Audio Compression

A Web-API Optimized Coder Design

Daniel Bohman
2016

Master of Science in Engineering Technology
Engineering Physics and Electrical Engineering

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Audio Compression - A Web-API
Optimized Coder Design

Daniel Bohman

Luleå University of Technology
Dept. of Computer Science and Electrical Engineering
Div. of Systems and Interaction

22nd May 2016
The goal of this thesis was to design a web implemented audio codec with the help of JavaScript and WebGL. The idea was to use the computer's graphic processor unit (GPU) for heavy computations through WebGL. The performance was then evaluated on a limited set of systems.

The codec structure is based on the existing G.719 codec which is briefly explained. Additionally, a custom low-complex vector quantizer was implemented instead of the existing one in G.719. The coder was adjusted to take advantage of WebWorkers and the parallelization of WebGL.

The final coder works in real time on all tested systems and produces satisfactory audio, even though a slight performance difference can be seen in favor of Chrome over Firefox.

It was found that WebGL is a really powerful tool but is costly to use. It is especially resource heavy to read the values from the GPU; it is obvious that WebGL wasn't implemented for that kind of use. An extra layer of difficulty is also added by the fact that values can only be sent and retrieved from WebGL as 8-bit values, this needs to be worked around since 8-bits are not enough for high-quality audio.

WebWorkers were found to be a good middle ground where the main tools of JavaScript exist while being able to perform heavy calculations without blocking the main script. This bodes well for the future when AudioWorkers are implemented. The suggestion is to wait for them and then evaluate their performance. The currently implemented coder will likely be easily converted when that happens.
At the end of 2014 I was looking for an interesting thesis to take on and I managed to find one that caught my eye on the Ericssons webpage, Audio Compression - A Web-API optimized Coder Design. I had little to no experience in both codecs and web programming, but it seemed really interesting to get into the world of frequency transformation and vector quantization in real world applicable use. However, before I had got together an application, the ad for the thesis was taken down. I thought my chance had passed me by, but I fortunately had a friend that managed to get me in touch with those behind the ad and the rest is history.

This thesis work is the last component of the master programme in engineering physics and electrical engineering on Luleå university of technology and has been performed from February to August, 2015.

I want to mainly thank Jonas Svedberg who I felt always was interested in my progress and whose door always stood open when I had a question or encountered a problem. Also a thank to Adam Bergkvist, whose door felt equally open but was much less frequently used since I often was too stubborn to ask for help with the programming.

Daniel Bohman
CONTENTS

Chapter 1 – Abbreviations 1

Chapter 2 – Introduction 3

Chapter 3 – Theory and implementation 5
  3.1 Codec selection 5
  3.2 G.719 transform structure 5
  3.3 Inverse transform 7
  3.4 Band structure and envelope quantization 8
  3.5 Vector quantization 10
  3.6 Implementation 13
    3.6.1 Javascript/webaudio 15
    3.6.2 WebWorker 16
    3.6.3 WebGL 17
    3.6.4 Vector quantization realization 19
    3.6.5 Sending values 21
    3.6.6 Decoding 21

Chapter 4 – Conclusions 25
  4.1 Codec testing 25
  4.2 WebGL 27
    4.2.1 Compatibility limitations 27
    4.2.2 Output and operators 28
    4.2.3 Parallelism 28
    4.2.4 The efficient side 29
  4.3 WebWorkers 29
  4.4 AudioWorkers 30
  4.5 Vector quantization 30
  4.6 Future work 33

Appendix A – Codec, main code 37

Appendix B – First worker, encoder.js 59

Appendix C – Second worker, decoder.js 61

Appendix D – webgl-utils.js 63
# Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transform</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphical Processing Unit</td>
</tr>
<tr>
<td>IDCT</td>
<td>Inverse Discrete Cosine Transform</td>
</tr>
<tr>
<td>MDCT</td>
<td>Modified Discrete Cosine Transform</td>
</tr>
<tr>
<td>MSE</td>
<td>Mean Squared Error</td>
</tr>
<tr>
<td>OLA</td>
<td>Overlap and Add</td>
</tr>
<tr>
<td>RGBA</td>
<td>Red, Green, Blue and Alpha</td>
</tr>
<tr>
<td>SNR</td>
<td>Signal to Noise Ratio</td>
</tr>
<tr>
<td>VQ</td>
<td>Vector Quantizer</td>
</tr>
</tbody>
</table>

*Table 1.1: Common abbreviations used throughout the thesis*
Audio compression is a vital part of efficient transmission and storage of digital audio signals. This is done by codecs, these codecs uses a specific algorithm to code the audio signal for storage or transmission and later decodes it for playback. This process is done for almost all audio that passes through today’s computers and cell phones. The codecs that are currently used in cellular networks are based on legacy speech models which doesn’t perform as well with the wideband audio content that users today want to use, e.g. music.

Today’s users are also spread among many different operating systems, for example iOS phones, Android tablets or Windows computers. To reach as many users as possible a codec would have to be compatible with each system.

Instead of creating an implementation of the codec for each system a common denominator can be used, their web browsers. In the most common cross platform browsers there are already a framework to handle audio files and recordings called Web Audio API. This API contains a built in function for time-to-frequency transformation that is an essential part in most codecs.

Coding and decoding can be a computationally heavy process since there are big blocks of information that needs to be transformed in real time and while cell phones and tablets’ computational powers are increasing it could be hard for them to handle. Therefore an often unused resource will be used, the graphics processing unit (GPU), which main purpose is to handle large computations in parallel. The GPU can be utilised through web browsers by WebGL which is a JavaScript API for 3D and 2D rendering of computer graphics in modern web browsers.

The goal with outsourcing the more computation heavy parts of the implementation to the GPU is to keep the web page responsive, something that otherwise might effect user experience negatively or cause choppy audio.

The work will be based on the existing G.719 codec developed by Polycom, Inc and Ericsson. The G.719 codec is a low complex codec that handles full band audio with good audio quality at a low computational load. The functionality of the G.719 codec
will be redesigned for web implementation in JavaScript and WebGL.

JavaScript is a high-level untyped programming language that is supported by all modern browsers and an important building block in modern web pages. In short HTML define the content of web pages, CSS specifies the layout and JavaScript programs the pages behaviour. The JavaScript code is executed on the client side, instead of sent to a server which make applications more responsive. Though the dynamic nature of JavaScript slightly limits its overall performance.

While internet speeds are constantly increasing and storage is becoming cheaper it is still desirable to reduce the size of the original audio stream. This is done by using a lossy codec algorithm which compresses the information stored with a reduction in quality. This transformation is done in a vector quantization process where the initial data gets projected onto a smaller and pre determined set of values. While the G.719 codec has a defined vector quantization process a different solution will be implemented for the web application.

To summarize, the aim of the thesis is to:

- Design and implement a codec into a web page that fulfill the requirements of
  - full band audio up to 20 kHz.
  - medium bit rate.
  - while maintaining reasonable audio quality.
- Investigate web audio effectiveness in Time-to-Frequency implementations.
- Examine the usefulness of WebGL in audio related implementations.
- Analyse audio workers.
- Implement, optimize and evaluate a low complex vector quantizer.
- Evaluate the resulting codec on a limited set of browsers and systems.

While codecs are a well studied area not many examples exist where they are implemented in web pages that doesn’t require plug-ins. Only one previous example can be found of a web implemented codec that utilizes the power of the GPU [1], that project is for a video codec instead of audio. Notable is also that Skype seems to have an upcoming project with a web implemented real time audio and video codec [2]. But that project seem to focus on adding browser tools for the coder instead of using existing tools.
3.1 Codec selection

Since there already exist a wide range of codecs, each with its own strengths and drawbacks, the thesis will be based on existing codec technology. The coding algorithm that was selected as a foundation for this thesis is called G.719 [3], which is a low complexity, full-band codec designed for speech and audio that operates from 32 kbit/s to 128 kbit/s. This codec was chosen because it fulfills the set requirements and since it was developed by Ericsson the results from this thesis could easily be used for patents or other implementations. The G.719 encoder input and decoder output is sampled at 48 kHz which enables a full bandwidth from 20 Hz to 20 kHz.

3.2 G.719 transform structure

The initial function of the codec is to transform the source audio into the frequency domain by a special type of Fourier transform [4]. The spectral coefficients are then compressed by a vector quantizer before sending or storing the values. This process is then reversed when the audio is played back. In this application it is a continuous process since it is meant to run in real time. The way the G.719 codec does this is to operate on 960 samples at a time, $x(n), n = 0, \ldots, L - 1$, which is called a frame. 960 samples at 48 kHz corresponds to 20 ms. Those samples are added to a buffer containing the samples from the previous frame, $x_{\text{old}}(n)$, which forms a 40 ms buffer (the $2L$ block in Fig 3.1).

The values in the buffer are then windowed with a sine window according to the formula in Eq. 3.1.

$$h(n) = \sin \left( \frac{\pi}{2} \frac{n + \frac{1}{2}}{2L} \right), \quad n = 0, \ldots, 2L - 1 \quad (3.1)$$
Figure 3.1: Time chart that shows which frames are sent to the transform

Leading to the windowed signal:

\[
x_w(n) = \begin{cases} 
  h(n)x_{\text{old}}(n) & n = 0, \ldots, L - 1 \\
  h(n)x(n - L) & n = L, \ldots, 2L - 1
\end{cases}
\]

(3.2)

The resulting buffer, \( x_w \), is a 40 ms windowed frame containing 1920 samples, seen in Fig 3.2. This new buffer is transformed back into a 20 ms frame by time domain aliasing.

\[
\tilde{x} = \begin{bmatrix} 
  0 & 0 & -J_{L/2} & -I_{L/2} \\
  I_{L/2} & -J_{L/2} & 0 & 0
\end{bmatrix} x_w
\]

(3.3)

Where \( I_{L/2} \) and \( J_{L/2} \) are the identity and time reversal matrices of order \( L/2 \).

\[
I_{L/2} = \begin{bmatrix} 
  1 & 0 \\
  \ddots & \ddots \\
  0 & 1
\end{bmatrix} \quad J_{L/2} = \begin{bmatrix} 
  0 & 1 \\
  \ddots & \ddots \\
  1 & 0
\end{bmatrix}
\]

(3.4)

Figure 3.2: Signal windowing
The signal is then scaled to maintain a high signal to noise ratio, SNR. In the normal case it is scaled to span the full range of a 32-bit signed integer, $[-2^{31}, \ldots, 2^{31} - 1]$. The frame is also investigated if it contains any transient signals. A transient signal is a short burst of energy, for example the sound from a snare drum. For simplicity it can be compared to a Dirac delta pulse, when the Dirac function is transformed to the frequency domain it becomes an even distribution over all frequencies [4]. This produces a problem when it is later transformed back into time domain, since the pulse can’t be reproduced. The G.719 codec handles these transient signals by dividing up the frame into smaller pieces to get higher time resolution, but that won’t be implemented in this study and therefore not explained further.

If no transient is detected a type IV discrete cosine transformation ($DCT_{IV}$) is applied on the aliased signal, $\tilde{x}$. This is a special form of Fourier transform which transform the time domain signal into a representation of all the frequencies it contains.

$$y(k) = \sum_{n=0}^{L-1} \tilde{x}(n) \cos \left( \left( n + \frac{1}{2} \right) \left( k + \frac{1}{2} \right) \frac{\pi}{L} \right), \quad k = 0, \ldots, L - 1$$ (3.5)

The result, $y(k)$, are the unscaled spectral coefficients of the input frame.

### 3.3 Inverse transform

To retrieve the initial signal the same principles that was introduced in section 3.2 is used in reverse. First a type IV inverse discrete cosine transform ($IDCT_{IV}$) is used with the spectral coefficients, $y(k)$

$$\tilde{x}^q(n) = \sum_{k=0}^{L-1} y(n) \cos \left( \left( n + \frac{1}{2} \right) \left( k + \frac{1}{2} \right) \frac{\pi}{L} \right), \quad n = 0, \ldots, L - 1$$ (3.6)

The output from the $IDCT_{IV}$ is of length $L$, but before it is windowed it needs to go through an inverse time aliasing to make it 2L long. This reverses the aliasing done on the signal in Eq 3.3.

$$\tilde{x}^{uw} = \begin{bmatrix} 0 & I_{L/2} \\ 0 & -J_{L/2} \\ -J_{L/2} & 0 \\ -I_{L/2} & 0 \end{bmatrix} \tilde{x}^q$$ (3.7)

The values $\tilde{x}^{uw}$ are then windowed with the same window that is defined in 3.1 according to:

$$\tilde{x}^{(r)}(n) = h(n)\tilde{x}^{uw}(n), \quad n = 0, \ldots, 2L - 1$$ (3.8)

To retrieve a lossless version of the input signal the values in $\tilde{x}^{(r)}$ are added to the
output of the previous frame with an overlap and add method (OLA):

\[ \tilde{x}^{(r)}(n) = \tilde{x}^{(r-1)}(n + L)\tilde{x}^{(r)}(n) \quad n = 0, \ldots, 2L - 1 \] (3.9)

This OLA operation is illustrated in Fig 3.3. The figure also shows the algorithmic delay which comes from the overlap and add. With a frame size of 20 ms and an effective look ahead of 20 ms the algorithmic delay amounts to 40 ms.

The practice of using a \( DCT_{IV} \) on consecutive frames where the output is overlapped on previous frames is called a modified discrete cosine transform (MDCT). This is often used in modern lossy audio formats, e.g. Opus, EVS and AAC. The window together with the overlap allows the signal to be perfectly recreated by avoiding coding artefacts from the frame boundaries that otherwise might distort the signal [5].

### 3.4 Band structure and envelope quantization

In the G.719 codec the spectral coefficients are divided into four different groups, where each group is divided into bands of equal sub-vector length. The sub-vectors length increase with frequency but are more densely packed on lower frequencies. This allows a spectral band representation that resembles the human ear, with higher frequency resolution for low frequencies. There are for example 16 sub-vectors with 8 coefficients in group 1 that span 0 - 3200 Hz while only 8 vectors with 16 coefficients span the next 3200 Hz. Group 3 consists of 12 sub-vectors with 24 coefficients but has an increased bandwidth to include values between 6.4 kHz and 13.6 kHz. The last group has 8 sub-vectors with 32 coefficients between 13.6 kHz and 20 kHz. Amounting to a total of 44 sub-vectors for the whole spectrum. Frequencies above 20 kHz are not encoded. This band sub-vector distribution is displayed in Fig 3.4.
3.4. Band structure and envelope quantization

The spectrum energy, $N(P)$ is then calculated for the sub-vectors. Defined as the root-mean-square (rms):

$$N(p) = \sqrt{\frac{1}{L_p} \sum_{k=s_p}^{e_p} y(k)^2}, \quad p = 0, \ldots, P - 1$$  \hspace{1cm} (3.10)

Where $L_p$ is the sub-vector length, $s_p$ is the index of the start of the vector and $e_p$ the end. The band norms are then scalar quantized with a quantizer containing a codebook with 40 steps of 3 dB. With index obtained by:

$$I_N(p) = 34 - 2 \log_2 \lfloor N(p) \rfloor$$  \hspace{1cm} (3.11)

The norm for the lowest frequency sub-vector, $I_N(0)$, is directly sent to the decoder, the remaining sub-vector norms are then differentially coded.

$$\Delta I_N(p) = I_N(p + 1) - I_N(p), \quad p = 0, \ldots, P - 2$$  \hspace{1cm} (3.12)

The resulting indices $\Delta I_N(p)$ are then constrained into an index range of $[-15, 16]$ by computing the indices from highest frequency to lowest frequency and then recomputing them from lowest frequency to highest frequency while truncating the result to $[-15, 16]$. This result is then adjusted to $[0, 31]$ by a offset of 15 and Huffman coded. This band structure from G.719 wasn’t implemented for simplicity and time constraints, instead groups of five values from the DCT is coded by a simplified vector quantizer, described in Ch 3.5.

\begin{figure}[h]
\centering
\begin{tikzpicture}
    \draw[->] (0,0) -- (0,5) node[above] {Coefficients};
    \draw[->] (0,0) -- (5,0) node[right] {Frequency};
    \draw (0,0) -- (0,5) -- (5,5) -- (5,0) -- (0,0);
    \draw (0.5,0) -- (0.5,5); \draw (1,0) -- (1,5); \draw (1.5,0) -- (1.5,5); \draw (2,0) -- (2,5); \draw (2.5,0) -- (2.5,5); \draw (3,0) -- (3,5); \draw (3.5,0) -- (3.5,5); \draw (4,0) -- (4,5);
    \end{tikzpicture}
\caption{Coefficients and sub-vector distribution}
\end{figure}
Vector quantization

Transforming the audio signal to frequency domain, as described in section 3.2, does not reduce the amount of information that is used to send or store the signal. To reduce the amount of bits used vector quantization is needed.

Vector quantization is a lossy compression method where a larger set of values gets mapped to a smaller number of fixed points. These fixed points are surrounded by regions called Voronoi regions where all values within one of these regions gets mapped to the fixed point. These Voronoi regions are in many cases small and close to each other where there’s a high probability that many numbers will be and large and far apart where there are less values. This gives small reconstruction errors for often observed values and larger errors on uncommon values.

Each Voronoi region is mapped to a specific codeword stored in codebooks that is used to send or store the compressed version of the signal. The designed and implemented vector quantizer (VQ) works on a principle called apple peeling, which is an algorithmic VQ that doesn’t require a codebook table for encoding and decoding [6]. Instead the values are encoded to regions through calculations that can be reversed in the decoder. Apple peeling is named due to its analogy to peeling an apple, in three dimensions this is easily visualized as the codewords are distributed around the sphere in horizontal lines which are ”peeled” until a solution is found.

This VQ starts with five values, \(x_1, \ldots, x_5\), and use them to construct a 5-dimensional sphere with radius 1. The result is 4 angles (\(\text{Angle}_{\text{shifted},i}\), \(i = 1, 2, 3, 4\)) that gets converted to codewords that maps to uniformly spaced sequential Voronoi regions between 0-90 degrees (\([0 \ldots 90]\)).

In Fig 3.5 it can be seen how the values \(x_1\) and \(x_2\) are used to calculate an angle that falls in Voronoi region 2, \(V_2\). The angle is then approximated by the middle of the region, marked by the black dot.

The calculation for the VQ starts with normalizing the five coefficients, \(x_1, \ldots, x_5\), to unit energy.

\[
\text{sca}l_{\text{e norm}} = \sqrt{x_1^2 + x_2^2 + x_3^2 + x_4^2 + x_5^2} \quad (3.13)
\]

\[
\hat{x} = \frac{x}{\text{sca}l_{\text{e norm}}} \quad x = x_1, \ldots, x_5 \quad (3.14)
\]

The first and second value are then used with an arctangent function for 2 values, \(\text{atan2}\). \(\text{atan2}\) is used to get additional info on which quadrant the computed angle, \(\text{angle}_i\), is in, which regular arctangent doesn’t. \(\text{atan2}(x_1, x_2)\) returns the angle, \(\text{angle}_i\), in radians, between the positive x-axis and the point given by \((x_1, x_2)\). The angle is positive for the upper half-plane and negative for the lower half-plane. Each quadrant, \(\text{quad}_i\), was numbered in such a way that the right quadrant could be calculated by taking the angle from \(\text{atan2}\) in degrees and use:
3.5. Vector quantization

The angle, \( \text{angle}_i \), is then shifted to the all positive quadrant two.

\[
\text{angle}_{\text{shifted},i} = \text{angle}_i - \text{quad}_i \cdot 90 + 180 \quad i = 1, 2, 3, 4
\]

(3.16)

Then the angle, \( \text{angle}_{\text{shifted},i} \), is approximated to a specific codeword. This codeword depends on the amount of bits, \( R \), that is allocated per value. For example if we use \( R = 5.6 \) bits/value there are 16 \( (2^4) \) different possible angles. We find the number of angles \( k \) from:
\[ k = 2^l \] (3.17)

Where \( l \) is calculated with:

\[ l = \frac{R \cdot 5 - 12}{4} \] (3.18)

5 comes from the five x-values, 12 are the bits allocated to scaling_{MSE} and quad info, this is then divided by the number of codewords, codeword_{i}, reserved for angles. With 32-bit output the R value is limited to 2.4, 3.2, 4.0, 4.8, 5.6 or 6.4 bits/coefficient. A design choice was made to have the angles spread evenly over the quadrant with neither 0 nor 90 degrees as part of the set, instead the first value is offset by a half step size, in this example 2.8125 degrees. Then there are 15 more steps of 5.625 degrees each. For higher bits/coefficient these approximations gets closer and closer to the real values.

\[ \text{offset} = \frac{90}{2^{R+1}} \] (3.19)

\[ \text{steps size} = \frac{90}{2^R} \] (3.20)

\[ \text{codeword}_{i} = \text{round} \left( \frac{\text{angle}_{\text{shifted},i} - \text{offset}}{\text{steps size}} \right) \] (3.21)

The 2 values, \( \tilde{x}_1 \) and \( \tilde{x}_2 \) are then squared and added together.

\[ r = \tilde{x}_1^2 + \tilde{x}_2^2 \] (3.22)

This \( r \) value is then used to calculate the second angle, \( \text{angle}_2 \). To describe it graphically, the first two values \( \tilde{x}_1 \) and \( \tilde{x}_2 \) are two perpendicular vectors that point to a point on a plane, with a distance of \( \sqrt{r} \) from the origin. The third value \( \tilde{x}_3 \) are perpendicular to the other vectors, it can be seen as a plane spanned by the point \( \sqrt{r} \) and \( \tilde{x}_3 \). By taking \( \text{atan2} \) of \( \sqrt{r} \) and \( \tilde{x}_3 \) a third angle, \( \text{angle}_3 \), is produced. Since \( r \) can’t be negative the angle between \( \tilde{x}_3 \) and the previous point can only exist in 2 different quadrants, quad_2 or quad_3. This angle is then shifted to the upper right quadrant with Eq 3.15 and 3.16 and approximated with Eq 3.21 to a second codeword, codeword_2, before another \( r \) is calculated:

\[ r = r_{old} + \tilde{x}_3^2 \] (3.23)

Since the x values were normalised at the beginning this \( r \) value will become 1 when we have calculated all angles.

This process is then repeated until all 4 angle codewords are found. The quad information, quad_{i}, and angle codewords, codeword_{i}, are used to reconstruct the ”shape” of the x values, \( \hat{x}_i \), but to completely reconstruct the initial x values a scaling coefficient
is needed. To calculate this coefficient the "shape" of x are calculated, $\hat{x}$, see Eq 3.25 - 3.27. These values are then multiplied with the original x values and summed together to get a scaling coefficient, $\text{scaling}_{MSE}$, that minimizes the mean squared error (MSE) to the original values.

$$\text{scaling}_{MSE} = \sum_{i=1}^{5} x_i \cdot \hat{x}_i$$  \hspace{1cm} (3.24)

The 4 codewords, quad information, $\text{scaling}_{MSE}$ and the R value can then be used to either store or send a compressed version of the values.

To retrieve the values the first thing that is needed to know is the amount of bits/value that was used to store them, R. When that value is known the offset and angle step size can be calculated from Eq 3.20 and 3.19.

$$\text{angle}_i = \text{codeword}_i \cdot \text{stepsize} + \text{offset}$$  \hspace{1cm} (3.25)

After the four angles are calculated the reconstruction values, $\hat{x}_i$, can be extracted in reverse order.

$$\hat{x}_5 = r \cdot \sin(\text{angle}_4 + \text{quad}_4 \cdot 90 - 90)$$  \hspace{1cm} (3.26)

The value $r$ is the radius of the 5-dimensional sphere, normalized to unit energy at the start of the VQ. After calculating $\hat{x}_5$ the $r$ value gets updated since radius of the remaining vectors span less dimensions.

$$r = r \cdot \cos(\text{angle}_4 + \text{quad}_4 \cdot 90 - 90)$$  \hspace{1cm} (3.27)

Eq 3.26 and 3.27 can then be repeated for the remaining 3 angles, giving us $\hat{x}_4, \hat{x}_3, \hat{x}_2$. After calculating the value for $\hat{x}_2$ the last value, $\hat{x}_1$, gets set to the last calculation of $r$. The $\hat{x}$ values are then scaled with the $\text{scaling}_{MSE}$ value to get values approximating those that was initially supplied to the VQ.

$$\text{OUT}_VQ = \hat{x} \cdot \text{scaling}_{MSE}$$  \hspace{1cm} (3.28)

It can easily be understood that this is a lossy method since the resulting values, $\text{OUT}_VQ$, only are a approximation to those that were supplied to the VQ, x. This peeling concept can be extended to any dimension, however it was decided to use a VQ dimension of five.

### 3.6 Implementation

The core of the project was to implement the parts, that was presented earlier, in JavaScript using the APIs available in a modern Web Browser. A cornerstone to get it to work was the Web Audio API. API stands for "application programming interface",
which is essentially a collection of tools, protocols and routines that make it easier to
develop a program by providing easy to use building blocks. The Web Audio API that is
accessed through JavaScript makes it easy to interact with any audio within a webpage.
Fig 3.7 shows a flow chart of the finished product and its source code can be found in
appendix A.

Figure 3.7: The audio path through the implemented coder. Shows workers to the right of the
main script and WebGL implementations to the left.
3.6. Implementation

3.6.1 Javascript/webaudio

Within the Web Audio API exists a concept called AudioContext, which can be described as an interface to AudioNode objects and their connections. AudioNodes are the building blocks that represents audio sources, processing modules and the audio destination. These nodes are then linked as a chain from one or multiple sources through one or more nodes that lastly gets connected to a destination.

In this implementation the sound can come from three different sources: microphone input, generated sine wave or short sound clips. The input signal is then run through an AudioNode called AnalyserNode, which lets the signal pass through without any modification. The AnalyserNode also outputs the amplitude of the signals frequency response in decibel which is used for scaling in a later stage (scaling\textsubscript{DCT}). The input signal is then passed to the next AudioNode called ScriptProcessorNode. This node has an input buffer and an output buffer, which is used to access the signal and run the values through the designed codec. When the values return from the codec they are put into the ScriptProcessorsNodes output buffer. The new audio signal is then sent through a GainNode that adjusts the signals amplitude depending on a volume adjuster on the web page before the signal reaches the speakers.

The implemented code is based on a 48kHz sample rate which corresponds to the original sample rate of the G.719 codec. That means that ideally the samples would be collected in chunks of 960 samples every 20ms and then run through the codec. But the ScriptProcessorNode buffers are limited to sizes that must be power of 2 between 256 and 16384. Whenever the input buffer is filled an event handler called "onaudioprocess" is triggered that initiates the code that specifies what should be done with the samples. A buffer size of 512 was chosen with consideration of how often the event handler will be run, 512 is fairly close to the 960 samples so it doesn’t add any unwanted delay and it doesn’t trigger as often as a buffer size of 256 would. That also means that an additional buffer has to be added with varying size depending on the number of times we add 512 values. Every time the buffer is larger than 1920 values the first 1920 values are sent to a web worker (Described in Ch 3.6.2 and illustrated in Fig 3.9), their first stop on their way through the codec. Then the 960 first values are deleted and the ScriptProcessorNode

![Figure 3.8: The audio's route through the AudioContext.](image-url)
keeps filling the buffer.

For example four times 512 is 2048, but the codec only wants 1920 values according to
EQ 3.2 that means that 128 unused values will remain in the buffer until the event handler
triggers again. A buffersize of 512 values means that the ”onaudioprocess” triggers every
10.67ms ($\frac{512\text{ samples}}{48000\text{ samples/s}} = 0.01067\text{s}$)

<table>
<thead>
<tr>
<th>1920 values sent to codec</th>
</tr>
</thead>
<tbody>
<tr>
<td>960 old values</td>
</tr>
<tr>
<td>512</td>
</tr>
<tr>
<td>960 new values</td>
</tr>
</tbody>
</table>

Figure 3.9: Shows how the ScriptProcessorNode’s buffer size don’t match with what the codec
is designed for.

Each time the 1920 values gets sent to the web worker their maximum amplitude is
measured and stored, this is done to scale the signal in later stages to utilize all available
bits in WebGL (Described in Ch 3.6.3).

### 3.6.2 WebWorker

A webWorker is an API that creates scripts that can run in the background, on a different
thread than the rest of the script. This makes the workers perfect for heavier calculations
because time consuming work doesn’t block the main thread, which allows the user
interface to remain responsive.

The worker is loaded from an external file that contains the code and stays dormant
until it gets a message from the main script. When the worker receives the message it
activates and executes its code, it can also be passed values through those messages. The
values that are sent to the worker are copied, not shared. This means that the worker
can’t modify values in the main script on its own. This also means that there can be a
delay if a large number of values are transfered, but the buffer sizes being sent in this
implementation were small enough that no extra delays were noticed.

When the worker has executed the code it sends back a message to the main script
containing the results. In the main script there must therefore exist an event handler
that listens for these messages. When the event handler gets the message it can execute
code that decides what to do with the results from the worker, mostly adding the results
in the right buffers so other functions can access them.

An example is an simple web worker that only calculates $x + 1$ where $x$ is a number sent
to the worker. The main script initiates the worker who goes into a dormant mode. The
main script then decides to send the variable $A = 5$ to the worker. The worker receives
the message and the value 5. It calculates $5 + 1$, but can’t change the initial variable $A$ which still is equal to 5. The worker then send the result back to the main script. The main script receives the value and updates the value $A = 6$.

The first worker the signal passes through does what’s described in the beginning of section 3.2. The source code can be found in appendix B. It takes the 1920 samples from the ScriptProcessorNode and applies a sine window and the time domain aliasing according to Eq 3.2 and 3.3. Basically making the signal ready to be processed by the DCT. The time-to-frequency transform in Eq 3.5 is a computer heavy computation and to calculate it the GPU (Graphics Processor Unit) is being used. To access the values with the GPU the values must be supplied to the GPU through a texture. A texture is a buffer that only may contain 8-bit values. Truncating the values to 8-bits isn’t a viable solution since 8-bits are too few to get a high quality signal. Instead the values from Eq 3.3 is being transformed to one complement 16-bit values. These values are then divided into 2 parts, the upper 8 bits and the lower 8 bits, and inserted into the texture.

### 3.6.3 WebGL

The GPU is the part of the computer that calculates what the display will show and during regular web browsing its power is often unused. Modern GPUs are highly parallel which makes them more effective than the CPU when processing large data blocks.

To access the GPU through JavaScript an API called WebGL [7] is used. WebGL works on a predetermined area (a canvas) on the web page where the program can display advanced graphics. The canvas consists of a set number of pixels and each pixel consists of four 8-bit values, each value equals the amount of Red, Green, Blue and Alpha for that pixel. The Alpha value specifies the opacity of the pixel, see Fig 3.10.

WebGL is initiated in the main JavaScript code but the code that runs on the GPU is compiled in advance on the graphics card as a fragment- and vertex shader. The vertex shaders purpose is to manipulate the image coordinate of the object that’s supposed to be displayed, mostly used when projecting a 3D image onto the 2D canvas. Since no location manipulation is needed for this implementation the vertex shader is as simple as possible and is only used to map the image in a 1 : 1 relation to the canvas. The position values are then transfered to the fragment shader who calculates the color on each pixel by setting values on the red, green, blue and alpha channel. For the fragment shader to have values to work with a texture is supplied with the values that is needed for the DCT calculation. A texture is a special buffer that WebGL can access that contains the RGBA value for each pixel. In this case the texture is 4096 8-bit values which corresponds to 1024 pixels in a 32x32 grid, see Fig 3.10. The output from the fragment shader are then usually displayed on the canvas but in this case a frame buffer is used, which can be seen as a middle step. WebGL draws the image in the frame buffer but it isn’t displayed on the screen. This can be used to have future frames ready if there are computer heavy computations in the shaders. It is used here because it was found
that it took less resources to read the values from the frame buffer instead of a canvas.

From Eq 3.5 we can see that the output from the DCT is 960 values, therefore the
canvas size was set to to 1024 pixels in a $32 \times 32$ grid and we let the first 960 pixels
contain the results from the DCT.

So when WebGL gets the signal to calculate the DCT it starts with the vertex shader.
The vertex shader supplies the fragment shader with the pixel coordinates and is then
done. The fragment shader then runs the code that calculates the DCT, called Codec-
Shader. The CodecShader is used to both calculate the DCT and the iDCT, but let us
focus on DCT for now. The fragment shader works by executing exactly the same code
on each pixel, $pixel_k$, in the framebuffer. The code for calculating the DCT, Eq 3.5, is
therefore run once on each pixel.

$$y(pixel_k) = \sum_{n=0}^{L-1} \hat{x}(n) \cos \left( \left( n + \frac{1}{2} \right) \left( pixel_k + \frac{1}{2} \right) \frac{\pi}{L} \right) \quad pixel_k = 0, \ldots, L - 1 \quad (3.29)$$

The first thing the CodecShader calculates is which pixel, $pixel_k$, it is calculating.
WebGL uses the bottom left corner as 0 and increases towards the right, see Fig 3.10.
A loop then fetches all the $\hat{x}$ values from Eq 3.3 that is supplied through a texture that
the worker constructed and calculates the summation from Eq 3.29. The values from the
texture are read as two 8-bit numbers but converted back to a one complement 16-bit
number. While the texture interface is limited to 8-bit values the calculations within WebGL are done with 32-bit precision floats. The frame buffer that WebGL writes to has the same limitations as the texture, therefore the result for each pixel is converted from a 32-bit float to a one complement 16-bit value that is then sent out through the red and green channel as two 8-bit values. Not all 32-bits are used because the other two 8-bit values are used for the decoding. To maximize the dynamics of the 16-bits output the signal is globally scaled by the scaling$^{DCT}$ value retrieved from the AnalyserNode mentioned in the start of section 3.6.1. If the values are not scaled there is a chance for either overflow or that small values are rounded to zero. That would considerably reduce the audio quality. With a smaller dynamic range there is also problems with how values close together are rounded. The sound samples path from input to calculated DCT values can be seen in Fig 3.11.

![Diagram](image-url)

*Figure 3.11: Shows the sound samples routing from the audio context to coded values*

### 3.6.4 Vector quantization realization

The values that are retrieved from the CodecShader’s frame buffer are then directly added to a second texture for use in a second fragment shader, the VQShader, which runs together with the CodecShader but output its values on a second frame buffer, seen in Fig 3.12. This fragment shader is used to calculate the VQ values according to the method described in section 3.5.

The VQ works on 5 values each iteration and the pixel location is again used to determine which 5 values that are being fetched from the texture and transformed into codewords. The VQ depends on the $R$ value that sets how many bits each value is allocated, this is separately being supplied by the main JavaScript as a variable. The whole 32 bit (4x8 – bits) pixel output is now available for each VQ output, and is fully used if the maximum amount of 6.4 bits/value is utilized. The red channels 8 bits are always
fully utilized, the top 5 bits store the quadrant information while the three lower bits are scaling information. The VQ scaling coefficient from Eq 3.24 is converted to base 2 logarithm and scaled to 7 bits. Eq 3.30 is obtained from experiments and can probably be improved with further experimentation. If the result from Eq 3.30 is larger than 127 it is set to 127.

\[ scale_{7\text{bit}} = \text{round}(5 \cdot \log_2(\text{scaling}_{\text{MSE}}) - 160) \] (3.30)

This 7 bit number, \( scale_{7\text{bit}} \), is divided into its top 3 bits and bottom 4 bits, the top 3 bits are used to fill up the red channel and the 4 bottom bits are placed as the upper 4 bits in the green channel. Then the remaining green, blue and alpha channel can be seen as a 20-bit value that gets filled by angle codewords, \( \text{codeword}_i \), from the left depending on the selected R value. This is illustrated in Fig 3.13.

The quadrant information, \( \text{quad}_i \), is stored as following, the first quad, \( \text{quad}_1 \), can be any of the 4 quads therefore it takes up 2 bits. The second to fourth quadrants can, as explained in section 3.5, only be in quad 1 or 2. Therefore those quadrants are calculated and reduced by 1 and represented by 1 bit each, filling the next 3 bits. The two 8-bit values from the texture can be seen as a sign bit, 7 high bits and 8 low bits. In WebGL these are treated as unsigned values between 0 - 1. If the high bits are scaled with 127 and added to the low bits the result is a value between 0 - 128.

\[ \text{Pixel(\text{red})} = 64 \cdot \text{quad}_1 + 32 \cdot (\text{quad}_2 - 1) + 16 \cdot (\text{quad}_3 - 1) + 8 \cdot (\text{quad}_4 - 1) + \text{scaling}_{\text{MSE top 3 bits}} \] (3.31)

The four codewords, \( \text{codeword}_i \), are \( 2^l \) bits each, with \( l \) from Eq 3.18. Those are added to a total according to:

\[ \text{Tot}_{\text{sum}} = 2^{20 - i \cdot 2^l} \cdot \text{codeword}_i \quad i = 1, 2, 3, 4 \] (3.32)
This means the codewords will take up the highest $4 \times 2^4$ bits of a 20 bit value. These 20 bits are then distributed to the 3 remaining pixel channels according to:

\[
\text{Pixel(green)} = \text{floor}\left(\frac{\text{Tot}\text{sum}}{2^{16}}\right) + 2^4 \cdot \text{scaling}_{\text{MSE}\text{bottom4bits}}
\]

\[
\text{Pixel(blue)} = \text{floor}\left(\frac{\text{Tot}\text{sum} - \text{floor}\left(\frac{\text{Tot}\text{sum}}{2^{16}}\right) \cdot 2^{16}}{2^8}\right)
\]

\[
\text{Pixel(alpha)} = \text{Tot}\text{sum} - \text{floor}\left(\frac{\text{Tot}\text{sum}}{2^{16}}\right) \cdot 2^{16} - \text{Pixel(blue)} \cdot 2^8
\]

<table>
<thead>
<tr>
<th>Quad 5bits</th>
<th>Scaling_{MSE} 7bits</th>
<th>Codewords 0 – 20bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red 8 bits</td>
<td>Green 8 bits</td>
<td>Blue 8 bits</td>
</tr>
</tbody>
</table>

Figure 3.13: VQ bit allocation over WebGL’s RGBA output

3.6.5 Sending values

The pixel values are then retrieved from the VQshader’s frame buffer. These are the values that are interesting when the goal is to store or send the values to a remote decoder, this can be seen in Fig 3.14. In this research implementation the VQ values are directly decoded. If the codes are going to be stored or sent some overhead needs to be added. Mainly the scaling_{DCT} coefficient from the first worker and the R value need to be stored with the values from each frame. The R value is between 0 and 6.4 while the scaling_{DCT} is a 32 bit number used to maximise the output bits from WebGL, not to be confused with the scaling_{MSE} in the vector quantization. Also the VQ values need to be converted from the 4x8-bits it comes out from the frame buffer to values that are the same bits as maximum needed. For example if $R$ is set to 6.4 all 32 bits are needed, but if $R$ is set to 5.6 only 28 bits are being used. How the values are decoded can be seen in Fig 3.15.

3.6.6 Decoding

To decode the values the VQshader’s frame buffer is directly copied to a third texture. This texture can be accessed by the CodecShader, which handles both the coding and decoding. The R value is also sent into the CodecShader to tell it how to decode the
Figure 3.14: Encoder block diagram. Shows the audio path through the coder.

Figure 3.15: Decoder block diagram. Shows how the values from the coder gets decoded back into an audio signal.
values according to section 3.5. The CodecShader calculates the VQ values and then sum
them according to Eq 3.36 which is the WebGL implementation of Eq 3.6.

\[
\tilde{x}^q(\text{pixel}_n) = \sum_{k=0}^{L-1} y(\text{pixel}_n) \cos \left[ \left( \text{pixel}_n + \frac{1}{2} \right) \left( k + \frac{1}{2} \right) \frac{\pi}{L} \right], \quad \text{pixel}_n = 0, \ldots, L - 1
\]  

(3.36)

The calculated values are then output on the blue and alpha channel as 16 bit one
complement values. These values are then read from the frame buffer and sent to a
second worker. Its source code can be found in appendix C. This worker takes the values
and applies the inverse time aliasing from Eq 3.7, windows the values and adds to the
previous frame, seen in equation 3.8 and 3.9. See the values way from vector quantized
to audio in Fig 3.16.

The first 960 values are then scaled by the amplitude measured before the first web
worker and then added to the ScriptProcessorNodes output buffer, which whenever it gets
full sends the values through the GainNode where they get scaled by a volume adjuster
before they reach the speakers.

A second decode option is also added in this implementation. The values that gets
coded in the CodecShader are sent back into the first worker where they are added
to the very first texture, on the blue and alpha channel. This allows the decoding
algorithm in the CodecShader to read the coded values that hasn’t been through the
vector quantization. This gives the possibility to process the audio in 16-bit quality.

Figure 3.16: Shows the vector quantized values way from the texture to being decoded.
Chapter 4

Conclusions

The created web application shows that it is possible to use JavaScript and WebGL to implement a working reasonable low complex codec in a web browser. But there are many aspects that needs to be taken into consideration. The final version has been tested and works on 6 different computers with different hardware ranging from laptops with integrated graphical processors to stationary computers with dedicated graphics cards. The computers were using either Windows or Linux and the codec was run on both Firefox and Chrome. But the compatibility is difficult to guarantee since only newer versions of the browsers support workers, WebGL and WebAudio [8].

ASM.js was briefly considered to be used for some heavy calculations. ASM.js is a subset of JavaScript that tries to speed up code execution by limiting language features. I came to the conclusion that it wasn’t worth the overhead and limitations imposed when it is slower on everything except really ”big” calculations [9].

A similar project that claim to have implemented a working H.264 video decoder into JavaScript and WebGL can be found online since 2011 [1]. While this thesis shows that a low complex audio coder is possible it can’t be ignored that a video decoder is a lot more complex and since its introduction in 2011 it is hard to find any updates, improvements or similar projects. This could indicate that it has probably been abandoned or encountered big problems.

4.1 Codec testing

A basic user interface was created to test the codec for different signals and settings, this interface can be seen in Fig 4.1.
Conclusions

A Volume

B Using codec
○ No codec

C Voice
☐ 300Hz ☑ 5000Hz ☐ 11000Hz

Set number of bits/sample in VQ:

D

| Bits/Sample: | 16 |
| Bitrate: [kbit/s] | 768 |

E Real Time visualisation

(Figure 4.1: Web application interface for codec testing.)

F Beethoven soundclip

G Scaling

| Encoding range used: | 92.9318 % |
| Decoding range used: | 29.9987 % |
| Missed worker deadlines: | 1 |

H

Figure 4.1: Web application interface for codec testing.
The interface consist of:

A. A volume slider which sets the value the GainNode (mentioned in Ch 3.6.1) scales the outgoing samples with.

B. Two radio buttons that sets if the audio samples are run through the codec process or just passed through to the speaker. It can be useful to compare different settings to the original audio.

C. Checkboxes for either sine signals in the displayed frequency or the ability to use the microphone as an input.

D. A slider to set the number of bits per sample used in the VQ, if the slider is to the far right the VQ is not used and the samples are only run through the DCT and iDCT process. Moving the slider to the left sets the "R" value introduced in Ch 3.5.

E. An option to start and stop real time visualization of the graphic cards output unto a canvas.

F. Three short sound clips which can be used as input. The clips have a broad difference in frequency and audio levels.

G. Scaling slider for the dct/idct process. With the slider to the far right it is set to use the automatic adjustment, mentioned in Ch 3.6.3. But it can also be set at predetermined values.

H. Real time display of the percentage of bits used for the DCT/iDCT from the GPU. The higher % the better the scaling is. The square on the far right blinks red whenever there is a risk for overflow. The last line counts the amount of workers that takes over 20 ms.

4.2 WebGL

4.2.1 Compatibility limitations

Firstly WebGL is a powerful tool for parallel computing but it can be difficult to utilize because of a lack of clear rules for its implementation on the graphic cards. This was first noticed when a loop greater than 960 iterations was attempted. The error message that was returned was that a loop greater than 240 wasn’t allowed, this error message was however gone when the loop iterations was decreased to 959. From this it can be concluded that the graphics driver automatically applies an amount of loop unrolling that the user has no control over, i.e increasing the speed of the loop by reducing instructions that control the loop, for example pointer arithmetic and loop conditions. The same code
was then tested on a different computer and the result was no error message no matter how many iteration set. So the code is implemented differently on different hardware and can be limited at the manufacturer’s discretion. Similar error messages was also received when using if-statements within large loops.

4.2.2 Output and operators

Another drawback with WebGL is that it is only through 8-bit values large sets of numbers can be sent and received, even though WebGL works internally with 32-bit floats. This means a lot of shuffling between buffers and calculating values in JavaScript because 8-bit values are not always sufficient, as in the case of audio coding. The 8-bit values are also unsigned, meaning negative values need to be converted manually. This is difficult since WebGL lacks support for bitwise ’or’ and ’and’ operation, also ’remainder’ is not implemented and there are no definite answers to when these will be implemented [7]. Bitwise shift is also a desired operator that isn’t implemented. This means the code becomes more complex where these operators could have been used. For example it made me decide to use one complement values instead of two complement for interface between JavaScript and WebGL. There isn’t a big difference in output but I get one value less in dynamic range and get a positive and negative zero value.

4.2.3 Parallelism

The power with WebGL is its ability to process the same code multiple times simultaneously. But it is difficult to utilize that power when in many cases it is desirable to calculate something from a previously calculated number.

For example if WebGL would be used to calculate a Fibonacci sequence. Lets say the first 2 pixels are programmed to return 1’s, the third pixel would then return the sum of the first two pixels. The problem is that there is no way for the third pixel to access the values from the first two. The solution would be to calculate which number in the series that pixel would return and then calculate all the previous numbers. This isn’t a problem for the first numbers but it will quickly become large calculations that needs to be done on each pixel. This example would be much better suited for execution in JavaScript.

A related problem is when the outcome isn’t known it can be hard to scale the output to maximize the dynamic range. The WebGL output has four 8-bit values per pixel, that means 0-255 per pixel. If the calculations in WebGL returns 255, 300 or 1 000 000 the output will saturate to 255. Lets take the example with the Fibonacci sequence and lets say that each pixel can calculate the value corresponding to their Fibonacci index. The first 12 values would be correctly shown but when the index reaches 13 the Fibonacci value is 377. Usually this can easily be adjusted by scaling all numbers by an arbitrary number so it ends up within the correct range. However since the pixels don’t know the output of any other pixel it is impossible to know with which number to scale.
4.3. WebWorkers

4.2.4 The efficient side

The majority of my testing was done with an integrated graphic processor, which in many cases isn’t as powerful as dedicated graphics cards. When trying to implement the DCT calculations straight into JavaScript the execution time took towards 200 ms, that improved drastically when implemented into a worker where it went down to around 60 ms. Both those cases were too long for the expected maximum of 20 ms, which WebGL easily handles, see Table 4.1. These values were from a laptop and would probably be a lot lower on a high end processor, but it still shows that the graphic card on a less powerful machine can easily do heavy computations.

<table>
<thead>
<tr>
<th>DCT Implementation</th>
<th>Calculation time</th>
</tr>
</thead>
<tbody>
<tr>
<td>JavaScript</td>
<td>200 ms</td>
</tr>
<tr>
<td>Web worker</td>
<td>60 ms</td>
</tr>
<tr>
<td>WebGL</td>
<td>&lt; 5 ms</td>
</tr>
</tbody>
</table>

Table 4.1: Shows the calculation time for different implementations of the same DCT

The exact execution time is hard to calculate for WebGL calculations since there isn’t anything that indicates when the calculations are done. The only way to know when the values are updated is to read the canvas or frame buffer and look for changes and this happens to be the heaviest drain on the computer resources. When I tested the calculation time for WebGL I could read the frame buffer a millisecond after I sent the calculation signal and get the right result. I put it as below 5 milliseconds to be a bit on the safe side and to take the overhead into consideration which the other two methods doesn’t need.

Other than that the implemented codec isn’t particularly calculation heavy, but there is a lot of overhead to use WebGL. The values that are being used for calculations needs to be converted to values that can be used on the graphics card and sent into textures. This isn’t ideal for a language like JavaScript that in its core is loosely typed.

The values that come out from WebGL often needs to be converted if they are being used somewhere else. The same conversion calculations needs to be performed inside WebGL and it is evident that coefficient values over 8-bits weren’t really considered when WebGL was implemented.

4.3 WebWorkers

WebWorkers are a good middle solution between JavaScript and WebGL. They are able to perform heavy calculations without blocking the main thread while still being able to
use the same data types as the main thread when sending and receiving results unlike WebGL. Unlike WebGL the WebWorkers are limited by the CPU and can be affected by what goes on in the main script. Early into the implementation I had a worker that normally finished its calculations within 1 ms but some times it took over 20 ms for it to return values. When everything in the audio chain depends on those 20 ms it wasn’t an option to sometimes miss this deadline. I tried to find the fault by watching the browsers built in CPU profiles, but those showed no unnatural CPU activity that could explain the problem. I was lucky and found the problem by chance when I stopped reading the WebGL canvas, I found that reading the canvas is resource heavy enough on JavaScript that when they happened at the same time the worker couldn’t make its deadlines. This was later fixed by writing and reading a frame buffer instead of a canvas. I also found that dragging the window will stop the worker results from coming into the main script even though the main script runs fine.

4.4 AudioWorkers

Initially it was meant to implement and evaluate the audio related code into an AudioWorker, which basically is a WebWorker with direct access to the AudioContext (see Ch 3.6.1). It was quickly found out that AudioWorkers wasn’t yet implemented in any browser [10]. Firefox and Chrome are the browsers that seems in forefront but there isn’t yet a set release date. From what I have gathered much of my work will easily be implemented into an AudioWorker as they seem to be based on the existing web audio API. I personally think AudioWorkers will improve the overall usefulness of web audio as it places time critical computation directly into a worker which doesn’t really need any communication with the main script. The problem with AudioWorkers and my work is that there isn’t a way for a worker to directly communicate with WebGL. So if someone wants to utilize that resource they need to go through the main script and that will drastically increase the chances of ”deadlines” being missed.

AudioWorkers wasn’t the only thing that didn’t turn out as hoped. Web Audio has a built in FFT in the AnalyserNode, but it was found that it outputs the amplitude of the FFT without the phase, which isn’t as useful for this application. It’s evident that its existence isn’t tailored for advanced audio use but implemented to visualize an equalizer. Because of the nature of the WebGL implementation it helps to know an approximation of the amplitude of the DCT output, and the amplitude value from the AnalyzerNode can therefore help with dynamics for the values sent out from WebGL.

4.5 Vector quantization

The implemented vector quantization allows the codec to send information in bit-rates ranging from 78.8 kbits/s to 307.2 kbits/s. But the lower bitrates doesn’t produce good
enough audio to be usable. My subjective evaluation is that the VQ manages good enough quality down to 192 kbits/s or 4 bits/sample. The transfer rates could be lowered even more by adding a few more samples into each band. With double the samples per band it would mean good enough audio with bit-rates around 100 kbits/s which would be better and closer to the initial goal for the codec. Other methods to reduce bit-rate would be to change how the bits are distributed over the frequency range, it isn’t necessary to have as many bits at higher frequencies as on low since it is harder to hear the difference on high frequencies (described in Ch 3.4). The bits are in the current implementation uniformly distributed across the whole frequency band.

The first implementation of the VQ in WebGL used only 3 bits for scaling to maximize the obtainable R-value, but it was found that 3 bits scaling wasn’t sufficient. This can be seen in Fig 4.2. The R value in the figure is adjusted to not include scaling bits to be easier to compare the different results. From the image it can be seen that 3 bit scaling reaches their best signal-to-noise ratio (SNR) already at the R value 3.4, which means increasing R further doesn’t improve the audio quality. Increasing the scaling to 5 bits gives a big boost in SNR but a limitation in scaling can be seen for R values over

![SNR for different scaling bits](image-url)
4.2. 7 bits scaling increases over the whole spectrum in the figure and isn’t far from the
minimized mean squared error (MSE) scaling in SNR. Increasing the scaling bits to 7
from 3 limits the amount of bits that can be used for codewords. But since there are no
improvements in the extra codeword bits for the 3 and 5 scaling bits it was decided to
use 7 scaling bits to actually get a difference between all selectable R.

In Fig 4.3 it is shown how the SNR from the vector quantization behaves with minimized
MSE scaling and high values on R. It shows that with as good scaling as possible the
vector quantization algorithm stops improving the SNR values at R larger than 11. With
the current implementation R higher than 5 can’t be used because of limitation in the
amount of bits that WebGL is able to output. These high R values would be possible to
obtain if a viable solution of using more than 1 pixel per output for each calculation was
implemented or if the VQ was implemented straight into JavaScript.

![SNR of shape quantization](image.png)

*Figure 4.3: SNR for the implemented VQ with optimal MSE scaling.*
4.6 Future work

The results presented in this thesis can be seen as a proof of concept. It shows that it can be done, but is still not a fully functional codec. To improve the codec I would suggest to implement a transient detector and a way to handle transients, since I think that is a remaining issue with respect to the audio quality. Also a bit allocator and overhead for sending or storing the VQ values would be a high priority to allow this to be implemented on more than one device. For continued work with computational heavy audio implementations in web environment I would recommend to wait until AudioWorkers are implemented and see what differences they bring. I would also suggest to research the compatibility with smartphones, since they have less computational power than computers and could really benefit from GPU solutions. Testing could also be done on the effects of WebGL implemented calculations when other applications use the graphic card’s resources e.g games or videos. The vector quantizer could probably also be improved to a more computation heavy algorithm, as it is now it probably doesn’t need to be calculated on the graphics card or improved to be able to use higher R than the current limitation of 6.4 bits/coefficient.
REFERENCES


APPENDIX A

Codec, main code

```html
<!DOCTYPE html>
<html lang="en">
<style>
table, th, td {
  border: 0px solid black;
  border-collapse: collapse;
}
th, td {
  padding: 10px;
}
tr.Border td { border: 1px solid black; }
</style>
<body>
<meta charset="utf-8" />
<form>
<input type="range" id="Blength" min="0" max="10" value="1" onchange="updatevalue(this.value)"> Volume
</form>
<form>
<input type="radio" id="codec" name="codec" value="cod" checked> Using codec  
<br>
<input type="radio" id="codec" name="codec" value="nocod"> No codec
</form>
<br>
Set number of bits/sample in VQ:
<form>
<input type="range" id="VQ" min="0" max="6" value="6" onchange="updateVQvalue(this.value)">
</form>
</body>
```

37
Appendix A

```html
<table style="width:10%" id="BitsTable">
  <tr class="Border">
    <td>Bits/sample:</td>
    <td>16</td>
  </tr>
  <tr class="Border">
    <td>Bitrate: [kbit/s]</td>
    <td>768</td>
  </tr>
</table>

<br>
<br>
<br>
<button onclick="Visualize()">Real time visualisation</button> (high performance cost!)
<br>
<br>
<canvas id="canvas" width="8" height="8">test</canvas>
<canvas id="VQcanvas" width="32" height="32"></canvas>
<canvas id="Vis" width="32" height="32"></canvas>
<script src="webgl-utils.js"></script>
<br>
<br>
<button onclick = "play.soundclip(0)">Beethoven soundclip</button>
<br>
<button onclick = "play.soundclip(1)">Mozart soundclip</button>
<br>
<button onclick = "play.soundclip(2)">Brahms soundclip</button>
<br>
<br>
<form>
  <input type="range" id="Scaling" min="0" max="10" value = "10" onchange="updatescalevalue(this.value)"> Scaling
</form>
<br>
<br>
<div id="enc"></div>
<br>
<br>
<table style="width:30%" id="table">
  <tr>
    <td></td>
    <td></td>
    <td>Overflow warning:</td>
  </tr>
  <tr class="Border">
    <td width="35%">Encoding range used:</td>
    <td>0%</td>
    <td width="20%"></td>
  </tr>
</table>
```
<table>
<thead>
<tr>
<th>Decoding range used</th>
<th>0%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Missed worker deadline</td>
<td>0</td>
</tr>
</tbody>
</table>

```javascript
// mono
var volume = 1/20;

function updatevalue(gainval) {
    volume = gainval / 8;
    console.log("volume: "+gainval);
}

var scaling = 10;

function updatescalevalue(value) {
    scaling = value;
    console.log("scaling: "+value);
}

var firefox;
if (navigator.userAgent.toLocaleLowerCase().indexOf('firefox') > -1) {
    firefox = true;
}

var viz = false;
function Visualize() {
    if (viz) {
        viz = false;
    } else {
        viz = true;
    }
}

var R = 6;
function updateVQvalue(VQval) {
    // Calculates amount of bits/s
    var BitsTableV = document.getElementById('BitsTable');
    if (VQval < 6) {
        BitsTableV.rows[0].cells[1].innerHTML = 12/5+VQval*4/5;
        BitsTableV.rows[1].cells[1].innerHTML = (12/5+VQval*4/5)*48000/1000;
    } else {
        BitsTableV.rows[0].cells[1].innerHTML = 16;
        BitsTableV.rows[1].cells[1].innerHTML = 16*48000/1000;
    }
    console.log("VQ: "+VQval);
    R = VQval;
}
```

```javascript
var MusicSource;
var MusicBuffer = {};
```
var playing = false;

play.soundclip = function(n){
  if (play._playing[n]){
    play.stop(n);
  } else {
    play.source[n] = audioCtx.createBufferSource();
    play.source[n].buffer = MusicBuffer[n];
    play.source[n].loop = true;
    play.source[n].connect(analyser);
    play.source[n].start(0);
    play._playing[n] = true;
    playing = true;
  }
}

play.stop = function(n){
  play.source[n].stop(0);
  play._playing[n] = false;
}

/* --- Canvas Parameter --- */

var width = 32; // faktor 2, width*height >= 256
var height = 32;
canvas.width = width;
canvas.height = height;

window.onload = main;

/* --- Global Parameter --- */

var audioCtx;
var analyser;
var musicgain;

var texture = [ new Uint8Array(width*height*4), new Uint8Array(16*16*4) ]; // 4096 for Values into DCT and 1024 for values from VQ
var VQtexture = new Uint8Array(width*height*4); // 4096 for values from the DCT

var playbackbuffer = Array.apply(null, new Array(1920)).map(Number.prototype.valueOf,0); //Buffer for outgoing audio

var buf = [ new Uint8Array(width*height*4), new Uint8Array(16*16*4) ]; //Values from canvas is placed here

var soundBuffer = Array.apply(null, new Array(960)).map(Number.prototype.valueOf,0); // Buffer for incoming audio

var preframesamp = Array.apply(null, new Array(960)).map(Number.prototype.valueOf,0);

var frameindex = 0; // Shows how many frames that has been processed

var SoundClipBuffer;

/* --- Main Script --- */

function main(image) {
  if (!!window.Worker){
    var fftworker = new Worker("encoder.js");

    // Code for processing audio
  }
}
41

```javascript
var decworker = new Worker("decoder.js")
else{
  console.log("Ingen worker :(")
}

/* --- WebGL context --- */
var gl = canvas.getContext("webgl", {preserveDrawingBuffer: true, antialias: false})
|| canvas.getContext("experimental-webgl", {preserveDrawingBuffer: true, antialias: false});

/* --- VQ --- */
var VQgl = VQcanvas.getContext("webgl", {preserveDrawingBuffer: true, antialias: false})
|| canvas.getContext("experimental-webgl", {preserveDrawingBuffer: true, antialias: false});

if (!gl) {
  console.log("WebGL not implemented!")
  return;
}

/* --- Precision --- */
var FragFloatPrec = gl.getShaderPrecisionFormat(gl.FRAGMENT_SHADER, gl.HIGH_FLOAT); // Shows WebGL precision
console.log(FragFloatPrec)
var FragIntPrec = gl.getShaderPrecisionFormat(gl.FRAGMENT_SHADER, gl.HIGH_INT);
console.log(FragIntPrec)

/* --- GLSL/Shader Set-up --- */
var program = createProgramFromScripts(gl, ["2d-vertex-shader", "2d-fragment-shader"])
|| Imports WebGL shaders
  gl.useProgram(program); // Sends the Shaders to the GPU

var VQprogram = createProgramFromScripts(VQgl, ["2d-vertex-shader", "2dVQ-fragment-shader"]);
VQgl.useProgram(VQprogram);

// look up where the vertex data needs to go.
var positionLocation = gl.getAttribLocation(program, "a_position");

var VQpositionLocation = VQgl.getAttribLocation(VQprogram, "a_position");

/* --- Texture coordinates --- */
var buffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, buffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array([ // Set up to span whole canvas
    0.0, 0.0,
    0.0, 1.0,
    1.0, 0.0,
    1.0, 1.0
  ]), gl.STATIC_DRAW);
  gl.enableVertexAttribArray(positionLocation);
  gl.vertexAttribPointer(positionLocation, 2, gl.FLOAT, false, 0, 0);

/* --- VQ --- */
```
var VQbuffer = VQgl.createBuffer();
VQgl.bindBuffer(VQgl.ARRAY_BUFFER, VQbuffer);
VQgl.bufferData(VQgl.ARRAY_BUFFER, new Float32Array([0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 1.0, 1.0]), VQgl.STATIC_DRAW);
VQgl.enableVertexAttribArray(VQpositionLocation);
VQgl.vertexAttribPointer(VQpositionLocation, 2, VQgl.FLOAT, false, 0, 0);

/* --- Create Framebuffer --- */
var Framebuffer = gl.createFramebuffer();
gl.bindFramebuffer(gl.FRAMEBUFFER, Framebuffer);
var outtex = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, outtex);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, width, height, 0, gl.RGBA, gl.UNSIGNED_BYTE, null);

var renderbuffer = gl.createRenderbuffer();
gl.bindRenderbuffer(gl.RENDERBUFFER, renderbuffer);
gl.renderbufferStorage(gl.RENDERBUFFER, gl.DEPTH_COMPONENT16, 32, 32);
gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0, gl.TEXTURE_2D, outtex, 0);
gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT, gl.RENDERBUFFER, renderbuffer);

/* --- VQ --- */
var VQFramebuffer = VQgl.createFramebuffer();
VQgl.bindFramebuffer(VQgl.FRAMEBUFFER, VQFramebuffer);
var VQouttex = VQgl.createTexture();
VQgl.bindTexture(VQgl.TEXTURE_2D, VQouttex);
VQgl.texImage2D(VQgl.TEXTURE_2D, 0, VQgl.RGBA, 32, 32, 0, VQgl.RGBA, VQgl.UNSIGNED_BYTE, null);

var VQrenderbuffer = VQgl.createRenderbuffer();
VQgl.bindRenderbuffer(VQgl.RENDERBUFFER, VQrenderbuffer);
VQgl.renderbufferStorage(VQgl.RENDERBUFFER, VQgl.DEPTH_COMPONENT16, 32, 32);
VQgl.framebufferTexture2D(VQgl.FRAMEBUFFER, VQgl.COLOR_ATTACHMENT0, VQgl.TEXTURE_2D, VQouttex, 0);
VQgl.framebufferRenderbuffer(VQgl.FRAMEBUFFER, VQgl.DEPTH_ATTACHMENT, VQgl.RENDERBUFFER, VQrenderbuffer);

/* --- Create Texture --- */
var tex = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, tex);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST); // Prints error without this
var VQdetex = gl.createTexture();
gl.bindTexture(gl.TEXTURE_2D, VQdetex);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST); // Prints error without this
var u_image0loc = gl.getUniformLocation(program, "u_image0"); // Set up GPU memory location for 2 textures
var u_image1loc = gl.getUniformLocation(program, "u_image1");
gl.uniform1i(u_image0loc,0);
gl.uniform1i(u_image1loc,1);
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, tex);
gl.activeTexture(gl.TEXTURE1);
gl.bindTexture(gl.TEXTURE_2D, VQdetex);

var VQtex = VQgl.createTexture();
VQgl.bindTexture(VQgl.TEXTURE_2D, VQtex);

VQgl.texParameteri(VQgl.TEXTURE_2D, VQgl.TEXTURE_MIN_FILTER, VQgl.NEAREST); // Prints
error without this

/* --- Check Web Audio --- */
navigator.getUserMedia = (navigator.getUserMedia ||
navigator.webkitGetUserMedia ||
navigator.mozGetUserMedia ||
navigator.msGetUserMedia);

if (navigator.getUserMedia) {
  console.log("getUserMedia funkar");
navigator.getUserMedia(
audio:true),
Success
, function(err){
  console.log("Error: " + err)
});
} else{
  console.log("getUserMedia doesn’t exist!")
}

/* --- Web Audio Context --- */

function Success(e) {
  audioCtx = new (window.AudioContext || window.webkitAudioContext)();
  var source = keep(audioCtx.createMediaStreamSource(e));

  var BufferSize = width*height/2; // 512;
  //console.log(audioCtx.sampleRate) //48kHz
  var scriptNode = keep(audioCtx.createScriptProcessor(BufferSize, 1, 1));
  var oscillator = audioCtx.createOscillator(); // Sine generator
  var oscillator2 = audioCtx.createOscillator();
  var oscillator3 = audioCtx.createOscillator();
  analyser = audioCtx.createAnalyser();
  var gain = audioCtx.createGain();
  musicgain = audioCtx.createGain(); // Scales down the sine signals
  musicgain.gain.value = 0.3;
  SoundClipBuffer = new BufferLoader(
audioCtx,
[
'Sound/brahms_pcl.wav',
'Sound/mozart_requiem.wav',
'Sound/beethoven_missa.wav',
],
Loaded);
  SoundClipBuffer.load();
function Loaded(buffervals){
  MusicSource = audioCtx.createBufferSource();
  MusicBuffer[0] = buffervals[2];
  MusicBuffer[1] = buffervals[1];
  MusicBuffer[2] = buffervals[0];
}

var fftArray = new Float32Array(analyser.frequencyBinCount);
oscillator.type = "sine"; // sine, square, sawtooth, triangle, custom
oscillator.frequency.value = 300; //Hz
oscillator.start();
oscillator2.type = "sine"; // sine, square, sawtooth, triangle, custom
oscillator2.frequency.value = 5000; //Hz
oscillator2.start();
oscillator3.type = "sine"; // sine, square, sawtooth, triangle, custom
oscillator3.frequency.value = 11000; //Hz
oscillator3.start();

scriptNode.onaudioprocess = function (ev) {
  var inputData = ev.inputBuffer.getChannelData(0);
  var outputData = ev.outputBuffer.getChannelData(0);
  soundBuffer.push.apply(soundBuffer, inputData);
  gain.gain.value = volume;
  var output = (soundBuffer.length > 1857) ? true : false; // Checks that soundbuffer contains 960 new values
  analyser.getFloatFrequencyData(fftArray); // Gets the dB values from the built in FFT

  var len = 512, fftmax = -99999;
  while (len--){ // Calculates the max dB value from the FFT
    if (fftArray[2*len] > fftmax) {
      fftmax = fftArray[2*len];
    }
    if (fftArray[2*len-1] > fftmax) {
      fftmax = fftArray[2*len-1];
    }
  }

  if (firefox){
    fftmax = fftmax-7.5; // Adjusts the dB value if in Firefox
  }

  if (output){
    var intid = new Date().getTime() // checks the input time to later calculate Worker timing
    fftworker.postMessage([frameindex, soundBuffer.slice(0,1920), buf[0], intid, fftmax]); // Sends values to worker
  }

  var tableV = document.getElementById('table');
  if (frameindex != decframe+1){ // Checks if workers are running behind and displays that on webpage
    preframesamp = Array.apply(null, new Array(960)).map(Number.prototype.valueOf, 0);
    tableV.rows[3].cells[1].innerHTML = Number(tableV.rows[3].cells[1].innerHTML) + 1;
    tableV.rows[3].cells[2].style.backgroundColor = "red";
  }
  tableV.rows[3].cells[2].style.backgroundColor = "white";
}
var a = maxframe.indexOf(frameindex-1);
var b = encmax[a];
maxframe.splice(0,a);
encmax.splice(0,a);
deckworker.postMessage([buf, preframesamp, intid, frameindex, b, R]); // Sends values to decoding
soundBuffer.splice(0,960);
frameindex++
}

var codecstate = document.getElementsByTagName("codec")[0].checked ? 'cod' : 'nocod'; // Checks if decode is chosen on webpage
if (codecstate == "cod") { // sets which values that are set into outgoing audio
    for (var n = 0; n < 128; n++) {
        outputData[4*n] = playbackbuffer[4*n]; // sets the encoded values
        outputData[4*n+1] = playbackbuffer[4*n+1];
        outputData[4*n+2] = playbackbuffer[4*n+2];
        outputData[4*n+3] = playbackbuffer[4*n+3];
    }
} else {
    for (var n = 0; n < 128; n++) {
        outputData[4*n] = inputData[4*n]; // just copies the input data
        outputData[4*n+1] = inputData[4*n+1];
        outputData[4*n+2] = inputData[4*n+2];
        outputData[4*n+3] = inputData[4*n+3];
    }
}
playbackbuffer.splice(0,512);
VQgl.uniform1f(VQgl.getUniformLocation(VQprogram, "R"), R); // Sends R value to GPU

var boxes = document.getElementsByTagName("tone");
if (boxes[0].checked) { // Connects inputs to rest of audio chain
    source.connect(analyser);
} else {
    source.disconnect();
}
if (boxes[1].checked) {
    oscillator.connect(musicgain);
} else {
    oscillator.disconnect();
}
if (boxes[2].checked) {
    oscillator2.connect(musicgain);
} else {
    oscillator2.disconnect();
}
if (boxes[3].checked) {
    oscillator3.connect(musicgain);
} else {
    oscillator3.disconnect();
}
```javascript
/* --- Connects The Nodes ---*/
musicgain.connect(analyser); // connects musicgain to Analyser
analyser.connect(scriptNode);
scriptNode.connect(gain);
gain.connect(audioCtx.destination); //Connects the last gain node to speakers

var oldframe = -1;
var time = []
var tidvarning=0;
var encmax = [1]
var maxframe = [1]
var oldscale = 0;

fftworker.onmessage = function(e){ // Receives data from FFT worker
  encmax[encmax.length] = e.data[3];
texture[0] = e.data[2]; // receives texture
  var tid = e.data[1]; // time worker started
var outframe = e.data[0]; // which frame it worked on
var fftmaxval = e.data[4]
maxframe[maxframe.length] = outframe;

  time[outframe] = new Date().getTime()-tid;
var scale = 0;
if (scaling == 10) { //Automatic scaling
  scale = Math.pow(10,fftmaxval)+2 // Can be done better, First scaling algorithm was made useless by Chrome update
  var scale2 = 1.9*100/ oldscale; // Scale for iDCT, oldscale because it works on a previous "frame"
  oldscale = scale;
} else {
  scale = Math.pow(10,-1)*(scaling+1)/3; //Uses value from user input slider
  var scale2 = 200/oldscale;
  oldscale = scale;
}

//scale2 = 100.0;

gl.uniform1f(gl.getUniformLocation(program, "skalning"), scale) // Send values to GPU
  gl.uniform1f(gl.getUniformLocation(program, "skalning2"), scale2)
var flag = true;

buf = glfunction(gl,VQgl,tex,VQdetex,flag);
oldframe = outframe;

var test3 =1;
var decframe = 0;
var framecount = 0;

decworker.onmessage = function(e){ // Receives values from Decoding worker
decframe = e.data[0];
  var dectid = e.data[1];
var sound = e.data[2]; //The decoded sound
var max = e.data[4]; //The dynamic range used
var max2 = e.data[6]; //The dynamic range used
test3 = e.data[6];
```

var test4 = e.data[8];

if (decframe % 5 == 0) // only checks every fifth frame to reduce load
{
  skalning( max , "dec")
}

if (decframe % 5 == 2)
{
  skalning( max2 , "enc")
}

preframesamp = e.data[3];

if(framecount == 0 && playing == true)
{
  framecount = decframe;
}

if (frameindex - decframe < 2){ // applies the decoded audio to outputbuffer if the worker is in sync
  playbackbuffer.push.apply(playbackbuffer, sound);
}

var scriptNodes = {};

var keep = (function () { // Needed to avoid ScriptNode to get garbage collected
  var nextNodeID = 1;
  return function (node) {
    node.id = node.id || (nextNodeID++);
    scriptNodes[node.id] = node;
    console.log(scriptNodes);
    return node;
  };
})(function ());

function skalning( value , b) // Calculates dynamic range used and displays on webpage
{
  var tableVar = document.getElementById('table');

  if ( b == "dec")
  {
    var asdf = 100 * value;
    tableVar.rows[2].cells[1].innerHTML = asdf.toFixed(4) + " % ";
    if (asdf/100 > 254/255)
    {
      tableVar.rows[2].cells[2].style.backgroundColor = "red";
    }
    else{
      tableVar.rows[2].cells[2].style.backgroundColor = "white";
    }
  }
  else{
    var asdf = 100 * value/32767;
    tableVar.rows[1].cells[1].innerHTML = asdf.toFixed(4) + " % ";
    if (asdf/100 > 254/255)
    {
      tableVar.rows[1].cells[2].style.backgroundColor = "red";
    }
    else{
      tableVar.rows[1].cells[2].style.backgroundColor = "white";
    }
  }
}
function glfunction(glin, VQglin, tex, VQdetex, flag)
{
  if (flag){
    for (var p=0; p < 4096; p++) {
      VQtexture[p] = buf[0][p];
    }
  }
  texture[1] = buf[1];
  glin.bindTexture(glin.TEXTURE_2D, tex); // Updates the Textures to be used by GPU
  glin.texImage2D(glin.TEXTURE_2D, 0, glin.RGBA, width, height, 0, glin.RGBA, glin.UNSIGNED_BYTE, texture[0]);
  glin.bindTexture(glin.TEXTURE_2D, VQdetex);
  glin.texImage2D(glin.TEXTURE_2D, 0, glin.RGBA, 16, 16, 0, glin.RGBA, glin.UNSIGNED_BYTE, texture[1]);
  VQglin.texImage2D(glin.TEXTURE_2D, 0, glin.RGBA, width, height, 0, glin.RGBA, glin.UNSIGNED_BYTE, VQtexture);
  glin.readPixels(0,0, width, height, glin.RGBA, glin.UNSIGNED_BYTE, buf[0]); // Resource heavy!! // Reads the results from WebGL into buf
  VQglin.readPixels(0,0,16,16,VQglin.RGBA,VQglin.UNSIGNED_BYTE,buf[1]); // Resource heavy!!
  glin.drawArrays(glin.TRIANGLE_STRIP, 0, 4); // Starts the GPU calculations
  VQglin.drawArrays(glin.TRIANGLE_STRIP, 0, 4);
  if (viz) {
    printTexture(buf[0]);
    printTexture(buf[1]);
  }
  return buf
}

function printTexture(texture){ // Displays the result from WebGL
  var Viscanvas = document.getElementById('Vis');
  Viscanvas.width = 32;
  Viscanvas.height = 32;
  var context = Viscanvas.getContext("2d");
  var Data = context.createImageData(32,32);
  Data.data.set(texture);
  context.putImageData(Data,0,0);
  var img = new Image();
  img.src = Viscanvas.toDataURL();
  context.drawImage(img,0,0)
}

function BufferLoader(context, urlList, callback) {
  this.context = context;
  this.urlList = urlList;
  this.onload = callback;
  this.bufferList = new Array();
  this.loadCount = 0;
}

BufferLoader.prototype.loadBuffer = function(url, index) {
  var request = new XMLHttpRequest();
  request.onreadystatechange = function() {
    if (this.readyState === 4) {
      if (this.status === 200) {
        var data = this.responseText;
        this.bufferList[index] = data;
      } else {
        console.log('Failed to load URL ' + url);
      }
    }
  };
  request.open("GET", url, true);
  request.send(null);
}
```
var loader = this;

request.onload = function() {
    // Asynchronously decode the audio file data in request.response
    loader.context.decodeAudioData()
        request.response,
        function(buffer) {
            if (!buffer) {
                alert('error decoding file data: ' + url);
                return;
            }
            loader.bufferList[index] = buffer;
            if (++loader.loadCount == loader.urlList.length)
                loader.onload(loader.bufferList);
        },
        function(error) {
            console.error('decodeAudioData error', error);
        });

    request.onerror = function() {
        alert('BufferLoader: XHR error');
    }
    request.send();
}

BufferLoader.prototype.load = function() {
    for (var i = 0; i < this.urlList.length; ++i)
        this.loadBuffer(this.urlList[i], i);
}

<script id="2d-vertex-shader" type="x-shader/x-vertex">
attribute vec2 a_position;
uniform float skalning;
uniform float R;

varying vec2 v_texCoord;

void main() {
    vec2 test = a_position*2.0; // from 0->1 to 0->2
    vec2 test2 = test-1.0; // from 0->2 to -1->1
    gl_Position = vec4(test2, 0, 1);
}
</script>

<script id="2d-fragment-shader" type="x-shader/x-fragment">
precision highp float; // highp
uniform float R;

//-----------DCT and iVQ+iDCT-------------//

// The textures
uniform sampler2D u_image0; // read as values between 0 and 1.0
uniform sampler2D u_image1;
```
uniform float skalning; // Single values sent in from JavaScript
uniform float skalning2;
#define points 240.0
#define pi 3.1415926535897932384626433832795
// the texCoords passed in from the vertex shader.

varying vec2 v_texCoord; // read as value between 0 and 1.0
vec2 position1;
vec2 position2;
vec2 position3;
vec2 position4;
vec2 position5;
vec2 pos;

float RAD2DEG = 180.0/pi;
float delta = 90.0/exp2(R);
float offset = delta/2.0;
float d = 4.0;
float e = 4.0;
float f = 8.0;
float g = 8.0;
float k = 0.0;
float red; // encode high bits
float green; // encode low bits
float alpha; // decode high bits
vec4 colvalues1;
vec4 colvalues2;
vec4 colvalues3;
vec4 colvalues4;
vec4 colvalues5;
float quad0;
float quad1;
float quad2;
float quad3;
float bluesum=0.0;
float greensum=0.0;
float redsum=0.0;
float alphasum=0.0;
float cosval1=0.0;
float cosval2=0.0;
float cosval3=0.0;
float cosval4=0.0;
float decsum = 0.0;
float encsum = 0.0;
float encval1 = 0.0;
float encval2 = 0.0;
float encval3 = 0.0;
float encval4 = 0.0;
float test = 0.0;
float test2 = 0.0;
float test3 = 0.0;
float scaletop;
float scale;
float sum;
float rms = 1.0;
float val1;
float val2;
float val3;
float val4;
float val5;
float ind1;
float ind2;
float ind3;
float ind4;
float prov1;
float prov2;
float prov3;
float prov4;
float prov5;

void main()
{
// v_texCoord.r coordinates in x 4..12..20...252
// v_texCoord.t coordinates in y
k = (v_texCoord.r - 4.0 / 256.0) / 8.0 + 32.0 * (v_texCoord.t - 4.0 / 256.0) / 8.0; // Calculates current pixel
k = 256.0 * k; // 0 - 960
for (float l = 0.0; l < 240.0; l++) { // unrolled to 240 iterations instead of 960

if (e > 230.0)
{
    d += 8.0;
    e = 4.0;
}

if (g > 253.0)
{
    f += 16.0; // row
    g = 8.0; // column
}

position1 = vec2(e / 256.0, d / 256.0); // Position in Texture to fetch data from
position2 = vec2((e + 8.0) / 256.0, d / 256.0);
position3 = vec2((e + 16.0) / 256.0, d / 256.0);
position4 = vec2((e + 24.0) / 256.0, d / 256.0);
e += 32.0;
pos = vec2(g / 256.0, f / 256.0);
g += 16.0;

colvalues1 = texture2D(u_image0, position1);
colvalues2 = texture2D(u_image0, position2);
colvalues3 = texture2D(u_image0, position3);
colvalues4 = texture2D(u_image0, position4);

if (R > 5.0) { // Fetches values that hasn't gone through VQ only DCT

cosval1 = cos((4.0 * 1.0 + 0.5) * (k + 0.5) * pi / 960.0);
cosval2 = cos((4.0 * 1.0 + 1.5) * (k + 0.5) * pi / 960.0);
cosval3 = cos((4.0 * 1.0 + 2.5) * (k + 0.5) * pi / 960.0);
cosval4 = cos((4.0 * 1.0 + 3.5) * (k + 0.5) * pi / 960.0);

if (colvalues1[2] > 127.0 / 255.0) { // DECODE
decval1 = -(255.0 * colvalues1[2] - 128.0 + colvalues1[3]); // Restore the real value from 2 8-bit values
}
else{
decval1 = 255.0 * colvalues1[2] + colvalues1[3];
}
}
if (colvalues2[2] > 127.0/255.0 ){
    decval2=-(255.0*colvalues2[2]-128.0+colvalues2[3]);
} else{
    decval2=255.0*colvalues2[2]+colvalues2[3];
}

if (colvalues3[2] > 127.0/255.0 ){
    decval3=-(255.0*colvalues3[2]-128.0+colvalues3[3]);
} else{
    decval3=255.0*colvalues3[2]+colvalues3[3];
}

if (colvalues4[2] > 127.0/255.0 ){
    decval4=-(255.0*colvalues4[2]-128.0+colvalues4[3]);
} else{
    decval4=255.0*colvalues4[2]+colvalues4[3];
}

decsum += decval1*cosval1+decval2*cosval2+decval3*cosval3+decval4*cosval4; // decode
}
else {
    // Decoding values from VQ
    rms = 1.0; // Might not be needed
    cosval1 = cos((5.0*1+0.5)*(k+0.5)*pi/960.0);
    cosval2 = cos((5.0*1+1.5)*(k+0.5)*pi/960.0);
    cosval3 = cos((5.0*1+2.5)*(k+0.5)*pi/960.0);
    cosval4 = cos((5.0*1+3.5)*(k+0.5)*pi/960.0);
    cosval5 = cos((5.0*1+4.5)*(k+0.5)*pi/960.0);
    colvalues5 = texture2D(u_image1,pos); // fetch values
    if (colvalues5[0] == 0.0) {
        val5 = 0.0;
        val4 = 0.0;
        val3 = 0.0;
        val2 = 0.0;
        val1 = 0.0;
    } else{
        quad0 = floor(255.0*colvalues5[0]/exp2(6.0)); // Calculate quad info
        quad2 = floor((255.0*colvalues5[0]-quad0*exp2(6.0)-quad1*exp2(5.0)-quad2*exp2(4.0)))/exp2(3.0));
        quad3 = floor((255.0*colvalues5[0]-quad0*exp2(6.0)-quad1*exp2(5.0)-quad2*exp2(4.0)-quad3*exp2(3.0)))/exp2(2.0));
        scaletop = 255.0*colvalues5[0]-floor(255.0*colvalues5[0]/exp2(3.0))*exp2(3.0); // Calculates the top part of scaling
        quad1 += 1.0; // Gets the right quadrant
        quad2 += 1.0;
        quad3 += 1.0;
        scale = exp2(4.0)*scaletop + floor(255.0*colvalues5[1]/exp2(4.0)); // Calculates the right scaling
    }
val5 = (255.0*colvalues5[1]-exp2(4.0)*floor(255.0*colvalues5[1]/exp2(4.0)))/255.0;
sum = 255.0*(exp2(16.0)*val5+exp2(8.0)*colvalues5[2]+colvalues5[3]);
ind1 = floor(sum/exp2(20.0-R)); //Decoding indexes
sum -= ind1*exp2(20.0-R);
ind2 = floor(sum/exp2(20.0-2.0*R));
sum -= ind2*exp2(20.0-2.0*R);
ind3 = floor(sum/exp2(20.0-3.0*R));
sum -= ind3*exp2(20.0-3.0*R);
ind4 = sum/exp2(20.0-4.0*R);
test3 = (ind4*delta+offset)/RAD2DEG+quad3*pi/2.0-pi;
val5 = sin(test3); //Decoding values
rms = cos(test3);
test3 = (ind3*delta+offset)/RAD2DEG+quad2*pi/2.0-pi;
val4 = rms*sin(test3);
rms = rms*cos(test3);
test3 = (ind2*delta+offset)/RAD2DEG+quad1*pi/2.0-pi;
val3 = rms*sin(test3);
rms = rms*cos(test3);
test3 = (ind1*delta+offset)/RAD2DEG+quad0*pi/2.0-pi;
val2 = rms*sin(test3);
rms = rms*cos(test3);
val1 = rms;
else if ( l > 191.0){ //Needed since this loop is differently unrolled than when decoding
    decsum += 0.0;
} else{
    decsum += exp2((scale+220.0)/5.0)/exp2(60.0)*(cosval5*val5+cosval4*val4+cosval3*
    val3+cosval2*val2+cosval1*val1);}
if (R < 6.0){
cosval1 = cos((4.0*1+0.5)*(k+0.5)*pi/960.0);
cosval2 = cos((4.0*1+1.5)*(k+0.5)*pi/960.0);
cosval3 = cos((4.0*1+2.5)*(k+0.5)*pi/960.0);
cosval4 = cos((4.0*1+3.5)*(k+0.5)*pi/960.0);
}
if (colvalues1[0] > 127.0/255.0 ){ //ENCODE
    encval1=(255.0*colvalues1[0]-128.0+colvalues1[1]); //Restore the real value from
    2 8-bit values
} else{
    encval1=255.0*colvalues1[0]+colvalues1[1];
}
if (colvalues2[0] > 127.0/255.0 ){
    encval2=(255.0*colvalues2[0]-128.0+colvalues2[1]);
} else{
    encval2=255.0*colvalues2[0]+colvalues2[1];
if (colvalues3[0] > 127.0/255.0 ){
    encval3=-(255.0*colvalues3[0]-128.0+colvalues3[1])
} else{
    encval3=255.0*colvalues3[0]+colvalues3[1]
}

if (colvalues4[0] > 127.0/255.0 ){
    encval4=-(255.0*colvalues4[0]-128.0+colvalues4[1])
} else{
    encval4=255.0*colvalues4[0]+colvalues4[1]
}

encsum += encval1 * cosval1 + encval2 * cosval2 + encval3 * cosval3 + encval4 * cosval4; //encode
}
decsum = decsum*skalning2; // Scales sum with value from JavaScript

// Sets the Decoded value into blue and alpha channel
if (decsum < 0.0) {
    bluesum = 128.0/255.0;
    decsum *= -1.0;
}
test2 = floor (decsum /255.0);
if (decsum > 32767.0) // overflow prevention
{
    bluesum += 127.0/255.0;
} else{
    bluesum += test2/255.0;
}
alphasum = decsum/255.0 - test2;

// Sets the Encoded value into red and green channel
if (encsum < 0.0) {
    redsum = 128.0/255.0;
    encsum *= -1.0;
}
encsum = skalning*encsum;
test = floor(encsum/255.0);
if (encsum > 32767.0) // overflow prevention
{
    redsum += 127.0/255.0;
} else{
    redsum += test/255.0;
}
greensum = encsum / 255.0 - test;
gl_FragColor = vec4(redsum, greensum, bluesum, alphasum);
</script>
<script id="2dVQ - fragment - shader" type="x-shader/x-fragment">
precision highp float; //highp
uniform sampler2D u_image;
uniform float R;
varying vec2 v_texCoord;
#define pi 3.1415926535897932384626433832795
float k = 0.0;
float d = 4.0;
float e = 0.0;
float colsum = 0.0;
float test = 0.0;
float val1 = 0.0;
float val2 = 0.0;
float val3 = 0.0;
float val4 = 0.0;
float val5 = 0.0;
float RAD2DEG = 180.0 / pi;
float delta = 90.0 / (pow(2.0, R));
float offset = delta / 2.0;
float skalning;
float quad;
float angle1;
float angle2;
float angle3;
float angle4;
float comp;
float rsq;
float a;
float b;
float angle_ind;
float bluesum = 0.0;
float greensum = 0.0;
float redsum = 0.0;
float alphasum = 0.0;
float totalsum = 0.0;
float scale = 0.0;
float scaleval = 0.0;
float maxval = 0.0;
float sinval = 0.0;
float partscale = 0.0;
vec2 position1;
vec2 position2;
vec2 position3;
vec2 position4;
vec2 position5;
vec2 testpos;
vec4 colvalues1;
vec4 colvalues2;
vec4 colvalues3;
vec4 colvalues4;
vec4 colvalues5;
vec4 angles;
vec4 values;
vec4 quads;
vec4 ind;
vec4 xVQ;
vec4 radians;

void main(){
  k = (v_texCoord.r-4.0/256.0)/8.0 + 16.0*(v_texCoord.t-4.0/256.0)/8.0; //Calculates pixel
  k = 256.0*k;
  //5*k
  e = floor(5.0*k/32.0); //y koordinat
  position1 = vec2((8.0*(5.0*k-32.0*e)+4.0)/256.0, (8.0*e+4.0)/256.0); //Calculates which
  // 5*k+1
  //values to read from texture
  //5*k+2
  e = floor((5.0*k+1.0)/32.0);
  position2 = vec2((8.0*(5.0*k+1.0-32.0*e)+4.0)/256.0, (8.0*e+4.0)/256.0);
  //5*k+3
  //5*k+4
  e = floor((5.0*k+4.0)/32.0);
  position5 = vec2((8.0*(5.0*k+4.0-32.0*e)+4.0)/256.0, (8.0*e+4.0)/256.0);
  // takes 5 values from DCT result texture
  colvalues1 = texture2D(u_image, position1);
  colvalues2 = texture2D(u_image, position2);
  colvalues3 = texture2D(u_image, position3);
  colvalues4 = texture2D(u_image, position4);
  colvalues5 = texture2D(u_image, position5);

  if (colvalues1[0] > 127.0/255.0){
    val1 = -(255.0*colvalues1[0]-128.0+colvalues1[1]); //Restores value from 2 8-bit
  }else{
    val1 = 255.0*colvalues1[0]+colvalues1[1];
  }

  if (colvalues2[0] > 127.0/255.0){
    val2 = -(255.0*colvalues2[0]-128.0+colvalues2[1]);
  }else{
    val2 = 255.0*colvalues2[0]+colvalues2[1];
  }

  if (colvalues3[0] > 127.0/255.0){
    val3 = -(255.0*colvalues3[0]-128.0+colvalues3[1]);
  }else{
    val3 = 255.0*colvalues3[0]+colvalues3[1];
  }

  if (colvalues4[0] > 127.0/255.0){
    val4 = -(255.0*colvalues4[0]-128.0+colvalues4[1]);
  }else{
    val4 = 255.0*colvalues4[0]+colvalues4[1];
  }

  if (colvalues5[0] > 127.0/255.0){

val5 = -(255.0* colvalues5[0]-128.0+colvalues5[1]);
else{
    val5 = 255.0* colvalues5[0]+colvalues5[1];
}
values[0] = val2;
values[1] = val3;
values[2] = val4;
values[3] = val5;

skalning = sqrt(pow(val1,2.0)+pow(val2,2.0)+pow(val3,2.0)+pow(val4,2.0)+pow(val5,2.0));
// unity scaling
if (skalning > 0.01 ){
    val1 = val1/skalning;
    values = values/skalning;
if (val1 == 0.0 || values[0] == 0.0 ){
    quads[0] = sign(values[0])+2.0;
    angle1=0.0;
    if (values[0]==0.0 && val1 < 0.0){
        quads[0] = 0.0;
    }
}
else{
    angle1=2.0*atan((sqrt(pow(val1,2.0)+pow(values[0],2.0))-val1)/values[0])*RAD2DEG;  // Calculates angle
    quads[0] = floor((angle1+180.0)/90.0);  //First quad
    angle1 = angle1-quads[0]*90.0+180.0;  //adjust to second quadrant
    comp = (angle1-offset)/delta;
    if (comp-floor(comp)< 0.5){  //Assign an index to the angle
        ind[0] = floor(comp);
    }
else{
        ind[0] = ceil(comp);
    }
rsq = val1*val1+values[0]*values[0];  //updates the r_squared value
    totalsum = ind[0]*exp2(20.0-R);  //sets the index as the top bits in output
for (int l = 2; l < 5; l++){  //does the same thing for 4 remaining angles
    a = sqrt(rsq);
    b = values[l-1];
    if (b == 0.0 ){
        quads[l-1] = 2.0;
        angles[l-2] = 0.0;
        ind[l-1] = 0.0;
    }
else{
        angles[l-2]=2.0*atan((sqrt(pow(a,2.0)+pow(b,2.0))-a)/b)*RAD2DEG;  //angle
        quads[l-1] = floor((angles[l-2]+180.0)/90.0);  //quad
        angles[l-2]=angles[l-2]-quads[l-1]*90.0+180.0;  //adjusted angle
        comp = abs((angles[l-2]-offset)/delta);
        if (comp-floor(comp)<0.5){
            ind[l-1]=floor(comp);  //index
        }
else{
            ind[l-1]=ceil(comp);
        }
    }
rsq = rsq+b*b;
```
float totalsum += exp2(20.0-float(l)*R)*ind[l-1]; //index placed as next bits
}
rsq = 1.0;
//xVQ = [x2 x3 x4 x5]
//Calculate the angles based on calculated indexes
radians = (ind*delta+offset)/RAD2DEG;
test = rsq;
for (int i = 0; i < 4; i++){
sinval = radians[i]*quads[i]*pi/2.0-pi; //restore angle and quad
xVQ[i] = rsq*rsq*sinval; //calculates restored value
rsq = rsq*cos(sinval); //updates r_squared
}
scaleval = log2(exp2(60.0)*skalning*(xVQ[0]*values[0]+xVQ[1]*values[1]+xVQ[2]*values[2]+xVQ[3]*values[3]+rsq*val1))*5.0-220.0; //Calculates adjusted log2 of MSE scaling
if (scaleval < 0.0)
{
scaleval = 0.0;
}
if (scaleval - floor(scaleval) < ceil(scaleval) - scaleval )
{
scale = floor(scaleval);
}
else {
scale = ceil(scaleval);
}
if (scale > exp2(7.0)-1.0) { //overflow protection
scale = exp2(7.0)-1.0; //127
}
partscale = floor(scale/exp2(4.0));
redsum = 64.0*quads[0]+32.0*(quads[1]-1.0)+16.0*(quads[2]-1.0)+8.0*(quads[3]-1.0)+partscale; //sets red value
greensum = (scale-exp2(4.0)*partscale)*exp2(4.0)+floor(totalsum/exp2(16.0)); //sets green value
bluesum = floor((totalsum-floor(totalsum/exp2(16.0))*exp2(16.0))/exp2(8.0)); //sets blue value
alphasum = totalsum - floor(totalsum/exp2(16.0))*exp2(16.0)-bluesum*exp2(8.0); //sets alpha value
}
gl_FragColor = vec4(redsum/255.0,greensum/255.0,bluesum/255.0,alphasum/255.0); //outputs values
```
```javascript
onmessage = function (e) {
    var buf = e.data[0];
    var bufdata = buf[0];
    var oldsamples = e.data[1];
    var tid = e.data[2];
    var frame = e.data[3];
    var skalfaktor = e.data[4];
    var R = e.data[5];
    var xwq = [];
    var output = [];
    var newold = [];
    var pi = Math.PI;
    var skalning;
    var a1 = 0;
    var sign1 = 0;
    var max = 0;
    var max2 = 0;
    for (var n = 0; n < 960; n++) {
        // restore value from 2-8bit values (1-complement)
        if (bufdata[2 + 4 * n] > 127) {
            a1 = bufdata[2 + 4 * n] - 128;
            sign1 = -1;
        } else {
            a1 = bufdata[2 + 4 * n];
            sign1 = 1;
        }
        test[n] = sign1 * (256 * a1 + bufdata[3 + 4 * n]) / 32767;
        if (test[n] > max) {
            max = test[n];
        }
    }
}
```
```javascript
if (bufdata[4*n] < 128 && 256*bufdata[4*n]+bufdata[4*n+1] > max2) {
    max2 = 256*bufdata[4*n]+bufdata[4*n+1];
}

if (bufdata[4*n] > 127 && 256*(bufdata[4*n]-128)+bufdata[4*n+1] > max2) {
    max2 = 256*(bufdata[4*n]-128)+bufdata[4*n+1];
}

for (var n=0; n < 120; n++) { // unrolled
    xwq[4*n] = test[480+4*n]; // anti time aliasing
    xwq[4*n+1] = test[481+4*n];
    xwq[4*n+2] = test[482+4*n];
    xwq[4*n+3] = test[483+4*n];

    xwq[480+4*n] = -test[969-4*n];
    xwq[481+4*n] = -test[958-4*n];
    xwq[482+4*n] = -test[957-4*n];
    xwq[483+4*n] = -test[956-4*n];

    xwq[960+4*n] = -test[479-4*n];
    xwq[961+4*n] = -test[478-4*n];
    xwq[962+4*n] = -test[477-4*n];
    xwq[963+4*n] = -test[476-4*n];

    xwq[1440+4*n] = -test[4*n];
    xwq[1441+4*n] = -test[4*n+1];
    xwq[1442+4*n] = -test[4*n+2];
    xwq[1443+4*n] = -test[4*n+3];
}

for (var n=0; n < 120; n++) { // 8 samples / loop
    output[8*n] = oldsamples[8*n]+Math.sin((8*n+0.5)*Math.PI/1920)*xwq[8*n]; // windowing and overlay to make new output sound
    output[8*n+1] = oldsamples[8*n+1]+Math.sin((8*n+1.5)*Math.PI/1920)*xwq[8*n+1];
    output[8*n+2] = oldsamples[8*n+2]+Math.sin((8*n+2.5)*Math.PI/1920)*xwq[8*n+2];
    output[8*n+3] = oldsamples[8*n+3]+Math.sin((8*n+3.5)*Math.PI/1920)*xwq[8*n+3];

    output[8*n+5] = oldsamples[8*n+5]+Math.sin((8*n+5.5)*Math.PI/1920)*xwq[8*n+5];
    output[8*n+7] = oldsamples[8*n+7]+Math.sin((8*n+7.5)*Math.PI/1920)*xwq[8*n+7];

    newold[8*n] = Math.sin((8*n+960.5)*Math.PI/1920)*xwq[8*n+960];
    newold[8*n+1] = Math.sin((8*n+961.5)*Math.PI/1920)*xwq[8*n+961];
    newold[8*n+2] = Math.sin((8*n+962.5)*Math.PI/1920)*xwq[8*n+962];
    newold[8*n+3] = Math.sin((8*n+963.5)*Math.PI/1920)*xwq[8*n+963];

    newold[8*n+4] = Math.sin((8*n+964.5)*Math.PI/1920)*xwq[8*n+964];
    newold[8*n+5] = Math.sin((8*n+965.5)*Math.PI/1920)*xwq[8*n+965];
    newold[8*n+6] = Math.sin((8*n+966.5)*Math.PI/1920)*xwq[8*n+966];
    newold[8*n+7] = Math.sin((8*n+967.5)*Math.PI/1920)*xwq[8*n+967];
}

postMessage([frame, tid, output, newold, max, max2, skalfaktor, bufdata]);
```
onmessage = function (e){
  var frame = e.data[0];
  var SampleData = e.data[1];
  var buf = e.data[2];
  var tid = e.data[3];
  var fftmax = e.data[4];
  var xw = [];
  var xtak = [];
  var data = new Uint8Array(1024 * 4);
  var pi = Math.PI;

  // windowing
  for (var n = 0; n < 120; n++) {
    xv[8*n] = Math.sin((8*n + (1/2))*pi/(1920))*SampleData[8*n];
    xv[8*n+1] = Math.sin((8*n + (3/2))*pi/(1920))*SampleData[8*n+1];
    xv[8*n+2] = Math.sin((8*n + (5/2))*pi/(1920))*SampleData[8*n+2];
    xv[8*n+3] = Math.sin((8*n + (7/2))*pi/(1920))*SampleData[8*n+3];
    xv[8*n+4] = Math.sin((8*n + (9/2))*pi/(1920))*SampleData[8*n+4];
    xv[8*n+5] = Math.sin((8*n + (11/2))*pi/(1920))*SampleData[8*n+5];
    xv[8*n+6] = Math.sin((8*n + (13/2))*pi/(1920))*SampleData[8*n+6];
    xv[8*n+7] = Math.sin((8*n + (15/2))*pi/(1920))*SampleData[8*n+7];
    xv[960+8*n] = Math.sin((960+8*n + (1/2))*pi/(1920))*SampleData[960+8*n];
    xv[961+8*n] = Math.sin((960+8*n + (3/2))*pi/(1920))*SampleData[961+8*n];
    xv[962+8*n] = Math.sin((960+8*n + (5/2))*pi/(1920))*SampleData[962+8*n];
    xv[963+8*n] = Math.sin((960+8*n + (7/2))*pi/(1920))*SampleData[963+8*n];
    xv[964+8*n] = Math.sin((960+8*n + (9/2))*pi/(1920))*SampleData[964+8*n];
    xv[965+8*n] = Math.sin((960+8*n + (11/2))*pi/(1920))*SampleData[965+8*n];
    xv[966+8*n] = Math.sin((960+8*n + (13/2))*pi/(1920))*SampleData[966+8*n];
    xv[967+8*n] = Math.sin((960+8*n + (15/2))*pi/(1920))*SampleData[967+8*n];
  }
  SampleData.sort(function(a, b){return b-a});
  var max = SampleData[0]; // Maxvalue in [0]

  for (var i=0; i < 480; i++){
    // Time aliasing
    xtak[i+480] = (xw[i]-xw[959-i]);
    xtak[479-i] = (-xw[i+960]-xw[1919-i]);
  }
}
var skalad1 = xtak[i+480]*100; // shouldn't be over 127, eventually scale for maximizing dynamics

if (skalad1 > 127)
  console.log('DCT texture overflow')
  debugger;

var skalad2 = xtak[479-i]*100;

var sign1 = 0;
var sign2 = 0;

// transform value to 1-complement and divide into 2 8-bit values
if (skalad1 < 0)
  skalad1 *= -1;
  sign1 = 128;
}

if (skalad2 < 0)
  skalad2 *= -1;
  sign2 = 128;
}

var b1 = (skalad1 - (skalad1&skalad1)) * 255;

var b2 = (skalad2 - (skalad2&skalad2)) * 255;

// Insert values into Texture

data[4*i+1920] = skalad1 + sign1; // red, sent to DCT
data[4*i+1921] = b1; // green, sent to DCT

data[4*i+1922] = buf[4*i+1920]; // previous red to blue, sent to iDCT if not VQ is used

data[4*i+1923] = buf[4*i+1921]; // previous green to alpha, sent to iDCT if not VQ is used

data[1916-4*i] = skalad2 + sign2; // red, same as above

data[1917-4*i] = b2; // green

data[1918-4*i] = buf[1916-4*i]; // blue

data[1919-4*i] = buf[1917-4*i]; // alpha

postMessage([frame, tid, data, max, fftmax]);
/*
 * Copyright 2012, Gregg Tavares.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 * *
 * * Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above
 * copyright notice, this list of conditions and the following disclaimer
 * in the documentation and/or other materials provided with the
 * distribution.
 * * Neither the name of Gregg Tavares, nor the names of his
 * contributors may be used to endorse or promote products derived from
 * this software without specific prior written permission.
 * *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 * "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 * LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 * A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 * DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 * THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 * (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 * OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
"use strict";
/** @module webgl-utils */

// These functions are meant solely to help unclutter the tutorials.
// They are not meant as production type functions.

(function() {

/**
 * Wrapped logging function.
 * @param {string} msg The message to log.

*/
var log = function(msg) {
  if (window.console && window.console.log) {
    window.console.log(msg);
  }
};

/**
 * Wrapped logging function.
 * @param {string} msg The message to log.
 */
var error = function(msg) {
  if (window.console) {
    if (window.console.error) {
      window.console.error(msg);
    } else if (window.console.log) {
      window.console.log(msg);
    }
  }
};

/**
 * Turn off all logging.
 */
var loggingOff = function() {
  log = function() {};
  error = function() {};
};

/**
 * Check if the page is embedded.
 * @return {boolean} True of we are in an iframe
 */
var isInIFrame = function() {
  return window != window.top;
};

/**
 * Converts a WebGL enum to a string
 * @param {WebGLRenderingContext} gl The WebGLRenderingContext to use.
 * @param {number} value The enum value.
 * @return {string} The enum as a string.
 */
var glEnumToString = function(gl, value) {
  for (var p in gl) {
    if (gl[p] == value) {
      return p;
    }
  }
  return "0x" + value.toString(16);
};

/**
 * Creates the HTLM for a failure message
 * @param {string} canvasContainerId id of container of the canvas.
 * @return {string} The html.
 */
var makeFailHTML = function(msg) {
  return '' +
    '<table style="background-color: #8CE; width: 100%; height: 100%;">'<tr>' +
    '<td align="center">' +
    '</td>' +
    '</tr>' +
    '</table>';
```javascript
/* Message for getting a webgl browser */
* @type { string }
*/
var GET_A_WEBGL_BROWSER = '';

'"This page requires a browser that supports WebGL.<br/>
<a href="http://get.webgl.org">Click here to upgrade your browser.</a>";

/**
 * Message for need better hardware
 * @type { string }
*/
var OTHER_PROBLEM = '';

'"It doesn't appear your computer can support WebGL.<br/>
<a href="http://get.webgl.org/troubleshooting/">Click here for more information.</a>"

/**
 * Creates a webgl context. If creation fails it will
 * change the contents of the container of the <canvas>
 * tag to an error message with the correct links for WebGL.
 * @param { HTMLCanvasElement } canvas The canvas element to
 * create a context from.
 * @param { WebGLContextCreationAttributes } opt_attribs Any
 * creation attributes you want to pass in.
 * @return { WebGLRenderingContext } The created context.
*/
var setupWebGL = function (canvas, opt_attribs) {

    function showLink(str) {
        var container = canvas.parentNode;
        if (container) {
            container.innerHTML = makeFailHTML(str);
        }
    }

    if (!window.WebGLRenderingContext) {
        showLink(GET_A_WEBGL_BROWSER);
        return null;
    }

    var context = create3DContext(canvas, opt_attribs);
    if (!context) {
        showLink(OTHER_PROBLEM);
    }
    return context;

    /**
     * Creates a webgl context.
     * @param { HTMLCanvasElement } canvas The canvas tag to get
     * context from. If one is not passed in one will be
     * created.
     * @return { WebGLRenderingContext } The created context.
     */
    var create3DContext = function (canvas, opt_attribs) {
        var names = ["webgl", "experimental-webgl"]; 
        var context = null;
    }
```
for (var ii = 0; ii < names.length; ++ii) {
  try {
    context = canvas.getContext(names[ii], opt_attribs);
  } catch (e) {} 
  if (context) {
    break;
  }
}
return context;

var updateCSSIfInIFrame = function () {
  if (isInIFrame()) {
    document.body.className = "iframe";
  }
};

/**
 * Gets a WebGL context.
 * makes its backing store the size it is displayed.
 */
var getWebGLContext = function (canvas, opt_attribs, opt_options) {
  var options = opt_options || {};
  if (isInIFrame()) {
    updateCSSIfInIFrame();
    // make the canvas backing store the size it's displayed.
    if (!options.dontResize) {
      var width = canvas.clientWidth;
      var height = canvas.clientHeight;
      canvas.width = width;
      canvas.height = height;
    }
  } else {
    var title = document.title;
    var h1 = document.createElement("h1");
    h1.innerText = title;
    document.body.insertBefore(h1, document.body.children[0]);
  }
  var gl = setupWebGL(canvas, opt_attribs);
  return gl;
};

/**
 * Loads a shader.
 * @param {WebGLRenderingContext} gl The WebGLRenderingContext to use.
 * @param {string} shaderSource The shader source.
 * @param {number} shaderType The type of shader.
 * @param {function(string): void} opt_errorCallback callback for errors.
 * @return {WebGLShader} The created shader.
 */
var loadShader = function (gl, shaderSource, shaderType, opt_errorCallback) {
  var errFn = opt_errorCallback || error;
  // Create the shader object
  var shader = gl.createShader(shaderType);
  // Load the shader source
  gl.shaderSource(shader, shaderSource);
  // Compile the shader
  gl.compileShader(shader);
// Check the compile status
var compiled = gl.getShaderParameter(shader, gl.COMPILE_STATUS);
if (!compiled) {
  // Something went wrong during compilation; get the error
  var lastError = gl.getShaderInfoLog(shader);
  errFn("*** Error compiling shader '" + shader + ":" + lastError);
  gl.deleteShader(shader);
  return null;
}

return shader;

/**
 * Creates a program, attaches shaders, binds attrib locations, links the
 * program and calls useProgram.
 * @param {WebGLShader[]} shaders The shaders to attach
 * @param {string[]} opt_attribs The attribs names.
 * @param {number[]} opt_locations The locations for the
 * attribs.
 * @param {function(string): void} opt_errorCallback callback for errors.
 */
var loadProgram = function (gl, shaders, opt_attribs, opt_locations, opt_errorCallback) {
  var errFn = opt_errorCallback || error;
  var program = gl.createProgram();
  for (var ii = 0; ii < shaders.length; ++ii) {
    gl.attachShader(program, shaders[ii]);
  }
  if (opt_attribs) {
    for (var ii = 0; ii < opt_attribs.length; ++ii) {
      gl.bindAttribLocation(
        program,
        opt_locations ? opt_locations[ii] : ii,
        opt_attribs[ii]);
    }
  }
  gl.linkProgram(program);

  // Check the link status
  var linked = gl.getProgramParameter(program, gl.LINK_STATUS);
  if (!linked) {
    // something went wrong with the link
    var lastError = gl.getProgramInfoLog(program);
    errFn("Error in program linking:" + lastError);
    gl.deleteProgram(program);
    return null;
  }

  return program;
};

/**
 * Loads a shader from a script tag.
 * @param {WebGLRenderingContext} gl The WebGLRenderingContext to use.
 * @param {string} scriptId The id of the script tag.
 * @param {number} opt_shaderType The type of shader. If not passed in it will
 * be derived from the type of the script tag.
 * @param {function(string): void} opt_errorCallback callback for errors.
 * @return {WebGLShader} The created shader.
 */
var createShaderFromScript = function (gl, scriptId, opt_shaderType, opt_errorCallback) {
gl, scriptId, opt_shaderType, opt_errorCallback) {
    var shaderSource = "";
    var shaderType;
    var shaderScript = document.getElementById(scriptId);
    if (!shaderScript) {
        throw("*** Error: unknown script element" + scriptId);
    }
    shaderSource = shaderScript.text;
    if (!opt_shaderType) {
        if (shaderScript.type == "x-shader/x-vertex") {
            shaderType = gl.VERTEX_SHADER;
        } else if (shaderScript.type == "x-shader/x-fragment") {
            shaderType = gl.FRAGMENT_SHADER;
        } else if (shaderType != gl.VERTEX_SHADER && shaderType != gl.FRAGMENT_SHADER) {
            throw("*** Error: unknown shader type");
            return null;
        }
    } else if (opt_shaderType) {
        if (opt_shaderType == "x-shader/x-vertex") {
            shaderSource = shaderScript.text;
        } else if (opt_shaderType == "x-shader/x-fragment") {
            shaderSource = shaderScript.text;
        } else if (opt_shaderType != gl.VERTEX_SHADER && opt_shaderType != gl.FRAGMENT_SHADER) {
            throw("*** Error: unknown shader type");
            return null;
        }
    }
    return loadShader(
        gl, shaderSource, opt_shaderType ? opt_shaderType : shaderType,
        opt_errorCallback);
};

var defaultShaderType = [
    "VERTEX_SHADER",
    "FRAGMENT_SHADER"
];

/**
 * Creates a program from 2 script tags.
 * @param { WebGLRenderingContext } gl The WebGLRenderingContext
to use.
 * @param { string[] } shaderScriptIds Array of ids of the script
tags for the shaders. The first is assumed to be the vertex shader,
the second the fragment shader.
 * @param { string[]? } opt_attribs The attribs names.
 * @param { number[]? } opt_locations The locations for the attribs.
 * @param { function ( string ): void ) opt_errorCallback callback for errors.
 * @return { WebGLProgram } The created program.
 */
var createProgramFromScripts = function(
    gl, shaderScriptIds, opt_attribs, opt_locations, opt_errorCallback) {
    var shaders = [];
    for (var ii = 0; ii < shaderScriptIds.length; ++ii) {
        shaders.push(createShaderFromScript(
            gl, shaderScriptIds[ii], gl[defaultShaderType[ii]], opt_errorCallback));
    }
    return loadProgram(gl, shaders, opt_attribs, opt_locations, opt_errorCallback);
};

/**
 * Creates a program from 2 sources.
 * @param { WebGLRenderingContext } gl The WebGLRenderingContext
to use.
 * @param { string[] } shaderSources Array of sources for the shaders. The first is assumed to be the vertex shader,
the second the fragment shader.
 * @param { string[]? } opt_attribs The attribs names.
 */
var createProgramFromSources = function(
    gl, shaderSources, opt_attribs, opt_locations, opt_errorCallback) {
* @param {number[]} opt_locations The locations for the
  attribs.
* @param {function(string): void} opt_errorCallback callback for errors.
* @return {WebGLProgram} The created program.
*/

```
var createProgramFromSources = function (gl, shaderSources, opt_attribs, opt_locations, opt_errorCallback) {
  var shaders = [];
  for (var ii = 0; ii < shaderSources.length; ++ii) {
    shaders.push(loadShader(gl, shaderSources[ii], gl[defaultShaderType[ii]], opt_errorCallback));
  }
  return loadProgram(gl, shaders, opt_attribs, opt_locations, opt_errorCallback);
};
```

```
var getBindPointForSamplerType = function (gl, type) {
  if (type == gl.SAMPLER_2D) return gl.TEXTURE_2D;
  if (type == gl.SAMPLER_CUBE) return gl.TEXTURE_CUBE_MAP;
};
```

```
var createUniformSetters = function (gl, program) {
  var textureUnit = 0;
  ```
  `*/
  ```
  ```/*
  /**
  * Creates a setter for a uniform of the given program with it’s
  * location embedded in the setter.
  */
  */
  ```
```
  var createUniformSetter = function (program, uniformInfo) {
    var location = gl.getActiveUniformLocation(program, uniformInfo.name);
    var type = uniformInfo.type;
    // Check if this uniform is an array
    var isArray = (uniformInfo.size > 1 & uniformInfo.name.substr(-3) == "[0]");
    if (type == gl.FLOAT & isArray)
      return function (v) { gl.uniform1fv(location, v); };
    if (type == gl.FLOAT)
      return function (v) { gl.uniform1f(location, v); };
    if (type == gl.FLOAT_VEC2)
      return function (v) { gl.uniform2fv(location, v); };
    if (type == gl.FLOAT_VEC3)
      return function (v) { gl.uniform3fv(location, v); };
    if (type == gl.FLOAT_VEC4)
      return function (v) { gl.uniform4fv(location, v); };
  };
```
if (type == gl.INT && isArray)
    return function(v) { gl.uniform1iv(location, v); };
if (type == gl.INT)
    return function(v) { gl.uniform1i(location, v); };
if (type == gl.INT_VEC2)
    return function(v) { gl.uniform2iv(location, v); };
if (type == gl.INT_VEC3)
    return function(v) { gl.uniform3iv(location, v); };
if (type == gl.INT_VEC4)
    return function(v) { gl.uniform4iv(location, v); };
if (type == gl.BOOL)
    return function(v) { gl.uniform1iv(location, v); };
if (type == gl.BOOL_VEC2)
    return function(v) { gl.uniform2iv(location, v); };
if (type == gl.BOOL_VEC3)
    return function(v) { gl.uniform3iv(location, v); };
if (type == gl.BOOL_VEC4)
    return function(v) { gl.uniform4iv(location, v); };
if (type == gl.FLOAT_MAT2)
    return function(v) { gl.uniformMatrix2fv(location, false, v); };
if (type == gl.FLOAT_MAT3)
    return function(v) { gl.uniformMatrix3fv(location, false, v); };
if (type == gl.FLOAT_MAT4)
    return function(v) { gl.uniformMatrix4fv(location, false, v); };
if ((type == gl.SAMPLER_2D || type == gl.SAMPLER_CUBE) && isArray)
    { var units = [];
        for (var ii = 0; ii < info.size; ++ii) {
            units.push(textureUnit++);
        }
        return function(bindPoint, units) {
            return function(textures) {
                gl.uniform1iv(location, units);
                textures.forEach(function(texture, index) {
                    gl.activeTexture(gl.TEXTURE0 + units[index]);
                    gl.bindTexture(bindPoint, texture);
                });
            })(getBindPointForSamplerType(gl, type), units);
        }
    if (type == gl.SAMPLER_2D || type == gl.SAMPLER_CUBE)
    return function(bindPoint, unit) {
        return function(texture) {
            gl.uniform1i(location, unit);
            gl.activeTexture(gl.TEXTURE0 + unit);
            gl.bindTexture(bindPoint, texture);
        };
    })(getBindPointForSamplerType(gl, type), textureUnit++);
    throw ("unknown type: 0x" + type.toString(16)); // we should never get here.
}

var uniformSetters = {};
var numUniforms = gl.getActiveUniform(program, ii);
if (!uniformInfo) {
    break;
}
var name = uniformInfo.name;
// remove the array suffix.
if (name.substr(-3) == "[0]"avanaugh{
    name = name.substr(0, name.length - 3);
}
var setter = createUniformSetter(program, uniformInfo);
uniformSetters[name] = setter;
}
return uniformSetters;

/**
 * Set uniforms and binds related textures.
 *
 * @example
 * 
 * var program = createProgramFromScripts(
 *     gl, ['some-vs', 'some-fs']);
 * 
 * var uniformSetters = createUniformSetters(program);
 * 
 * var tex1 = gl.createTexture();
 * var tex2 = gl.createTexture();
 * 
 * ... assume we setup the textures with data ... 
 *
 * var uniforms = {
 *     u_someSampler: tex1,
 *     u_someOtherSampler: tex2,
 *     u_someColor: [1,0,0,1],
 *     u_somePosition: [0,1,1],
 *     u_someMatrix: [
 *         1,0,0,0,
 *         0,1,0,0,
 *         0,0,1,0,
 *         0,0,0,0,
 *     ],
 * };
 * 
 * gl.useProgram(program);
 * 
 * This will automatically bind the textures AND set the
 * uniforms.
 * 
 * setUniforms(uniformSetters, uniforms);
 * 
 * @param {Setters} setters the setters returned from
 * createUniformSettersForProgram
 * @param {Object.<string, value>} an object with values for the
 * uniforms.
 * */
var setUniforms = function(setters, values) {
    Object.keys(values).forEach(function(name) {
        var setter = setters[name];
        if (setter) {
            setter(values[name]);
        }
    });
};

/**
 * Creates setter functions for all attributes of a shader program
 * 
 * @see setAttributes for example
 * 
 * @param {WebGLProgram} program the program to create setters for.
 * */
var createAttributeSetters = function(gl, program) {
    var attribSetters = {
        function createAttribSetter(index) {
            return function(b) {
                gl.bindBuffer(gl.ARRAY_BUFFER, b.buffer);
                gl.enableVertexAttribArray(index);
                gl.vertexAttribPointer(index, b.numComponents || b.size, b.type || gl.FLOAT, b.normalize || false, b.stride || 0, b.offset || 0);
            }
        }
    }

    var numAttribs = gl.getProgramParameter(program, gl.ACTIVE_ATTRIBUTES);
    for (var ii = 0; ii < numAttribs; ++ii) {
        var attribInfo = gl.getActiveAttrib(program, ii);
        if (!attribInfo) {
            break;
        }
        var index = gl.getAttribLocation(program, attribInfo.name);
        attribSetters[attribInfo.name] = createAttribSetter(index);
    }

    return attribSetters;
};

/**
* @returns {Setters} an object with a setter for each uniform
* by name.
* @example
* var program = createProgramFromScripts(
*     gl, ["some-vs", "some-fs"]);
* var attribSetters = createAttributeSetters(program);
* var positionBuffer = gl.createBuffer();
* var texcoordBuffer = gl.createBuffer();
* var attribs = {
*    a_position: {buffer: positionBuffer, numComponents: 3},
*    a_texcoord: {buffer: texcoordBuffer, numComponents: 2},
*};
* gl.useProgram(program);
* This will automatically bind the buffers AND set the
* attributes.
* setAttributes(attribSetters, attribs);
* Properties of attribs. For each attrib you can add
* properties:
* type: the type of data in the buffer. Default = gl.FLOAT
* normalize: whether or not to normalize the data. Default = false
* stride: the stride. Default = 0
* offset: offset into the buffer. Default = 0
For example if you had 3 value float positions, 2 value float texcoord and 4 value uint8 colors you’d setup your attribs like this

```javascript
var attribs = {
  a_position: { buffer: positionBuffer, numComponents: 3 },
  a_texcoord: { buffer: texcoordBuffer, numComponents: 2 },
  a_color: {
    buffer: colorBuffer,
    numComponents: 4,
    type: gl.UNSIGNED_BYTE,
    normalize: true,
  }
};
```

*/
```javascript
var setAttributes = function (setters, buffers) {
  Object.keys(buffers).forEach(function (name) {
    var setter = setters[name];
    if (setter) {
      setter(buffers[name]);
    }
  });
};
```
canvas.width = width;
canvas.height = height;
return true;
}
return false;
}

/* export functions */
window.createAttributeSetters = createAttributeSetters;
window.createProgram = loadProgram;
window.createProgramFromScripts = createProgramFromScripts;
window.createProgramFromSources = createProgramFromSources;
window.createShaderFromScriptElement = createShaderFromScript;
window.createUniformSetters = createUniformSetters;
window.getWebGLContext = getWebGLContext;
window.updateCSSIfInIFrame = updateCSSIfInIFrame;
window.getExtensionWithKnownPrefixes = getExtensionWithKnownPrefixes;
window.resizeCanvasToDisplaySize = resizeCanvasToDisplaySize;
window.setAttributes = setAttributes;
window.setUniforms = setUniforms;
window.setupWebGL = setupWebGL;

// All browsers that support WebGL support requestAnimationFrame
window.requestAnimationFrame = window.requestAnimationFrame;  // just to stay backward compatible.
window.cancelRequestAnimationFrame = window.cancelAnimationFrame;  // just to stay backward compatible.
}

}}();