and overlaps them so that the last half of the first block and the first half of the second block are crossed over so that the 3/4 mark of the first block intersects with the 1/4 mark of the second block. This overlapping is used to eliminate artifacts in the block boundaries. The MDCT is for that reason often used in signal compression applications such as MP3, OGG Vorbis and AC3 to name a few. Since it uses overlapping the output of the MDCT is half as many as the inputs. [11] The formula for calculating the MDCT is as follows:
The N real numbers $x_0..x_{2N-1}$ are transformed to the N real numbers $X_0..X_{N-1}$. If $k = 0..N-1$ the equation 3.3 makes the transformation.

$$X_k = \sum_{n=0}^{2N-1} x_n cos(\frac{\pi}{N}(n + \frac{1}{2} + \frac{N}{2})(k + \frac{1}{2}))) \tag{3.3}$$

3.2.2 Inverse Modified Discrete Cosine Transform

Inverse MDCT (IMDCT) is used to invert the data from the MDCT. The output from the IMDCT is therefore twice the size of the input. It uses the overlapping to add the output of the first block with the IMDCT output of the second block. This causes the

errors to cancel out and the original data to be retrieved without loss. This technique is known as time-domain aliasing cancellation (TDAC). The algorithm for IMDCT:
The N real numbers $x_0..x_{N-1}$ are transformed to the N real numbers $X_0..X_{2N-1}$. If $k = 0..N-1$ the equation 3.4 makes the transformation. [11]

$$X_k = \frac{1}{N}\sum_{n=0}^{N-1} X_n cos(\frac{\pi}{N}(k + \frac{1}{2} + \frac{N}{2})(n + \frac{1}{2}))) \tag{3.4}$$

To improve the transform properties by smoothing out the start and end of the blocks the transform can use a window function. When a window function is used the equation for the MDCT and IMDCT is changed to the following two equations[11]. For the window function f_k the equation 3.5 shows windowed MDCT and 3.6 shows the windowed IMDCT for $k = 0..N-1$ and $l = 0..2N-1$.

$$X_k = \sum_{n=0}^{2N-1} f_n x_n cos(\frac{\pi}{N}(n + \frac{1}{2} + \frac{N}{2})(k + \frac{1}{2}))) \tag{3.5}$$

$$x_l = f_l \frac{2}{N}\sum_{n=0}^{N-1} X_n cos(\frac{\pi}{N}(l + \frac{1}{2} + \frac{N}{2})(n + \frac{1}{2}))) \tag{3.6}$$

An example of a window function that holds the Princen-Gradley condition 3.7 which is a property required is the one used by Vorbis[11] 3.8.

$$W_n^2 + W_{n+N}^2 = 1 \tag{3.7}$$

$$W(n) = sin((\frac{\pi}{2})sin^2((\frac{\pi}{2N})(n + \frac{1}{2}))) \tag{3.8}$$

Chapter 4

Formats

4.1 The OGG encapsulation format

The OGG encapsulation format version 0 is a format for encapsulating of data streams that is freely-available. The encapsulation format is created by xiph.org foundation and is intended to be used with a larger project to design encoding and decoding multimedia content. Each packet of data that are encapsulated is called a Page. A Page could be of any size but the recommended maximum size for each Page is about 64 kBytes. Each Page starts with a header that contains 27 bytes plus the nr of segment bytes of information and after that the rest of the page consists of a number of segments.
Each segment is between 0-255 bytes and the size of each segment is described in the header of the Page. If a segment is 255 bytes big then that is an indication that the next segments data belongs to the same packet of original data. A packet of data can consist on more then one OGG Page and if that is the case a flag in the header will tell the program that this page contains data that belongs with the previous Page data and the last segment of the previous Page will have the size 255 bytes. The header for each Page consists first of all with 4 bytes that shows that a new Page starts here. The 4 bytes consists of the magical numbers that represents the word *OggS*. The next byte in the header shows the version of the OGG format (currently 0) and the byte after that shows what kind of Page this is. The different Page type is: Beginning of stream (bos), new packet starts Page, continue packet from previous Page, End of stream (eos).
After that the next 8 bytes contains granule information which is information used by a media player to find out where in the media this data should be. The next 4 bytes consists of the serial number of the bitstream. This is used to identify which bitstream this Page is intended to. This is followed by a sequence number to indicate which page number in order this Page is. To ensure that the data in this Page is not tampered with and that no errors in the data consists each page has a checksum included in the 4 next bytes in the header which is used to compare with the sum that is calculated from the data in the packet. If the sums do not match, the data in the Page is not correct and should therefore be thrown away. After this there is one byte indicating how many segments that the Page consists of and the last bytes of the header will be the length in bytes of each segment.[6]

4.2 The Vorbis Audio format

The Vorbis I audio codec is an audio codec that can represent up to 255 discrete channels with sampling rates between 8kHz to 192kHz which is based on the Modified Discrete Cosine Transform. Each packet of Vorbis I data can be of four different kinds, three header packets (Identification header, Comment header and Setup header) and audio packets. In order to setup up the decoder the 3 header packets must come in that order and all three must come before the decoder can decode an audio packet. All the header packets start with the same bytes. The first byte of each header is the type of the header. If the byte has the value 1 the packet is a identifier header, 3 is the comment header, 5 is the setup header. Every packet that has the leading bit set to 0 is considered to be a audio packet. After the type is read then the upcoming six bytes should represent the letters: v, o, r, b, i, s. Each of the packet types will now be described closer:

4.2.1 Identification header

The identification header is the first header that should be received in the bitstream. The identification header holds information about the number of audio channels, sample rate, bitrate max, bitrate nominal, bitrate min, the two different block sizes and the Vorbis version that the audio track is decoded in. If the value of the bitrate max, bitrate nominal and bitrate min is the same value this means that the audio track is encoded using fixed rate. The values of the sample rate and the audio channels field must be larger then zero. The legal sizes for the blocks are: $64, 128, 256, 512, 1024, 2048, 4096, 8192$ and the block0 must have a smaller value then the block1. If any of these conditions are broken then the decoder will throw the entire stream.

4.2.2 Comment header

The comment header contains information about the media such as vendor name, artist name and song title. Here the value read shows the number of bytes in the vendors name. After that the upcoming number of bytes is the string containing the vendors string. After that the number of user comments is read and then the user comments length and text are read from input.

4.2.3 Setup header

The setup header is responsible for reading codebook information, Time domain transforms, the different floor curve representations, the different residue configurations, the different mapping configurations and the different mode configurations. These different sections will be described in the following subsections:

Codebooks

Each of the codebook configurations contains a specific Huffman representation which is used for decoding compressed codewords from the audio packets. Usually the audio codec uses a statically configured probability model, but with Vorbis both the Huffman representation and the VQ lookup in the bitstream. Each of the codebooks has information about the codebook dimension and codebook entries. Each of the entries in the codebook has a specific length which is read from input. The codeword length of an entry is used when creating the Huffman decision tree for this codebook (More

information about Huffman entropy encoding algorithm in the compression techniques section). After reading all the codeword lengths the lookup type for this codebook is read and if the type is one or two a number of values are read and stored for later use. Read more about codebooks at [4].

Time Domain Transform

The Time domain transform section is not used in Vorbis I but it must be read to obtain synchronization in the setup packet. All the values should be set to zero.

Floor

In Vorbis there are two types of floors. They are called floor 0 and floor 1. The floor 0 type is not modernly used since it has a very high execution cost. The floor information is used to generate a floor curve which is a rough representation of the actual sound curve. In the setup header packet information for all the floors of the stream are gathered.

Residue

The residue vector for a channel in an audio frame contains the difference when the floor curve is subtracted from the actual audio curve. There are tree different kinds of encoding and decoding the residue vector. In the setup header all the tree types are decoded in the same way.

Mapping

The mapping is used to set up pipelines for encoding multichannel audio. Vorbis I uses a single mapping type (type 0) with implicit PCM channel mappings. The mapping configuration contains of submaps and each submap uses a specific floor and residue number.

Modes

Each frame is coded according to a mode. A mode could be described as a preset where the best options for the frame are selected. The different options that the mode holds information about are frame size, window type, transform type and the mapping configuration to use.

4.2.4 Audio packet

After the tree setup packets, the rest of the packets in the stream are of type audio packet. The decoding of the audio packet can be described in twelve steps that are described here.

1. Decode packet type flag

To ensure that the packet is an audio packet the decoder reads the packet type flag and verifies that it is the flag of the audio packet.

2. Decode mode number

There are different modes that are defined during the setup packet. The decoder here reads which preset to use in this frame.

3. Decode window shape

In the setup packet the long window size and the short window size are defined. These sizes represent the amount of samples a long/short frame should decode.

4. Decode floor

The floor packet for each channel is decoded. This information is later used when the floor curve is generated (see step 7).

5. Decode residue into residue vectors

The residue packets are decoded according to residue type. The output will be one residue vector for each audio channel.

6. Inverse channel coupling of residue vectors

This step is used on pairs of residue vectors to convert the square polar representation (where one vector is the magnitude and one vector the angle) of the residue back to Cartesian representation. After this step the residue vectors will represent the fine spectral detail of each audio channel.

7. Generate floor curve from decoded floor data

The floor curve is a rough representation of the actual sound curve. In this step the decoder uses the vector of values gathered in step 4 to get the spectral representation of the floor curve. Type 0 uses Line spectral pair (LSP) representation to encode a smooth spectral envelope curve. This type is not used in modern decoders and therefore not that important to understand. Type 1 uses a piecewise straight-line representation that plots the curve on a linear frequency axis and a logarithmic amplitude axis. This representation first draw a straight line from the point (x_0, y_0) to the last point (x_n, y_n). After this the algorithm chooses x axis value that exists between x_0 and x_n and checks which y values the straight line has for this x value. The decoder then reads the amount of difference that the actual curve has from the straight line from the inputstream. This step is then repeated for many points until the floor curve is close enough for the decoder.

8. Compute dot product of floor and residue, producing audio spectrum vector

This step multiplies each value from the residue vector to the value in the floor curve vector with the same index. The output of this step is the audio spectrum of this channel. [4]

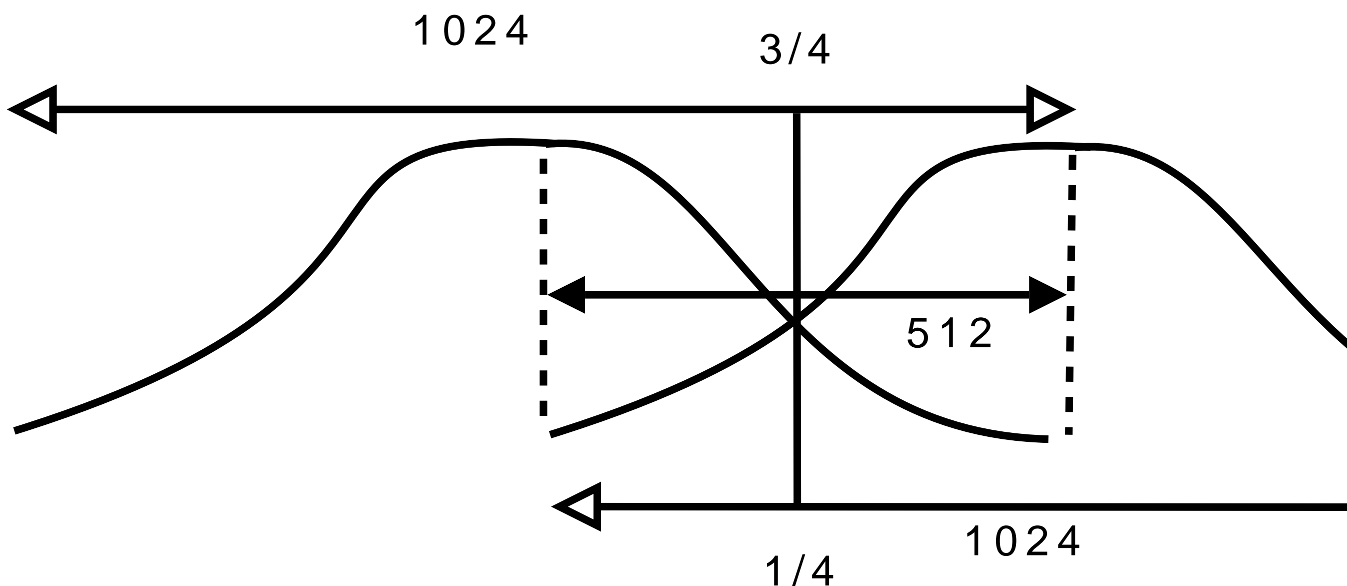

Figure 4.1: Equal size window overlap add. Output will be from middle of first to middle of second.

9. Inverse monolithic transform of audio spectrum vector, always an MDCT in Vorbis I

This step uses equation 3.4 to generate the time domain PCM audio to overlap/add with the previous frame.

10. Overlapp/add left-hand output of transform with right-hand output of previous frame

There are two different cases of overlap/add showed in Figure 4.1 and 4.2. The result from overlap add is the result of overlapping the data from the middle of the first frame to the middle of the second frame where the curves must intersect at the $\frac{3}{4}$ point of the first frame and the $\frac{1}{4}$ point of the second. Read more about it at [4]

11. Store right-hand data from transform of current frame for future lapping

The right half of the frame is stored so that it can be used in step 10 for the next frame.

12. If not first frame, return results of overlap/add as audio result of current frame

The first frame of audio is only used to get a half to overlap with the upcoming frame and will not return any audio. All the other frames will return audio data. The amount of data to be returned is $\frac{1}{4}$ of the size of the first added with $\frac{1}{4}$ of the size of the second. [4]

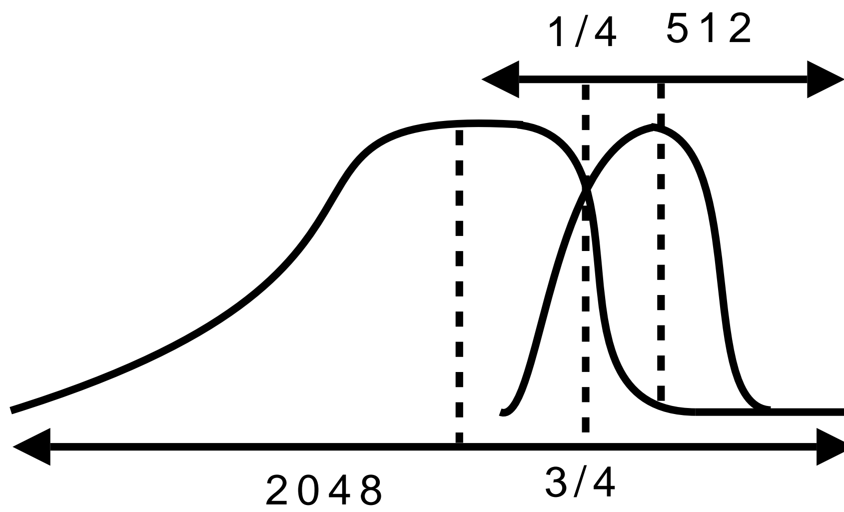

Figure 4.2: Unequal size window overlap add. Output will be from middle of first to middle of second.

Chapter 5

Implementation

The problem wiht the OGG Vorbis format is that it uses float values for the representation of the audio curves and calculations using float values are very time-demanding. The use of floats in J2ME has for that reason not been allowed until recent. If one should design an integer based solution the problem would instead be to get a precision good enough so that the result of the decoder could be used in some sense. Another problem with the use of Java as programming language is that the Java Virtual machine that interprets the code to the operating system uses some memory and processor power.
The goal with the implementation part of the master thesis was to implement an OGG Vorbis decoder in Java ME that did not use a lot of memory and at the same time decoded the data in a reasonable time. The decoder should at first be tested using a simulator and then on real handheld devices.

There is an existing Java SE implementation of the OGG Vorbis decoder called JOrbis. One initial thought was to use that code and just adjust it to Java ME but, after looking at the code, it felt like it would take a lot of re-writing to get it as optimized as it had to be in order to run on a standard handheld device the decision was not to use it. Another initial idea was to convert an integer-only based solution written in C called Tremor to Java ME. After some discussion regarding this it was decided that a decoder was to be implemented from scratch at first and if there was time a re-write of the Tremor codec was planned. The reason that a decoder was to be implemented from scratch instead of using an existing code was that once could optimize the different parts of the code and get a greater knowledge of the decoder parts and how a decoder is constructed.

Once this design-choice was made the implementation of the decoder started.

The first part of the implementation was to make a small midlet that showed the progress of the decoder. This was used to start the decoding process of the OGG Vorbis file in the simulation tool. The simulation tool used was the J2ME Wireless Toolkit designed by Sun[10]. This step was also made in order to familiarize myself with Java ME and the different libraries that one could use. Once this midlet was developed it was tested in both a simulator and on my mobile phone to ensure that the configurations were working.

After this the first part of the decoder implementation started. This step was to open a file from a Java ME midlet and getting the data to a byte buffer. This was done by

placing a file in the jad file that is created when one deploy a midlet.

Once the data is in the byte buffer the decoder takes over. The decoder consists of two parts; one to decode the OGG encapsulation and one to decode the Vorbis Audio format into raw PCM audio.

The OGG decoder is a very basic algorithm that removes the header and combines the segments into Vorbis packets. This was implemented according to the specification that given by S Pfeiffer [6]

Once this was finished it was tested for different OGG Vorbis files both with files that other people had decoded and with reference files that I decoded using a generic OGG Vorbis encoder. The testing process work very well from the start and there were no problems with the OGG decoder through out the implementation.

After the developing of the OGG decoder was finished it was used with the initial midlet to make it into a working midlet that decodes OGG files.

Once the implementation of the Vorbis decoder started one notice that it uses a bit-buffer instead of the more usually byte-buffer that the OGG decoder used. This meant that the first thing to do was to implement a bit-buffer reader that was effective and of course reliable. The solution that was implemented was a byte buffer with an index indicating which bit that was the currently read. Methods for reading and interpreting different amount of bits as integers was also implemented and tested to make sure that the decoder dont misinterpret any data.

The main part of the whole implementation was to make the decoder that takes the Vorbis packets and decodes them to raw PCM digital audio. The first part of this was to decode the three setup-packets that tells the decoder crucial information about the audio stream. In the Vorbis decoder four different kinds of packets can occur. Either the packet is one of the three different kinds of headers that the audiostream must start with in order for the decoder to manage to decode the audiostream or the packet is an audio packet that contains the actual data to decode into raw PCM.

The first two packets were really easy to decode. The packets are decoded just by reading some different bit-sized integers and storing them for later. This was easy to make optimized and the implementation was tested on a number of files with a good result. The testing consisted of files that were encoded using different settings and then decoded to see that the decoder read the same settings.

The third packet of the stream is the most complicated packet of the entire decoder. This packet holds information for setting up floor and residue and the codebooks to use with the Huffman table building. This is absolutely the packet that if you make a mistake, no matter how small, the entire decoder is trash. This is also the packet that took the longest time implementing. Luckily the specification for Vorbis contains a step-by-step instruction for each part of this packet but it still was difficult to understand the different parts and their meaning in the later steps of the decoding. This decoding step was not as easy to test to ensure that it worked properly since no values for the different things are known. The way that it was tested out was that the results were compared with

print-outs from the Jorbis decoder. The problem was that it was rather hard linking and checking the data due to the large amount of it. In this step one was also forced to make a Huffman tree traverser since it is used to handle the Huffman threes that the codebooks hold. Once the implementation was tested the implementation went on to the audio packet decoding.

The final packet type is the audio packet. This packet contains of a header with some information on what preset to use for this audio frame and after that it is just audio data to decode. The specification is a bit vague about this part of the decoder and kind of takes for granted that the developer has a background developing this type of decoders. It took a lot of time understanding what a floor curve and residue is and how they build up the digital audio. At first a straight-on-implementation was done just following the steps described in the specification without really understanding what each part did. But after a while it occurred that one really have to understand what role each part has in the decoding process to make sure that the decoder actually produces a proper result.

Therefore a deeper study in digital audio was performed and with help from staff members working at Vimio a clearer picture was drawn about the role of the floor curve and the residue.

Once that was done the specification was more understandable and the decoder implementation was done producing the raw PCM audio. The values that were produced were float numbers in the interval -1 to 1. These numbers was then converted to the interval 0 to 255 to fit the interval used by the Wav audio format that is a standard audio format for raw audio.

When this part was done the decoder was tested on an actual audio file. The first test was a simple one channel clip that lasted about 3 seconds. The decoding process of this clip lasted for about 5 minutes and the result was not that great. It had some resemblance to the original sound but was not close to the original clip. After implementing some optimization like lookup-tables and change the way much of the data was calculated the time for the decoding went down to about 3 minutes with the same result as before. The reason for the difference in audio was suspected to come from a possible error in the code in respect of a signed or unsigned integer at some point. Since the goal of the master thesis was to investigate if one can use a Java ME built OGG Vorbis decoder this sound difference was left and the time was spent on trying to cut the decoding time instead.

The main time consumer was pin-pointed to be the float operations that are needed for the calculation of the audio frames data. This operation is really tough for a device using small resources such as a mobile phone or PDA. Some additional lookup tables were implemented and that reduced the time but only by about a half minute. More lookup tables was implemented but when trying them out the implementation ran into the other issue with writing software to low resource devices, the memory of the device ran out ant the new lookup tables were removed.

By this time it seems clear that a float based solution was not possible since float operations was to resource consuming for these kinds of devices.

Since time was running out for the master thesis a quick attempt to convert the C code in the integer based Tremor decoder but there was not enough time to finish the implementation.

Chapter 6

Platform

This chapter describes the different platforms used for the implementation. The two platforms described are Java ME and MIDP.

6.1 Java ME

A problem when designing programs for the mobile phones is that different models of phones uses different operating systems, making it more difficult to design a program that can be used by all phones. This is one of the main advantages of using the Java programming language. Since Java uses a virtual machine (JVM) to help the operating system to interpret the Java code that is being executed. In 2000 Sun released the $Java^{TM}$ 2 Micro Edition ($J2ME^{TM}$). [8] The difference between $Java^{TM}$ Standard Edition ($J2SE^{TM}$) is that the $J2ME^{TM}$ uses a different virtual machine called KVM that uses less virtual memory compared to the JVM. There are two configurations that uses the $J2ME^{TM}$ named CLDC (Connected limited device configuration) and CDC (Connected device configuration) where CLDC is designed for mobile phones and other low memory devices while CDC is designed for set-top boxes and smart phones. The first version of CLDC didn´t use floating-point operations and was very limited in it´s libraries. In mid 2004 CLDC 1.1 was released and this version had support for floating-point operations which increased the use of $J2ME^{TM}$ in mobile phones. [8]

6.2 MIDP

MIDP (Mobile Information Device Profile) together with CLDC (Connected Limited Device Configuration) provides the standard J2ME runtime environment that is used by the most popular mobile phones and PDAs today [9]

The version used in this thesis was the 2.0 version.

Chapter 7

Results

The result of my thesis shows that a float-based Java ME OGG Vorbis decoder couldnt be used to decode music in a reasonable time on a device with small resources such as a standard mobile year 2007. The problem lies within the massive float operations that Java ME really has a problem with. Much of the calculations can be pre-calculated and stored in lookup tables but that would demand a large amount of memory for storing and since the memory also is very limited that really doesnt seem to be a solution.

Although the implemented decoder actually manages to decode the audio file the decoding time is too large to make the decoder useful. One can only imagine how long a 2 channel full-length song would take to decode if a 3 second mono audio clip takes a couple of minutes.

The initial idea of rewriting the integer based decoder from Tremor was never done due to lack of time. It would have been interesting to see the difference between the integer based and the float based solutions.

To conclude this section one should say that this result was expected but it was interesting to see the limitations that a mobile phone developer faces and the hard work that they are facing.

Chapter 8

Conclusions

When I selected this Master Thesis from a list of possible work I got a mail from Göte at Vimio saying that this was the toughest of the alternatives and that the guys at Vimio where exited to see the result of it. This of course made me a bit nervous but also it motivated me to do my absolute best to finish the thesis. I feel happy with my work although it did not result in a working decoder. The most important was to show if the decoder manages to decode the audio-stream in a reasonable time.

I wish although that I would have had a bit more time so that I could finish the rewriting of the Tremor decoder that is a integer only solution because I really think that a integer based version really could work out fine.

8.1 Future work

A rewrite of the integer based solution that Tremor has done for C to Java ME would absolutely be the way to go if one wants to continue the investigations of a Java ME OGG Vorbis decoder.

Chapter 9

Acknowledgements

I would really like to take a couple of rows to thank the guys that worked on Vimio during my time there and specially my supervisor Samuel Carlsson that did a amazing job taking time out of his busy schedule to help me out. You guys really know how to take care of a new member of the team and make him feel right at home.

I would also like to thank Thomas Johansson and Per Lindström at CS that helped me with the administrative parts of the thesis as well as the report.

Finally I really would like to thank my friends at CS, P-a, Erik W, Tele, Björn H, Makke and Jen, and also my good friend Micke J that really supported and pushed me to finalize this Master thesis.

References

[1] Owen L. Astrachan. Huffman Coding: A CS2 Assignment. http://www.cs.duke.edu/csed/poop/huff/info/ (visited 2007-01-22).

[2] Xiph.org foundation. Ogg logical and physical bitstream overview. http://xiph.org/vorbis/doc/oggstream.html (visited 2007-01-30).

[3] Xiph.org foundation. Ogg logical bitstream framing. http://xiph.org/vorbis/doc/framing.html (visited 2007-01-30).

[4] Xiph.org foundation. Vorbis I specification. http://xiph.org/vorbis/doc/Vorbis_I_spec.html (visited 2007-01-22).

[5] Bosse Lincoln. Modified Discrete Cosine Transform (MDCT). http://www-ccrma.stanford.edu/~bosse/proj/node27.html (visited 2007-01-22).

[6] S Pfeiffer. The Ogg Encapsulation Format Version 0. Technical report, CSIRO, North Ryde, Australia, 2003.

[7] Th Sporer, Kh Brandenburg, and B Edler. The use of multirate filter banks for coding of high quality digital audio. *6th European Signal Processing Conference (EUSIPCO), Amsterdam, June 1992*, 1:211–214, 1992.

[8] Sun. Java ME Technology. http://java.sun.com/javame/technology/index.jsp (visited 2008-07-01).

[9] Sun. Java MIDP. http://java.sun.com/products/midp/overview.html (visited 2008-07-01).

[10] Sun. Sun Java Wireless Toolkit for CLDC. http://java.sun.com/products/sjwtoolkit/ (visited 2008-07-01).

[11] Andrew B. Watson. Image Compression Using the Discrete Cosine Transform. Technical report, NASA Ames Research Center, Mathematica Journal, 4(1), 1994, p. 81-88, 1994.

[12] Wikipedia. Huffman wikipedia page. http://en.wikipedia.org/wiki/Huffman_coding (visited 2007-01-22).