FRODIGI (FREE RUNNING OSCILLATOR DIGITAL AUDIO METHODS) FFT & AWD ALGORITHM / ALGOTECH 2014

INTRODUCTION

Recreating audio from a source and playing it back on a c64 requires any of the below to act as a solution.

Recreate the audio via manually re-composing the song

This method by far results in the most compact of data, however one major drawback is that it can take a considerable amount of work and time and mainly results in the output being a melody based representation of the audio

Imagine an audio piece with drums, chords and bass line. The idea is to create the instrumentation via specifying the waveforms or/and filter/pulse, frequency and then to recompose the bass line, lead and chords by ear. The result in many cases is a rough approximation of the original source on a melody level only. Many of the top musicians can get quite a decent approximation even when considering the lack of high channel availability on the c64.

Sample the original audio

By far the most accurate method but requires a huge amount of ram and cpu usage is very high. The audio source waveform is resampled usually to the required amount that will be updated at the same speed on the c64 end. (E.g. resampling and saving as 8bit 8000 Hz would require the c64 decoder to update once per two raster lines. This uses many kilobytes of ram per second. 3916 bytes per second for 4bit samples and 7812 bytes per second for 8bit.

There are many methods of compressing the audio data and decoding at the desired sample rate in order to satisfy the issue with less ram availability on the c64, but this would end up eating even more CPU time due to the additional calculations required for decoding.

What other solutions are there? . Would it not be ideal if we were to analyse this audio data and to place specific values in the SID chip registers only at very few intervals (e.g. once per frame) and let the SID do the rest?

History

Since June 2014, I have been interested in recreating digitized audio via using the C64's SID oscillators. Since this current date (07/12/2014) I have released 4 demonstrations utilising this technique as follows.

Frodigi 1 – Single file c64 demo – single speed
9bit lookup table based with around 64 frequencies per waveform (fixed notes) and master volume using custom wave deconstruction method

Frodigi 2 – Single file c64 demo – single speed
12bit frequency and master volume with more advanced analysis using custom wave deconstruction method

Frodigi 3 – single file c64 demo – single speed
12bit frequency and 3 individual sustains with more advanced analysis using custom wave deconstruction

Frodigi 4 – single file c64 demo – single speed
12bit frequency and 3 individual sustains using the AWD wave deconstruction technique

Frodigi 5 – single file c64 demo – quad speed
To be released. Using the New quad speed FFT method.

Since June 2014, I have developed the console based encoder that was used in Frodigi 1 and 2. Since Frodigi 3 to 5, I have created the encoder from scratch using GUI and recently implementing the FFT module with its additional capabilities that are mentioned in this document

FRODIGI SUPER - FFT METHOD

Introduction and issues

The Fast Fourier Transform (FFT) transforms time domain data (e.g. PCM audio) into the frequency domain. We can then pick out the frequencies that have more dominance and then map these over to the c64 SID chip. There are a few issues however and we will need these fixed.

Firstly, taking the FFT of an audio piece will give us the frequency information for the audio, but it will not tell us where in time that frequency was present. To solve this issue, I am using a variant and my own implementation of what is known as the STFT (Short Time Fourier transform)

This allows frequency data to be localised in time to a certain extent.

The other main issue with the Fourier transform is that having high frequency resolution results in poor time localisation and vice versa. Excellent time localisation results in poor frequency resolution.

Let's assume we have a piece of data that is 500kilobytes in length and was sampled at 44100 Hz.

A frame worth of sample data would be 44100/50 = 882 bytes on average.

The FFT operates normally in powers of 2. Hence 882 would not be a valid amount. What we can however do is to read 1024 samples, and then in the next update, offset the required amount (882)

We can read in the first 1024 samples and perform a FFT with this.

This will result in real and imaginary components. In this context of encoding, the imaginary components are not required, hence will not be mentioned. This is because we are not interested in reconstructing back a PCM waveform. We only want to find out the amplitudes and frequency to map over to the c64. What we are interested is in the 'real' section of the FFT

The first byte of the real array is the DC component and can be safely ignored.

The second byte of the real array (offset 1) until the (fftsize/2)+1 is what is known as the frequency "bins." In this example, taking an FFT of chunk 1024 results in the useful data being in the range 1-512. The other data from 513-1023 are not required for this purpose.

Each of the slots in the frequency bins are mapped to a frequency. How do we find out the frequency offsets between each slot? The formula is sample rate/fftsize

In this case 44100/1024 will result in the frequency of 43.06. Hence the first bin would correspond to the frequency 43.06. The next one would correspond to 43.06x2) =86.12 etc

The max frequency available would be sample rate/2. In this case bin 512 would correspond to 22050 Hz.

The C64's SID oscillators can only go up to around 3.9 kHz. Hence the frequency bins that we are interested in are 1-88

More issues and solutions

Can you see the issue that arises from the above? Yes. The frequency resolution is too low. There are only 88 usable frequencies and if the source audio consists of low frequency data, this cannot be represented well with gaps of 43 Hz between each bin. Having high frequency resolution is important more in the low frequencies.

How can we increase the frequency resolution but keep time localisation intact? The answer is that we cannot do this. For example using an FFT size of 8192 would equate to reading in data for a 5^{th} of a second. During this time, there may be a considerable amount of changes of frequency based on time. The plus point however is that the frequency resolution would be increased to 5.4 Hz per bin.

In order to keep time localisation, the solution is to use what is known as window overlap and padding.

For example if wishing to use a FFT size of 8192. Read in the first frame of audio data (882 bytes) followed by around 50% or 100% addition to that data. (More on this later) e.g. in this case 50% addition would be 882+441 samples=1323. 100% would be 1764 samples.

Then pad the rest with zero's and perform the FFT.

What does this do? Well, it does not actually increase the frequency resolution, it does however interpolate the data in the frequency domain and this works well in what we need to extract. It also ensures that there is not too many of any future frames frequencies which would end up in too much spectral smoothing.

Once we have processed this data, the next frame can be offset by 882 bytes and the next overlapping samples can be read.

Unfortunately there is another issue known as spectral leaking. Sending over a chunk of samples into the FFT will result in some unwanted frequencies. This is inherent due to the sudden start and end edges in the time domain which result in the unwanted frequencies when the FFT process is applied.

To solve this issue, before the FFT is performed, a windowing over the viewport is performed which gradually rises the sample content from zero amplitude to current, then falls down to zero. This is applied to the read chunk only, not the full fft length. E.g. if using 8192 chunk fft but only 1764 samples are at the beginning, then apply the window to the 1764 samples.

There are dozens of common types of windows, most common being the hamming and hann window. Each window may work better/worse dependant on the content of the audio source and amount of overlap.

It is also important to realise that performing windowing gives us a narrower amount of data at the normal amplitude, hence why reading in overlapping data is important. E.g. reading in 882 bytes of audio would effectively result in around 200 sample data being focused on after windowing.

For decoding purposes the overlap needs to also be done so that the overlap between each block results in data being faded out from the previous chunk and faded into the next. However as we are not concentrating on audio reconstruction into PCM, the overlapping can be more flexible.

Ok, so now we have solved the issue of spectral leakage as well as low frequency resolution. What next?

Analysing magnitude

We need to analyse the maximum magnitude of the whole sample data.

This is done simply by squaring the real component of the fft and the imaginary, adding these together and getting the square root for each bin. The highest value is retained, the next block is read, and the highest value is replaced if there are any higher values in the audio stream

The maximum magnitude is then scaled as 14. (More on this later)

Ok, the next part will then be recreating the data so that it can be played back on a c64. We have to consider the following limitations.

C64 maximum oscillator frequency is around 3.9 kHz.
There are three channels. Each having 16bit resolution
Each channel can have its own sustain 0-15, but there is less flexibility

Analysis process

The first block is read, and after overlapping, windowing, the FFT is performed.

The Bin values above 3.9 kHz are reset to zero. In the case of a 8192 point fft, this would give around 724 frequency locations in the range 0-3900Hz.

Scan the frequency locations and find the highest magnitude. This is done again via squaring the real and imaginary component and getting the square root (which also ensures that the value is a positive number)

If for example, the highest magnitude was at offset 7. The frequency value for this would be (44100/8192)*7 = 37.68 Hz

Convert this frequency value to the C64's 16bit frequency.

The formula for PAL machines is

Palvalue=312(screenlines)*63(cycles per line)*50.12454212(framerate) = 985248
multvalue=16777216/palvalue = 17.02841924

Then we only have to multiply the multvalue by the frequency which will map over the frequency to the relevant c64's 16bit frequency

Then we need to scale the magnitude to the values 0-14. (0 being silence and 14 maxvolume)

Formula for this would be the below

Sustain=(currentmagnitude*14)/maxmagnitude

Where maxmagnitude was the highest magnitude value found in the entire audio and currentmagnitude is the highest found in the current analysed block

Great. Now we have converted the most dominant frequency in a frames worth of sample into a 16bit c64 frequency as well as a sustain value which can be mapped over directly to the c64. The waveform to use on the C64's end is the triangle waveform (as that is the one of the waveforms that is closest to sine (and even more so when filtered) Pulse can also be used (filtered) to give the sine based curve but can suffer from more distortion. But..

Sustain issues

Yes, the c64 does not give full control over the sustain levels. It's possible to reduce the sustain, but not to increase it at any point at immediate request. In order to increase it, it requires that the decoder on the c64 side sets gateoff/on which causes some distortion (click)

In order to minimise this from occurring, the current sustain value is compared with the previous. If the new sustain value is equal or lower, then it is let through without modification.

If the sustain value however is higher than the previous but lower than the threshold value (user definable), a quick distance calculation is used to determine whether or not the sustain should be changed to the previous sustain value (no change) or to the sustain value of (previoussustainvalue+threshold)

If the sustain value is higher than the threshold, then it is changed to the higher value.

The sustain value is changed accordingly and this will be used in the next iteration as the previous sustain value.

This ensures that the sustain is held to the same value if possible unless it is over a certain threshold, and in the case of it requesting to be held, it is still compared and either set to the same level as previously or to the value of the previoussustain+threshold.

The c64 has 15 sustain values, why only use 14? (More on this later)

The process is then repeated for the next two channels, finding the second and third most dominant magnitudes.

Issues with frequency selection

But again there are issues...

Interpolating frequencies results in dominant frequencies being very close together. This unfortunately has an effect of the encoder picking similar frequency values and sustain levels for each of the channels which reduces output definition considerably.

How do we overcome this? Two methods are as below

Frequency zoning

Remember that we are only interested in the frequencies 0-3900Hz. We separate this into three zones with the zones being more concentrated in the lower frequencies and less concentrated in the higher.

Ideal values would be 100-300 Hz for the first zone, followed by 310-800 Hz for zone2 and then 810-3900 for zone 3

We then find the most dominant frequency in each of the zones (instead of finding the second one in the same zone)

The above also results in data that can be compressed very well (but quality can suffer) with this method (as well as the other methods) lookup tables can be used that point to the exact frequency.

For example there are 724 useful frequency amounts when using a 8192 point FFT. Dividing this into 3 equal zones would give around 241 values per zone. The actual frequency can be replaced by the 8bit offset into the lookup table which would save an extra byte of data per channel.

As mentioned before, quality will suffer, as there will no doubt be area's in the audio that do not have any dominant frequency in one/two of the zones but have them clustered in the same zone.

Naive psychoacoustic method

This method works very well but is a very naive approach. First of all the most dominant frequency in the range is found, then this (along with its nearby neighbours are cleared) This prevents the second most dominant frequency being similar/identical to the first. This (with the relevant distance value) dynamically adjusts based on the content of the audio and works very well

Writing the output

Ok, now we have all the data that is ready to be written on the c64. How do we make sure that the data is compact enough?

Various approaches. Some of these that are in the encoder are as follows..

Convert 16bit frequency to 12bit value (and $fff0) and place sustain 0-14 in bits 0-3) this results in a channel using two bytes (and 3 channels using 6 bytes)

6x50 updates per seconds = 300 bytes per second data rate and can be compressed further.

Use lookups that point to tables (Using smaller FFT sizes, 4096,8192 etc) can be beneficial for more compact packing where the exact sid frequency values are saved as a table of data and the bit stream (9-10bits per channel) point to the lookup

The player

It was mentioned that there are only 0-14 values of sustain used for each channel, why is this?

As mentioned earlier on, the sustain of a waveform can be lowered or kept the same, but not increased unless the gate bit is reset and set again. This causes audible distortion.

When the player reads a sustain value that's the same or lower, it only needs to feed the frequency to the sid channel and nothing else.

If it is higher, then the new sustain value is placed, but does not make any difference. Clearing the gate bit and setting it again, causes the attack to reach maximum amplitude and then to fall to the desired sustain.

In order to reduce this distortion, the encoder ensures that the sustain for each channel is increased only if absolutely required, however in the case where the sustain needs to be higher than the previous, the player will use one of two methods of increasing the sustain

If new sustain value is higher than previous but lower than a fixed value (such as midrange $8) then setting sustain to $f and then the required sustain and then setting gate off/on drops the sustain to zero and then the required sustain. This is more beneficial than increasing to max sustain then dropping to a lower sustain – lower than 8.

If the new sustain value is higher than previous but higher than the fixed value (such as midrange $8) then setting the desired sustain and then setting gate off/on starts the attack to maximum sustain, then drops to the desired sustain. This is more beneficial than dropping to zero sustain and then increasing to required sustain higher than 8

Quality improvements

Dual / Quad speed definition and update

In order for more clarity, the encoding process can easily be adjusted to work at dual or quad speed update. The only main adjustment that needs to be done would be to offset a quarter frames worth of samples per analysis (instead of 882) – in combination with adjusting viewport sizes if required

For example for quad speed update, adjust analysis block around 220 bytes instead of 882 and then update the c64 decoder evenly 4 times per frame.

Bear in mind that data rate is now quadrupled (for quad speed) but quality is dramatically a lot higher

Dual / Triple SID

Again, there is less modification required, instead of picking the 3 frequencies and sustain, pick 6 or 9 (dual SID / triple SID) In the case of the zoning method, separate into 6 or 9 zones instead of 3.

BIN Averaging and further interpolation

Can perform multiple overlaps and merging of the FFT amplitudes into an averaged approximation. Even though having a higher FFT size or input chunk data applies the frequency addition to the bins after FFT, the overlap shifting approach may result in more stability.

Can also low pass interpolate neighbouring magnitude values which can work well in some cases

Filtering

Setting a mid/low pass filter in the decoder can smooth and hide some of the artifacts and recommended to use.

Master volume / fixed sustain

By default on average, the third dominant frequency is considerably less than the second, which in turn the second is considerably less than the first. In this case, fixed sustains of values such as $f, $a, $6 can be used and then the master volume used to adjust all three (this uses a byte less data (only need one 4bit sustain value)

Shortcomings

The FFT approach mentioned above has removed the limitations of low frequency resolution and time locality and works very well and better than the AWD method (mentioned in next chapter) for audio that is more simple in nature

The AWD method can recreate timbre more accurately and work better for audio that would otherwise choke utilising the FFT method, BUT vice versa, the FFT method works better for simpler data and the AWD method would choke more on this simple data!

Visualise it in graph form. For data that is simple to average, the FFT method beats the quality of AWD. But when the data becomes more complex, the FFT stops in its tracks and the AWD method wins in quality. However in the case of extremely complex audio data, even though the AWD method would be more accurate, it may sound very harsh

Next chapter, the AWD method