Evaluation of perceptual sound compression with regard to perceived quality and compression methods

Hans Petter Sirevaag Selasky,
hselasky@c2i.net

Thesis in partial fulfillment of the degree of Master
in Technology, in Information and Communication Technology,
at Agder University College, Faculty of Engineering and Science

Grimstad, Norway
June 2006

I want to express my gratitude to Supervisor Ola Aas at Agder University College, AUC, in Norway, and Professor Donald E. Knuth, at Stanford University in the USA, for their kind support and help in completing my thesis.

“And he said, Go, and tell this people, Hear ye indeed, but understand not; and see ye indeed, but perceive not.”, Isaiah, Chapter 6:9

Contents

List of Figures

1 Introduction

In the area of sound compression there exist several different codecs. In general there are two categories of sound codecs. There are so called “lossless sound codecs” that compress sound like data that must not be altered. Some examples are Flac[**?**], WavPack, Shorten, Monkey’s Audio, Apple Lossless, Ogg Squish, Bonk, La, OptimFROG, LPAC, RKAU and many more. The typical compression ratio is 0.75 .. 0.50, relative to 1.0. The other general category of sound codecs is “lossy codecs”, which will alter the sound. Some examples here are: Ogg Vorbis, mp3, mp4, WMA, QuickTime and AAC. The compression ratio of these sound codecs depends on the quality setting applied, but is typically 0.10 .. 0.04 relative to 1.0.

When working with audio compression, you should keep in mind the typical audio bit-rate, which is defined like this:

$$typical\,audio\,bit\,rate \quad = \quad 44100\,Hz * 16\,bit * 2\,channels = 1.411\,MBit/sec$$

1.4 Megabit per second is a high data rate with regard to audio, and there is a desire to reduce this rate when transferring audio to the end users. This will allow the users to put more audio on their music players and computers, than otherwise possible. This document is going to describe how the codec Ogg Vorbis achieves that among other things. Some other topics that will be discussed are modulo arithmetics, FIR filters, white noise generation, how one can work through source code and finally an evaluation of the Ogg Vorbis sound codec.

2 Ogg Vorbis Compression Techniques

2.1 Bit-packing

When one is dealing with binary compression, every bit of storage is used. It is important to choose appropriate data types for storing information. The data types used for storage of data, should not be too small, neither should they be too large, so that many bits are left unused. Bit-packing means that unused bits are removed from the data. This is not where most compression is gained. But consider the following example where a data type consists of 3 bits. If one does not do bit-packing, maybe one is doing byte-packing. Then instead of 3 bits, 8 bits are used. If this data type is very much used, then the compression gain of using bit-packing over byte-packing is $\frac{(8-3)}{8} = \frac{5}{8} = 62\%$. Bit-packing is the finishing touch. It also forces one to consider carefully the size of all data types. A badly designed codec will have all its parameters 32-bits in size. This is bad because it is very seldom the case that one needs all data types 32-bits in width. On the other hand, such optimizations are often suitable if one wants to increase CPU performance, but not compression performance.

To handle bit packing I have designed my own tool. See the attached CD-ROM. A bit packer will resemble buffer fragments into a single continuous data stream.

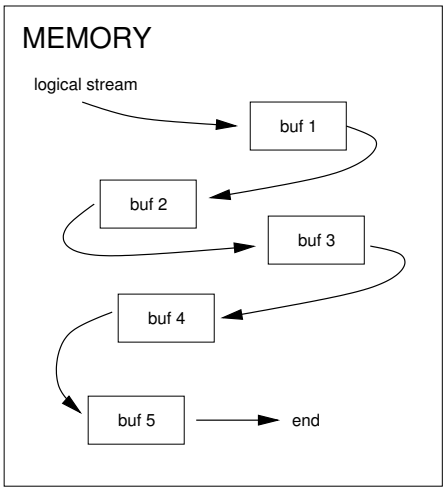

Figure 2.1: Data fragments

The original bit packer of Ogg Vorbis, was very cleverly designed, and it was hard to beat it, but not impossible. To do the bit de-packing fast, one keeps track of the bit level that is always less than

8 bits. The bit level is the offset where a read starts. Bits including and following after the bit level offset belongs to the next read. When one reads out bits, the bit level is incremented. The stream is advanced by the bit level divided by 8 bits, which results in byte(s) and the remainder is kept in the bit level variable.

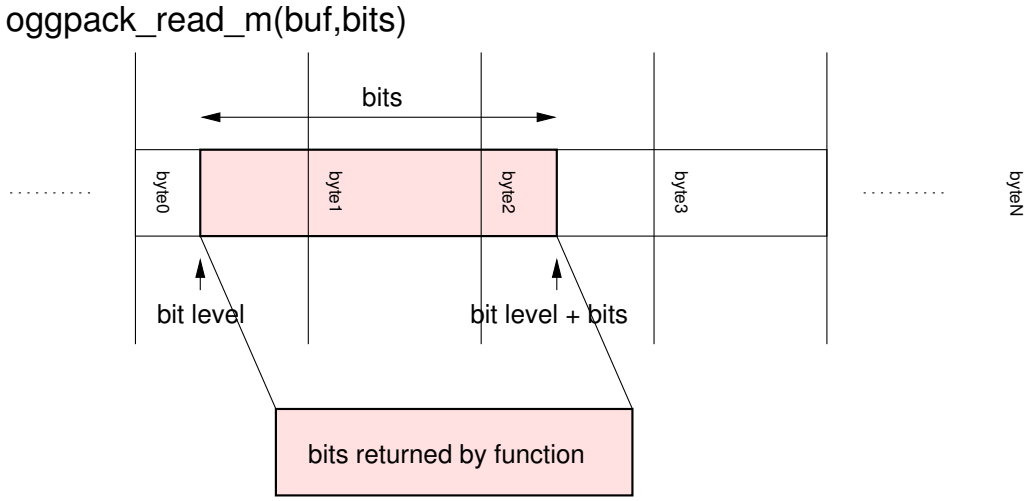

Figure 2.2: Illustration of bit-depacking routine

When one wants to read out data it proves very smart to compute the end of the read in bits. This is achieved by adding the bit level and the number of bits to read. Knowing the end of the read, it is very easy to find out how many bytes are needed for a read. The problem is that the bits that make up a read, can be spread across multiple bytes. Reading a fixed number of bytes, for example 5, will not yield the best performance. The best performance is obtained when one reads as little extra as possible. For example if one wants to read one bit, then it is very wasteful and slow to read 32-bits from memory. By adding some simple “if “ statements to the code, performance can easily be improved for short reads. Below is shown the original code, which appears fairly good.

```
 1     ...
 2
 3         ret=b->headptr[0]>>b->headbit;
 4         if(bits>8){
 5           ret|=b->headptr[1]<<(8-b->headbit);
 6           if(bits>16){
 7             ret|=b->headptr[2]<<(16-b->headbit);
 8             if(bits>24){
 9               ret|=b->headptr[3]<<(24-b->headbit);
10               if(bits>32 && b->headbit){
11                 ret|=b->headptr[4]<<(32-b->headbit);
12               }
13             }
14           }
15         }
16
17     ...
18
```

Figure 2.3: Core part of old bit de-packer

Here is my new code, after some careful rewrites and considerations.

```
 1     ...
 2
 3         if (bits >= 2) {
 4             if ((bits >= 4) && shift) {
 5               temp &= ((htole32(((u_int32_p_t *)ptr)->data) >> shift)|
 6                        (ptr[4] << (32-shift)));
 7             } else {
 8               temp &= (htole32(((u_int32_p_t *)ptr)->data) >> shift);
 9             }
10         } else {
11           temp &= (htole16(((u_int16_p_t *)ptr)->data) >> shift);
12         }
13
14     ...
15
```

Figure 2.4: Core part of new bit de-packer

Here is the result:

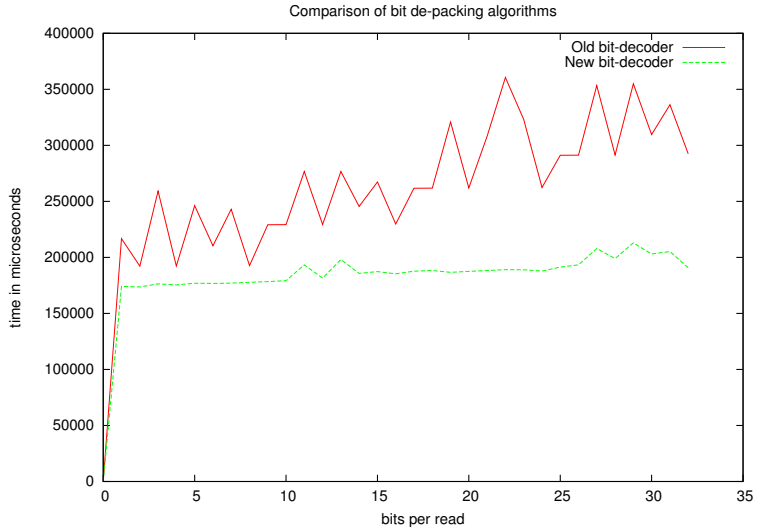

Figure 2.5: Comparison of bit de-packers

2.2 Huffman coding

One lossless compression technique that is used by Ogg Vorbis, is so called Huffman coding. What this implies, I will explain shortly. Audio is a continuous signal that is often limited to a certain range. Let's consider a 16-bit sound card. Each sample is represented by a 16-bit signed value. When one is playing back sound, very often only a certain part of the 16-bit dynamic range will be used. If the sound is very quiet, then the range <-1024, +1024> may be sufficient to represent the signal. Then only 11 bits are required per sample. If one has got 1024 samples, perhaps there are only 256 different sample values used. The PDF, probability distribution function, is a tool to find out what sample values are mostly used, and which are not used so much.

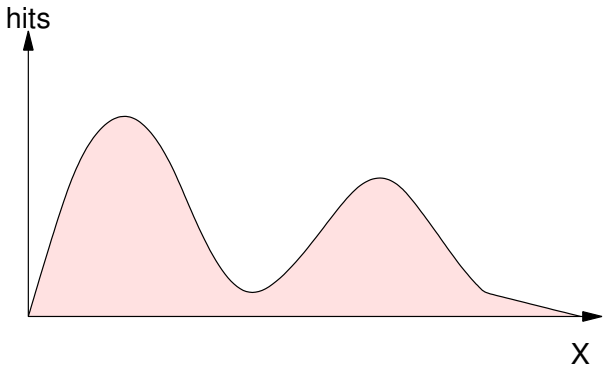

Figure 2.6: Example of a probability distribution function, PDF

Huffman's idea is that sample values that appear more often than other sample values, should be

given shorter codes, and the sample values that appear less often, longer codes.

bit-length	code	real value
1	0xxxxxx	1
2	10xxxxx	2
3	110xxxx	0
3	111xxxx	3

Figure 2.7: Huffman decoding table

NOTE: The least significant bit is leftmost.

By doing this one achieves that the data takes up less space. Consider for example that 80% of the data is '1', 10% are '2', 5% are '0' and 5% are '3'. How efficient is Huffman coding over no coding?

Huffman coding in the example above, requires $(0.80 * 1) + (0.10 * 2) + (0.05 * 3) + (0.05 * 3) = 1.3\, bits\, per\, unit$ Else $2.0\, bits\, per\, unit$ are required. How much data can be saved: $(2.0 - 1.3)/2.0 = 35\%$

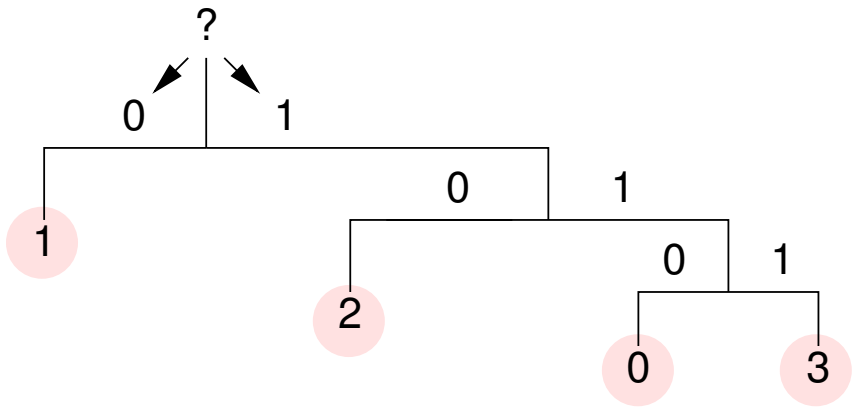

Figure 2.8: Huffman decision tree

NOTE: The decoded values are at the bottom of the tree and the least significant bit is at the top.

The Ogg Vorbis codec supports multiple "Huffman decision trees". Each such tree is called a "book" in Ogg Vorbis. These books are transmitted at the beginning of the "audio stream", in the header. Given a book, Ogg Vorbis provides two functions to decode the bit-packed-data using a Huffman decision tree. These are "decode_packed_entry_number()" and "vorbis_book_decode()".

2.3 Sound decoding

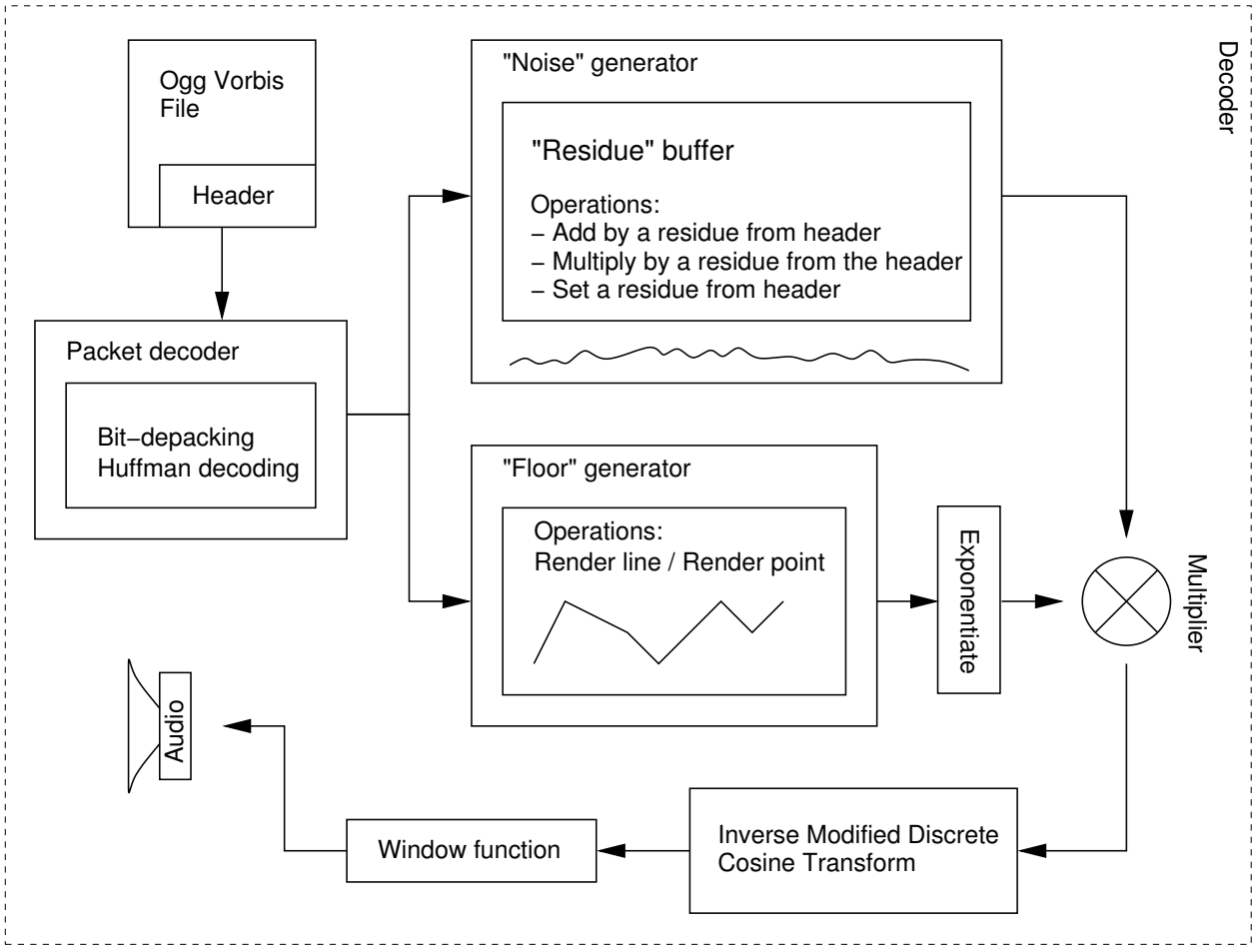

Figure 2.9: Schematic view of the Ogg Vorbis decoding process

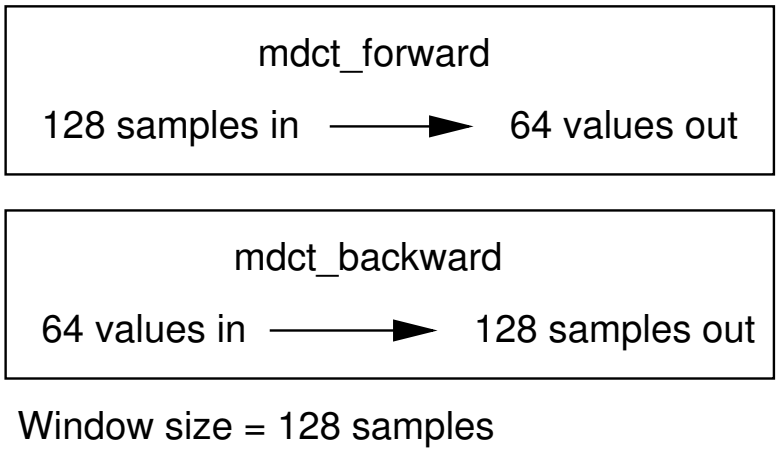

Figure 2.10: The [Inverse] Modified Discrete Cosine Transform

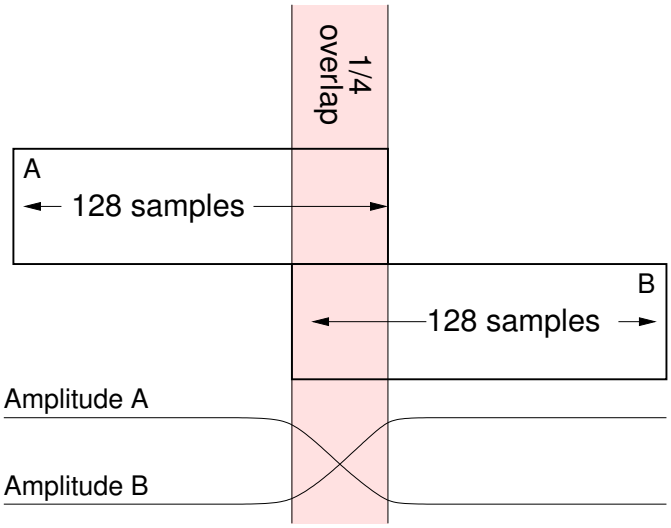

Figure 2.11: Windowing

When Ogg Vorbis decodes sound, two parts are needed. These are called “residue” and “floor” curves. The “residue” is the fine detail of the audio cosine-spectrum that is left after that the “floor” has been divided out. The “floor” consists of a series of points. Between each point a line is drawn, using a function called “render_line()”. The “floor” curve is stored in deciBel, dB, and is exponentiated before it is multiplied by the “residue”. Then the result is passed on to an “Inverse Discrete Cosine Transform” and a “Window function”, before it ends up in the speaker. The Ogg Vorbis synthesizer process is not very complicated. The “residue” buffer is composed of multiple “residues” that are transmitted in the Ogg Vorbis header. Composition means using addition and multiplication of the “residues” available. To decode a so called “audio frame”, the decoder is passed a number that selects

"residue" and a number that selects whether the "residue" should replace, add, or multiply the current "residue" buffer. Also a set of points are transferred to make up the "floor" curve. All values are transferred using Huffman coding.

Here is another figure that shows how the decoding process is carried out, and in which order:

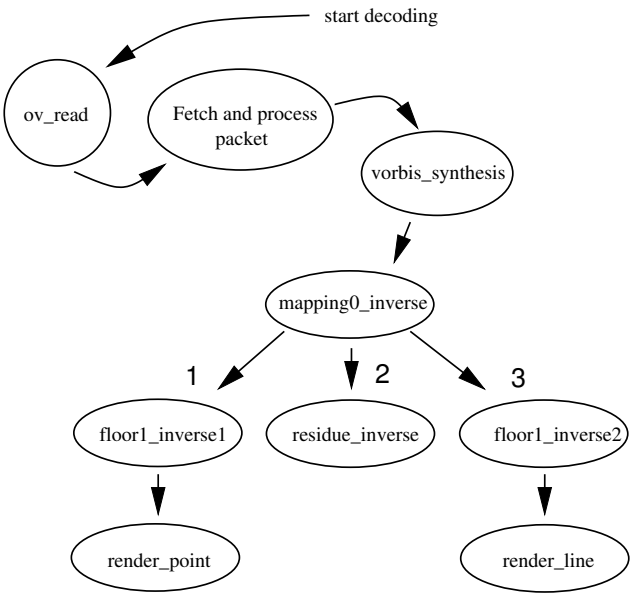

Figure 2.12: Ogg Vorbis decoding overview

2.4 Sound encoding

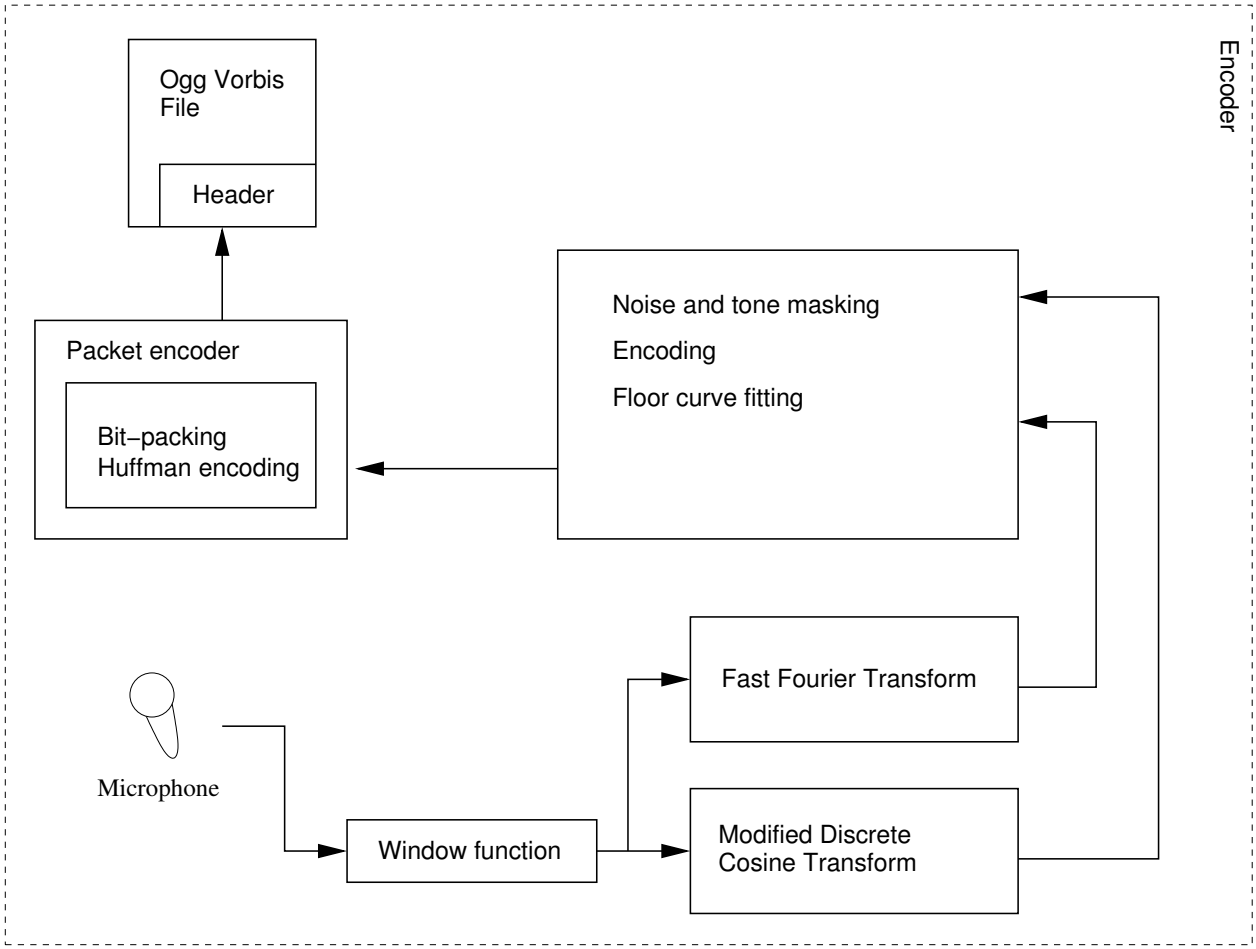

Figure 2.13: Schematic view of the Ogg Vorbis encoding process

When Ogg Vorbis encodes sound, the sound is first passed through a window function. Then the sound is analyzed in two ways. First using a Fast Fourier Transform. This transform reveals information about which sine-wave-frequencies that are strong, and is used to perform so called “tone masking”. Second a modified discrete cosine transform is performed. This transform reveals information about the noise in the signal, and is used for so called “noise masking”. The results from the transforms are combined to generate the “residue” and “floor” curves. Then all data is packed using bit-packing and Huffman encoding. In the end an Ogg Vorbis file is produced. This is the encoding process in a nutshell. There are a lot of more details and complicated formulas used in the encoding process, which I will not go into any details.

3 My own research

In hope that it will be useful in the continuation of this project, I will here mention software that I have been using and some research I have done on modulo arithmetics, white noise generation and FIR filters.

3.1 Source code analysis

The Ogg Vorbis decoder consists of 80'000 lines of code and 34 files. To analyse this I concatenated all the files into a single file, and started unwinding the program flow. Hence not all functions are called directly, but indirectly, through function pointers, I inserted breakpoints in the sourcecode to stop the program when it was running, so that I could see the stack backtrace. In that way it was easy to see how the decoder worked. The operating system which I have been using, is "FreeBSD 5.2", see http://www.freebsd.org . Here is a list of software which I have used during this project:

- sox, see /usr/ports/audio/sox
  This is a sound processing tool, consisting of many different utilities. The most frequently used utility was "play", which I used to play back raw 16-bit sound. A typical command:
  "(fetch -q -o - http://159.33.6.141:80/cbcr1-toronto.ogg | ./a.out | play -c 1 -s w -r 22050 -t raw -f s /dev/fd/0 > /dev/null)"
  This command will retrieve the web-radio called "cbcr1-toronto.ogg", then pass it to my modified Ogg Vorbis decoder, "a.out", which will output raw 16-bit samples, which are passed to the play command.

- octave, see /usr/ports/math/octave
  This is a math tool comparable to Matlab. I used it to make and plot graphs to see what was going on inside the decoder. A typical command was:
  "(fetch -q -o - http://159.33.6.141:80/cbcr1-toronto.ogg | ./a.out | play -c 1 -s w -r 22050 -t raw -f s /dev/fd/0 > /dev/null) |& octave"
  The last "|&" just redirects the "standard error" messages to the octave program. In my software I have a function that will print out matlab commands, to "standard error", called "dump2matlab()". This function will print Matlab commands like:
  X=[1,2,3];
  Y=[1,2,3];
  plot(X,Y,"...");

Then the graph shows up immediately on the screen. By pressing “CTRL+z” I can pause the program. Pressing “%+ENTER” will resume the program.

- cc
  This is the system built in C-compiler. Compiling the Ogg Vorbis Tremor decoder was simple: cc -o a.out -lm -g -Wall *".c"

- gdb
  This is the system built in GNU C-debugger. This program I used to debug the Ogg Vorbis Tremor decoder.

- svn, see /usr/ports/devel/subversion
  This is a tool similar to CVS, that I used to retrieve the latest version of the Ogg Vorbis files.

- xemacs, see /usr/ports/editors/xemacs
  This is the main editor I used for writing code.

- xfig, see /usr/ports/graphics/xfig
  All graphics was drawn using this program.

- lyx, see /usr/ports/print/lyx
  The main document editor used.

3.2 Modulo arithmetics

To simplify calculations modulo arithmetics can sometimes be used. Modulo arithmetics can be used when the input and the output of a function is an integer. That means that the result does not contain any fractions, although decimals are allowed. It is important to note the difference between a fraction and a decimal. The decimals 0.333 equals the fraction $\frac{333}{1000}$ and not $\frac{1}{3}$. Decimals are specific to the number system used. Decimals always result in a fraction with a divisor that is a power of the number system used. Sometimes it is not possible to convert decimals from one number system to another. For example 0.1 in the 3-base will equal 0.333333... in the 10-base, which is a number with infinite decimals. Therefore, when something is not divisible, and one does not want to introduce a rounding error, fractions must be kept as fractions, and not written as decimals, unless it is possible to do so. Here is a definition of the 4 basic modulo arithmetic operators:

3.2.1 Modulo addition

$$(a+b) \bmod c \quad = \quad ((a \bmod c) + (b \bmod c)) \bmod c,\ a \,\epsilon\, integer,\ b \,\epsilon\, integer,\ c \,\epsilon\, integer$$

3.2.2 Modulo subtraction

$$(a-b)\,mod\,c \quad = \quad ((a\,mod\,c)-(b\,mod\,c))\,mod\,c,\; a\,\epsilon\,integer,\; b\,\epsilon\,integer,\; c\,\epsilon\,integer$$

3.2.3 Modulo multiplication

$$(a*b)\,mod\,c \quad = \quad ((a\,mod\,c)*(b\,mod\,c))\,mod\,c,\; a\epsilon integer,\; b\epsilon integer,\; c\epsilon\,integer$$

3.2.4 Modulo division

$$(a/b)\,mod\,c \quad = \quad ((a\,mod\,c)*((1/b)\,mod\,c))\,mod\,c,\; a\,\epsilon\,integer,\; b\,\epsilon\,integer,\; c\,\epsilon\,integer$$

How do you compute the multiplicative inverse modulo an integer? Unlike the 3 previous operators, the answer is not very self explaining. The multiplicative inverse, if it exists, will result in an infinitely large number. For example:

$$(-3/9)\,mod\,1000000 \quad = \quad 333333$$

Because

$$(333333*9/3*999999)\,mod\,1000000 \quad = \quad 1.0$$

Here $(-1)\,mod\,1000000 = 999999$. In general I will write:

$$d \quad = \quad (1/x)\,mod\,(a^n)$$

As "n" approaches infinity, "d" will also approach infinity. If "a" is a prime number, then "d" is called a "p-adic number". If "a" is not a prime number, then "d" is called a "m-adic number". Infinitely large decimal numbers, like shown above, are well known under the name 10-adic numbers. But it is more common to use p-adic numbers, hence these will form a so called "field". To compute the value "d", there exists an old and well known algorithm called Euklid's extended algorithm. I will not go into any details on that algorithm here. I will rather present another and newer algorithm to find "d" when one is working 2-adically, that means when "a" $= 2$. First let's look at the following formula:

$$\frac{1}{1+x} = \sum_{n=0}^{+\inf}(-x)^n = 1 - x + x^2 - x^3 + x^4 - x^5 \cdots, \; x\,\epsilon < -1, 1 >$$

Figure 3.1: Maclaurin series for division

The figure above shows a so called Maclaurin series for division. The "x" has been limited to the range $< -1, 1 >$, so that x^n gets smaller and smaller as "n" increases. In the end the series will converge, because x^n will reach near zero and not affect the result noticeably. But when one is working 2-adically, the series will also converge when "x" is outside the range $< -1, 1 >$. This happens when "x" is an even number. The reason is that when "n" increases x^n will have more and more trailing zeros. That means that the bottom of "d"will converge to a constant value. This fact is well known in higher mathematics.

Here I will present a new formula of my own, that I have developed, and which will compute the Maclaurin's series for division. Then I will prove that the formula is right. Pseudocode follows:

```
u_int32_t : unsigned 32-bit integer
u_int16_t : unsigned 16-bit integer
u_int8_t  : unsigned 8-bit integer
^         : binary XOR
&         : binary AND
|         : binary OR
~         : binary NOT
+         : binary addition
-         : binary subtraction
*         : binary multiplication
/         : binary division
!         : logical NOT
&&        : logical AND
||        : logical OR
```

```
static u_int32_t
f(u_int32_t r, u_int32_t x)
{
    u_int32_t mask = 1;

    /*
     * This function computes the
     * Maclaurin's series for (1/(1-x)):
     * ================================
     *
     * f(r,x) =
     *
     *   r * (x**0 + x**1 + x**2 + x**3 + x**4 ... ) =
     *
     *   r * (1 + x**1) * (1 + x**2) * (1 + x**4) *
     *       (1 + x**8) * (1 + x**16) ...
     */
    while(x)
    {
        if(r & mask)
        {
            r += x;
        }
        x <<= 1; /* multiply by 2 */
        mask <<= 1; /* multiply by 2 */
    }
    return r;
}

/*
 * The function "mult_inverse_32(,)" computes a
 * multiplicative fraction:
 * ================================================
 *
 * mult_inverse_32(rem,x) = ((rem / x) mod (2**32))
 *
 * NOTE: the principle can probably be extended to
 * compute "(rem / x) mod (y**n)"
 *
 * NOTE: "x" must be odd
 *
 * NOTE: (-x) +1 = (~x) +2
 */
#define mult_inverse_32(r,x) f(r,(-(x))+1)
```

Figure 3.2: New formula for 2-adic division

The following proof was established with the help of professor Donald E. Knuth, at Stanford University in the USA. Let $f(r, x)$ be the function that the pseudo code above defines. Then $f(r, x)$ has

got the following properties which might not all be directly obvious:

$$1)\ (f(a,x)+f(b,x)=f(a+b,x)) \mod 2^{32},\ x\,\epsilon\,even$$

$$2)\ (f(r,x)=r*f(1,x)) \mod 2^{32},\ x\,\epsilon\,even$$

$$3)\ \left(f(2r+1,x)=1+2f(r+\frac{x}{2},x)\right) \mod 2^{32},\ x\,\epsilon\,even$$

Combining 1+2+3 gives the recurrence equation:

$$\left(f(1,x)=xf(1,x)+1=1+x^1+x^2+x^3\cdots\right) \mod 2^{32}, x\,\epsilon\,even$$

In the 2-adic system there only exists an inverse for odd numbers. But that is no problem, hence all numbers can be written like an odd number multiplied by 2^n. Then the 2-base exponential value is treated separately when performing modulo arithmetics. Dividing by 2^n is very trivial and will only yield a single exponent subtraction: $\frac{2^n}{2^m} = 2^{n-m}$. "mult_inverse_32()" will always return an odd number or zero, so there is no need to divide down the answer after division.

Below is a definition of my floating point number.

My floating point number:

```
remainder * 2^n , 'remainder' is always odd
```

Implementation:

```
struct fp_num {
  u_int32_t remainder;
  int8_t    exponent;  /* 2**exponent */
};
typedef struct fp_num fp_num_t;
```

Figure 3.3: My floating point number

I will not go into any details on how to split up a multiplicative fraction into its numerator and denominator. The most interesting consequence of doing the computations modulo 2^n is that one can easily compute large factorials without having to use approximations and large integers. For example

like this:

$$f(i, \pi(i)) \quad = \quad \prod_{j=1}^{q_i} \frac{a_{ij0}! * a_{ij1}!}{(a_{ij0} + a_{ij1} + 1)!}$$

Figure 3.4: K2 score function

Modulo arithmetics has the advantage that factors in the numerator cancel factors in the denominator exactly, when working with integer values. The disadvantage is so far that one needs tables to subtract out fractions. All arithmetics are always modulo something, only that it is not so common to think about it.

3.3 White noise generator

The following pseudocode makes up a pseudo-random generator, with uniform distribution, that is used to make perceptually white noise. The precision is 24-bits and the period is $0xffff1c$ samples. What the code below computes, is simply the remainder from division modulo the special prime number:

$$get\,white\,noise(n) \quad = \quad (2^{-n})\,mod\,PRIME$$

```
#define PRIME 0xffff1d

static int32_t
get_white_noise_1(void)
{
    u_int32_t temp;
    static u_int32_t white_noise_rem = 1;

    if (white_noise_rem & 1) {
        white_noise_rem += PRIME;
    }
    white_noise_rem /= 2;
    temp = white_noise_rem;

    temp ^= 0x800000; /* unsigned to signed conversion */
    if(temp & 0x800000) {
       temp |= (-0x800000); /* sign extension */
    }
    return temp;
}
```

Figure 3.5: My white noise generator

3.4 FIR filter design

The four basic types of sine-wave filters are band-pass, band-stop, low-pass and high-pass:

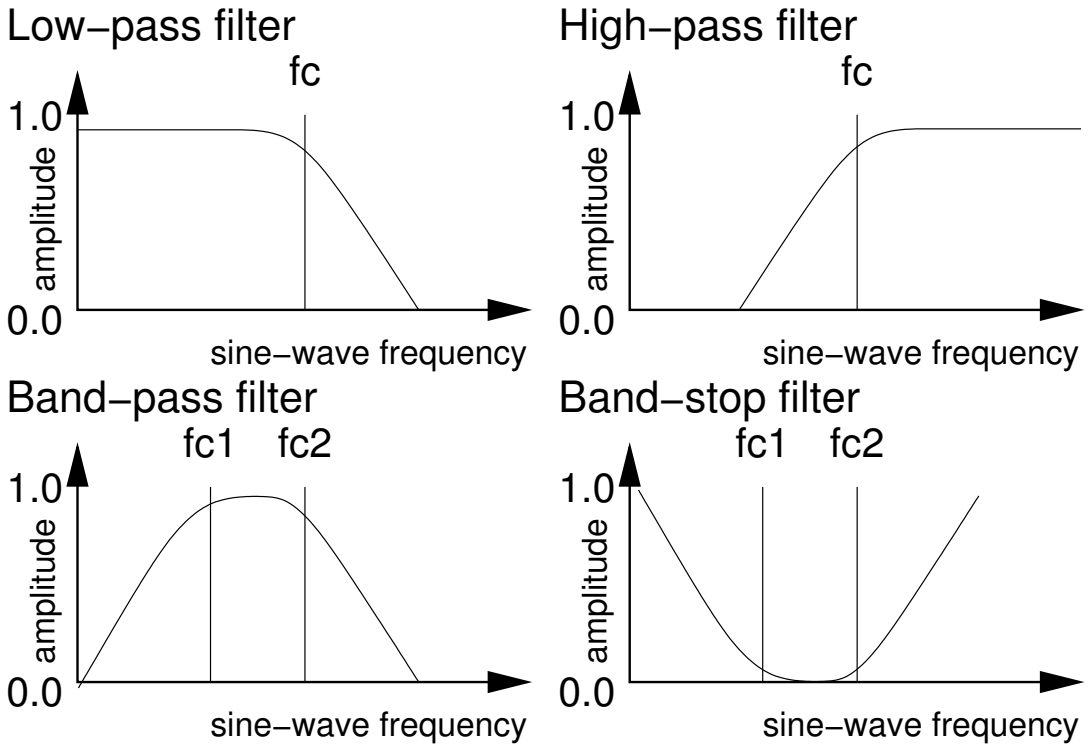

Figure 3.6: The four basic sine-wave filter types

FIR filters are realized like a correlation function. One simply computes how much the filter coefficients, here "$a(x)$", correlates with the signal. The result is the output sample. Mathematically, a FIR filter is written like: $g(t) = \sum_{n=-\inf}^{+\inf} f(t-n) * a(n)$, where "$g(t)$" is the output function, "$a(n)$" are the filter coefficients, and "$f(t-n)$" is the input function.

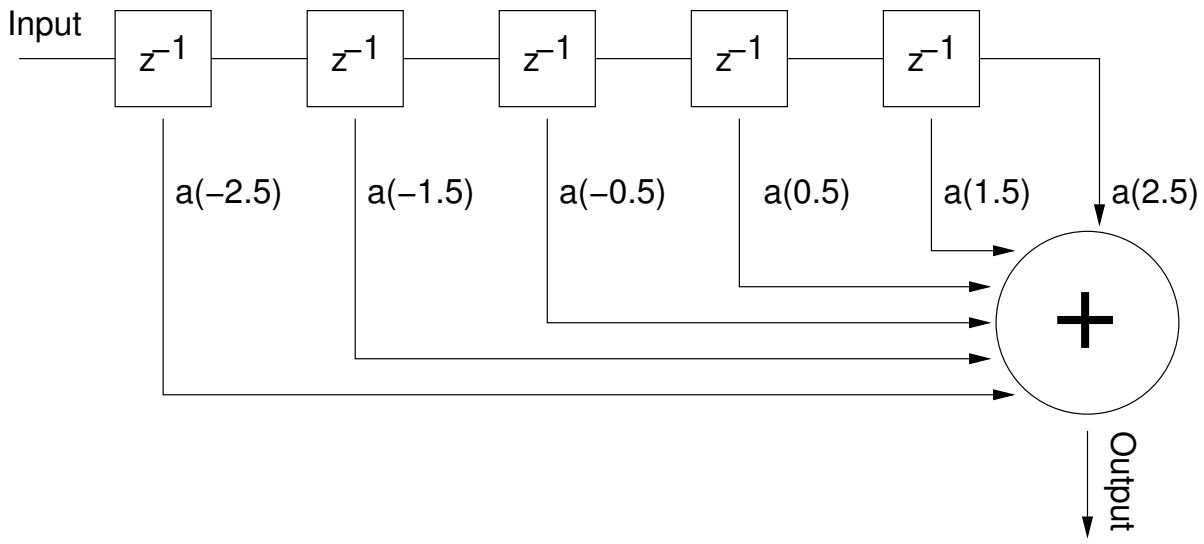

Figure 3.7: Schematic FIR filter realization (6 tap, even FIR filter)

There are four types of FIR filters. There are even and odd tap numbered filters. There are positive and negative symmetry filters. Combining the two types gives a total of four different types. All the four types of FIR filters have the following advantages over other filter types, like IIR filters: [1]

- FIR filters have a linear phase response. That means that filtered sine-waves are not phase distorted.

- FIR filters are realized non-recursively, and are always stable.

The only disadvantage about FIR filters, is that they need some CPU power, and many coefficients, also called taps, to get sharp sine-frequency cutoff edges. The following table shows how one can generate filter coefficients for the four basic sine-wave filter types:

	ideal impulse response	
filter type	$a(n),\ n <> 0$	$a(0)$
low pass	$amp * \frac{sin(w_c * n)}{\pi * n}$	$amp * 2.0 * f_c$
high pass	$amp * -\frac{sin(w_c * n)}{\pi * n}$	$amp * (1.0 - 2.0 * f_c)$
band pass	$amp * (\frac{sin(w_2 * n)}{\pi * n} - \frac{sin(w_1 * n)}{\pi * n})$	$amp * 2.0 * (f_2 - f_1)$
band stop	$amp * (\frac{sin(w_1 * n)}{\pi * n} - \frac{sin(w_2 * n)}{\pi * n})$	$amp * (1.0 - (2.0 * (f_2 - f_1)))$
$w_c = \frac{4\pi f_c}{WINDOW\_SIZE},\ f_c is\ frequency\ relative\ to\ window\ size$		

The following pseudocode implements the above table:

```
#define M_PI 3.14159
#define D(x) ((double)(x))
#define WINDOW_SIZE 256

static void
low_pass(double freq, double amp, double *factor)
{
   int32_t x, z;

   freq -= ((int32_t)(freq / D(WINDOW_SIZE/4))) * (WINDOW_SIZE/4);

   z = (D(D(WINDOW_SIZE/2) / (2.0*freq)) * D((int32_t)(2.0*freq)));

   if(z < 0) {
     z = -z;
   }

   if(z > (WINDOW_SIZE/2)) {
     z = (WINDOW_SIZE/2);
   }

   factor[(WINDOW_SIZE/2)] += (2.0 * amp * freq) / D(WINDOW_SIZE/2);

   freq *= (2.0*M_PI) / D(WINDOW_SIZE/2);

   for(x = -z+1; x < z; x++)
   {
      if(x != 0) {
        factor[(x + (WINDOW_SIZE/2))] += (amp * sin(freq * D(x))) / (M_PI*D(x));
      }
   }
   return;
}

static void
high_pass(double freq, double amp, double *factor)
{
   /* NOTE: freq must be negative */

   factor[WINDOW_SIZE/2] += 1.0*amp;

   low_pass(-freq, amp, factor); /* high-pass */

   return;
}

static void
band_pass(double freq_a, double freq_b, double amp, double *factor)
{
   low_pass(freq_b, amp, factor); /* lowpass */
   low_pass(-freq_a, amp, factor); /* highpass */

   return;
}

static void
band_stop(double freq_a, double freq_b, double amp, double *factor)
{
   factor[WINDOW_SIZE/2] += 1.0*amp;

   low_pass(-freq_b, amp, factor); /* lowpass */
   low_pass(freq_a, amp, factor); /* highpass */

   return;
}
```

Figure 3.8: FIR filter coefficient generator

In the examples above, the vector pointed to by “factor” has “WINDOW_SIZE” elements. The first check performed by the low-pass filter, is to ensure that the sine-wave frequency is within plus/minus

a quarter of the “WINDOW_SIZE” constant. If the sine-wave frequency is outside this range, then one simply gets an alias frequency, due to the discretization of the sine-wave that is used in the filter. The next thing the lowpass filter function does, is to compute the number of coefficients used by the filter, into the variable "z". This is very important, and is illustrated in the following figure:

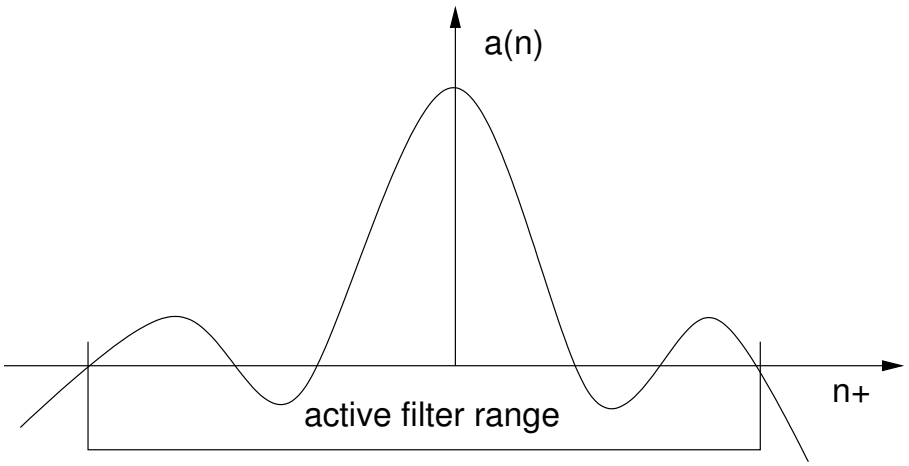

Figure 3.9: Zero-truncating of filter coefficients after last zero crossing point at the ends

If one does not zero-truncate the end of the impulse response, then one will get additional noise into the signal, hence one is demultiplexing using whole sine waves, and not sine-waves cut into pieces. By assuming that all other sound components are independent or orthogonal to a sine-wave at a given frequency, one can demultiplex a sine-wave by correlating with a sine-wave. This might not seem obvious at first, but that is the way it is.

4 Evaluation of perceived quality

Source: http://www.webervst.com/fm.htm

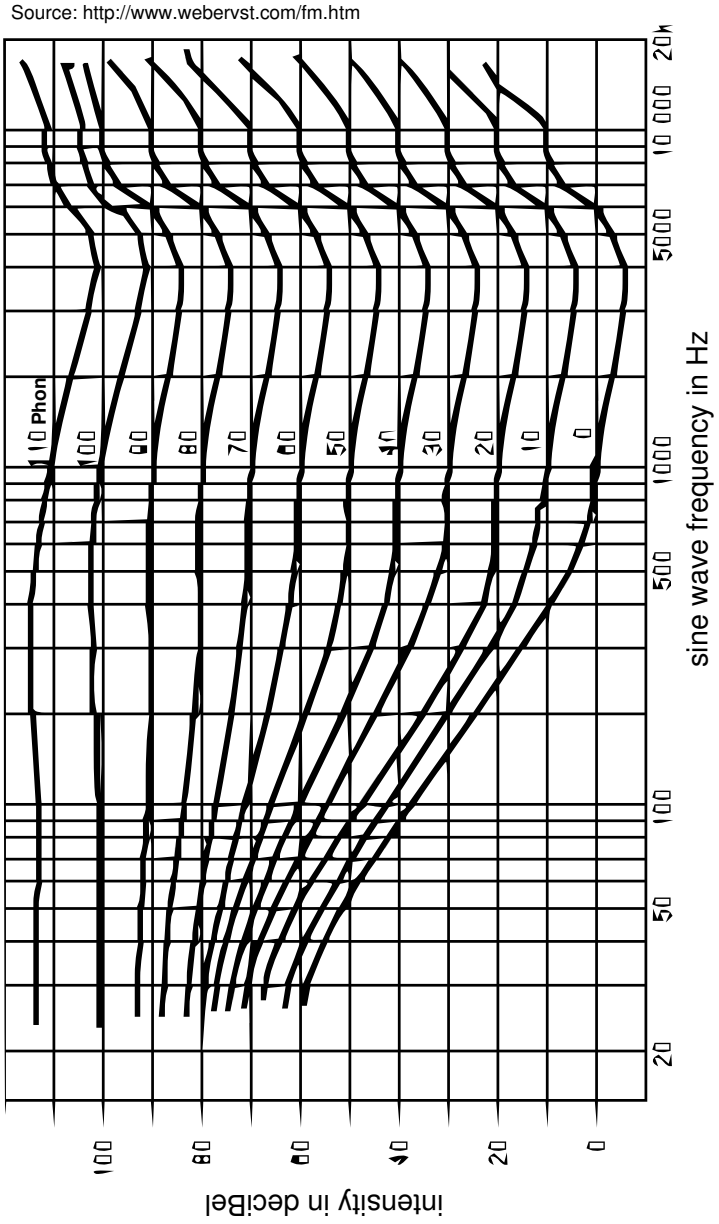

Figure 4.1: Equal loudness curves

Perceived quality is about the subjective opinion of a human. In the 1930s Fletcher and Munson performed an experiment at Bell labs. The experiment was about letting people judge when two sine waves with different frequencies had the same amplitude. The result is presented in the graph above. Based on the results a new intensity scale was made, called phons, which is intensity relative to a 1000Hz sine wave in deciBel. This just shows that one has to keep in mind that the hearing system will apply an impulse response on the sound before it reaches the brain. Knowing which frequencies become amplified and attenuated might help the sound encoder to make the right decision whether to keep a sine-wave component or not. In contrast to subjectiveness, objectiveness is about statistics, how well the synthesized sound matches the original sound, in this case.

4.1 Seeing with sound

There is a relationship between sound and images in our brain. Hearing can cause seeing. There is a web-site[3] on the internet where blind people can download software that will make sound from images. When blind people listen to the sound, they can train their brain into actually producing real images. What this software basically does is that it transforms each pixel column in the picture into representing a piece of sound. The brightness of a pixel directly affects the loudness of the corresponding sine-wave-frequency. Usually the image is scanned from left to right, and pixels at the top have higher frequencies than pixels at the bottom. This transform is very simple, but when implemented, it will require some CPU.

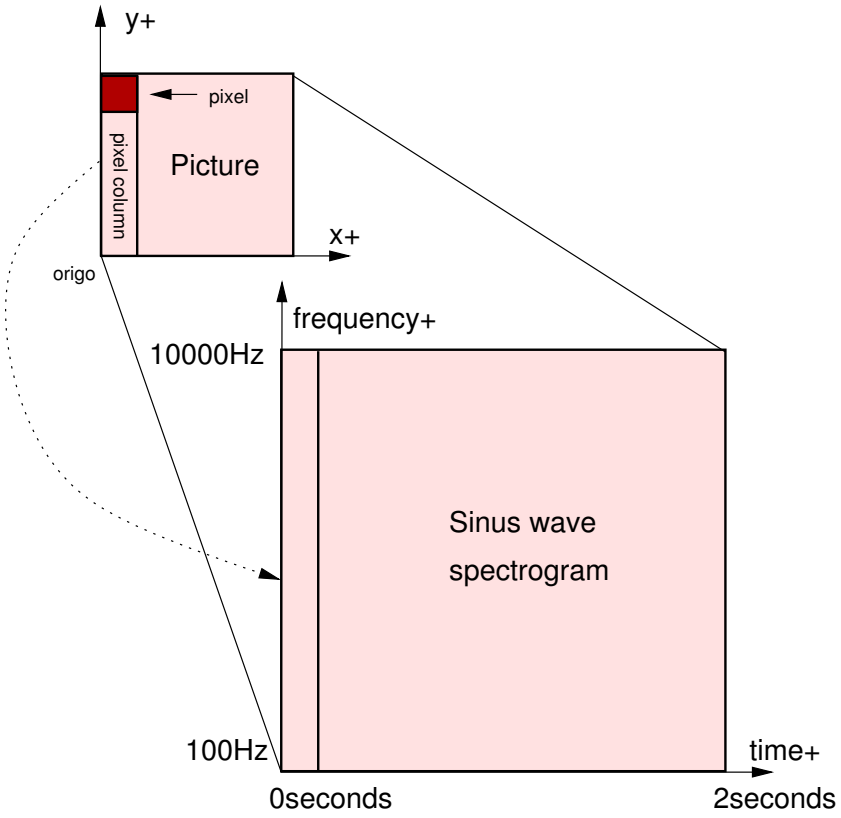

Figure 4.2: Image to sound mapping

Using the Fast Fourier Transform, also called FFT, to directly transform the image into sound, is possible. But in my opinion it will sound better if white noise is passed through a so-called FIR filter, with for example 128 taps. This will require something like a 40MHz computer for a 128x256 sized image.

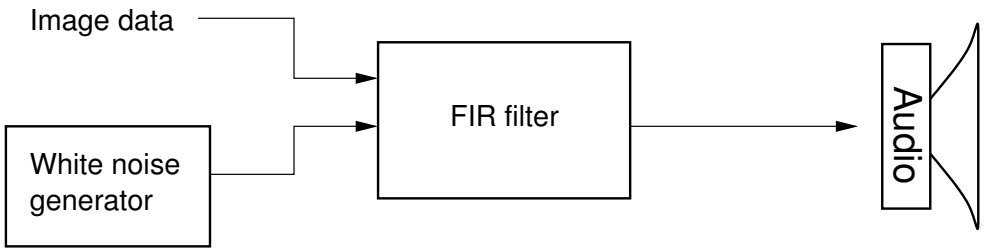

Figure 4.3: Realization of image to sound mapping

See the attached CD-rom and the directory called “image_sound” for sound samples. All of the samples were compressed using Ogg Vorbis, at a bit-rate of 24kbit per second. Then the result was compared with the original files. Except for one file the result was perceptually the same. This was a file where an image with lots of fine details was transformed into sound. Ogg Vorbis was very clearly

hiding details in the image, when comparing the sound with the original wave file. The storage of a 128x256 black/white image is $128 * 256\,bits = 32768\,bits = 4096\,bytes$. If the time used per image is $2\,seconds$ then approximately $2048\,bytes$ are transmitted per second. This is below $3000\,bytes$ per second which equals $24kbit/second$. In theory Ogg Vorbis should be able to reproduce the images without artifacts, but the problem is that probably Ogg Vorbis does not have a high fidelity white noise generator to synthesize the noisy sound. This has to be investigated further.

4.2 Comfort noise

With the invention of digital sound, the noise levels were significantly reduced. Especially when speaking on the telephone one can be fooled into thinking that there is no one at the other end, when the sound level falls below a certain limit. One might not believe it, but in digital telephone systems, white noise is mixed with the recorded sound from the microphone, in periods of silence. This is just done to make the user comfortable, that the call is still active[4]. Ogg Vorbis does not have a comfort noise generator, but maybe it should. If the sound contains noise, then it really does not matter if the noise is matched accurately. As long as the sine-wave intensity is the same for all frequencies, the noise would sound perceptually the same. On the other hand, Ogg Vorbis performs noise-masking in the encoding process, but it uses its own encoder to generate noise, which might not be any good idea, since noise when being poorly reproduces, may sound like something is “bubbling”.

4.3 “S” sounds

At low bit-rates, like 24kbit/s, I have noticed that Ogg Vorbis has problems with “S” sounds. An “S” sound is the sound that appears when one tries to pronounce the “S” consonant. The problem is that Ogg Vorbis does not have a white noise generator, and the encoder does not do this so well, at 24kbit/s, due to the low symbol rate. This is something that might be considered for the next version of Ogg Vorbis, that “S”-sounds are masked and treated separately.

4.4 Ogg Vorbis at higher bit-rates

At bit-rates from 128kbit/s and above, Ogg Vorbis produces perceptually the same sound as if a wave file was used. There are no artifacts with “S” sounds, like at lower bit-rates. I have compared Ogg Vorbis with MP3, and the difference is noticeable. At 128kBit per second, one can clearly hear that MP3 boosts the mid-sine-wave frequencies. Ogg Vorbis sounds more like high-fidelity, in other words, quieter and clearer.

5 Discussion

5.1 Introduction

I started this project with little or no knowledge of the sound codec Ogg Vorbis. I knew what it could accomplish, but not how it worked. I have not been able to find many sources to base my knowledge upon, and the time has mostly been spent studying and modifying the Ogg Vorbis source code.

Prior to the project I had some ideas about using modulo arithmetics in the codec, to get an integer implementation, but that was easier said than done. Therefore I have a section about modulo arithmetics. At present it has not been used, but I hope that it will be used in the continuation of the project. Therefore I did not remove it.

Maybe picking this project was too difficult with regard to what we have learnt at school. I haven't taken any courses in sound compression, so it was not so easy to point at good and bad things about Ogg Vorbis. Most helpful was knowledge of digital signal processing, that I have acquired from my 3-year telecommunication study at HiA. But again, what I have learnt at school, was not sufficient to judge Ogg Vorbis properly. But that was not my personal goal either. Most important for me was to acquire knowledge in something I consider very interesting and then try to improve it.

5.2 Compression techniques

Ogg Vorbis uses Huffman coding, bit-packing, windowing, and discrete cosine transform. These are well-known and recognized compression techniques used by many applications like MP3, JPEG, BZIP2 and PKZIP. Speaking about the "floor curves" and the "residue" that are multiplied together, there is nothing wrong about it. The principle is completely invertible, meaning one can encode any sound and get it losslessly decoded. This is not so difficult to see. Simply set the "residue" to all ones, and store the sound using the "floor curve". How good the principle is, I will not try to answer. This is a time taking task, that will require looking at many sides of the principle, like how well it fits into existing CPU's, parallel execution, what kind of noise it introduces and so on. This might be subject for a continuation of the project.

5.3 My own research

The principles behind my white noise generator and the FIR filters are well-known, and considered to be good. My formula for modulo inverse, is not wrong, it is simple and it appears somewhat new,

according to some mathematicians I have contacted per e-mail. How good the formula is, I will not try to answer. Again, this is a time taking task and it might even be beside the point of this project.

5.4 Evaluation of perceived quality

Many people have applied listening tests, comparing the sound quality of the various sound codecs available on the market today. Just search for “MP3 vs. OGG” at http://www.google.com . In general the perceived quality by Ogg Vorbis is good. That is what the test results show. But when it comes to which sound codec is better, there is some debate. Also there is a difference between live-audio and offline-audio. Live audio will usually not have the same quality as offline audio, due to the way compression is done. At least with Ogg Vorbis, offline audio will sound better.

5.5 Suggestions for continuation

- The Ogg Vorbis encoder needs to be explained and studied further.
- Is it possible to design a sound compression algorithm using pure modulo arithmetics? Maybe one can use some of the ideas in Ogg Vorbis?
- Implement a “S” sound FIR filter in the Ogg Vorbis encoder, to attenuate “S” sounds.

6 Conclusion

It has been an interesting project and I have learnt a great deal about the Ogg Vorbis codec. The sound decoding process is very simple and straight forward, and has been fully explained. The sound encoding process is more complicated and has not been fully explained. Ogg Vorbis has a sophisticated and clever design which is hard to improve, and as one reads the source code, one learns more than one is able to improve. The only problem I have been able to find, is that Ogg Vorbis does not handle “S” sounds at low bit-rates very well. Probably this can be fixed by applying a FIR filter before the sound is encoded. I am very satisfied with the results I have achieved.

Hans Petter Selasky, Grimstad, June 2006

Bibliography

[1] Digital Signal Processing 2nd Ed, Emmanuel C. Ifeachor, Barrie W. Jervis

[2] Ogg Vorbis
http://www.vorbis.com

[3] The VOICE
http://www.seeingwithsound.com

[4] ITU G.168

[5] Huffman coding
http://en.wikipedia.org/wiki/Huffman_coding

[6] Ogg vs. MP3 listening test
http://www.lagom.nl/misc/oggmp3test

[7] Ogg Vorbis explained at Wikipedia
http://en.wikipedia.org/wiki/Ogg_Vorbis